AN INTELLIGENT AGENT BASED AUTONOMOUS MISSION MANAGEMENT SYSTEM FOR UAVS

PAOLO GUNETTI

Thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Control Systems

Department of Automatic Control and Systems Engineering

University of Sheffield

The University Of Sheffield.

Submitted JUNE 2011

Abstract

Unmanned Aerial Vehicles (UAVs) have been around for almost eight decades, but they evolved from the basic remotely-controlled form only during the last two. While UAV-related research encompasses several areas, the increase of autonomy is certainly one of the most important topics. Current-generation UAVs are typically able to autonomously execute a pre-planned mission, and there is a definite trend towards increased UAV autonomy. The main challenge related to UAV autonomy is the reaction to unforeseen events; the capability to execute pre-planned tasks does not take into account the fact that in a dynamic environment the plan might need to be changed. This Mission Management activity is usually tasked to human supervision.

Within this thesis, the problem of UAV autonomy will be addressed by proposing a software architecture that allows the integration of various technologies to obtain a significant improvement in the autonomy level of UAV. The first step in this is the definition of a set of theoretical concepts that allow the computational description of three different types of information: user-generated mission objectives, knowledge regarding the external environment and complete flight plans. The second step is then to develop a software system that autonomously accomplishes the Mission Management task: combining mission objectives and environmental knowledge to generate a viable flight plan, then update it when situational awareness changes.

The software system presented in this thesis is implemented using a combination of three separate Soar Intelligent Agents and traditional control techniques. Soar (*State, Operator and Result*) is the computational implementation of a general theory of cognition, allowing for the definition of complex software agents which can efficiently apply the rules defining their behaviour to large amounts of knowledge. Soar agents are used to provide high-level reasoning capability to the system, while low-level control functions are better performed by traditional algorithms. The software system is fully described and implemented, then tested in a simulation environment. Simulations are used to demonstrate the system's ability to automatically generate, execute and update (when necessary) an entire flight plan after being assigned a set of mission objectives.

Acknowledgements

During the past years, I realized that the biggest challenge in doing a PhD was dealing with myself: the elusive nature of research requires a lot of focus to carry on working, especially when results seem far away. With no one there to push you, the strength to finish what was started must be found within oneself.

Nevertheless, this would not be possible without expert guidance. I am very grateful to Dr. Tony Dodd for his supervision; his critical eye made me see my mistakes, and his encouragement made me see the value of my work. A work which would have never begun without the help of Prof. Haydn Thompson, who not only encouraged me to go beyond the scope of the original ASTRAEA project but also continued to spark ideas for improvement.

Encouragement does not only come professionally; during my stay in Sheffield, I met many people with whom I shared the PhD experience or just life in general. So, all my thanks go to Simone, Stefano, Ben, Miguel, Helen, Rachael, Caterina and all the others who made this experience a fun one.

Most importantly, this entire work is dedicated to my beloved mother and to the memory of my father, with the hope that this accomplishment will make them proud.

Contents:

List of acronyms

AI	Artificial Intelligence
ATC	Air Traffic Control
BDI	Belief-Desire-Intention
COTS	Common-off-the-shelf
EHM	Engine Health Management
EXAG	Execution Agent
FADEC	Full Authority Digital Engine Controller
FMECA	Failure Modes Effects and Criticality Analysis
IA	Intelligent Agent
I/O	Input/Output
OODA	Observe, Orient, Decide, Act
MAS	Multi-Agent System
MEMA	Multi-Engine Manager Agent
MMA	Mission Manager Agent
PHM	Propulsion Health Management
SAMMS	Soar-based Autonomous Mission Management System
SEPA	Single-Engine Planner Agent
SLE	System-Level Effect
SOAR	State, Operator and Result
TRL	Technology Readiness Level
UAS	Uninhabited (Unmanned) Aerial System
UAV	Uninhabited (Unmanned) Aerial Vehicle
UGV	Uninhabited (Unmanned) Ground Vehicle
UMV	Uninhabited (Unmanned) Marine Vehicle
WME	Working Memory Element

1. Introduction

The introduction of Uninhabited Aerial Vehicles (or UAVs) dates to even before the Second World War (Goebel, 2008), radio technology being the enabler technology. The first examples of UAVs were in fact simple radio-controlled target drones, and were built during the 1930s. During and after the War, advances in control technologies allowed for the introduction of a different type of UAV: the missile. Finally, in the 1960s the first reconnaissance UAVs were developed by the United States and the Soviet Union.

These military applications focused on only one of the main characteristics of UAVs compared to piloted aircraft: expendability. While this is obviously paramount for target drones and missiles, it was immediately evident that UAVs could bring other substantial advantages in more generic applications such as reconnaissance. However, technological limitations did not allow for the development of advanced UAV functionality (Tan, Zhu et al, 2007):

- constant radio communication could not be guaranteed, especially with a sufficient bandwidth that could ensure both upload of controls and download of sensor data
- on-board instrumentation was not sufficient to provide complete situational awareness to the pilot on-ground
- sufficiently complex control systems could not be developed, and had to rely on analog technology
- safety issues were exacerbated by the lack of situational awareness

As of 2010, UAVs have become a significant part of many military aviation forces, despite being relegated to marginal uses until the 1990s. More importantly, UAV usage is constantly increasing and is expected to be ever more relevant in the future. Three key enabler technologies can be considered responsible for this trend: sensors, computers and satellites. Satellites provide not only the means to ensure a constant radio link, but also navigation capabilities via the use of the Global Positioning System (GPS). Computers allow complex control systems to be installed on-board the UAV, especially when coupled with more sophisticated sensors that can provide a large amount of information that was previously unavailable.

As these technologies became available, UAVs started to be considered as an alternative to manned systems for several mission types. Interest in UAVs is fostered by specific advantages that can be highlighted (Schaefer et al, 2001):

- expendability, which is always an asset in military applications and sometimes also for civilian ones (for example, data gathering in extreme environmental conditions, such as tornadoes or volcanoes)
- better flight performance due to the absence of the pilot and life support systems, allowing for longer endurance, heavier payload and possibly tighter manoeuvres
- reduced operating costs, especially when the payload is small and light (advances in sensor technologies were significant from this point of view)

In (USAF, 2009), ten key assumptions are made, forming the basis for the entire report; while the report is military in focus, many of these assumptions are valid for civilian applications too:

- Integration of manned and unmanned systems increases capability across the full range of operations
- UAS are compelling where human physiology limits mission execution (e.g. persistence, speed of reaction, contaminated environment)
- Automation with a clear and effective user interface are the keys to increasing effects while potentially reducing cost, forward footprint, and risk
- Modular systems with standardized interfaces are required for adaptability, sustainability, and cost reduction
- Agile, redundant, interoperable and robust command and control creates the capability of supervisory control of UAS ("man on the loop")

A very important emerging concept is the fact that, while UAVs are unlikely to completely replace manned aircraft in the near future, they can certainly operate alongside, especially once UAV operations will be able to integrate seamlessly with manned ones. It is clear that this is not only valid in the military field, however at present civilian use of UAVs is very limited, especially when compared to the widespread and fast-increasing adoption by military forces. This is mainly due to safety concerns (DeGarmo and Nelson, 2004), as the circulation of UAVs within civilian airspace will be allowed only when sufficient guarantees are in place. In fact, there is at present a regulatory gap regarding UAVs, which are usually confined to segregated airspace in European and North-American countries. One of the bigger challenges towards their civilian use is the development of new legislation, tailored for the particular conditions of operation that are typical of a UAV.

Once this is achieved, it is foreseeable that the civilian UAV market will see a rapid expansion, since UAVs are ideal platforms for certain types of applications, including surveillance, environmental monitoring (Wegener et al, 2004) and communications relay. In such cases, a small UAV could possibly perform the mission at a fraction of the cost, which would open up large market segments that were previously unreachable. For example, in (Ambrosia et al, 2004) a pioneering project regarding the use of UAVs in a fire-prevention role is presented. In such a case, it is clear that a fleet of small UAVs with a temperature sensor payload could monitor large forest areas more efficiently than manned alternatives: at the state of the art, such UAVs would probably weigh less than 10 Kg, whereas on a manned aircraft the pilot himself is weighing far more, thus the potential advantage in terms of operating costs is great and evident.

A possible future scenario in which UAVs are fully integrated within normal Air Traffic Management (ATM) will require consistent advances in terms of safety, especially regarding situational awareness and autonomy. Autonomy in particular is set to be one of the key development areas for UAVs: the absence of a pilot on-board can inevitably cause a lack of situational awareness, especially in cases where communication with ground control systems is severed. In such cases, a UAV capable to make autonomous decisions will evidently perform better than one which is fully controlled by a human pilot, which is on the ground rather than on-board.

1.1 UAVs and autonomy

How can the autonomy of a UAV be defined? In (Ferry et al., 2007), the authors state that "true autonomy will exist when a UAV can operate within a command and control structure, accepting inputs and generating outputs in order to appear indistinguishable from a human-controlled vehicle". While this is not currently feasible, it provides a good long-term objective for UAV autonomy-related research.

It is an interesting exercise to think about what normally happens when a pilot is assigned a mission, and this can be very helpful in understanding the various levels of decision-making processes that are involved in true autonomy. Considering for example a scenario where a pilot is asked to take aerial pictures at several locations, the mission would probably evolve along these lines:

- The pilot's supervisor assigns him the mission, providing information about the locations to be photographed
- The pilot gathers available information regarding the area of flight, such as the current weather, the situation of air traffic and similar
- Based on this information, the pilot prepares a flight plan
- The pilot then prepares the aircraft for departure and begins the mission
- As the mission is in progress, the pilot adjusts the flight plan according to newly available information (for example, avoiding an area of stormy weather that was previously unknown)
- In case of contingencies (for example, an aircraft fault or a problem related to Air Traffic Control), the pilot decides the course of action to be taken (for example, continuing to follow the original plan, or diverting to a different destination)
- The mission is finished when the aircraft lands at its destination airport, regardless of whether the objectives have been accomplished

There are thus four major activities involved: communication (to obtain objectives and information), flight planning, flight plan execution and replanning. Out of these, communication is the most complex, when trying to make the UAV indistinguishable from a human-controlled aircraft. However, a properly designed user interface and data-link can make the communication task straight-forward. Autonomous flight plan execution is completely achievable with the current technical capabilities (many current aircraft are equipped not only with an autopilot/autothrottle system to keep aircraft attitude and speed, but also with a Flight Management System that provides navigation functions). Autonomous flight planning and replanning are instead not easily achieved.

In (Chandler and Pachter, 1998), it is stated that "the essence of autonomous control is rapid in-flight replanning under uncertainty". Thus, a truly autonomous UAV must be able not only to plan how to best execute a given mission, but also to adapt the plan as the mission is being carried out. The key word is uncertainty: a flight plan can only be generated using available data, however much of the data that is needed cannot be guaranteed to be correct; in a realistic scenario, this data can be at best expected to be incomplete, if not completely wrong. Thus, the need for replanning arises: the UAV must be able to correct its flight plan quickly enough so that data uncertainty does not disrupt the outcome of the mission.

From the point of view of autonomy, current UAVs are widely different. On one side, there are UAVs that are little more than remotely-controlled planes. On the other side, certain UAVs (such as the Predator and the Global Hawk operated by the United States Air Force) are capable of autonomously performing an entire pre-planned mission (Miller et al, 2005). However, even these UAVs need human supervision, since flight plans might need to be changed during flight, especially in a dynamic environment such as a battlefield.

In (Clough, 2002), the author proposes a framework to identify the autonomy level of a UAV. Eleven different autonomy levels are defined, ranging from level zero (remotely-controlled vehicle) to level ten (fully autonomous, meaning the capability to make decisions without supervision), as depicted in Table 1.1. The breadth of autonomous capabilities described is very large and in fact each level of autonomy brings significant additions on the previous one. A remotely-controlled UAV is classified at level zero, which still assumes an appropriate suite of sensors, actuators and communication devices. Level one autonomy is defined as the ability to execute a pre-planned mission, and this means that a whole set of capabilities is being added on top of level zero autonomy, and in particular flight control and navigation capabilities. Levels two to five focus on the development of single-UAV capabilities, most importantly health management, trajectory optimization and contingency management; the type of autonomy reached within these levels allows a UAV to perform better in uncertain and dynamic environments, since the UAV can adapt its flight plan during the mission in order to actively respond to changes. Levels six to ten are fully dedicated to the integration of the UAV within a team of UAVs, that can share sensor information and cooperate to reach overall mission objectives.

Current generation UAVs rank very low in this framework, as can be seen in Sholes (2007). Furthermore, progress in the autonomy field has been very slow. For example, in 1985 the Pioneer UAV provided capabilities that placed it a little below level one (execute pre-planned mission), while the Predator UAV in 1996 and the Global Hawk in 2004 can be placed between level one and level two (changeable mission, meaning that the UAV can autonomously switch between several pre-determined flight plans). In other words, current UAVs can autonomously carry out an entire pre-planned mission, but need human supervision in order to respond to unforeseen situations, since they do not include Mission Management functionality.

Table 1.1. Autonomy Level framework (Reproduced from (Clough, 2002)).

Level	*Level Descriptor*	*Perception/Situational Awareness*	*Analysis/Decision Making*	*Communication/Cooperation*
10	*Human-Like*			
9	***Multi-Vehicle Tactical Performance Optimization***	Detection & tracking of other air vehicles within airspace	Full decision making capability on-board Dynamically optimize multi-ship group for tactical situation	Distributed cooperation with other air vehicles On-board deconfliction and collision avoidance Fully independent of supervision/control if desired; No centralized control within multi-UAV group
8	***Multi-Vehicle Mission Performance Optimization***	Detection & tracking of other air vehicles within local airspace OK to operate in controlled airspace w/o external control	Continuous mission/trajectory evaluation & replan - optimize for current mission situation Avoid collisions and replan/optimize trajectory to meet goals, etc	External supervision - abort/recall or new overall goal On-board deconfliction & collision avoidance Distributed cooperation with other A/V's
7	***Real-Time Multi-Vehicle Cooperation***	Detection of other A/V's in local airspace Multi-threat detection/analysis on-board	Continuous flight path evaluation & replan Compensate for anticipated system malfunctions, weather, etc - optimize trajectory to meet goals, manage resources, avoid threats, etc	On-board collision avoidance Uses off-board data sources for deconfliction & tracking Hierarchical cooperation with other A/V's
6	***Real-Time Multi-Vehicle Coordination***	Detection of other A/V's in local airspace Single threat detection/analysis on-board	Event-driven on-board. RT flight path replan - goal driven & avoid threats RT Health Diagnosis. Ability to compensate for most failures and flight conditions - inner loop changes reflected in outer loop performance	On-board collision avoidance Uses off-board data sources for deconfliction & tracking Assumed acceptance of replan, External supervision - rejection of plan is exception Possible close air space separation (1-100 yds)
5	***Fault/Event Adaptive Vehicle***	Automated Aerial Refueling & Formation sensing Situational awareness supplemented by off-board data (threats, other A/Vs, etc)	Event-driven on-board. RT traj replan to new destination RT Health Diagnosis. Ability to compensate for most failures and flight conditions. Ability to predict onset of failures (e.g. Prognostic Health Mgmt) On-board assessment of status vs trajectory	On-board derived vehicle trajectory "corridors" Uses off-board data sources for deconfliction & tracking External supervision - accept/reject of replan Possible close air space separation (1-100 yds) for AAR, formation in non-threat conditions
4	***Robust Response to Anticipated Faults/Events***	Threat sensing on-board	RT Health Diagnosis (Can I continue with these problems?). Ability to compensate for most failures and flight conditions (e.g. Adaptive inner loop control). Automatic trajectory execution. On-board assessment of status vs mission completion	Secure within LOS electronic tether to nearby friendlies Offboard derived vehicle "corridors", Medium vehicle airspace separation (100's of yds) Threat analysis off-board
3	***Limited Response to Real Time Faults/Events***		RT Health Diag (What is the extent of the problems?) Ability to compensate for limited failures (e.g. Reconfigurable Control) Automatic trajectory execution	Health Status monitored by external supervision Off-board replan, Waypoint plan upload Wide airspace separation requirements (miles)
2	***Pre-loaded Alternative Plans***		RT Health diagnosis (Do I have problems?) Automatic trajectory execution (via waypoints) Preloaded alternative plans (e.g. abort)	External commands - alternative plans, approvals, aborts Reports status on request or on schedule Wide airspace separation requirements (miles)
1	***Execute Preplanned Mission***	Situational awareness via Remote Operator Flight Control and Navigation Sensing	Robotic/Preprogrammed Pre/Post Flight BIT	External control via low level commands Reports status on request Wide airspace separation requirements (miles) No on-board knowledge of other air vehicles - all actions are preplanned
0	***Remotely Piloted Vehicle***	Flight Control (attitude, rates) sensing Nose camera Situational awareness via Remote Pilot	N/A	Remotely Piloted Vehicle status data via telemetry

It is expected that UAVs will present increasing levels of autonomy in the future. Many research studies focus on two trends: control of UAVs by personnel without extensive pilot training, and control of multiple UAVs by a single user. The latter trend especially has attracted significant research efforts, focusing on various aspects of the problem: in particular, formation flying (Mehra, Boskovic and Li, 2000), UAV group hierarchy (Li et al, 2002; Chandler et al, 2002) and UAV task planning (Gancet et al, 2005; Cummings et al, 2007).

Military applications usually allow for earlier implementation of new technologies, thus the biggest advances in UAV autonomy are being made in this area. Current civilian applications instead involve low levels of autonomy, focusing most of the UAV functionality on the pilot (UAV Task Force, 2004). However, it is clear that they would benefit even more from increased levels of autonomy, since it would allow decreased operating costs and would result in UAVs being routinely used in applications where they are not currently considered.

Increased autonomy introduces new safety concerns, however ultimately it can improve safety of the systems; for example, an autonomous UAV should be capable to quickly make decisions in the case where a system fault has occurred, while a remote pilot could possibly not be aware of the problem (due to a lack of situational awareness that cannot be negated for a remote pilot).

While operational UAVs have not progressed much in terms of autonomy, significant research efforts have been undertaken. (Veres et al, 2010) provides a comprehensive review of the different subjects that have been explored, focusing not only on UAVs, but on autonomous vehicles in general. Significant theoretical and practical advances have been achieved, especially in the areas of path planning (Rathbun et al., 2002; Higashino et al., 2004; Chantery, Barbier et Farges, 2004; Geiger et al., 2006), fault diagnosis (Boskovic and Mehra, 2003; Drozeski et al., 2004) and navigation (Riseborough, 2004; Ludington et al., 2006).

However, most of this research is focused on very specific features and in fact there is little literature regarding the integration of these functionalities into a single architecture. While these advances represent the technological enablers for a truly autonomous UAV, autonomy will only be achieved through their integration into a "system of systems".

1.2 Reason for the study

This section will explain the reasoning and practical processes that were at the basis for the present PhD project. Theoretical considerations are made, then a separate project which heavily influenced the PhD work is presented. Finally, the objectives for the project are clearly stated; precise goals and limitations are defined, so as to outline a project which is both significant and feasible.

1.2.1 Intelligence, safety and affordability in autonomy

In (Clough, 2005), the author declares that the basic challenges faced by researchers in autonomous control of UAVs are intelligence, safety and affordability. Let us consider these aspects separately.

The intelligence of a UAV can be described as its ability to perform with minimal supervision the same tasks normally performed by a manned aircraft. This involves several types of capabilities: planning, communication, navigation, health management. An intelligent UAV must be able to perform all of these activities, at least to some degree. However, the activities are very different in nature and state-of-the-art technology can accomplish them through the use of a wide range of techniques and algorithms. Thus, an intelligent UAV must be able to integrate different types of technology and algorithms, so that each function can be achieved efficiently.

Safety has always been one of the most important development paths for the aeronautical industry and recent commercial aircraft are statistically one of the safest transportation modes (Wells and Rodrigues, 2004). However, this has always focused on ensuring the safety of on-board passengers and crew. Furthermore, a key component of safety features is the on-board presence of extensively trained pilots, who can quickly recognize abnormalities and take mitigation action. Safety for UAVs has been under-evaluated, since they have always been considered expendable. While this is only a moderate issue for the military sector (mainly concerned by the cost-effectiveness of UAVs), for the civilian sector safety remains paramount. If UAVs are to be used within civilian airspace, they must become able to seamlessly integrate with air traffic management, while guaranteeing the same safety levels of manned aircraft. This is made even more challenging by autonomy, since a large part of the safety cycle is human-centric (Krause, 2003). Two main separate development paths can be identified: the development of appropriate safety-related algorithms and techniques and the development of software engineering (since an autonomous UAV will certainly rely on advanced hardware/software systems, the problem of guaranteeing its safe operation has to be faced).

Affordability is supposed to be one of the key advantages of UAVs versus manned aircraft, however several factors have to be considered. Normally a UAV is not only constituted by the vehicle itself, but also by its ground-based components (it is more correct to talk about Unmanned Aerial System, UAS). Affordability of a UAV should therefore consider the cost of both the vehicle itself and its supporting infrastructure. For many mission types, it should generally be possible to implement UAV systems that are cheaper to operate, since absence of an on-board pilot brings an obvious weight reduction. This becomes particularly important for the civilian sector, since lower operation costs should represent the main advantage of unmanned versus manned aircraft. The key point is then achieving intelligence and safety while maintaining affordability.

1.2.2 The ASTRAEA project

In 2006, the Autonomous Systems Technology Related Airborne Evaluation and Assessment (ASTRAEA) programme was launched in the UK. This was a multi-million pound programme, involving both industry and academia, whose declared goal was the development of technologies that would make possible the use of autonomous UAVs in civilian airspace in the near future. The University of Sheffield participated in ASTRAEA as a partner of Rolls-Royce, who were tasked with the development of the propulsion and power subsystem. Current jet engines are equipped with Full Authority Digital Engine Controllers (FADECs), and pilot interaction is often limited to engine activation/deactivation and throttle demand setting (Harris et al., 2000). Since the FADEC automatically reacts to dangerous engine running conditions (surge, compressor stall, etc.), the absence of an on-board pilot would mainly cause long-term safety issues: in case of a developing fault, it is the pilot's task to determine the best course of action (for example, aborting the mission or limiting the throttle demand on an engine).

Rolls-Royce's role in ASTRAEA was focused on the development of supervisory on-board software that would perform advanced real-time Engine Health Management (EHM) functions, including fault detection, fault isolation, fault prognosis and fault mitigation. The University of Sheffield collaborated with Rolls-Royce by working in parallel to provide similar functionality using a very specific type of technology: Intelligent Agents (IAs). IAs represent a relatively new software engineering paradigm, that has found wide adoption in certain application fields (mostly web-related, but also

Air Traffic Management, for example), but are not yet fully developed. The project was aimed at developing proof-of-concept EHM functionality, but also at evaluating the technology readiness level (TRL) of IAs for the conservative aerospace industry.

The work undertaken resulted in the development of a proof-of-concept EHM system, based on the fusion of Soar IAs (which will be thoroughly described in Chapter 2) with other control techniques, that could perform the fault prognosis and fault mitigation actions. While the system was demonstrated to be feasible, no significant advantages over more conventional technology could be highlighted in this application field. However, this lack of clear advantages was deemed to be caused by the restricted degree of freedom which was characteristic of the problem: since the pilot (or the EHM system in this case) has actually a very limited range of possible actions to take regarding an engine, the IA technology could not be fully exploited.

While pilot decisions regarding EHM are quite restricted in scope, during the course of a mission a pilot usually has to take a large number of decisions of different types. Automated systems can usually help the pilot in performing low-level activities, but the Mission Management activity is normally fully accomplished by the pilot. The problem space for EHM is quite limited; instead, it is extremely complex for Mission Management. It was thus decided to explore the possibility of applying the IA technology developed under ASTRAEA to the realm of UAV Mission Management.

1.2.3 Statement of intention

The generic goal of the work which is presented in this thesis is the development of a software system based on the fusion of Soar IAs and traditional control techniques and capable of managing and controlling a UAV for an entire mission with minimal human supervision. This is clearly too generic and to further define the goal it is useful to refer to concepts outlined in Section 1.1. Considering the framework for the evaluation of the autonomy level of a UAV that was introduced in (Clough, 2002), the goal is to achieve a system that can be placed in the proximity of autonomy level 5 (realtime multi-vehicle coordination). Table 1.1 (from Sholes, 2007) summarizes the definitions and properties of the autonomy levels. Using this framework, the goal is to build a system that:

- controls a single UAV, thus excluding cooperation with other vehicles
- coordinates with other vehicles at a simple level, involving collision avoidance but also sensor data fusion (particularly for information on external targets) but not formation flying or autonomous tactical cooperation
- is assigned a set of objectives by an external operator, whose supervision is not further required (if the mission is to be performed according to the original parameters – the operator might want to change mission objectives in-flight)
- can autonomously make decisions (if, when and how an objective is to be pursued), depending on the current situational awareness (environment, threat, aircraft health, etc.)
- can autonomously develop a flight plan and then update it during mission execution in response to changes in the perceived situation
- can react to airframe faults taking appropriate mitigation action, by prognosing fault effects at the mission level
- possesses a sufficient level of intelligence so that flight plans developed are not only viable but also reasonable (if not optimal), for example minimising fuel consumption and avoiding flight within known danger areas
- can fully operate under real-time constraints

- can be implemented using common-off-the-shelf (COTS) hardware, so as to reduce costs
- can easily integrate external components that add new functionality to the basic one

The choice of Soar IAs as the basis for the system (even though integrated with other technologies) will be motivated in detail in Chapter 2. In this section it is important to highlight how the Soar architecture is assumed to be capable of both providing a sufficient degree of intelligence (which is needed to provide the features previously listed) and operating under real-time constraints when deployed on COTS hardware, as demonstrated in (Kalus and Hirst, 1998).

The project is not aimed at the development of a specific UAV, but rather at the development of a generic software architecture for UAV mission management; nonetheless, generic long-term goals should be set:

- the software architecture is not explicitly military or civilian in nature, but civilian applications should always be considered
- the architecture is built for fixed-wing UAVs; while many high-level concepts can be shared with rotorcraft UAV (or even marine vehicles), low-level algorithms are completely different between these types of vehicles. The architecture can be easily adapted to support other types of vehicles, but the project will stay focused on fixed-wing UAVs
- advanced (and expensive) UAVs should not differ much from low-cost ones in terms of mission management; however, the architecture is generically developed for deployment on low-weight low-cost UAVs
- certification of UAVs is an unresolved issue, but it is important to design the architecture so that future certification is a viable process; in this light, the Soar architecture is considered to be a valid alternative, since it is already used on other systems (although not aeronautical ones)

Finally, it is important to define the level of advancement to be reached during the project. The architecture and in particular the Soar IAs on which it is based are to be fully developed. Supporting software will be developed to the extent that is necessary in order to validate functionality of the architecture. Validation will be achieved using simulation techniques, thus implementation of the system will be limited to simulation environments (Matlab/Simulink), excluding deployment on actual real-time systems at this stage. The deployment of the system on actual COTS hardware, eventually leading to flight testing, is a possible continuation path for this project. The system is designed with prospective implementation in mind, such as in (Jang and Tomlin, 2002), where COTS PC/104 hardware is used to deploy the developed software on the target aircraft. However, this work will not constitute part of this thesis.

In light of its scope and structure, the software architecture is called Soar-based Autonomous Mission Management System (SAMMS).

1.3 State of the Art

As already stated, the amount of research on UAV autonomy during the 2000-2010 decade has been significant, but focused mostly on the development of separate advanced technologies that could be used to improve autonomy. Many examples of such research projects are listed in (Veres at al, 2010). It can be noted that most projects are aimed at achieving very specific goals in terms of UAV autonomy, particularly focusing on the improvement of capabilities such as path planning, fault detection and sensor-based navigation.

In this section, several UAV-related projects will be briefly presented, with the objective of describing the state-of-the-art in the field of UAV autonomy. The projects are presented in chronological order, so as to outline progress and changes in perspective made during the decade. The Australian project focused on operation of the Codarra Avatar UAV is presented last; this project shares many characteristics with the present PhD project, so it is described more in detail, so as to allow the a clear definition of similarities and distinctions between the projects.

1.3.1 WITAS

The Wallenberg Laboratory for Information Technology and Autonomous Systems (WITAS) project was launched in 1997 at Linkoping University in Sweden and is still being continued through the Unmanned Aircraft Systems Technologies Lab (UASTech Lab) which was created in 2004 as a spin-off activity. In (Doherty et al, 2000), the project's goals and organization are described. WITAS is a project with a wide scope, and is focused at the development of:

- reliable software and hardware architectures with both deliberative and reactive components for autonomous control of UAV platforms
- sensory platforms and interpretation techniques (active vision systems)
- efficient inferencing and algorithmic techniques to access geographic, spatial and temporal information regarding the operational environment
- planning, prediction and chronicle recognition techniques to guide the UAV and predict and act upon behaviours of vehicles on the ground
- appropriate modelling tools and simulation, specification and verification techniques for the project

Many publications are linked to the WITAS project, but two in particular are significant in showcasing the achievements. In (Wzorek and Doherty, 2006), a motion planning framework for a fully deployed autonomous UAV is presented; two sample-based motion planning techniques, Probabilistic Roadmaps and Rapidly Exploring Random Trees, are used to perform short-term path planning (accurate trajectory planning), also providing real-time reconfiguration capabilities; the framework is verified through simulation and in actual flight. In (Doherty et al, 2009), a temporal logic-based task planning and execution monitoring framework is presented; the framework is integrated into a fully deployed rotor-based unmanned aircraft system, showing the potential use of such a framework in real-world applications; the TALplanner, a task planner based on temporal logic, is used to generate mission plans and show how knowledge gathered from the appropriate sensors during plan execution can be used to create state structures, incrementally building a partial logical model representing the actual development of the system and its environment over time.

The project's highlight is the integration of vision-based navigation and task-based planning. Differences with the present PhD project are:

- vision-based navigation is not included in the present PhD project
- a higher amount of human supervision is expected in WITAS
- plans in WITAS are generated using a reduced amount of information, due to a focus on internally provided information as opposed to externally provided
- the WITAS project is focused on rotorcraft

1.3.2 ReACT

The Reliable Autonomous Control Technologies (ReACT) and Technologies for Reliable Autonomous Control (TRAC) are projects undertaken by Lockheed Martin and

General Electric with the sponsorship of NASA. Their generic goal is the development of highly reliable, adaptable air vehicles that are capable of safely and transparently operating within controlled airspace (FAA or military) in the same manner that piloted aircraft do today (Schaefer et al, 2001). These parallel projects are multi-year efforts that will include a significant flight test element, with a planned demonstration on an F-16 testbed aircraft.

Within ReACT/TRAC, a UAV's autonomy is augmented by introducing the concept of the "Active State Model" (ASM). Borrowed from psychology, a system becomes "active" when it has these properties: self-image, self-awareness, and ability to make self-decisions. The system is then able to perform purposeful transitions or motions with a certain degree of autonomy from the environment. Under ReACT, the diagnostic and prognostic components provide the vehicle with self-image and self-awareness. Intelligent decision making combined with execution and monitoring capabilities are offered by the planning components to further extend the self-awareness of the agent.

Implementation of the system led to an architecture based the integration of four components: Beacon-based Exception Analysis for Maintenance (BEAM), Spacecraft Health Inference Engine (SHINE), Closed-Loop Execution and Recovery (CLEaR) and Autonomous Command Executive (ACE). BEAM and SHINE provide health management capabilities, while ACE provides low-level trajectory control and CLEaR provides high-level mission planning. The ACE component in particular is important as it has to fuse information from the other components in order to make appropriate decisions. Some of the features included in ACE are flight envelope protection, large disturbance rejection and fault corrective action (Johnson et al, 2001).

The project's highlights are its advanced health management capabilities and the capability to react quickly to mishaps. However, the work is focused on reflexive actions (immediate responses to detected events) rather than planning functions. The CLEaR component should provide the planning and high-level mission management capabilities, but its functions are not clearly defined and as a result it is underdeveloped.

1.3.3 Other studies

In (Boskovic et al., 2002), a multi-layer control architecture for UAVs is presented. This architecture is based on the hierarchical subdivision of UAV mission management tasks. Four layers are outlined: at the top is the *decision-making* layer, which makes high-level mission-related decisions in near-real-time; next is the *path planning* layer, which controls long-term navigation; then there is the *trajectory generation* layer, which oversees short-term navigation; finally, the fault-tolerant *redundancy management* layer detects airframe faults and performs mitigation actions through reconfiguration of the UAV. The concept of a hierarchy between functions is adopted in the present PhD study, although with several modifications; in particular, fault detection and mitigation is integrated across all functions rather than as the bottom-level function. While the project was initially aimed at generic UAV mission management, it later focused on fault-detection capabilities (Boskovic and Mehra, 2002) and mission management for multiple UAVs (Li et al, 2002).

In (Kim and Shim, 2003), a hierarchical flight control system for rotorcraft UAVs is presented. The system is implemented on a Yamaha R-50 radio-controlled model helicopter. Radio controls are substituted by a flight control computer based on PC/104 hardware running the QNX real-time operating system. A multi-loop controller is implemented, including three loops: the inner attitude controller, the mid-loop linear velocity controller and the outer position controller. A non-linear model predictive controller is added as a tracking layer, in order to ensure stability of the UAV, that is subject to very complex helicopter dynamics. At the top of the hierarchy is the path planning layer, which implements the following functions: waypoint navigation, pursuit-evasion, target tracking and autonomous landing. While the project is very successful in developing low- and mid-level control of the UAV, high-level decision-making capabilities are lacking, especially regarding health management. The project is focused on rotorcraft, and it is most successful in developing rotorcraft-exclusive functionality (low- and mid-level control).

In (Sullivan et al, 2004), a NASA project on intelligent mission management for UAVs is presented. The approach divides goal-directed autonomy into two components, an on-board Intelligent Agent Architecture (IAA) and a ground based Collaborative Decision Environment (CDE). These technologies cut across all aspects of a UAV system, including the payload, inner- and outer-loop onboard control, and the operator's ground station. The paper describes each UAV and ground station component (human interface, avionics, autonomous operation and sensor and payload). The autonomous operation system is based on APEX, a general-purpose autonomy system that has been used to model diverse systems such as autonomous helicopters and humans monitoring air traffic control systems. The project's highlight is the extensive focus on sensor and payload management. In the present PhD project, sensor and payload management are assumed to be performed by external components, which filter sensor information and format it so that it is understandable by the mission management system, while at the same time operating the payload according to the current mission needs.

In (Valenti et al, 2004), a UAV is assigned to support a piloted aircraft. A Natural Language Interface is used to issue mission level commands to the UAV in real-time. Several mission types are available, including fly to waypoint, loiter pattern, search pattern, classify target, attack target, battle damage assessment and return to base. The pilot can issue these high-level tasks to the UAV by simply speaking to it, and the UAV is capable of autonomously performing the assigned tasks. Several of the task types identified in the paper are also implemented in the present PhD project; the system provides an excellent human user interface and low- to mid-level control capability, but lacks the ability to manage multiple concurrent tasks.

In (Theunissen et al, 2005), the problem of integration of autonomous UAVs into controlled airspace is addressed. A typical UAV mission profile is outlined and corresponding Air Traffic Control (ATC) and Command and Control (C2) interaction is described. ATC communication is mostly important in order to fly within civilian airspace, while C2 communication is necessary to integrate the UAV with its operators. The highlight of the project is the clear definition of interfaces and communication protocols that are needed in order to fly within controlled airspace. In the present PhD project, ATC interaction is taken into account by introducing "no-fly" zones, which the UAV has to avoid while autonomously flying.

In (Sinsley et al, 2008), an Intelligent Controller developed at Penn State University is described. The Intelligent Controller is responsible for mission-level control of each autonomous device participating in an independent or collaborative operation. The controller has two main modules, Perception and Response. The Perception module forms an internal representation of the external world, while the Response module uses this world-view to plan and carry out the mission. The system is used to demonstrate leader-follower formation flight using two heavily-modified radio-control trainer aircraft, fitted with on-board computers that implement the Intelligent Controller. The project is successful in achieving low- to mid-level autonomous flight, but lacks multi-objective planning capabilities (which are among the goals of the present PhD project).

In (Veres, 2010), several autonomous systems developed at the University of Southampton are described, including marine, aerial and space vehicles. These systems share a common autonomy definition framework. Autonomy is tiered into three levels and at each level is assigned a corresponding type of implementation technology. Low-level autonomy is achieved using reactive agents; medium-level autonomy is achieved through deliberative architectures; high-level autonomy is achieved through belief-desire-intention (BDI) architectures or multi-layered architectures. The intelligent-agent-based approach is similar to the one used in the present PhD project, although the specific agent software used is different.

1.3.4 The Codarra Avatar project

The Codarra Avatar is a lightweight UAV, purpose-built for reconnaissance and surveillance missions. It is launched by hand, and propelled via an electric motor, with a flight endurance of approximately sixty minutes. Recovery is by parachute. The payload-bay can accommodate sensors up to 1.5 kg in weight and standard sensors include: GPS receiver, barometric altimeter, and a Pitot-static air speed indicator. Communication range to the Ground Control Station is approximately 10 km.

As part of a program led by the Air Vehicles Division of the Australian Defence Science and Technology Organisation (DSTO), the Codarra Avatar was provided with an autonomous operation capability (Lucas et al, 2004). The goal is not to develop a mission-ready UAV but to provide DSTO scientists with a research test bed suitable for studying the impact of providing UAVs with autonomy. The research program driving the UAV development focuses on two distinct areas: aircraft platform management – including air and flight worthiness, health monitoring and cost of ownership; and autonomous software development issues – including validation and verification, architectures for coordinated autonomous behavior, and autonomous controllers inspired by models of human cognition.

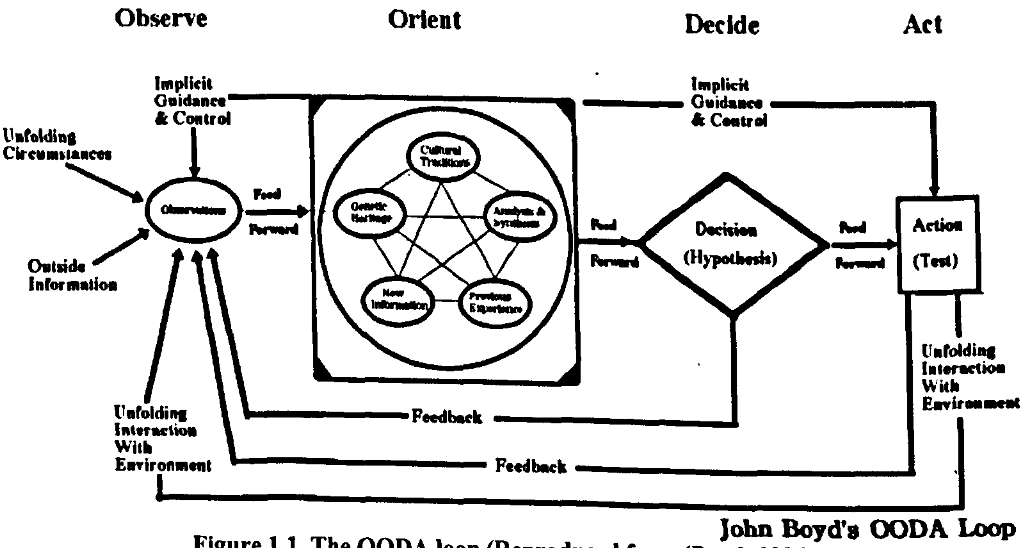


Figure 1.1. The OODA loop (Reproduced from (Boyd, 1996)).

The mission management system developed for the Codarra Avatar is designed and implemented using the agent-programming paradigm, considered to be a powerful, scalable and flexible framework for building autonomous systems (Karim et al, 2004).

Using the JACK Intelligent Agents programming language, a Belief-Desire-Intention (BDI) model of cognition (Wooldridge, 2001) was implemented to provide high-level decision-making capabilities. The system was integrated with an existing autopilot and auto-stabilisation system that performs basic flight control on the UAV.

During the project, two distinct approaches were used for the design of the mission management capability (Karim and Heinze, 2005); both make use of the JACK Intelligent Agents platform for their implementation. The first approach adds a simple mission-control layer matched to the flight control system; the second adheres more strongly to the agent metaphor and implements a design based on Boyd's observe-orient-decide-act (OODA) loop model of military decision making (Boyd, 1976; Boyd, 1996; see Figure 1.1).

The Codarra Avatar guided by the autonomous mission management system was flight tested for the first time in 2004 (Ferry et al, 2007). The flight began with ground control flying the UAV to an altitude of 400 ft, then engaging both the autopilot and the Intelligent Agent software. The pre-programmed mission was to hold a northerly course and altitude for about 1 km to waypoint "Alpha", turn 180 deg. and return to the launch area. However, the onboard Intelligent Agent was given a further objective. It received two optional waypoints, about 500m on either side of the original track, one to the east and the other to the west. It also received a GPS decision point along the track. Wind conditions were calculated through observations of differences between aircraft heading and the true path and velocity as defined by GPS. When the Avatar reached this decision point, the Intelligent Agent was able to reason which optional waypoint could be reached fastest under the prevailing conditions. The Intelligent Agent (IA) then directed the autopilot to turn onto the new course, fly to the selected waypoint, and return to the launch area.

The Codarra Avatar project presents obvious similarities with the present PhD project; most notably, they are both driven by the idea of using an IA-based cognitive model of the human brain to provide high-level reasoning capabilities that oversee the operation of a UAV, while low-level functions are achieved through normal algorithms (autopilots and such). However, the implementation is radically different: in the Codarra project, the JACK IA programming language is used, as opposed to the Soar IA architecture used within the present project.

In its original form, the *State Operator and Result* (Soar) architecture is a proposed unified theory of cognition (e.g. a set of general assumptions for cognitive models that account for all of cognition); as such, it explains how intelligent organisms (or machines) flexibly react to stimuli from the environment, how they exhibit goal-directed behaviour and acquire goals rationally, how they represent knowledge and how they learn from their experiences. The computational implementation of Soar enables the definition of Soar agents; these are software agents whose behaviour is determined by both the Soar generic rules and by dedicated rules which actually define the behaviour of the agent.

The Soar architecture will be extensively described in Chapter 2; here, it is important to highlight the differences between the Soar and JACK architectures. Similarly to JACK agents, Soar IAs implement the OODA loop (although not explicitly) as they proceed through their normal 5-phase execution cycle (input, proposal, decision, application, output), but the similarities end here. Soar is a symbolic Artificial Intelligence (AI) programming language, useful in trying to fully model the human brain, while JACK is an evolution of the JAVA programming language, and as such relies on coding to provide cognitive behaviour. This is achieved through the implementation of the BDI model of agency, which instead is not included in Soar.

The Codarra Avatar project proved that achieving mission management capability through cognitive Intelligent Agents is a viable approach towards UAV autonomy. However, it is our opinion that the Soar architecture can provide a higher level of functionality compared to the JACK architecture, while maintaining similar performance regarding real-time operation and safety.

1.4 Notable Contributions

Due to the nature of this project, it is difficult to highlight specific aspects which constitute a significant contribution: the real value of the work does not lie in a particular component or finding, but in their connection and integration. It is possible however to highlight some achievements that more evidently stand out.

The most important contributions brought forward during this PhD project are:

- a Soar/Simulink interface (detailed in Section 2.8); while this is just an enabler component for the actual project, it is to our knowledge the only freely available interface between Soar and Matlab/Simulink; the Soar team provided support during development, and expressed interest for a possible inclusion within future Soar releases
- a set of Mission Management abstractions (detailed in Section 3.1), which form the basis for the autonomous planning functionality; no sufficiently detailed framework for the definition of flight plans in computerized form was found, so a dedicated one was developed; the framework is very generic and unrelated to the actual implementation techniques; furthermore, while currently optimized for use with fixed-wing UAVs, it could be easily adapted for use with any UAV type, or even with UMV (Unmanned Marine Vehicles) or UGVs (Unmanned Ground Vehicles), since many basic concepts are shared
- a fully-contained autonomous UAV simulation; while the project is focused on the development of high-level planning capabilities, a full simulation architecture has been developed, including low-level control algorithms (autopilots), a mathematical model of a UAV and a visual interface; its modularity allows it to be re-used within other UAV-related projects as a validation tool (as in the SEAS-DTC project described later)
- a set of three Soar Intelligent Agents which, integrated with low-level control software, reach the goals described in Section 1.2.3; two of the agents are used to perform the Mission Management Activity, with a third agent needed as an intermediate layer between them and the low-level control functions
- a generic discussion regarding the problem of Intelligent Agent verification (Section 2.6); while not presenting a solution to the problem, this section raises the need of novel validation strategies for IAs to be developed, in order to allow their use within safety-critical systems (such as a UAV)

It is important to note that the software architecture that will be described in this thesis does not embed all of the functionality that would be required by a UAV to achieve autonomy at level 4/5 of the Clough classification. The approach has been instead to develop constructs and techniques that allow the integration of subsystems that perform these functions into an overarching architecture.

Examples of such functionality are represented by failure management and payload management: while dedicated subsystems were not developed, a methodology to include them into the processes of flight plan generation and execution was embedded within the agents. As a result, information provided by these subsystems can be used to improve the flight plans generated during the mission management activity, and commands can be sent towards the subsystem during mission execution.

As will be demonstrated in this thesis, the SAMMS software architecture is capable, once integrated with the required subsystems, to achieve functionality which places it between levels 4 and 5 of the Clough classification. This level of autonomy is not achieved by any commercially available UAV; within the high-budget military application field, currently flying UAVs are below this level (state-of-the-art prototypes are probably beyond it, but not publicly known). It must also be noted that the SAMMS architecture is primarily meant to be applied to low-cost small UAVs.

Overall, its characteristics make SAMMS a strong candidate for use in civilian applications, where multiple-UAV coordination and cooperation are not required; low cost and the possibility of use by personnel without specific training (e.g. without pilot training) are particularly important for such applications.

1.4.1 Papers and documents

The work presented in this thesis led to the publication of five papers. Three of these originate from the original ASTRAEA work, while the other two describe the development process for a Soar-based Autonomous UAV Mission Management System. The papers are:

- Gunetti P., Mills A., Thompson H., "*A distributed Intelligent Agent architecture for Gas-Turbine Engine Health Management*", Proceedings of the 46th AIAA Aerospace Sciences Meeting and Exhibit, 7 – 10 January 2008, Reno, NV
- Gunetti P., Thompson H., "*A Soar-based Planning Agent for Gas-Turbine Engine Control and Health Management*", Proceedings of the 17th IFAC World Congress, Seoul, Korea, July 2008
- Gunetti P., Thompson H., "*Development and Evaluation of a Multi-Agent System for Gas-Turbine Engine Health Management*", Automatic Control in Aerospace on-line Journal, Year 3, Number 1, May 2010, http://www.aerospace.unibo.it
- Gunetti P., Dodd T., Thompson H., "*A Software Architecture for Autonomous Mission Management and Control*", American Institute of Aeronautics and Astronautics, AIAA InfoTech@Aerospace Conference 2010, Paper No. 2010-3305
- Gunetti P., Dodd T., Thompson H., "*Autonomous Mission Management for UAVs using Soar Intelligent Agents*", submitted to the International Journal of Systems Science, accepted for publication

Finally, the Autonomous UAV Mission Management System has been used during a Systems Engineering for Autonomous Systems Defence Technology Centre (SEAS-DTC) backed study on the use of Intelligent Agents for intelligent power management. This is captured in the Technical Report "Agent Architecture for Intelligent Power Management", which is Technical Report Number 4, written by M. Ong, P. Gunetti and H. Thompson, and delivered to SEAS-DTC in July 2010.

1.5 Thesis Overview

This thesis is organized into six chapters, plus an introductory and a conclusive chapter. References are at the end the main text body. The present chapter is the

"Introduction", in which the entire project was outlined by describing the context in which it is developed, justifying its validity and stating clear goals and limitations

Chapter 2 "The Technology" represents an overview of Intelligent Agent technology and highlights the expected advantages that justify its application to the field of UAV autonomy. In Section 2.1, theoretical aspects regarding IAs are introduced, first by clearly defining an IA, then by outlining the capabilities an IA should possess, and in particular describing inter-agent relationships that form the basis for their social ability. Section 2.2 gives an overview of the most significant among the multitude of IA architectures that have been developed; first, generic architecture types are described, then specific examples are presented. In Section 2.3, the choice of the Soar architecture between the ones presented in the previous section is explained and justified in detail. Section 2.4 provides a more detailed description of the Soar architecture, exploring its origins as a generic theory of cognition, describing the underlying concepts that are necessary to understand its functionality and briefly summarizing its technical implementation. In Section 2.5, examples of practical applications of the Soar architecture are presented; the applications range from the behavioural simulation of fighter pilots to the control of mobile robots. Section 2.6 is a generic discussion introducing the problem of IA verification; one of the key goals for the present PhD project is to develop a system for which an eventual certification process is viable, and this section underlines specific issues typical of IA software that could hamper this process. In Section 2.7, the work performed within the ASTRAEA is presented; this work can be considered as the origin of the present PhD work, and thus it is described in great detail, since it can be considered proof of the viability of the Soar/Simulink approach to building a complex system. Section 2.8 then focuses the attention on the development of the Soar/Simulink interface; this was originally developed for the ASTRAEA project and later adapted to the needs of the present PhD project. Finally, Section 2.9 briefly summarizes the chapter while introducing the next one.

In Chapter 3 "Architecture description", a high-level overview of the entire project is given. The chapter lays out the foundations for the following chapters, presenting useful concepts (that are used throughout the work), describing the relationship between various parts of the work and detailing minor components of the architecture. In Section 3.1, the theoretical background of the entire project is laid out; in particular, three "abstractions", used throughout the project both as a communication standard and a development standard, are described in detail. In Section 3.2, an overview of the general architecture design for the project is given; this is particularly important, since it explains how the overall system was decomposed and why, and then describes the interactions between the Soar intelligent agents and the other system components. Section 3.3 moves forward from the theoretical considerations made in the previous two sections by depicting the actual Simulink implementation of the simulation environment; the aspects discussed in Section 3.2 are thus put in practice, and some technical details needed by the system to work are explained. In Section 3.4, the UAV mathematical model used throughout the system for testing and validation purposes is described; the model is derived from a third-source model, to which several modifications had to be performed in order to adapt it to the project needs. Section 3.5 deals with the Autopilot subsystem, a minor component of the architecture which is nonetheless necessary for a meaningful validation; a fairly standard autopilot is used, and all control loops are described in detail, together with the intended modes of operation. Finally, Section 3.6 concludes the chapter by summarizing it and introducing the next chapter.

Chapter 4 “The Planner Agent” is entirely dedicated the description and testing of the most important component of the SAMMS architecture. Having defined Objectives, Entities and Actions in Chapter 3, The Planner Agent is a Soar agent that takes a set of Objectives and transforms it into a flight plan (a series of Actions) which also takes into account the presence of Entities. The chapter is divided into two main parts, with two sections per part (Chapter 5 and Chapter 6 will also follow this scheme). The first part of the chapter is dedicated to the description of the Planner Agent. Section 4.1 presents details regarding the Input/Output interface, that is used to execute the Planner within a Simulink model. The interface is based on the one described in Section 2.8; this has to be then adapted to the actual I/O needs of the agent. Section 4.2 is instead dedicated to a very thorough description of the Planner Agent. The agent is portrayed in detail, showing both the overarching structure and the most important practical points in coding. Separate subsections are dedicated to the algorithms embedded in the agent; in most cases, a general description is given first, followed by the details regarding their implementation within Soar. The second part of the chapter presents the testing campaign that was carried out in order to verify the agent’s functionality. In Section 4.3, the test scenarios that were used during the testing campaign are described. Due to the large number of input variables, it is unpractical (if not impossible) to test every possible input combination. Thus, a set of representative scenarios was introduced; the scenarios are developed with the aim of testing the Planner (and later the entire SAMMS architecture) under a wide range of conditions. A total of seven scenarios is described in the section; the scenario descriptions are separated from the test results since the same scenarios will be used during testing of the Mission Manager Agent and the Execution Agent in Chapters 5 and 6. Section 4.4 is dedicated to the results of the Planner Agent testing campaign. Results are shown for each of the scenarios introduced in Section 4.3; in most cases each scenario has several scenario variations. Results are mostly shown using graphical plots of the flight plans developed under each scenario. The chapter is concluded by Section 4.5, which summarizes it and introduces the next chapter.

Chapter 5 “The Mission Manager Agent” gives a thorough description of the titular component of the SAMMS architecture. The Mission Manager Agent can be seen as an accessory component, since the rest of the architecture can function without it, but in fact it brings some of the most advanced functionality within SAMMS. A consistent part of the novelty brought forward in this whole PhD project comes from the capabilities that are implemented within the Mission Manager Agent; in particular, the ability to dynamically change, abort or add mission objectives. The chapter is structured in a very similar manner to Chapter 4, being divided in two main parts that deal respectively with the agent’s description and its testing campaign. In Section 5.1, details regarding the Input/Output interface for the agent are given; as for the Planner Agent, the interface is based on the one described in Section 2.8 and adapted to suit the I/O needs of the Mission Manager Agent. Section 5.2 consists of a thorough description of the Mission Manager Agent; its structure is analyzed, giving great importance to the way the agent is coded in terms of the Soar language. Specific algorithms are described both from a general point of view and from the point of view of their Soar implementation. Section 5.3 begins the second part of the chapter, where the Mission Manager Agent testing campaign is reported; the section is dedicated to the description of test scenarios used during the testing campaign. The scenarios are the same as for the Planner Agent, however specific scenario variations are introduced in order to test specific algorithms and functions of the Mission Manager Agent. The scenarios are then put into practice in Section 5.4. Tests of the Mission Manager Agent are realized using a subset of the whole SAMMS architecture, and the results are visualized using the same plotting function as for the Planner Agent; while this means in fact that the results shown are of an indirect nature, it was deemed to be the best manner to show them. Finally, the chapter is concluded by Section 5.5, which summarizes it and introduces the next chapter.

Chapter 6 "The Execution Agent" presents a thorough description and a complete testing campaign of the third component of the SAMMS architecture. The Execution Agent constitutes an intermediate layer between the high-level planning functions, provided by the Planner Agent and the Mission Manager Agent, and the low-level control functions provided by the autopilot system. The chapter follows the structure of Chapters 4 and 5, with two separate parts, treating respectively the theoretical description of the agent and the testing campaign carried out to validate it. Section 6.1 gives details regarding the Input/Output interface for the agent; as for the Planner Agent, the interface is based on the one described in Section 2.8 and adapted to suit the I/O needs of the Execution Agent. In Section 6.2, the Execution Agent is thoroughly described, explaining the algorithms used for its implementation. In particular, for each Action type the resulting planned flight trajectory is shown. Section 6.3 begins the second part of the chapter, where the Execution Agent testing campaign is reported; the section highlights the two separate test methodologies that will be used during the testing campaign. In Section 6.4, the results for the first test methodology are presented; the results are presented showing the execution of each Action type, so as to allow an immediate comparison with the theoretical description presented in Section 6.2. Results for the second test methodology are instead presented in Chapter, since they constitute the final testing of the entire SAMMS architecture, with all its components integrated. Finally, Section 6.5 concludes the chapter with a brief summary and introduction to the next chapter.

Chapter 7 "Final Results" represents the conclusion of the practical part of the thesis, providing results of tests conducted on the entire system. In the chapter, scenarios selected from the ones presented in Chapters 4 and 5 are tested using the entire SAMMS simulation architecture. The results are shown focusing on the display of entire missions (rather than on single Actions as in Section 6.4). The first six sections (7.1 to 7.6) are dedicated to showing the results from the test scenarios; each section is dedicated to a separate test scenario. Section 7.7 concludes the chapter, summarizing results and introducing the next chapter.

Chapter 8 "Conclusions and Future Work" presents an analysis of the achievements of the project, trying to highlight significant contributions. The project is put into the perspective of actual implementation, exemplifying possible implementation strategies. A significant part of the chapter is related to the analysis of perceived issues for the project; in general, during SAMMS development some simplifications were made to limit the time required to complete the work, especially when more advanced versions were not expected to be meaningful within the project scope. Such issues are highlighted and possible solution strategies given. Section 8.1 mainly deals with the summary of achievements and the analysis of accomplishments. In Section 8.2, prospective future work regarding SAMMS is described, starting from possible improvements on the current architecture and finishing with a proposal for a realistic implementation strategy. Section 8.3 concludes the chapter and the thesis.

2. The Technology

In Section 1.2, it was stated that "the generic goal of the work which is presented in this thesis is the development of a software system based on the fusion of Soar Intelligent Agents and traditional control techniques and capable of managing and controlling a UAV for an entire mission with minimal human supervision". Thus, a key characteristic of this work is the use of Intelligent Agent (IA) technology.

This chapter will introduce the Intelligent Agent technology which will be used in the present PhD project. The generic form of Intelligent Agent is at first treated, from both a theoretical (Section 2.1) and a practical (Section 2.2, with examples of applications) point of view. The choice of the Soar architecture as the main development tool is motivated in Section 2.3, and then the Soar architecture is described in detail, again both from a theoretical (Section 2.4) and practical (Section 2.5) point of view. Considerations regarding the validation process for Intelligent Agents are made in Section 2.6. In Section 2.7, the ASTRAEA work from which the present PhD project originated is presented; this extends into Section 2.8, which introduces the Soar/Simulink interface that is used in the project and that was originally developed during the ASTRAEA work.

2.1 Intelligent Agents

Software affects almost every aspect of our daily lives and is an essential part of many military and civilian systems, including safety-critical and mission-critical systems (Long, 2008). The complexity of many of these systems has been growing exponentially, and this is particularly true within the aerospace industry. Software engineering as a discipline is becoming more and more important, in response to the need for software exhibiting a high level of functionality while guaranteeing safe operation. The concept of Intelligent Agent is one the main new software engineering approaches that are aimed at delivering the kind of capabilities that is going to be expected from software in the future.

Intelligent Agents (IAs) have been thoroughly discussed by computer science experts during the last ten to fifteen years. They represent a new approach to software programming which focuses on giving software a decision-making ability that can be used to manage unpredictable situations (Wooldridge, 1999).

In the traditional approach to software programming, a computer program is not expected to deal with unforeseen situations; the designer will anticipate all possible conditions and tell the program what to do in each situation. If the program is presented with a situation that it cannot recognize, at best it will give a message error and enter a new cycle of operation, while at worst it will simply crash without warning. For many applications this is acceptable; especially for simple systems, it is relatively easy to accurately predict all possible situations and therefore avoid unexpected exceptions. As systems become more complex, the number of possible conditions grows exponentially, leading to the practical impossibility of predicting all possible situations and the corresponding actions to be performed.

Several techniques which basically introduced the idea of mimicking human behaviour have been used to deal with these problems. In fact, one of the defining characteristics of a "human agent" as opposed to a traditional computer agent is the ability to deal with unexpected issues.

Intelligent Agents are based on this idea: they are built from scratch to be autonomous, by giving them the ability to understand their environment and to decide their course of action in order to actively change it. An IA also has to be able to deal with unexpected situations; it should always be able to decide for a course of action which the agent thinks will satisfy its goals. If the course of action proves to be wrong, the agent should understand this, try a new course of action and possibly learn from the mistake so that it will not be repeated.

Computer science experts have tried for many years to present a definition of IA. No definition has yet been agreed by everybody, as usually they are either too generic or too focused on a specific subject. In Jennings and Wooldridge (1998), an Intelligent Agent is defined as "a computer system that is situated in some environment and that is capable of flexible autonomous action in order to meet its design objectives". This definition is quite generic, but it points out some very important characteristics:

- an IA is a computer system
- an IA is situated in some environment and is capable of autonomous action upon it, implying that autonomy is paramount; the agent must have the ability to affect the environment directly or be able to communicate with other agents that can affect it
- an IA is flexible, distinguishing it from a simple Agent.

What is "flexibility" when speaking about an Intelligent Agent? Many researchers identify three characteristics which define it (Wooldridge and Jennings, 1995):

- reactivity: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives
- pro-activeness: intelligent agents are able to exhibit goal-directed behaviour by taking the initiative in order to satisfy their design objectives
- social ability: intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives

It is important to note how interaction is conceived when speaking about IAs: while in normal computer applications communication is functional, with IAs we have to view agent interaction as a kind of society. In fact, this is also an attempt to mimic human behaviour; human interactions are not straightforward, and IAs are built so that their social ability is not only functional but can also affect behaviour. To clarify this concept, an example is useful.

Let's consider two agents, agent A and agent B; agent B provides a service needed by agent A. In normal computer computing, agent A would send a request to agent B for the service and, provided that it has the proper authorisations, the service will be automatically executed by the agent B as soon as resources are available. When dealing with IAs, this is not true anymore. Agent A would send the request for the service as before, but agent B could just decide that this request is conflicting with its design objectives and deny it. Agent A should then be able to acknowledge the negative response and react accordingly, for example by presenting another request which is more likely to be accepted. Ultimately, a deal will be reached, unless the conflict is impossible to overcome. Note how this is similar to the way humans interact with each other: cooperation is sought and if a conflict arises, the search for a deal starts; if no deal is acceptable to both parts, then no cooperative action is taken – leading perhaps to competition or overpowerment.

The social ability of IAs is critical in understanding why they can be preferred to more traditional approaches (Wooldridge and Ciancarini, 2001). In fact, normal approaches, especially in the control field, involve modelling entirely a system. As systems grow more complex, deriving a precise model becomes more and more difficult and time-consuming. With IAs, the idea is to try to solve these problems focusing on simplicity and interaction. Relatively simple agents can be realized, which focus only on a specific section of a complex problem, but show intelligent behaviour in their own restricted environment. If properly laid out, the interaction of all the agents needed to cover the entire problem will result in a kind of "society" which is much more complex than the agents that form it. The interaction between the different simple IAs will result in a greater intelligence than the simple sum of each agent's intelligence.

It is evident from this that the true advantage of using IAs comes forward when multiple agents are used. In fact, while, especially in the Internet environment, single agents can be used, when dealing with complex systems we rarely come across single agents; it is usual to talk about Multi-Agent Systems (MAS), which represent the kind of agent society described in the previous paragraph. MAS can be organized into two different types of hierarchy: the horizontal relationship and the vertical relationship. Note how this reflects human society.

A horizontal relationship is one that occurs between two agents that are at the same level. They may have different environments and objectives (sometimes completely different), but neither will be able to overcome the other. If they need to interact, every effort is made to reach a deal, but agreement from both sides is needed if action is to be taken. This kind of relationship is useful when dealing with agents that are situated in different environments but whose actions are mutually effective.

A vertical relationship is one that incurs between two agents at different levels. Environment and objectives might be the same or different (often the lower level agent will be acting in a subpart of the environment of the higher level agent) but the higher level agent will always get the priority to decide the course of action. The lower level agent will be forced to do what required by the higher level agent, but may give suggestions as about what is perceived to be useful at a lower level. The higher level agent will basically use the lower level agent to achieve its own purposes, but should consider its "opinion" during the decision-making process. In this sense, a lower level agent is not really autonomous unless it gets authorisation from the higher level agent to act as it deems appropriate.

The combination of horizontal and vertical relationships can result in the creation of a complex agent society which should carry a much higher degree of intelligence than that carried by the simple agents forming it.

2.2 Intelligent Agent architectures

From a practical point of view an IA might look just as any other computer program. When writing a normal program, a programmer needs to determine the functions that it is supposed to perform (functional requirements), then he translates the requirements into some programming language, which is then compiled in order to obtain an executable optimized for the target platform. The process of building an IA is essentially the same; the difference actually lies in the functional requirements definition. The requirements have to be derived by thinking about how to give the agent a good degree of autonomy, how agents will be interacting if there is the need of a Multi-Agent System, and what is making the agent "intelligent". When this definition process is finished, the programmer can actually use normal programming techniques which are found to be suitable to give the agent its defining characteristics.

It is important at first to distinguish between IAs and other types of software. In particular, IAs presents many similarities with objects (as in object-oriented programming) and expert systems.

Object-oriented programmers often fail to see anything novel in the idea of agents. Considering the relative properties of agents and objects, this is not surprising (Wooldridge, 1999). Objects are defined as computational entities that encapsulate some state, are able to perform actions, or methods on this state, and communicate by message passing. Two distinctions with IAs can be made. The first is in the degree of autonomy; the defining characteristic of object-oriented programming is the principle of encapsulation (the idea that objects can have control over their own internal state), but an object can be thought of as exhibiting autonomy over its state, not over its behaviour. The second distinction between object and agent systems is with respect to the notion of flexible (reactive, pro-active, social) autonomous behaviour; the standard object model has nothing whatsoever to say about how to build systems that integrate these types of behaviour. Note that there is nothing preventing the implementation of agents using object-oriented techniques.

During the two decades over which the concept of IA has been in development, several types of agents and agent architectures have been proposed. From a theoretical point of view, the main types are:

- Logic architectures
- Reactive architectures
- Belief-Desire-Intention architectures
- Layered architectures

Logic architectures are based on symbolic Artificial Intelligence (AI) techniques. In these systems, the IA is given a symbolic representation of its environment and its desired behaviour, and syntactic manipulation of this representation provides the intelligence. Ultimately, intelligence is represented as logical formulae linked through logical deduction. Purely logic IAs are not practical for three main reasons: first, they rely on symbolic representations of the environment, but it is not clear how to properly construct this representation in a real-world environment; second, they become exceedingly complex as the agent problem space becomes larger, to the point of being intractable; finally, it is difficult to ensure real-time functionality when complex decision-making is to be made.

In Reactive architectures each agent is characterised by simple reactive behaviour, so that it acts only in response to some perceived change in the external environment. The agents are not intelligent, but intelligence in these systems is brought forward by the complex interactions between them. The subsumption architecture is the most prominent example (Brooks, 1991); within it, decision-making is realised through a set of task accomplishing behaviours. Each behaviour is codified as an action to be accomplished when a determinate situation is perceived. No symbolic reasoning is performed, and in fact the behaviours take the form of finite state machines in the implementation. Intelligent behaviour is achieved through the possibility of multiple behaviours firing simultaneously, although the various modules must be arranged into a subsumption hierarchy in order to avoid conflicts. Reactive architectures possess many desirable characteristics (most notably the simplicity of agent design), however they also presents problems: the lack of models of the environment using symbolic reasoning means that the perception stage can be difficult to treat, and the "emerging" nature of intelligence means that it is difficult to understand, replicate and apply.

Belief-desire-intention (BDI) architectures have their roots in the philosophical tradition of understanding practical reasoning, or the process of deciding, moment by moment, which action to perform in the furtherance of our goals. Practical reasoning involves two important processes: deciding what goals we want to achieve, and how we are going to achieve these goals. The former process is known as *deliberation*, the latter as *means-ends reasoning*. In (Georgeff and Lansky, 1987), practical reasoning in BDI agents is schematized as in Figure 2.1. There are seven main components:

- a set of current *beliefs*, representing information the agent has about its current environment
- a *belief revision function*, which takes a perceptual input and the agent's current beliefs, and on the basis of these, determines a new set of beliefs
- an option generation function, which determines the options available to the agent (its *desires*), on the basis of its current beliefs about its environment and its current intentions
- a set of current options, representing possible courses of actions available to the agent
- a filter function, which represents the agent's deliberation process, and which determines the agent's *intentions* on the basis of its current beliefs, desires, and intentions
- a set of current intentions, representing the agent's current focus
- an action selection function, which determines an *action* to perform on the basis of current intentions

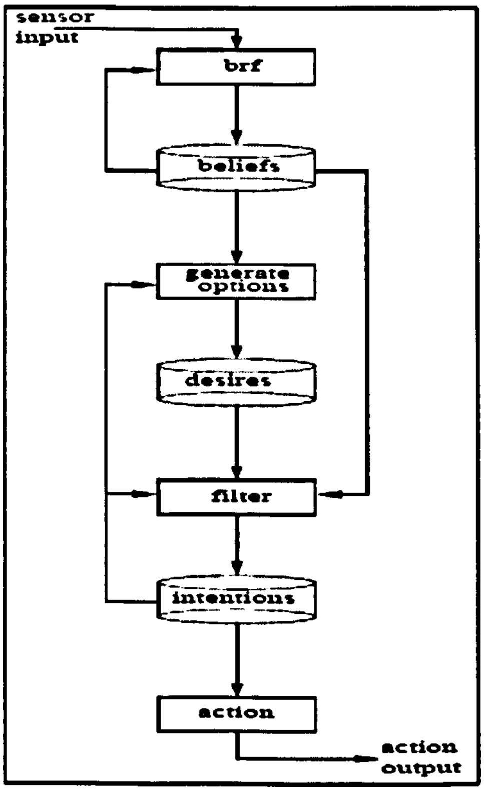


Figure 2.1. The BDI agent loop.

The BDI model is intuitive and gives clear functional decomposition of the system to be built, however it is difficult to balance between the pro-active and reactive behaviours, and there is no clear guideline on how to structure a system as a BDI architecture agent.

In Layered architectures the various subsystems are arranged into a hierarchy of interacting layers. This enables the IA to be injected with a variety of complementary behaviours, such as uniting reactive capabilities to more deliberate ones. For example, a layered architecture might include a BDI agent layer and a reactive layer, respectively providing deliberative and reactive capabilities. Two types of control flow can be identified within layered architectures:

- Horizontal layering, where the software layers are each directly connected to the sensory input and action output (in effect, each layer itself acts like an agent, producing suggestions as to what action to perform)
- Vertical layering, where sensory input and action output are each dealt with by at most one layer each

Horizontal and Vertical layering present a trade-off between flexibility and conflict-prevention: Horizontal is more flexible, but can incur in conflicts, whereas Vertical eliminates conflicts at the cost of flexibility. Significant layered architectures are the horizontally-layered TouringMachines (Ferguson, 1992) and the vertically-layered InterRap (Muller et al, 1995). Layered architectures are capable of providing both reactive and pro-active behaviour, but are lack the conceptual and semantic clarity of other approaches (in particular if compared to logic architectures).

Software systems developed from the concept of Intelligent Agent have been used in many occasions for the control of autonomous vehicles, but no "perfect" approach has yet been identified: ultimate autonomy means achieving capabilities similar to those of biological systems, especially humans, and an autonomous intelligent system will need to incorporate capabilities such as sensing, reasoning, action, learning, and collaboration. Future systems will most likely rely on combinations of computational intelligence methods, such as traditional control, fuzzy logic, neural networks, genetic algorithms, rule-based methods, or symbolic artificial intelligence (Long et al, 2007).

Many intelligent software architectures have been applied to provide autonomy to vehicles (not only aerial, but also ground, marine and space vehicles). We are now going to review some of the most significant experiments. Table 2.1 summarizes the main features of the presented systems. The systems are presented in order of relevance with respect to this PhD project.

Table 2.1. Features of reviewed Intelligent Systems (Reproduced from (Long et al., 2007)).

	Sub-Sumption	AuRA	SOAR	ACT-R	Fuzzy CLIPS	JESS	NIST RCS	PSU/ARL IC	FuzzyJ Toolkit	NASA ASE
Neural Network		✓?								
Genetic Algorithm		✓?								
Fuzzy Logic	✓				✓			✓	✓	
Symbolic AI		✓	✓	✓		✓				
Learning			✓	✓		✓				✓
Language	Lisp & Other	Lisp & Other	C & C++	Lisp	C	Java	C++	C++ & Ada	Java	C & Other
Reactive (R) or Deliberative (D)	R&D	R&D	D	D	D	D	R&D	R&D	D	R&D
Easy Sensor Input	✓	✓	✓		✓	✓	✓	✓	✓	
Freely Available			✓	✓	✓	✓	✓		✓	
Collab-oration			✓			✓	✓	✓		
Approx. Year	1986	1987	1987	1987	1994	1995	1997	1997	2001	2002

2.2.1 JESS

The Java Expert System Shell (JESS) is a rule-based engine written in the Java programming language. Using a forward-chaining Rete algorithm to process rules, JESS rules make changes to the working memory which is a collection of knowledge. Two notable applications of JESS are the Ubiquitous Robotic Companion (a network-based service robot that can use external sensors in the network and can also access a remote computer; Kim et al, 2005) and the PKD android (a humanoid intelligence architecture that can socialize with people by detecting faces and facial expressions and recognizing speech, through a control system composed of Jess and Natural Language Generation Functions; Hanson et al, 2005).

2.2.2 Subsumption architecture

The subsumption architecture introduced earlier in this section has been applied to mobile robots. Since there is no internal representation of the environment, single behaviour can be coded easily. Control layers, each of which represents a behaviour, are added on top of each other, giving the robot the capability to perform another, more complicated behaviour. The architecture has been applied to several robotic experiments, such as six-legged walking robots (Brooks, 1989) and small rovers for the exploration of Mars (Matijevic, 1998).

2.2.3 AuRA

The Autonomous Robotic Architecture (AuRA) is a hybrid architecture that combines deliberative and reactive components to use the advantages of both symbolic reasoning and reactive control (Arkin and Balch, 1997). The hierarchical deliberative components include a mission planner, spatial reasoner, and a plan sequencer. The reactive component is a schema controller that is responsible for controlling the robot behaviours at run-time using motor and perceptual schemas. The motor schemas define primitive robot behaviours such as avoiding static obstacles and moving ahead while the perceptual schemas use computer vision and ultrasound to sense obstacles to avoid. Planning is hierarchical, going from the bottom level plan sequencer to the spatial reasoner and then to the mission planner. Deliberation only occurs during execution if a problem with the previous plan occurs. The AuRA architecture and its components have been used in several robotic applications (not aerial vehicles, though).

2.2.4 ARL/PSU Intelligent Controller

The Applied Research Laboratory at the Pennsylvania State University (ARL/PSU) has developed a behaviour-based Intelligent Controller architecture (Stover and Kumar, 1999). The Intelligent Controller architecture was initially based on the subsumption approach, but was then modified to include symbolic and collaborative capabilities. The architecture has two main components: Perception and Response. The role of the Perception Module is to create an internal representation of the external world; the role of the Response Module is to create a plan of action to perform a specific mission using the situational awareness created by the Perception Module. The architecture has been applied to the control of UAVs and Unmanned Underwater Vehicles (UUVs). The UUVs have been demonstrated and are at a fairly high technology readiness level, but the work is mostly unpublished. UAV application is more recent; the UAVs are equipped with commercial autopilots that handle sensor signal processing and low level control. The architecture has been used to provide specific capabilities to UAVs, operating alone or in groups. For example, a team of two UAVs demonstrated formation flying capability during flight tests (Sinsley et al, 2008), using a combination of leader/follower behaviour patterns.

2.2.5 ASE

The Applied Sciencecraft Experiment (ASE) is a software package developed by the NASA Jet Propulsion Laboratory and designed to facilitate autonomous science. It is capable of dynamically planning and executing missions related to the capture and downlink of scientific data. The software is organized as a three level architecture (Chien et al, 2005). The top level is known as the Continuous Activity Scheduling Planning Execution and Response (CASPER) system. This system controls long term planning. It is responsible for scheduling activities that will facilitate the collection and downlink of scientific data, while obeying resource constraints (battery power, instrument constraints, processing power, available downlink bandwidth, etc.). It utilizes dynamic planning, which means that the mission plan is continuously updated as new data or goals (from users on the ground) are received. The middle layer in the architecture is known as the Spacecraft Command Language (SCL) layer. The role of this middle layer is to translate high level activities sent to it by the CASPER system to a detailed sequence of commands that must be executed in order to complete the activity. Commands from the ground in the form of SCL instructions can also be uploaded at this layer. At the lowest level is the flight software. This flight software provides low level control of the vehicle hardware. The most interesting application of this software has been onboard the Earth Observing One (EO-1) spacecraft. Since it is flying onboard a spacecraft, the software has to be extremely efficient and reliable.

2.2.6 NIST RCS

Reference model architectures, such as the NIST Real-time Control System (RCS) have been successfully used in intelligent systems (Albus, 1997). RCS focuses on intelligently controlling real machines in real world environments. The architecture is organized into a hierarchy of nodes determined by the decomposition of a system's mission into increasingly simple tasks. Each node has a combination of cognitive and reactive mechanisms, has a knowledge base, and is capable of sensory processing, world modelling, value judgment, and behaviour generation. The nodes at the lowest level of the hierarchy interact with sensors using their sensory processing modules and with actuators using their behaviour generation modules. Applications of RCS have focused on autonomous driving ground vehicles in real world environments and autonomous tactical behaviours for teams of unmanned military vehicles.

2.2.7 ACT-R

Cognitive architectures define the fixed processes that their designers believe are important for intelligent behaviour, and have been primarily used to understand and simulate human cognitive processes. They also have interesting potential for intelligent systems software, such as mobile robots and unmanned vehicles. ACT-R is a cognitive architecture which is implemented in the Lisp programming language. The architecture consists of goal, declarative memory, perception, and motor modules that are hypothesized to represent processes in specific brain regions (Anderson et al, 2004). A central production system uses production rules and a subset of information from each module to select a best production to fire for each reasoning cycle. Sub-symbolic processes, such as activation of declarative memory elements, base-level learning, and calculating the utility of each production rule, also have an important role in the ACT-R architecture. In addition to this sub-symbolic learning, ACT-R is also capable of learning new knowledge "chunks" as well as links between chunks and creating new productions from combinations of existing productions. ACT-R has been mainly used to develop cognitive models of human performance, but has also found application in robotics. An example is found in (Bugajska et al, 2002), where a hybrid controller is introduced; this is developed with reactive aspects for low-level calculations and cognitive aspects using ACT-R for high level processes such as reasoning.

2.2.8 JACK

The JACK Intelligent Agents framework has been built by the Australian company Agent Oriented Software (AOS). It is an implementation of the Belief-Desire-Intention model of agency, based on the Java language and adapted for development of multi-agent systems. From an engineering perspective, JACK is an expansion to the Java language (Howden et al, 2001). There are three main extensions:

- The first is a set of syntactical additions to its host language, such as keywords for the identification of the main components of an agent and sets of statements that allow control over the agents' state
- The second is a compiler that converts the syntactic additions described above into pure Java classes and statements that can be loaded with, and be called by, other Java code
- The third is a set of classes (called the kernel), that provides the required run-time support to the generated code, including concurrency management, default agent behaviour definition and communications for multi-agent applications

JACK agents have been used for applications such as decision support and human behaviour modelling for military simulations. But more relevant to the scope of this thesis is the application described in Section 1.3, where JACK agents were used for the control of an autonomous UAV (Karim et al, 2004; Lucas et al, 2004; Karim and Heinze, 2005; Ferry et al, 2007).

2.2.9 Soar

Soar is a cognitive architecture which was originally written in Lisp and has since been rewritten using C and C++. In a Soar model, domain knowledge is encoded as production rules and information about the current state is stored in working memory in attribute-value form (Newell, 1990). If a Soar agent does not have enough knowledge to select the best operator for each reasoning cycle, an impasse occurs in which Soar automatically creates a subgoal to determine the best operator. The learning mechanism in Soar (chunking) stores the results of the problem solving that is used to achieve the subgoal as a new production. Although the Soar architecture originally focused on internal problem solving and execution, its planning, execution, and learning processes have also been used to interact with dynamic real world environments. In particular, in the TacAir-Soar research project Soar agents were used to control unmanned vehicles in simulation environments (Jones et al, 1999). TacAir-Soar agents used 5200 production rules, 450 total operators, and 130 abstract operators to fly U.S. military fixed-wing aircraft in a dynamic nondeterministic simulation environment that was accessible only through simulated sensors. Included in the flights performed by these agents were collaborative missions that required interaction between agents through simulated radio systems.

2.3 The choice of Soar

As stated in Section 1.2, the goal of the present research project is the development of a software system capable of managing and controlling a UAV during the course of an entire mission. It is clear that such a system has to possess two main characteristics: autonomy and intelligence.

A clear definition of broad concepts such as autonomy and intelligence is difficult to achieve. The following definitions will be adopted:

- Autonomy is the capability possessed by systems that can operate in the real-world environment without any form of external control for extended periods of time (Bekey, 2005)
- Intelligence is a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience (Gottfredson, 1997)

It is to be noted that the concept of autonomy is defined here in its generic form. Autonomy as described in Section 1.1 is the application of such generic concepts to the very specific field of UAV control, leading to a more restricted and detailed definition.

In Section 2.2, a number of systems meant to possess autonomy and intelligence (at least to a certain degree) were reviewed. All of these systems revolve around the concept of Intelligent Agent, but the various implementations are very different, because of the fragmentation of research efforts regarding IAs. It is clear that, for the purposes of the present project, a choice had to be made regarding the architecture on which the system to be developed was to be based.

Some of the systems reviewed in (Long et al, 2007) are based on the use of cognitive architectures for the control of unmanned vehicles. This appears as an interesting option, since cognitive architectures can be expected to provide a high degree of intelligence and autonomy.

A cognitive architecture specifies the underlying infrastructure for an intelligent system (Langley et al, 2009). Briefly, an architecture includes those aspects of a cognitive agent that are constant over time and across different application domains. These typically include:

- the short-term and long-term memories that store content about the agent's beliefs, goals, and knowledge
- the representation of elements that are contained in these memories and their organization into larger-scale mental structures
- the functional processes that operate on these structures, including the performance mechanisms that utilize them and the learning mechanisms that alter them

In fact, cognitive architectures are more correctly indicated as computational cognitive models. They are broadly-scoped, domain-generic computational models of the human brain functionality, focusing on essential structures, mechanisms, and processes (Sun, 2009). They are used for broad, cross-domain analysis of cognition. A cognitive architecture provides a concrete framework for more detailed modelling of cognitive phenomena, by specifying essential structures, divisions of modules, relations among modules, and a variety of other essential aspects.

Notable examples of cognitive architectures include ACT-R, Soar, CLARION (Sun et al, 2001), ICARUS (Langley et al, 2004) and PRODIGY (Carbonell et al, 1990). However, the latter three are tailored for use in cognitive modelling and psychological studies, rather than autonomous control applications. This is mainly due to computational requirements: "heavy" architectures are unsuitable since an unmanned vehicle will always have limited on-board computational power, and this is all the more true for UAVs, because of the stringent weight limits typical of aircraft.

In (Long and Hanford, 2007), cognitive architectures are presented as an interesting opportunity for the development of autonomous unmanned systems. Furthermore, criteria for the evaluation of architectures are introduced. These criteria are quite different from the ones that are important for projects that use cognitive architectures for other purposes, such as modelling human behaviour. For unmanned vehicles, the applicability of the architecture to real-time systems is more important than features of the architecture that are important for its models to be able to predict data from psychological experiments. The evaluation criteria are:

- interfaceability with external environments, through a large number of sensors (inputs) and actuators (outputs)
- capability to show reactive, deliberative and reflective behaviours that overlap each other without conflicting
- collaboration ability, to be intended both as between agents (multiple agents for a single vehicle) and between vehicles (multiple vehicles, each represented by an agent, interacting within the same environment)
- possibility to integrate additional software components to the architecture (for example, a traditional autopilot for low-level control of a UAV)
- practical issues such as licensing costs, implementation programming language and support availability

An initial survey of available architectures and systems allowed restricting the decision between three possible choices: ACT-R, Soar and JACK. While JACK is not strictly a cognitive architecture, it possesses many of their typical characteristics, and has already been demonstrated as capable of supporting real-time operation.

Looking at the evaluation criteria, ACT-R presents problems regarding the interaction with external environments, and is also not promising in terms of inter-agent collaboration. Furthermore, it is implemented in the Lisp language, which for the purposes of this project is an inferior choice compared to C/C++. Soar scores well in all of these criteria, and was thus preferred over ACT-R.

From a functional point of view, a comparison between Soar and JACK based on the evaluation criteria does not yield a winner: although very different in nature, both are viable alternatives with demonstrated real-time operation capability. However, the JACK framework is developed by a company and is proprietary in nature. This not only involves licensing costs, but also introduces the problem of availability of source code: in particular, integration of additional software components can be expected to be more problematic compared to the open-source Soar architecture. Finally, JACK is implemented in the Java language, whose real-time capabilities are yet to be proven (C/C++ is widely used in real-time systems). Soar was thus preferred over JACK and chosen as the basic development platform for the system.

Summarizing, the Soar architecture was chosen as the base platform for the development of the UAV management and control capability because of the following characteristics:

- it is a cognitive modelling tool which enables the replication of human thought processes through the use of symbolic AI techniques, thus providing very high potential in developing intelligent capabilities
- it is proven to be capable to deal with very complex problem spaces while maintaining real-time operation
- it provides a good I/O interface, both between separate agents and with external components

- its core is written in the C++ language, which for aerospace applications is generally considered a better solution than Java, especially from the point of view of validation and verification processes (Jacklin et al, 2005)
- it is a fully open-source project

The system to be developed was then named Soar-based Autonomous Mission Management System, or SAMMS.

It has already been stated several times that the Soar architecture is capable of providing high-level reasoning capabilities. However, the architecture was developed for cognitive modelling rather than control applications, and presents very limited capabilities regarding specific domains which are useful for control applications. For example, using complex mathematics within Soar is not easy. In fact, such operations should be executed by external components that are interfaced with the Soar agent (or agents). Thus, for the development of SAMMS, it is imperative to build a framework where Soar agents can be interfaced seamlessly with external components.

It was our choice to develop this framework using the Matlab/Simulink software package. Details about this implementation strategy will be provided in Section 2.7, but here it is important to justify this choice. In fact, Matlab/Simulink is the most commonly used software packaged for model-based design of control systems, and provides very good capabilities in areas where Soar is weak (such as linear algebra and calculus). Most traditional control techniques (such as PID controllers, state-space models and nonlinear control) can be implemented using it, together with other techniques such as fuzzy-logic and neural networks.

Seamless interfacing with external components is not the only advantage of the integration of Soar agents in a Simulink framework. There are two other significant advantages: the possibility to develop an appropriate simulation environment for system validation and verification, and the availability of Real-Time Workshop, which allows the conversion of Simulink models into real-time executable code (a particularly interesting option for the rapid deployment of prototype control systems).

While a Simulink framework allows integration of multiple agents, it is important to underline the fact that within SAMMS multiple agents are used to provide intelligent and autonomous behaviour to a single UAV. Different agents are used to perform different functions, and in fact SAMMS will be based around a set of three Soar IAs which share very little in terms of structure and implementation. In general, the concept of IA could be naturally applied to the UAV field by implementing a multi-UAV system where each UAV is represented by an agent. While this is an interesting concept which could eventually be applied as a top layer for SAMMS, it is important to clarify that the system presented here instead uses multiple agents that control a single UAV (performing different functions).

2.4 Introduction to Soar

The Soar (State, Operator And Result) architecture was originally created in 1982, with the purpose of being a system capable of general intelligence. The mains goals were for Soar to be able to: work on the full range of tasks, from highly routine to extremely difficult open-ended problems; employ the full range of problem solving methods and representations required for these tasks; and learn about all aspects of the tasks and its performance on them (Laird et al, 1987).

The architecture has continued to evolve and, while the first implementations were only tested on classical AI problems (such as the eight puzzle, missionaries and cannibals, tic-tac-toe, magic squares and the water jug task), more recent implementations have been applied in real-world systems (as detailed in Section 2.5). The most recent version of Soar is Soar 9, however it must be noted that for the SAMMS project version 8.6.3 was used (Soar 9 was released after the beginning of the project and a version change was considered to be a risk).

To illustrate the relationship between the Soar architecture and SAMMS, it is possible to use an analogy. On a computer, hardware processing is necessary for software to operate (Lehman et al, 2006); similarly, for any complex system, we can make a distinction between the fixed set of mechanisms and structures of the architecture itself (the "hardware"), and the content those architectural mechanisms and structures process (the "software"). In these terms, the Soar architecture provides the "hardware" for an intelligent system and will be described thoroughly in this section. However, the novelty of SAMMS obviously resides in the "software" components.

2.4.1 Theoretical aspects

In common with the mainstream of problem solving and reasoning systems in AI, Soar has an explicit symbolic representation of its tasks, which it manipulates by symbolic processes. It encodes its knowledge of the task environment in symbolic structures and attempts to use this knowledge to guide its behaviour. It has a general scheme of goals and subgoals for representing what the system wants to achieve, and for controlling its behaviour. Beyond these basic commonalities, Soar embodies mechanisms and organizational principles that express distinctive hypotheses about the nature of the architecture for intelligence (Laird et al, 1987).

In Soar, every task of attaining a goal is formulated as finding a desired state in a problem space (a space with a set of operators that apply to a current state to yield a new state). Hence, all tasks take the form of heuristic search (Newell, 1980). Routine procedures arise, in this scheme, when enough knowledge is available to provide complete search control (e.g. to determine the correct operator to be taken at each step). The adoption of the problem space as the fundamental organization for all goal-oriented symbolic activity is a principal feature of Soar. Problem space search occurs in the attempt to attain a goal. Whenever a new goal is encountered in solving a problem, the problem solver begins at some initial state in the new problem space. The problem space search results from the problem solver's application of operators in an attempt to find a way of moving from its initial state to one of its desired states.

Another key concept of Soar is subgoaling. For any problematic decision, a subgoal can be set up. Since setting up a goal means that a search can be conducted for whatever information is needed to make the decision, Soar can be described as having no fixed bodies of knowledge to make any decision. The ability to search in subgoals also implies that further subgoals can be set up within existing subgoals so that the behaviour of Soar involves a tree of subgoals and problem spaces.

In Soar, there is only a single memory organization for all long-term knowledge, called the production system (McDermott and Forgy, 1978). Productions deliver control knowledge, for example when a production action rejects an operator that leads back to the prior position. Productions also provide procedural knowledge for simple operators. The data structures examinable by productions (the knowledge in declarative form) are all in the production system's short-term working memory. However, the long-term storage of this knowledge is in productions which have actions that generate the data structures. Within Soar, all satisfied productions are fired in parallel, without conflict resolution. Productions can add data elements to working memory, but modification and removal of data elements is influenced by the architecture, that places constraint on how an operator can affect working memory.

Search control knowledge is brought to bear by the additive accumulation (via production firings) of data elements in working memory. One type of data element, the preference, represents knowledge about how Soar should behave in its current situation (as defined by a current goal, problem space, state and operator). The preferences admit only a few concepts: acceptability, rejection, better (best, worse and worst), and indifferent. The architecture contains a fixed decision procedure for interpreting the set of accumulated preferences to determine the next action (Soar Technology Inc, 2002).

When problem solving cannot continue, an impasse has been reached. The architecture can detect impasses and automatically creates a goal for overcoming the impasse. This feature is called automatic subgoaling, and is one of the main features of Soar. The architecture continuously monitors for the termination of active goals in the goal hierarchy, and all working-memory elements local to the terminated goals are automatically removed.

Soar learns continuously by automatically and permanently caching the results of its subgoals as productions. This mechanism is called chunking, and is a limited form of practice learning. However, when combined with the problem-solving and automatic subgoaling features, it can produce significant results.

2.4.2 Architecture description

Figure 2.2 shows a diagram of the Soar architecture. It is possible to notice the working memory that holds the complete processing state for problem solving in Soar. This has three components: a *context stack* that specifies the hierarchy of active goals, problem spaces, states and operators; *objects*, such as goals and states (and their sub-objects); and *preferences* that encode the procedural search-control knowledge. The processing structure has two parts. One is the production memory, which is a set of productions that can examine any part of working memory, add new objects and preferences, and augment existing objects, but cannot modify the context stack. The second is a fixed decision procedure that examines the preferences and the context stack, and changes the context stack. The productions and the decision procedure combine to implement the search-control functions. Two other fixed mechanisms are shown in the figure: a working memory manager that changes, updates and deletes elements from working memory, and a chunking mechanism that adds new productions.

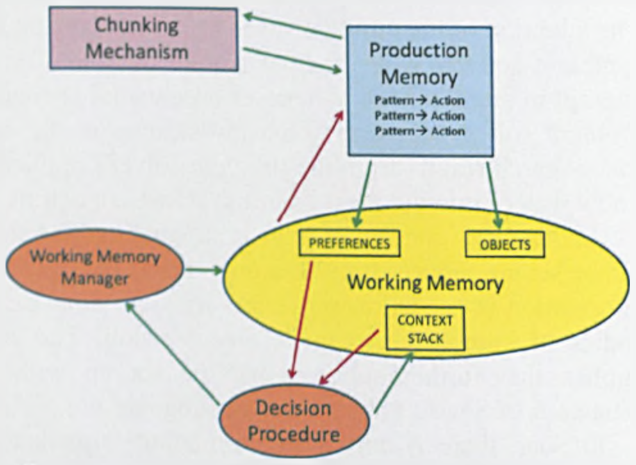


Figure 2.2. Soar architecture concepts (Laird et al., 1987).

Working memory consists of a context stack, a set of objects linked to the context stack, and preferences. The *context stack* contains the hierarchy of active contexts. Each context contains four slots, one for each of the different roles: goal, problem space, state and operator. Each slot can be occupied either by an object or by the symbol undecided, the latter meaning that no object has been selected for that slot. The object playing the role of the goal in a context is the current goal for that context; the object playing the role of the problem space is the current problem space for that context and so on. The top context contains the highest goal in the hierarchy. The goal in each context below the top context is a subgoal of the context above it. The *object* is the basic entity in working memory. An object, such as a goal or a state, consists of a symbol, called its identifier, and a set of augmentations. An augmentation is a labelled relation (the attribute) between the object (the identifier) and another symbol (the value), thus defining an identifier-attribute-value triple. An object may have any number of augmentations, and the set of augmentations may change over time. A *preference* is a more complex data structure with a specific collection of eight architecturally defined relations between objects. Working memory can be thought of as a set of objects organized into a tree hierarchy.

The processing structure implements the functions required for search in a problem space, using task-implementation knowledge to generate objects and search-control knowledge to select between alternative objects. The search-control functions are all realized by a single generic control act: the replacement of an object in a slot by another object from the working memory. The representation of a problem is changed by replacing the current problem space with a new problem space. Returning to a prior state is accomplished by replacing the current state with a pre-existing one in working memory. An operator is selected by replacing the current operator (often undecided) with the new one. A step in the problem space occurs when the current operator is applied to the current state to produce a new state, which is then selected to replace the current state in the context. A new state can replace the state in any of the contexts in the stack, not just the lowest or most immediate context but any higher one as well. When an object in a slot is replaced, all of the slots below it in the context are reinitialized to undecided. Each lower slot depends on the values of the higher slots for its validity: a problem space is set up in response to a goal; a state functions only as part of a problem space; and an operator is to be applied to a state.

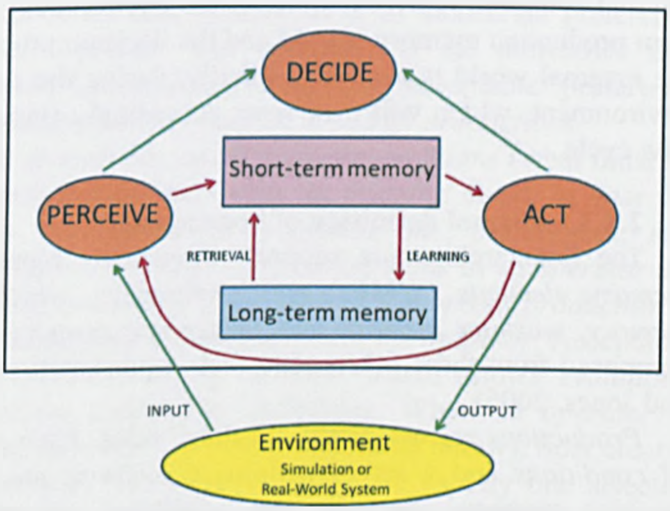


Figure 2.3. Perceive-Decide-Act cycle.

The replacement of context objects is driven by the decision cycle. Each cycle involves two distinct parts. First, during the *elaboration* phase, new objects, new augmentations of old objects and preferences are added to working memory. Then the *decision procedure* examines the accumulated preferences and the context stack, and either it replaces an existing object in some slot, or it creates a subgoal in response to an impasse. Decisions are made on the base of preferences. Three main types of preferences are available: acceptable (a choice is to be considered), reject (a choice is not to be made), desirable (a choice is better/worse/indifferent than a reference choice). Together, the acceptability and rejection preferences determine the objects from which a selection will be made, and the desirability preferences partially order these objects. If the decision procedure determines that the value of the slot should be changed, the change is applied, all of the lower slots are reinitialized to undecided, and the elaboration phase of the next decision cycle ensues. If the current choice is maintained, then the decision procedure considers the next slot lower in the hierarchy. If either there is no final choice, or all of the slots have been exhausted, then the decision procedure fails and an impasse occurs.

In Soar, four impasse situations are distinguished:

- *Tie*; this impasse arises when there are multiple choices that are not mutually indifferent and do not conflict; these are competitors for the same slot for which insufficient knowledge (expressed as preferences) exists to discriminate among them
- *Conflict*; this impasse arises when there are conflicting choices in the set of available operators
- *No-change*; this impasse arises when the current value of every slot is maintained (the applied operator has produced no significant effect)
- *Rejection*; this impasse arises when the current choice is rejected and there are no other choices

These four conditions are mutually exclusive, so at most one impasse will arise from executing the decision procedure. The response to an impasse in Soar is to set up a subgoal in which the impasse can be resolved.

From an external point of view, Soar implements a Perceive-Decide-Act cycle (Laird and Rosenbloom, 1990) to sample the current state of the world, make knowledge-rich decisions in the service of explicit goals, and perform goal-directed actions to change the world in intelligent ways. The cycle is depicted in Figure 2.3. During the Perceive phase, input is received from the environment, inserted in the short-term working memory, and eventually processed during the elaboration phase (to create internal symbolic representations of the environment). During the Decide phase, long-term production memory is used and the decision procedure is applied until an output to the external world is available. Finally, during the Act phase the output is sent to the environment, which will then react accordingly (input should change), thus starting a new cycle.

2.4.3 Formal definition of Soar agents

The Soar architecture supports three basic representations: *productions*, *working memory elements* (WMEs) and *preferences*, which are represented in *production memory*, *working memory* and *preference memory* respectively. Soar operators are composed from these others, and their representation spans the three memories (Wray and Jones, 2005).

Productions are essentially "if-then" rules. Each production is specified by a series of *conditions* and a set of *actions*. *Conditions* are matched against the contents of working memory, and, when all conditions are satisfied, the *actions* are executed, usually specifying changes to elements in the *working memory*. Because Soar expresses the conditions and actions of productions in a form of predicate logic, rather than the propositional logic used in procedural programming languages, the match process can generate multiple *instantiations* of the production, with variable bindings specific to each match. Thus, two (or more) instances of the same production can fire simultaneously.

Soar asserts and maintains active memory objects in *working memory*; as the objects expressed in working memory change, they trigger new productions, resulting in further changes to the memory. Soar's working memory is highly structured: it is a directed graph, with each element described by a triple (*identifier*, *attribute*, *value*). Complex objects can be composed by making the value of an object another *identifier*. Working memory is also segmented into *state partitions*. Soar assigns each *problem space* created in response to an *impasse* a distinct *state object*; each *state partition* encapsulates the search for additional knowledge to resolve that state's *impasse*. Every state includes a pointer to its *super-state*, so that the reasoning within the state can access relevant context from the problem space in which the impasse occurred. Because the *top-state* is the root of the *working memory* graph, and each state is the root of some sub-graph, every element in memory can be traced to a state (and possibly a chain of states). The "lowest" such state in a chain is the object's "local state".

Soar's *top-state* also includes an *input/output* partition, which is divided into *input-link* and *output-link* objects. Individual input and output objects are represented in the same representation language as other objects. However, *input* objects are placed on the *input-link* by an "input function" that transforms environmental data into the (identifier, attribute, value) representation required by Soar. Similarly, an *output* function interprets objects on the *output-link* as commands and attempts to execute them.

The *preference* data structure expresses preferences between candidate *operators* competing for selection. Unary preferences (such as "acceptable" and "best") express preferences about a single candidate; binary preferences (such as "better" and "worse") compare one operator to another. When it is time for Soar to select an operator, a preference semantics procedure interprets all the preferences to determine if a unique option can be identified. If no unique choice can be made, Soar generates an *impasse*; the impasse type is indicated by the problem in the preferences. Soar stores preferences in a *preference memory*, which is impenetrable to *productions*. That is, productions cannot test whether one operator is better than another, or if an indifferent preference has been asserted for a particular operator. The exception is the preference that represents whether an operator should be considered at all. This "acceptable" preference is represented in Soar's *working memory* and thus can be tested by *productions*.

Soar *operators* represent small procedures, specifying *preconditions* (what must be true for the operator to be activated) and *actions* (what the operator does). In Soar the representation of an *operator* is distributed across *productions*, *preferences*, and *working memory elements* within the architecture. The *preconditions* of an *operator* are expressed in one or more *proposal productions*. The *action* of a *proposal production* is to assert into preference memory the *acceptable preference* for the *operator*. Acceptable preferences are also represented on the *working memory*, which allows *evaluation productions* to compare and evaluate acceptable candidates. When an *operator* is selected (during the execution of the *decision procedure*, described below), Soar creates an *operator object* in *working memory*. Soar allows each *state* exactly one selected *operator* at any time. Therefore, attempting to create zero or multiple operator objects will result in an *impasse* for that state's *problem space*. Once the selected *operator* is represented in *working memory*, it can trigger *productions* that produce the post-conditions of the operator, resulting in *operator application*.

Soar's general processing loop (its *decision cycle*) is basically a *perceive-decide-act* loop. Individual components of the Soar decision cycle are termed *phases*. During the *input phase*, Soar invokes the *input function* (as described above), communicating any changes indicated by the environment to the agent through the *input-link* portion of *working memory*. In the *output phase*, the agent invokes the *output function*, which examines the *output-link* and executes any new commands indicated there. Feedback about the execution of commands is then provided during the *input phase*.

Within Soar, reasoning is focused on the selection (and application) of *operators*. Each Soar decision consists of three phases within the "decide" portion of the *perceive-decide-act* loop. During the *elaboration phase*, the agent iteratively fires any *productions* other than operator applications that match against the current *state*, including new input. This process includes "elaborating" the current state with any derived features, proposing new operators, and asserting any preferences that evaluate or compare proposed operators. This phase uses Soar's truth maintenance system to compute all available logical entailments (i.e., those provided by specific productions) of the assertions in working memory. When no further *elaboration productions* are ready to fire, the *decision cycle* is said to have reached *quiescence*. At this point, the elaboration process is guaranteed to have computed the complete entailment of the current state; any immediately knowledge applicable to the proposal and comparison of operators will have been asserted. At *quiescence*, Soar enters the *decision phase* and invokes the preference semantics procedure to sort and interpret *preferences* for *operator selection*. If a single *operator* choice is indicated, Soar adds the *operator object* to memory and enters the *application phase*. In this phase, any *operator application productions* fire, resulting in further changes to the state, including the creation of *output commands*. Application productions are similar to elaboration productions with two exceptions: their *conditions* must include a test for the existence of a *selected operator*, and any changes they make to *working memory* are *persistent*. Persistent objects do not get retracted automatically by Soar's truth maintenance system but must be deliberately removed by another *operator*. If there is not a unique choice for an *operator*, Soar creates a new *state* object in response to the *impasse*, so that the agent can undertake a deliberate search for knowledge that will resolve the impasse and thus enable further progress in the original *state*.

A Soar agent will always cycle between the phases of the decision cycle in this order: input phase, elaboration phase, decision phase, application phase and output phase. Within the decision cycle, Soar implements and integrates a number of influential ideas and algorithms from artificial intelligence. These include:

- *pattern-directed control*; Soar automatically considers any knowledge relevant to the current problem using a Rete algorithm to match patterns within the production system
- *reason maintenance*; Soar uses computationally inexpensive reason maintenance algorithms to update its beliefs about the world. The justification-based truth maintenance system ensures that any non-persistent changes made to working memory remain in memory only as long as the production that added the object continues to match, thus ensuring that agents are responsive to their environments
- *preference-based deliberation*; Soar balances automatic reason maintenance within the decision cycle with the deliberate selection of operators. Assertions that result from deliberation (i.e., operator applications) persist independently of reason maintenance
- *automatic sub-goaling and task decomposition*; in some cases, an agent may find it has no available options or has conflicting information about its options. Soar responds to these impasses and automatically creates a new problem space, in which the desired goal is to resolve the impasse

To define a Soar agent, a user must define two separate things: a *set of productions* (including operation proposal and application productions, elaboration productions and preference productions) and a set of *input* and *output functions* which are necessary to interface the agent with its environment.

2.4.4 Technical implementation

The Soar architecture was originally coded in the Lisp language, but has since been converted to C++ and, recently, Java. In SAMMS, Soar Version 8.6.3 is used; this is only available in the C++ version, which is also the desired language.

Technically speaking, the Soar kernel resides into a set of dynamic C++ libraries (*dll* files). A Soar agent is implemented through a C++ class; it has to be linked to the external world through appropriate I/O functions, and must be assigned a set of production rules.

During implementation of a Soar-based system, the user must then develop both the I/O functions and the production rules, while the kernel is usually left unchanged (the kernel source code is available, but the need to change the kernel would arise from architectural concerns rather than implementation issues). I/O functions will usually be coded in the same language as the kernel (C++, in our case), while productions are normally coded as text-files.

In fact, a production is an *if-then* statement coded in the Soar language. A production set might be created using a simple text editor, however a more advanced editor is available. The VisualSoar editor is not a full Integrated Development Environment (IDE) for Soar, but is instrumental in the coding of the production set due to many features which ease the process of writing and organizing the productions.

Another very important component of the Soar package is the Soar debugger. This allows to monitor Soar agents as they execute, so that at any execution cycle it is possible to explore the entire working memory, see the operators that are being proposed, and observe the decision process as it is carried out.

2.5 Soar Applications

The Soar architecture has been used in several non-related research projects, ranging from behavioural simulation of fighter pilots to the control of mobile robots. In this section, relevant experiments will be reviewed, and their significance for the SAMMS project highlighted.

2.5.1 Robo-Soar

Robo-Soar is an early project undertaken by the creators of the Soar architecture. It consists of a Soar agent controlling a Puma robotic arm using a camera vision system (Laird and Rosenbloom, 1990). The vision system provides position and orientation of the blocks in the robot's work area. Also, the vision system can detect a "trouble" light; when this is switched on, the robot must immediately press a button. Although the task is relatively simple, the environment is unpredictable: the light can go on at any time, outside agents can move the blocks (helping or hindering Robo-Soar) and some blocks may not be visible because of the presence of the arm.

One possible goal for Robo-Soar is to align the blocks in the work area. A subgoal is then to align a pair of blocks. Within a goal, the first decision is the selection of a problem space. The problem space determines the set of operators that are available in a goal. In Robo-Soar, the problem space for manipulating the arm consists of operators such as *open-gripper* and *move-gripper*. The second decision selects the initial state of the problem space. For goals requiring interaction with an external environment, the states include data from the system sensors, as well as internally computed elaborations of this data. In Robo-Soar, the states include the position and orientation of all visible blocks and the gripper, their relative positions, and hypotheses about the positions of occluded blocks. Once the initial state is selected, decisions are made to select operators, one after another, until the goal is achieved.

In Soar, operator selection is the basic control act for which planning can provide additional knowledge. Planning in Robo-Soar is performed by creating an internal model of the environment and then evaluating the result of applying alternative operators using available domain knowledge. Soar invokes planning whenever knowledge is insufficient for making a decision and it terminates planning as soon as sufficient knowledge is found, thus planning is always in service of execution. Soar productions are continually matched against all of working memory, including incoming sensor data, and all goals and subgoals. When a change is detected, planning can be revised or abandoned if necessary. Thus, a Soar system can exhibit both pro-active and reactive behaviours: Robo-Soar plans pro-actively when moving the blocks, but behaves reactively to respond to the "trouble" light.

Robo-Soar is an early and relatively simple application, however it demonstrates the ability of Soar agents to interface with and control robotic systems, even with the limited hardware available at the time.

2.5.2 Hero-Soar

Hero-Soar is another system developed by the creators of the Soar architecture. A Soar agent is used to control a Hero 2000 robot (Laird and Rosenbloom, 1990). The Hero 2000 is a mobile robot with an arm for picking up objects and sonar sensors for detecting objects in the environment. In the experiment, Hero-Soar's task is to pick up cups and deposit them in a waste basket.

The basic motor commands for the robot include positioning the various parts of the arm, opening and closing the gripper, orienting sonar sensors, and moving and turning the robot. However, these are low-level commands and it is useful to take advantage of the hierarchical planning capabilities inherent to Soar. A high-level set of commands includes operators such as *search-for-object*, *centre-object*, *pickup-cup*, and *drop-cup*. The execution of each of these operators involves a combination of more primitive operators that can only be determined at run-time.

Hero-Soar is a more complex application compared to Robo-Soar, introducing hierarchical planning. An interesting observation that could be made is that the execution time for the internal cycle of a Soar agent does not increase much with the complexity of the system. This means that Soar is well suited to the modelling of very complex systems, since it can maintain a similar level of performance regardless of the complexity of the system. However, Soar has inherent lags during execution, which make it unsuitable for high-frequency systems. It is possible to state that a Soar agent can be useful for modelling complex behaviour at a timescale of nearly 1 s, but not for systems reacting on timescales of 10 ms or less (simple systems working at this frequency will be better controlled by a normal PID controller).

2.5.3 TacAir-Soar

TacAir-Soar is an intelligent, rule-based system that generates believable humanlike behaviour for large-scale distributed military simulations (Jones et al, 1999). The innovation of the application is primarily a matter of scale and integration. The system is capable of executing most of the airborne missions that the U.S. military flies in fixed-wing aircraft. It accomplishes its missions by integrating a wide variety of intelligent capabilities, including real-time hierarchical execution of complex goals and plans, communication and coordination with humans and simulated entities, maintenance of situational awareness, and the ability to accept and respond to new orders while in flight.

TacAir-Soar consists of more than 5200 production rules, that allow it not only to make flight-related decisions, but also to fully integrate within a Command-and-Control structure (for example, by sending reports after a mission). The environment is a real-time large-scale simulation of a battlefield. TacAir-Soar participates in the simulation environment using the Soar-MODSAF Interface (SMI) to translate simulated sensor and vehicle information into the internal symbolic representation and Soar actions into MODSAF function calls.

TacAir-Soar is built to simulate human behaviour. The intention is to model the tactical decision-making of a fighter pilot within a battlefield scenario. Furthermore, the simulated pilot is to be able to interact appropriately with other participants in the simulation. To limit the cost, various efficiency improvements were made: the Soar architecture was rewritten in C++ (from Lisp) during this project, yielding large improvements in computational times. Sensor data processing was also addressed: it became evident that at least part of this processing is more efficiently achieved outside of Soar.

TacAir-Soar was used in two major exercises: STOW '97 and Roadrunner '98. It is deployed at the WISSARD Laboratory on the Oceana Naval Air Station at Virginia Beach, Virginia, where it is used for continued testing of the underlying network infrastructure as well as for evaluation and demonstration of the technology. There is also a deployed installation at the Air Force Research Laboratory in Mesa, Arizona, with plans for the system's use in future training exercises and technology demonstrations.

Relating it to SAMMS, TacAir-Soar has demonstrated the feasibility of real-time high-level control of aircraft using Soar agents. However, this is a system which has a very different focus: while the tasks might look similar, the intention in TacAir-Soar is to simulate human behaviour, whereas the intention for SAMMS is to achieve actual control of a vehicle. Furthermore, TacAir-Soar is used to control planes that are normally manned and are thus designed very differently from UAVs.

2.5.4 OOTW Simulation

In an application similar to TacAir-Soar, Soar agents have been used to provide training facilities for commanders in Operations-Other-Than-War (OOTW) scenarios (Kalus and Hirst, 1998). OOTW has come to mean many things including: peacekeeping and peace enforcement operations, humanitarian aid, disaster relief, and non-combatant evacuation. In these situations, it is difficult to make certain decisions. In particular, doctrine and rules of engagement (RoE) can be very complex areas to decide upon in OOTW, and this type of training can help in these areas, by running "what-if" scenarios and ironing out possible differences of interpretation between commanders engaged in these operations.

The kind of problems involved in such a simulation is very complex and multi-faceted in nature. The knowledge to be embedded in the Soar production rules is not readily available, but has to be obtained from Subject Matter Experts, which is a long and difficult process. In the development of SAMMS, such knowledge might be needed, so this experiment can be useful in providing the means for quickening the process of knowledge acquisition.

2.5.5 Blue Bear Systems Research

Blue Bear Systems Research (BBSR) is a British company that develops UAVs and related control hardware and software. Recognizing autonomy as one of the main research areas for UAVs, and on the assumption that most research is focusing on software and algorithms, they focused their research on the development of hardware optimized to execute the algorithms (Smith and Willcox, 2005). With the development of practical approaches in achieving autonomy, their work has tried to determine what tools might be required to provide it, and what the implementation might be in a systems engineering context.

BBSR concentrated work on the development of multi-agent systems based on the Soar architecture, in order to provide to an autonomous UAV functionality such as situational awareness, combat tactics, team-working or optimal route planning. In particular, the work investigated hardware solutions suitable for the implementation of distributed agents in small UAVs. Small UAVs can carry a limited amount of computational power, but this has to be able to provide the entire set of autonomous functionality. Using techniques from computer science, a dedicated hardware platform was developed combining clusters of low-cost X-scale and PC based processors and Floating Point Gate Arrays (FPGA). As a demonstration, a complex multi-agent system designed to search a network of roads was developed.

The kind of research accomplished by BBSR is very similar in nature to the declared purposes of SAMMS, even if it takes a much more hardware-focused approach. However, results are largely undisclosed due to the commercial nature of the project.

2.5.6 HexCrawler

The Soar architecture was used at the Department of Aerospace Engineering of Penn State University to control a hexapod. The base platform for this research is a commercially available six-legged robot called HexCrawler. It provides the entire actuator system and an on-board computer based on a Mini-ITX motherboard (Janrathitikarn and Long, 2008). A complete set of sensors was integrated with the robot and its on-board computer, including weight-on-wheel sensors (for the two front legs), sonar sensors, an electronic compass, a GPS module and a camera.

A set of Soar productions was developed to control the robot's behaviour at high level, implementing behaviours such as "avoid obstacle" and "go to GPS position". Experiments were performed with three different types of obstacles to test whether the system could control the hexapod to navigate to a GPS location while avoiding obstacles. The system was successful during the first two tests, where navigation was almost linear. In the third test, the presence of a cul-de-sac in the robot's path prevented success (Hanford et al, 2008). Possible improvements to the system include the addition of more advanced detection and planning capabilities, and might lead to a system capable of passing the third test.

While the focus of this research is on a ground vehicle, the system is conceptually very similar to SAMMS. In both cases, Soar agents are used to provide reasoning at the top level, with middle-level software and low-level hardware providing basic functionality. However, in SAMMS the approach is to first develop the Soar functionality and then deploy it on a test system. The approach is the opposite in this system, and consequentially the Soar agents are not developed enough. This project is an excellent demonstration of the feasibility of the entire concept of controlling autonomous vehicles through Soar agents.

2.6 IA Verification

Intelligent Agents are a relatively new concept in software engineering. They have found extensive application in some applications; for example agent software is widely used in the World Wide Web (WWW). However, the technology is still not mature and there is a distinct lack of standards and best practices (or even of well-established agent programming languages). This does not affect severely low-risk applications such as the internet, but is a big issue for industrial applications. The aerospace industry in particular is naturally very conservative regarding the adoption of new technology, although it is very fast in adopting it once the technology is proven to be safe and to bring substantial advantages. The advantages of using IAs are still to be demonstrated, and safety concerns are exacerbated by the lack of standards.

The current lack of legislation regarding UAVs means that there is no way to know what exactly will be required in the future for UAVs to operate in civil airspace. However, certainly the control software will need to be certified in some way. Since IAs are a new technology, a UAV under the control of IAs will need a consistent verification process.

In current software engineering practices, Validation and Verification (V&V) activity is performed using different techniques and methodologies, which have been defined over the last two-three decades following the increasing use of complex electronics in safety-critical applications. Such methodologies are now trusted to be effective in determining if a software system is compliant with its requirements.

From a theoretical point of view, V&V techniques can be divided into two broad categories: formal methods and model-checking methods (Wooldridge and Ciancarini, 2001). Formal methods involve the complete logical modelling of the software; starting from the assumption that every software system can be described using logic, the V&V activity consists in deriving a mathematical proof that the system meets the requirements. This technique is theoretically very sound, but it is impractical, as complexity of the software systems will lead to exponentially increasing complexity of the logical description.

Model-checking techniques take instead another approach, which basically involves modelling the software system by stating assumptions that reduce its complexity, in order to obtain a model which is simpler to validate. It is however important to understand that, with this methodology, it is not possible to test the response of software systems to every possible combination of events, since the assumptions made will reduce the number of variables that affect the system. A key point for the success of this type of technique is obviously making correct assumptions that do not leave out of the V&V process important factors that will influence the real system.

If IAs are ever to be used on commercial products, it is going to be imperative to establish the possibility to certify them according to the common certification rules such as the DO-178B/C (RTCA, 1992). Moreover, aviation experts have to be convinced of the reliability of such software systems. While V&V processes for "conventional" software are now well-accepted and trusted, there is concern as about whether the same techniques can be used for the V&V of Intelligent Agents.

The use of completely new techniques is not acceptable, especially since new techniques will likely be rejected as valid methodologies due to the sheer difficulty to understand them for people without experience on IAs. Rather than developing completely new V&V techniques, it is then important to establish appropriate solutions and expedients that will both allow and give confidence in the application of conventional techniques to IAs. This means analyzing how IAs have to be treated differently from conventional software, in terms of both technical issues and "external perception".

It is important to state an important concept: Intelligent Agents are not a completely new technology, but are instead a type of software that is based on specific theoretical paradigms that impact heavily on high-level functionality but do not affect low-level functionality. With the terms high-level functionality and low-level functionality, we indicate respectively the set of more "abstract" functions such as ahead-planning and decision-making (high-level) and the set of basic functions such as I/O management and process flow (low-level).

This means that low-level functionality of an IA can be verified using the same techniques that are used with conventional software, such as code analysis, stress testing and Hardware-In-the-Loop simulations. Therefore, V&V of low-level functionality does not represent a problem.

On the contrary, high-level functionality raises several issues. While the ability to operate at different levels of abstraction is one of the major advantages of IA systems such as Soar, it is yet unclear how to deal with the increased complexity deriving from this when looking at V&V activities. In fact, it is clear that IAs are often perceived as undermining the determinism of a software system (Jonker and Treur, 1998). While this is not true, since IAs are still a piece of software which is going to behave deterministically if coded correctly, it cannot be denied that certain IA systems are so complex that explaining their behaviour and demonstrating their determinism can be difficult.

It can be stated that the main issue in for the V&V of IAs is complexity (Fisher and Wooldridge, 1997). Complexity takes two forms: IAs that are highly complex in nature due to the introduction of several levels of abstraction within their reasoning processes, as opposed to multi-agent systems where the complexity is due to the unclear interaction of large numbers of relatively simple IAs.

Usually, the main issue with such systems is understanding the internal behaviour of the agent (Councill et al, 2003). Due to their complexity, monitoring the responses to specific inputs is usually not sufficient to verify functionality. In particular, while unexpected output will obviously lead to the identification of a problem, correct output does not give the guarantee that the system is behaving correctly. The typical case of this problem is the one in which the system gives the correct response to an input, but for the wrong reason. It is clear that the only way to solve this issue is to be able to follow the decision process of the IA.

In some cases, verification of the high-level functionality of IAs is achieved using formal logic verification. As previously stated, these methodologies are unpractical when the complexity of the agents increases. Very complex agents or multi-agent systems using large numbers of agents require different techniques.

The Soar architecture involves a certain level of complexity by its sole use, however it has to be noted that the architecture is widely used in a variety of applications, which can provide a solid background on which to base the V&V activity for the architecture itself. The rules defining our agents instead obviously need to be verified, and this will be achieved using model-checking techniques. In fact, the V&V process for SAMMS will rely on simulations that will test the agents in a variety of situations. It is important to develop an appropriate set of test scenarios; since it is virtually impossible to cover all possible combinations, these scenarios will have to be representative of all the possible situations which might affect a UAV's mission. The scenarios that have been created will be described in detail in Chapter 4.

The problem of verification for highly complex IAs originates mainly from the difficulty of explaining the agent's behaviour. In many cases, the difficulty lies in collecting sufficient data to prove that an IA-based system made a specific decision due a specific reason. This can lead to the misinterpretation of external results for the IA system, or more specifically the possibility that a flaw in the decision process for the agent or multi-agent system is not discovered because it does not influence the external output which is normally the focus of V&V activities.

Such cases in which IAs provide "the correct answer for the wrong reason" have attracted the attention of several researchers, since such software problems can be extremely difficult to find. Clearly, it is necessary to overcome such problems, as they can ultimately lead to unexpected and non-deterministic behaviour. As stated before, the main problem is the system's complexity and the difficulty of filtering out irrelevant information while highlighting the data which can explain an agent or multi-agent system's behaviour.

An interesting approach to solve this problem has been used in at least two different research programs; in (Wong, 1997), the focus is on IAs in general; in (Taylor et al, 2006), the authors deal with the explanation of the behaviour of Soar agents. The approach basically involves the development and use of additional IAs, dedicated to the collection and presentation of data from an agent-based system, in order to facilitate the explanation of its behaviour.

Such "Verification Agents" would be responsible for monitoring the execution of the system to be verified, helping in the explanation of its internal decision processes and ultimately providing a powerful tool in the V&V activity. The verification agents certainly do not need a high level of complexity to perform their function. Furthermore, while such agents probably need to be tuned for use with the specific target system, it is possible to think that the general structure could be reused. Combined together, these two characteristics should solve the problem of the "verification of the verification tool". However, it is important to highlight that, while constituting an interesting theoretical approach, this methodology is at a very early stage of development and, at present, would need consistent work to be successfully applied as a software-engineering tool.

2.7 Soar, Simulink and the ASTRAEA programme

In Section 1.2, it was noted that the SAMMS project evolved from the research work undertaken at the University of Sheffield under the ASTRAEA programme. In this section, this research work will be presented.

The ASTRAEA work is not only an influential predecessor to the present PhD project, but it also constitutes significant proof of the viability of the Soar/Simulink approach for the development of a complex control system. For this reason, the project is now presented in great detail, describing the theoretical aspects, the practical development and the simulation tests that were carried out. The ASTRAEA work shares the idea of integrating Soar IAs with Simulink functions with the SAMMS project, and by treating all aspects regarding it, it is hoped to bring forward the validity of the approach.

2.7.1 A Multi-Agent System for gas-turbine engine Health Management

As discussed in Section 1.2, Sheffield University contributed to the ASTRAEA programme as a partner of Rolls-Royce, by developing long-term Engine Health Management (EHM) functionality based on Intelligent Agents. Using Failure Modes Effect and Criticality Analysis (FMECA) data provided by Rolls-Royce, a Prognosis framework for quantitative estimation of the effects on the aircraft system of an engine fault was developed. On the basis of this Prognosis information, a Multi-Agent System capable of devising fault mitigation plans was created. The Fault-Prognosis component is based on conventional technologies, whereas the Fault Mitigation component uses cognitive Intelligent Agents (Soar agents). Validation and Verification of the entire system was performed using a simulated environment in order to demonstrate its ability to properly assess faults and propose effective mitigation plans.

Current jet engine technology has adopted the use of Full Authority Digital Engine Controllers (FADECs). Using various sensors and actuators, FADECs are capable of detecting unusual or dangerous running conditions of an engine (such as a surge or an airflow stall) and perform immediate reversionary action to counter these conditions. These actions are fully automated, since they can happen in the timescale of tens of milliseconds, so that the pilot would not be able to react in time.

However, a typical FADEC can detect several other types of anomalies that do not represent an immediate risk to the engine but are rather general hints of deteriorating engine performance. At present, this information is presented to the pilot as a warning, or recorded for ground inspection. The course of action to be taken in-flight is determined by the pilot, on the basis of his

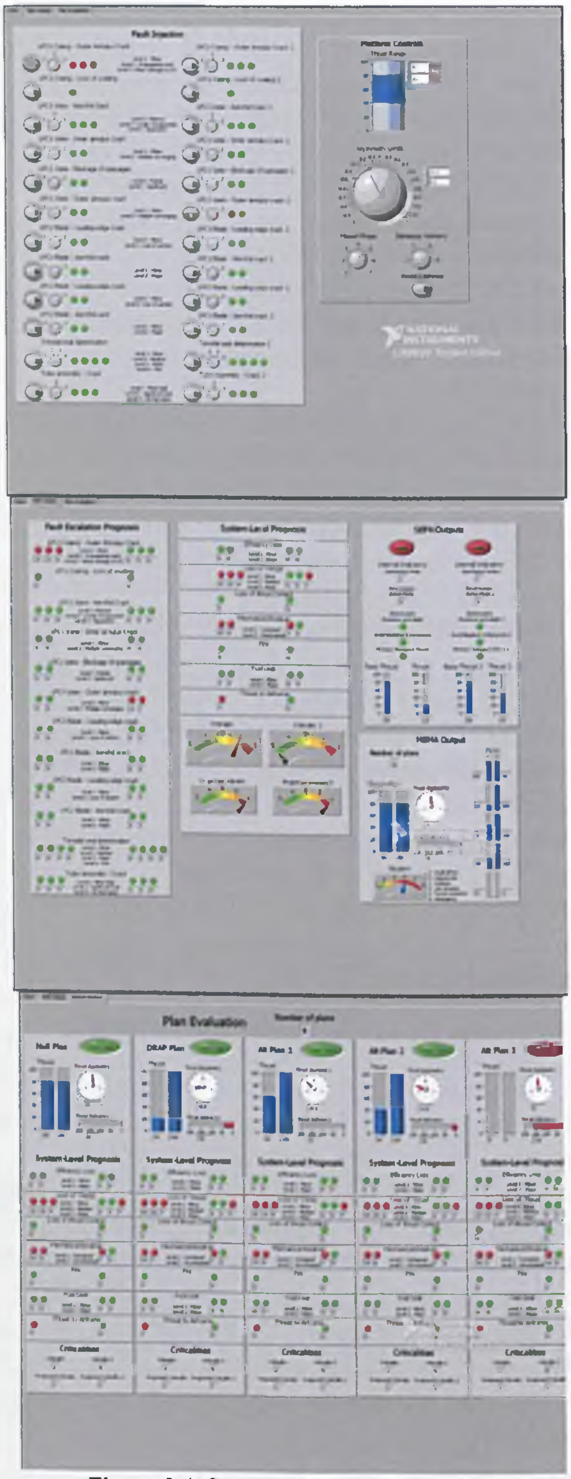


Figure 2.4. Screens of Visual Interface.

experience and knowledge in operating the specific aircraft and engine configuration. When the pilot is taken out-of-the-loop, it is particularly this type of decision-making which is mostly affected. In fact, there is a great level of uncertainty related to such corrective action, and the judgement of a pilot is usually driven by a mixture of expert knowledge and experience that is very difficult to recreate in a computer system. While the mitigating actions for specific faults such as a surge are straight-forward decisions (fault is detected -> mitigating action is taken), decisions about the course of action to take in the case of uncertain faults and in general non-severe propulsion subsystem anomalies are influenced by many factors and in general will fall under the realm of multi-objective optimization (since usually decisions involve a trade-off between various aspects of the mission). In fact, such decisions are influenced by not only the situation of the propulsion subsystem, but also the general condition of the aircraft and knowledge about the current state of the mission.

The core of this work is the development of a UAV Propulsion Health Management (PHM) system, fully interfaced with the UAV supervisory authority and capable of two main functions: Fault Evaluation, which is the ability to prognose the escalation of fault to higher levels of criticality and evaluate the effect of a fault in terms of airframe operations; Fault Mitigation, which is the ability to counteract a fault by performing various types of reversionary action, such as placing limitations on engine usage or demanding an engine shutdown and relight. The Fault Detection and Fault Isolation functions will be considered as already been achieved, but are of course necessary for Fault Evaluation and Fault Mitigation to be performed.

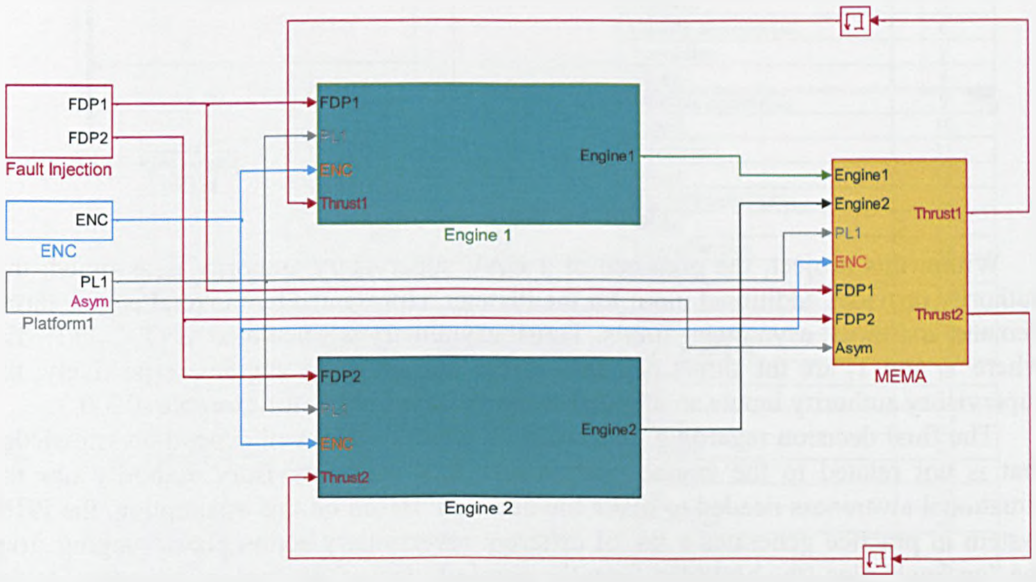


Figure 2.5. Top-level Planner architecture.

Since this is a proof-of-concept work, a representative subset of the FMECA database was used. The PHM system is modelled using the Simulink software tool as a basis; other technologies (Soar Intelligent Agents) are integrated within Simulink. The system is configured to handle a twin-engine UAV configuration. An input interface was developed, allowing injection of faults at different severity stages, along with thrust demands from the UAV supervisory authority. Inputs can come in two different formats: as manual input or as recorded input. Manual input is mainly used for demonstration purposes; a dedicated visual interface (Figure 2.4) was developed using NI LabView software, and is fully integrated with the Planner system model. The visual interface also shows the output in an easily understandable format, without the use of graphs. Recorded input takes the form of pre-prepared files that take the system through a series of different input conditions, and are mainly used during simulations. In the case of recorded input, data is also recorded in data files for further analysis. The FMECA database subset models a total of 12 realistic faults; many of these faults can escalate through different severity stages, for a total of 28 possible fault input conditions for each engine. It is assumed that a single engine will only ever be in one of these states – in case of multiple faults, it is assumed that only the most critical will be addressed by the system. However, it is possible to inject separate faults into the two engines, leading to a total number of fault input combinations of 841 (including no-fault states).

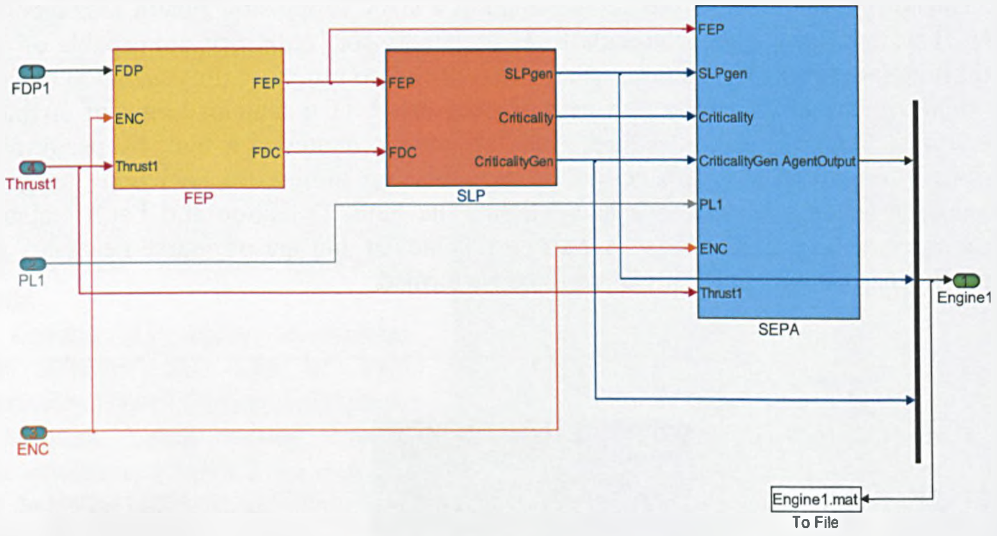


Figure 2.6. Engine subsystem.

Within this project, the presence of a UAV supervisory authority is assumed; this authority provides additional input for the Planner, represented by the total engine thrust demand and thrust asymmetry limits. Thrust asymmetry is calculated as $(T_l-T_r)/(T_l+T_r)$, where T_l and T_r are the thrust demands in the left and right engines respectively; the supervisory authority inputs an allowed range for asymmetry, for example -0.5/0.5.

The final decision regarding the course of action to take will depend on knowledge that is not related to the engine subsystem. Only the supervisory authority has the situational awareness needed to make the decision. Based on this assumption, the PHM system in practice generates a list of different reversionary action plans, ranging from the "optimal" plan (the best plan from the point of view of the engine subsystem) to the "do-nothing" plan (which basically ignores the fault). The number of generated plans is dependent on fault criticality and additional plans between the two extremes present "middle" options that are a trade-off. The Fault Evaluation algorithms are used to give an estimate of how effective a plan will be in mitigating the fault. The plans are then sent to the supervisory authority, together with the prognosis results from the Fault Evaluation algorithms. The authority can then decide which plan to apply, combining the data sent by the PHM system with its situational awareness.

The PHM system was designed using Simulink as the main development tool. Within it, three major subsystems can be identified: the Prognosis Framework, the Single-Engine Planner Agent (SEPA) and the Multi-Engine Manager Agent (MEMA). Figure 2.5 represents the top-level architecture of the PHM system; in the figure, it is possible to observe blocks labelled "*Engine 1*", "*Engine 2*" and "*MEMA*", representing respectively the two engines and the MEMA agent. Each of the engine blocks contains the subsystem shown in Figure 2.6; in the figure, it is possible to distinguish three blocks, labelled "*FEP*", "*SLP*" and "SE*PA*". The FEP and SLP components together constitute the Prognosis Framework, while the SEPA block represents the SEPA agent. In general, the Prognosis Framework is based on standard control systems modelling techniques, while the SEPA and MEMA components are Soar Intelligent Agents.

2.7.2 Components of the PHM system

The Prognosis Framework performs the Fault Evaluation function, complementing Fault Diagnosis data with two types of data: a prognosis of how the fault can be expected to escalate based on engine usage and FMECA data (Fault Escalation Prognosis), and a prognosis of how the airframe is affected by the fault (System-Level Prognosis).

Table 2.2. List of faults and severity stages.

No	Fault type	Severity Stages	Code
1	LPC1 Casing - Outer annulus crack	Minor	1
		Medium	2
		Major	3
2	LPC1 Casing - Loss of coating		4
3	LPC1 Vane - Aerofoil crack	Minimal	5
		Locally unsopported	6
		Separation	7
4	LPC1 Vane - Inner annulus crack	Minor	8
		Multiple converging	9
5	LPC1 Vane - Blockage of passages	Partial	10
		Significant	11
6	LPC1 Vane - Outer annulus crack	Minor	12
		Multiple converging	13
7	LPC1 Blade - Leading edge crack	Minor	14
		Loss of section	15
8	LPC1 Blade - Aerofoil crack	Minor	16
		Major	17
9	LPC2 Blade - Leading edge crack	Minor	18
		Loss of section	19
10	LPC2 Blade - Aerofoil crack	Minor	20
		Major	21
11	Toroidal seal deterioration	Minor	22
		Medium	23
		Major	24
		Fire	25
12	Tube Assembly - Crack	Minor leak	26
		Significant leak	27
		Oil starvation	28

Fault Escalation Prognosis is basically a direct implementation of data extrapolated from the FMECA database. It uses a three-dimensional look-up table to prognose how a fault is expected to escalate in time. Each modelled fault is classified in a varying number of severity stages and the Fault Escalation Prognosis block estimates the timescale after which the fault can be expected to escalate to a higher severity stage. The key point is that the escalation time is heavily influenced by engine usage, so the time to escalation generally increases as thrust demand reduces. This is the reason for using a 3D look-up table, as it necessary to provide escalation estimates for different running conditions. The output of the look-up table is the "time to escalation" estimate for the current stage of the fault towards more critical stages. Both the fault severity stages and the timescale values are discretized. Table 2.2 lists the currently modelled faults and their stages, while Table 2.3 indicates the timescale discretization used. The whole idea of a PHM system relies on the concept of fault escalation: the PHM system will generate reversionary action plans to increase or maximise the time to escalation for a fault, enabling the UAV to successfully complete its mission.

Table 2.3. Timescale codes.

Code	Time to failure
1	30 sec
2	1 min
3	5 min
4	10 min
5	30 min
6	1 h
7	2 h
8	5 h
9	10 h
10	20 h
11	50 h
12	100 h
13	150 h
15	Fault has happened
0	Fault not detected

System-Level Prognosis is instead aimed at extracting useful information from the Fault Diagnosis input and the Fault Escalation Prognosis. From a theoretical point of view, it provides an answer to the question: "How will this engine fault affect the operation of the entire UAV?". The rationale behind the System-Level Prognosis function is the fact that a UAV supervisory authority is not concerned about the actual nature of an engine fault, but only about its effects on UAV capabilities. As an example, the supervisory authority is not interested in knowing that the outer annulus in the low-pressure compressor casing is cracked, but it needs to know that this will cause a reduction in actual thrust that is dependent on how severe the crack is. Table 2.4 lists the 7 different types of System-Level Effects (SLEs) that have been identified for the PHM system. Some of these can evolve through subsequent stages, which are discretized as is the case with fault severity stages, leading to a total of 12 SLEs. The SLEs are directly derived from the FMECA database. The system uses a simple two-dimensional look-up table to calculate the SLEs related to a fault, since the relationship between fault and SLE is straight-forward. The System-Level Prognosis function also performs another calculation: it assigns a Criticality level to the current detected fault and also to the relative escalation stages that are prognosed. These Criticality levels provide an immediate way of classifying the severity of a fault, and are discretized as per common practice within FMECA databases. Table 2.5 lists Criticality levels and their definitions.

Table 2.4. System Level Effects (SLEs).

SLE	SLE description	Stages
1	Efficiency loss	Minor
		Major
2	Loss of thrust	Minor
		Medium
		Major
3	Loss of thrust control	
4	Mechanical break-up	Contained
		Uncontained
5	Engine fire	
6	Fuel leak	Minor
		Major
7	Threat to airframe	

Table 2.5. Criticality levels and definitions.

Level	Definition	Description
5	No fault	No fault is detected
4	Negligible	Fault only has negligible effects
3	Minor	Fault affects performance but does not compromise mission
2	Severe	Fault can compromise the completion of the mission
1	Catastrophic	Fault can lead to loss of engine and eventually loss of aircraft

Overall, the output of the Prognosis Framework consists of three vectors, reporting respectively Fault Escalation Prognosis, System-Level Effects and Criticality levels. The Framework operates on single engines, so that three of these vectors will be generated for each engine, just as Fault Diagnosis input is separate for each engine.

The Single-Engine Agent Planner or SEPA is the agent entity which performs the Fault Mitigation function related to a single engine. Its task is to develop reversionary action plans that address a fault from the point of view of a single engine. Once a fault is detected and evaluated through the Prognosis Framework, the SEPA proposes two courses of action: a "do-nothing" plan and an "optimal" plan, which is the best action course for the engine regardless of what is the situation in the rest of the Propulsion system or the entire UAV. It is then the task of MEMA (subsequently described) to contextualize the plans and derive alternative ones.

Table 2.6. SEPA states.

No	State	Description
0	no-fault	No fault is detected
1	no-action	Fault is negligible and does not require reversionary action
2	increase-power	Fault causes a minor reduction in effective thrust that can be compensated
3	reduce-prognosis	Fault is minor but can escalate to critical, so usage limitation is requested
4	reduce-power	Fault is severe and usage limitation is requested
5	shutdown-engine	Fault should be addressed with an engine shutdown

The SEPA is modelled using the Soar Intelligent Agent tool. The SEPA core rules implement a decision making scheme that uses data generated by the Prognosis Framework to propose reversionary action plans. The SEPA enters different states depending on the current input and goes through a stepped decisional tree in order to derive a full action plan. The plan consists of a proposed thrust value for the engine (usually limited to some degree if a fault is detected), a "do-nothing" thrust value and three binary indicators that complement the plan by indicating other possible reversionary action. Table 2.6 lists the possible states that the SEPA can enter when generating the most important part of the plan, which is the proposed engine thrust value.

Once a reversionary action plan is generated for a single engine by the SEPA, this plan needs to be put in the context of the entire propulsion system. This involves considering the situation of the other engine at first, and then the demands that are coming from the UAV supervisory authority.

The MEMA generates at first a "do-nothing" plan and an "optimal" plan, which are basically a symmetric thrust distribution and an asymmetrical one respectively. The asymmetric thrust distribution usually found in the optimal plan is a consequence of the usage limitations that are placed on a faulty engine. In practice, the plan will usually involve following the advice from SEPA regarding the faulty engines, and then compensating a decrease in thrust on that engine by increasing the thrust on the other engine. The "optimal" plan will always keep the usage limits requested by SEPA, whereas the "do-nothing" plan disregards these and just provides a symmetrical thrust distribution that matches the total engine thrust demand by the supervisory authority. It is important to understand that the optimal plan does not guarantee that the total thrust provided will be meeting the total demand. The MEMA then eventually generates alternative plans that are a trade-off between the optimal and "do-nothing" plans. The number of alternative plans is dependent on the Criticality of the faults being addressed, and the total number of plans ranges between two (Criticality Level 4, no alternative plans) and five (Criticality level 1, three alternative plans).

Since the system is meant to address the situation where both engines present a fault, a series of decisional behaviours have been outlined. Depending on the state of SEPA on each engine (see Table 2.6), the MEMA enters a combined state, which is an abstraction of the type of action that must be taken, basically indicating whether one engine should get priority in Fault Mitigation or if it is instead advisable to simply decrease the total thrust demand. Table 2.7 represents the decisional matrix for the

MEMA state, depending on SEPA state, and explains the meaning of MEMA states. In the end, the MEMA outputs a list of plans; each plan is represented by two thrust values, one for each engine. The plans are then sent to the supervisory authority, which has the final word on choosing the plan to be actuated. In order to make this decision, the supervisory authority needs an evaluation of the expected outcome of the plans. To obtain this, the proposed thrust levels are fed into the Fault Escalation Prognosis algorithms used in the Prognosis Framework; in this way, each plan can be presented to the supervisory authority together with an estimate of how the plan will affect the escalation timescale. The authority is then able to make a decision based on this data and other data situational awareness data that is not concerning the propulsion subsystem.

Table 2.7. MEMA decisional matrix and states.

SEPA	0/1	2	3	4	5
0/1	1	1	2	3	3
2	1	1	2	4	4
3	2	2	5	4	4
4	3	4	4	5	6
5	3	4	4	6	6

No	State	Description
1	small-effect	No reversionary action is taken, apart from compensation of missing thrust due to minor faults
2	reduce-risk	Thrust is decreased on faulty engine in order to postpone further escalation of fault, increased on other engine
3	mitigate	Serious fault on one engine is addressed by placing usage limitation and compensating with other engine
4	risk-situation	Both engines are faulty, thrust is decreased where fault is more serious and increased on other engine, shortening the escalation time
5	forced-reduction	Both engines are faulty with similar criticality, a reduction of total thrust demand is requested
6	emergency	Both engines are in critical condition and need a shutdown (very unlikely case)

As an example, let us consider the following case: while the total engine thrust demand is 70%, a fault at the first severity stage is detected on the right engine. At 70% thrust, the fault would escalate to the next severity stage in one hour time (timecode 6). The SEPA proposes a plan that reduces thrust on the faulty engine to 40%, thus the MEMA proposes an optimal plan with thrust distribution 40%-100%. This plan is evaluated to extend the escalation of the fault to 5 hours time (timecode 8). However, this means that the asymmetrical thrust coefficient is -0.43. Due to the low fault severity, only one alternative plan is generated, corresponding to a 55%-85% thrust distribution (asymmetry coefficient -0.21) and an escalation timescale of 2 hours (timecode 7). These options are presented to the supervisory authority; we can assume that the supervisory authority has knowledge of conditions that do not allow for an asymmetry coefficient greater than ±0.3, therefore excluding the optimal plan; however, it knows that the remaining mission time is between one and two hours, so the middle option is chosen since it represents a good trade-off, as the fault does not escalate before the end of the mission and the thrust is provided with an acceptable asymmetry.

This is just an example of how the decision process might work for the PHM system; the range of possible situations is much wider and largely different strategies will be adopted under different conditions, however the governing philosophy remains the same: mitigating faults by reducing engine usage, while at the same time considering situational awareness that is not related directly to the Propulsion system.

2.7.3 PHM system tests

The Propulsion Health Management System was tested using a simulation environment, also modelled in Simulink. Within this environment, the system receives input including Isolated Faults and UAV supervisory authority commands, processes it and then outputs the plans together with their respective evaluation data. Simulation runs use files for input and output, so that pre-recorded input can be used and output can be recorded for later analysis.

Two main types of simulations were performed. The first type consists of assigning fixed demands from the supervisory authority and then inputting all of the 841 possible fault input combinations; this type of simulation is useful in proving the determinism of the system and verifying that it always responds within certain constraints. The simulation involves changing the fault input at every second of simulation. Figure 2.7 shows part of the results for such a test; in this case, the total thrust demand is set to 70% (which would normally require 70% on each engine). It is possible to note that the "do-nothing" plan is stable at 70%, while the "optimal" plan varies depending on the type of fault. The number of generated plans also changes, ranging from cases where only two plans are present to cases with five generated plans. It is also possible to note how the non-faulty engine is used to compensate for proposed limitations to the faulty engine, but in many cases cannot guarantee the same amount of thrust since it cannot go over 100%.

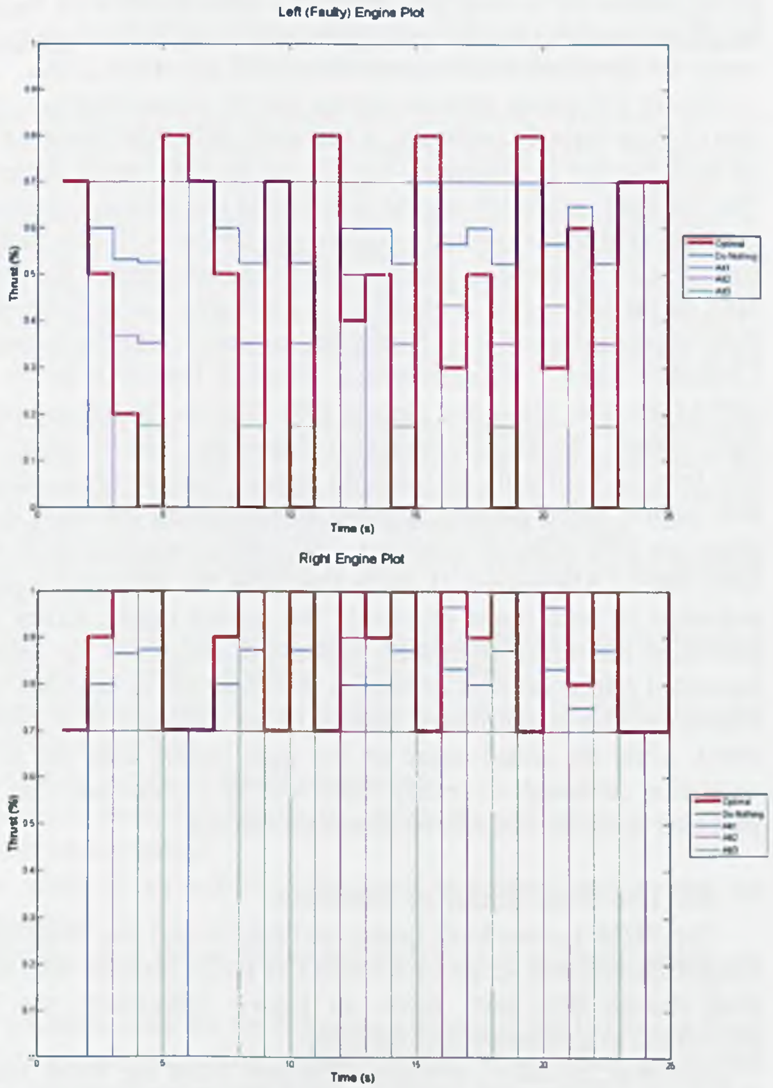


Figure 2.7. Full fault input combination tests.

The second type of simulations involves verifying the behaviour of the system when demands from the supervisory authority are changing. In these tests, a specific fault condition is fixed and the behaviour of the system on the occurrence of changes in the demands from the supervisory authority (or platform) is monitored. For example, an outer annulus crack fault at the second severity stage can be injected in the left engine, while total thrust demand is varied in time. Tests show that four plans are generated in this case; the optimal plan (limitation at 20% thrust), the do-nothing plan (no limitation), and two intermediate plans, respectively asking for limitations between the 20% limit and the actual thrust demand. Thrust limitations on a faulty engine are compensated (if possible) with increases on the other engine.

Finally, since evaluating the correctness and appropriateness of generated plans can be a very complex task, manual simulations were also performed. In these cases, a set of input configurations was determined randomly and then the behaviour of the system verified directly by a human user, which could evaluate the performance of the various components of the system. Such tests made large use of the LabView visual interface, which allows for a clear and immediate understanding of the results. A representative set of input configurations was chosen and visually verified by a human user in order to verify the correctness and appropriateness of generated plans.

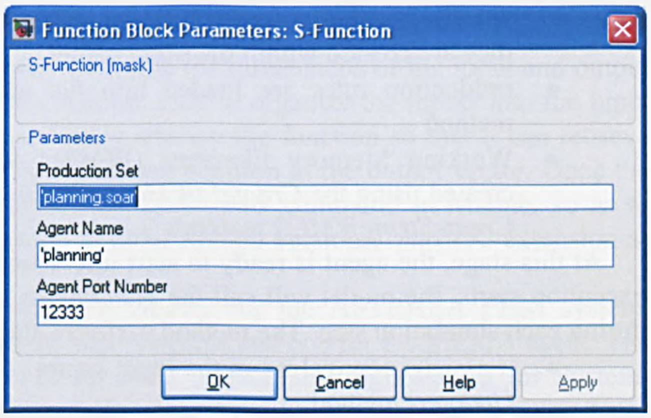
Figure 2.4 shows capture screens for the visual interface (input and output) during one of these tests. In particular, a case with 80% total thrust demand and different faults on both engines is presented. Looking at the first output screen, it is possible to notice that the fault on the left engine is expected to escalate in timecode 4,while the fault on the right engine is expected to escalate in timecode 7. The faults present different SLEs that are also expected to escalate when the fault reaches the highest severity stage. The fault on the left engine is classified at Criticality Level 2 (Severe) while the fault on the right engine is classified as Negligible; however, both faults are prognosed to escalate to Criticality Level 1 (Catastrophic), although they will do so at different times. The SEPAs generate plans that place a 20% limit on the left engine and a 50% limit on the right engine. The MEMA decides to follow the "risk-situation" protocol, due to current low level of Criticality on the right engine. This involves actually increasing usage on this engine, since priority is given to addressing the other fault. The four generated plans are 80%-80% (do-nothing), 20%-100% (optimal), 40%-100% (alternative 1) and 60%-100% (Alternative 2). Note that only the Alternative plan 2 does not require a reduction of total thrust provided. The second output screen analyzes plans in detail, providing the thrust asymmetry and thrust deficiency for each plans and showing the estimated effects on fault escalation. It is possible to note that the limitations on the left engine provide a substantial benefit (from timecode 4 to timecode 7 for the optimal plan), while the added usage on the right engine does not involve a reduced time to escalation (although in reality there will be a reduction, this is not big enough to be captured using the discretized timescale values).

2.8 The Soar/Simulink interface

The PHM system built under ASTRAEA and the SAMMS project are apparently quite different; the scope of SAMMS is much broader than that of the PHM system, even though they both focus on UAVs. Ultimately, the PHM system could be considered as a subsystem of SAMMS.

However, SAMMS actually originates from the PHM research work. The whole concept of Soar intelligent agents integrated within a Simulink environment was first ideated and implemented during the PHM work. SAMMS is a logical progression from the PHM experience. It is important to note that the PHM work was critical in providing not only the concept of Soar/Simulink integration, but also a complete set of implementation strategies. The most important of these is the Soar/Simulink interface which was developed for the PHM system and applied (with the due modifications) also for the SAMMS project.

As stated in Section 2.4, the Soar kernel resides in a set of C++ dynamic libraries (*dll* files). In order to execute a Soar agent, processes from these libraries must be invoked, so that the agent can be created, load a set of productions and then interface with its environment.



Production Set
'planning.soar'
Agent Name
'planning'
Agent Port Number
12333

Figure 2.8. Soar agent interface mask.

The Soar/Simulink integration interface is based on a Simulink S-function. An S-function is a computer language description of a Simulink block (The Mathworks, 2006). S-functions can be written using the Matlab, C, C++, Ada, or Fortran; S-functions not written in Matlab must be compiled before execution of the simulation. S-functions use a special calling syntax that enables interaction with Simulink equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks. In short, S-functions allow to add custom blocks to Simulink models. By following a set of simple rules, any algorithm can be coded in an S-function.

Since the Soar kernel is written in C++, the Soar/Simulink interface S-function was written in C++ too. A template for S-functions is available and was used as the base for the code. S-functions are divided in several methods, which are executed during different parts of the simulation. The following methods are normally present:

- *mdlCheckParameters*; called at simulation start-up to validate the parameters needed by the S-function
- *mdlInitializeSizes*; called at simulation start-up to determine the S-function block's characteristics (number of inputs, outputs, states, etc.)
- *mdlInitializeSampleTimes*; called at simulation start-up to determine the sample time for the S-function
- *mdlStart*; called once at start of model execution;
- *mdlOutputs*; called at every simulation step, this is the part where the outputs of the S-function are computed
- *mdlTerminate*; called at the end of a simulation to perform actions that are necessary at this stage

In the Soar/Simulink interface, it is the *mdlStart* and *mdlOutputs* methods which are most important.

Figure 2.8 shows the interface mask for the S-function, which is used to assign three parameters to the agent: the set of productions to load (stored in a text file), a univocal agent name and an agent communication port number. When the *mdlStart* method is called, the following actions are performed:

- a new thread containing the Soar kernel is created using the *CreateKernelInNewThread()* method; if an existing kernel is detected, a pointer to that kernel is retrieved instead; the specified communication port number is assigned to the kernel, so that communication with the agent can be established during execution

- a new agent is created within the kernel using the *CreateAgent()* method; the specified name is assigned to the agent; when multiple agents are created, they are created within the same kernel
- production rules are loaded into the agent using the *LoadProductions()* method
- Working Memory Elements (WMEs) for the entire input structure are created using the *CreateIdWME()*, *CreateIntWME()*, *CreateFloatWME()* and *CreateStringWME()* methods

At this stage, the agent is ready to start interacting with the Simulink model. When execution starts, the model will call the *mdlOutputs* method at the appropriate moment during each simulation step. The method performs the following tasks:

- it assigns new values in the input vector to the appropriate WMEs, using the *Update()* method
- it retrieves outputs of the Soar agent and assigns them to the S-function output vector, using a combination of the *GetNumberCommands()*, *GetCommand()*, *GetCommandName()* and *GetParameterValue()* methods
- it instructs the agent to execute; the *RunSelf(number)* method executes *number* decision cycles; the *RunSelfTilOutput()* method executes decision cycles until the agent has produced output; the *RunSelfForever()* method executes the decision cycle continuously

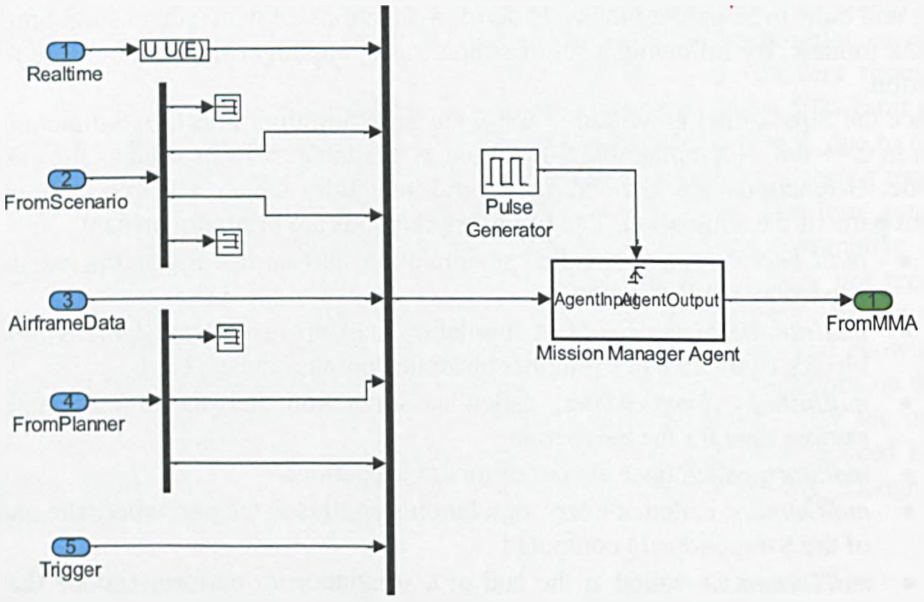


Figure 2.9. Soar agent within a Simulink model.

Figure 2.9 shows the Soar/Simulink S-function placed within a model. All inputs are united into a single vector and all outputs are sent back as a single vector. Vector composition and decomposition occurs within Simulink. Within Simulink models, Soar agents are normally placed in *Triggered Subsystems*, which are subsystems that execute only when certain conditions are met. This is because it is usually desirable for the agent to have a different sample time than the Simulink model. Use of a *Triggered Subsystem* block ensures that the agent will be called only when specified (for example, the model might have a step size of 0.1 s, but call the agent only every 0.5 s).

The Soar/Simulink interface does not work without adjustment for any agent. It is basically an S-function template, which needs modifications in order to embody an agent. So for each set of production rules that constitute an agent, a dedicated version of the interface will be needed. The adjustments to make are mostly regarding the Input/Output structures: it is necessary to define the dimensions of the input and output vectors between Simulink and the S-function, then to organize the inputs into the input structure expected by the agent, and finally arrange the function so that it can retrieve output from the agent and place it at the correct position in the output vector. Once the S-function is ready, it can be compiled using the *mex* command in Matlab, so as to create a dynamic library (*.dll* extension in older Matlab versions, *.mexw32* extension as of release R2006a).

The Soar/Simulink interface was first created for the ASTRAEA PHM system, where two versions existed: one for the SEPA agents and one for the MEMA agent. The interface for SEPA has 70 inputs (28 for Fault Escalation Prognosis, 12 for System-Level Prognosis, 14 for Criticality, 6 for Supervisory Authority demands, 8 for Environmental Conditions, 1 for Current Engine Thrust and 1 to distinguish between engines) and 9 outputs (two thrust values, two mode indications, three additional planning binary switches, a state indicator and an asymmetric command indicator). The interface for MEMA has 60 inputs (6 for Supervisory Authority demands, 8 for Environmental Conditions, and for each engine 12 for the System-Level Prognosis, 2 for Criticality and 9 for the SEPA output) and 16 outputs (2 Thrust values for each of the 5 plans, 2 values for missing thrust and asymmetry and 4 state indicators). With these numbers of inputs/outputs, it was not necessary to use nested structures for the agent I/O.

The interface proved the technical feasibility of a Soar/Simulink system. The same concept and the same interface were then utilized for the SAMMS project. It is to be noted that I/O in SAMMS is constituted by much larger numbers of variables, so the interface had to be improved to utilize nesting.

2.9 Concluding remarks

In this chapter, an overview of Intelligent Agent technology was given. First, the theoretical background behind the concept of Intelligent Agent was introduced, and its significance from a software engineering point of view discussed. This was followed by a report of available IA architectures. The choice of Soar as the basic development platform was then motivated, leading to a detailed description of the Soar architecture, looking both at theoretical aspects and at practical applications. Considerations about validation and verification of IA software were elicited. The work carried out under the ASTRAEA programme was presented, detailing in particular the use of Soar agents within a Simulink environment, which is a concept that is the foundation also for the present work. Finally, the Soar/Simulink interface first developed for the ASTRAEA programme and then adapted for use in the present PhD project was described.

The next chapter will be dedicated to a thorough overview of the architecture of the Soar-based Autonomous Mission Management System (SAMMS). The Soar Intelligent Agents are detailed in later chapters, so the focus of the chapter is the description of the overarching architecture. At first, theoretical abstractions that are at the base of the work will be introduced, then the division of functionality between the different components of SAMMS will be described. Finally, accessory components will also be presented.

Chapter 2 – The Technology

3. Architecture description

The Soar-based Autonomous Mission Management System (SAMMS) presented in this thesis is a complex system with several components developed using different technologies and performing different functions. As stated in Chapter 1, the goal for the SAMMS project is to develop a UAV control system that is capable of performing an entire mission with multiple objectives without human supervision. In order to accomplish this, careful thought must be placed on the integration of the various components of the system.

In fact, laying out the theoretical background and base architecture was the first step for the SAMMS project. The precursor project described in Section 2.7 heavily influenced this planning phase, and the activity resulted in two main background achievements: the formalization of a set of abstractions for UAV Mission Management, and the development of the core SAMMS architecture. The architecture defines SAMMS components and their interactions, also specifying the technology upon which they are based and the functions that they are expected to perform.

In this chapter, both these theoretical background studies will be thoroughly described. The set of UAV Mission Management abstractions will be introduced in Section 3.1; its importance within SAMMS will also be motivated. Then, in Section 3.2 an overview of the SAMMS architecture will be presented, detailing the components and the functions they are supposed to perform.

While the core of the architecture is represented by three Soar intelligent agents that are each described in its own dedicated chapter, other components of the architecture (in particular, the UAV dynamics model and the autopilot model) will be introduced in this chapter. These components do not present any novelty, but are very important in the validation process of the core SAMMS functionality and are extremely useful in putting the SAMMS work into a realistic development perspective. In particular, Section 3.3 presents the entire simulation environment for SAMMS, detailing specific minor components. The UAV mathematical model used for validation purposes within SAMMS is described in Section 3.4. Finally, the autopilot subsystem, that is an important low-level component of the SAMMS architecture, is presented in detail in Section 3.5.

3.1 Abstractions

In Section 1.1, an example scenario was introduced, outlining the flow of information and the decision process that happens when a pilot is assigned a mission by his supervisor. Summarizing, the following phases are accomplished: the supervisor communicates mission objectives; the pilot collects additional information and formulates a flight plan; the aircraft is readied for departure and the mission begins; as the mission is being accomplished, the flight plan is adjusted in flight, in response to new available information or contingencies; the pilot lands the aircraft at a destination airport, having or not achieved the mission objectives, and thus the mission ends.

In an autonomous UAV, this entire process must be automated. Phase by phase, this means that:

- mission objectives must be defined by the UAV operator so that they can be understood univocally by the UAV control system
- similarly, additional information (such as weather status, Air Traffic Control information, threats) must be presented to the UAV in a specific format

- the UAV must be able to formulate a flight plan using some form of internal representation
- changes in mission objectives or situational information result in changes to the flight plan

Mission objectives and situational information can be presented to a pilot in a rather unstructured manner, as the pilot is obviously capable of ordering this information in a more significant way, and eventually requesting clarifications. Also, a human pilot can define flight plans quite loosely (the main constraint in this sense is ATC). When the pilot is taken out of the loop, all this information must be structured so that it is treatable by a computer system.

This problem is certainly not unique to SAMMS, however it was not possible to find a sufficiently detailed set of abstractions that could be used for the implementation of SAMMS. Thus, a set of abstractions was developed internally.

An interesting starting point for the abstractions is the work presented in (Nehme, Cummings and Crandall, 2006) and (Nehme, Crandall and Cummings, 2007). In this work, various UAV mission types are defined, together with the relative functional requirements and the actions expected from the UAV operator. While the work has a tendency to be more concerned with the payload management aspect of UAV autonomy (which is not contemplated in the present PhD project), it significantly defines a limited number of generic mission types that can or could be performed by a UAV. The mission types are shown in Figure 3.1. It is to be noted that several of these are very similar from the point of view of a mission profile, only involving the use of a different type of payload; for example, many of the "Intelligence/Reconaissance" mission types (Battle Damage Assessment, Target Acquisition and Target Designation) are very similar (the Mapping mission type is instead significantly different).

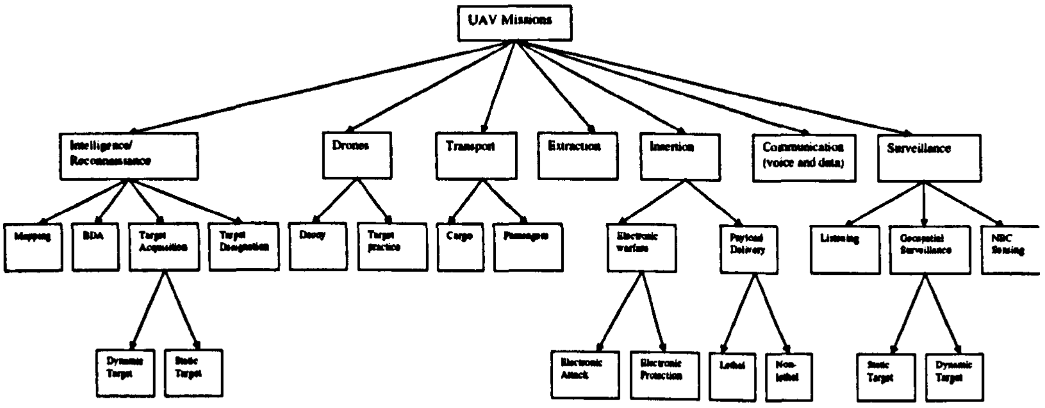


Figure 3.1. UAV missions overview (Reproduced from (Nehme, Cummings and Crandall, 2006)).

The set of abstractions for SAMMS is very different from this, as it performs a different function, but was developed so that all mission types defined in (Nehme, Cummings and Crandall, 2006) could be accomplished, and that all information listed as required for each mission type would be available to the system. For example, as will be seen in Section 3.1.1, the Objective types that can be defined within SAMMS are representative of the mission types defined in Figure 3.1, although in many cases several mission types have been grouped together into a single Objective type since from a mission management point of view they are very similar (they can be very different from a payload management point of view).

Three main abstractions or concepts have been defined for SAMMS: the Objective, the Entity and the Action. In short, the Objective is the format used by the UAV operator to define mission goals, the Entity is the format used by the UAV sensor and communication systems to define external presences that can have an influence on the UAV, and the Action is a high-level task (but at a lower level than the Objective) that represents a significant part of a mission (so that an entire mission can be decomposed into a series of Actions). Each of these abstractions has a corresponding data structure that is used throughout SAMMS for inter-agent and agent-component communication. In order to better understand the relationship between the abstractions and how they fit within the overall architecture, please refer to the conclusion of Section 3.2, and in particular to Figure 3.3, which shows the way each abstraction is used and the SAMMS component that need it.

3.1.1 Objective

The Objective is the abstraction that is used to define mission goals within SAMMS. The SAMMS operator must formulate a mission by defining a number of Objective structures (currently limited to ten). An Objective represents a very high-level task for the UAV, defining a significant part of a mission.

The types of Objectives can vary greatly depending on the specific type of UAV, but at present five generic types have been defined, as in Table 3.1. In fact each of these types is representative of various mission types that present very similar characteristics. The *analyze target* and *attack target* Objective types both imply the presence of a target (which within SAMMS is an Entity, and can present very different natures); the UAV must fly to the (sometimes estimated) position of this target and then perform a "pass" over it, in order to either use a sensor payload on the target or deliver a payload on it (not necessarily a weapon, a supply drop is essentially very similar). The *orbit position* Objective type is used to have the UAV loiter about a position; this could be necessary for several reasons, such as waiting for some external event, acting as a communications relay, using a sensor payload that requires to maintain the current position. The *search area* Objective type is generically indicative of the need to fly over large areas searching for targets; for example, a maritime patrol mission is implemented by this Objective type; various standard search patterns can be used (as will be detailed in

Table 3.1. Objective types.

1	Analyze Target	Go to a position to gather data on a specific target using payload sensors
2	Attack Target	Deliver a payload on a specific target
3	Orbit Position	Circle about a position for a specified time, for example to act as communications relay
4	Search Area	Patrol an area using standard patterns in order to identify targets
5	Transit	Travel to a destination airport and land there

Chapter 4) and as targets are identified, specific action can be taken (e.g. a new analyze/attack target can be added automatically by SAMMS). Finally, the *transit* Objective type is representative of all missions that end at a different airport from the starting one; for example, a single transit Objective would form the core of a normal cargo/passenger transport mission.

Each Objective needs several parameters (or properties) in order to be completely defined; some of these are common to all Objective types, while others are relevant only for certain types. Table 3.2 summarizes all 12 Objective properties. Some properties are self-explanatory, however further detail is needed for others.

The *Objective Tag* is a unique numeric identifier that is assigned to an Objective in order to avoid confusion between Objectives; it does not represent the order in which

Objectives will be executed, since this is determined by SAMMS on the basis of various factors. The *Target Tag* property is used by analyze target and attack target Objectives and refers to the Entity Tag property of the Entity that represents the target.

Table 3.2. Objective properties.

ID	Property	Property description	Objective Type Relevance				
			1	2	3	4	5
1	Objective Type	Analyze, attack, orbit, search or transit	x	x	x	x	x
2	Objective Tag	A numeric code that identifies an Objective	x	x	x	x	x
3	Objective Position	Coordinates for the specific Objective	x	x	x	x	x
4	Priority	Time and execution priority	x	x	x	x	x
5	Duty	Task being accomplished during Orbit			x		
6	Area Type	Defines the type of Search pattern				x	
7	Search Accuracy	Accuracy for a Search mission				x	
8	Box Corner	Defines a Box-type Search area				x	
9	Search Radius	Defines a Circle-type Search area				x	
10	Target Tag	Identifies a specific target for analyze\attack	x	x			
11	Orbit Time	Defines the time limit for an Orbit mission			x		
12	Orbit Altitude	Defines the loiter altitude for an Orbit mission			x		

The *Priority* property is actually constituted by two different values. The *time priority* is used to instruct SAMMS to execute an Objective within certain time constraints; three types of time priority are available: *immediate* (instructing SAMMS to execute the Objective before all others – hence only one Objective at a time can have this time priority), *preference* (instructing SAMMS to ensure execution of the Objective before a specified time limit) and *normal* (time is not an issue for the execution of the Objective). The *execution priority* is a measure of how important the Objective is within the overall mission; this in turn influences the risks that SAMMS will deem acceptable when executing an Objective, and which Objectives will take precedence upon the others should a choice be made (for example, if a failure limits the UAV's range and thus makes it impossible to complete all Objectives, the ones with lower execution priority will be discarded). Execution Priority levels are defined on the basis of the threat levels, as in Table 3.3; during a mission, an Objective will be aborted if a threat level higher than its execution priority is perceived (for example, an Objective with execution priority 5 cannot be aborted, while an Objective with execution priority 3 will be aborted if a threat of level 4 is perceived in the Objective area). It is important to understand that time and execution priority are unrelated: an Objective could have an immediate time priority and an execution priority of level 1 (thus indicating an Objective that must be accomplished immediately, but only if minor risks are involved), or a normal time priority and an execution priority of level 5 (thus indicating that the Objective can be accomplished at any time, but must be accomplished at all costs).

Table 3.3. Threat levels.

Code	Definition
1	Negligible
2	Can affect mission performance
3	Can compromise mission
4	Can damage airframe
5	Can destroy airframe

The *Duty* property is used to indicate the type of mission to be performed during an Orbit Objective; this is not strictly necessary, but could be useful in optimizing the loiter pattern. Also used during an orbit Objective are the *Orbit Time* and *Orbit Altitude* properties, indicating respectively the time at which the Orbit Objective is finished (so that the UAV can move on to the next Objective) and the altitude to be kept during this particular Objective. The *Area Type*, *Search Accuracy*, *Box Corner* and *Radius* properties are all used to define the search pattern used during a search area Objective.

3.1.2 Entity

An autonomous UAV has an obvious need to be aware of its environment. This involves the need to gather large amounts of information from a huge variety of sources. For example, the autonomous UAV will need information about the weather, the current Air Traffic Control (ATC) situation, other aerial traffic in the area, the position and behaviour of its targets.

It is clear that all this information is not readily available to the UAV, but must be gathered and then correctly interpreted. Sensor data processing presents several issues:

- the information coming from different sources (internal UAV sensors as opposed to externally provided through a radio link) might need to be fused to improve situational awareness
- information regarding different subjects will usually be presented in a different format
- some types of information might need consistent amounts of processing to be actually meaningful (an example of this is described in Section 2.7.2, where fault detection/isolation data, not very useful to a UAV supervisory authority in its raw form, is processed by fault evaluation algorithms to extrapolate more significant information)

In SAMMS, the Entity abstraction is used to standardize the format in which all external information is provided. During the present PhD project, it is assumed that all information which is not directly sensed by the UAV's navigational sensors will be

Table 3.4. Entity types.

Code	Definition	Description
1	Air Vehicle	An external flying vehicle, to be considered for collision avoidance purposes
2	Ground Vehicle	An external ground presence possessing a known movement capability
3	Building	An external ground presence that does not have a movement capability
4	Weather Zone	A clearly defined area where particular meteorological phenomena have been detected
5	ATC Zone	A clearly defined area for which ATC rules are in effect (e.g. no-fly zone, airway, etc.)
6	Generic Threat	A particular object or area that presents a threat to the UAV (for example, high-voltage cables, a mountain peak, etc.)

presented to the UAV using this format. This means that in an actual system an intermediate layer will be needed to process external information and format it using the Entity abstraction. However, this intermediate layer was not implemented during this project. The intermediate layer is largely dependent on the characteristics of the system which is providing the information to be processed, and since the objective for this project is the development of a generic UAV control architecture rather than an actual flying UAV, this "information gathering" system has not been detailed. In fact, throughout the project the availability of external information formatted using the Entity abstraction will be assumed.

Within SAMMS, information of a very varied nature is treated as Entities. Examples of this include navigational hazards, mission targets, hostile presences, weather information and ATC instructions. For the purposes of the present PhD project, the

Entity types in Table 3.4 have been considered; however, this is not meant as an exhaustive list, as the amount of Entity types in an actual UAV depends on available information.

Just as with the Objective abstraction, an Entity abstraction is defined by a set of parameters (or properties) which are structured in a pre-determined manner. Table 3.5 summarizes these, however a more detailed description is order. .

The *Entity Type* property uses codes from Table 3.4 to specify the nature of the Entity. The *Entity Tag* property is a unique numeric identifier that is used to avoid confusion between Entities; for example, the Target Tag property of an analyze/attack target Objective will contain an Entity Tag value so that the mission objective is univocally identified. The *Entity Position* property is self-explanatory; it is interesting to confront this with the Objective Position property of Objective abstractions: since Objectives are manually entered by the UAV operator, the Objective Position will not change even if the target moves, however the corresponding Entity Position will be updated (if detected at all) and thus the UAV will be able to correctly update its flight plan in order to intercept the moving target. The *Movement Info* property is used for moving Entities, so that the UAV can actually estimate the future position of the Entity (and thus avoid it or intercept it, depending on the mission).

Table 3.5. Entity properties.

ID	Property	Description
1	Entity Type	Type of entity (building, vehicle, weather zone, etc.)
2	Entity Tag	ID tag for entity
3	Entity Position	Most current position info for entity
4	Movement Info	Speed and direction of movement
5	Entity Behaviour	Friendly, Neutral or Hostile
6	Threat Level	Threat to the UAV, from negligible to catastrophic
7	Area of effect	Definition of area entities
8	Stance	Behaviour pattern for the UAV towards entity

The *Entity Behaviour* property is an indication of the expected behaviour of the Entity; this can be friendly, neutral or hostile, and the flight plan developed by SAMMS will take this into account. The *Threat Level* property indicates the potential threat that the Entity represents to the UAV; the threat levels are defined as in Table 3.3. The *Area of Effect* property is used when an Entity spans over an area rather than being a punctual object; for example, weather zones always span an area, and this area is defined by combining the Entity Position information with the Area of Effect information. Finally, the *Stance* property is a generic indication of the behaviour that the UAV should assume towards the Entity; examples of such behaviour are *ignore* (the flight plan is not influenced by the Entity), *avoid detection* (the flight plan is generated so as to remain farther than a known detection range from the Entity), *escape* (the UAV actively keeps clear of the Entity), *mission objective* (the Entity is a mission objective, thus it is part of the flight plan, but does not influence it in other ways). In fact, the Stance property was included to enable planning capabilities that have not yet been implemented; consequently, the Stance modes are not fully defined. It is important to underline that algorithms based on the Stance property could be easily implemented in the future, allowing the introduction of very specific features to the flight plan generation and replanning aspects of SAMMS (for example, a military UAV could be easily instructed to immediately run away from newly detected threats).

3.1.3 Action

The Objective and the Entity are abstractions that are used to provide to SAMMS all the information it needs in order to generate a flight plan. However, the flight plan itself needs to be expressed in computerized form, and for this reason a third type of abstraction is needed. This abstraction is the Action, which represents a high-level task, but at a lower level than the Objective. In general, an Objective will correspond to two or more Actions (an exception to this is the Transit Objective, which might possibly be achieved by a single Action, although the number of Actions can increase, for example to detour around a known danger area). The Action can be thought of as the finest flight plan subdivision which is relevant from a Mission Management point of view.

In order to be able to accomplish the five Objective types defined, a total of 12 Action types have been identified. Six Action types are related to the take-off and approach phases of flight; these are the *Park*, *Taxi*, *Take-off*, *Climb*, *Descent* and *Landing* Action types. Two further types (the *Main Mission Start* and *Main Mission End*) are "virtual" Actions: they are included in the flight plan to clearly divide the take-off and approach phases from the actual (main) mission.

Table 3.6. Action types.

Code	Definition	Description
1	Park	Wait until Mission Start time
2	Taxi	Move from initial position to runway position (or runway to final)
3	Take-off	Perform take-off manoeuvre
4	Climb	Climb to specified altitude
5	MMS	Main Mission Start
6	Travel	Travel to position along a loxodromic or orthodromic route
7	Recon	Perform Reconnaissance on target (part of an Analyze Target Objective)
8	Attack	Perform Attack on target (part of an Attack Target Objective)
9	Circle	Loiter about specified position (part of an Orbit Objective)
10	MME	Main Mission End
11	Descent	Enter and manoeuvre along a descent path
12	Landing	Perform landing manoeuvre

The remaining four Action types are the ones used while performing the actual mission. The *Travel* Action is by far the most important: this involves navigation from the current position to a desired destination, using either a loxodromic (rhumb line) route or an orthodromic (great-circle) route. A loxodrome (Alexander, 2004) is a point-to-point route that always maintains the same magnetic bearing; it is not the shortest route, but is very easy to maintain, both for a human pilot or an autopilot (it is sufficient to manoeuvre the aircraft so that the magnetic compass is on a fixed heading). An orthodrome (Bowditch, 1802) is instead the point-to-point route that covers the shortest distance between the points; considering Earth spherical, this occurs along great circles (the intersection between the sphere and a plane which passes through the centre point of the sphere); the downside is that such a route is characterized by a constantly changing bearing, making it difficult to maintain from a control point of view. In reality, when not forced otherwise by ATC restrictions or other considerations, aircraft usually fly along a decomposition of the orthodrome, formed by several loxodromic segments. However, within SAMMS a Travel Action is carried out by travelling along a purely orthodromic route. Finally, the *Recon*, *Attack* and *Circle* Action types are used to perform manoeuvres during the corresponding Objective types (respectively, analyze target, attack target and orbit position). The flight patterns used during these manoeuvres will be described in detail in Chapter 4. Table 3.6 summarizes the Action types defined within SAMMS.

Like the Objective and Entity abstractions, Actions are defined as structures that are formed by a set of parameters (or properties). These parameters are listed in Table 3.7. Some parameters are common to all Action types, while others are relevant only to certain types (the table shows this, using the Action type codes from Table 3.6). Further clarification for some properties is needed.

The *Sequence* property will be explained shortly in this section, when introducing the concept of flight plan. The *Time* property is used to define time limits after which the Action should be considered as accomplished. The *Heading* property is used during the take-off and approach phases and normally indicates the runway bearing that is to be maintained during these phases. The *Duty* property is used in two ways: for Circle Actions, it reports the Duty property defined by the parent Objective; however, for other Action types, this is used to indicate the type of the parent Objective (note that for Circle Actions, the parent Objective type will obviously be an Orbit Objective). Finally, the *Objective* property is used to identify the parent Objective of the Action; the Objective Tag properties are used.

Table 3.7. Action properties.

ID	Definition	Description	Relevance
1	Action Type	One from Table 3.6	All
2	Sequence	Sequence number for the Action	All
3	Start Position	Initial position for certain Action Types	2, 6
4	Position	Position coordinates relevant to Action	All
5	Time	Time properties of Action (usually time limit)	1, 9
6	Heading	Bearing to be kept for certain Action Types	3, 4, 11, 12
7	Altitude	UAV Altitude specified for Action	4, 6, 7, 8, 9, 11, 12
8	Duty	Duty type for Circle Actions, also used to indicate parent Objective type	All
9	Speed	UAV Speed for Action	6, 7, 8, 9
10	Target	Defines a specific target for Recon and Attack	7, 8
11	Objective	Parent Objective ID tag, for Actions related to take-off and approach phases a code is used	All

Having defined the Action abstraction, it is possible to define what a flight plan is in SAMMS: a flight plan is an ordered sequence of Actions. The *Sequence* property of Actions is thus explained: it indicates the sequence number assigned to the Action within the flight plan. Considering an example scenario where a single Transit Objective is assigned to the UAV, the flight plan would then be formulated as the following sequence of Actions:

- Park, sequence number 1; the UAV is waiting for the commanded mission start time
- Taxi, sequence number 2; the UAV taxies to the runway
- Take-off, sequence number 3; the UAV takes off and clears 50 ft altitude
- Climb, sequence number 4; the UAV establishes positive rate of climb until it reaches the commanded altitude
- Main Mission Start, sequence number 5; virtual Action indicating that mission Objectives are now beginning to be executed
- Travel, sequence number 6; a Transit Objective instructs the UAV to travel and land at a destination airport, and this leads to a single Travel action from the starting airport to the destination one
- Main Mission End, sequence number 7; virtual Action indicating that mission Objectives have been fulfilled

- Descent, sequence number 8; the UAV enters an approach path in preparation for landing, climbing down to a specified altitude and getting in line with the airport runway
- Landing, sequence number 9; the UAV performs the final phases of landing, including flare and roll-out, until it stops completely
- Taxi, sequence number 10; the UAV taxies from the runway to the designated parking position
- Park, sequence number 11; the UAV signals "mission accomplished" to the operator and waits for further instructions (that would begin a new mission)

A SAMMS flight plan always begins with the same five Actions that form the take-off phase (the Action parameters will obviously be different from time to time); these are then followed by the main mission Actions, which are originated by the actual mission objectives; finally, the flight plan is always concluded by the same five Actions (that form the approach phase).

3.1.4 SAMMS operation

Having defined the abstractions needed by SAMMS, it is interesting to review how SAMMS could be operated in a routine mission:

- the operator assigns a set of Objectives
- additional information is gathered and then converted into Entities
- a flight plan (ordered sequence of Actions) is generated on the basis of Objectives and Entities
- the mission starts
- as the mission is in progress, the flight plan is adjusted in response to (significant) changes in the Objectives or Entities
- at some point, the UAV lands and the mission is then completed

It is possible to notice the similarity between this sequence of events and the example mission scenario delineated in Section 1.1; in fact, the abstractions have been introduced in order to allow a univocal computerization of such a scenario.

3.2 Architecture overview

As a system, SAMMS has to accomplish a wide variety of tasks: not only it must generate a flight plan from available information (Objectives and Entities), but then it must be capable to execute and update it as necessary.

The mission management activity, which consists of the development and constant update of the flight plan, is very different in nature from the actual operation of the UAV. The algorithms that perform these separate functions are just as different: even Soar intelligent agents, which are based on the same technology, can present significant differences, and this is all the more true for external non-Soar components.

For this reason, during the development of SAMMS careful thought had to be spent on designing an overarching architecture so that functions to be performed could be divided between clearly defined components. The adopted architecture is shown in Figure 3.2, and is based on a set of three Soar intelligent agents, plus various external components. While the Soar agents and other components will be thoroughly described over the rest of this chapter and the next three chapters, it is important at this stage to clearly define the subdivision of functions between each component.

The first (and probably most important) component is the *Planner Agent* (or simply *Planner*). This is a Soar intelligent agent that is tasked with the generation of the flight plan for the UAV. It performs the most complex function within SAMMS: on the basis of Objectives received from the Operator and Entities received from some sensor/communication system (and representing its knowledge of the environment), the Planner must formulate a flight plan (as defined in Section 3.1, an ordered sequence of Actions) that tries to accomplish all Objectives within the defined parameters. As will be described, several algorithms are used within the Planner in order to generate a flight plan that is not only viable, but also reasonable. While the flight plan is not guaranteed to be "optimal" in a mathematical sense, it is generated so as to incorporate desirable

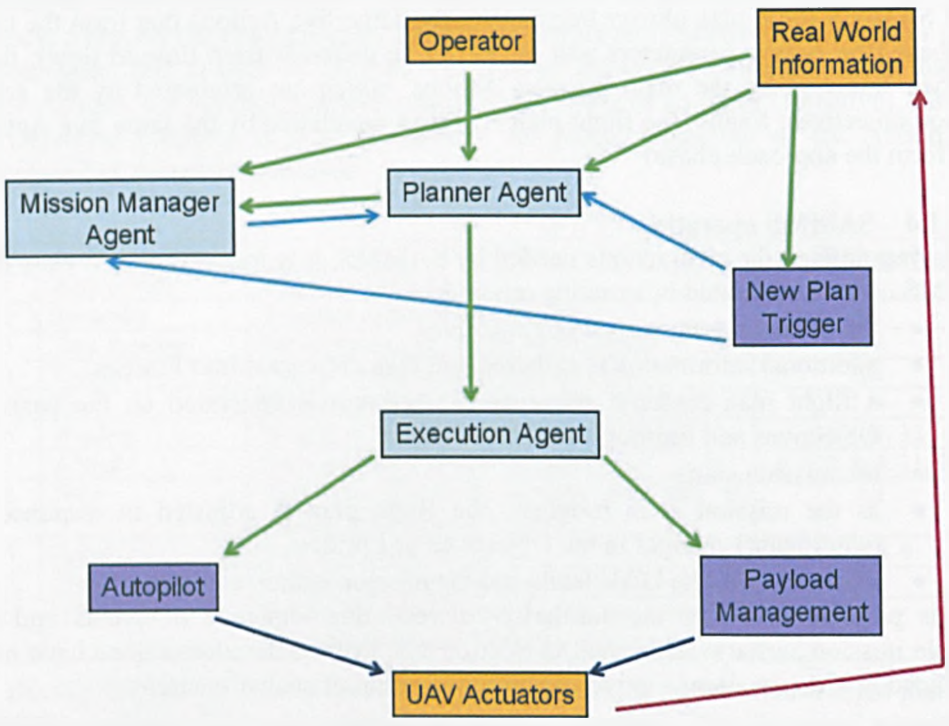


Figure 3.2. Architecture overview.

characteristics: the distance to be covered is minimized (as possible when time constraints are placed on Objectives), detours around risky or no-fly zones are planned, speed and altitude are adjusted according to current mission requirements, and in general a "no-nonsense" policy is pursued. The Planner agent also performs the re-planning function, updating the flight plan during the mission when necessary. For this reason, timely generation of a flight plan is essential; Soar intelligent agents have proved to be very efficient in this sense, since the plan generation process does not require more than 0.5 sec even for very complex missions with as many as ten Objectives (these times refer to experiments performed within the Simulink simulation environment and a standard Windows PC; using dedicated hardware and software, such as a PC/104 board running a real-time operating system, will probably lead to even shorter computation times).

A very important consideration to be made regarding the Planner agent is that the generated flight plan is designed to accomplish all of the Objectives; if an inconsistency is detected (for example, time priority constraints of the Objectives cannot be respected, estimated fuel consumption is greater than the available fuel, or an Objective is placed within the area of effect of a known threat), the Planner registers the inconsistency, if possible tries to minimize its effect (for example, increasing flight speed to respect time constraints, or decreasing it to save fuel), but in general maintains the plan as it is, without major changes. In more theoretical terms, this means that the Planner agent does not have the authority to change mission Objectives. This authority resides within the second SAMMS component, which is called *Mission Manager Agent* (or *MMA*).

The MMA is also a Soar intelligent agent and its functions overlap with those of the Planner. In short, the MMA is tasked with dealing with inconsistencies that are detected within the flight plan generated by the Planner. It does so by changing, removing or adding Objectives to the list of Objectives provided by the UAV operator. This is the reason for the physical separation between the Planner and the MMA: the authority to change Objectives is a controversial point. On one side, the ability to change mission Objectives is desirable; in fact, human pilots can decide to abort an Objective, if the situation demands it. On the other side, this represents a significant step in the realm of autonomy, and it could be claimed that it severely impacts the determinism and predictability of the system.

In fact, every decision made by the MMA is motivated by a change in the current situation, but a lack of determinism arises since in a realistic environment the situation can change unpredictably. It is clear that giving a UAV this degree of autonomy means that it can more flexibly perform its mission; however, this may not be desirable from the operator point of view, since it represents a loss of control over the UAV. The issue of how much autonomy should be granted to UAVs (or autonomous systems in general) is almost philosophical in nature and has long been debated (for example, see Haselager, 2005; Sharkey, 2008). Within SAMMS, a decision was made to separate these two levels of functionality and autonomy; while the MMA brings significant functionality to the system, it is entirely possible to completely disregard it, without losing the capabilities of the other components.

The third SAMMS component is called the *Execution Agent* (or *Exag*). It is also a Soar intelligent agent, albeit one which is radically different from the Planner and MMA. The function of the Exag is to act as an intermediate layer between the Planner and low-level control of the UAV. In fact, while a flight plan in the form of an ordered sequence of Actions is meaningful from a mission management point of view, it is meaningless for a traditional low-level flight control system (e.g., an autopilot). The task of the Exag is to translate Actions into low-level commands that can be actuated by the underlying control system. In practice, the Exag goes through all the Actions in the flight plan, one by one, and for each it computes the precise commands that need to be fed to the autopilot and payload management systems.

Operationally, the Exag is radically different from the Planner and MMA. The main difference lies in the timing: Exag operation is basically continuous, while the Planner and MMA operate in cycles; they will be waiting most of the time, until a full cycle (flight plan generation for the Planner, flight plan checking for the MMA) is needed.

In fact, a dedicated SAMMS component is used to command the execution of cycles for the Planner and MMA. This is called *New-Plan Trigger*, and is implemented using normal Simulink blocks. The New-Plan Trigger function separately commands operation of both the Planner and MMA. In practice, the New-Plan Trigger function continuously monitors current Objectives, Entities and flight plan, triggering the execution of a Planner or MMA cycle when a change is detected. Since minor changes can be expected very often (especially for Entities), it is paramount to implement the system so that only significant changes will cause the triggering of a new cycle: otherwise, the system may be bogged down by continuous unneeded re-planning. In fact, this component is critical in defining the reactive versus pro-active characteristics of SAMMS (as depicted in Section 2.2).

Another component of SAMMS is the *Autopilot*, which includes all "low-level" control functionality. This is not different from a traditional autopilot: its basic function is to maintain the desired attitude and speed (internal loops), with additional functionality providing navigation (along great-circle routes) and altitude hold capabilities (external loops). It is entirely implemented within Simulink.

A *Payload Management* component is present in the architecture, however this was not implemented (since it is largely dependent on the availability of an actual system). It is important to note that, in the prospect of a real-world implementation of SAMMS, a Payload Management subsystem would need to be developed, so as to both provide sensor feedback for the entire system and manage the payload in order to accomplish mission Objectives.

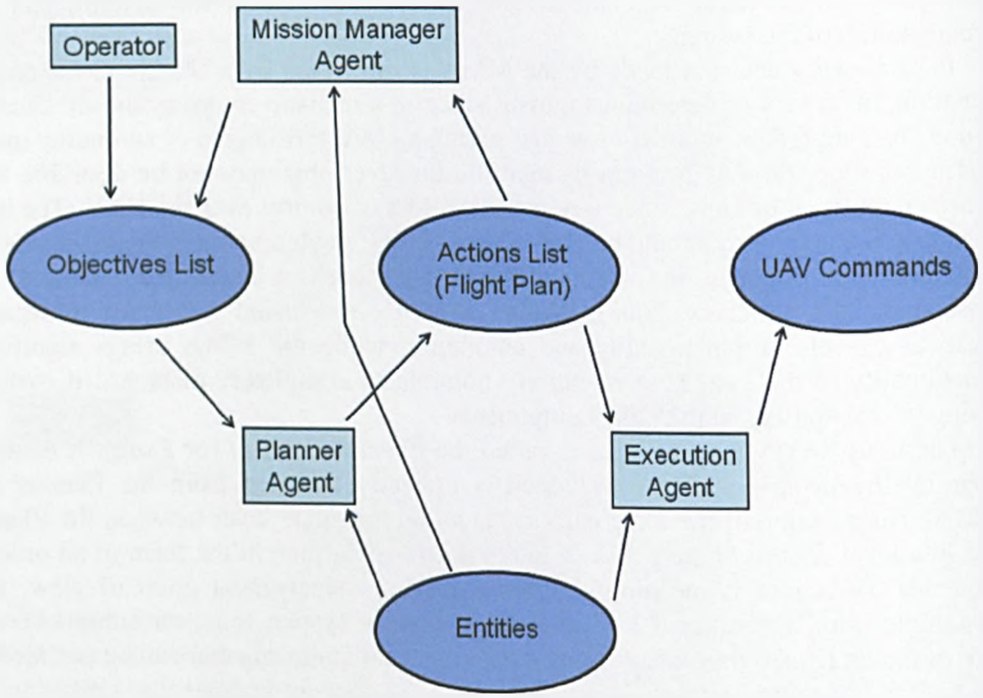


Figure 3.3. SAMMS flow diagram.

Finally, while it is not strictly part of SAMMS as a system, the UAV mathematical model used throughout the development for validation and testing purposes can also be considered a key component. This is only needed during the development phase, however its importance should not be considered. Without a reference UAV model, the Autopilot component could not have been implemented. While Planner and MMA functionality can be tested without this, the Exag can only be properly validated when operating in conjunction with its target Autopilot system.

In conclusion, it is interesting to see how the abstractions described in Section 3.1 fit within the SAMMS architecture described in the present section. To do this, let us consider the flow of information within SAMMS, as depicted in Figure 3.3. First, the operator gives instructions to the UAV, in the form of a list of Objectives; the Planner Agent then translates this into a flight plan (a list of Actions). The Mission Manager Agent acts checks for inconsistencies in the flight plan (or between the flight plan and the Entities), and then eventually modifies the Objective list (thus causing the generation of an updated flight plan). Finally, the Execution Agent translates the flight plan into appropriate commands that can be actuated by the UAV autopilot system.

This flow of operation is adopted throughout SAMMS. The Soar intelligent agents (Planner, Exag and MMA) will be described separately in the next chapters, but they should always be thought as operating within this structure, so that their operation is interlinked in a constructive manner.

3.3 Simulation Environment

In Section 2.3, the choice of Soar intelligent agents as the base technology for SAMMS was motivated. It was noted that Soar agents need to operate within an appropriate framework in order to fully take advantage of their characteristics. The Matlab/Simulink package was chosen for the development of this framework, for these main reasons:

- it allows seamless integration of external components (Soar agents, in our case)
- most traditional control techniques can be modelled with it (including fuzzy-logic and neural networks), opening up possibilities for integration of a variety of externally (non-Soar) provided capabilities
- it provides a simulation environment for validation of control software
- it allows rapid prototyping of control systems via the automatic generation of code (Real-Time Workshop)

In this section, the simulation environment developed for SAMMS will be accurately described; some components, and specifically the UAV model and the autopilot, will be described in Section 3.4 and Section 3.5 respectively.

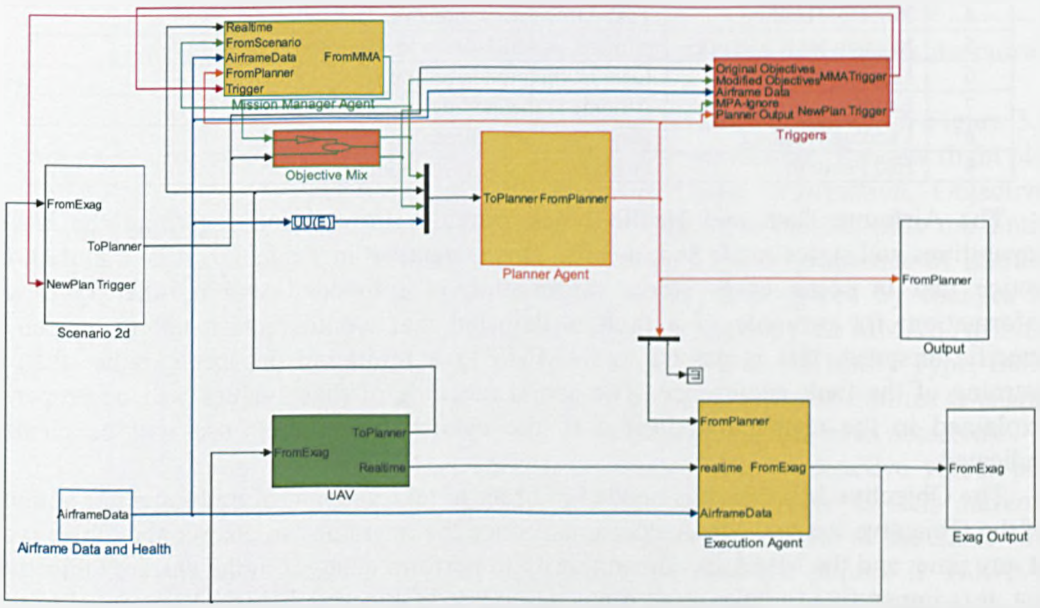


Figure 3.4. SAMMS top level Simulink diagram.

Figure 3.4 shows the top level of the Simulink model for SAMMS; it is possible to notice the various blocks and their connections. From left to right, and top to bottom:

- the white blocks on the left, *Scenario* and *Airframe Data and Health*, are used to provide inputs to SAMMS
- the yellow block on the top left encapsulates the *Mission Manager Agent*

- the orange *Objective Mix* block is used to incorporate changes to the Objective list performed by the MMA
- the central yellow block encapsulates the *Planner Agent*
- the green block in the bottom left contains the *UAV* mathematical model
- the orange block on the top right is the *New Plan Trigger* function
- the white blocks on the right, *Output* and *Exag Output*, are used for real-time analysis of the (very large) amount of data which the agents provide as output

The Scenario block contains information that is used by both the Planner and the MMA. In particular, three sets of data are contained: base airport information, the Objectives list and the Entities list. The base airport data contains various types of information that are needed during the take-off and approach phases, as detailed in Table 3.8. The Objectives and Entities lists are just the collection of all external input that is provided to the UAV using the abstractions described in Section 3.1. The three data sets are grouped into the Scenario block for ease of use; since together they entirely define the external situation, they can be seen as defining the scenario into which SAMMS is operating. As will be seen in Chapter 4, several example scenarios were developed for the validation of SAMMS; each scenario is constituted by base airport data, Objectives and Entities, and grouping these together allows to quickly change between scenarios.

Table 3.8. Scenario airport information.

Code	Definition	Description
1	Parking Position	Initial position of the UAV
2	Mission Start Time	Time at which mission begins (from Park to Taxi Action)
3	Runway Position	Ideal position for the Take-Off manoeuvre
4	Runway Heading	Take-Off runway direction
5	Mission Start Altitude	Altitude at which Main Mission phase should begin
6	Landing Position	Ideal landing touch position
7	Landing Altitude	Altitude at the landing airport
8	Landing Heading	Landing runway direction
9	End Position	Position to taxi to after landing

The Airframe Data and Health block provides information regarding the UAV capabilities and status to all Soar agents. This is detailed in Table 3.9. It is important to notice that in some cases status information is embedded within other types of information: for example, if a fault is detected that would require not to exceed a specific airspeed, this is passed to SAMMS as a limitation on speed, rather than a warning of the fault occurrence. The actual meaning of these values will be properly explained in the chapters dedicated to the agents, where their use will be clearly indicated.

The Objective Mix block is needed in order to take account of both operator actions of the Objective list and MMA decisions. Since the operator can change the Objectives at any time, and the MMA has the authority to perform changes to the current Objective list, it is imperative to have at all times an updated Objective list to be presented to the Planner. This block performs this function, by correctly updating the Objective list as the mission is being accomplished.

The New Plan Trigger function is used to trigger flight plan generation in the Planner Agent and flight plan checking in the Mission Manager Agent. As stated in Section 3.2, the Planner and MMA operate in cycles, with a full cycle being performed when needed. It is the New Plan Trigger function that signals the need for one such

Table 3.9. Airframe Data and Health.

ID	Type	Definition
1	Control Status	Pitch control status
2		Roll control status
3		Yaw control status
4	Speeds	Maximum speed
5		Cruise speed
6		Minimum (stall) speed
7		Optimal speed
8		Take-off rotation speed
9		Landing speed
10		Ground Manoeuvre speed
11		Ground Movement speed
12		Current status
13		Current speed limitation
14		Search speed
15		Analyze speed
16		Attack speed
17	Altitude	Maximum altitude
18		Cruise altitude
19		Minimum ground altitude
20		Climb turn altitude
21		Descent altitude
22		Pre-land altitude
23		Flare altitude
24		Current altitude limitation
25		Search altitude
26		Analyze altitude
27		Attack altitude
28	Distance and angles	Recon distance
29		Attack distance
30		Circle distance
31		Descent distance
32		Pre-land distance
33		Take-off angle
34		Climb angle
35		Flare angle
36	Failures	Loss of thrust control
37		Structural damage
38		Loss of power
39		Loss of GS connection
40		Navigation failure (GPS)
41		Navigation failure (IMU)
42		Loss of autopilot
43	Payload failures	Analyze target
44		Attack target
45		Search target
46		Orbit Duty 1
47		Orbit Duty 2
48		Orbit Duty 3
49	Fuel and range	Maximum Fuel
50		Remaining Fuel
51		Consumption constant
52		Minimum speed adjust
53		Maximum speed adjust
54		Fuel system status

cycle to be executed (e.g. the need to generate a new flight plan, or to check the current flight plan for inconsistencies).

The Simulink diagram for the New Plan Trigger function is shown in Figure 3.5. Triggering for the Planner and the MMA is separate. The generation of a new flight plan (by the Planner) is triggered by changes in: airport data information, Objectives (including changes from the MMA) and Airframe Data and Health, plus the Entity Movement, Entity Behaviour, Threat Level and Area of Effect properties of Entities. The consistency check of a flight plan (by the MMA) is triggered by changes in: Objectives (as provided by the operator, e.g. excluding changes by the MMA), Airframe Data and Health and the availability of a new flight plan, plus the Entity Type, Entity Tag, Entity Behaviour, Threat Level and Area of Effect properties of Entities (a change in Entity Type and Entity Tag basically means that a new Entity has been detected).

As stated earlier, this function is critical in determining the reactive versus pro-active characteristics of the entire system. The New Plan Trigger as it is currently implemented tries to limit the number of re-planning events while at the same time ensuring such events when significant changes in the situation require it. However, there is always a trade-off between reactivity and pro-activity, and finding the best solution would require more advanced stages of production, since it depends on the available on-board computing resources and the type of mission that is to be performed. For example, a military UAV might require very high reactivity towards specific occurrences (e.g., the detection of a new threat), so it would require tailoring of the New Plan Trigger function to meet these demands.

Finally, the Output blocks allow to monitor in real-time results produced by the Planner Agent and the Execution Agent. The *Plan Descriptor* function in particular is

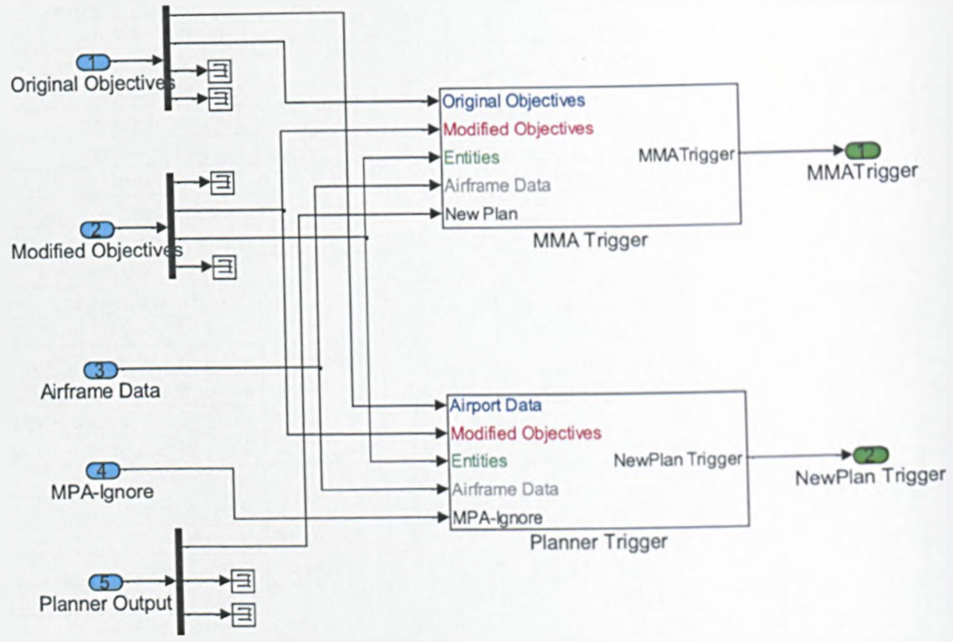


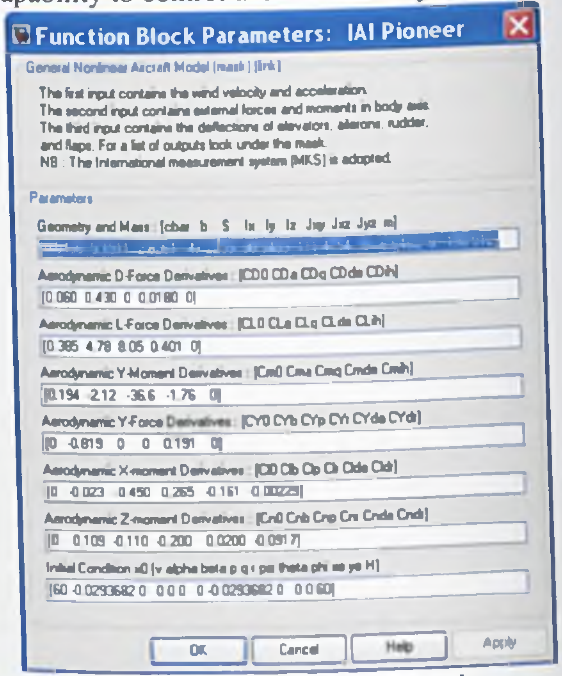
Figure 3.5. New Plan Trigger block diagram.

useful, as it allows immediate visualization of a flight plan. This function will be thoroughly described and widely used in Chapter 4, which is dedicated to the Planner Agent.

3.4 UAV Model

In order to accomplish the goals outlined in Section 1.2.3, SAMMS should reach a development point where it possesses the capability to control a UAV not only through high-level commands, but also using low-level commands. In practice, this means that the high-level reasoning layer of SAMMS must be proved to be capable of interacting with traditional low-level control algorithms (autopilots) and thus with real-world actuators.

Since the aim for the present PhD project is to keep SAMMS limited to the simulation stage of implementation, the development of a valid UAV mathematical model (and associated autopilot model) is imperative. In this section, the UAV model used throughout this project will be described thoroughly. In a real system the autopilot is part of the control software, unlike the UAV model (which is only used for simulations).



Function Block Parameters: IAI Pioneer

General Nonlinear Aircraft Model (mask) (link)

The first input contains the wind velocity and acceleration.
The second input contains external forces and moments in body axis.
The third input contains the deflections of elevators, ailerons, rudder, and flaps. For a list of outputs look under the mask.
NB : The International measurement system (MKS) is adopted.

Parameters

Geometry and Mass : [cbar b S Ix Iy Iz Jxy Jxz Jyz m]

Aerodynamic D-Force Derivatives : [CD0 CDa CDq CDde CDih]
[0.060 0.430 0 0.0180 0]

Aerodynamic L-Force Derivatives : [CL0 CLa CLq CLde CLih]
[0.385 4.78 8.05 0.401 0]

Aerodynamic Y-Moment Derivatives : [Cm0 Cma Cmq Cmde Cmih]
[0.194 -2.12 -36.6 -1.76 0]

Aerodynamic Y-Force Derivatives : [CY0 CYb CYp CYr CYda CYdr]
[0 -0.819 0 0 0.191 0]

Aerodynamic X-moment Derivatives : [Cl0 Clb Clp Clr Clda Cldr]
[0 -0.023 -0.450 0.265 -0.161 -0.00229]

Aerodynamic Z-moment Derivatives : [Cn0 Cnb Cnp Cnr Cnda Cndr]
[0 0.109 -0.110 -0.200 0.0200 -0.0917]

Initial Condition x0 [v alpha beta p q r psi theta phi xe ye H]
[60 -0.0293682 0 0 0 0 0 -0.0293682 0 0 0 60]

Figure 3.6. IAI Pioneer model mask.

However, the UAV model is described earlier in this thesis, since the autopilot is a direct consequence of the characteristics of the UAV model.

Several observations must be made regarding the requirements for the UAV model:

- the intention is to validate the SAMMS architecture and agents through simulations; since these operate at a rather low frequency, and since they are very generic in nature, a high-fidelity simulation was deemed unnecessary
- while the SAMMS architecture is not specifically designed for a particular UAV type, it is generally developed for deployment on low-weight low-cost UAVs (see Section 1.2), so the model of such a UAV should be favourable over the model of a more sophisticated UAV
- the development of UAV and, in general, aircraft simulation models is a complex task, often involving large expenses (especially for high-fidelity simulations); this is clearly outside the scope of the SAMMS project, therefore rather than developing on-purpose a full model, a pre-existing model was sought, and then adapted to the project's needs

In 1994, Marc Rauw, then a student at the Delft University of Technology, developed and released the Flight Control Dynamics toolbox for Simulink (Rauw, 2003). This toolbox is meant to be integrated within the Matlab/Simulink package and

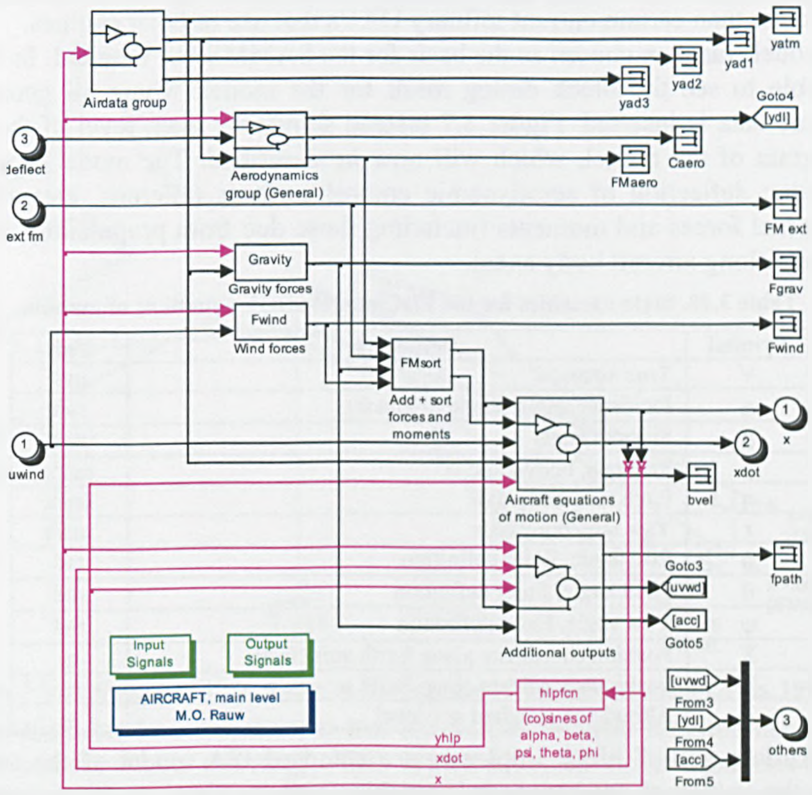


Figure 3.7. IAI Pioneer model main level, as in Airlib.

includes several tools which are useful in implementing flight simulation, flight dynamics analysis and flight control system design:

- a full non-linear model of the DeHavilland DHC-2 Beaver aircraft
- a steady-state flight trimming utility
- an aircraft model linearization tool
- radio navigation models

- wind and turbulence models
- a complete implementation of a classical autopilot system for the Beaver aircraft

While the entire toolbox can potentially be useful for the development of SAMMS, it is particularly the full aircraft model that attracts the attention. This model has a very modular design and the entire package is open-source software, hence it is an ideal starting point for the development of the UAV model for SAMMS.

In 2003, Giampiero Campa, then at West-Virginia University, released his Airlib library for Simulink (Campa, 2003). The core of this library is a set of various aircraft models, which are all derived from the Beaver model, but obviously use different data sets regarding the aerodynamic, inertial and propulsive characteristics. The library contains 13 different aircraft models, including the IAI Pioneer, which is one the first operational UAVs (developed in 1986 for the US military). The Pioneer is a relatively small UAV, weighing 205 kg and having a wingspan of 5.2 m; its top speed is 110 knots, and it can reach a ceiling altitude of 4600 m. Due to these characteristics, it was deemed representative of the category of UAVs which is targeted by SAMMS: it is a rather small UAV, and while its cost was certainly high at the time, it is based on technology that at present would be significantly cheaper. In particular, the fact that it uses a propeller and piston engine propulsion system places it in cost category which certainly lower than certain current military UAVs that use turbojet engines.

This model was then chosen as the basis for the SAMMS UAV model. In Figure 3.6, it is possible to see the block dialog mask for the model, where all geometric and aerodynamic data is inserted. Figure 3.7 instead shows the main level of the Simulink block diagram of the model, which will now be described. The model receives three input vectors: deflection of aerodynamic control surfaces (ailerons, elevator, rudder, flaps), external forces and moments (including those due from propulsion), wind speed (components along aircraft body axes).

Table 3.10. State variables for the FDC model 6DOF equations of motion.

Symbol	Definition	Unit
V	True airspeed	m/s
α	Incidence angle (angle of attack)	rad
β	Sideslip angle	rad
p	Roll rate, body-axis	rad/s
q	Pitch rate, body-axis	rad/s
r	Yaw rate, body-axis	rad/s
ϕ	Roll angle, Euler definition	rad
θ	Pitch angle, Euler definition	rad
ψ	Yaw angle, Euler definition	rad
X	Position coordinate along Earth axis North	m
Y	Position coordinate along Earth axis East	m
Z	Altitude on standard sea-level	m

The "Airdata group" block implements a standard ISA model of the atmosphere, outputting the values of environmental variables such as air pressure, air temperature, air density, air viscosity and local gravity acceleration. Derived atmospheric variables are also calculated, such as speed of sound, Mach number, dynamic pressure, total temperature and Reynolds number. The "Aerodynamics group" block calculates the current aerodynamic forces and moments on the aircraft, using a linear model based on the aerodynamic derivates provided (as in Figure 3.6). The forces and moments depend on current flight conditions (the state vector, described later), atmospheric conditions and control surfaces deflection. The "Gravity forces" and "Wind forces" blocks are selfdescriptive, while the "FMsort" block is used to add all the forces and moments (aerodynamic, gravitational, wind and propulsion).

The most important block is certainly the "Aircraft equations of motion" block, which implements the solution of the aircraft equations of motion in order to calculate the position, attitude and velocities of the aircraft. The equations used are the standard rigid-body six degrees-of-freedom (6DOF) equations, which are thoroughly treated in aeronautical literature (see for example (Nelson, 1989)). Within the FDC toolbox (and thus also the Airlib models), the equations of motion are formulated using the 12 state variables in Table 3.10.

Several modifications were needed in order to adapt this model to the needs of the SAMMS project. In particular, a propulsion model and the possibility to handle ground operations were added to the Pioneer model. The propulsion model is implemented as a simple transfer function between the throttle command and the actual thrust provided.

Extensive modifications were instead required to allow for ground operations modelling. A detailed gear model is outside the scope of this simulation, so many simplifications were used. Ground reaction forces and moments were assumed to be equal and contrary to the sum of the other forces and moments to which the aircraft is subject; this is true only for certain axes, specifically the Y and Z axes for the forces and the X and Y axes for the moments. In practice, this means that when grounded the

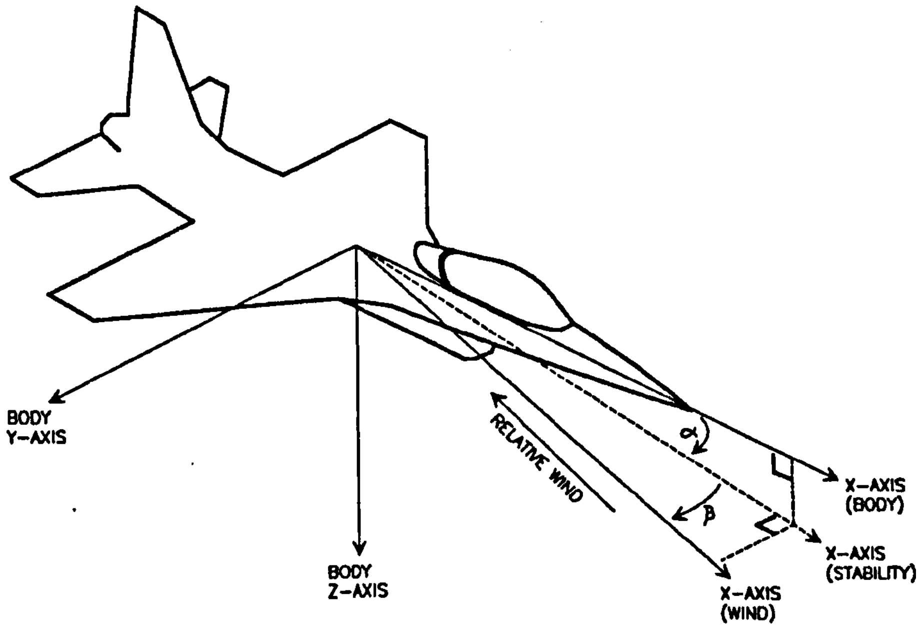


Figure 3.8. Definition of *V*, *α*, *β*, *u*, *v*, *w* (Reproduced from (Stevens and Lewis, 1992)).

aircraft is assumed to be constrained to two types of movement: longitudinal motion and rotation around the Z axis (steering). This is also reflected in the equations of motion. However, these changes are not sufficient for satisfactory ground operations. In fact, the equations of motion as formulated in the Pioneer model cannot be used for ground operations because of problems arising when the airspeed in close to zero: in this situation, the values of α and β can present discontinuities, hence causing problems in calculating the state vector. To solve this problem, the equations had to be reformulated (at least those concerning the aircraft translational speeds).

There are two options when expressing aircraft translational speeds: since this is in fact a 3D vector, the first option is to provide the vector value (in our case, the airspeed

V) plus the two angles of rotation from the reference axes (in our case, α and β). The second option is to provide the scalar components along the reference axes; in aeronautical engineering, these components are usually called *u*, *v* and *w*, representing respectively the projection of V along the X, Y and Z axes. Figure 3.8 shows how the airspeed *V* and its components are defined: the airspeed is by definition the speed vector along the X wind axis, which is obtained from the X body axis by two rotations of α and β respectively; *u*, *v* and *w* are instead the projections of V on the three body axes.

The equations of motion adopted are then the following (Rauw, 2005):

$$\dot{u} = \frac{F_X}{m} - qw + rv$$
$$\dot{v} = \frac{F_Y}{m} + pw - ru$$
$$\dot{w} = \frac{F_Z}{m} - pv + qu$$

$$\dot{p} = P_{pp}p^2 + P_{pq}pq + P_{pr}pr + P_{qq}q^2 + P_{qr}qr + P_{rr}r^2 + P_lL + P_mM + P_nN + p'$$
$$\dot{q} = Q_{pp}p^2 + Q_{pq}pq + Q_{pr}pr + Q_{qq}q^2 + Q_{qr}qr + Q_{rr}r^2 + Q_lL + Q_mM + Q_nN + q'$$
$$\dot{r} = R_{pp}p^2 + R_{pq}pq + R_{pr}pr + R_{qq}q^2 + R_{qr}qr + R_{rr}r^2 + R_lL + R_mM + R_nN + r'$$

where *u*, *v*, *w*, *p*, *q*, *r* are the state variables as defined earlier; F_X, F_Y, F_Z, *L*, *M*, *N* are the total forces and moments on the aircraft (with respect to body axes); P_{pp}, P_{pq}.... R_n are inertia coefficients, which are derived from the matrix multiplications involving the inertia tensor; *p'*, *q'*, *r'* express the effects of the gyroscopic couples.

The equations of motion are complemented by the kinematic equations that allow calculation of the other state variables:

$$\dot{\psi} = \frac{q\sin\varphi + r\cos\varphi}{\cos\theta}$$
$$\dot{\theta} = q\cos\varphi - r\sin\varphi$$
$$\dot{\varphi} = p + (q\sin\varphi + r\cos\varphi)\tan\theta$$
$$\dot{X} = [u\cos\theta + (v\sin\varphi + v\cos\varphi)\sin\theta]\cos\psi - (v\cos\varphi - v\sin\varphi)\sin\psi$$
$$\dot{Y} = [u\cos\theta + (v\sin\varphi + v\cos\varphi)\sin\theta]\sin\psi + (v\cos\varphi - v\sin\varphi)\cos\psi$$
$$\dot{Z} = -u\sin\theta + (v\sin\varphi + v\cos\varphi)\cos\theta$$

The current state vector is fed through these equations to obtain the state variable derivatives for the next iteration; the derivates are then integrated in order to calculate the new state vector. The values of *V*, *α* and *β* are then derived from *u*, *v*, *w*, since they are needed in the Aerodynamics block.

3.5 Autopilot model

The main function of the SAMMS autopilot is to provide low-level control functionality. As can be seen in Figure 3.3, it receives commands from the Execution Agent (Exag) and transforms them into actual commands for the UAV actuator system.

The Pioneer UAV model, depicted in Section 3.4, is controlled using a throttle command and traditional three-axis aerodynamic control surfaces (ailerons, elevator and rudder). An additional control variable is represented by the Ground Steering command.

Table 3.11. Exag output to the Autopilot subsystem.

Code	Type	Definition	Description
3.1.1	Direct commands	Direct Speed	Speed target for the Autopilot in Direct mode
3.1.2		Direct Pitch	Pitch target for the Autopilot in Direct mode
3.1.3		Direct Yaw	Yaw (bearing) target for the Autopilot in Direct mode
3.1.4		Direct Roll	Roll target for the Autopilot in Direct mode (unused)
3.1.5		Brakes	Activation of brakes (airbrakes and ground brakes)
3.2.1	Auto commands	Auto Speed	Speed target for the Autopilot in Auto mode
3.2.2		Auto Altitude	Altitude target for the Autopilot in Auto mode
3.2.3		Initial Position	Initial position of a segment in Auto mode
3.2.4		End Position	Final position of a segment in Auto mode
3.3		Direct/Auto	Switch between Direct and Auto modes

Ultimately, the function of the autopilot is to translate commands received from the Exag into commands in the form of throttle demand and control surfaces deflection. Thus, the first thing to be explained is the format of commands outputted by the Exag.

The Exag is thoroughly described in its dedicated chapter, together with its input/output interface. Only part of the Exag output is relevant to the Autopilot subsystem, as listed in Table 3.11. To understand this, it is important to define the two main operating modes for the SAMMS Autopilot. The Direct mode is the simplest Autopilot mode: three main commands are received from the Exag, a Speed target, a Pitch target and a Bearing target. In Direct mode, the Autopilot manoeuvres the UAV so

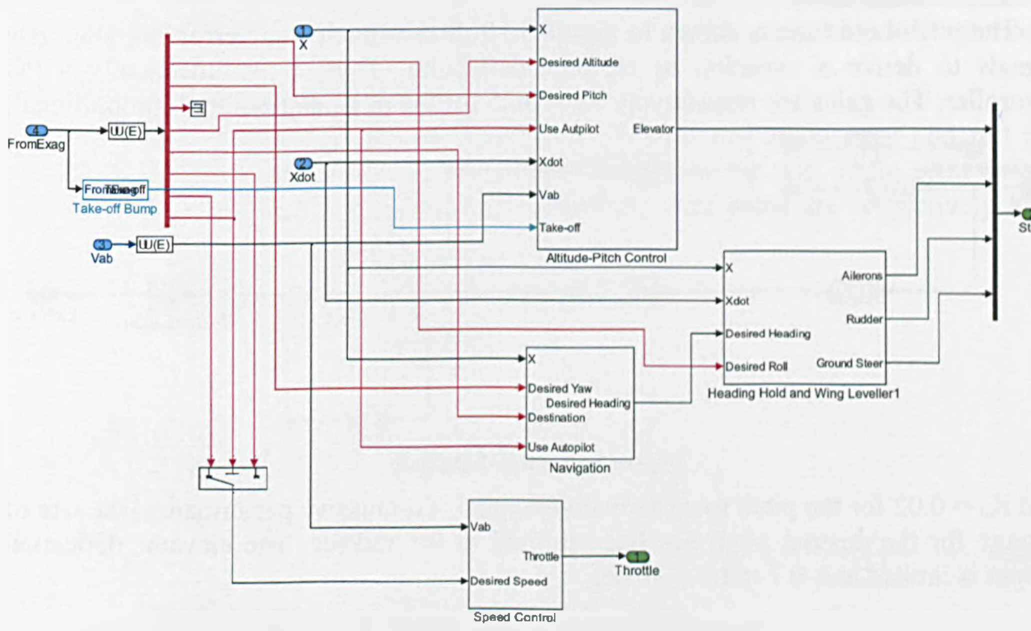


Figure 3.9. Simulink block diagram of the Autopilot.

as to keep it at the desired values of speed, pitch angle and magnetic bearing. The Direct mode is supplemented by the Auto mode, which basically represents the addition of external control loops on top of the ones used for the Direct mode. In Auto mode, the Exag provides target values for the desired speed and altitude, and specifies the aircraft trajectory by indicating an initial and a final position for the current navigational segment.

The Exag commands the Autopilot to switch between these two modes using the appropriate output value (code 3.3 in the table). In general, the commanded mode depends on the type of Action being performed. The Direct mode is used during the take-off and approach phases, thus for Park, Taxi, Take-off, Climb and Landing Action types. The Auto mode is instead used for the main-mission phases, thus for the Travel, Recon, Attack and Circle Action types, and additionally for the Descent Action.

Figure 3.9 shows the Simulink block diagram of the Autopilot subsystem. It is divided into four main parts:

- altitude-pitch control, which provides elevator commands and is implemented with two separate control loops (a pitch-hold loop and an altitude-hold loop which is placed on top of the other)
- heading hold and wing leveller, which mainly provides aileron commands and also rudder and ground steer commands; there are two main loops (a roll-hold loop and a heading-hold loop on top of it), plus the ground steer loop (which is functionally similar to the heading hold loop but does not use the roll-hold loop)
- navigation, which is only used in Auto mode, providing a desired heading to the heading hold block
- speed control, implementing a single loop that provides throttle commands

All of the control loops were implemented using traditional autopilot techniques, such as those described in (Stevens and Lewis, 1992). Of course, while the design of the control loop is standard, appropriate values for the feedback gains had to be calculated (just as in a standard PID controller). The control loops will now be analyzed in detail.

3.5.1 Pitch-hold

The pitch-hold loop is shown in Figure 3.10. It uses pitch angle error and pitch rate signals to derive a variation in elevator deflection. Thus, it is functionally a PD controller. The gains are respectively $K_p = 0.25$ for the pitch angle signal (proportional)

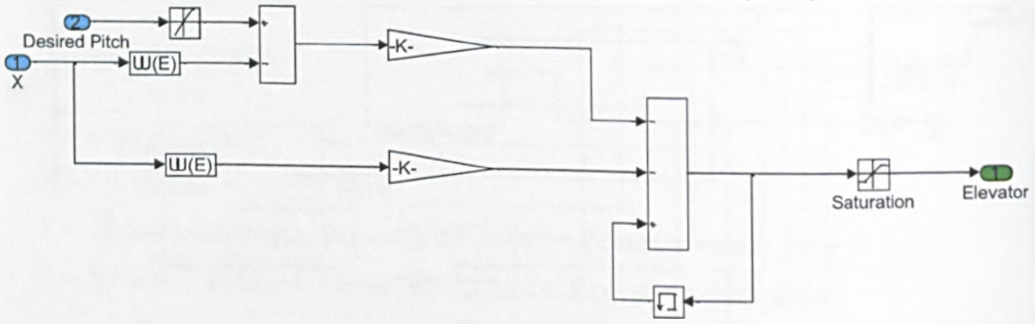


Figure 3.10. Pitch-hold loop.

and $K_d = 0.02$ for the pitch rate (derivative) signal. To improve performance, the rate of change for the desired pitch has been limited to 0.1 rad/sec. The elevator deflection output is limited at ± 0.7 rad ≈ ± 40 deg.

3.5.2 Altitude-hold

The altitude-hold loop is shown in Figure 3.11. It uses altitude error and altitude rate signals, but the altitude error signal is also integrated to obtain what is in fact a standard PID controller. This provides a pitch angle command as output, that can then be fed to the pitch-hold loop. The gains are respectively $K_p = 0.005$ for the altitude (proportional) signal, $K_d = 0.005$ for the altitude rate (derivative) signal and $K_i = 0.0005$ for the integrated altitude (integrative) signal. The integrated signal is reset to zero when its absolute value exceeds 500. The pitch angle output is limited between 0.5 rad (28.6 deg) and -0.2 rad (-11.4 deg). A speed limiter algorithm is also used to avoid excessive speed (especially during descent phases); when the true airspeed signals exceeds 60 m/s, the pitch angle demand is proportionally increased (with a gain of 0.016) with it.

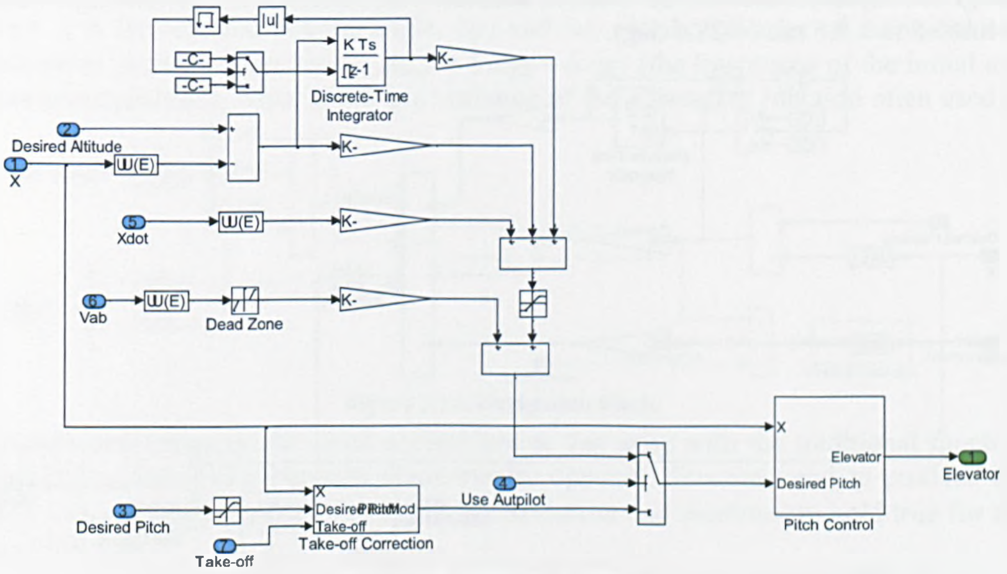


Figure 3.11. Altitude-hold loop.

In the figure, it is also possible to notice how the desired pitch value is switched between the Direct command and the altitude-hold command, depending on the current autopilot mode. The altitude-hold loop is used only when the autopilot is in Auto mode.

3.5.3 Roll-hold (wing leveller)

The roll-hold loop is shown in Figure 3.12. It uses roll angle error and roll rate signals to calculate the value of the aileron deflection; the roll angle error is also integrated (thus forming a standard PID controller). The gains are respectively K_p =

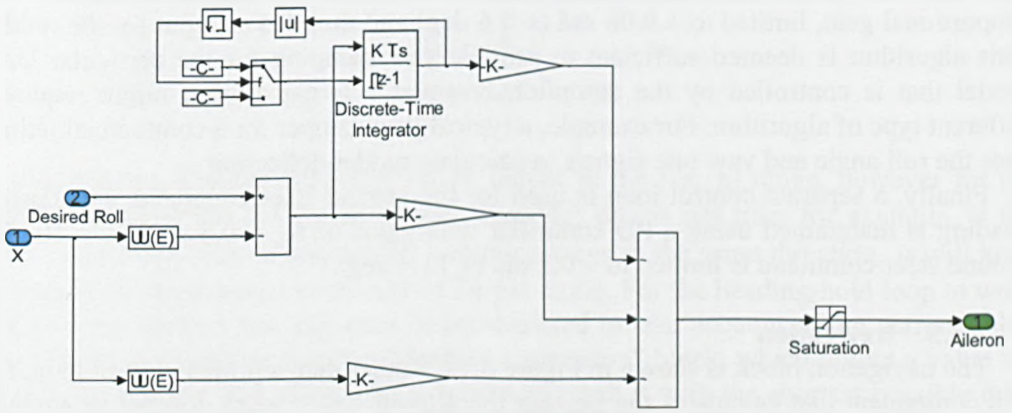


Figure 3.12. Roll-hold loop.

0.18 for the roll angle (proportional) signal, K_d = -0.03 for the roll rate (derivative) signal and K_i = 0.035 for the integrated roll angle (integrative) signal. The integrated signal is reset to zero when its absolute value exceeds 0.4. The aileron angle output is limited between ± 0.05 rad (± 2.9 deg).

3.5.4 Heading-hold

The heading-hold loop is shown in Figure 3.13. It uses the yaw angle error signal to calculate a value of desired roll angle; the yaw angle error is also integrated (thus forming a standard PI controller). The gains are respectively $K_p = 0.6$ for the roll angle (proportional) signal and $K_i = 0.01$ for the integrated roll angle (integrative) signal. The integrated signal is reset to zero when its absolute value exceeds 1. The roll angle output is limited to ± 0.5 rad (± 28.6 deg).

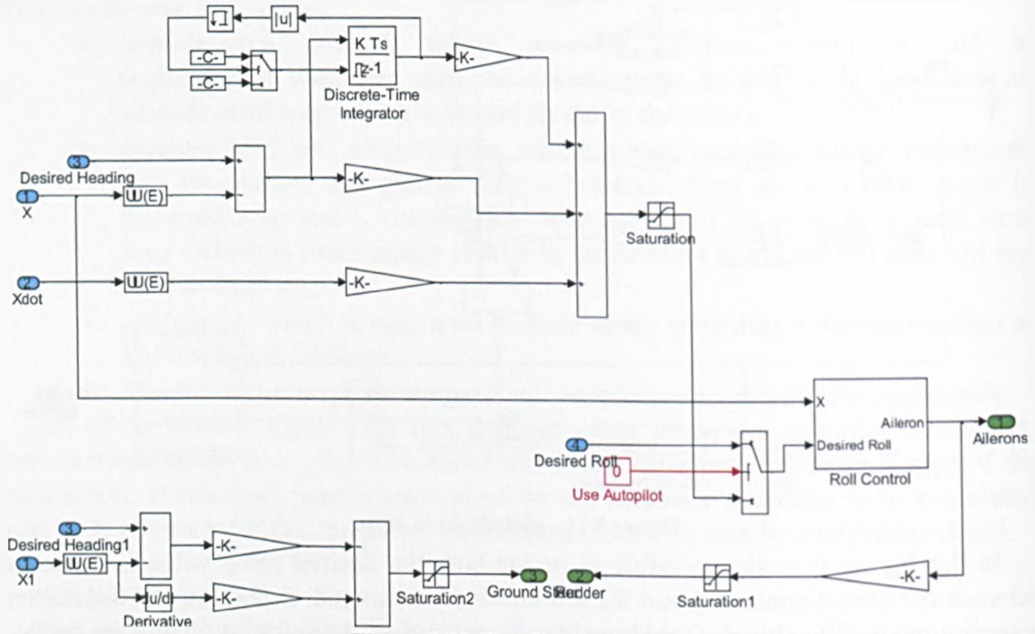


Figure 3.13. Heading-hold loop.

The possibility to use the roll-hold loop directly, receiving the target roll angle value from an external source rather than from the heading-hold loop has been implemented, however this is not used at the current development stage of SAMMS.

A simple yaw damper and turn coordinator algorithm can be also seen in Figure 3.11. In this case, the commanded aileron deflection is passed through a 0.3 proportional gain, limited to ± 0.08 rad (± 4.6 deg) and then fed as input for the rudder. This algorithm is deemed sufficient to provide yaw damping for the particular UAV model that is controlled by the autopilot, however a larger aircraft might require a different type of algorithm. For example, a typical yaw damper for a commercial jetliner uses the roll angle and yaw rate signals to calculate rudder deflection.

Finally, a separate control loop is used for the ground steer command. The desired heading is maintained using a PD controller with gains of $K_p = 0.5$ and $K_d = 10$. The ground steer command is limited to ± 0.2 rad (± 11.4 deg).

3.5.5 Navigation

The navigation block is shown in Figure 3.14. Rather than a proper control loop, this is a component that calculates the bearing the aircraft has to keep in order to navigate along a great circle route (as defined in 3.1.c, the intersection between a sphere and a plane which passes through the centre point of the sphere, and also the shortest distance between two points on a sphere). In the block, the aircraft's current position (in meters) and the coordinates of the starting position are used to calculate the current latitude and longitude, then the bearing needed to reach the current destination is calculated using the following formula:

$$\chi = \mathrm{atan2}\left(\sin\Delta_{long}\cdot\cos lat_2,\quad \cos lat_1\cdot\sin lat_2 - \sin lat_1\cdot\cos lat_2\cdot\cos\Delta_{long}\right)$$

where χ is the required bearing angle, lat_1 and lat_2 are the latitudes of the initial and destination position respectively, $\Delta_{long} = long_2 - long_1$ (the longitudes of the initial and destination position). Also, *atan2* is a variation of the arctangent function often used in

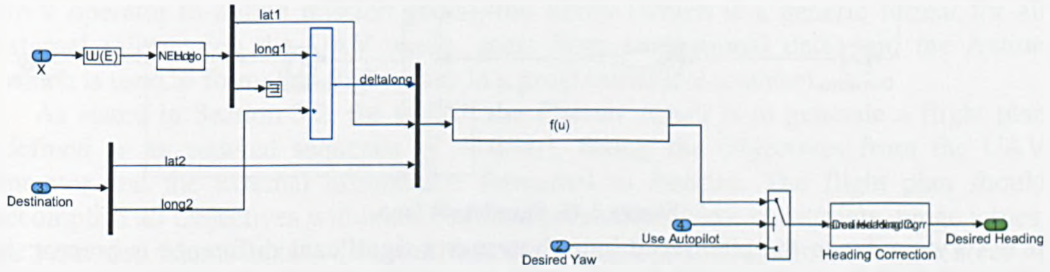


Figure 3.14. Navigation block.

programming languages to avoid several issues that arise with the traditional function (inability to distinguish between diametrically opposite directions and to produce the result of $\pm\pi/2$ when needed); the following definition and relationship hold true for the *atan2* function:

$$\mathrm{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \pi + \arctan\left(\frac{y}{x}\right) & y \geq 0, x < 0 \\ -\pi + \arctan\left(\frac{y}{x}\right) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ undefined & y = 0, x = 0 \end{cases} \qquad \mathrm{atan2}(y,x) = 2\cdot\arctan\left(\frac{y}{\sqrt{x^2+y^2}+x}\right)$$

The bearing formula always gives an output in the [-π, π] range, however for the heading angle of the UAV (the ψ state variable) this is not true; for example, if the UAV is loitering about a position by circling always in the same direction, ψ will keep increasing (or decreasing) at the rate of 2π per circle. For the heading-hold loop to work properly, the desired heading must be recalculated to take account of the current value of y. This is accomplished in the "Heading Correction" block, which output a value for the desired heading that allows the autopilot to reach it with the shortest possible turn. For example, if the UAV is on a *90 deg* heading and the bearing formula outputs a desired heading of *-135 deg*, the correction block changes this value to *225 deg*, which is in fact the same bearing (225 = -135 + *360*) but is reached by the heading-hold loop using the correct right turn (without the change, the loop would undergo a left turn, which is considerably longer).

3.5.6 Speed-hold loop

The speed-hold loop is shown in Figure 3.15. It uses true airspeed error and true airspeed rate (an acceleration, in fact) signals to calculate a variation to the throttle demand, meaning that this is effectively a PD controller. The gains are respectively $K_p = -0.05$ for the true airspeed error (proportional) signal and $K_d = 0.05$ for the true airspeed rate (derivative) signal. The throttle demand output is limited between 0 and 1.

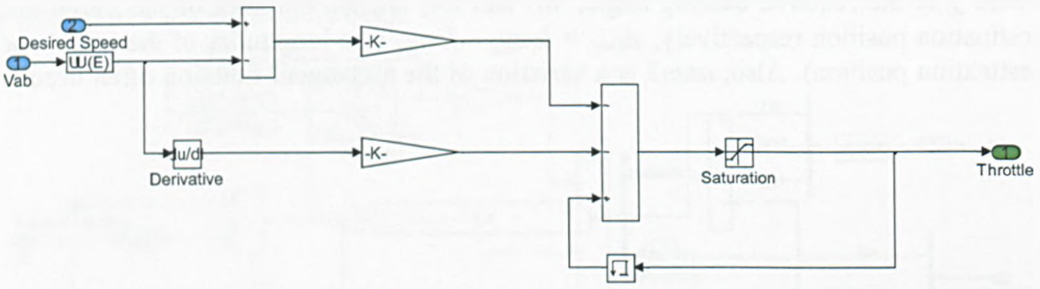


Figure 3.15. Speed-hold loop.

It is very similar to the pitch-hold loop, however a significant difference is present: in the pitch-hold loop, the derivative signal is sensor derived (pitch rate), while for the speed-hold loop the derivative signal is calculated on purpose, since there is no sensor for the true airspeed rate (an accelerometer would sense the accelerations along the body axes, but these would be in reference to the ground, not the air).

3.6 Concluding remarks

In this chapter, specific technical aspects of the project have been described. First, the theoretical concepts which are used throughout SAMMS are outlined, then an overview of the entire system architecture is given. Starting from this, minor (with respect to the Soar intelligent agents) SAMMS components are presented, and in particular the Simulink simulation environment in general, the UAV mathematical model and the autopilot model.

In short, this chapter laid the foundations for the next three chapters, in which the Soar intelligent agents will be described. The theoretical aspects are needed in order to understand how the agents work, the architecture description is useful in understanding how the Soar IAs interact between each other and with other components, and the description of non-Soar components is critical in explaining the simulations which were performed to test and validate the entire system.

The next chapter is the central chapter of the thesis, and is entirely dedicated to the Planner Agent, which will be described from theoretical and practical points of view and then tested, demonstrating the operation of all algorithms.

4. The Planner Agent

In Chapter 3, the general architecture of the Soar-based Autonomous Mission Management System (SAMMS) was described. The system is based on three Soar intelligent agents, interacting in a complex manner and complemented by several traditional control functions and algorithms. The chapter also introduced three abstractions that are used throughout the project: the Objective (which is used by the UAV operator to define mission goals), the Entity (which is a generic format for all external information the UAV needs, apart from navigational data) and the Action (which is used to formulate flight plans in a programmatical manner).

As stated in Section 3.2, the task of the Planner Agent is to generate a flight plan (defined as an ordered sequence of Actions), fusing the Objectives from the UAV operator and the external information formatted as Entities. The flight plan should accomplish all Objectives within the required parameters (time constraints, range issues, etc.) but also take account of the Entities, for example avoiding known danger areas or navigating through specific waypoints as requested by Air Traffic Control (ATC). The formulated plan needs not be optimal in a mathematical sense, but should not present contingencies and must be updated in real-time as the mission evolves.

The Planner Agent (or Planner in short) is arguably the most important component in the entire SAMMS architecture, and will be thoroughly described and tested in this central chapter of the thesis. Unlike the other components, it can be tested on its own, without the need for interaction with either the Execution Agent (Exag) or the Mission Manager Agent (MMA). This is because the input for the Planner is mainly constituted by the list of Objectives and the list of Entities: this information is by default provided as part of the test scenario, hence the Planner can be operated detached from the rest of the architecture. In fact, operating the Planner separated from the Exag and the MMA has two consequences: firstly, some feedback from the Execution Agent is needed, but this can be easily replicated; secondly, the Objective list to be fed as input to the Planner is not the one modified by the MMA but instead is the one provided by the UAV operator, however this does not interfere with the plan generation process.

The chapter is divided into four sections. In Section 4.1, the Input/Output (I/O) interface for the Planner agent is described; while this is based on the interface depicted in Section 2.8, details regarding the actual data being transferred are given here. Section 4.2 introduces the Planner from a discursive point of view; the agent is thoroughly described by looking at the actual Soar code and the way it is organized. The main principles of operation are elicited, the algorithms used to perform specific functions described, and critical code components detailed. Section 4.3 is dedicated to the presentation of the test scenarios that will be used to test the Planner; the scenarios are described in great detail, especially since the same scenarios will be used for testing the Exag and the MMA (and thus SAMMS as a whole system). Finally, Section 4.4 will present the results of the Planner testing campaign; generic results will be complemented by results obtained from the various scenarios to demonstrate specific capabilities and algorithms.

4.1 Planner I/O interface

In Section 2.8, the Soar/Simulink interface was described. In practice, a Soar agent is run within a Simulink simulation as an S-function: the S-function creates a Soar kernel and initializes the agent during the Simulink model initialization phase, then it commands execution of the agent and performs input/output operations while the simulation is running.

Soar agents manage input and output using two dedicated parts of working memory:

- the *input-link*, which is an area of working memory where only external applications (in our case, the Simulink S-function) can write new elements, while the agent has read-only access
- the *output-link*, which is an area of working memory where elements written by the agent are immediately passed to external applications

A normal simulation step will see the S-function update data in the *input-link*, retrieve data from the *output-link* and then call agent execution (using the *RunSelfTilOutput()* method, so that the agent will cycle until a new working memory element is added to the output-link).

Because of the large amount of data that needs to be exchanged between the Simulink framework and the Soar agent, the data is written in a structured manner, utilizing the characteristics of Soar. In Section 2.4, working memory elements (WMEs in short) were defined as identifier-attribute-value combinations; for example, the identifier "*cat*" might have the attribute "*colour*" with the value "*black*". The value of an attribute can be an identifier itself, thus making possible the organization of working memory into trees of WMEs. Figure 4.1 shows an example of how the working memory of an agent might be organized; in the figure, the identifier "*animals*" has three "*type*"

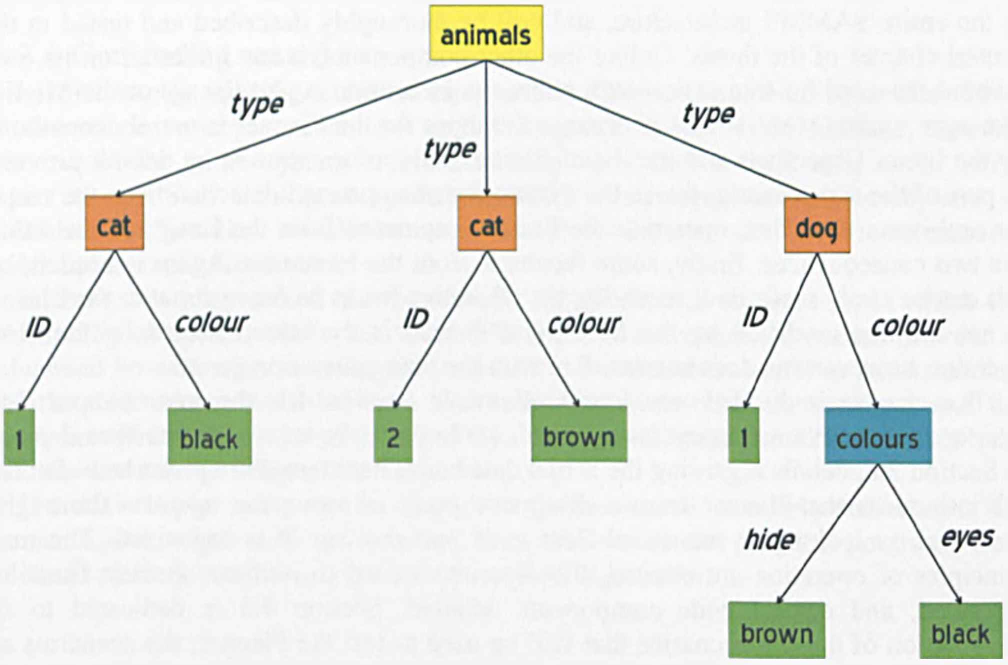


Figure 4.1. Example of working memory organization.

augmentations, two "*cat*" and one "*dog*", that are all identifiers themselves. Each of these has then its own set of augmentations: the "*cat*" identifiers have an "*ID*" attribute (with values of *1* and *2* respectively) and a "*colour*" attribute (with values of *black* and *brown* respectively), while the "*dog*" identifier has an "*ID*" attribute (value *1*) and a "*colour*" attribute that is further structured. In total, the working memory structure shown in the figure has 11 WMEs: animals-type-cat, cat-ID-1, cat-colour-black, animals-type-cat, cat-ID-2, cat-colour-brown, animals-type-dog, dog-ID-1, dog-colour-colours, colours-hide-brown, colours-eyes-black.

The I/O structures for the Planner Agent present a similar format, albeit a more complex one with hundreds of WMEs. The Input and Output structures will now be analyzed separately.

4.1.1 Input

Inputs for the Planner Agent are constituted by 6 different main categories, as in Table 4.1.

Table 4.1. Main level of input-link structure for the Planner Agent.

Code	Name	Values	Description
1	Airport Data	9	Data for the take-off and approach phases of the mission
2	Objectives	10	All Objectives and their properties
3	Entities	10	All Entities and their properties
4	Performed Actions	5	Feedback from Execution Agent; Total Distance and Time
5	MMA Feedback	1	Feedback from MMA
6	Flight Parameters	4	Airframe Data and Health

The Airport Data input structure contains data which is needed during the take-off and approach phases. The source of this data was discussed in Section 3.3, and its content detailed in Table 3.8. Table 4.2 shows information that was missing in Table 3.8, such as data types for the variables (float stands for a floating point variable, as opposed to integer or character variables). In total, this part of the input structure amounts to 13 separate values.

Table 4.2. Scenario airport information.

Code	Type	Definition	Type
1.1.1	Parking Position	Parking Position Latitude	float
1.1.2		Parking Position Longitude	float
1.2		Mission Start Time	float
1.3.1	Runway Position	Runway Position Latitude	float
1.3.2		Runway Position Longitude	float
1.4		Runway Heading	float
1.5		Mission Start Altitude	float
1.6.1	Landing Position	Landing Position Latitude	float
1.6.2		Landing Position Longitude	float
1.7		Landing Altitude	float
1.8		Landing Heading	float
1.9.1	End Position	End Position Latitude	float
1.9.2		End Position Longitude	float

The Objectives input structure contains the list of Objectives, as defined in Section 3.1, together with all of their properties. While in theory there is no limitation to the number of Objectives that can be handled by SAMMS, a limit of 10 Objectives is currently placed. In fact, this is due to the difficulty of handling all of this information within Simulink (and in particular within the interface S-functions) rather than to issues deriving from the Soar agents. The meaning of each property was discussed in the aforementioned section; in Table 4.3, the properties are summarized, together with the relevant data types. It is important to note that in order to optimize the exchange of data, a "compression" of the structure has been implemented. In fact, the number and type of Objectives properties varies depending on the Objective type (as detailed in Table 3.2). Since the amount of data exchanged between the Planner and its Simulink environment is quite large, the re-use of some variables was deemed necessary; thus, the "Variable 1" and "Variable 2" properties in Table 4.3 have different meanings depending on the Objective type. For *search area* Objectives, these will represent "Search Accuracy" and

"Search Radius" respectively; for *orbit position* Objectives, they will represent "Orbit Altitude" and "Orbit Time" respectively. Note that the data types for these properties are the same. With 10 Objectives with 13 properties each, this part of the input structure amounts for a total of 130 separate values.

Table 4.3. Objectives input structure.

Code	Type	Definition	Type
2.x.1	Objective Type	As in Table 3.1	integer
2.x.2	Objective Tag	Unique Objective numeric ID tag	integer
2.x.3.1	Objective Position	Latitude of Objective Position	float
2.x.3.2		Longitude of Objective Position	float
2.x.4.1	Priority	Time Priority for Objective	float
2.x.4.2		Execution Priority for Objective	integer
2.x.5	Duty	Type of Duty for Orbit Objectives	integer
2.x.6	Area Type	Type of Area for Search Objectives (box or circle)	integer
2.x.7	Variable 1	Search Accuracy or Orbit Altitude (see text)	float
2.x.8.1	Box-Corner	Latitude of position to define Area of Box Search	float
2.x.8.2		Longitude of position to define Area of Box Search	float
2.x.9	Variable 2	Search Radius or Orbit Time (see text)	float
2.x.10	Target Tag	Unique ID tag for the Entity that is a mission target	integer

The Entities input structure contains the list Entities, as defined in Section 3.1, together with all their properties. Just as with the Objectives, the number of Entities was limited to 10, even though in theory SAMMS should be able to handle any number of Entities. Table 4.4 shows the Entity properties and the relative data types. With 10

Table 4.4. Entities input structure.

Code	Type	Definition	Type
3.x.1	Entity Type	Type of entity (building, vehicle, weather zone, etc.)	integer
3.x.2	Entity Tag	ID tag for entity	integer
3.x.3.1	Entity Position	Latitude of most current position info for entity	float
3.x.3.2		Longitude of most current position info for entity	float
3.x.4.1	Movement Info	Entity speed of movement	float
3.x.4.2		Entity direction of movement	float
3.x.5	Entity Behaviour	Friendly, Neutral or Hostile	integer
3.x.6	Threat Level	Threat to the UAV, from negligible to catastrophic	integer
3.x.7.1	Area of Effect	Latitude for Box area, radius for Circle area	float
3.x.7.2		Longitude for Box area, zero for Circle area	float
3.x.8	Stance	Behaviour pattern for the UAV towards entity	integer

Entities with 11 properties each, this part of the input structure amounts for a total of 110 separate values.

The Performed Actions input structure (detailed in Table 4.5) contains information regarding what has already been accomplished by the UAV during a mission. A total of five variables are present. The first two variables, *Current Action* and *Comm-Obj*, are feedback from the Execution Agent, telling the Planner what Action of the flight plan is

Table 4.5. Performed Actions input structure.

Code	Variable	Description	Type
4.1	Current Action	Sequence number of the action being performed	integer
4.2	Comm-Obj	Indicates whether the current objective should be completed anyway or not	integer
4.3	New Plan Trigger	A computed index that determines how much the input situation has changed since last plan generation	float
4.4	Total Distance	Total distance travelled during current mission	float
4.5	Total Time	Total time since the mission has started	float

currently being performed and whether the Exag is committed to finish the current Objective before moving to another one. This information is used by the Planner during re-planning events: the new plan will take into account what Objectives have already been accomplished and whether the Objective currently being performed should be completed (more on this is Section 4.2.1 and Chapter 5). The third variable is the *New Plan Trigger* (provided by the function described in Section 3.3), which prompts the generation of a new flight plan. The fourth and fifth variables are the *total distance* covered and the *total flight time* for the current mission, as would be measured by an Inertial Measurement Unit and a Chronometer in a real UAV.

The MMA Feedback input structure contains direct feedback from the MMA. Normal feedback from the MMA is in the form of a change in the Objective list (see Section 3.2 for the details), however further information is contained here. In particular, this information is used by a particular Planner algorithm: the *mission-path-adjust* algorithm (described in Section 4.2.8). In short, the algorithm would plan detours.

Table 4.6. Flight Parameters input structure.

Code	Type	Definition	Type
6.1.1	Speeds	Maximum speed	float
6.1.2		Cruise speed	float
6.1.3		Minimum (stall) speed	float
6.1.4		Optimal speed	float
6.1.5		Take-off rotation speed	float
6.1.6		Landing speed	float
6.1.7		Ground Manoeuvre speed	float
6.1.8		Ground Movement speed	float
6.1.9		Current status	float
6.1.10		Current speed limitation	float
6.1.11		Search speed	float
6.1.12		Analyze speed	float
6.1.13		Attack speed	float
6.2.1	Altitudes	Maximum altitude	float
6.2.2		Cruise altitude	float
6.2.3		Minimum ground altitude	float
6.2.4		Climb turn altitude	float
6.2.5		Descent altitude	float
6.2.6		Pre-land altitude	float
6.2.7		Flare altitude	float
6.2.8		Current altitude limitation	float
6.2.9		Search altitude	float
6.2.10		Analyze altitude	float
6.2.11		Attack altitude	float
6.3.1	Distances and angles	Recon distance	float
6.3.2		Attack distance	float
6.3.3		Circle distance	float
6.3.4		Descent distance	float
6.3.5		Pre-land distance	float
6.3.6		Take-off angle	float
6.3.7		Climb angle	float
6.3.8		Flare angle	float
6.4.1	Fuel and range	Maximum Fuel	float
6.4.2		Remaining Fuel	float
6.4.3		Consumption constant	float
6.4.4		Minimum speed adjust	float
6.4.5		Maximum speed adjust	float
6.4.6		Fuel system status	float

around known danger areas (Entities with a Threat Level higher than 1); however, if an Objective lies within a danger area, the MMA instructs the Planner to bypass the mission-path-adjust algorithm for said Objective. The fact that the MMA might decide to cancel the Objective altogether is separate from this: in fact, when an Objective is in a threat area, the MMA will either cancel it (if the threat level is higher than the Objective execution priority) or instruct the Planner to bypass the mission-path-adjust algorithm. A single integer value is sent for each Objective, thus at present this part of the input structure is formed by 10 separate values.

Finally, the Flight Parameters input structure contains information regarding the aircraft capabilities and status. In fact, this is a part of the data that is provided by the Airframe Data and Health block in the Simulink simulation environment (see Section 3.3 and Table 3.9). Table 4.6 displays all variables that are part of this input structure, together with their relative data types. The variables amount for a total of 38 separate values.

Summarising, the input structure for the Planner Agent is constituted by a total of 306 variables, organized in the manner that was described in this subsection.

4.1.2 Output

The output of the Planner Agent is structured into three main categories: the Planner Metadata, the Flight Plan and the Plan Estimations.

The Planner Metadata part of the output structure contains general information regarding the current flight plan. There are two separate variables: *Number of Actions* is a count of the total number of Actions that form the current flight plan (to be used in various checks), while *Cycles Valid* indicates the number of Soar cycles over which the current flight plan has been valid (as calculated in the *modify-plan* state, which will be described in Section 4.2).

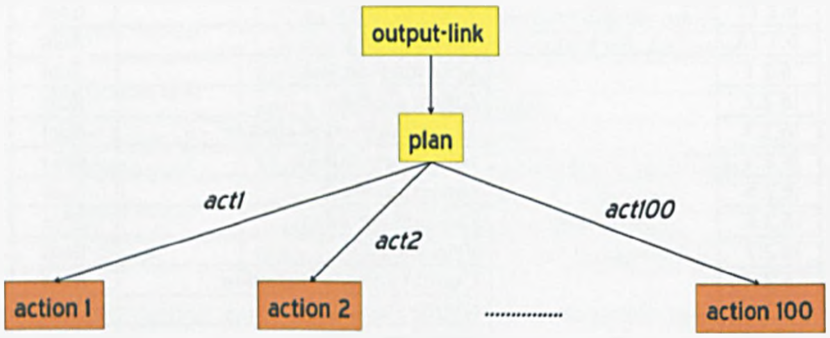


Figure 4.2. Organization of flight plan output of the Planner Agent.

The Flight Plan part of the output structure contains the entire current flight plan, that is, it contains all of the Actions that form the flight plan, together with the relative properties. In practice, this part of the output-link is structured as in Figure 4.2. While in theory SAMMS should have no limitation in the number of Actions it can handle, a limit of 100 Actions was effected (also as a consequence of the limitation to 10 Objectives). Each Action has its own data structure, just as defined in Section 3.1. Table 4.7 summarizes this, also adding details about data types. With 100 Actions with 13 properties each, this part of the output structure amounts for a maximum of 1300 separate values.

The Plan Estimations part of the output structure contains additional information regarding the flight plan; for each action, estimates of the distance covered, time needed and fuel spent are provided. It is organized similarly to the flight plan (see Figure 4.2), but with a reduced number of Actions since the take-off and approach phases are

Table 4.7. Action output structure.

Code	Type	Definition	Type
2.x.1	Action Type	As defined in Table 3.6	integer
2.x.2	Sequence	Sequence number for the Action	integer
2.x.3.1	Start Position	Latitude of initial position (used for some Action types)	float
2.x.3.2		Longitude of initial position (used for some Action types)	float
2.x.4.1	Position	Latitude of Action reference position	float
2.x.4.2		Longitude of Action reference position	float
2.x.5	Time	Time property of Action (usually time limit)	float
2.x.6	Heading	Bearing to be kept for certain Action Types	float
2.x.7	Altitude	UAV Altitude specified for Action	float
2.x.8	Duty	Duty type for Circle Actions, otherwise parent Objective type	integer
2.x.9	Speed	UAV Speed for Action	float
2.x.10	Target	Defines a specific target for Recon and Attack	integer
2.x.11	Objective	Parent Objective ID tag	integer

grouped into a single estimate (a single estimate for the take-off phase and a single estimate for the approach phase). For each Action, the six variables in Table 4.8 are provided. In short, distance, time and fuel estimates are derived for the Action, then these are added to the estimates of the Actions that precede it in the flight plan, thus providing the "total" values. In addition to this, the output structure provides four variables: the *Total Distance Estimate*, *Total Time Estimate* and *Total Fuel Estimate* simply reproduce the *Total Distance*, *Total Time* and *Total Fuel* values for the last Action in the Flight Plan (which will obviously be the approach phase), while the *Estimated Actions Number* is a count of the actual number of augmentations in the *Plan Estimations* part of the output structure. With 93 Estimation Actions with 6 properties each, plus the 4 generic variables, this part of the output structure amounts for a total of 562 separate values.

Table 4.8. Estimations output structure.

Code	Type	Definition	Type
3.x.1	Action Distance	Distance that will be covered for this Action	float
3.x.2	Action Time	Time to complete this Action	float
3.x.3	Action Fuel	Fuel to complete this Action	float
3.x.4	Total Distance	Total distance covered after this Action	float
3.x.5	Total Time	Total mission time after the Action is completed	float
3.x.6	Total Fuel	Total fuel used after Action is completed	float

Summarising, the output structure for the Planner Agent is constituted by a total of 1864 variables, organized in the manner that was described in this subsection.

4.2 Planner Algorithms and Soar code

In this section, the Planner Agent will be decomposed into its various parts; these will then be thoroughly described, so as to provide an accurate description of the agent's structure and of how it performs its functions.

Before detailing the various parts, a general overview of the agent structure is in order. Figure 4.3 shows the hierarchical organization of the Planner. Referring to the Soar terminology defined in Section 2.4, each block in the figure represents an agent state. It can be noted that these blocks are organized into different levels. From left to right:

- the top-level state (the Planner agent itself) has two mutually exclusive substates, *modify-plan* and *generate-plan*
- the generate-plan state has five possible substates (*old-plan*, *take-off*, *main-mission*, *approach*, *plan-optimization*) that are normally entered in this sequence
- the main-mission state has two possible substates, *plan-sequencing* and *actions-definition*
- the plan-optimization state has three possible substates, *mission-path-adjust*, *estimations* and *plan-finalization*

Before going into the details of lower-level substates, it is important to understand the way higher-level states operate. When first initialized, the Planner agent will obviously not have a currently selected flight plan; thus, the *generate-plan* state is entered. From a Soar point of view, this means that after agent initialization, a Soar production will search the working memory for a currently selected plan; not finding it, it will create an impasse, and the *generate-plan* substate will be created to solve it. Once the *generate-plan* state completes its plan generation process, a flight plan will be present in working memory, thus the impasse will be solved and the substate deleted. Another production will then fire, creating the *modify-plan* substate. This is basically the normal operating mode for the Planner agent; when in this state, the Planner keeps the current valid plan and is waiting for an external signal that will trigger the generation of a new plan.

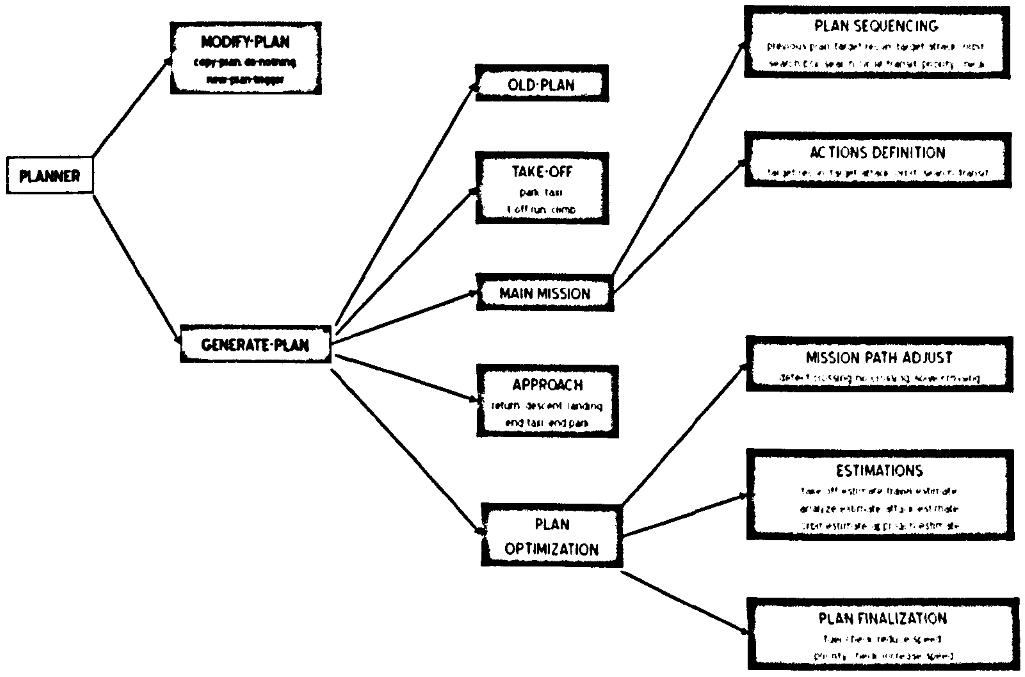


Figure 4.3. Planner Agent structural overview.

The *modify-plan* state is implemented by three operators. At the first cycle when the state is entered, the *copy-plan* operator fires; this copies the current flight plan into a specific working memory position, where a record of all the generated flight plans is kept. Then, the *do-nothing* operator is executed at each perceive-decide-act cycle (as described in Section 2.4); this just indicates the fact that no new plan is needed, but additionally the operator keeps a count of the number of cycles during which the plan has been valid. When the New-Plan Trigger function (see Section 3.3) sends the appropriate signal, the *new-plan-trigger* operator fires; this causes the cancellation of the current valid plan from working memory, hence an impasse arises again, prompting the creation of a new *generate-plan* substate. A new (updated) flight plan will be generated, closing the cycle since at this point the *modify-plan* substate can be entered again waiting for a new signal from the New-Plan Trigger function.

An important technical detail is that the agent is executed using the "*run-til-output*" option, which means that the agent will continuously cycle until at least a new working memory element is added to the *output-link*. Before the *generate-plan* state is closed, the entire flight plan is copied into the *output-link*; when in the *modify-plan* state, the *do-nothing* operator is constantly updating the number of valid cycles (which is an output), thus also regulating the agent execution. When the *new-plan-trigger* operator fires, it cancels the current flight plan not only from the top state working memory, but also from the *output-link*, hence prompting the agent to continue executing cycles until a new plan is present.

In the text, the Planner Agent is described but the actual code is not shown because of its length (the Planner is constituted by 268 productions). It is however interesting at this point to show some actual code, so as to give an idea about how Soar code is written and provide the possibility to understand some of the concepts that will be used in the agent description. Figure 4.4 shows the code for the *park* operator that is part of the *take-off* substate and will be described in Section 4.2.2. As all Soar operators, it has a proposal production and an application production. The proposal production is called *take-off*propose*park*; it fires when the current state is *take-off* and in the upper state in the hierarchy (the *superstate*) the *counter* working memory element (WME) is present and no Action of type 1 (a Park Action) is present in the flight plan . When it fires, it proposes the *park* operator, with an associated *counter* variable; no particular preferences are specified regarding operator choice. The application production (*take-*

```
sp {take-off*propose*park
  (state <s> ^name take-off
             ^superstate <ss>)
  (<ss> -^flight-plan.plan.<act>.act-type 1
        ^counter <counter>)
-->
  (<s> ^operator <o> +)
  (<o> ^name park
       ^counter <counter>)
}

sp {take-off*apply*park
  (state <s> ^name take-off
        ^superstate <ss>
        ^operator <o>
        ^io.input-link.air-data <air-data>
        ^knowledge-base.numbers.<act-num>
<counter>)
  (<ss> ^flight-plan.plan <plan>)
  (<o> ^name park
       ^counter <counter>)
  (<air-data> ^park-pos <park-pos>
              ^ms-time <ms-time>)
  (<park-pos> ^lat <park-lat>
              ^long <park-long>)
-->
  (<ss> ^counter (+ <counter> 1)
        ^counter <counter> -)
  (<plan> ^<act-num> <act>)
  (<act> ^sequence <counter>
      ^act-type 1
      ^position <position>
      ^time <ms-time>
      ^alt 0
      ^duty 0
      ^heading 0
      ^obj -1
      ^speed 0
      ^st-pos <st-pos>
      ^target 0)
  (<st-pos> ^lat 0
            ^long 0)
  (<position> ^lat <park-lat>
              ^long <park-long>)
}
```

Figure 4.4. Park operator in the take-off state.

*off*apply*park*) is longer and fires if the *park* operator has been chosen. In that case, it retrieves useful data from working memory (particularly the *input-link*) then proceeds to perform two tasks: it writes a new Park Action into the flight plan stored in working memory and increases the *counter* variable by 1. Figure 4.5 shows instead an example of a preference production; in particular, the production is used in the *plan-sequencing* state to implement the Nearest-Neighbour algorithm (more on this in Section 4.2.4). The rule says that if two operators are proposed in the *plan-sequencing* state, the operator with the smaller corresponding *range* value (*range* indicates the distance from the current planned position) should be preferred over the other. Note that the rule will fire for any number of proposed operators, so for example with three proposed operator the rule would fire once for each operator couple (three times in total).

```
sp {plan-sequencing*prefer*short-range
  (state <s> ^name plan-sequencing
             ^operator <o1> +
             ^operator <o2> +)
  (<o1> ^range <range1>)
  (<o2> ^range <range2> > <range1>)
-->
  (<s> ^operator <o1> > <o2>)
}
```

Figure 4.5. A preference rule in the plan-sequencing state.

Having explained the general principles of operation of the Planner Agent, it is now possible to look at the plan generation process itself, by analyzing the Soar code and productions state by state.

4.2.1 Old-plan

The *old-plan* state is only entered during re-planning events, since it can only be triggered when the *current action* input from the Exag is higher than zero. When SAMMS is initialized, no plan will be available, thus the *current action* variable will be set at zero; hence, the Planner will (rightly) skip this phase and go on with the plan generation process.

When the state is entered, it enables the previous flight plans to be taken into account during the generation of a new one; in particular, it is the part of the flight plan that has already been executed that is preserved. All Actions of the previous flight plan that have already been executed are directly copied into the new flight plan, together with Actions which the Exag is committed to complete.

To make things clearer, let us consider this example: an original flight plan is composed of 30 Actions and Action number 19 is being executed when the re-planning event occurs. Figure 4.6 presents how such a flight plan might be originating from 4 Objectives, and the two possible courses that the old-plan state might take.

In the first case, the re-planning event might be caused by a change in the time priority of Objective 4, making it necessary to execute it before Objective 3. The Execution Agent is not committed to completing Objective 3, hence the state will directly copy Actions 1 to 19 into the new plan, however Actions 18 and 19 will be marked as having been unsuccessful (because the relative Objective has not been accomplished). Objectives 1 and 2 are marked as completed, hence the plan generation process will complete the flight plan by adding Actions for Objective 3 and 4 (which are now executed in the inverse order).

In the second case, re-planning might be caused by the detection of a new threat on the route towards Objective 4, prompting the addition of a new waypoint to detour around it. In this situation, the Exag is committed to complete Objective 3, hence the old-plan state copies into the new flight plan not only Actions 1 to 19, but also the

Actions needed to complete Objective 3. The new flight plan will be unchanged until Action 21 (after the re-planning event, the Exag will still be executing Action 19 and correctly keep performing it) and Objectives 1, 2 and 3 will be marked as completed. The plan generation process will then complete the plan by adding Actions for Objective 4 (one more Action compared to the original flight plan, since a new waypoint is added, e.g. a new Travel Action).

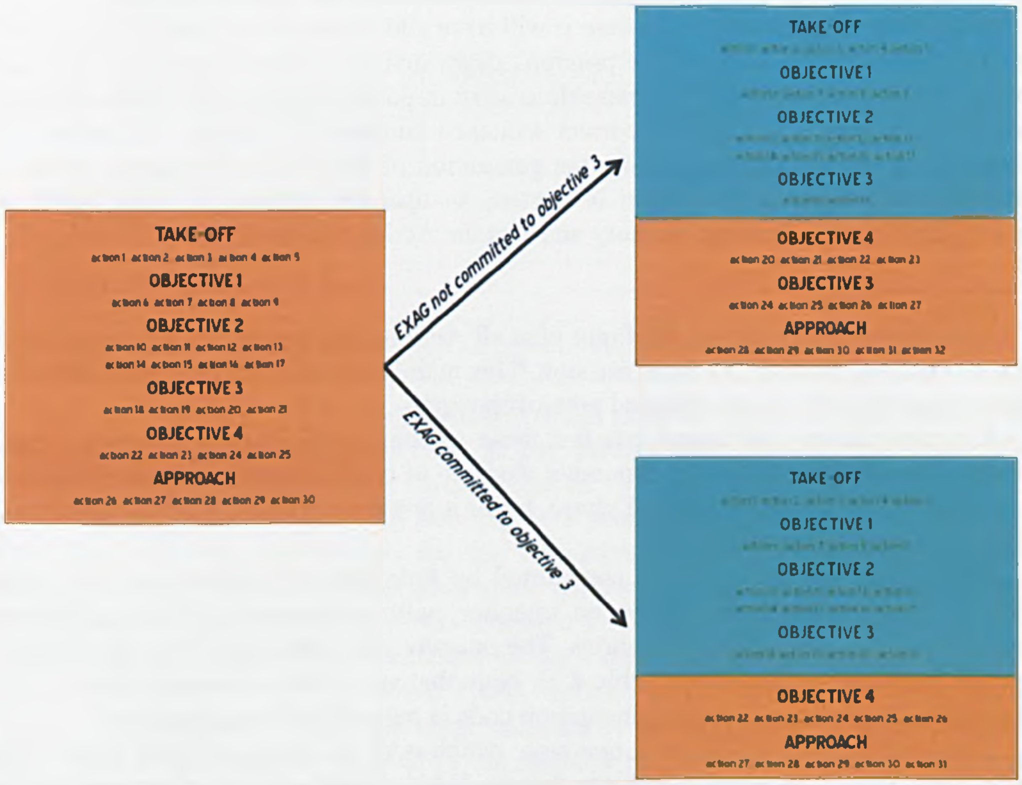

TAKE-OFF
action 1 action 2 action 3 action 4 action 5
OBJECTIVE 1
action 6 action 7 action 8 action 9
OBJECTIVE 2
action 10 action 11 action 12 action 13
action 14 action 15 action 16 action 17
OBJECTIVE 3
action 18 action 19 action 20 action 21
OBJECTIVE 4
action 22 action 23 action 24 action 25
APPROACH
action 26 action 27 action 28 action 29 action 30
EXAG not committed to objective 3
EXAG committed to objective 3
TAKE OFF
OBJECTIVE 1
OBJECTIVE 2
OBJECTIVE 3
OBJECTIVE 4
action 20 action 21 action 22 action 23
OBJECTIVE 3
action 24 action 25 action 26 action 27
APPROACH
action 28 action 29 action 30 action 31 action 32
TAKE OFF
OBJECTIVE 1
OBJECTIVE 2
OBJECTIVE 3
OBJECTIVE 4
action 22 action 23 action 24 action 25 action 26
APPROACH
action 27 action 28 action 29 action 30 action 31

Figure 4.6. Example of the effects of the old-plan state during a re-planning event.

The Soar implementation of this state is achieved by nine productions: two operators with two productions each (operators must have a proposal production and an application production), four elaborations and a preference production. The first operator is *executed-plan*, which basically prepares the stage for the elaborations. The four elaborations fire after executed-plan: the first copies all Actions until the current action into the new flight plan, the second fires when the Exag is committed to the current Objective and copies into the new plan the Actions needed to complete the Objective, the third and the fourth are used to mark unsuccessful Actions (Actions that have been at least partly executed but will have to be repeated since their parent Objective was not completed). The second operator is *executed-plan2*, which closes the state by setting variables that are required to continue with the plan generation process (most importantly, the internal Action counter). Note that elaborations in Soar typically have *instantiation-support* (or *i-support*, meaning that the changes they do to working memory are cancelled once the condition for the elaboration to fire is not any longer valid); the elaborations used here instead have *operator-support* (or *o-support*, meaning that changes they do to working memory are permanent).

Looking at the previous examples, it is possible to see the order in which productions fire: in the first case, after the *executed-plan* operator (the proposal production fires first, before the application production), the first elaboration fires 19 times concurrently (one for each Action to be copied), then the third and fourth elaborations fire one time each to mark Actions 18 and Action 19 as unsuccessful, and finally the *executed-plan2* operator fires (again with a proposal production and an application production). In the second case, the executed-plan operator fires, then it is followed by the first elaboration firing 19 times concurrently with the second elaboration firing two times (adding Actions 20 and 21 to the new flight plan, to complete the Objective); as usual, the *executed-plan2* operator closes the state.

When the old-plan state is closing, it will have added two main things to the agent's working memory: the parts of the previous flight that have already been executed and the internal Action counter (this variable is very important as it is used during the plan generation process to assign the correct sequence numbers to Actions). When the old-plan state is not needed, e.g. during the generation of the first flight plan or when re-planning occurs before a mission is started, an operator (called *plan-start*) adds an empty flight plan to working memory and sets the Action counter to 1.

4.2.2 Take-off

The take-off state adds to the flight plan all Actions that are needed to put the UAV in the condition to begin its main mission. This mainly involves take-off and climb, but also taxiing the UAV to its intended take-off position.

It is important to understand here that these Actions must be executed in block: once the mission starts (e.g. the action number variable of the Exag goes to 2), the Exag will be committed to the entire take-off phase, hence a re-planning event will not interrupt or effect an ongoing take-off phase.

The take-off phase is always constituted by four separate Actions; in Soar terms each of these is implemented by an operator, with a proposal and an application production, for a total of 8 productions. The operators use input data from the Scenario Airport Information block (see Table 4.2). Note that an Action contains all data needed to perform it, but the actual implementation code is part of the Execution Agent.

The first operator is the *park* operator, which adds an Action of type Park to the flight plan. This is just used to indicate that the UAV should wait at the starting location before actually beginning the mission at the time specified in the Mission Start Time input (code 1.2 in Table 4.2). The Action which is added to the flight plan has five relevant properties: *action type* (set to 1 to indicate a park Action), *sequence number* (which will always be 1), *position* (simply the initial position, as set in the Parking Position input), *objective* (set to -1 to indicate that the Action is part of the take-off phase) and *time* (which is set to the Mission Start Time value).

The second operator is the *taxi* operator, which adds an Action of type Taxi to the flight plan. This is used to move the UAV on the ground, navigating it from the initial position to the expected take-off position. There are five relevant properties: *action type* (set to 2 to indicate a taxi Action), *sequence number* (which will always be 2), *start position* (coordinates of the initial position, as set in the Parking Position input), *position* (coordinates of location where take-off can be initiated, e.g. the runway start, as defined in the Runway Position input) and *objective* (set to -1 to indicate that the Action is part of the take-off phase).

The third operator is the *t-off-run* operator, which adds an Action of type Take-off to the flight plan. This is the Action during which the UAV actually accelerates along the runway, rotates and gains a positive rate of climb, until 50 feet from the ground are cleared. There are five relevant properties: *action type* (set to 3 to indicate a take-off Action), *sequence number* (which will always be 3), *position* (coordinates of location where take-off can be initiated, e.g. the runway start as defined in the Runway Position input), *heading* (the orientation of the runway to be kept during take-off, as defined in the Runway Heading input) and *objective* (set to -1 to indicate that the Action is part of the take-off phase).

Table 4.9. Actions of the take-off phase.

ID	Definition	Park	Taxi	Take-off	Climb
1	Action Type	1	2	3	4
2	Sequence	1	2	3	4
3	Start Position	/	Parking Position	/	/
4	Position	Parking Position	Runway Position	Runway Position	Runway Position
5	Time	Mission Start Time	/	/	/
6	Heading	/	/	Runway Heading	Runway Heading
7	Altitude	/	/	/	Mission Start Altitude
8	Duty	/	/	/	/
9	Speed	/	/	/	/
10	Target	/	/	/	/
11	Objective	-1	-1	-1	-1

The fourth operator is the *climb* operator, which adds an Action of type Climb to the flight plan. This Action causes the UAV to gain altitude after take-off, until a specified altitude is reached and the main mission started. At least part of the climb will be effected keeping the initial runway heading; it is then possible to specify an altitude at which the UAV will turn towards the first Objective while still performing the climb manoeuvre. There are six relevant properties: *action type* (set to 4 to indicate a climb Action), *sequence number* (which will always be 4), *position* (as defined in the Runway Position input), *heading* (the orientation of the runway, to be kept during the initial phase of the climb or for the entire climb, defined in the Runway Heading input), *altitude* (the altitude at which the climb manoeuvre is considered completed and the main mission will begin, as defined in the Mission Start Altitude input) and *objective* (set to -1 to indicate that the Action is part of the take-off phase). The altitude at which the turn towards the first Objective is effected is not contained in the Action, but is rather defined in the Airframe Data and Health parameters (see item 6.2.4 in Table 4.6).

Table 4.9 summarizes the Actions added to the flight plan during the take-off and the values at which Action properties are set.

4.2.3 Main mission

The main mission state is where Objectives are ordered (so as to minimize the distance covered while still respecting the possible time constraints) and then converted into the basic Actions which will accomplish them (the flight plan might be changed by other algorithms, for example adding waypoints to avoid a danger area, but in the main mission state the Objective is converted into its corresponding basic series of Actions). These two functions are accomplished within the relative substates that can be seen in Figure 4.3: *plan-sequencing* and *actions-definition*. These states are described in Sections 4.2.4 and 4.2.5.

The main mission state is also responsible for adding the *main-mission-start* and *main-mission-end* virtual Actions to a flight plan. Unlike others, these Actions do not represent actual manoeuvres of the UAV, but are instead placed within a flight plan to clearly signal the passages from the take-off phase to the main mission phase and from the main mission phase to the approach phase.

A main-mission-start Action is always placed in a flight plan at sequence number 5, since the take-off phase always results in four Actions and is immediately followed by the main mission. However, further main-mission-start Actions might be present in a flight plan; this is due to the fact that each time a re-planning event occurs a new main-mission-start Action is placed into the new plan, signalling the separation between the completed parts of the old plan and the newly generated parts. Looking at the example in Section 4.2.1 (Figure 4.6):

- in the first case, a main-mission-start Action would be added after Action 19, hence becoming Action 20
- in the second case, a main-mission-start Action would be added after Action 21, hence becoming Action 22

In both cases, the following Actions would then obviously have to "slide" by one in the sequence.

Only one main-mission-end Action is ever present in a flight plan, and it is placed just before the approach phase Actions. The main-mission-end Action signals the conclusion of the "main mission", meaning that it indicates the fact that mission Objectives at this point have been accomplished and the UAV now only has to return to its base. The main-mission-end Action is followed by Actions from the approach phase, which will be seen in Section 4.2.6.

4.2.4 Plan-sequencing

The plan-sequencing state is where the order in which Objectives will be executed is decided. While the order can be controlled indirectly via the Time Priority properties of Objectives, the UAV operator does not decide it directly, so a dedicated algorithm has been implemented. To avoid confusion, a clarification is in order: the Objective ID Tag property does not have any consequence on the actual order in which Objectives will be executed.

The plan-sequencing state has two goals to accomplish while ordering the Objectives: to minimize distance covered to complete the mission, and to make sure that the Time Priorities of Objectives are respected. In theory, separate algorithms are implemented for each of these goals, however in practice the Soar code is designed to implement them in an integrated manner.

Minimizing the distance covered is essentially a classical Travelling Salesman Problem (TSP). This is a very well known problem in mathematics and computer science, which was first formulated in the 1930s and is one of the most intensively studied problems in optimization. The literature on the subject is very extensive; for example, see (Bellman, 1960) and (Applegate et al., 2007).

The problem is formulated as such: given a list of points on a plane and their pair-wise distances, find the shortest possible tour that visits each city exactly once. Exact solutions of the problem are demonstrated to take large amounts of computational time, since the problem complexity scales with (*n!*). For example, applied to SAMMS the problem would have a maximum of 11 points to visit (10 Objectives plus the take-off and landing position), although the starting and ending point is known, leading to 10! (3628800) possible combinations. In practice, a large amount of heuristic methods to solve TSPs have been devised.

The simplest heuristic method is the Nearest-Neighbour (NN) algorithm, belonging to the class of algorithms known as "greedy", which is formed by the following steps:

- choose an arbitrary starting point
- move to the point which is closest to the current one

- if all points have been visited, terminate
- go back to step 2

This algorithm is very fast in computing a short route, however this will usually not be the optimal one. In fact, it has been demonstrated that, under certain conditions, the algorithm will provide the worst possible answer.

Despite these drawbacks, the NN-algorithm was chosen to be used in SAMMS for several reasons:

- it is very fast in execution
- most other algorithms assume the knowledge of the entire set of pair-wise distances; these would have to be calculated in SAMMS, leading to increased computational time
- it always gives a solution, even if not the optimal one
- the solution might have to be changed anyway because of time constraints

The second goal is accomplished by estimating the time needed to accomplish Objectives and then eventually prioritizing an Objective whose Time Priority is not being respected. For each Objective type, estimates of the distance and time required to accomplish it are calculated. If the estimate is found to be exceeding the Time Priority (e.g. the Objective is being accomplished after the time limit that was set), then this Objective is moved up in the sequence, even though this will of course hamper the effectiveness of the NN-algorithm.

From a Soar code point of view, the two algorithms are intertwined. Within the plan-sequencing state, there are a total of eight possible operators, plus three elaboration productions and nine preference productions. The main output of this state is an Objective sequence.

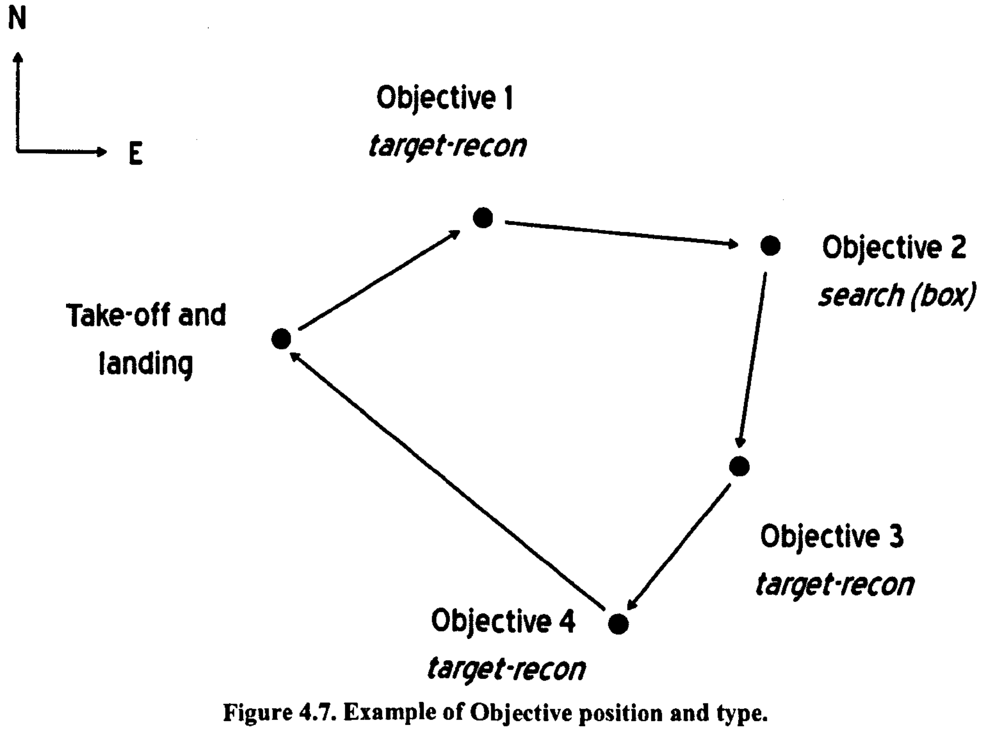


Figure 4.7. Example of Objective position and type.

The first operator is *previous-plan*, which deals with Objectives that have already been executed or have been cancelled (by either the UAV operator or the MMA). In short, it looks at the current Objective list and at the current flight plan (which at this stage will only include Actions created by the old-plan and take-off states) and takes account of Objectives that should not be considered by the plan-sequencing algorithms. The operator also needs a preference rule to ensure that it is executed before the other ones.

Next is a set of six operators, one per Objective type: *target-recon*, *target-attack*, *orbit*, *search-box*, *search-circle* and *transit*. These operators fire (or better, are proposed) when an Objective of the relative type is present in the Objective list and has not been already placed in the Objective sequence. The operator proposal productions include the calculation of distance and time estimates for the Objective; for this reason, the search Objective type has two different operators, since the estimates are calculated differently depending on the search pattern. Since accurate estimates for time and distance can only be calculated when the flight plan is available, the estimates which are used here are only approximations. Table 4.10 shows how estimates are calculated for each of the Objective types. For search Objectives, appropriate equations are used; lat_1, lat_2, lon_1 and lon_2 indicate the latitude and longitude of the first and second corners for search Objectives of type box (respectively, the position and box-corner properties of the Objective), R_E indicates Earth radius, *acc* indicates the search accuracy property (which is an Objective property), R indicates the search radius for search Objectives of type circle (which is also a property), the *div* function indicates the integer quotient and the *int* function indicates the integer part of a floating point number.

Table 4.10. Calculation of estimates based on Objective type.

Objective	Travel Time	Mission Time
target-recon	time = distance/cruise-speed	time = 2 * target-recon-distance / recon-speed
target-attack	time = distance/cruise-speed	time = 2 * target-attack- distance / attack-speed
orbit-position	time = distance/cruise-speed	time = orbit-time – previous-time
search-box	time = distance/cruise-speed	time = box-distance / search-speed
search-circle	time = distance/cruise-speed	time = circle-distance / search-speed
transit	time = distance/cruise-speed	/

Definitions	Description
distance	Distance calculated along great circle route
target-recon-distance	Distance for recon manoeuvre (item 6.3.1 in Table 4.6)
target-attack-distance	Distance for attack manoeuvre (item 6.3.2 in Table 4.6)
orbit-time	Time at which the orbit Objective is set to finish
previous-time	Mission time before reaching the orbit position
cruise-speed	Expected cruise speed for UAV (item 6.1.2 in Table 4.6)
recon-speed	Expected speed for recon manoeuvre (item 6.1.12 in Table 4.6)
attack-speed	Expected speed for attack manoeuvre (item 6.1.13 in Table 4.6)
search-speed	Expected speed during searches (item 6.1.11 in Table 4.6)

$$\text{box}-\text{distance} = (lon_2 - lon_1) \cdot R_E \cdot \cos\frac{lat_1 + lat_2}{2} + (lat_1 - lat_2) \cdot R_E \cdot div\left(\text{int}\left((lon_2 - lon_1) \cdot R_E \cdot \cos\frac{lat_1 + lat_2}{2}\right), \text{int}(acc)\right)$$

$$\text{circle}-\text{distance} = (1 + 3 \cdot div(\text{int } R, \text{int } acc) \cdot (div(\text{int } R, \text{int } acc) - 1)) \cdot \sqrt{2} \cdot acc$$

To better understand how these operators work, let us consider the example from Section 4.2.1 (Figure 4.6), starting from what would happen in the generation of the original flight plan. Figure 4.7 completes the example scenario by detailing how the four Objectives are positioned and the Objective types.

At the generation of the first plan, the previous-plan operator is not needed. The plan-sequencing state then sets the current position to the take-off position, and an elaboration calculates the distance to each Objective. Four operators then fire: three target-recon operators and a search-box operator. The choice between these is then governed by preference productions. The preferences rules are:

- immediate Time Priority forces the choice of the relative Objective
- Transit Objectives must be placed last in a plan
- if an Objective has a Time Priority and the time estimate indicates that it will not be respected, then it is chosen
- the closest Objective is preferred over the others

Let us assume that in our example Objective 1 has an immediate Time Priority, while the others have normal Time Priority. Objective 1 is chosen as first in the Objective sequence, and the current position is set to its position. The cycle is then repeated: the range elaboration calculates distances to each remaining Objective, three operators fire (two target-recon and a search-box) and Objective 2 is chosen for being the nearest. The algorithm continues until all Objectives have been placed into the sequence. When a re-planning event occurs, the previous-plan operator will exclude from this process Objectives that have already been completed (looking at Figure 4.6, Objectives 1 and 2 in the first case; Objectives 1, 2 and 3 in the second case), but the state will otherwise operate in the same manner.

The final operator of the plan-sequencing state is the *priority-check* operator. This fires when an Objective is added to the sequence, has a preference Time Priority (a time limit) and the time estimate for it shows that the priority will not be respected. When this happens, the Objective is scaled upwards in the sequence until the time estimation shows that the time priority is respected. To continue with the previous example, let us consider the re-planning event of the first case in Figure 4.6; Objectives 1 and 2 have been completed and a Time Priority is added for Objective 4. The *previous-plan* operator fires to exclude Objectives 1 and 2 from the current choice, then two target-recon operators fire for Objectives 3 and 4; the time estimate for Objective 4 would still be respected, so Objective 3 is chosen for being the nearest. When the sequence is completed by adding Objective 4, the time estimate for it will show that the time priority is not respected. The priority-check operator then fires, moving Objective 4 "up the ladder", in practice exchanging it with Objective 3. Hence, the resulting new plan will be the one shown in Figure 4.6.

When all non-accomplished Objectives have been placed into the Objective sequence, the plan-sequencing state closes.

4.2.5 Actions-definition

The actions-definition state is where Objectives are translated into the basic set of Actions that can accomplish them. Its operation is very straightforward: it goes through the Objective sequence provided by the plan-sequencing state, adding Actions to the flight plan for each Objective. There are operators for each Objective type; all Objective types apart from search are implemented by the relative proposal and application productions, while for search Objectives a further substate is entered. The Action counter from the old-plan state (see Section 4.2.1) is used to ensure that Actions are correctly ordered.

The *analyze* and *attack* operators are used for target-analyze and target-attack Objective types respectively. These two Objective types are very similar and so is their implementation within actions-definition. The operators add two Actions to the flight plan: a Travel Action and Recon/Attack Action (the latter obviously depending on the Objective type). Actions are detailed with all relative properties as in Table 4.11. Looking at the example scenario of Figures 4.6 and 4.7, the target-recon Objectives are translated within this state to two separate Actions each. So, for example, in the original plan Objective 1 would result in Actions 6 (Travel) and 7 (Target-recon). However, the Travel Action might be split into separate Actions by the algorithms that will be described later in this section, thus resulting in three Travel Actions (sequence numbers 6, 7 and 8) plus a Target-recon Action (sequence number 9).

Table 4.11. Actions for target-analyze and target-attack Objectives.

ID	Definition	Travel	Target-recon	Target-attack
1	Action Type	6	7	8
2	Sequence	as appropriate	as appropriate	as appropriate
3	Start Position	Position property of previous Action in the flight plan	/	/
4	Position	As defined in Entity Position for the target Entity	As defined in Entity Position for the target Entity	As defined in Entity Position for the target Entity
5	Time	/	/	/
6	Heading	/	/	/
7	Altitude	Cruise Altitude	Analyze Altitude	Attack Altitude
8	Duty	-1/-2	-1	-2
9	Speed	Cruise Speed	Analyze Speed	Attack Speed
10	Target	/	Entity Tag from Objective	Entity Tag from Objective
11	Objective	Parent Objective Tag	Parent Objective Tag	Parent Objective Tag

The *orbit* operator is used for orbit-position Objectives. Again, this is translated to two separate Actions: a Travel Action and a Circle Action. Things are even simpler for the *transit* operator, since this results in a single Travel Action added to the flight plan. Action properties are defined as in Table 4.12.

As previously stated, a new substate is entered for Objectives of the search type. This is necessary since the search Objectives are implemented by a series of Actions that is much more complex than for other Objective types. Objectives of the search type

Table 4.12. Actions for orbit-position and transit Objectives.

ID	Definition	Travel	Circle	Travel (Transit)
1	Action Type	6	9	6
2	Sequence	as appropriate	as appropriate	as appropriate
3	Start Position	Position property of previous Action in the flight plan	/	Position property of previous Action in the flight plan
4	Position	Position from Objective	Position from Objective	As defined in Entity Position for the target Entity
5	Time	/	Time from Objective	/
6	Heading	/	/	/
7	Altitude	Cruise Altitude	Orbit Altitude (from Objective)	Cruise Altitude
8	Duty	Duty from Objective	Duty from Objective	-4
9	Speed	Cruise Speed	Orbit Speed	Cruise Speed
10	Target	/	/	/
11	Objective	Parent Objective Tag	Parent Objective Tag	Parent Objective Tag

involve flying over an area along pre-determined search patterns (see Section 3.1.1). In actual search missions, several standardized patterns are used. Within SAMMS, two of these patterns are currently implemented, each related to a basic area shape: a rectangular area, covered using a parallel track search pattern, and a circular area, covered by an expanding diamond spiral search pattern.

The parallel track pattern is a widely used search pattern that is designed to cover a rectangular area uniformly; it is most effective for large search areas and when the object being searched has a probability of being anywhere in the search area. Figure 4.8 shows an example of the pattern; "search" legs are the vertical legs in the figure, horizontal legs are used to connect the vertical ones. Vertical legs are distanced by a fixed amount, which will usually depend on the type of sensors available and the conditions under which the search is being carried out. For example, on a manned search mission using simple visual recognition, the distance between the tracks might be of 2 km, decreasing as visibility decreases. The distance between tracks is defined *search accuracy* within SAMMS, and is one of the main operator inputs for a search Objective (note that search accuracy is also valid for the other search pattern, but with a different definition). A search Objective of this type is defined by four properties: the *position* property that defines the commence search point (point where the search is initiated), the *area type* property (set to 1 for this type of search), the *search accuracy* property and the *box-corner* property which defines the rectangular area by indicating

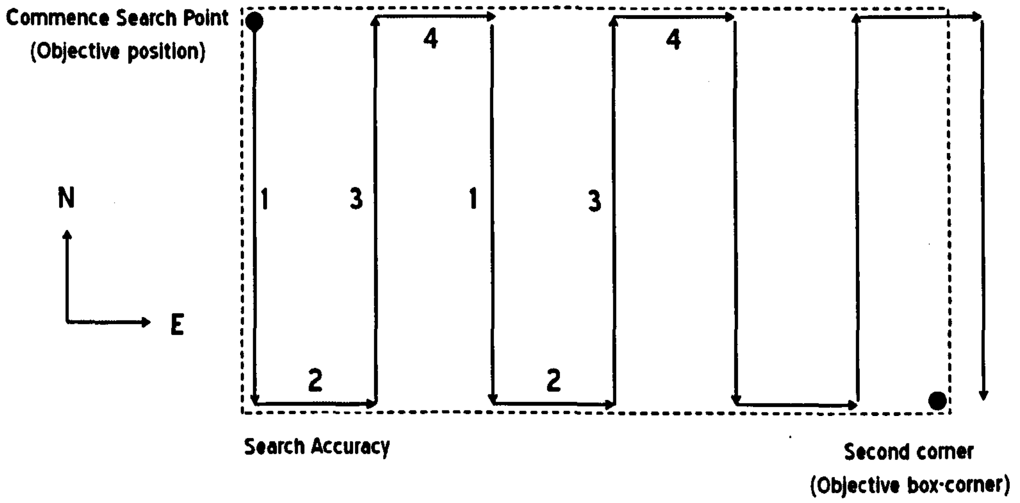


Figure 4.8. Parallel track search pattern for rectangular area.

the corner opposite from the commence start point. Each search leg (including "horizontal" ones) constitutes a Travel Action; a first operator (*box-begin*) adds to the flight plan a Travel Action to the commence search point, then four operators iteratively add legs, calculating the final position for each leg on the basis of the coordinates of the commence search point and the box corner, plus the search accuracy. The state closes when the longitude of the last vertical leg added is within the search accuracy distance from the box-corner longitude. Normally the pattern can be used with any orientation; however for simplicity the current SAMMS implementation has the vertical "search" legs being north-south legs, with the connecting legs being west-east legs. Adding the capability to orient the search area is certainly feasible (it just requires the use of spherical equations), but at present this is felt to be not relevant within the project aims.

The expanding diamond spiral pattern is not a standard search pattern, but is used within SAMMS to provide the capability of searching a circular area. In practice, it is very similar to the standard expanding square pattern, the main difference being in the orientation of certain legs (the expanding square pattern is designed to cover rectangular areas). Figure 4.9 shows the pattern: starting from the centre of the circular area, the first leg is north-ward for a distance equal to the *search accuracy*; from this point, the second leg goes to a position whose latitude is at search accuracy distance south and whose longitude is at search accuracy distance west. When the cycle is being closed (e.g. after leg 4), the north-ward distance is increased to twice the search accuracy distance, then the next legs will move on both axes for twice the distance. When the cycle is again closed, three times the distance is used, and eventually four, five, etc., times the distance will be used. The search is considered finished when the search accuracy distance multiplied by the number of "rounds" covered around the centre is greater than the *search radius*. A search Objective of this type has four relevant properties: the *position* property defines the centre of the circular area, the *area type* property (set to 2 for this type of search), the *search accuracy* property and the *search radius* property (which defines the radius of the circular area). As with the parallel track pattern, each leg of the search constitutes a Travel Action; a first operator (*circle-begin*) adds to the flight plan a Travel Action to the commence search point, then four operators iteratively add legs, calculating the final position for each leg on the basis of the coordinates of the commence search point and the search accuracy distance. Since the pattern is designed to cover a circular area (although in an approximate way, but in the best manner without using unpractical circular trajectories), it is not deemed necessary to implement the possibility to orient the pattern on a pre-defined heading.

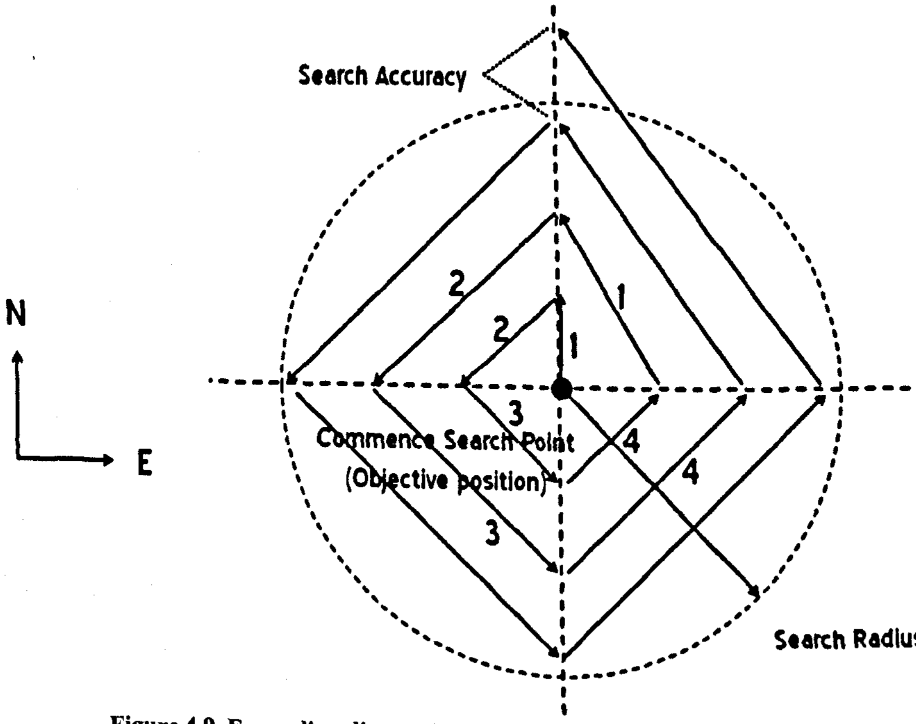


Figure 4.9. Expanding diamond spiral search pattern for circular area.

Finally, the actions-definition state is closed when all Objectives have been translated into a set of Actions.

An observation to be made at this point is that search Objectives are very different from other Objective types from a Soar point of view; they were incorporated into SAMMS with the purpose of showing how the modularity of Soar can be beneficial to a system with SAMMS' goals. As was shown in this subsection, most Objective types are translated into Actions by a single operator; however, for more complex Objective types (like the search Objective), a substate can be entered and generate a complex flight plan. In general, a future implementation of SAMMS might have other Objective types that require complex planning, and the presence of search Objectives demonstrates how this is entirely possible thanks to the innate modularity of the Soar architecture.

4.2.6 Approach

This state is entered after a main-mission-end Action is added to the flight plan and adds to it all Actions pertaining the final phases of flight. These include return to base, descent and landing, but also taxiing back to the final position and stopping operations. Just as for the take-off state, these Actions (apart from the eventual return to base) must be executed in block: once the Exag is committed to descent, the entire approach sequence will be executed as planned.

The approach phase is mainly constituted by four separate Actions, plus a return Action if needed. In Soar terms each of these is implemented by an operator, with a proposal and an application production, for a total of 10 productions. The operators use input data from the Scenario Airport Information block (see Table 4.2).

The *return* operator is needed to bring the UAV back to the base where it started its mission. It does so by adding to the flight plan a Travel action towards the base location. There are seven relevant properties: *action type* (set to 6 to indicate a Travel Action), *sequence number* (as needed), *start position* (coordinates of the final Objective accomplished during the main mission), *position* (coordinates of location where landing should happen, e.g. the ideal runway touch-down location, as defined in the Landing Position input), *altitude* (set to the Cruise Altitude input), *speed* (set to the Cruise Speed

Table 4.13. Actions of the approach phase.

ID	Definition	Return	Descent	Landing	End-Taxi	End-Park
1	Action Type	6	11	12	2	1
2	Sequence	as needed	as needed	as needed	as needed	as needed
3	Start Position	from last Objective	/	/	Landing Position	/
4	Position	Landing Position	Landing Position	Landing Position	End Position	End Position
5	Time	/	/	/	/	/
6	Heading	/	Landing Heading	Landing Heading	/	/
7	Altitude	Cruise Altitude	Landing Altitude	Landing Altitude	/	/
8	Duty	/	/	/	/	/
9	Speed	Cruise Speed	/	/	/	/
10	Target	/	/	/	/	/
11	Objective	-4	-4	-4	-4	-4

input) and *objective* (set to -4 to indicate that the Action is part of the approach phase). The return operator is not needed (and will not fire) when a Transit Objective is present: in this case, a Travel Action towards the expected landing location is added as part of main mission (the Travel Action added by *return* is not part of the main mission).

The *descent* operator adds a Descent Action to the flight plan, which will lead the UAV into a descent path that prepares it for landing. There are six relevant properties: *action type* (set to 11 to indicate a Descent Action), *sequence number* (as needed), *position* (coordinates of location where landing should happen, e.g. the ideal runway touch-down location, as defined in the Landing Position input), *altitude* (set to the

Landing Altitude input), *heading* (set to the Landing Heading input) and *objective* (set to -4 to indicate that the Action is part of the approach phase).

The *landing* operator adds a Landing Action to the flight plan; this will conduct the final phases of landing, aligning the UAV path with the runway direction while maintaining an appropriate negative rate of climb and then performing a flare prior to touchdown. There are six relevant properties: *action type* (set to 12 to indicate a Landing Action), *sequence number* (as needed), *position* (coordinates of location where landing should happen, e.g. the ideal runway touch-down location, as defined in the Landing Position input), *altitude* (set to the Landing Altitude input), *heading* (set to the Landing Heading input) and *objective* (set to -4 to indicate that the Action is part of the approach phase).

The *end-taxi* operator adds an Action of type Taxi to the flight plan. This will move the UAV on the ground, navigating it from the position where landing is concluded to the expected parking position. There are five relevant properties: *action type* (set to 2 to indicate a Taxi Action), *sequence number* (as needed), *start position* (coordinates of the initial position, can be set to Landing Position input but is ideally updated in real-time after landing is concluded), *position* (coordinates of location where the UAV is expected to park to conclude the mission, as defined in the End Position input) and *objective* (set to -4 to indicate that the Action is part of the approach phase).

The final operator is the *end-park* operator, which adds an Action of type Park to the flight plan. This is used to indicate that the UAV should wait at the final parking location, where it will likely receive further instructions (initiating a new mission) or be deactivated by ground crew. The Action which is added to the flight plan has five relevant properties: *action type* (set to 1 to indicate a park Action), *sequence number* (as needed), *position* (simply the final parking position, as set in the End Position input) and *objective* (set to -4 to indicate that the Action is part of the approach phase).

Table 4.13 summarizes the Actions added by the approach state. The state is closed when the final Park Action is added to the flight plan. At this stage, a full flight plan is already available, however this is a basic plan that needs to be improved by the algorithms of the plan-optimization state.

4.2.7 Plan-optimization

The *plan-optimization* state includes several algorithms that are designed to improve the flight plan that is available after the plan generation process has reached the approach phase. At this stage, a viable flight plan is already available, however this is not refined and in fact lacks all modifications that might be caused by the Entities being perceived. It can be said that the flight plan that is available before entering the plan-optimization state demonstrates autonomy, but not intelligence.

Depending on the actual UAV platform needs, the plan-optimization phase might be entirely skipped or instead be very complex. In the current SAMMS implementation, it is implemented with three capabilities: the ability to define waypoints in order to avoid navigation through known danger areas, the ability to estimate the time needed to perform a mission and the amount of fuel it will require, and the ability to use these estimates to detect contingencies in the flight plan and consequently improve it.

Each of these capabilities is implemented within a substate of the plan-optimization state: respectively, *mission-path-adjust*, *estimations* and *plan-finalization*; these will be described in the following subsections (4.2.8, 4.2.9 and 4.2.10). It is important here to understand how the substates interact; while *mission-path-adjust* operates before the other states, the operation of *estimations* and *plan-finalization* is intertwined. The algorithms in *plan-finalization* need the output from *estimations* in order to function; thus *estimations* is executed first. However, when *plan-finalization* algorithms operate a change to the flight plan, the time and fuel estimates will not be valid any longer; thus, the estimations state must be recalled. The cycle continues until the *plan-finalization* state approves the current flight plan without further modifications.

When all three substates are definitively closed, the plan-optimization state also closes, bringing the end of the entire plan generation process. The conclusion to the process sees the entire flight plan being copied to the appropriate top-state working memory location, as well as to the output-link. When generating a plan, the Planner Agent is continually cycling until this moment, when its operation is stopped since output has been written to the output-link. The Planner will then be ready to enter the modify-plan state, acting as described at the beginning of this Section.

The plan generation process is completed in roughly 0.5 sec during the SAMMS simulations that were run to validate the system. As will be seen In Section 4.4, these simulations are run in a Simulink environment and using normal computer hardware running the Windows XP operating system.

4.2.8 Mission-path-adjust

The *mission-path-adjust* state implements the algorithm that will modify the flight plan so that known danger areas are avoided. It does so by using three different operators: *detect-crossing*, *no-crossing* and *solve-crossing*. First, possible cases where a Travel Action is intersecting with a known danger area are detected by detect-crossing;

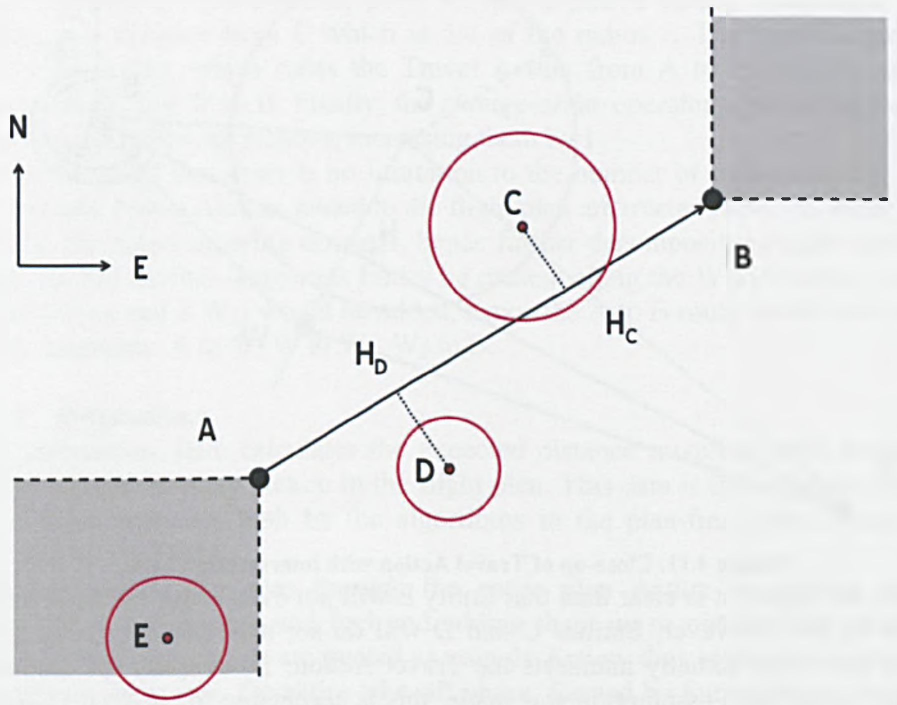


Figure 4.10. Example of Travel Action with Entities.

when a possible intersection has been detected, this will be checked; if it is not an actual intersection, the no-crossing operator will fire and rule it out. If it is an actual intersection, the solve-crossing operator will fire, opening a further substate which will result in the addition of a new waypoint to the flight plan.

The *detect-crossing* operator is one of the most complex operators within the entire SAMMS software. Its most basic form fires for any Travel Action – Entity combination (when the Entity has a threat level higher than 1 and an area of effect of radius higher than zero). To reduce the number of operator proposals during run-time, 12 proposal productions have been implemented. Consider Figure 4.10, which shows an example Travel Action from point A to point B (which are the positions of two Objectives), with three dangerous Entities represented by the three circles and respectively centred at positions C, D and E. It is immediately possible to see that any Entity whose centre lies within the greyed areas in the figure cannot possibly intersect the Travel route without also encompassing either point A or point B. Since when a dangerous Entity encompasses an Objective position the MMA will either instruct the Planner to ignore the mission-path-adjust algorithm for that Entity (using the MMA-Feedback input described in Section 4.1.1) or cancel the Objective (if the threat level is higher than the Objective execution priority), Entities that lie within the greyed areas can be immediately excluded from checking whether they intersect the Travel Action. In the figure, this means that an Entity with latitude and longitude both lower than those of A (positive latitude is North, positive longitude is East), or both higher than those of B will certainly not present a possibility of intersecting the Travel Action. The reasoning must be modified depending on the relative positions of A and B, hence the need for 12 different proposal productions to cover all possible situations.

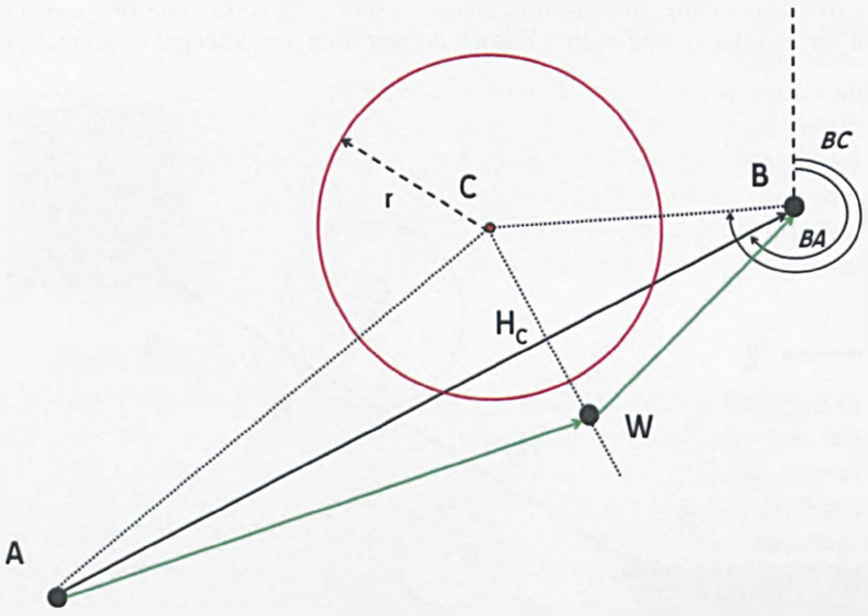


Figure 4.11. Close-up of Travel Action with intersecting Entity.

From the figure, it is clear then that Entity E will not even cause the detect-crossing operator to fire. However, Entities C and D will do so; it is then necessary to check whether the Entity actually intersects the Travel Action. To simplify the mathematics involved, a flat Earth assumption was made; this is acceptable for small distances (less than 100 km).

Figure 4.11 shows an improved diagram for Entity C. Point H_C is the base of the altitude of triangle ACB; *BC* and *BA* are the angles calculated as bearing from point B to points C and A respectively. Hence, angle *CBA* is in general equal to $|BC - BA|$. Knowing distance BC, the distance CH_C is then equal to:

$$\overline{CH_C} = \overline{BC} \cdot \sin\left(C\hat{B}A\right)$$

It is immediately clear that the Entity will intersect the Travel Action only if the distance CH_C is smaller than the Entity radius *r*.

For every Travel Action – Entity combination that is not evidently without an intersection (e.g. the Entity is not in the greyed areas), the detect-crossing operator will fire. In the example, this means that the operator would fire twice, for Entities C and D. The detect-crossing operator has indifferent preference, thus during the internal decision process of Soar the choice between the two instances of the operator is random. Assuming that the proposal for the D Entity is chosen first, the detect-crossing operator then calculates the angles *BA* and *BD* and the distance DH_D. At this stage, two operators can fire: *no-crossing* and *solve-crossing*. The former fires if the distance DH_D is lower than the Entity radius *r*, the latter if the distance is higher. For Entity D, the no-crossing operator will fire; this marks the Travel Action – Entity combination as not being an actual intersection, so that the proposal for detect-crossing is retracted and the state can continue to check other possible intersections.

The *solve-crossing* operator would instead fire for Entity C in the example. This results in the addition of a new waypoint that will ensure that the danger area represented by the Entity is not entered. Solve-crossing is actually a substate, with three operators that will modify the flight plan as needed. The first operator is *calculate-waypoint*, which decides the position of the waypoint to be added. Looking at Figure 4.9, the waypoint is represented by point *W*; this is placed on the continuation of the CH_C line, at a distance from C which is 5/4 of the radius *r*. The second operator is *generate-new-action*, which splits the Travel Action from A to B into two separate Actions, A to W and W to B. Finally, the *change-order* operator updates the sequence numbers for the following Actions, increasing them by 1.

It is to be noted that there is no limitation to the number of waypoints that can be added: the new Travel Actions added to the flight plan are treated just as all other Travel Actions by the detect-crossing operator, hence further decomposition might occur. For example, should another dangerous Entity be present along the W to B route, a second waypoint (let us call it W_2) would be added, hence the A to B route would now be split into three segments: A to W, W to W_2, W_2 to B.

4.2.9 Estimations

The *estimations* state calculates the expected distance travelled, time needed and fuel used to perform every Action in the flight plan. This data is then used to check the plan for inconsistencies, both by the algorithms in the plan-finalization state of the Planner and by the MMA.

The state essentially goes through the entire plan Action by Action, deriving estimates for distance, time and fuel and adding them up to obtain total values. The take-off and approach phases are treated as a single Action, thus estimates are calculated for the entire phases (e.g. the entire take-off phase, formed by four Actions, results in a single instance of the three estimates, distance, time and fuel; the same holds true for the approach phase). Also, during a re-planning event estimates are not calculated for the parts of the plan that have already been executed; the actual total distance and total time values (from the Performed Actions input structure, Table 4.5) are instead used for the calculation of estimates of the total distance and time.

A first set of estimates is calculated in the plan-sequencing state, where they are needed to choose the order in which Objectives are to be executed while respecting eventual time priorities. These estimates are approximate, since they are calculated without knowing the actual flight plan and only on the basis of the Objectives and their type and relative position. At this stage instead a full flight plan is available, so the estimates can be much more accurate (although they are still estimates and are therefore calculated assuming perfect flight conditions). The calculations are effected by a total of six operators: *take-off-estimate*, *travel-estimate*, *analyze-estimate*, *attack-estimate*, *orbit-estimate* and *approach-estimate*. Adding the elaborations that are needed to calculate the estimated fuel consumption, this accounts for a total of 15 productions for the state.

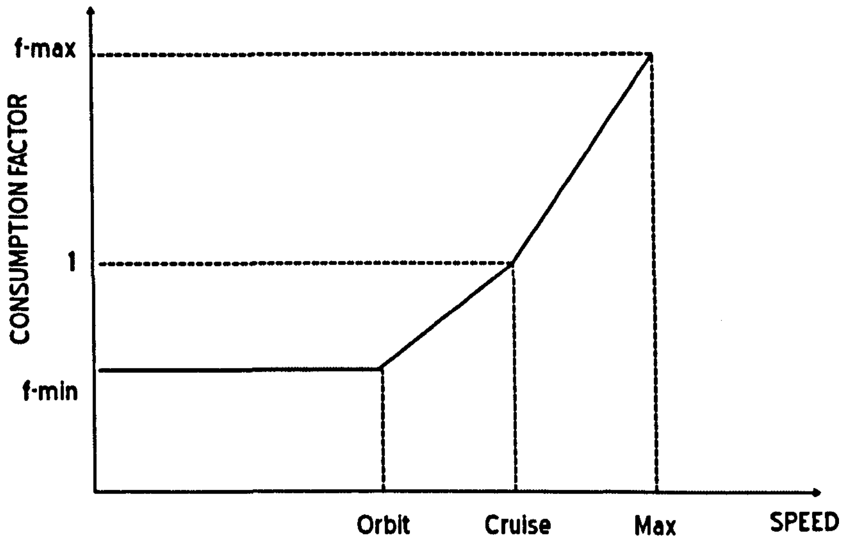


Figure 4.12. Consumption factor *f* versus speed.

Before describing the operators in detail, a digression regarding the fuel consumption model is in order. Within SAMMS, fuel consumption is calculated as a function of distance: a UAV flying for a certain distance will use an amount of fuel that is directly proportional to the distance. The consumption constant K is defined as the ratio between used fuel and covered distance while flying at cruise speed; K is measured in Kg/m and is constant for a specific UAV. It is one of the inputs given to the Planner Agent as part of the Airframe Data and Health input structure (see Table 4.6, code 6.4.3). However, any aircraft will use different amounts of fuel depending on the actual running conditions of the engine. This is taken into account within SAMMS by introducing a correction factor f, which is calculated depending on the airspeed at which the UAV is flying. Figure 4.12 shows how the factor is calculated; f is constant with a value of *f-min* for any speed below the orbit speed (in reality, the UAV will never reach very low speeds in flight, since it would stall; since orbit speed is used when loitering, this is assumed to be the speed for which minimum consumption is achieved), then it increases linearly between orbit speed and cruise speed. At cruise speed f has a value of 1, and then it again increases linearly between cruise speed and maximum speed (but with a different rate), until reaching the value *f-max* at maximum speed. The orbit, cruise and maximum speeds are defined for the UAV as part of the Airframe Data and Health input structure, thus to fully define the fuel consumption characteristics three more constants are needed: K, *f-min* and *f-max*; these are also passed to the Planner as inputs. Finally, some faults might cause an increase in fuel consumption; when this happens, it is assumed that a fault detection algorithm will detect the fault and estimate its effect on fuel consumption. The effect is taken into account by introducing a second correction factor *S*, which is representative of the airframe condition. The general expression of fuel consumption is then:

$$fuel = \text{distance} \cdot K \cdot f \cdot S$$

which is used throughout the estimations state to derive fuel estimates from distance estimates.

The *take-off-estimate* operator calculates the distance, time and fuel estimates for the entire take-off phase. In general, fixed values are used for the Park, Taxi and Take-off Actions; an exception is the time estimate for the Park Action, which is obviously set equal to the Mission Start Time (as defined in the Scenario information, Table 4.2). Distance covered during the Climb Action is calculated as:

$$\text{distance} = \frac{(climbaltitude - airportaltitude)}{\sin(angle)}$$

where *climbaltitude* is the target altitude where the Climb Action is concluded, *airportaltitude* is the altitude at the starting airport and *angle* is the pre-defined climb angle which is used to effect the climb. Time for the climb is calculated dividing the estimated distance by the cruise speed (while the thrust setting during climb will normally be maximum thrust, the achieved speed will not be maximum speed). Fuel consumption during climb is estimated using the normal formula, using the *f-max* value for the correction factor *f*. The combined distance, time and fuel estimates are written in working memory as estimates for Action 5 (which is always the main-mission-start); at this stage, the action estimate coincides with the total estimate (the sum of estimates until the Action, including the Action itself), since this will always be the first executed part of the flight plan.

In a flight plan, the main mission phase is a sequence of Actions of four types; each of these types has a related operator. The *travel-estimate* operator calculates distance, time and fuel estimates for Travel Actions. Distance is calculated as the distance along a great circle route between the locations defined by the *start-position* and *position* properties of the Action. The Haversine formula (Gellert et al., 1989) is used here (and throughout SAMMS when calculating great circle distance):

$$\text{distance} = 2 \cdot R_E \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$$
$$a = \sin^2\left(\frac{lat_2 - lat_1}{2}\right) + \cos lat_1 \cdot \cos lat_2 \cdot \sin^2\left(\frac{lon_2 - lon_1}{2}\right)$$

where R_E is the Earth radius, lat_2 and lat_1 the latitudes of position and start-position respectively, lon_2 and lon_1 the longitudes of position and start-position respectively, and *atan2* is the variation of the arctangent function (defined in Section 3.5). Time is calculated as distance divided by the *speed* property of the Action; this is normally the cruise speed, but could be different (for example if the algorithms that will be described in subsection 4.2.10 change it). Fuel consumption is calculated using the normal formula, where the factor *f* is calculated on the basis of the *speed* property of the Action (using linear interpolation between the known constant points).

The *analyze-estimate* operator calculates distance, time and fuel estimates for Actions of the Target-Recon (or Target-Analyze) type. Distance is calculated as twice the Recon Distance (as defined in Table 4.6, code 6.3.1); this is because of how Recon Actions are actually performed by the Execution Agent (as will be described in Chapter 5). Time is calculated as distance divided by the *speed* property of the Action (normally, the Analyze Speed, code 6.1.11 in Table 4.6). Fuel consumption is calculated using the normal formula, with the factor *f* calculated on the basis of the *speed* property of the Action.

The *attack-estimate* operator calculates distance, time and fuel estimates for Actions of the Target-Attack type. Distance is calculated as twice the Attack Distance (as defined in Table 4.6, code 6.3.2). Time is calculated as distance divided by the *speed* property of the Action (normally, the Attack Speed, code 6.1.12 in Table 4.6). Fuel consumption is calculated using the normal formula, with the factor *f* calculated on the basis of the *speed* property of the Action.

The *orbit-estimate* operator calculates distance, time and fuel estimates for Actions of the Circle type. In this case, the time estimate is the first to be calculated, since the Action continues until the time limit (set in the *time* property) is reached; the time estimate is calculated as the difference between the time property (the time limit) and the estimated total time before the Action is started. The distance estimate is obtained multiplying the time estimate by the *speed* property of the Action (which should be the orbit speed). The fuel consumption is computed using the normal formula, with the factor *f* calculated on the basis of the *speed* property of the Action (unless a speed higher than orbit speed is used, *f* will be equal to *f-min*).

As for the take-off phase, the distance, time and fuel estimates for the entire approach phase are calculated in block (notice however that this does not include the eventual *return-to-base* Action, as this is a Travel Action and will be accounted for by the travel-estimate operator). The *approach-estimate* operator performs all the relative calculations; the final Park Action is excluded, since distance and fuel are zero and the time is unknown (the UAV is waiting to be de-activated or ordered to perform a new mission). Fixed values are used for the Landing and Taxi Actions. For the Descent Action, distance is calculated as the sum of the Descent Distance and Pre-Land Distance values (from Table 4.6, codes 6.3.4 and 6.3.5); this will be explained in Chapter 5 when describing how the descent manoeuvre is accomplished by the Execution Agent. Time for the Descent action is obtained dividing distance by the orbit speed; fuel consumption is calculated using the normal formula with *f* = *f-min*. The distance, time and fuel estimates for the approach phase are obtained adding the Descent values with the fixed values for the Landing and Taxi Actions; this is stored in working memory as the estimated for the main-mission-end Action.

The total estimates obtained from the approach-estimate operator are also the final estimates for the entire flight plan. When these are available, the estimations state is closed; however, if one of the algorithms in the plan-finalization state changes the flight plan, the estimates will not be valid any longer; the plan-finalization state will cancel the estimates and the estimations state will recalculate them (using values from the updated flight plan).

4.2.10 Plan-finalization

The *plan-finalization* state is where certain types of inconsistency within the flight plan are detected and final improvements to the plan are implemented. Unlike the *mission-path-adjust* state, it needs the distance, time and fuel estimates from the *estimations* state to perform its function.

In general, many algorithms could be implemented within the state; at the current development stage for SAMMS, two separate algorithms are present. The first algorithm deals with situations in which the total estimated fuel consumption is higher than the current fuel on-board; it does so by reducing flight speed, which is assumed to reduce fuel consumption over the same distance. The second algorithm deals with situations in which an Objective has a certain time priority (a time limit to successfully complete the Objective) and the time estimates reveal that the priority will not be respected; in this case flight speed is increased.

Before going into the details of both algorithms, it is important to underline the fact that the flight plan at this stage is almost completely defined: improvements to the plan at this stage can only be minor, and in fact both algorithms have a rather low level of authority in changing it. It can be said that the function of these algorithms is to correct minor inconsistencies; if successful, they will prevent intervention of the Mission Manager Agent, which instead has a much higher degree of authority but operates rather drastically, effectively changing the mission Objectives. For example, if a flight plan requires slightly more fuel than what is currently available, the algorithm in the plan-finalization state might be able to solve the issue (slowing down the flight, making the completion possible); however, mission Objectives might just be unrealistic, in which case the algorithm will be unsuccessful and the MMA will have to intervene (usually by cancelling one or more Objectives). The same goes for non-respected time priorities: the increase in speed might be sufficient to correct the issue, especially since the plan-sequencing state will likely already have ordered Objectives so as to ensure that time priorities are respected; however, the time limit might simply be unrealistic, in which case it is the MMA's duty to decide to ignore the priority (and inform the UAV operator).

The *fuel-check* operator is the main operator for the first algorithm; it checks whether the total fuel consumption estimate for the flight plan is higher or lower than the current fuel on-board. In the latter case, it simply signals that fuel to execute the plan is sufficient; in the former case, it detects the problem and sets up working memory so that the *reduce-speed* operator will fire. The reduce-speed operator is an iterative operator (meaning that it fires once for every Action to be modified) that modifies all Actions in the flight plan, apart from the ones pertaining to the take-off and approach phases. It changes directly the *speed* property of Actions, decreasing it by a fixed amount (currently set to 10% of the speed). When the flight plan has been entirely updated with the new flight speeds, the *reduce-speed-reset* operator fires; this closes the plan-finalization state and cancels the distance, time and fuel estimates from working memory (since they are no longer valid). The estimations state will be entered again, the estimates will be recalculated and then the plan-finalization state will be entered again. The fuel-check operator will fire again and determine if the issue has been solved or not; if not, then reduce-speed will fire again and the cycle will be repeated, achieving a further reduction of flight speed. However, speed cannot be reduced indefinitely; to prevent this, a limit has been placed on the number of times the reduce-speed operator can change the flight plan; currently, the fuel-check operator will not activate reduce-speed more than three times, so flight speed will not be reduced by more than roughly 27%.

The *priority-check* operator is the main operator for the second algorithm; it fires for any Objective which has a time priority. Once it fires, a substate is entered, where the time priority will be checked against the correct estimated time. Depending on the Objective type, the precise time estimate to be used for the check changes; for Analyze and Attack Objectives, it is the time estimate for the Recon or Attack Action; for Orbit Objectives, it is the time estimate for the last Travel Action before the Circle Action; for Search Objectives, it is the time estimate for the last Travel Action in the search. If the time priority is respected, the Objective will be marked as successful; otherwise, the fact that the priority is not respected is evidenced and working memory is set up so that the *increase-speed* operator will fire. The increase-speed operator is very similar to the reduce-speed operator: it works iteratively and changes the speed property of Actions. Naturally, it increases speed rather than reducing it; speed is increased by a fixed amount (which is currently set to 10%). However, it does not change speed for the entire flight plan, but only for the Actions that are placed before the Objective with the non-respected priority. For example, in a plan with three Objectives ordered in the sequence 1-2-3 and in which the second Objective has a non-respected time priority, the increase-speed operator will change speed only for the Actions relative to Objectives 1 and 2. When the flight plan has been entirely updated with the new flight speeds, the *increase-speed-reset* operator fires; this closes the plan-finalization state and cancels the distance, time and fuel estimates from working memory (since they are no longer valid). The estimations state will be entered again, the estimates will be recalculated and then the plan-finalization state will be entered again. After fuel-check, the priority-check operator will fire again and determine if the issue has been solved; if not, a new increase will be attempted, however for this algorithm only a second iteration is allowed; if a speed increase of 19% is not sufficient, then the matter will be left to the MMA.

Finally, it is to be noted that fuel-check and priority-check are executed in this order; this is because a fuel issue is deemed to be more important than a priority issue, since it would impact the ability of the UAV to finish its mission safely. If fuel-check detects an inconsistency and the reduce-speed operator is used, the increase-speed operator will be inhibited from firing, since they obviously perform opposite Actions.

4.3 Planner Testing Strategy

In Section 4.1, it is stated that input for the Planner Agent consists of 306 different variables. Considering that many of these are floating point variables that can each vary within its own range, and that each of these can vary as the mission is being executed, the impracticality (or even impossibility) of testing every possible combination of inputs becomes obvious.

In order to test and validate the SAMMS software, a set of six representative scenarios was then prepared. The scenarios are designed to verify the operation of SAMMS in the most varied conditions. In general, each scenario is oriented towards testing specific algorithms or demonstrating specific functionality; however, the basic principles of operation (such as the plan-sequencing and actions-definition states) are tested throughout the range of scenarios.

All scenarios are available in several variations, which can differ in many manners. The scenarios are numbered from 1 to 6 and lowercase letters are used to distinguish between the scenario variations. For example, scenario 1 refers to the overall scenario, while scenario 2d refers to the *d* variation of scenario 2.

Finally, a seventh scenario will also be introduced. Unlike the others, this scenario was not developed internally for testing purposes, but originates externally from the project. As stated in Section 1.4, in 2010 the University of Sheffield was involved in a SEAS-DTC (Systems Engineering for Autonomous Systems Defence Technology

Centre) backed study on the use of Intelligent Agents for intelligent power management. The SAMMS software was used during the project to provide a simulation of an autonomous UAV. A UAV mission scenario was provided by project stakeholders and this was then translated into SAMMS terms (e.g. in Objectives). This is captured in the Technical Report "Agent Architecture for Intelligent Power Management", which is Technical Report Number 4, written by M. Ong, P. Gunetti and H. Thompson, and delivered to SEAS-DTC in July 2010. While the scenario is not particularly interesting in terms of demonstrating specific SAMMS algorithms, it is significant since it originates externally and because it shows how a mission scenario could be easily and quickly "imported" into SAMMS.

The scenarios will now be described in detail; for each scenario, Objectives will be defined and their geographic position shown. The same holds true for Entities and the take-off and landing locations. It is to be noted that at this stage no consideration regarding the order in which Objectives will be executed will be drawn.

Some scenario variations have been specifically defined in order to test the functionality of the Mission Manager Agent. These variations will be introduced in the relative chapter, Chapter 5.

4.3.1 Scenario 1

Scenario 1 is the simplest of the scenarios; it is aimed at demonstrating basic operation of SAMMS and does not test particular algorithms. The scenario involves three Objectives and two Entities. Figure 4.13 shows the locations involved, including the starting location (Sheffield Airport), two Target-Analyze Objectives (each corresponding to an Entity) and the landing location, which is set at Doncaster Airport by a Transit Objective.

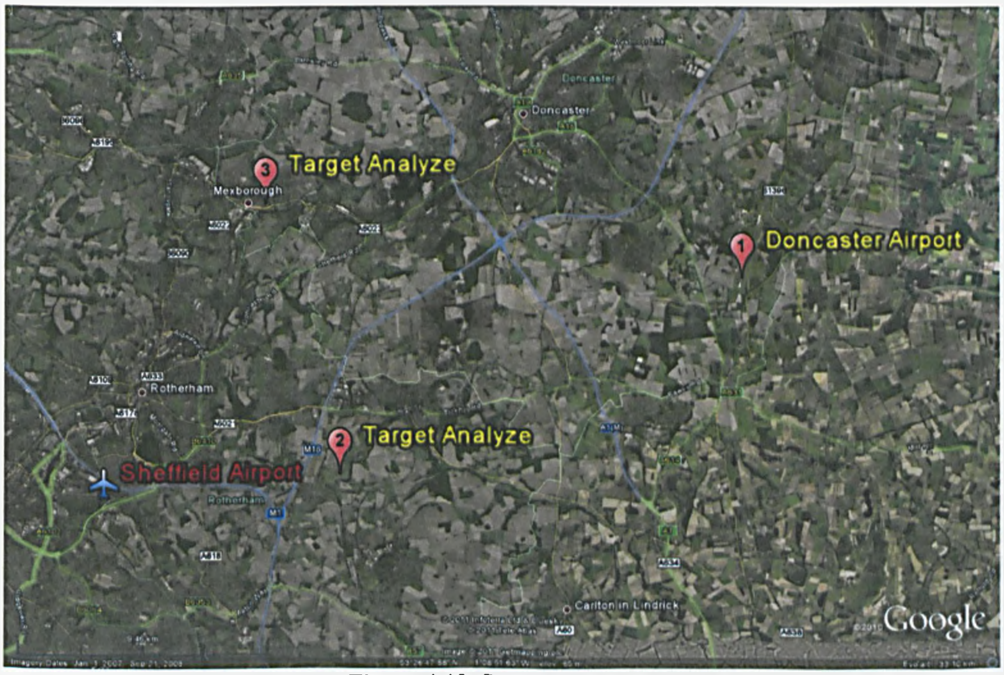


Figure 4.13. Scenario 1 map.

Details regarding the locations and Objectives are provided in Table 4.14; these include the coordinates for starting parking location, starting runway and Objectives (the *position* and *box-corner* properties for the Transit Objective indicate respectively landing location and final parking location), the altitudes and headings of the runways (*variable 1* in the Transit Objective indicates Altitude, *variable 2* indicates runway heading), the priorities of Objectives (no time priority is given, all Objectives have Execution priority 5 e.g. they will be executed at all costs) and the Entity Tag assignment for the Target-Analyze Objectives (Entity 1 is assigned as the target for Objective 2, Entity 2 as the target for Objective 3).

Table 4.14. Details of Scenario 1 locations and Objectives.

Starting	ID	Property	Transit	Target-Analyze	Target-Analyze
Park: 53°23’42”N 1°22’51”W	1	Type	5	1	1
	2	ID	1	2	3
	3	Position	53°27’58”N 1°00’32”W	53°24’05”N 1°14’33”W	53°29’39”N 1°17’39”W
Runway: 53°23’37”N 1°22’48”W	4	Priority	0, 5	0, 5	0, 5
	5	Duty	/	/	/
	6	Area Type	/	/	/
Runway Heading: 280	7	Variable 1	13	/	/
	8	Box-Corner	53°28’20”N 1°00’34”W	/	/
Runway Altitude: 71	9	Variable 2	20	/	/
	10	Target	/	1	2

There are three variations of the scenario:

- variation 1a only includes Objectives 1 and 2
- variation 1b has all three Objectives
- variation 1c starts with Objectives 1 and 2, then sees a re-planning event when Objective 3 is added while the mission is being executed; the new Objective is added while performing Action 7 in the flight plan, with the Exag committed to the completion of the current Objective

Scenario variation 1c is clearly the most significant of the variations and will be used during testing.

4.3.2 Scenario 2

Scenario 2 was designed to test the search pattern algorithms and the behaviour of the Planner when time priorities are assigned to Objectives. The scenario has the UAV take-off and land at the same airport and involves two Objectives and no Entities. Figure 4.14 shows the locations involved, including the starting and landing location (Sheffield Airport), a Search Objective of type box (Objective 1) and two variations of a Search Objective of type circle (Objective 2, which is changed during the scenario). The two possible variations of Objective 2 are named Objective *2a* and Objective *2b*.

Details regarding the locations and Objectives are provided in Table 4.15; these include the coordinates for starting parking location, starting runway and Objectives, the altitude and heading of the runway, the priorities of Objectives (only Objective 1 has a time priority, all Objectives have Execution priority 5) and the definitions of the search areas and search patterns. The time priority for Objective 1 is particularly significant; as will be shown in Section 4.4, the flight plan will vary depending on its value, with three main possible cases:

- Objective 2 is executed first (being the nearest from take-off), then Objective 1 is completed within the time limit without further intervention
- Objective 2 is executed first, but to complete Objective 1 within the time limit the increase-speed algorithm must change the flight plan

- Objective 1 is executed first since the time limit could not be respected otherwise

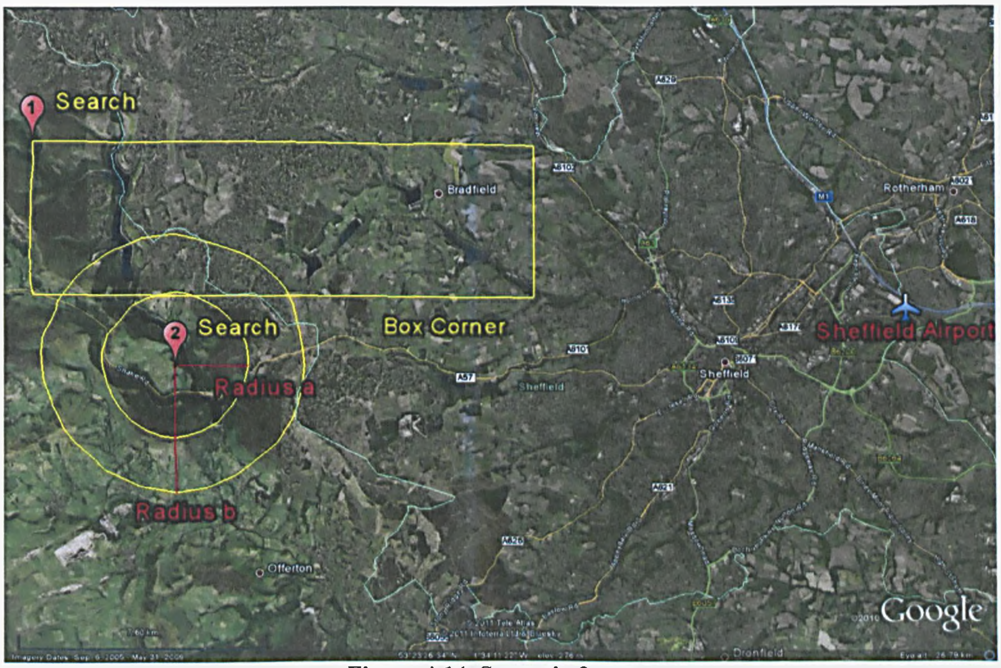


Figure 4.14. Scenario 2 map.

The effect of changing the value of this time priority will be studied without actually introducing a scenario variation.

There are four variations of the scenario:

- variation 2a includes Objectives 1 and 2a
- variation 2b includes Objectives 1 and 2b
- variation 2c starts with Objectives 1 and 2a, then sees a re-planning event when Objective 2 is changed from the 2a version to the 2b version; this change occurs while performing Action 10 of the original flight plan, with the Exag not committed to the completion of the current Objective
- variation 2d is similar to variation 2c, but introduces the concepts of search-and-analyze and search-and-attack Objectives; these are related to MMA operation, and will be explained in Chapter 6

During tests, scenario variation 2a will be used to show how search patterns are planned

Table 4.15. Details of Scenario 2 locations and Objectives.

Starting	ID	Property	Search 1	Search 2a	Search 2b
Park: 53°23’42”N 1°22’51”W	1	Type	4	4	4
	2	ID	1	2	2
	3	Position	53°26’36”N 1°47’07”W	53°22’55”N 1°43’12”W	53°22’55”N 1°43’12”W
Runway: 53°23’37”N 1°22’48”W	4	Priority	800, 5	0, 5	0, 5
	5	Duty	/	/	/
	6	Area Type	1	2	2
Runway Heading: 280	7	Accuracy	1200	1000	700
	8	Box-Corner	53°24’02”N 1°33’14”W	/	/
Runway Altitude: 71	9	Radius	/	2250	4000
	10	Target	/	/	/

and how the time priority value is used to determine the order in which Objectives are executed. Scenario variation 2c will also be investigated.

4.3.3 Scenario 3

Scenario 3 was designed to test the behaviour of the Planner Agent in situations with several Objectives, some of which have time priorities. The scenario includes a Transit Objective, so the UAV takes-off at an airport and lands at another; there are five other Objectives of different types, and seven Entities (four of which constitute targets for the Objectives, while the other three constitute threats to the UAV). Figure 4.15 shows the locations involved, including the starting location (Sheffield Airport), an Orbit Objective (Objective 1), a Target-Attack Objective (Objective 6), three Target-Analyze Objectives (Objectives 2, 4 and 5) and the landing location, which is set at Doncaster Airport by a Transit Objective (Objective 3). Also, the threat areas represented by Entities 5, 6 and 7 are shown; these Entities are defined as circular areas, so for them the centre location and an area of effect radius are provided.

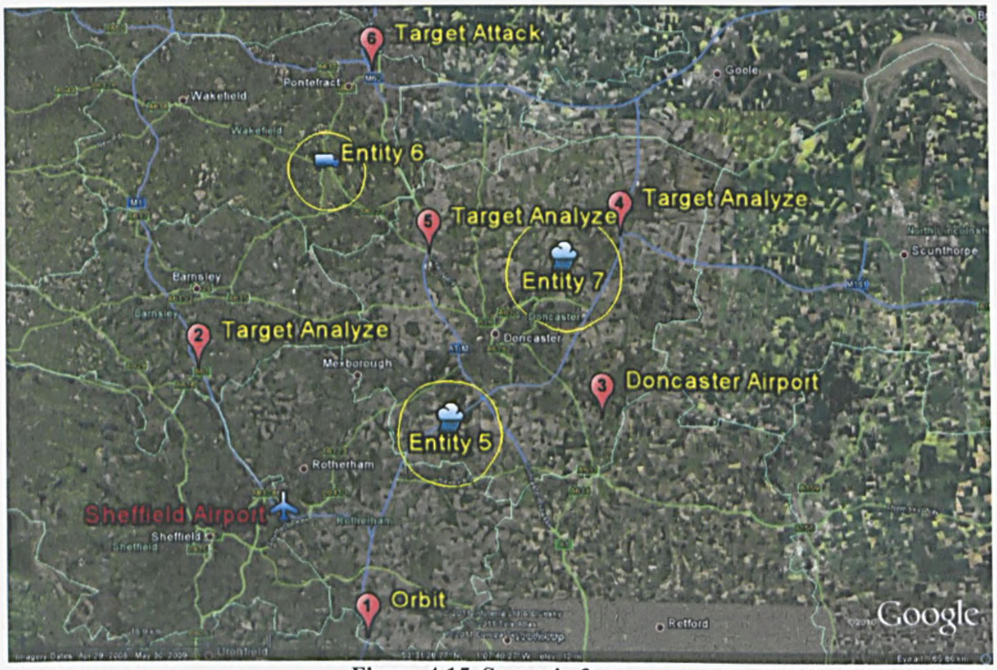


Figure 4.15. Scenario 3 map.

Details regarding the Objectives are provided in Table 4.16; the coordinates for the starting airport and the landing airport are the same as for Scenario 1 (in Table 4.14, the Transit Objective is the same as the Transit Objective for this scenario, apart from the ID tag). Details for the other Objectives include locations, priorities and the time limit for the Orbit Objective. Time priorities are particularly important in this scenario; in fact, a re-planning event is triggered by changing the priorities of some Objectives. In the table, the priorities within parentheses represent the eventual new value for the time priority after the re-planning event has occurred (in this case, re-planning is due to a change in the Objectives which will have been performed by the UAV operator). Entities 1 to 4 are positioned in coincidence with the Objective for which they constitute the target; notice however that Entity Tag numbers and Objective Tag numbers do not coincide (in fact, the Target property of Objectives allows to assign any Entity).

Details for the three dangerous Entities are provided in Table 4.17; notice that since all Objectives have an execution priority of 5, the areas will be avoided if possible, but not at the cost of ignoring an Objective. Entities 5 and 7 are areas of bad weather, therefore their behaviour is set as neutral and their threat level is low; Entity 6 is an enemy ground vehicle, so its threat level is higher. All Entities are static.

Table 4.16. Details of Scenario 3 Objectives.

ID	Property	Orbit	Analyze	Analyze	Analyze	Attack
1	Type	3	1	1	1	2
2	ID	1	2	4	5	6
3	Position	53°18'52"N 1°16'56"W	53°29'54"N 1°28'43"W	53°35'29"N 0°59'14"W	53°34'40"N 1°12'41"W	53°42'15"N 1°16'41"W
4	Priority	-1, 5	0 (-1), 5	2500 (0), 5	1500, 5	0 (2500), 5
5	Duty	1	/	/	/	/
6	Area Type	/	/	/	/	/
7	Altitude	800	/	/	/	/
8	Box-Corner	/	/	/	/	/
9	Time	600	/	/	/	/
10	Target	/	1	2	3	4

There are five variations of the scenario:

- variation 3a includes Objectives with the original time priorities; the dangerous Entities are not present
- variation 3b includes Objectives with the modified time priorities; the dangerous Entities are not present
- variation 3c starts with Objectives set up with the original time priorities, then sees a re-planning event during which time priorities are switched to their modified values; this change occurs while performing Action 8 of the original flight plan, with the Exag not committed to the completion of the current Objective; notice that Objective 2 gets an immediate priority, but this should not happen before Objective 1 is completed, since two Objectives cannot have an immediate priority at the same time; the dangerous Entities are not present

Table 4.17. Details of Scenario 3 dangerous Entities.

ID	Property	Entity 5	Entity 6	Entity 7
1	Entity Type	4	2	4
2	Entity Tag	5	6	7
3	Entity Position	53°27'15"N 1°11'11"W	53°37'59"N 1°19'51"W	53°33'54"N 1°3'13"W
4	Movement Info	0, 0	0, 0	0, 0
5	Entity Behaviour	2	3	2
6	Threat Level	2	4	2
7	Area of Effect	4000	3000	4500
8	Stance	/	/	/

- variation 3d includes Objectives with the original time priorities (as in variation 3a), but introduces the three dangerous Entities, which have a threat level and thus can trigger the mission-path-adjust algorithm
- variation 3e includes Objectives with the original time priorities, but introduces a reduced amount of available fuel, causing the fuel-check algorithm to reduce flight speed in order to save fuel

During tests, scenario variations 3a, 3b and 3c will be used to show how the time priorities influence the plan generation process (and the plan-sequencing phase in particular). Furthermore, variation 3d will be used to demonstrate how SAMMS avoids dangerous areas and variation 3e will be used to demonstrate the fuel-check algorithm.

4.3.4 Scenario 4

Scenario 4 was designed to test the ability of the Planner Agent to deal with a large number of Objectives (up to the current maximum of ten); also, the scenario involves the presence of dangerous Entities, so functionality of the mission-path-adjust algorithm is also tested. The scenario has the UAV take-off and land at the same airport and involves ten Objectives and nine Entities (six being targets, while the other three represent threats to the UAV). Figure 4.16 shows the locations involved, including the starting and landing location (Sheffield Airport), five Target-Analyze Objectives, one Target-Attack Objective, two Orbit Objectives and two Search Objectives. The locations of the dangerous Entities are also shown (the other Entities are located in coincidence with the relative Objective).

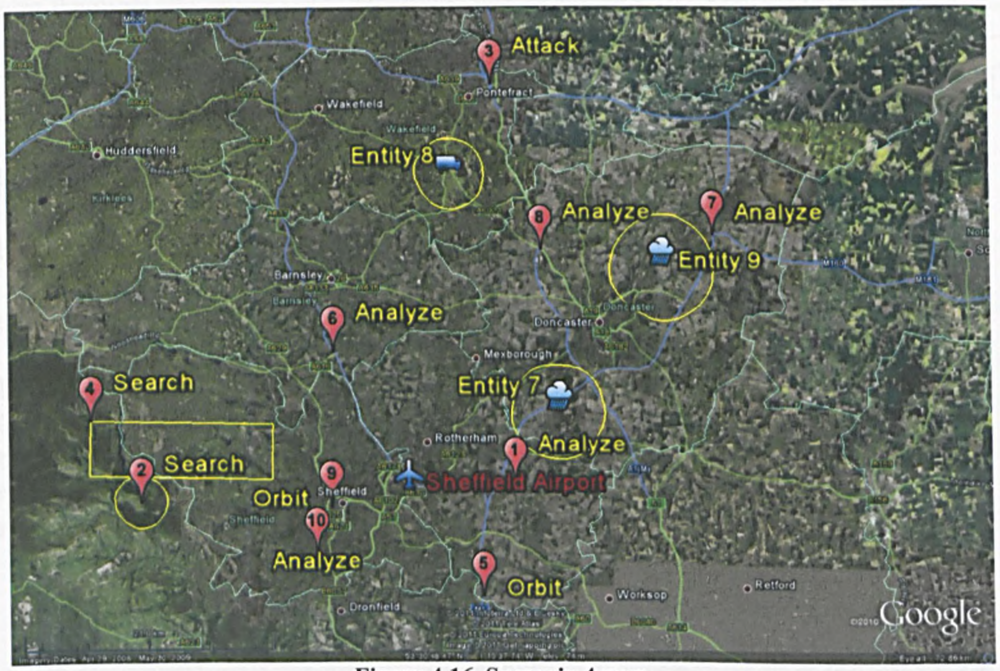


Figure 4.16. Scenario 4 map.

Details regarding the Objectives are provided in Table 4.18; the coordinates for the starting and landing airport are the same as for Scenario 2 (see Table 4.15). Details for the Objectives include all relevant data (locations, time priorities, target ID tags, area description, orbit time limits). Time priorities are used in the scenario, and have great influence over the plan generation process (as will be clear in the results presented in Section 4.4). It is to be noted that great care must be placed in determining the time limit for orbit Objectives; this is especially true for Objectives that are introduced after the mission has begun, since a wrong assignment might lead to a set of Objectives which is impossible to complete (for example, if Objective 9 is added during flight at time 3000, but has a time limit of 2500, an error would be detected and a flight plan will be generated, but clearly not a correct one).

The dangerous Entities in the scenario are the same that were used for scenario 3, with the only difference that their ID tags are different (Entity 5 is now Entity 7, Entity 6 is now Entity 8, Entity 7 is now Entity 9). Details regarding the Entities can be found in Table 4.17.

There are three variations of the scenario:

- variation 4a includes all Objectives apart from the last two, Objectives 9 and 10); Objective 5 has a time limit of 1000 sec and no time priority
- variation 4b includes all Objectives; notice that Objective 3 has its time priority changed (from 0 to 4000) and Objective 5 has an increased time limit of 1600 (to avoid conflicting with Objectives 9 and 10)
- variation 4c starts with the same situation as variation 4a, then sees a re-planning event during which Objectives 9 and 10 are added and Objective 3 has its priority changed; the time at which this change occurs is modified, so as to see how re-planning events occur depending on how much the mission has progressed; specific time values are adjusted accordingly

Table 4.18. Details of Scenario 4 Objectives.

ID	Property	Analyze	Search	Attack	Search	Orbit
1	Type	1	4	2	4	3
2	ID	1	2	3	4	5
3	Position	53°25'05"N 1°14'33"W	53°22'55"N 1°43'12"W	53°42'15"N 1°16'41"W	53°26'36"N 1°47'07"W	53°18'52"N 1°16'56"W
4	Priority	0, 5	0, 5	0 (4000), 5	0, 5	0 (1400), 5
5	Duty	/	/	/	/	1
6	Area Type	/	2	/	1	/
7	Variable 1	/	1000	/	2500	500
8	Box-Corner	/	/	/	53°24'02"N 1°33'14"W	/
9	Variable 2	/	2250	/	/	1000 (1600)
10	Target	1	/	5	/	/

ID	Property	Analyze	Analyze	Analyze	Orbit	Analyze
1	Type	1	1	1	3	1
2	ID	6	7	8	9	10
3	Position	53°29'54"N 1°28'43"W	53°35'29"N 0°59'14"W	53°34'40"N 1°12'41"W	53°22'53"N 1°28'42"W	53°20'45"N 1°29'48"W
4	Priority	0, 5	0, 5	0, 5	-1, 5	1000, 5
5	Duty	/	/	/	1	/
6	Area Type	/	/	/	/	/
7	Variable 1	/	/	/	1500	/
8	Box-Corner	/	/	/	/	/
9	Variable 2	/	/	/	2500	/
10	Target	2	3	4	/	6

During tests, all variations will be used, to show how the same set of Objectives can lead to different flight plans depending on certain parameters (in particular, time priorities).

4.3.5 Scenario 5

Scenario 5 was designed to test some Planner Agent abilities: the behaviour in cases with multiple re-planning events, the possibility to cancel Objectives, the possibility to change the landing location while in flight. It is also set in a different area from the usual Sheffield area. The scenario has the UAV take-off from Ann Arbor airport (located close to Detroit, Michigan, USA) and land at either the same airport or the nearby Linden Price airport. It comprises six Objectives and three Entities. Figure 4.17 shows the locations involved, including the starting and landing locations, two Target-Analyze Objectives, a Target-Attack Objective, a Search Objective of type circle, an Orbit Objective and a Transit Objective.

Details regarding the locations and Objectives are provided in Table 4.19; these include the coordinates, altitude and heading for the starting and landing airports, and

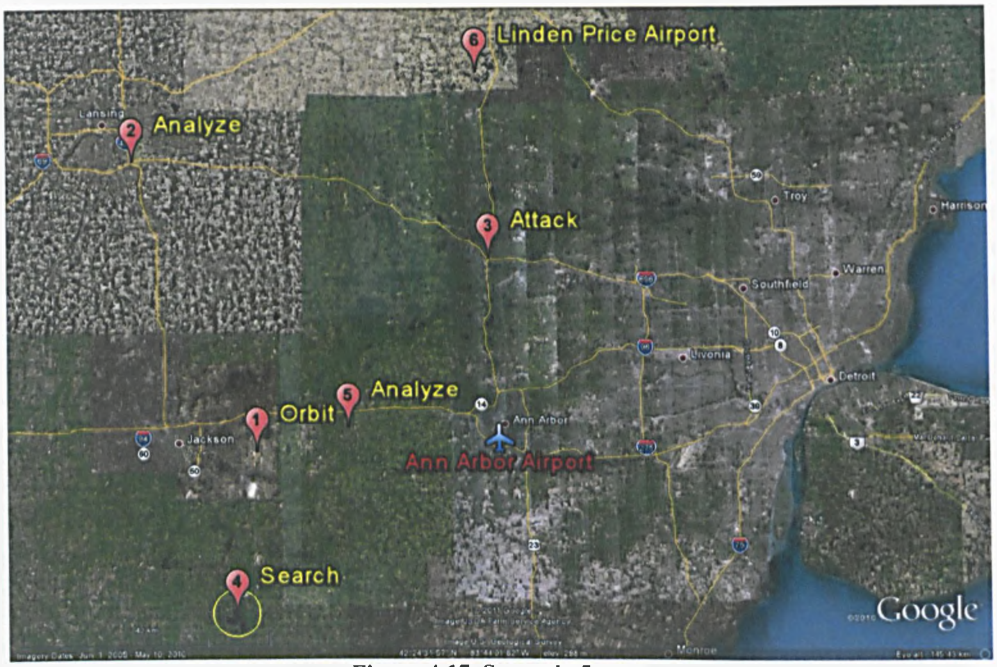


Figure 4.17. Scenario 5 map.

the properties of all Objectives (landing location properties are defined as part of Objective 6).

There are five variations of the scenario:

Table 4.19. Details of Scenario 5 locations and Objectives.

Starting	ID	Property	Orbit	Analyze	Attack
Park: 42°13'33''N 83°44'30''W	1	Type	3	1	2
	2	ID	1	2	3
	3	Position	42°04'00''N 84°14'26''W	42°40'30''N 84°29'44''W	42°31'27''N 83°45'16''W
Runway: 42°13'32''N 83°44'23''W	4	Priority	1500, 5	0, 5	0, 5
	5	Duty	1	/	/
	6	Area Type	/	/	/
Runway Heading: 240	7	Altitude	1200	/	/
	8	Box-Corner	/	/	/
Runway Altitude: 250	9	Time	1800	/	/
	10	Target	/	2	1

ID	Property	Search	Analyze	Transit
1	Type	4	1	5
2	ID	4	5	6
3	Position	41°59'04''N 84°17'08''W	42°16'10''N 84°03'05''W	42°48'27''N 83°46'29''W
4	Priority	2500, 5	-1, 5	0, 5
5	Duty	/	/	/
6	Area Type	2	/	/
7	Variable 1	1500	/	276
8	Box-Corner	/	/	42°48'33''N 83°46'17''W
9	Variable 2	4000	/	90
10	Target	/	3	/

- variation 5a includes Objectives 1, 2, 3 and 4
- variation 5b includes Objectives 1, 2, 3, 4 and 5
- variation 5c includes Objectives 1, 2, 4 and 5
- variation 5d includes Objectives 1, 2, 4, 5 and 6
- variation 5e is a very dynamic scenario during which three re-planning events occur; starting with Objectives 1, 2, 3 and 4, Objective 5 is added while performing Action 6 in the original flight plan and with the Exag not committed to the current Objective; then, while performing Action 17 in the updated flight plan and with the Exag committed to the current Objective, Objective 3 is removed; finally, while performing Action 21 in the third instance of the flight plan and with the Exag committed to the current Objective, Objective 6 is added

During tests, scenario variation 5e will be the most significant.

4.3.6 Scenario 6

Scenario 6 was designed to test the functionality of the Planner Agent under particular conditions: the scenario involves multiple re-planning events, a large number of Objectives and large distances to be covered (it is also set over all Europe). The scenario has the UAV take-off and land at Turin airport (located in Italy); it comprises nine Objectives and four Entities. Figures 4.18 and 4.19 show the locations involved, including the starting and landing location, two Target-Analyze Objectives, two Target-Attack Objectives, two Search Objectives (of types box and circle) and three Orbit Objectives. Because some Objectives (in particular, Objectives 1, 3, 6 and 9) are located very far from the majority of others, two figures at different scales are needed in order to avoid confusion. Objective 1 in particular is placed very far from the others, so as to test the behaviour of the Nearest Neighbour algorithm under such conditions.

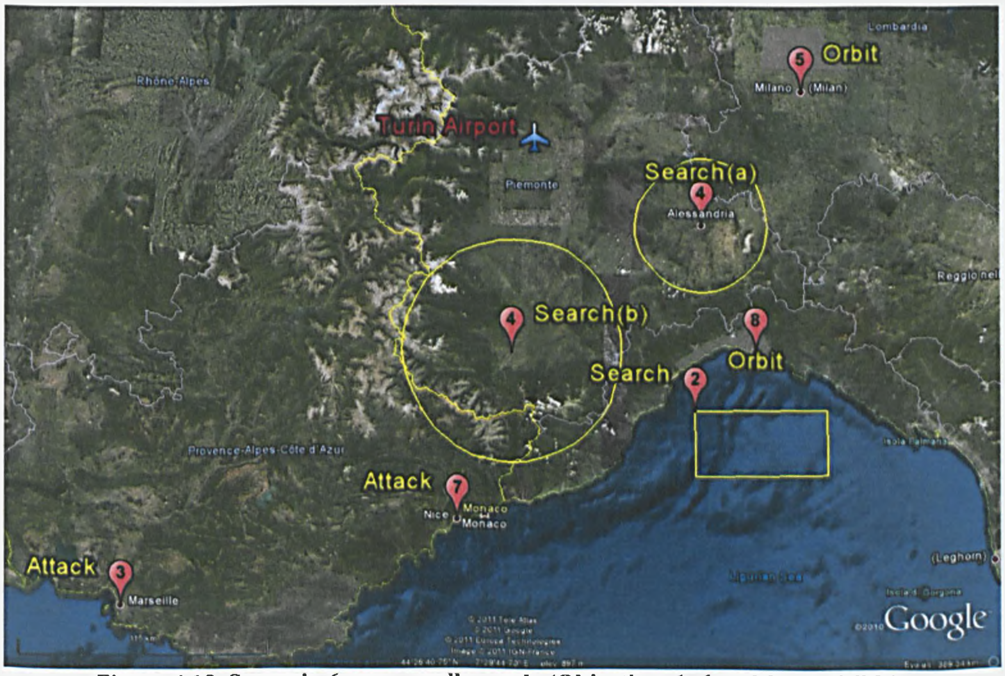


Figure 4.18. Scenario 6 map, smaller scale (Objectives 1, 6 and 9 not visible).

Details regarding the locations and Objectives are provided in Table 4.20; these include the coordinates, altitude and heading for the starting and landing airport, and the properties of all Objectives. Notice that Objectives 4 and 6 are subject to changes during the re-planning events, hence the updated values are shown in parentheses (the updated *position* property is shown as the *box-corner* property). Objective 4 in particular is

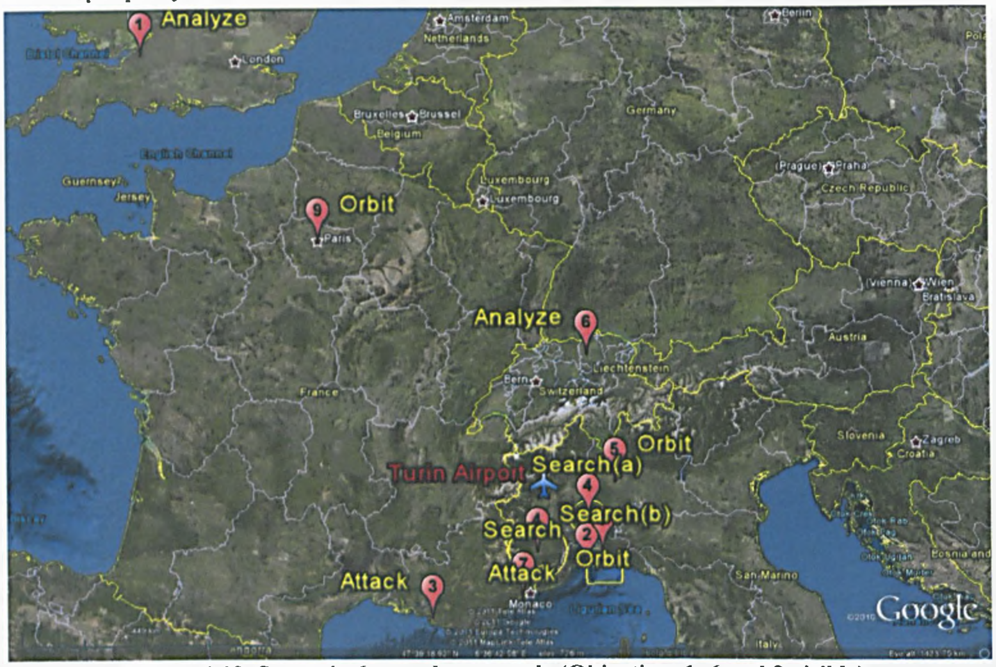


Figure 4.19. Scenario 6 map, larger scale (Objectives 1, 6 and 9 visible).

changed significantly, so two variations are identified (4a and 4b).

There are four variations of the scenario:

- variation 6a includes Objectives 1, 2, 3, 4a, 5, 6 and 7

Table 4.20. Details of Scenario 6 locations and Objectives

Starting		Property	Analyze	Search	Attack	Search
Park: 45°11'20"N 7°39'12"E	1	Type	1	4	2	4
	2	ID	1	2	3	4
	3	Position	51°27'19"N 2°35'30"W	44°09'49"N 8°35'38"E	43°17'54"N 5°22'59"E	44°54'47"N 8°37'01"E
Runway: 45°11'24"N 7°38'56"E	4	Priority	0, 5	0, 5	0, 5	0, 5
	5	Duty	/	/	/	/
	6	Area Type	/	1	/	2
Runway Heading: 4	7	Accuracy	/	20000	/	6000 (10000)
	8	Box-Corner	/	43°54'31"N 9°21'35"W	/	(44°23'19"N 7°32'49"E)
Runway Altitude: 285	9	Radius	/	/	/	30000 (50000)
	10	Target	2	2	4	/

	Property	Orbit	Analyze	Attack	Orbit	Orbit
1	Type	3	1	2	3	3
2	ID	5	6	7	8	9
3	Position	45°27'49"N 9°11'17"E	47°22'08"N 8°32'16"E	43°42'12"N 7°15'58"E	44°24'25"N 8°56'02"E	48°51'23"N 2°21'03"E
4	Priority	5000, 5	0 (18000), 5	0, 5	-1, 5	68000, 5
5	Duty	1	/	/	1	1
6	Area Type	/	/	/	/	/
7	Altitude	1000	/	/	2000	3000
8	Box-Corner	/	/	/	/	/
9	Time	7000	/	/	11000	80000
10	Target	/	1	3	/	/

- variation 6b includes Objectives 1, 2, 3, 4b, 5, 6, 7 and 8
- variation 6c includes Objectives 1, 2, 3, 4b, 5, 6, 8 and 9
- variation 6d represents the evolution from variation 6a to variation 6c, by having two re-planning events; the scenario start with Objectives 1, 2, 3, 4a, 5, 6 and 7, then a first re-planning event is caused by the addition of Objective 8 and changes to Objectives 4 (going to 4b) and 6, occurring while performing Action 8 of the original plan; the plan changes a second time while performing Action 15 of the updated flight plan, when Objective 9 is added and Objective 7 is removed

During tests, scenario variation 6d will be the most significant. It is to be noted that this scenario will be used only during testing of the Planner Agent: due to the high distances and long times involved, it is unsuitable for testing SAMMS in its entirety.

4.3.7 Scenario 7 (SEAS-DTC scenario)

As previously stated, scenario 7 was designed by an external stakeholder for use in a project separate from SAMMS. The project was related to autonomous energy management for UAVs and thus is more concerned about the altitude and flight speed of the UAV than about the actual position. The original scenario was provided in the format of an actual flight plan, mainly consisting of four weapon delivery events; the weapon deliveries are scheduled to happen at fixed times, so the UAV must loiter before each weapon delivery; the UAV is also expected to loiter before landing. Speed and altitude profiles are given for the entire mission, detailing in particular how weapon deliveries should be carried out (first descend towards target at low speed, then pass over target at full speed, then climb back to cruise altitude).

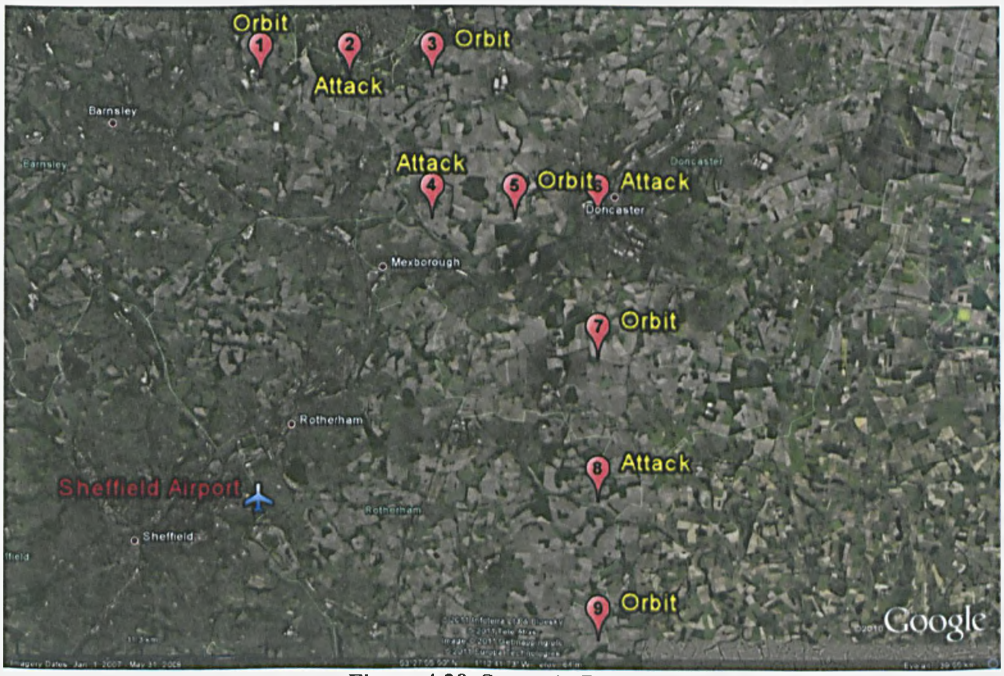


Figure 4.20. Scenario 7 map.

In SAMMS terms, weapon deliveries correspond to Target-Attack Objectives, while the loitering is achieved by Orbit Objectives. The scenario is then rendered within SAMMS by a set of 9 Objectives. The typical speeds, altitudes and distances that define the behaviour of the UAV (those from Table 4.6) had to be rearranged (within the

Pioneer model capabilities) in order to realize the required speed and altitude profiles. Figure 4.20 shows the locations involved: the UAV takes-off at Sheffield airport, heads north and then loiters waiting to execute the first Attack Objective; it repeat the loiter-attack cycle another three times, before loitering again and then landing back at Sheffield airport. As said, the four weapon deliveries correspond to four Target-Attack Objectives, while the loitering corresponds to five Orbit Objectives.

Table 4.21. Details of Scenario 7 locations and Objectives

Starting	ID	Property	Orbit	Attack	Orbit	Attack
Park:	1	Type	3	2	3	2
53°23’42”N	2	ID	1	2	3	4
1°22’51”W	3	Position	53°34’17”N 1°22’46”W	53°34’17”N 1°19’04”W	53°34’17”N 1°15’37”W	53°30’51”N 1°15’37”W
Runway:	4	Priority	-1, 5	0, 5	0, 5	0, 5
53°23’37”N	5	Duty	1	/	1	/
1°22’48”W	6	Area Type	/	/	/	/
Runway	7	Altitude	1500	/	1500	/
Heading: 280	8	Box-Corner	/	/	/	/
Runway	9	Time	800	/	1500	/
Altitude: 71	10	Target	/	1	/	2

ID	Property	Orbit	Attack	Orbit	Attack	Orbit
1	Type	3	2	3	2	3
2	ID	5	6	7	8	9
3	Position	53°30’51”N 1°12’11”W	53°30’51”N 1°08’45”W	53°27’25”N 1°08’45”W	53°23’58”N 1°08’45”W	53°20’32”N 1°08’45”W
4	Priority	0, 5	0, 5	0, 5	0,5	0, 5
5	Duty	1	/	1	/	1
6	Area Type	/	/	/	/	/
7	Altitude	1500	/	1500	/	1500
8	Box-Corner	/	/	/	/	/
9	Time	2000	/	2500	/	3500
10	Target	/	3	/	4	/

Details regarding the locations and Objectives are provided in Table 4.21; these include the coordinates, altitude and heading for the starting and landing airport, the location, time and altitude of loitering (Orbit Objectives) and the position and Entity ID tag for Target-Attack Objectives. Notice that Objective 1 is assigned an immediate time priority, since the scenario required Objectives to be carried out in a specified order. The scenario has no variations.

As an external scenario, scenario 7 is not designed to test any particular algorithm within SAMMS; its significance comes instead from how it allowed to demonstrate the easiness to set up a mission within SAMMS; mission Objectives can be defined without problems and, once properly set up, the generation of the flight plan (in SAMMS terms) and its execution are completely automatic and need no supervision.

4.4 Results of Planner Agent tests

While the other SAMMS main components (the Execution Agent and the Mission Manager Agent) cannot generally be tested on their own (mostly because this would mean generating a full flight plan as input), this is possible for the Planner Agent. In this section, results obtained from the Planner Agent operating detached from the rest of SAMMS will be presented. This type of testing will allow to validate the Planner without the longer simulations that are needed when running the entire system. Thus, the results presented here are purely representations of the Planner's output; since the Exag and MMA are not involved, the autopilot and UAV model are not involved as well, and the New-Plan-Trigger function is only simulated as needed for the Planner to generate a new plan during re-planning events.

The main tool used to show Planner results is the *Plan_Descripter* function; this is a Matlab S-function that takes data from a flight plan (intended in the SAMMS sense as a sequence of Actions) and graphically plots it. The plot is meant to show not only the route that the Planner has chosen to accomplish the mission, but also the Objective types, the distance and time estimates for each Action and the altitude and speed which at which the Action is to be executed.

In the following subsections, each of the scenarios introduced in Section 4.3 will be covered; many of them present several variations, which will be covered as necessary to demonstrate specific Planner functionality. The graphical plots of results will be adapted to suit the intentions which lead to the development of the scenario (for example, if a scenario variation was developed to test the increase-speed algorithm of Section 4.2.10, then the graphical plot will focus on showing Action speeds).

Before showing the results, a final consideration must be made; the Planner Agent expresses position as latitude/longitude/altitude coordinates. Latitude and longitude are expressed in radians within SAMMS, while altitude is expressed in metres. This is in contrast to the way the UAV model expresses position (using North/East/Altitude coordinates in metres). In fact, altitude is not very relevant while showing Planner results and can be shown separately, so all the position plots that will be shown here will be two-dimensional. However, an issue arises when plotting the position in 2D: plotting latitude/longitude values is very different from plotting North/East values. Latitude and longitude are defined on a spherical surface, so a 2D plot will lead to deformation (with North/East coordinates, this does not happen). In short, this is because a degree of latitude is always equal to the same distance at any latitude (about 60 nautical miles, or 111 km), but this is not true for a degree of longitude: in fact, the distance equal to a degree of longitude scales with the cosine of latitude:

$$dist_{lat} = arc_{lat} \cdot R_E$$
$$dist_{lon} = arc_{lon} \cdot R_E \cdot \cos lat$$

where $dist_{lat}$ and $dist_{lon}$ indicate the distance in metres, arc_{lat} and arc_{lon} indicate the distance expressed as an arc of latitude/longitude, R_E indicates the Earth radius and *lat* indicates the latitude of the parallel where the longitude arc is measured. The plots presented here do not take into account this effect; therefore, the flight plan plots will be deformed compared to the representation provided in the scenario descriptions of Sections 4.3.

In the many plots of the flight plans that are presented for each scenario, labels are present, indicating details of the Actions that constitute the flight plan. The notation used is the following: "*action-sequence-number*/*action-type*/*Objective-tag* SPD=*speed* ALT=*altitude*", where *action-sequence-number* is the sequence number assigned to the Action (thus defining the exact sequence in which Action are to be executed), *action-type* is one of the action types from Table 3.6 in Chapter 3, *Objective-tag* is the Objective property of the Action, and *speed* and *altitude* are the required speed and altitude (from the Speed and Altitude properties) for the Action. So, for example, the label "6/travel/4 SPD=40 ALT=500" indicates a Travel Action, which has sequence number 6 in the flight plan and is planned to be accomplished at speed 40 m/s and altitude 500 m (on sea level). Under this label, total distance and time estimates for the Action are placed (these are the estimates as calculated from the mission start). Labels are also used to indicate the position of Objectives. Not all Actions will be shown in complex scenarios to avoid confusion; in particular, for Search Objectives only the initial and final Travel Actions will be shown.

In the following subsections, results will be presented for each scenario; however, it is to be noted that not all of the scenario variations will be documented. While all of the scenario variations described in Section 4.3 have been tested, results that are included here are a subset of those obtained during testing. Portrayed scenario variations were chosen on the basis of their possibility to demonstrate specific capabilities in the Planner Agent.

4.4.1 Scenario 1 results

Scenario 1 is a simple scenario with three Objectives; only variation 1c will be shown, which is a variation where the third Objective is added during flight.

Figure 4.21 shows the plot of the flight plan for scenario 1c. The plan starts at the parking location of Sheffield airport, and the first Actions are Park and Taxi; the -1 value besides them in the figure indicates that these Actions are part of the take-off phase. Since this scenario involves short distances, it is possible to notice the Taxi Action as a short segment between the parking location and the runway location. The next Actions are Take-off and Climb, which are also part of the take-off phase; for the Climb Action, it is possible to see the distance and time estimates (which are those for the entire take-off phase, see Section 4.2.9). The take-off phase is expected to be completed after having covered 1557 m in 103 sec; notice that this is the distance and time after which the UAV will have reached the designated Mission-Start-Altitude, but

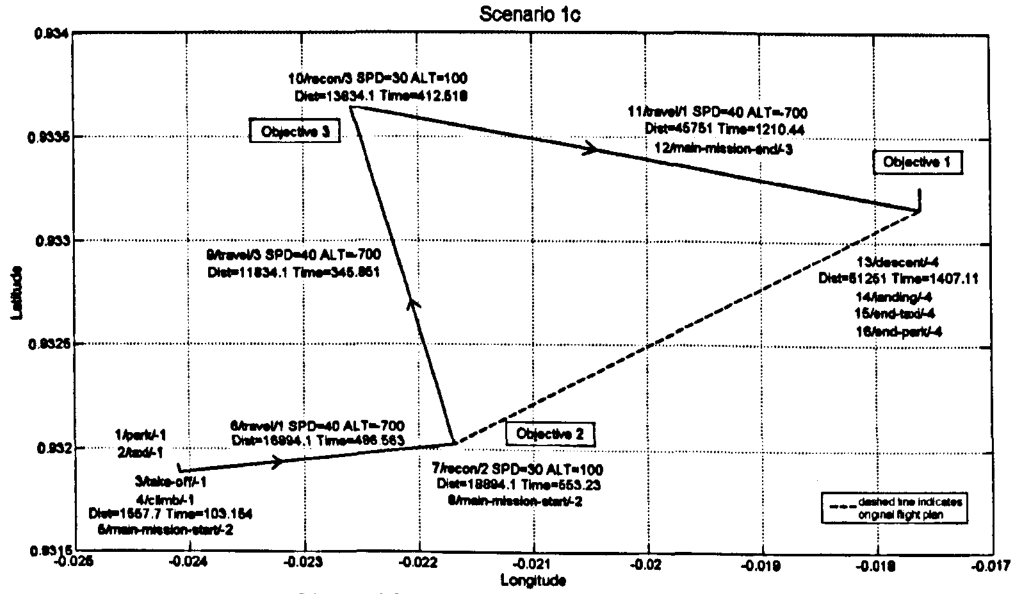


Figure 4.21. Plot of scenario 1c flight plan.

this is not necessarily the actual cruise altitude; thus, the UAV might actually climb after the climb Action is completed, but this will happen as part of the cruise rather the actual climb manoeuvre. More importantly, the Climb Action uses the Direct mode of the autopilot, while latter Travel Actions use the Auto mode (see Section 3.5 for details regarding the autopilot modes, and Chapter 5 for details regarding how the Exag actually converts the Action into UAV commands). The Climb Action represents then a climb to an altitude where control can be relinquished to the Auto mode, rather than a climb to cruise altitude. For example, in the scenario the Mission-Start-Altitude is set at 230 m; the UAV will perform a Climb to 230 m, stabilize flight at that altitude, then enter main mission mode and execute the first Travel Action. If this is set at a different cruise altitude (in the scenario it is set at 700 m), the UAV will resume climbing, however using a different control law: during the Climb Action, only the pitch-hold loop is active, while during main mission Actions the altitude-hold loop is also active.

After the take-off phase is completed, the main-mission-start Action signals the beginning of the main mission; the first executed Action is Action number 6, which is a Travel Action related to Objective 2; Objective 2 is prioritized over Objective 1 since not only it is closer to the starting location, but Objective 1 is a Transit Objective and therefore must be the last in a mission. The Action is carried out at a speed of 40 m/s and an altitude of 700 m above sea level (in the figure, this altitude is indicated as negative; this means that the altitude is above sea level, as opposed to altitude above ground, which is indicated as a positive altitude), and the Planner estimates to cover 16894 m for the Action, reaching the destination 486 sec after the mission has started (notice that the distance and time estimates are the total estimates). Having reached the Objective, the Recon Action is executed, at a speed of 30 m/s and an altitude of 100 m above ground.

At this point, the original flight plan would continue directly to Objective 1 and land there, since it is a Transit Objective; this appears on the plot as the dashed line. However, in variation 1c Objective 3 is added while performing Action 7, which is the Recon Action of Objective 2; thus the flight plan is updated adding Objective 3. The Exag is committed to complete Objective 2, so Action 7 is completed and the plan generation process will only consider Objectives 1 and 3. A new main-mission-start Action is added to signal the point where the new plan is in effect.

In the new plan, Objective 1 is again left last since it is a Transit Objective; the new Target-Analyze Objective is translated to a Travel Action and a Recon Action, which are executed at the same speeds as the ones for Objective 2. In the figure it is possible to see that the distance and time estimates for new Objectives are not consistent with the original ones; this is on purpose, to show how the estimations algorithm uses externally provided values for the total-distance and total-time values when a re-planning event occurs. In this case, those values are set to zero, so the estimates are the values obtained when starting the mission from the location of Objective 2; so for example the estimates indicate that Objective 3 will be reached 345 sec after finishing Objective 2, and will be completed after 412 sec. Normally, these values would be added to the real (not estimated) time used to reach that point in the mission.

After Objective 3 is completed, Objective 1 is executed; this results in a single Travel Action towards the destination airport, which is accomplished using the normal cruise speed and altitude. After the conclusion of this Action, the main-mission-end Action signals that the UAV is about to start the approach phase. This begins with the Descent Action; notice that in the figure distance and time estimates are available for this Action, however these estimates actually refer to the entire approach phase (e.g., the UAV will have reached it final parking location after the estimated 1407 sec, counted starting from the re-planning event after the conclusion of Action 7).

As stated in Section 4.3, this was an early testing scenario that is meant to test basic SAMMS functionality. During the generation of the first flight plan, the Planner activates the following states (described in Section 4.2):

- *take-off*, which adds the four initial Actions to the plan
- *main-mission*, which adds the main-mission-start and main-mission-end Actions to the plan, and controls the activation of plan-sequencing and actions-definition
- *plan-sequencing*, which decides that Objective 2 should be executed first since Objective 1 is a Transit Objective and should last
- *actions-definition*, which translates Objective 2 to two Actions (Travel and Recon) and Objective 1 to one Action (Travel)
- *approach*, which defines the four Action related to the approach phase
- *estimations*, which calculates the estimates

The other states may be entered but have no effect on the plan. When the re-planning event occurs, the following states are entered:

- *old-plan*, which copies Actions 1 to 7 from the original plan into the new plan
- *main-mission*, which adds a second main-mission-start Action as Action 8
- *plan-sequencing*, which only considers Objectives 1 and 3, choosing the latter as first
- *actions-definition*, which translates Objective 3 to two Actions (Travel and Recon) and Objective 1 to one Action (Travel)
- *approach*, which defines the four Action related to the approach phase
- *estimations*, which calculates the estimates

For the other scenarios, looking at the exact sequence of operators that are executed within the Soar agent is unpractical; however, to give an example of the internal workings of a Soar agent, the sequence of operator firings is reported for this scenario. Figure 4.22 shows a high-level view of the operators that are chosen during the plan generation process, also outlining how these are organized within the different substates.

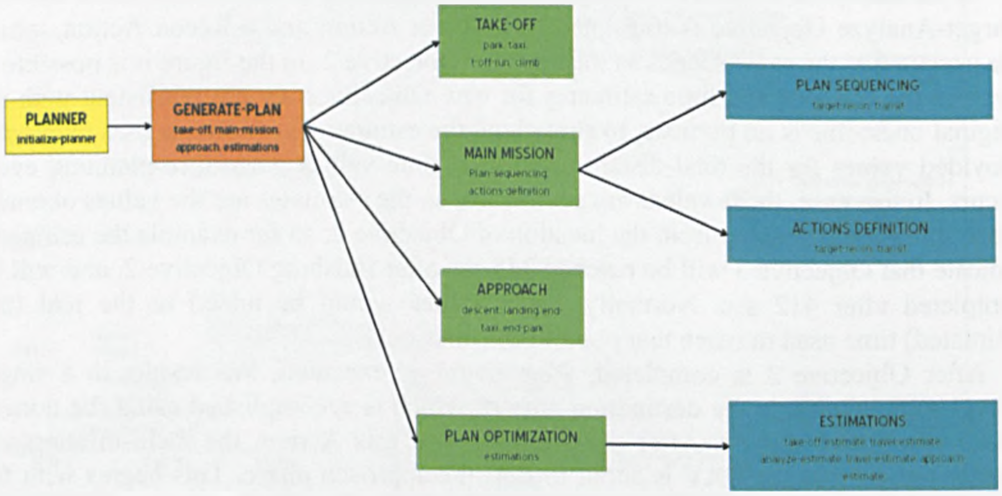

PLANNER
initialize-planner
GENERATE-PLAN
take-off main-mission approach estimations
TAKE-OFF
park taxi t-off-run climb
MAIN MISSION
Plan-sequencing actions-definition
APPROACH
descent landing end taxi end park
PLAN OPTIMIZATION
estimations
PLAN SEQUENCING
target-recon transit
ACTIONS DEFINITION
target-recon transit
ESTIMATIONS
take-off-estimate travel-estimate analyze-estimate travel-estimate approach estimate

Figure 4.22. Sequence of operators entered during scenario 1 (first plan).

The generic execution cycle for a Soar agent comprises five phases:

- input, during which external input is written to the input-link
- proposal, during which operator proposal productions fire (or are retracted)
- decision, during which the agent chooses between available operators

- application, during which the application production for the chosen operator fires
- output, during which output is written to the output-link, prior to the new input phase

At agent initialization, the top-state is entered; input is received, then the only operator to be proposed is the *initialize-planner* operator, which is used to name the top-state (as *planner*) and to write various constants and values to working memory (this is knowledge that is embedded in the agent). The initialize-planner operator is chosen (being the only one) and then its application production fires, writing to working memory. The changes to working memory cause the operator proposal production to retract (e.g. its conditions are no longer met), and no output has been written, so the agent skips the output and input phases and gets again to the proposal phase.

Since no plan is present in working memory, the *generate-plan* operator is proposed and chosen; however, this has no corresponding application production, hence a no-change impasse (see Section 2.4) is detected, leading to the creation of the generate-plan substate. At this point, a working memory element for the flight plan is created, but this does not yet contain any Actions. This condition causes the *take-off* operator to be proposed and chosen, and the take-off substate is entered (notice that during a re-planning event, the old-plan operator would fire instead). The generate-plan operator has not retracted yet, and will not do so until a flight plan is added to the top-state.

The take-off substate works in a very straightforward manner: the *park*, *taxi*, t-*off-run* and *climb* operators fire and are retracted in this sequence, each adding an Action to the flight plan; once a climb Action has been added, the take-off operator retracts and causes the substate to be closed (but its results will have been written to a working memory location belonging to the generate-plan state).

Concurrently with the retraction of the take-off operator, the *main-mission* operator is proposed and chosen, leading to the creation of the main-mission substate; within this the main-mission-start Action is added to the flight plan, then the *plan-sequencing* operator is proposed and chosen, and a further substate entered. This is the first place where the decision process is more complex; first, the elaboration production which calculates the distance from the current position to all remaining Objectives fires twice (once per Objective). Then, the *target-recon* and *transit* operators are proposed; the decision process relies on preference rules, which in this case indicate that the closest Objective is to be preferred, and that Transit Objectives should always be chosen last. The target-recon operator is chosen, its application production fires and then its proposal will retract, so the transit operator (still valid) is chosen and applied. Plan-sequencing is closed, having written to the main-mission state the order in which Objectives should be executed (Objective 2 first and then Objective 1).

The next operator to be proposed and chosen is *actions-definition*; a substate is entered again, and the internal operation is very linear. Since a counter is used, operators corresponding to the Objectives will fire and retract in the order defined during plan-sequencing. Hence in this case *target-recon* is proposed and chosen first, its application rule writing new Actions in the flight plan; then the same happens for *transit*. With all Objectives translated into Actions, the state closes and then the main-mission state closes as well (after adding a main-mission-end Action to the flight plan). The *approach* operator is proposed and chosen, entering a new substate which flows linearly from the *descent* operator to *landing*, and then to *end-taxi* and *end-park*.

The generated flight plan at this stage is lacking "intelligent" features, but is complete; in the scenario, it is composed of 13 Actions: park, taxi, take-off, climb, main-mission-start, travel, recon, travel, main-mission-end, descent, landing, taxi, park. The *mission-path-adjust* operator fires and results in the creation of a substate, however due to the lack of dangerous Entities this results in no significant changes.

The *estimations* operator fires and causes the generation of a new substate, where estimates for the flight plan are generated linearly, Action by Action. The operators that fire are *take-off-estimate*, *travel-estimate*, *analyze-estimate*, *travel-estimate* and *approach-estimate*. The estimations substate is closed and *plan-finalization* entered; within it, the fuel-check operator fires but finds no issues and thus results in no significant changes to the flight plan.

Finally, after plan-finalization is closed, the generate-plan state writes the entire flight plan to the top-state and to the output-link; this causes the generate-plan to close, and the normal Soar cycle (including the input and output) phases to reprise.

4.4.2 Scenario 2 results

Scenario 2 is a simple scenario with two Search Objectives, one of which has a time priority (a time limit for its execution). The value of this time limit has a great influence on the flight plan, and the consequences of varying it will be thoroughly analyzed.

Figure 4.23 shows two plots of flight plans obtained for scenario variation 2a; the first plan is obtained setting a time priority of 3200 sec for Objective 1, while the second is obtained setting the time priority to 4000 sec. Both plans start and finish at Sheffield airport and involve the same search patterns being flown: a parallel track search pattern covering a rectangular area, and an expanding diamond spiral pattern covering a circular area (please note that for the motives explained at the beginning of this section, the plot is not in scale). However, in the first case Objective 1 is executed first, while in the second case Objective 2 is executed first. Without a time priority, Objective 2 would be executed first since it is closer to the starting airport; but when a time priority is inserted, the time to finish Objective 1 is taken into account. If the time estimates calculated in the plan-sequencing state show that the time priority will not be respected, Objective 1 is prioritized over Objective 2.

In fact, the effect of the time priority value is not limited to this; as stated in Section 4.2.4, the estimates calculated in the plan-sequencing state cannot be very precise, thus borderline situations where the time priority for an Objective is slightly sufficient to achieve it might then result in the precise estimates revealing that the time priority is not respected after all. Rather than overhauling the entire plan, the priority-check algorithm from the plan-finalization state (see Section 4.2.10) increases the flight speed to ensure that the time limit is respected. This scenario provides an excellent opportunity to show the complex relationship between the plan-sequencing state and the plan-finalization state.

Studying the scenario while varying the time priority value, a total of seven possible situations can be encountered:

- Priority > 4200 sec; in this case, Objective 2 is executed first and no speed increases are needed
- 3810 sec < Priority < 4200 sec; in this case, Objective 2 is executed first and one speed increase is needed (this is the second case in the figure; normal cruise speed is 40 m/s, normal search speed is 35 m/s, and these are increased by 10%)
- 3700 sec < Priority < 3900 sec; in this case, Objective 2 is executed first and two speed increases are needed

- 3500 sec < Priority < 3700 sec; in this case, Objective 1 is executed first and no speed increases are needed
- 3250 sec < Priority < 3500 sec; in this case, Objective 1 is executed first and one speed increase is needed
- 3070 sec < Priority < 3250 sec; in this case, Objective 1 is executed first and two speed increases are needed (this is the first case in the figure; notice that only Objective 1 speeds are increased by 21%)

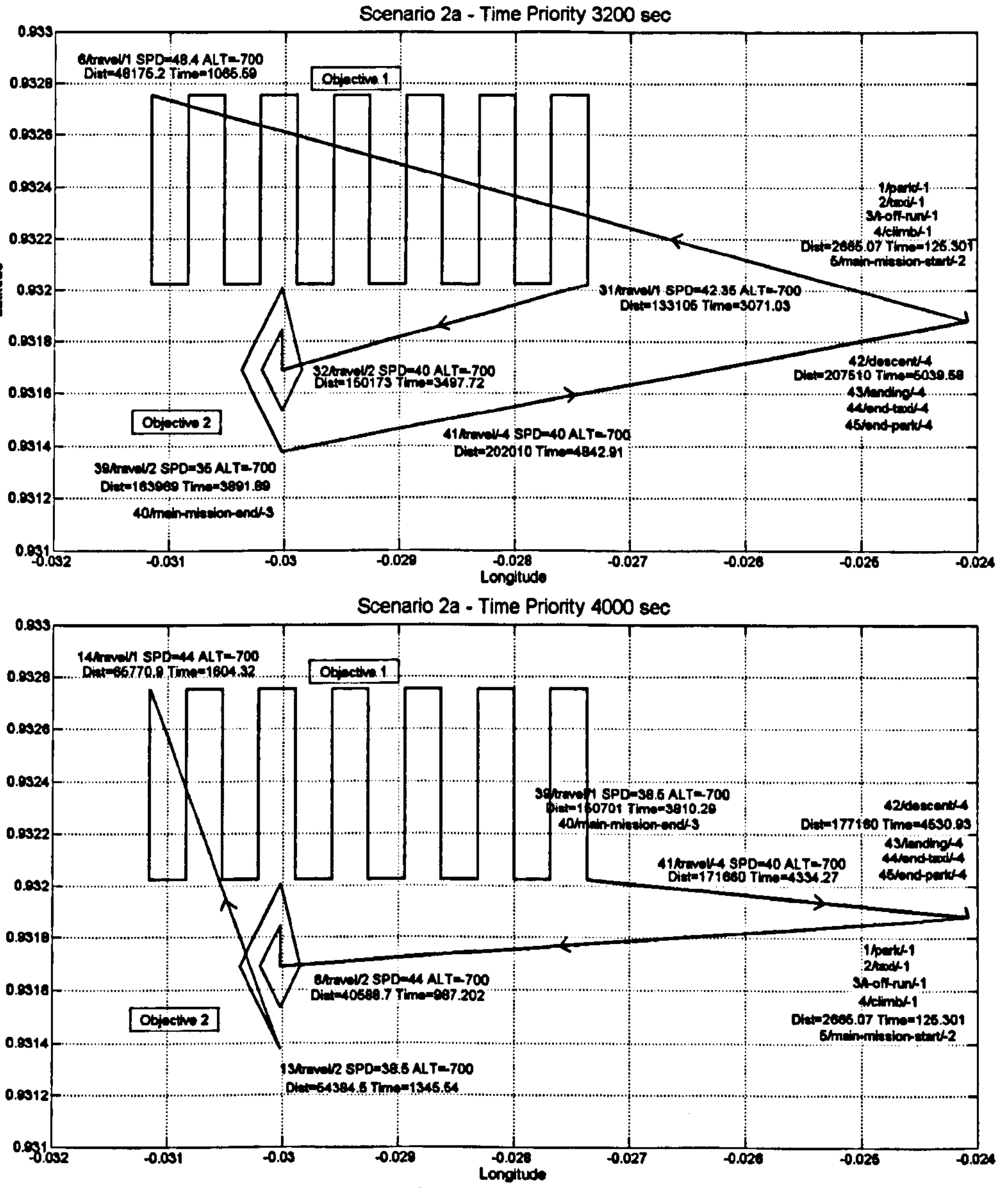
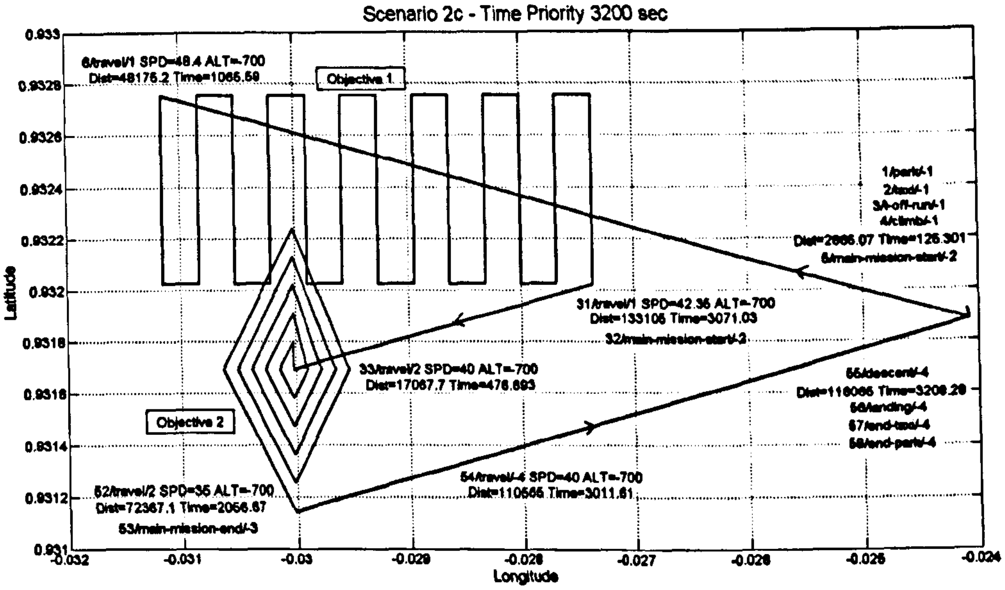


Figure 4.23. Plot of scenario 2a flight plans with different time priorities.

- Priority < 3070 sec; in this case, Objective 1 is executed first, two speed increases are used but this is still not sufficient to respect the time priority; the priority is simply impossible to achieve using the current flight parameters and the MMA will inform the UAV operator of this, but the plan will be executed in this form

These results were obtained varying the time priority for Objective 1 in scenario variation 2a; scenario variation 2c introduces changes to Objective 2while the mission is being flown: the search radius is increased and search accuracy is decreased (to obtain a tighter search). A re-planning event occurs while performing Action 10 of the original flight plan and the Exag may or not be committed to complete the current Objective. The resulting flight plan is shown in Figure 4.24.



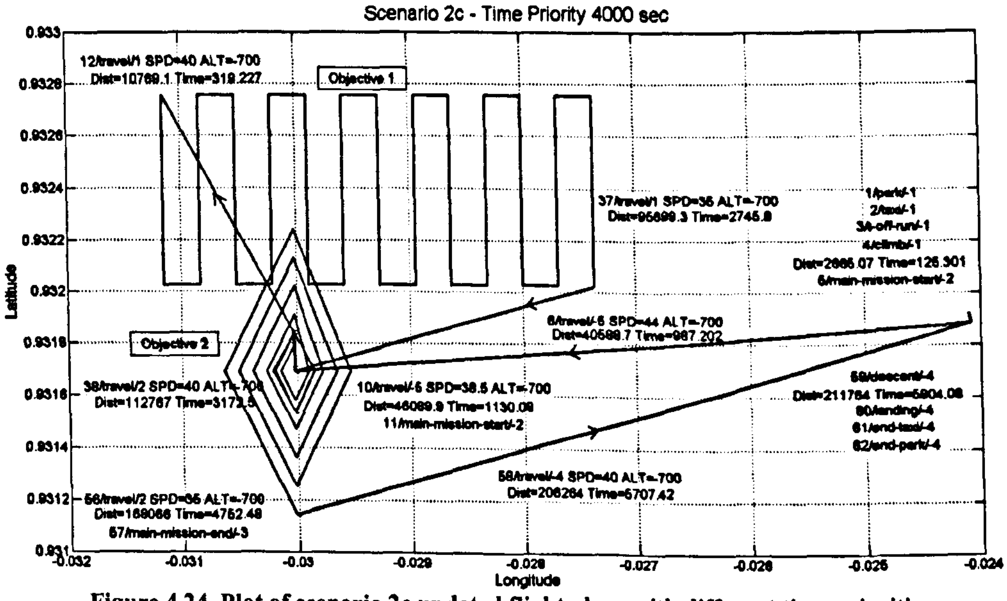


Figure 4.24. Plot of scenario 2c updated flight plans with different time priorities.

It is interesting to see how the re-planning event is also affected by the value of the time priority: the scenario evolves differently depending on the original flight plan. With time priority values lower than 3700 sec, the original flight plan has Objective 1 executed first. The re-planning event occurs while Objective 1 is being accomplished (Action 10 corresponds to the connecting leg between the second and third vertical legs of the search pattern); in fact, the change to Objective 2 does not modify conditions for Objective 1 and the Exag will be committed to complete Objective 1. In the Planner Agent, the old-plan state is entered and copies all Actions related to Objective 1 to the new flight plan; only Objective 2 will be considered in the main-missions state. In the new flight plan, only the part related to Objective 2 will change, thus the new main-mission-start Action is placed after Objective 1.

The situation is very different for time priority values higher than 3700 sec, with the original flight plan that expects Objective 2 executed before Objective 1 (since it is closer to the starting airport and the estimates show that the time priority will be respected). The re-planning event occurs while Objective 2 is being accomplished (Action 10 corresponds to the fourth leg of the search pattern). Two issues arise: first of all, the Actions already flown for Objective 2 are invalidated, since the new plan requires a better search accuracy; secondly, the new parameters for Objective 2 involve a much longer search time. The new flight plan sees the UAV dropping Objective 2 to execute Objective 1 first, with Objective 2 being executed from scratch at the end. The new flight plan contains indication of the Actions that were completed during the flight but have become unsuccessful because of the change in Objective 2 (these Actions are highlighted in red in the figure, and their Objective is set to -5 to indicate an unsuccessful Action). In this case, the old-plan state copies Actions 1 to 10 in the flight plan, marking Actions 6 to 10 as unsuccessful; the main-mission state places a new main-mission-start Action after Action 10 and will consider both Objectives, deciding to execute Objective 1 first. Notice that estimates for the new part of the plan are calculated starting from the position where the re-planning event occurs; normally, the actual total distance and time would be added to the estimates, but the data is not available in this simulation since the Planner is being tested separately from the SAMMS architecture. So, the 2745 sec estimate for the conclusion of Objective 1 is counted starting from the re-planning event; taking for valid the estimates of the original plan, the re-planning event occurs sometime before 1130 sec, and the sum of these times yields 3875 sec, meaning that the time priority would be still respected.

4.4.3 Scenario 3 results

Scenario 3 is a scenario of medium complexity, having six Objectives with different time priorities. There are five scenario variations; three of these differ only because of changes in the time priorities of the various Objectives, while the latter two involve the

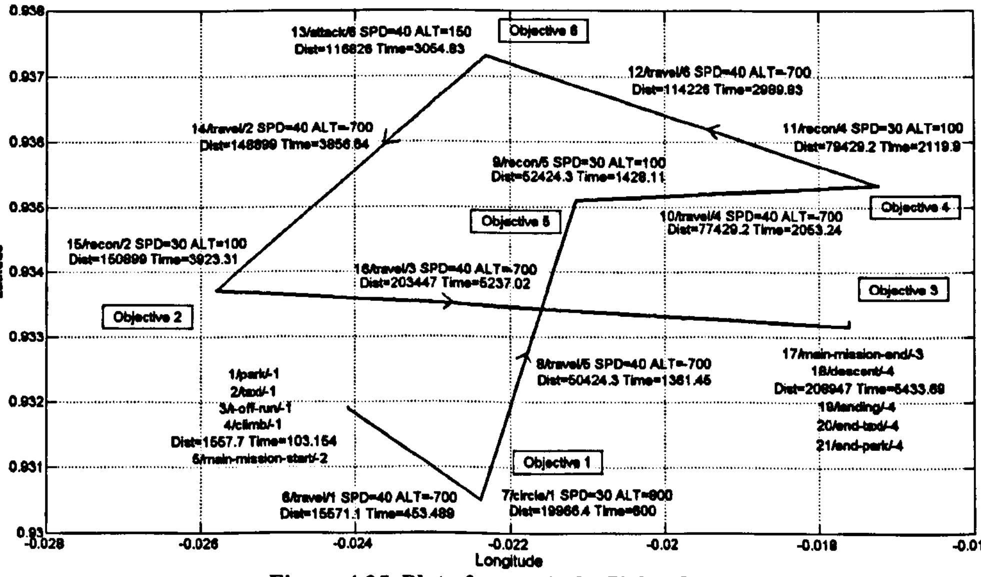


Figure 4.25. Plot of scenario 3a flight plan.

introduction of dangerous Entities and fuel limitations. In all variations, the scenario includes a Transit Objective, so the UAV takes-off and lands at different airports (Sheffield airport and Doncaster airport, respectively).

Figure 4.25 shows the plot of the flight plan generated for scenario variation 3a; in this variation, Objective 1 has an immediate priority and Objectives 4 and 5 have time priorities of 2500 sec and 1500 sec respectively. In the plan-sequencing state, the time priorities override the nearest neighbour algorithm (the Objective sequence would be: 1-2-5-6-4-3, but it becomes 1-5-4-6-2-3). The plan expects the UAV to take off and climb (to the main-mission-start altitude) in 103 sec, then travel to Objective 1 at 40 m/s and 700 m altitude above sea level, then loiter there at 30 m/s and 800 m altitude above ground until mission time 600 sec is reached. The plan is completed after covering 209 km in 5430 sec (about one hour and half). In the next paragraph, these values will be compared to those for the flight plan of scenario variation 3b.

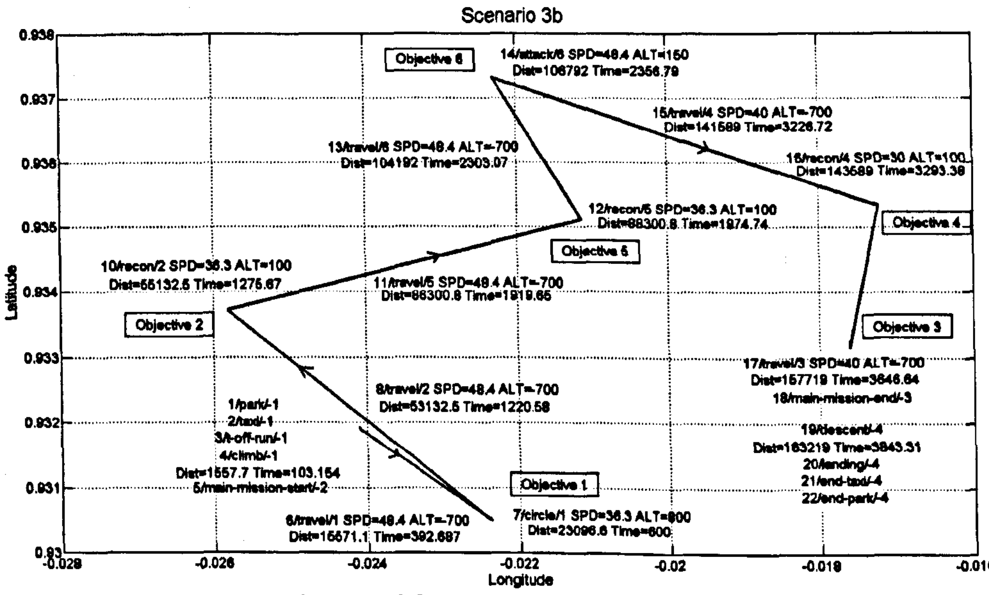


Figure 4.26. Plot of scenario 3b flight plan.

Figure 4.26 shows the flight plan generated for scenario variation 3b; in this variation, Objective 1 and 2 have an immediate priority and Objectives 5 and 6 have time priorities of 1500 sec and 2500 sec respectively. Notice that in theory it should not be possible to set an immediate priority for more than one Objective; the Planner can in fact handle the situation, however it will sort the Objectives using the nearest neighbour algorithm; ultimately, this hampers the meaning of setting an immediate priority (which is set to tell SAMMS that the Objective must be executed before all others). Incidentally, the Objectives are ordered in the sequence which would be chosen by the nearest neighbour algorithm on its own. It must be noted that it is not possible to respect the time priority for Objective 5, despite the attempt to increase flight speed (the increase-speed algorithm is used twice, as can be seen by the flight speeds being set to 21% more than regular values, e.g. cruise speed of 48.4 m/s instead of 40 m/s). The increase in speed however allows for the time priority of Objective 6 to be respected. The total distance estimate is 163 km, with a total time estimate of 3840 sec; compared to the estimates for scenario variation 3a, these are significantly better. It is evident that the time priorities in scenario variation 3a have a great influence on the flight plan, causing a significantly longer route to be chosen. In fact, this is a trade-off situation:

time priorities might be very important from the point of view of the UAV operator, however they are very likely to result in a non-optimal course to be chosen.

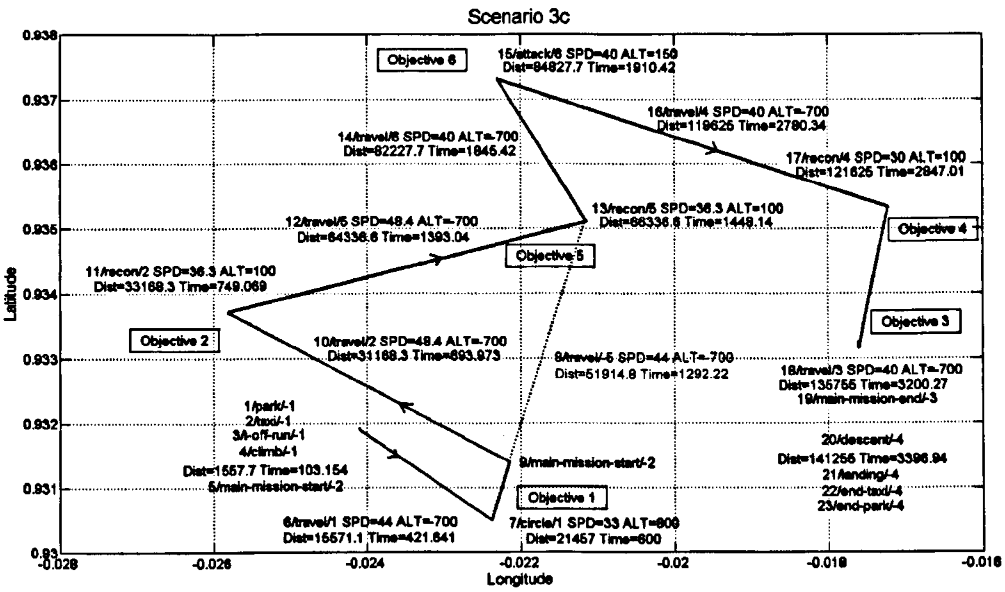


Figure 4.27. Plot of scenario 3c flight plan after re-planning.

Figure 4.27 shows the flight plan generated after the re-planning event in scenario variation 3c; in this variation, the time priorities are switched from the values they have in variation 3a (Objective 1 immediate, 2500 sec for Objective 4 and 1500 sec for Objective 5) to the values they have in variation 3b (Objectives 1 and 2 immediate, 1500 sec Objective 5 and 2500 sec for Objective 6). The change happens while

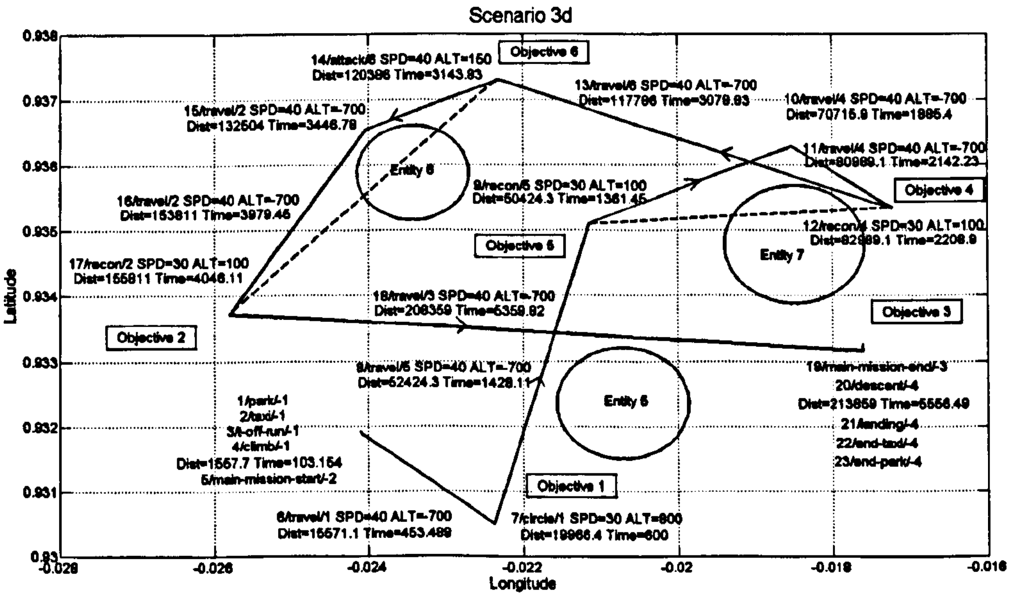


Figure 4.28. Plot of scenario 3d flight plan, with dangerous Entities in red.

performing Action 8 of the original flight plan, meaning that Objective 1 has been completed and the UAV is cruising towards Objective 5; the Exag is not committed to complete the current Objective, so the new flight plan diverts immediately towards the Objective that is now chosen as first (Objective 2). This is represented in the figure by the trajectory change that connects the second main-mission-start Action with Objective

2; however, in the figure it is possible to see that Action 8, not completed, is still copied to the new flight plan as an unsuccessful Action (it is the greyed Travel Action marked as having Objective -5, that indicates an unsuccessful Action, and represented by a dashed line). In truth, the change would happen while the UAV is cruising between Objectives 1 and 5, and the actual UAV trajectory would immediately divert towards Objective 2. Notice that the immediate priority for Objective 2 is fully valid in this case, since Objective 1 has already been completed.

Figure 4.28 shows the flight plan for scenario variation 3d; this variation has the same Objectives as variation 3a, but introduces three dangerous Entities that trigger the mission-path-adjust algorithm. The Entities are represented as red circles in the figure; it is possible to see that two Travel Actions intersect them. The Travel Action between Objective 5 and Objective 4 (shown as a dashed line in the figure) intersects with the threat range of Entity 7; thus, a new waypoint is added, and the flight plan modified to include a new Travel Action for Objective 4. The Travel Action between Objective 6 and Objective 2 intersects with the threat range of Entity 6; again, a new waypoint is added to avoid the area. Entity 5 does not interfere with the flight plan and causes no modifications. Notice that the waypoint causes an increase in distance and time estimates; the time priority for Objective 4 set at 2500 sec is still respected, but it would not be so if it were lower than 2200 sec. The increase-speed algorithm would then intervene, increasing the flight speed for Actions before Objective 4. Interestingly, in variation 3d a time priority of 2200 sec for Objective 4 would need a speed increase because of the waypoint added by mission-path-adjust, but this is not true for variation 3a where no waypoint is added (the speed increase compensates the new waypoint).

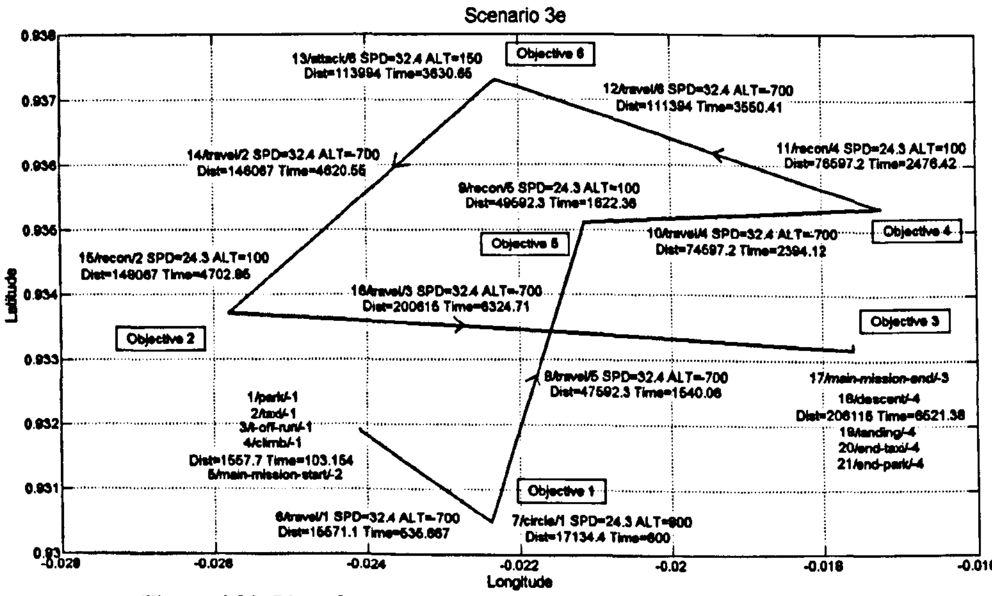


Figure 4.29. Plot of scenario 3e flight plan, with reduced flight speeds.

Figure 4.29 shows the flight plan for scenario variation 3e; this is exactly the same as variation 3a, except for the flight speed of all Actions. In this scenario variation, the amount of fuel available on board has been reduced so as to trigger the activation of the fuel-check algorithm (see Section 4.2.10). The fuel consumption estimate for scenario variation 3a is 28.35 kg. The current fuel model for the UAV expects a maximum fuel amount of 60 kg. For scenario variation 3e, the starting amount of fuel is set to 24 kg. This causes the fuel-check algorithm to reduce the flight speed for the entire plan; a first iteration reduces speeds by 10% and is not sufficient, so a second iteration of reduce-speed is triggered and speed are eventually reduced by 19% in total, across the entire plan (for example, the cruise speed for Travel Actions is reduced from 40 m/s to 32.4 m/s). This reduces the total fuel consumption estimate to 21.77 kg, so the flight plan is finalized with these speeds. Notice that the time priority for Objective 5 is no longer respected (the Objective is accomplished at 1622 sec, compared to a time limit of 1500 sec); however, the fuel-check algorithm is prioritized over this, since the UAV obviously needs to be certain to complete the mission with available fuel.

4.4.4 Scenario 4 results

Scenario 4 is a complex scenario involving up to ten Objectives (the maximum possible amount at the current development stage for SAMMS). The main goal for the scenario is to demonstrate SAMMS's ability to develop good flight plans when the number of Objectives is high. Some of the Objectives also present time priorities, rendering the development of the flight plan an even more complex process. The scenario comes in three variations, with only the third being a dynamic one (e.g. including a re-planning event). The scenario does not include a Transit Objective, so the UAV will always take-off and land at the same airport (Sheffield airport). The dangerous Entities from scenario 3 are present in all variations of scenario 4 (even though renamed as Entities 7, 8 and 9), so the mission-path-adjust algorithm is tested thoroughly. Please note that due to the scenario complexity, some of the usual information regarding the flight plan had to be cut from the plots, in order to keep the plots visually comprehensible.

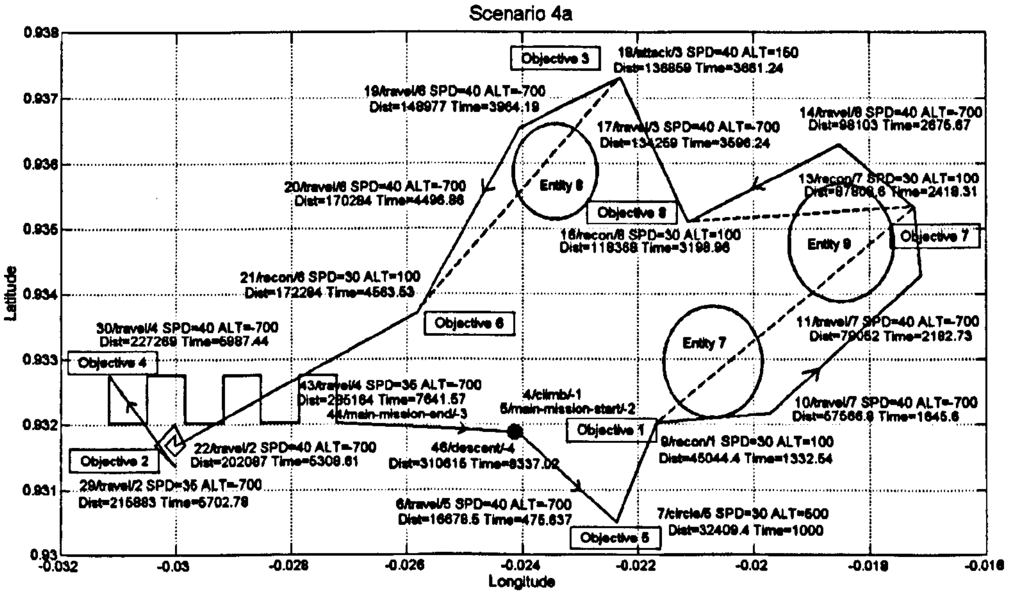


Figure 4.30. Plot of scenario 4a flight plan.

Figure 4.30 shows the plot of the flight plan generated for scenario variation 4a. In this variation, eight Objectives are present and there are no time priorities. Objectives are executed in this order: 5-1-7-8-3-6-2-4, which is in fact excellent from the point of view of the total distance covered. Three Travel Actions cross the danger areas represented by Entities 7, 8 and 9; the mission-path-adjust algorithm adds appropriate waypoints to avoid the danger areas (in the figure, dashed lines represent the unmodified flight plan). The Travel Action between Objectives 1 and 7 intersects two

Entities, therefore two waypoints are added and the Travel Action is split into three Travel Actions; interestingly, due to the way the algorithm works, two slightly different versions of the flight plan are possible, depending on which waypoint is added first (in

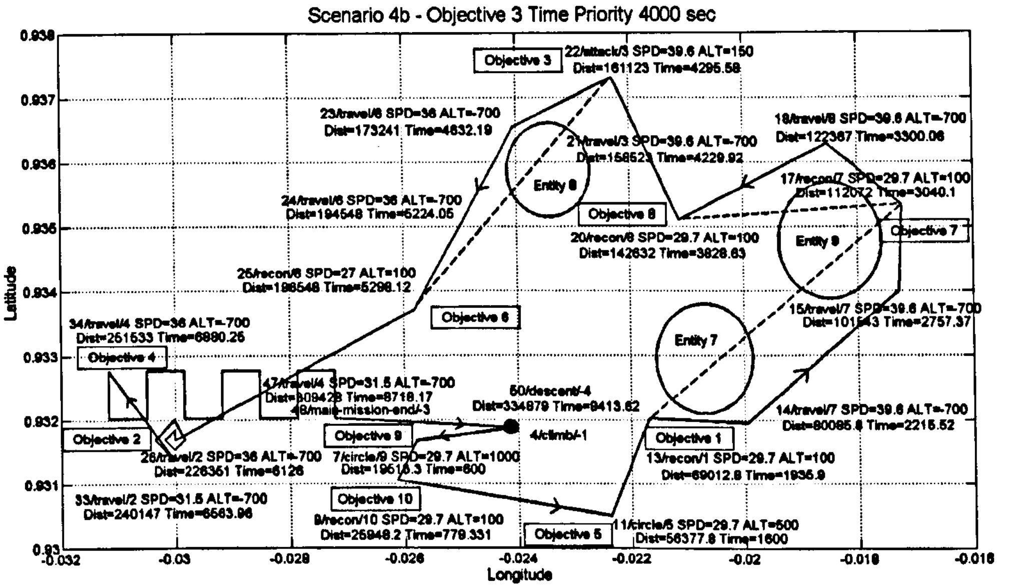


Figure 4.31. Plot of scenario 4b flight plan.

the mission-path-adjust state, the detect-crossing operator fires twice for the Travel Action, and the decision between those is random since it is ultimately indifferent from the point of view of flight plan validity). The plan is completed after covering 310 km in 8337 sec; fuel consumption is estimated at 38.25 kg, and if the initial amount of fuel is higher no modifications are needed to the flight speeds.

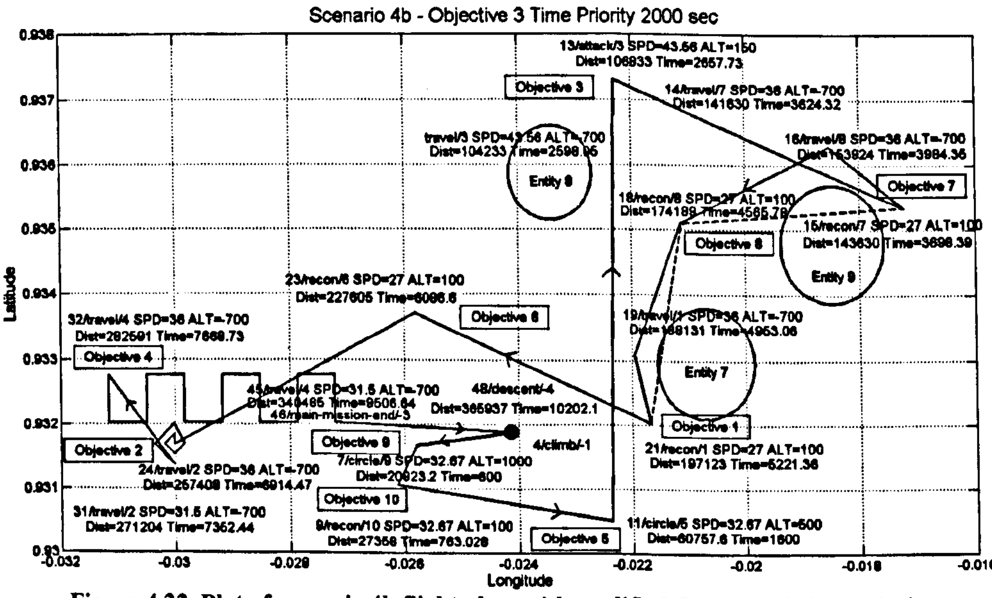


Figure 4.32. Plot of scenario 4b flight plan, with modified Objective 3 time priority.

Figure 4.31 shows the plot of the flight plan generated for scenario variation 4b. In this variation, all ten Objectives are present from the start and some details regarding certain Objectives are modified. In particular, Objective 5 is modified to have a time priority of 1400 sec and an orbit time limit of 1600 sec (as opposed to no time priority and 1000 sec time limit in variation 4a). A time priority of 4000 sec is also added for Objective 3. Objectives are executed in this order: 9-10-5-1-7-8-3-6-2-4. In fact, the flight plan is apparently very similar to that for variation 4a, only adding Objectives 9 and 10 at the beginning. However, more subtle differences are present. In order to reach Objective 3 within the time priority value of 4000 sec, flight speeds have to be increased (in the plan-finalization state by the priority-check algorithm). With this speed increase, the total fuel consumption is estimated at 69.64 kg; since the maximum fuel amount is 60 kg, the fuel-check algorithm intervenes at this stage and reduces flight speeds across the entire plan. The new fuel consumption estimate is 43.53 kg, so the speed reduction is effective in ensuring the completion of the mission, however the time priority for Objective 3 is not respected anymore. The flight plan is completed after covering 334 km in 9413 sec.

Figure 4.32 also shows a flight plan developed for scenario variation 4b, however the value of the time priority for Objective 3 is set to 2000 sec rather than 4000 sec. This causes a significant change in the way Objectives are ordered: the new order is 9-10-5-3-7-8-1-6-2-4. The waypoints added to avoid the dangerous Entities are completely different; only two waypoints are needed, between Objectives 7 and 8 and between Objective 8 and 1. The time priority for Objective 3 is not respected even if the flight speed is increased; this is because Objective 3 is too far from Objective 5, where the UAV is ordered to stay until time 1600 sec. The total fuel consumption estimate would be 62.41 kg without intervention from the fuel-check algorithm; it can be seen in the plan that flight speeds are increased for the first part of the plan (to reach Objective 3 quickly) and reduced for the rest of the plan (to save fuel). This allows for a final fuel consumption estimate of 54.66 kg; note that the different order of execution for Objectives results in a significantly longer distance and time: 366 km covered in 10200 sec.

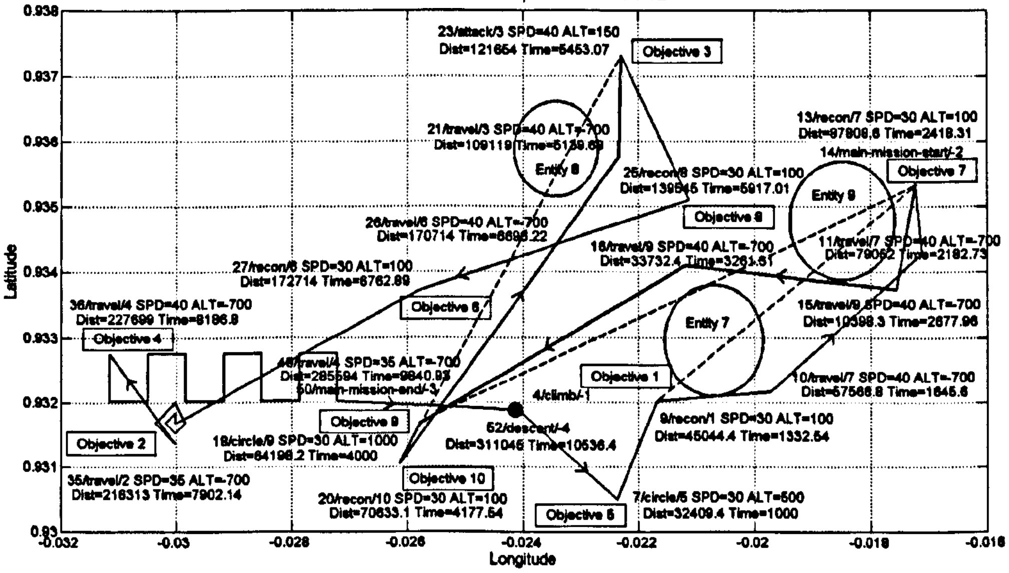


Figure 4.33. Plot of scenario 4c flight plan, re-planning event at Action 12.

Figure 4.33 shows the second flight plan generated for scenario variation 4c; in this variation, the mission starts with the Objectives as in variation 4a, hence the first flight plan will be that from Figure 4.28. The scenario then evolves adding Objectives 9 and

10 and changing some time priorities. The updated flight plan is obviously different depending on the time at which the re-planning event occurs. In the figure, it is possible to see what happens when the re-planning event occurs while performing Action 12 in

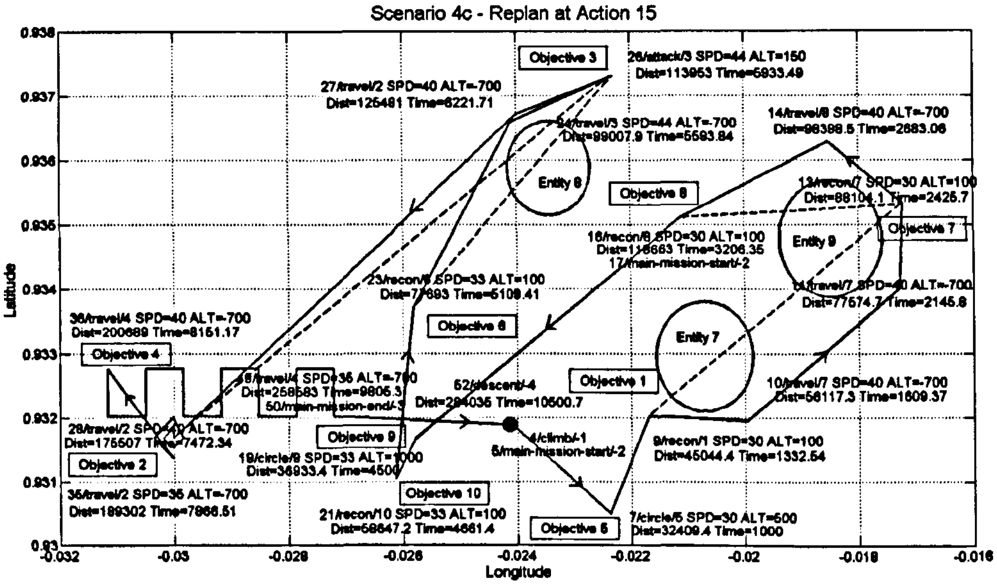


Figure 4.34. Plot of scenario 4c flight plan, re-planning event at Action 15.

the original flight plan (e.g. while travelling towards Objective 7). Objective 9 is set with a time limit of 4000 sec, Objective 10 has a time priority of 4500 sec and Objective 3 has a time priority of 6000 sec. Objectives 5 and 1 have already been completed (in this order) and the Exag is committed to complete Objective 7; the resulting final order

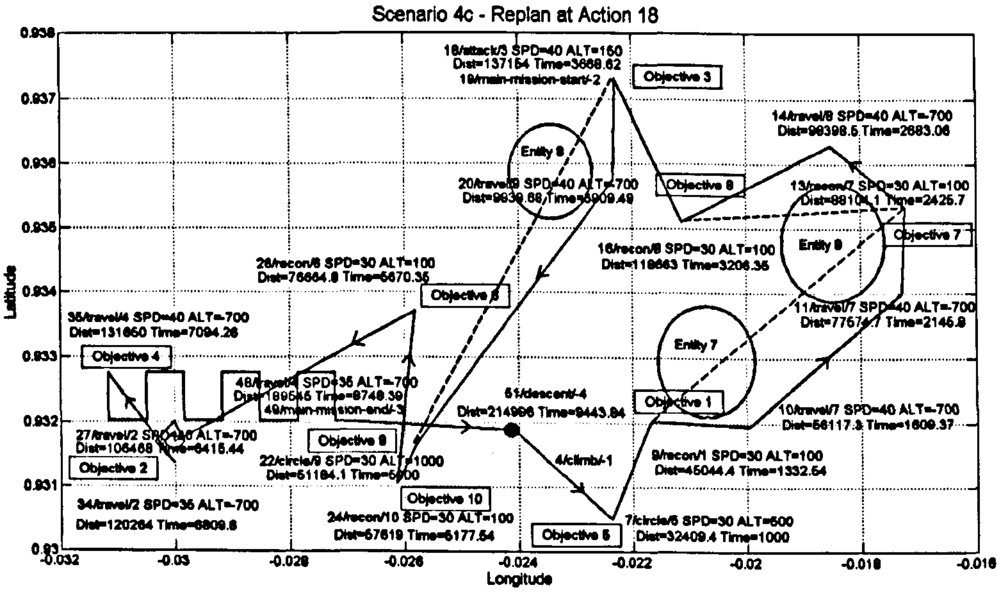


Figure 4.35. Plot of scenario 4c flight plan, re-planning event at Action 18.

of execution for Objectives is 5-1-7-9-10-3-8-6-2-4. The overall distance and time estimates are 399 km (obtained as 88 km up to Objective 7 and 311 km from Objective 7 to the end) and 10536 sec.

The flight plan shown in Figure 4.34 is also obtained from scenario variation 4c; in this case, the re-planning event is set to occur while Action 15 is being performed (e.g.

while travelling towards Objective 8). To keep the scenario realistic, Objective 9 is set with a time limit of 4500 sec, Objective 10 has a time priority of 5000 sec and Objective 3 still has a time priority of 6000 sec. The Exag is committed to complete Objective 8 so Objectives 5, 1, 7 and 8 can be considered completed; the resulting final order of execution for Objectives is 5-1-7-8-9-10-6-3-2-4. Note that to respect the time priority of Objective 3, flight speeds are increased by 10% while covering Objectives 9, 10, 6 and 3. Total distance and time estimates are of 402 km (118 + 284 km) and 10500 sec.

Finally, Figure 4.35 shows another case related to scenario variation 4c; in this case, the re-planning event is set to occur while Action 18 is being performed (e.g. while performing Objective 3). To keep the scenario realistic, Objective 9 is set with a time limit of 5000 sec and Objective 10 has a time priority of 5500 sec; the Exag is committed to complete Objective 3 so its changed time priority is not relevant. Objectives 5, 1, 7, 8 and 3 are already completed (in this order); the resulting final order of execution for Objectives is 5-1-7-8-3-9-10-6-2-4. No flight speed increases are needed in this case. The total distance and time estimates are of 351 km (137 + 214 km) and 9443 sec; compared to the cases in Figures 4.31 and 4.32, the flight plan does not change much because of the re-planning event, thus better routing is achieved.

4.4.5 Scenario 5 results

Scenario 5 is not particularly complex (compared to scenarios 4 and 6), but it is intended to test the ability of the Planner Agent to deal with multiple re-planning events. SAMMS is developed to be able to deal with any number of re-planning events, as long as these are not happening more rapidly than the time needed to generate a new plan, however this has not been tested in the previous scenarios. Additionally, the scenario is set in a different location (in the United States rather in the Sheffield area), and it involves particular changes to the flight plan: during flight, an Objective is cancelled

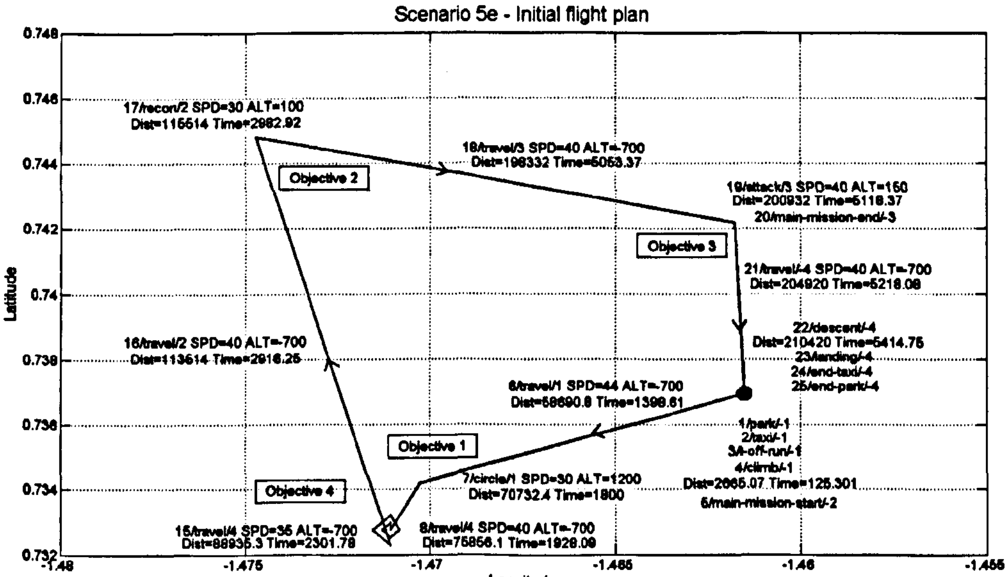


Figure 4.36. Plot of scenario 5e initial flight plan.

and the destination airport is changed. The scenario comes in five variations; the first four represent the various stages through which the scenario evolves, while the fifth (variation 5e) is the most significant variation, in which the scenario dynamically evolves from the original flight plan to the final situation. Only results from scenario variation 5e will be shown, as the other variations just reproduce the evolution steps of variation 5e, without the dynamic component.

Figure 4.36 shows the first flight plan generated for scenario variation 5e. Four Objectives are present and the UAV takes-off and lands at the same airport (Ann Arbor airport); Objective 1 has a time priority of 1500 sec and an Orbit time limit of 1800 sec, Objective 4 has a time priority of 2500 sec. In the flight plan, the order of execution for Objectives is 1-4-2-3. To reach Objective 1 in time, flight speed is increased by 10% (from 40 m/s to 44 m/s) for Action 6, which is the Travel Action taking the UAV from the starting airport to Objective 1. No speed increase is needed to complete Objective 4 in time. The total distance and time estimates are of 210 km covered in 5414 sec.

The first re-planning event occurs while the UAV is performing Action 6 (while it is travelling towards Objective 1). The UAV operator inserts a new Objective (Objective 5), which has an immediate priority; the Exag is not committed to complete Objective 1, so the new flight plan sees the UAV immediately diverting towards Objective 5. The new flight plan is shown in Figure 4.37; it includes the original Action 6 as an unsuccessful Action, that can be seen in the plot as a dashed line. Action 7 in the new flight plan is a main-mission-start Action; the new order of execution for Objective is 5-1-4-2-3. Because of the modifications to the flight plan, a new speed increase is needed to reach Objective 1 in time; however, the speed increase (21% more than normal speeds) for Actions 8, 9 and 10 is not sufficient, as the time estimate is 1533 sec with a time priority of 1500 sec. It is possible to see that the time estimates for the rest of the plan are unchanged, since the UAV is in any case loitering around Objective 1 until time 1800 sec, and the changes to the flight plan only affect earlier parts.

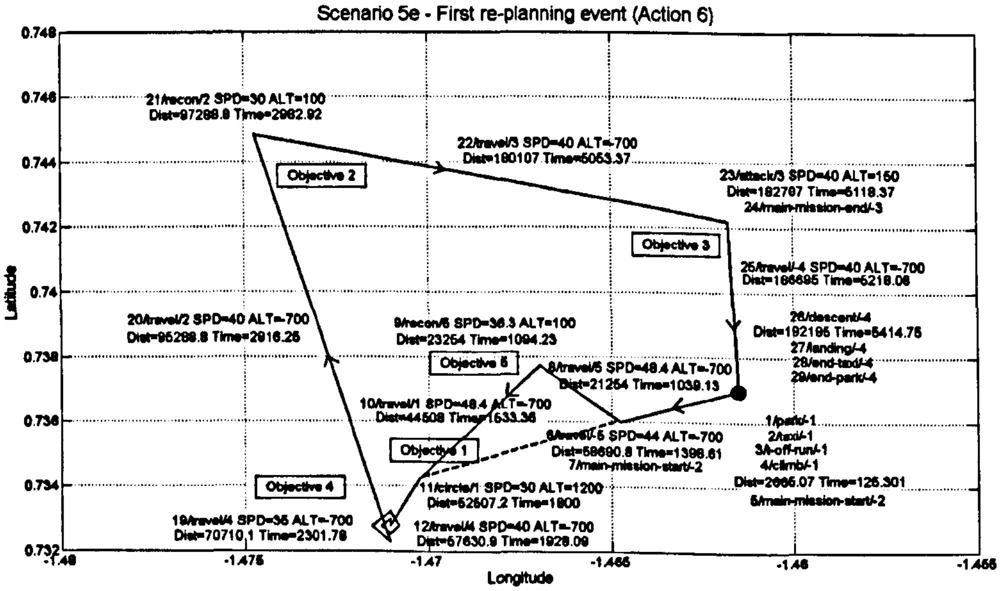


Figure 4.37. Plot of scenario 5e flight plan after first re-planning event.

The second re-planning event occurs while the UAV is performing Action 15 of the updated flight plan (while flying the third leg of the expanding diamond search pattern for Objective 4). The change involves only Objective 3, which is cancelled; the Exag is committed to complete Objective 4. The flight plan is changed accordingly, as shown in Figure 4.38; in the figure, the dashed line represents the flight plan as it was before the change. Since the Exag is committed to complete Objective 4, the new main-mission-start Action is placed after its completion, e.g. as Action 20. The time estimates for Objective 2 are unchanged, while the total time estimate becomes 5295 sec.

The third re-planning event occurs while the UAV performing Action 21 of the updated flight plan (while travelling from Objective 4 to Objective 2). A new Transit

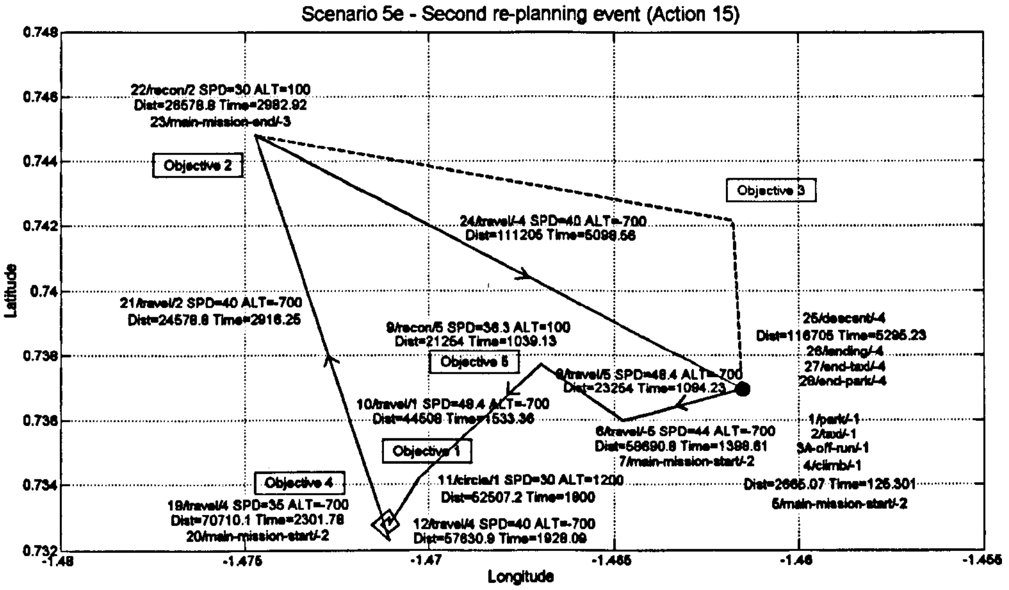


Figure 4.38. Plot of scenario 5e flight plan after second re-planning event.

Objective is added, thus the destination airport is no longer Ann Arbor where the UAV took off, but is instead Linden Price airport as required by the Transit Objective. The new flight plan is shown in Figure 4.39, where the previous flight plan is indicated as a dashed line. The Exag is committed to complete Objective 2, so no further change to the

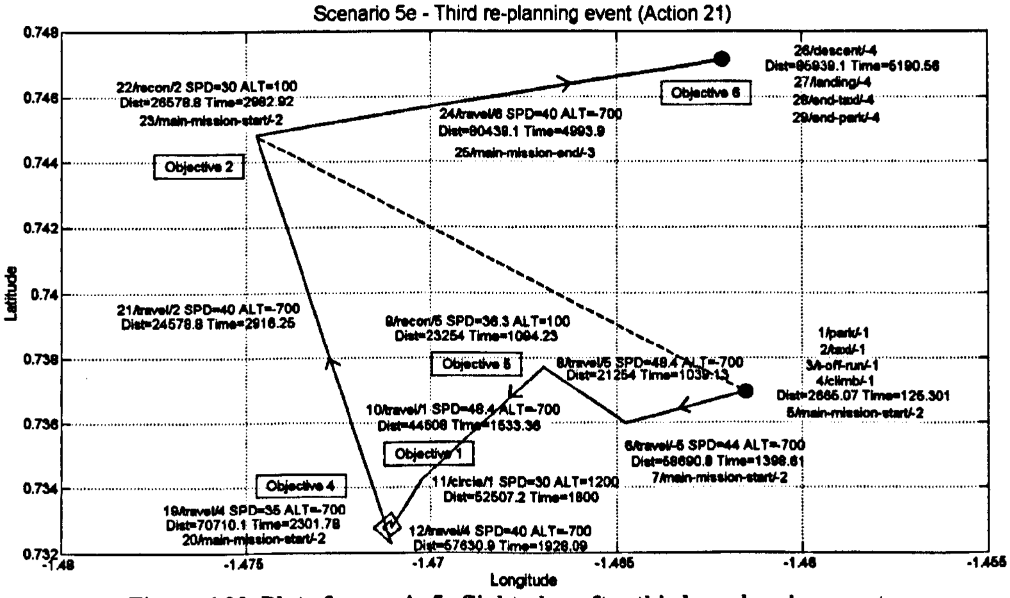


Figure 4.39. Plot of scenario 5e flight plan after third re-planning event.

plan is applied; in fact, when the new plan is generated, Objectives 5, 1, 4 and 2 are considered as completed, and with Objective 3 cancelled only the new Objective 6 is planned for. Again, the time estimates for Objective 2 are unchanged, and the new total time estimate is 5190 sec.

4.4.6 Scenario 6 results

Scenario 6 is a complex scenario involving up to 9 Objectives. Unlike other scenarios, which are designed around the capabilities of the Pioneer UAV, it is meant to be a scenario for a high-altitude long-endurance (HALE) type of UAV. The mission covers very large distances, which would normally be flown only with a HALE type of

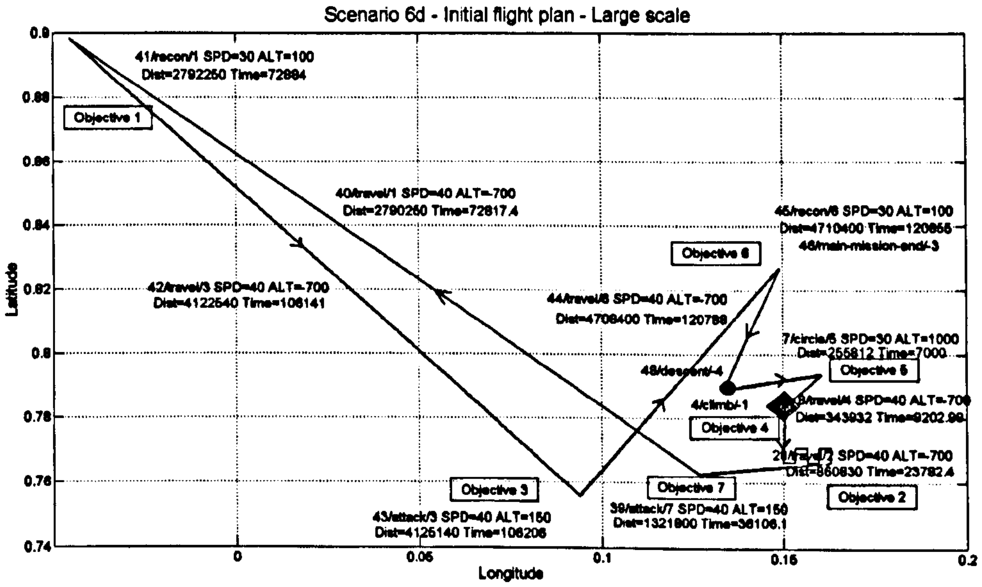


Figure 4.40. Large scale plot of scenario 6d initial flight plan.

UAV, probably equipped with turbojet engines for higher speed. The current implementation of SAMMS is targeted at the Pioneer UAV, and this results in the fact that the plan is extremely long to accomplish: the total time estimate for the most

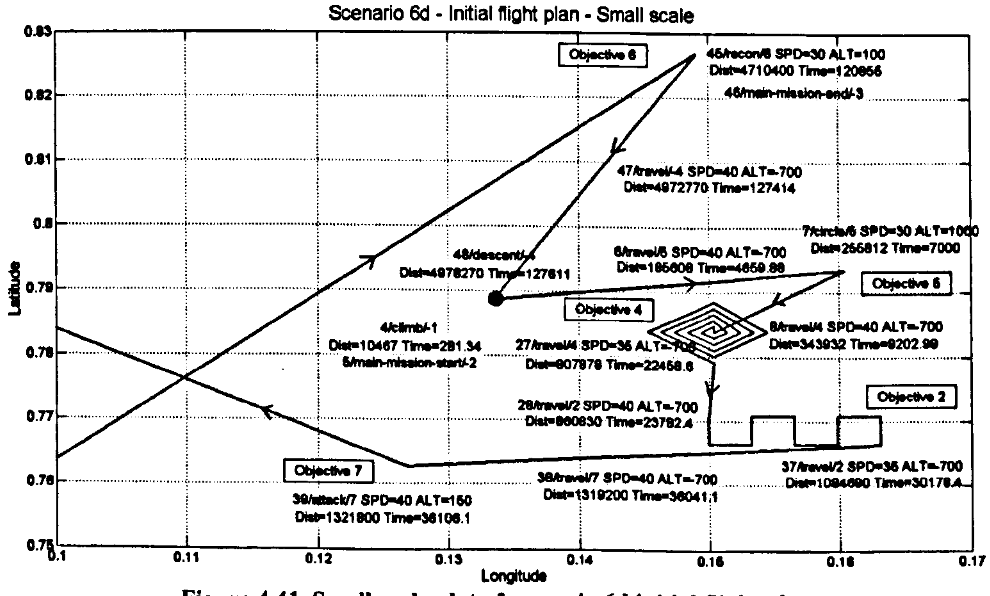


Figure 4.41. Small scale plot of scenario 6d initial flight plan.

important variation of the plan is 151000 sec, or 42 hours. Despite this, it was preferred to keep the Pioneer UAV characteristics for the calculation of estimates, which will then be much higher than those of other scenarios. Because of the very long times involved, the scenario will not be used during testing of the other SAMMS components (and of SAMMS as a system).

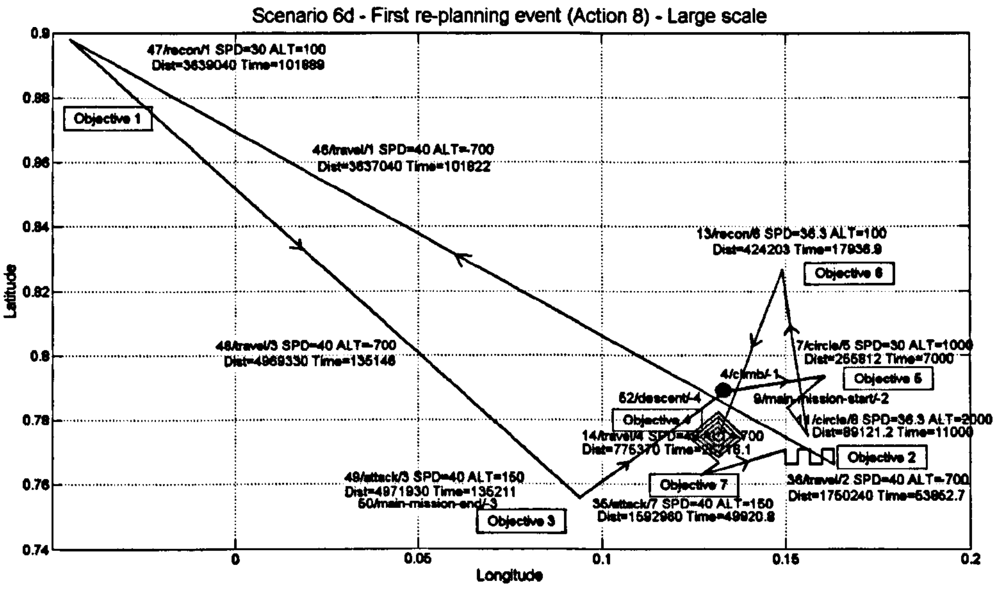


Figure 4.42. Large scale plot of scenario 6d after first re-planning event.

As stated in Section 4.3.6, four scenario variations are identified; only scenario variation 6d will be presented here, since the other variations just represent the different steps through which variation 6d goes through. The variation is dynamic, in the sense that it includes two re-planning events. Because of the scale of the area covered by the scenario, each plot will be presented in two versions with different scales.

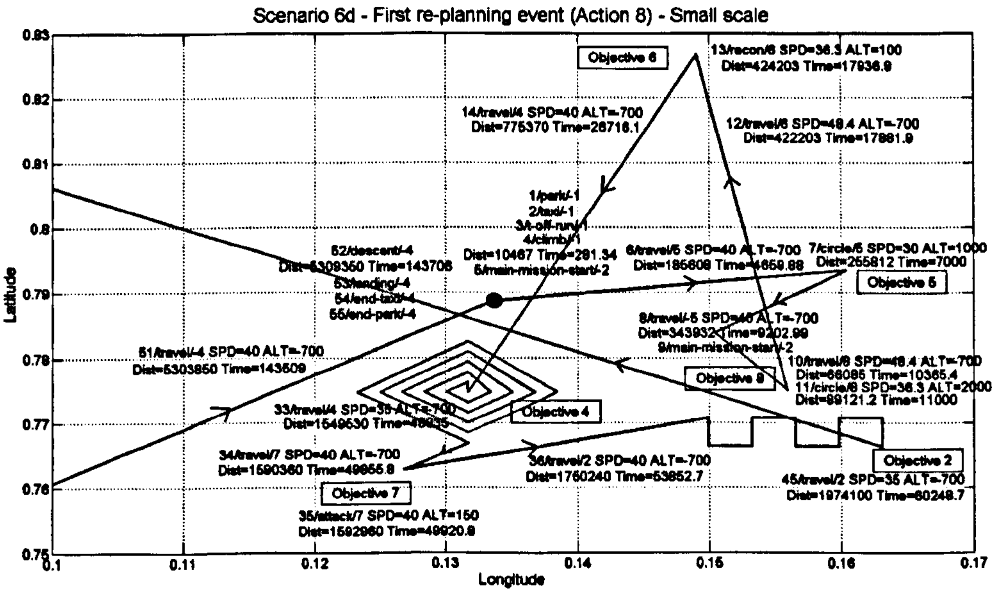


Figure 4.43. Small scale plot of scenario 6d after first re-planning event.

Figures 4.40 and 4.41 show the flight plan generated at the beginning of the scenario; at the start of the mission, seven Objectives are given, and only Objective 5 has a time priority (of 5000 sec). The order of execution for Objectives is 5-4-2-7-1-3-6; no speed increases are needed. The total distance covered is estimated at 4978 km, with a time estimate of 127600 sec (or 35.5 hours).

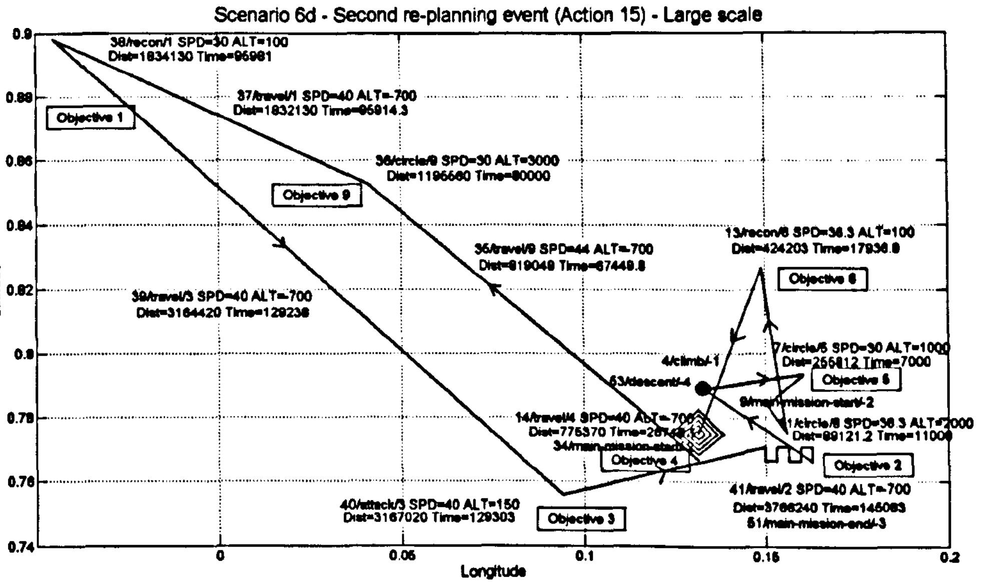


Figure 4.44. Large scale plot of scenario 6d after second re-planning event.

The first re-planning event occurs at time 9000 sec, while the UAV is performing Action 8 (it is travelling towards Objective 4). There are three changes: a new Objective is inserted (Objective 8, with an immediate priority); Objective 4 is heavily modified (changing the position of the search area centre, the search radius and the search accuracy); a time priority of 18000 sec is added for Objective 6. Figures 4.42 and 4.43

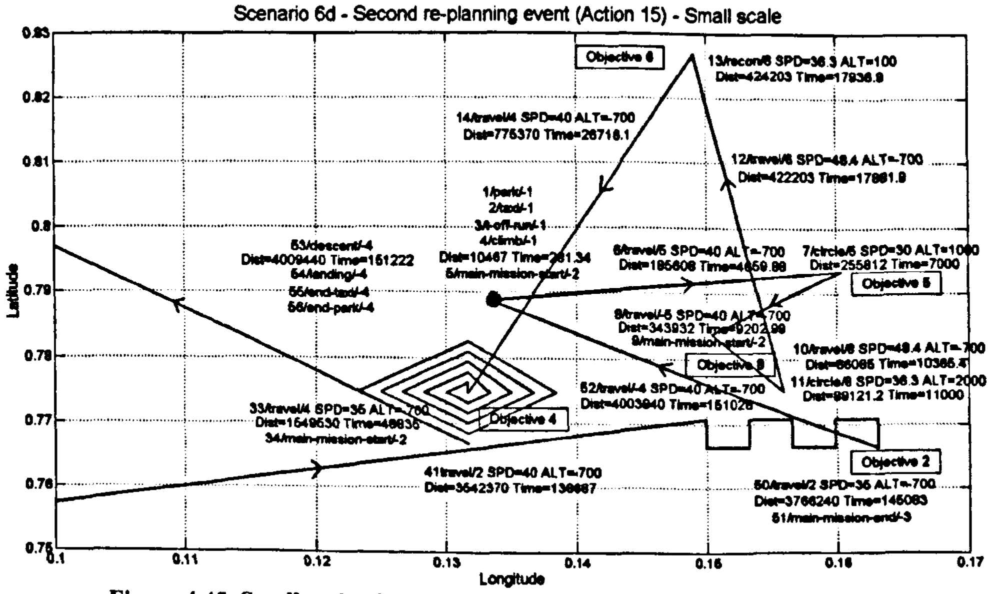


Figure 4.45. Small scale plot of scenario 6d after second re-planning event.

show the updated flight plan; Objective 5 has already been accomplished and the Actions already accomplished as part of Objective 4 are aborted, since the Objective is changed. The new order of execution for Objectives is 5-8-6-4-7-2-1-3. Notably, Objective 8 involves loitering until time 11000 sec, and to respect the time priority for Objective 6 a speed increase is needed (by 21%, hence two iterations of the speed-increase algorithm). The total distance and time estimates are greatly increased, because the updated Objective 4 requires a much longer distance to be covered and the time priority of Objective 6 causes the optimal path to be forfeited.

The second re-planning event occurs while the UAV is performing Action 15 of the updated flight plan, which is the first leg of the search pattern for Objective 4. There are two changes: Objective 7 is removed, and Objective 9 is added (with a time priority of 68000 sec). Figures 4.44 and 4.45 show the new flight plan that is generated; Objectives 5, 8 and 6 have already been accomplished and the Exag is committed to complete Objective 4, so the new part of the plan starts after the completion of Objective 4. Because of its time priority, Objective 9 is placed immediately after Objective 4; a moderate (10%) speed increase is needed in order to respect the time priority (a single iteration of the increase-speed algorithm is sufficient). The final order of execution for Objectives is 5-8-6-4-9-1-3-2. Because Objective 9 involves loitering until time 80000 sec, the final time estimate is increased even further, reaching more than 151000 sec (or 42 hours).

4.4.7 Scenario 7 results

As stated in Section 4.3.7, scenario 7 is an externally provided scenario that is not aimed at testing any particular SAMMS algorithm. The scenario involves 9 Objectives for which the order of execution is part of the scenario definition. The scenario involves four Target-Attack Objectives (weapon deliveries in the original scenario definition) and five Orbit Objectives (loiter times before each weapon delivery and before landing in the original scenario definition). The commanded altitudes are also different: cruise and loitering are set to occur at 1500 m above sea level, while attacks are performed at 800 m above ground. There are no variations for the scenario.

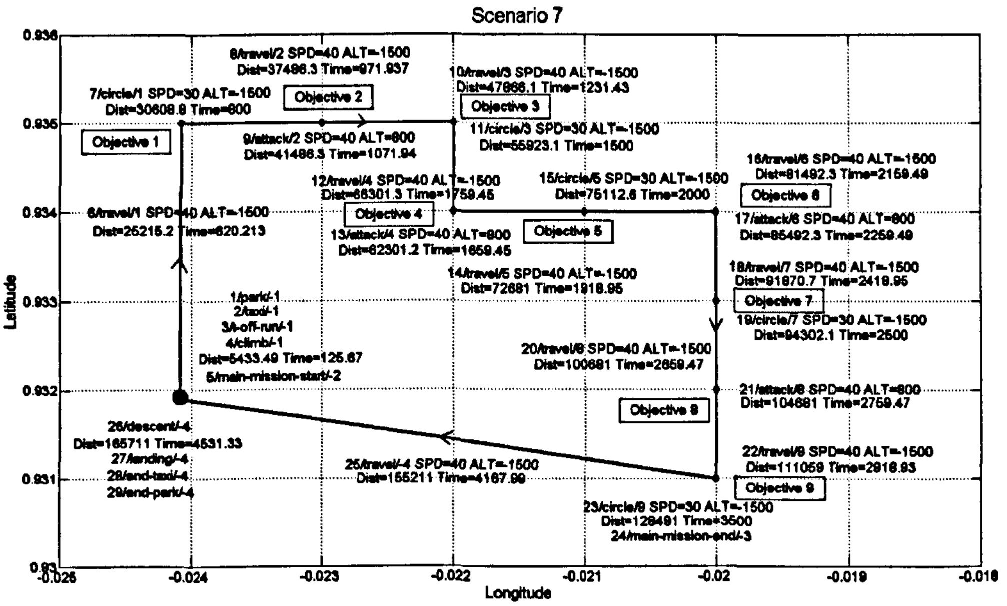


Figure 4.46. Plot of flight plan for scenario 7.

Figure 4.46 shows the flight plan generated by the Planner for the scenario; all Actions in the flight plan and their relative estimates are reported in the figure. Since there are no Search Objectives and no dangerous Entities to avoid, the plan consists of an alternation of Travel Actions immediately followed by either Attack or Circle Actions. The UAV is expected to land at the same airport where it took-off, so the last Action before Descent is a Travel Action which is added by the *return* operator in the *approach* state (see Section 4.2.6) and is marked as having Objective -4.

4.5 Concluding remarks

In this chapter, the Planner Agent was thoroughly described and tested. The Planner Agent is arguably the most important component of the Soar-based Autonomous Mission Management System, so great attention was spent in detailing its every aspect and in demonstrating its functionality.

The first part of the chapter focused on the description of the Planner Agent from a theoretical and technical point of view. Having defined the functions it is expected to perform in previous chapters (and particularly Chapter 3), implementation details were outlined here. The Input/Output interface was described first, followed by a thorough description of the algorithms embedded in the agent and of how they are implemented in terms of Soar coding.

The second part of the chapter was dedicated to the practical simulation tests that were executed on the agent in order to validate it. The Planner Agent can work separated from the other SAMMS components, and the tests presented in this chapter are accomplished in such a situation. A set of seven test scenarios was introduced; these scenarios will also be used during testing of the other SAMMS components, so a separate section is dedicated to them. Finally, results obtained during the testing campaign were displayed using graphical plots.

The next chapter will focus on another component of the SAMMS architecture: the Mission Manager Agent. While this agent is not strictly necessary to the operation of SAMMS, it brings some of the most advanced functionality. The Mission Manager Agent is structurally very similar to the Planner Agent and Chapter 5 will replicate this by being structured very similarly to Chapter 4. The Mission Manager Agent cannot function without the Planner (more precisely, it is difficult to simulate the output of the Planner that is needed by the MMA), so testing will occur with both agents running simultaneously. Results will be presented using the same type of graphical plots that was used for the Planner in Chapter 4 (and particularly in Section 4.4).

5. The Mission Manager Agent

Within the Soar-based Autonomous Mission Management System (SAMMS) presented in this thesis, the Mission Manager Agent is a component that embeds functionality that could also be placed into the Planner Agent presented in Chapter 4. However, as explained in Chapter 3, a design decision was made to separate this functionality into an agent that is supposed to constantly interact with the Planner Agent but is not strictly necessary to its operation. Hence, the Mission Manager Agent (or MMA) is an agent which provides an important set of functions to SAMMS, but which is not essential to it; should the MMA be removed from the SAMMS architecture, the functionality it brings will be lost, but the system will keep the rest of its functionality (e.g., the Planner Agent and the Execution Agent will still be able to operate).

The reason for this cautious approach is that the functionality embedded in the MMA gives it a rather large degree of autonomy in making decisions which affect the flight plan: in short, the MMA has the authority to change mission Objectives (albeit within certain limitations) and thus can severely impact the predictability of SAMMS. This is because while the MMA's response to external inputs is obviously predictable, the nature of these external inputs is not predictable in a realistic mission. The MMA can for example cancel an Objective while flying a mission, in response to a newly detected threat in the Objective area; while the behaviour of the MMA can be predicted (e.g., cancel Objective if a danger with a threat level higher than the Objective execution priority is detected), the occurrence of the external condition cannot be predicted *a priori* (e.g. the operator has no control over it).

This kind of functionality brings a consistent amount of "intelligence" into the system; however, it may be undesirable in certain cases and was separated from the rest of functionality to allow a choice.

In this chapter, the Mission Manager Agent will be described in detail and extensively tested. To perform its functions, the MMA needs a full flight plan (complete with distance, time and fuel estimates) developed by the Planner Agent; for this reason, it will not be tested independently from the Planner. The generic structure of tests will see the Planner generating a plan according to the test scenario, the MMA checking it and eventually modifying it by intervening directly on the mission Objectives. A more detailed explanation of how the two agents interact is provided in Sections 3.2 and 3.3, where the SAMMS architecture and its simulation counterpart are described.

The chapter is divided into four sections. In Section 5.1, the Input/Output (I/O) interface for the MMA is described; while this is based on the interface depicted in Section 2.8, details regarding the actual data being transferred are given here. Section 5.2 introduces the MMA from a discursive point of view; the agent is thoroughly described by looking at the actual Soar code and the way it is organized. The main principles of operation are elicited, the algorithms used to perform specific functions described, and critical code components detailed. Section 5.3 is dedicated to the test scenarios that will be used to demonstrate the MMA; these are based on the scenarios from Section 4.3, but new scenario variations are needed to test specific aspects of the MMA. Finally, Section 5.4 will present the results of the MMA testing campaign; most of the scenarios that will be presented are dedicated to testing a particular algorithm, thus the amount of tests will be consistently inferior compared to the testing campaign for the Planner Agent.

5.1 MMA I/O Interface

Just as for the Planner Agent, the Mission Manager Agent requires a dedicated Soar/Simulink interface, developed on the basis of the one described in Section 2.8. Details regarding this interface will be provided in this section.

The considerations presented in Section 4.1 as an introduction to the description of the Soar/Simulink interface for the Planner Agent are also valid for the MMA. The large amount of data exchanged between the MMA and the SAMMS Simulink architecture is organized into pre-defined structures. As for the Planner Agent (and any Soar agent), the I/O process is managed through two dedicated parts of working memory, the *input-link* and the *output-link*. These structures will now be thoroughly described.

5.1.1 MMA Input

Inputs for the Mission Manager Agent are constituted by six different main categories, as in Table 5.1.

Table 5.1. Main level of input-link structure for the Mission Manager Agent

Code	Name	Values	Description
1	Real-time Data	2	Real-time data, such as Current Time and MMA Trigger
2	Objectives	10	Includes all Objectives and their properties
3	Entities	10	Includes all Entities and their properties
4	Airframe Data	7	From Airframe Data and Health block (see Section 3.3)
5	Flight Plan	100	Flight plan generated by the Planner
6	Plan Estimates	93	Estimates relative to actions in the Plan

The Real-time Data input structure contains data which is updated continuously during a mission, with the exception of airframe status data (placed in a separate input structure); this is the type of data that can be expected to change continuously as opposed to other data which might change only at specific times. Examples of this are the current position and the current time. At present, only two input values are needed for MMA operation: the *Current Time* (which is constantly computed within the Simulink environment) and the *MMA Trigger*, which is a signal generated by the New Plan Trigger function that instructs the MMA to perform its full range of checks on the currently available flight plan. Both values are expressed by a floating point variable.

Table 5.2. Objectives input structure

Code	Name	Definition	Type
2.x.1	Objective Type	As in Table 3.1	integer
2.x.2	Objective Tag	Unique Objective numeric ID tag	integer
2.x.3.1	Objective Position	Latitude of Objective Position	float
2.x.3.2		Longitude of Objective Position	float
2.x.4.1	Priority	Time Priority for Objective	float
2.x.4.2		Execution Priority for Objective	integer
2.x.5	Duty	Type of Duty for Orbit Objectives	integer
2.x.6	Area Type	Type of Area for Search Objectives (box or circle)	integer
2.x.7	Variable 1	Search Accuracy or Orbit Altitude (see text)	float
2.x.8.1	Box-Corner	Latitude of position to define Area of Box Search	float
2.x.8.2		Longitude of position to define Area of Box Search	float
2.x.9	Variable 2	Search Radius or Orbit Time (see text)	float
2.x.10	Target Tag	Unique ID tag for the Entity that is a mission target	integer

The Objectives input structure is just the same as the one for the Planner Agent. It is to be noted that the Objectives that enter here are not the ones coming directly from the operator, but are instead the ones passed through the Objective Mix block described in Section 3.3; this is to ensure that the MMA is consistent with its own decisions, e.g. that once it decides to modify an Objective it will keep the modification, especially when further modifications are needed. The structure contains the list of Objectives with all of their properties. While in theory there is no limitation to the number of Objectives that can be handled by SAMMS, a limit of 10 Objectives is currently placed. In fact, this is due to the difficulty of handling all of this information within Simulink (and in particular within the interface S-functions) rather than to issues deriving from the Soar agents. The meaning of each property was discussed in the aforementioned section; in Table 5.2, the properties are summarized, together with the relevant data types. It is important to note that in order to optimize the exchange of data, a "compression" of the structure has been implemented. In fact, the number and type of Objectives properties varies depending on the Objective type (as detailed in Table 3.2). Since the amount of data exchanged between the Planner and its Simulink environment is quite large, the re-use of some variables was deemed necessary; thus, the "Variable 1" and "Variable 2" properties in Table 4.3 have different meanings depending on the Objective type. For *search area* Objectives, these will represent "Search Accuracy" and "Search Radius" respectively; for *orbit position* Objectives, they will represent "Orbit Altitude" and "Orbit Time" respectively. Note that the data types for these properties are the same. With 10 Objectives with 13 properties each, this part of the input structure amounts for a total of 130 separate values.

Table 5.3. Entities input structure

Code	Name	Definition	Type
3.x.1	Entity Type	Type of entity (building, vehicle, weather zone, etc.)	integer
3.x.2	Entity Tag	ID tag for entity	integer
3.x.3.1	Entity Position	Latitude of most current position info for entity	float
3.x.3.2		Longitude of most current position info for entity	float
3.x.4.1	Movement Info	Entity speed of movement	float
3.x.4.2		Entity direction of movement	float
3.x.5	Entity Behaviour	Friendly, Neutral or Hostile	integer
3.x.6	Threat Level	Threat to the UAV, from negligible to catastrophic	integer
3.x.7.1	Area of Effect	Latitude for Box area, radius for Circle area	float
3.x.7.2		Longitude for Box area, zero for Circle area	float
3.x.8	Stance	Behaviour pattern for the UAV towards entity	integer

The Entities input structure contains the list of Entities, as defined in Section 3.1, together with all their properties. Just as with the Objectives, the number of Entities was limited to 10, even though in theory SAMMS should be able to handle any number of Entities. Table 5.3 shows the Entity properties and the relative data types. With 10 Entities with 11 properties each, this part of the input structure amounts for a total of 110 separate values.

The Airframe Data input structure contains information regarding the aircraft capabilities and status. In fact, this is the entirety of the data that is provided by the Airframe Data and Health block in the Simulink simulation environment (see Section 3.3 and Table 3.9). Table 5.4 displays all variables that are part of this input structure, together with their relative data types. Unlike the Planner Agent, the MMA needs the full set of Airframe Data and Health variables; this is because the MMA uses this information to make decisions regarding the mission. For example, if a payload failure is detected, causing the impossibility to perform Search Objectives, the MMA will cancel all Search Objectives from the mission; or if an engine problem is causing a reduction in fuel efficiency, the MMA will make sure that the mission can be completed, eventually cancelling Objectives that cannot be accomplished. The variables amount for a total of 54 separate values.

The Flight Plan input structure contains the entire flight plan that is generated by the Planner Agent or, to be more precise, it contains all of the Actions that form the flight

Table 5.4. Airframe Data input structure

Code	Name	Definition	Type
4.1.1	Control Status	Status of pitch control system	float
4.1.2		Status of roll control system	float
4.1.3		Status of yaw control system	float
4.2.1	Speeds	Maximum speed	float
4.2.2		Cruise speed	float
4.2.3		Minimum (stall) speed	float
4.2.4		Optimal speed	float
4.2.5		Take-off rotation speed	float
4.2.6		Landing speed	float
4.2.7		Ground Manoeuvre speed	float
4.2.8		Ground Movement speed	float
4.2.9		Current status	float
4.2.10		Current speed limitation	float
4.2.11		Search speed	float
4.2.12		Analyze speed	float
4.2.13		Attack speed	float
4.3.1	Altitudes	Maximum altitude	float
4.3.2		Cruise altitude	float
4.3.3		Minimum ground altitude	float
4.3.4		Climb turn altitude	float
4.3.5		Descent altitude	float
4.3.6		Pre-land altitude	float
4.3.7		Flare altitude	float
4.3.8		Current altitude limitation	float
4.3.9		Search altitude	float
4.3.10		Analyze altitude	float
4.3.11		Attack altitude	float
4.4.1	Distances and angles	Recon distance	float
4.4.2		Attack distance	float
4.4.3		Circle distance	float
4.4.4		Descent distance	float
4.4.5		Pre-land distance	float
4.4.6		Take-off angle	float
4.4.7		Climb angle	float
4.4.8		Flare angle	float
4.5.1	Failures	Loss of Thrust Control	integer
4.5.2		Structural Damage	integer
4.5.3		Loss of Power	integer
4.5.4		Loss of GS Connection	integer
4.5.5		Loss of GPS	integer
4.5.6		Loss of IMU	integer
4.5.7		Loss of Autopilot	integer
4.6.1	Payload Failures	Failed Analyze Target	integer
4.6.2		Failed Attack Target	integer
4.6.3		Failed Search	integer
4.6.4		Failed Orbit (Duty 1)	integer
4.6.5		Failed Orbit (Duty 2)	integer
4.6.6		Failed Orbit (Duty 3)	integer
4.7.1	Fuel and range	Maximum Fuel	float
4.7.2		Remaining Fuel	float
4.7.3		Consumption constant	float
4.7.4		Minimum speed adjust	float
4.7.5		Maximum speed adjust	float
4.7.6		Fuel system status	float

plan, together with the relative properties. In practice, this part of the output-link is structured as in Figure 5.1. While in theory SAMMS should have no limitation in the number of Actions it can handle, a limit of 100 Actions was effected (also as a consequence of the limitation to 10 Objectives). Each Action has its own data structure,

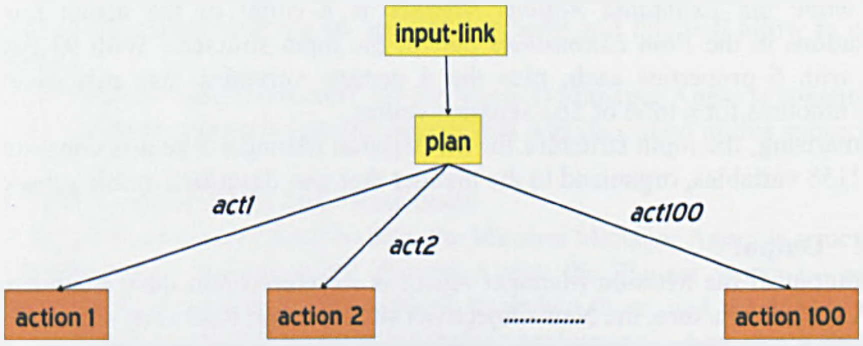


Figure 5.1. Organization of flight plan input of the Mission Manager Agent

just as defined in Section 3.1. Table 5.5 summarizes this, also adding details about data types. With 100 Actions with 13 properties each, this part of the input structure amounts for a maximum of 1300 separate values.

Table 5.5. Action input structure

Code	Name	Definition	Type
5.x.1	Action Type	As defined in Table 3.6	integer
5.x.2	Sequence	Sequence number for the Action	integer
5.x.3.1	Start Position	Latitude of initial position (used for some Action types)	float
5.x.3.2		Longitude of initial position (used for some Action types)	float
5.x.4.1	Position	Latitude of Action reference position	float
5.x.4.2		Longitude of Action reference position	float
5.x.5	Time	Time property of Action (usually time limit)	float
5.x.6	Heading	Bearing to be kept for certain Action Types	float
5.x.7	Altitude	UAV Altitude specified for Action	float
5.x.8	Duty	Duty type for Circle Actions, otherwise parent Objective type	integer
5.x.9	Speed	UAV Speed for Action	float
5.x.10	Target	Defines a specific target for Recon and Attack	integer
5.x.11	Objective	Parent Objective ID tag	integer

The Plan Estimates part of the input structure contains additional information generated by the Planner regarding the flight plan; for each action, estimates of the distance covered, time needed and fuel spent are provided. It is organized similarly to the flight plan (see Figure 5.1), but with a reduced number of Actions since the take-off and approach phases are grouped into a single estimate (a single estimate for the take-off phase and a single estimate for the approach phase). For each Action, the six

Table 5.6. Plan Estimates input structure

Code	Name	Definition	Type
6.x.1	Action Distance	Distance that will be covered for this Action	float
6.x.2	Action Time	Time to complete this Action	float
6.x.3	Action Fuel	Fuel to complete this Action	float
6.x.4	Total Distance	Total distance covered after this Action	float
6.x.5	Total Time	Total mission time after the Action is completed	float
6.x.6	Total Fuel	Total fuel used after Action is completed	float

variables in Table 5.6 are provided. In short, distance, time and fuel estimates are derived for the Action, then this are added to the estimates of the Actions that precede it in the flight plan, thus providing the “total” values. In addition to this, the output structure provides four variables: the *Total Distance Estimate*, *Total Time Estimate* and *Total Fuel Estimate* simply reproduce the *Total Distance*, *Total Time* and *Total Fuel* values for the last Action in the Flight Plan (which will obviously be the approach phase), while the *Estimated Actions Number* is a count of the actual number of augmentations in the *Plan Estimations* part of the input structure. With 93 Estimation Actions with 6 properties each, plus the 4 generic variables, this part of the input structure amounts for a total of 562 separate values.

Summarising, the input structure for the Mission Manager Agent is constituted by a total of 2158 variables, organized in the manner that was described in this subsection.

5.1.2 Output

The output of the Mission Manager Agent is structured into three main categories: the MPA Ignore structure, the New Objectives structure and Real-time structure.

The MPA Ignore part of the output structure contains a set of information which is needed by the *mission-path-adjust* (MPA in short) algorithm in the Planner Agent (see Section 4.2.8). This basically tells the Planner what to do in case an Objective is placed in the area of effect of a dangerous Entity. Note that the MMA instructs the Planner to ignore an Entity, rather than an Objective; this is because the Entity is to be ignored both whilst flying towards the Objective and whilst flying out from the Objective. Consequently, the MMA sends a binary value (set to 0 to indicate that the Entity should be considered by the mission-path-adjust algorithm, set to 1 to indicate that it should be ignored) for each Entity, and the structure contains therefore 10 separate values in the current implementation, corresponding to the currently implemented maximum of 10 Entities.

Table 5.7. MPA Ignore output structure

Code	Name	Definition	Type
1.1	Entity 1	0 for normal operation, 1 if Entity is to be ignored	integer
1.2	Entity 2	0 for normal operation, 1 if Entity is to be ignored	integer
1.3	Entity 3	0 for normal operation, 1 if Entity is to be ignored	integer
1.4	Entity 4	0 for normal operation, 1 if Entity is to be ignored	integer
1.5	Entity 5	0 for normal operation, 1 if Entity is to be ignored	integer
1.6	Entity 6	0 for normal operation, 1 if Entity is to be ignored	integer
1.7	Entity 7	0 for normal operation, 1 if Entity is to be ignored	integer
1.8	Entity 8	0 for normal operation, 1 if Entity is to be ignored	integer
1.9	Entity 9	0 for normal operation, 1 if Entity is to be ignored	integer
1.10	Entity 10	0 for normal operation, 1 if Entity is to be ignored	integer

The New Objectives part of the output structure contains the main MMA output: after having received a set of Objectives as input (see Table 5.2), the MMA will change them according to the perceived inconsistencies, and will output the set of modified Objectives using this structure. The structure is organized exactly as the Objectives input structure (ten separate Objectives, each with the properties shown in Table 5.2); obviously, if the MMA has decided any change on the Objectives, the data contained in the structure will be different. All considerations valid for the Objectives input structure are also valid for the New Objectives output structure. With 10 Objectives with 13 properties each, this part of the output structure amounts for a total of 130 separate values.

The Real-time output structure contains data which is updated continuously during a mission, meaning that this is the type of data that can be expected to change continuously as opposed to other data which might change only at specific times. At present, the only variable that is present in this structure is the *Current Time*. Current Time is obtained from the Real-time input structure and sent back to the Simulink architecture as a check. In cases where the MMA encounters problems (e.g. it crashes), the Current Time output will not be updated correctly and this will allow to detect the agent failure.

Summarising, the output structure for the Mission Manager Agent is constituted by a total of 141 variables, organized in the manner that was described in this subsection.

5.2 MMA Algorithms and Soar code

From the point of view of Soar coding, the Mission Manager Agent is structured in a manner which is very similar to the Planner Agent: the Planner works by generating flight plans and then waiting for instructions to update them, and the MMA works by checking the flight plans and then waiting before performing a new check. In practice, this means that the MMA, just as the Planner, has two main substates that originate from the top-state. This separates the Planner and the MMA from the Execution Agent, which is organized in a radically different manner (as was anticipated in Section 3.2 and will be seen in Chapter 6).

Figure 5.2 shows the hierarchy of states within the Mission Manager Agent. Comparing this to Figure 4.3 in Chapter 4, the *wait-for-changes* state is the equivalent of the *modify-plan* state and the *check-plan* state is the equivalent of the *generate-plan* state. The *check-plan* state has three further substates: *mpa-ignore*, *modify-objectives* and *add-objectives*; these are entered in this sequence during the process of checking a flight plan from the Planner. The various substates will be illustrated in the following subsections; for the moment, the operation of the *wait-for-changes* and *check-plan* states will be described.

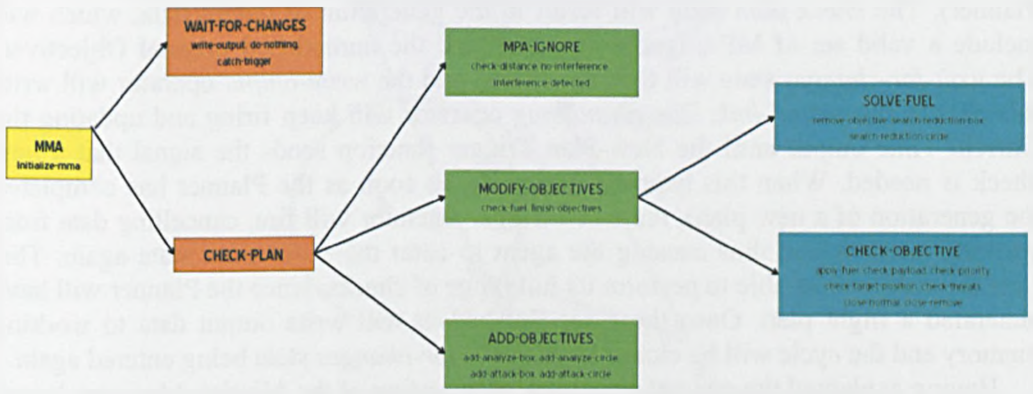

MMA
Initialize-mma
WAIT-FOR-CHANGES
write-output do-nothing
catch-bigger
CHECK-PLAN
MPA-IGNORE
check-distance no-interference
interference-detected
MODIFY-OBJECTIVES
check-fuel finish-objectives
ADD-OBJECTIVES
add-analyze-box add-analyze-circle
add-attack-box add-attack-circle
SOLVE-FUEL
remove-objective search-reduction-box
search-reduction-circle
CHECK-OBJECTIVE
apply-fuel check-payload check-priority
check-target-position check-threats
close-normal close-remove

Figure 5.2. Mission Manager Agent structural overview

When the MMA is first initialized, the *initialize-mma* operator will fire and set up specific bits of working memory where generic long-term knowledge is stored. The Planner will not yet have generated a flight plan, thus after initialization the *check-plan* state will be entered but cannot perform checks on a flight plan. However, it will produce the MPA-Ignore data (necessary for the Planner) and write output to indicate that the agent is active. From a Soar point of view, this means that after agent initialization, a Soar production will search the working memory for the output that the MMA is expected to produce; not finding it, it will create an impasse, and the *check-plan* substate will be created to solve it. Once the *check-plan* state completes its process of checking the current flight, it writes to the working memory, causing the impasse to be solved and the substate to be deleted. The fact that a flight plan is not currently available does not change this; in fact, the output from the MMA is needed within SAMMS since it is the set of Objectives from the MMA that should be fed to the Planner. Before the Planner has generated a plan, the MMA will still produce its output, but the Objectives will naturally be unmodified. In short, when the MMA is initialized, the *check-plan* state is entered even if there is no currently available flight plan, so as to generate its expected output (including the MPA-Ignore data).

After the *check-plan* state has been entered and closed (when it has produced the output data), another production will fire, creating the *wait-for-changes* substate. This is basically the normal operating mode for the MMA; when in this state, the MMA keeps updating the Current Time output (see Section 5.1.2) and monitors the MMA Trigger input, waiting to be instructed to perform another full check on the current flight plan. The *wait-for-changes* state is implemented by three operators:

- *write-output* is the operator that collects output data from working memory and writes it into the *output-link*; it executes immediately after closure of the *check-plan* state
- *do-nothing* is the operator that keeps updating the Current Time output during normal operation; it will always fire when the MMA is in the *wait-for-changes* state, but has a worst preference so that the *catch-trigger* operator will be preferred if available
- *catch-trigger* is the operator that fires when the New-Plan Trigger function (see Section 3.3) sends the signal that a new check on the current flight plan is needed; it results in the cancellation from working memory of the output data that is generated by the *check-plan* state, so that it will be entered again

A typical case of MMA operation will see the *initialize-mma* operator firing, followed by a first iteration of *check-plan* (without an available flight plan from the Planner). The *check-plan* state will result in the generation of output data, which will include a valid set of MPA-Ignore variables and the unmodified original Objectives. The *wait-for-changes* state will then be entered and the *write-output* operator will write this data to the *output-link*. The *do-nothing* operator will keep firing and updating the Current Time output until the New-Plan Trigger function sends the signal that a new check is needed. When this happens (normally, as soon as the Planner has completed the generation of a new plan), the *catch-trigger* operator will fire, cancelling data from working memory and thus causing the agent to enter the *check-plan* state again. This time the state will be able to perform its full range of checks, since the Planner will have generated a flight plan. Once these are finished, it will write output data to working memory and the cycle will be closed by the *wait-for-changes* state being entered again.

Having explained the general principles of operation of the Mission Manager Agent, it is now possible to focus on the lower level substates, where the actual algorithms of the agent are implemented.

5.2.1 Check-plan

The *check-plan* state is basically the container for the *mpa-ignore*, *modify-objectives* and *add-objectives* states. Aside from writing the list of updated Objectives to a working memory location that is directly linked to the top-state, the *check-plan* state does not perform any actual task.

The *mpa-ignore*, *modify-objectives* and *add-objectives* states are entered in this order. The *mpa-ignore* state (described in Section 5.2.2) outputs a vector of binary values that tell the Planner Agent what it should do when an Objective is placed within the area of effect of a dangerous Entity. The *modify-objectives* state, described in Section 5.2.3, outputs an updated list of Objectives, where the original Objectives might have been modified or cancelled. The *add-objectives* state takes this updated list and eventually adds further Objectives to the list (as shown in Section 5.2.6).

5.2.2 MPA-Ignore

The *mpa-ignore* substate is the state where the Mission Manager Agent decides how the Planner should behave in cases where an Objective is within the area of effect of a dangerous Entity. This is needed since in these cases the *mission-path-adjust* algorithm must be inhibited or it would be trying to accomplish an impossible task.

To better understand the concept, let us consider the example situation shown in Figure 5.3. In this example, there are four Objectives and two dangerous Entities with an area of effect, indicated as Entity A and Entity B in the figure. No Objective is placed in the area of effect of Entity B, so the mission-path-adjust can operate as normal and set a new waypoint in the flight plan so that the danger area will not be entered. However, Objective 1 is placed within the area of effect of Entity A; this obviously means that it will simply not be possible to both avoid the danger and accomplish the Objective. Two main options are available to SAMMS in such a case: if the threat level of the Entity is higher than the Execution priority of the Objective, the Objective will be cancelled (more on this in Section 5.2.5); otherwise, the flight plan will anyway include the Objective, and therefore it will at some point enter the danger area.

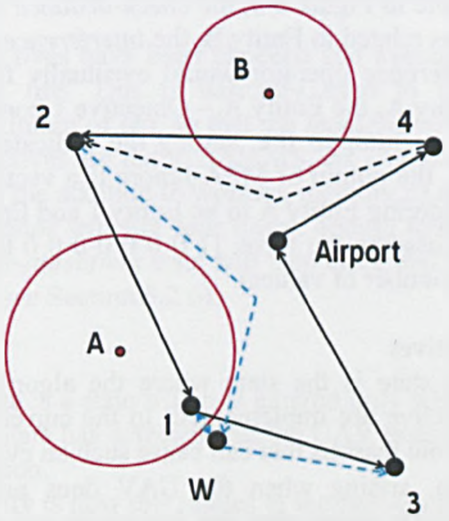


Figure 5.3. Example of Objectives and Entity positions

From a theoretical point of view, several courses of action can be taken once the decision to pursue the Objective placed within the danger area is made. The simplest course of action is to simply ignore the Entity; this is what is currently done in SAMMS, as can be seen in the example in the figure. The MMA decides that Objective 1 should be pursued (its execution priority is higher or equal to the threat level of Entity A) and therefore instructs the Planner to inhibit the *mission-path-adjust* algorithm for Entity A. Consequently, the two Travel Actions that intersect Entity A (from Objective 2 to Objective 1, and from Objective 1 to Objective 3) will not be modified by any waypoint.

A more intelligent approach might be to try minimizing the distance flown within the danger area; this would involve defining a waypoint which is placed outside the danger area, such as the waypoint W that can be seen in the figure. The *mission-path-adjust* algorithm would then work normally between points 2 and W, and the distance flown within the danger area would be minimized by flying to W, then to Objective 1, then back to W and finally continuing to Objective 3. Another option might be to keep the UAV at the maximum possible distance from the centre of the Entity area of effect. In any case, the present implementation of SAMMS simply chooses to ignore the Entity; but this is a clear possibility for improvement.

The *mpa-ignore* state performs its function using three operators. The *check-distance* operator fires for all the possible combinations of dangerous Entities and Objectives; in the example of Figure 5.3, it would fire eight times, once per Objective for Entity A and once per Objective for Entity B. The operator calculates the distance between the Objective position and the centre of the Entity area of effect, comparing with it the area of effect radius. It thus determines whether the Objective is or is not within the danger area, and writes this information to working memory. If an Objective is found to be within the area of effect of a dangerous Entity, the *interference-detected* operator will fire. This will cause the Entity to be marked as an Entity to be ignored, e.g. its corresponding variable in the MPA-Ignore output structure will be set to value 1. The *no-interference* operator fires instead once all the Entity-Objective combinations for an Entity have been checked without triggering the *interference-detected* operator. This means that no Objective is present within the Entity area of effect, therefore the corresponding variable in the MPA-Ignore output structure will be set to value 0 so that the Entity is treated normally by the mission-path-adjust algorithm in the Planner.

Referring to the example in Figure 5.3, the *check-distance* operator would fire eight times. For all combinations related to Entity B, the *interference-detected* operator would not fire and the *no-interference* operator would eventually fire, setting its dedicated output value to 0. For Entity A, the Entity A – Objective 1 combination would result in the *interference-detected* operator to fire, setting the dedicated output value to 1. As specified in Section 4.1.2, the output of MPA-Ignore is a vector of binary values, each related to an Entity. Considering Entity A to be Entity 1 and Entity B to be Entity 2, the output vector would then assume the form: [1 0 0 0 0 0 0 0 0 0] (the vector has been filled with the maximum number of values).

5.2.3 Modify-Objectives

The *modify-objectives* state is the state where the algorithms that can decide to change or cancel an Objective are implemented. In the current implementation of the MMA, there are five possible reasons that can cause such an event:

- a fuel problem, arising when the UAV does not carry enough fuel to complete a mission (even after the *fuel-check* and *reduce-speed* algorithms from the Planner have tried to reduce fuel consumption, see Section 4.2.10)
- a threat problem, arising when an Objective is placed within the area of effect of a dangerous Entity
- a target position problem, arising when the target of an Analyze or Attack Objectives is not static, but is instead moving
- a priority problem, arising when a time priority for an Objective cannot be respected in any case
- a payload failure problem, arising when a the on-board fault detection system has detected a payload failure that would inhibit the UAV from performing certain mission types

The *modify-objectives* state deals with these issues through the combination of two operators (*check-fuel* and *finish-objectives*) and two substates (*solve-fuel* and *check-objective*, which will be described in Sections 5.2.4 and 5.2.5).

The first operator to fire is *check-fuel*; this operator compares the total estimated fuel consumption provided by the Planner with the current on-board fuel and decides whether it is sufficient. It does so by writing a *fuel-checked* augmentation into the working memory related to the *modify-objectives* state; if the fuel is sufficient, the *fuel-checked* augmentation will be set to 1, otherwise it will be set to -1.

When the *fuel-checked* augmentation is set to 1, further algorithms will not be triggered. If instead it is set to -1, the *solve-fuel* substate will be triggered. This is a state that tries to solve the issue of insufficient fuel and will be described in Section 5.2.4. Whether or not the *solve-fuel* substate has been entered, the next step will see the agent entering the *check-objective* substate, which is the state where the modifications to Objectives are actually computed and written to working memory.

The *check-objective* substate is iterative, meaning that an instance of the state will be opened for each Objective. So, after *check-fuel* and eventually *solve-fuel*, a *check-objective* substate will be entered for Objective 1. After performing all the checks that will be described in Section 5.2.5, the *check-objective* substate for Objective 1 will be closed, writing an updated Objective 1 to working memory (this will be modified only if needed, e.g. if any of the problems described earlier have been encountered). Then, a new *check-objective* substate will be entered, related to Objective 2 (obviously only if an Objective 2 is present), and so on. The cycle will continue until all Objectives have been covered.

Once all actual Objectives have been checked and eventually modified, the *finish-objectives* operator will fire. This is basically used to fill the remaining empty Objectives up to the maximum of ten, so that the *modify-objectives* state can be closed. For example, if six Objectives are fed to SAMMS, six *check-objective* substates will be entered, each resulting in the addition to working memory of an updated Objective, then the *finish-objectives* operator will fire four times, adding four empty Objectives to the Objective list. The *modify-objectives* state will then close and the MMA will continue to the *add-objectives* state (see Section 5.2.6).

5.2.4 Solve-fuel

The *solve-fuel* substate is a state which is entered only when the *check-fuel* operator in the *modify-objectives* state has detected that the UAV does not have enough on-board fuel to complete the mission.

The first thing to clarify is how this relates to the *fuel-check* algorithm in the Planner Agent. Recalling from Section 4.2.10, the *fuel-check* algorithm will detect the same issue and try to solve it by reducing the flight speed. This in turn causes a reduction of the fuel consumption estimate which might be sufficient to ensure that the mission will be completed. However, flight speed cannot be reduced indefinitely (and, considering the fuel consumption model in use, speed reductions below the Orbit Speed value will not cause a reduction in fuel consumption), so this attempt might not be sufficient.

Since the MMA performs its functions on a flight plan which has been previously generated by the Planner, cases of insufficient fuel will already have been addressed (successfully or not) by the *fuel-check* algorithm in the Planner. If this is not successful, the *check-fuel* operator in the *modify-objectives* state will detect it and the *solve-fuel* substate will be entered. As will be seen, there are basically two possible options implemented within the *solve-fuel* substate; whichever is chosen will result in substantial modifications to the Objective set (albeit only concerning a single Objective). When the MMA finally outputs its modified set of Objectives, the Planner will be triggered to update the flight plan, and will thus recalculate the flight plan estimates (and therefore the fuel consumption estimate) for the updated flight plan. The cycle is then closed, since these new estimates will be used by both the *fuel-check* algorithm in the Planner and the *check-fuel* operator in the MMA to verify whether the changes to the flight plan have been sufficient in ensuring that it can be completed.

A reduction of fuel consumption can basically be obtained in two manners: by reducing the flight speed and by reducing the distance flown. The first option is the one attempted by the Planner Agent, so in the MMA the second option is considered. Two main ways to reduce the distance flown have been considered:

- modification of Objectives of the Search type, by changing the Search Accuracy mission parameter
- cancellation of an Objective

These options are reflected by the operators which are implemented within the solve-fuel substate.

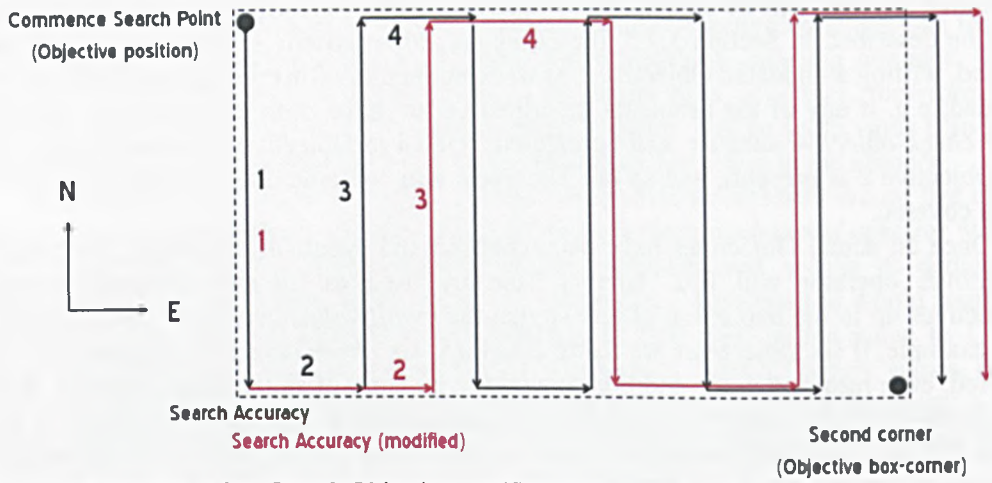


Figure 5.4. A box-Search Objective modified by the *search-reduction-box* operator

The *search-reduction-box* and *search-reduction-circle* operators attempt to reduce the flown distance by increasing the Search Accuracy parameter. This option is of course only available if a Search Objective is present. Search Objectives usually involve covering large distances and changing appropriately the Search Accuracy value can result in a consistent reduction of the covered distance. Figure 5.4 shows the effect of the *search-reduction-box* operator on a Search Objective of type box; the modified search pattern is shown in red. It can be seen that increasing the Search Accuracy value by roughly 50%, the resulting pattern will include five vertical legs rather than seven, yielding a distance reduction of slightly less than two full vertical legs (a small increase in the horizontal distance flown can be noted). Figure 5.5 shows instead the effect of the *search-reduction-circle* operator on a Search Objective of type circle; the modified search pattern is shown in red. It can be seen that increasing the Search Accuracy value by roughly 90%, the resulting pattern will include two "diamonds" rather than three, yielding a considerable distance reduction. For both operators, the increase of the Search Accuracy value is calculated as a function of the amount of missing fuel (e.g. the amount of fuel that would need to be added in order to complete the mission with the flight plan that is being checked).

The *remove-objective* operator attempts to reduce the flown distance by completely cancelling one of the Objectives. The operator fires for all current Objectives and it is

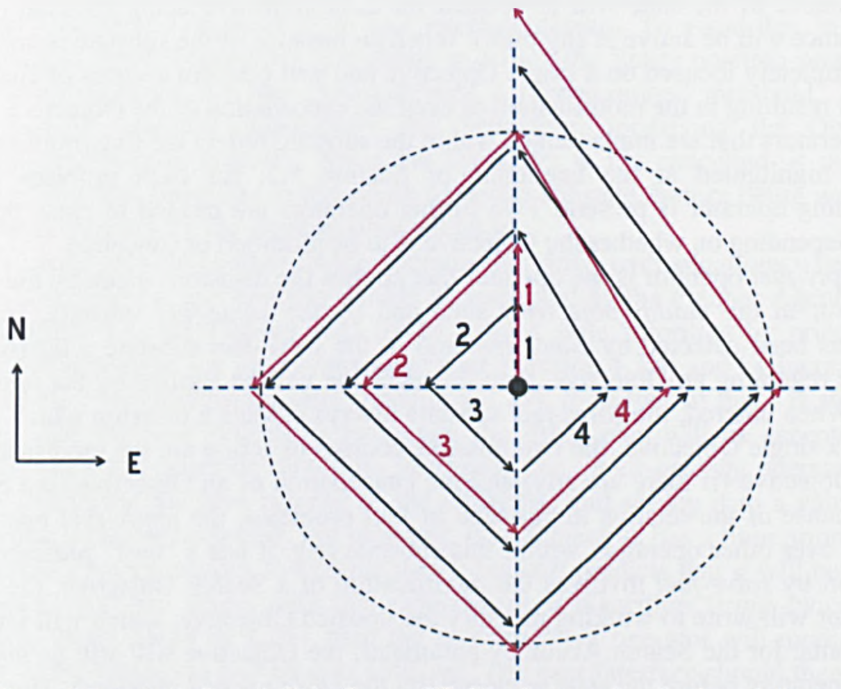


Figure 5.5. A circle-Search Objective modified by the *search-reduction-circle* operator

through the use of preference productions that a choice is made between the various Objectives. The following rules of preference are currently implemented:

- if a *search-reduction-box* or *search-reduction-circle* operator has been proposed, then prefer this to *remove-objective* (e.g. it is preferred to try changing the accuracy of a search rather than cancelling an Objective)
- if two *search-reduction* operators have been proposed (independently of their being of the *box* or *circle* type), then prefer the one which involves the longest flown distance
- if two *remove-objective* operators have been proposed, then ignore the one whose related Objective has a higher execution priority (e.g. Objectives with lower execution priority are cancelled before the ones with higher execution priority)
- if two *remove-objective* operators have been proposed, then prefer the one which is the farthest from the starting airport (e.g. the farthest Objectives is cancelled before others)

Note that when the *remove-objective* operator fires, it does so for all available Objectives; with these rules, it will choose the Objective to remove within those with the lowest execution priority, selecting the farthest from the base airport among these.

It is to be noted that the *solve-fuel* state will always close selecting a single course of action, whether it is a search-reduction or the removal of an Objective. The Objective list will be modified accordingly (details in Section 5.2.5) and then eventually a new flight plan will be generated by the Planner. Should this still not be sufficient to ensure mission completion, the *check-fuel* operator will detect this and the *solve-fuel* substate will be entered again, resulting in a new change to the Objective list.

5.2.5 Check-objective

The *check-objective* substate is the state where modifications to Objectives are actually implemented. As anticipated in Section 5.2.4, the substate is iterative, meaning that an instance of the state will be created for each Objective being checked (only a single instance will be active at any time). When an instance of the substate is entered, it will be completely focused on a single Objective and will perform a series of checks on it, possibly resulting in the modification or even the cancellation of the Objective.

The operators that are implemented within the substate reflect the five problem types that were highlighted at the beginning of Section 5.2; for each problem type a corresponding operator is present. Two further operators are needed to close the state properly, depending on whether the Objective is to be modified or cancelled.

The *apply-fuel* operator is the operator that applies the decisions made by the *check-fuel* operator in the *modify-objectives* state and by the *solve-fuel* substate. If a fuel problem has been detected by *check-fuel* and in the *solve-fuel* substate a decision has been made regarding an Objective, then this decision will be applied by the *apply-fuel* operator. When entered, the *solve-fuel* substate always outputs a decision which is only applied to a single Objective; the two possible courses of action are the modification of a Search Objective (if there are any) and the cancellation of an Objective (see Section 5.2.4). Because of the relative importance of fuel problems, the *apply-fuel* operator is prioritized over other operators within this substate (e.g. it has a "best" preference). If the decision by *solve-fuel* involves the modification of a Search Objective, the *apply-fuel* operator will write to working memory the updated Objective, which will include a different value for the Search Accuracy parameter; the Objective will still be available for other operators before the state is closed (by the *close-normal* operator). This means that a Search Objective that has been modified by the *apply-fuel* operator might still be modified (or even cancelled) because of other problems; for example, a priority problem might arise, thus the *priority-check* operator would also fire and in the end the updated Objective would have different values for its Search Accuracy and Time Priority parameters. If instead the decision by *solve-fuel* involves the cancellation of an Objective, further operators will be inhibited, since the Objective would be cancelled anyway; in this case, the *close-remove* operator would fire immediately.

The *check-threats* operator is the operator that verifies whether an Objective is placed within the area of effect of a dangerous Entity. In such cases, two courses of action are possible:

- if the threat level of the Entity is lower or equal than the execution priority of the Objective, the Entity is ignored (the decision is made within the *mpa-ignore* state, see Section 5.2.2)
- if the threat level of the Entity is greater than the execution priority of the Objective, the Objective is cancelled

When the second option is valid, the *check-threats* operator will fire and set up the cancellation of the Objective; this is put into practice by the *close-remove* operator, which will always fire immediately after the *check-threats* operator (more precisely, it will always fire immediately after a decision to cancel an Objective has been made).

The *check-target-position* operator is an operator that fires when the target of an Analyze or Attack Objective is moving. When the Objective is first assigned, its Position property will coincide with the Position property of its corresponding target Entity. However, while the Entity position property is automatically updated, the Objective position property is not (since this should be done by the UAV operator). This does not have any direct consequences, since the Execution Agent will use data from the Entity position property to reach the Objective, but eventually the difference between the Objective position and the Entity position might grow very large; this could in turn cause problems, for example the UAV operator might be unaware of how much the target has moved (depending on user interface design). To avoid this, in such cases the Objective position property is updated to reflect the Entity position property when the distance between the two grows beyond a predetermined threshold. The *check-target-position* operator implements this behaviour, modifying the Objective by changing its Position property. Since the Objective is not cancelled, it may also be modified by other operators in the *check-objective* substate, before being written down to working memory by the *close-normal* operator.

The *check-priority* operator is the operator that deals with situations where the time priority specified for an Objective cannot be respected by the Planner Agent. Recalling from Section 4.2, the Planner will try to respect time priorities by prioritizing the relative Objectives in the *plan-sequencing* state (Section 4.2.4) and by increasing flight speed in the *plan-finalization* state (Section 4.2.10). If neither of these is successful in ensuring that the time priority is respected, the flight plan will be accepted as it is, however the UAV operator should be informed. The *check-priority* operator performs this by modifying the time priority of the Objective and setting it to a value that will ensure that it can be respected. For example, if an Objective has a time priority of 1000 sec and the Planner decides to execute it first but still predicts that it will be completed only after 1500 sec (even with the *increase-speed* algorithm firing twice and thus increasing flight speed by 21%), then the *priority-check* operator will modify the time priority of the Objective and set it to 1500 sec. The user interface should then report this to the UAV operator.

The *check-payload* operator is the operator that acts when a payload failure is detected by the on-board fault detection system. It is assumed that in order to perform the various Objective types (apart from Transit Objectives), specific payload will be needed; the payload might or might not change depending on the Objective type, and the ability to perform an Objective type might be affected by a payload failure. When a payload failure that makes it impossible to perform an Objective type is detected, then the *check-payload* operator will cancel that Objective. For example, if a camera payload sensor is necessary to perform Analyze Objectives, when the fault detection system detects the failure of that camera, the check-payload operator will fire for all Analyze Objectives and set up their cancellation; this is put into practice by the *close-remove* operator, which will always fire immediately after the *check-payload* operator.

The *close-normal* and *close-remove* operators are mutually exclusive; these operators are the ones that actually close the *check-objective* substate (for a single Objective). The *close-normal* operator fires when an Objective has undergone all five tests in the *check-objective* substate and no decision to remove it has been taken. The operator simply writes the updated Objective (e.g., with all the eventual changes applied by the *apply-fuel*, *check-target-position* and *check-priority* operators) to a specific location in working memory (which is linked to the *modify-objectives* state). This in turn will cause the *check-objective* substate to close, then either a new instance of *check-objective* will be entered or the *finish-objectives* operator will fire (see Section 5.2.3).

The *close-remove* operator fires immediately after one among the *apply-fuel*, *check-threats* and *check-payload* operators has formalized the decision to remove an Objective. The operator writes the updated Objective to the same location in working memory as for the *close-normal* operator. It is to be noted here that when an Objective is cancelled, it is not entirely removed from the Objective list; instead, its Objective Type property is assigned a value of 6, which is a code that effectively tells the Planner to ignore it. This is to keep a record of all Objectives that have been assigned during a mission, and also to avoid confusion should the UAV operator add a new Objective during the mission. The cancelled Objective will only keep its Objective Type and Objective Tag properties; once it has been written to working memory by the *close-remove* operator, the *check-objective* substate will close.

As detailed in Section 5.2.3, after all Objectives have been checked by the *check-objective* operator and the *finish-objectives* operator has completed the Objectives list, the *modify-objectives* will be closed, having provided an updated list of Objectives.

5.2.6 Add-objectives

The *add-objectives* state is the state where new "opportunity" Objectives are added to the Objective list. In such cases, the UAV operator will not have specified those Objectives, but will have indicated that if during flight the opportunity arises, additional Objectives might be pursued.

The typical case (and the only one currently implemented within SAMMS) is related to Search Objectives: for these, the UAV operator might specify that the Search Objective is a Search-and-Analyze or Search-and-Attack Objective. This means that if during the Search a new Entity of the specified type is detected within the search area, an Analyze/Attack Objective will be added to the Objective list. For example, the UAV operator might want to identify all vehicles in an area and then take pictures of them. A Search-and-Analyze Objective focused on vehicle Entities for the area would accomplish this; the area would be searched using one of the patterns described in Section 4.2.5 and all newly detected Entities of type vehicle would have a new corresponding Analyze Objective added (set with the Entity as the target).

The state performs its function using four operators; the operators are similar, but differ depending on the type of search pattern and on the action which is required (e.g. whether an Analyze or Attack Objective should be added).

The *add-analyze-box* operator is the operator that adds Objectives during a Search-and-Analyze Objective that is using the box area type (e.g. the parallel track search pattern). Its proposal production verifies that an Entity is detected within the search area of a Search Objective that has the box area type and has been specified as a Search-and-Analyze Objective. The application production will add a new Objective only if the Entity is of the required type and if no other Objective in the list has the Entity as Objective; otherwise, no Objective will be added. The new Objective has the following four relevant properties: the *objective-type* is set to 1, the *objective-tag* is set to the appropriate value (current Objective count + 1), the *position* is set to the current Entity position, the *target* is set to the value of the Entity tag property.

The *add-analyze-circle* operator is the operator that adds Objectives during a Search-and-Analyze Objective that is using the circle area type (e.g. the expanding diamond spiral search pattern). Its proposal production verifies that an Entity is detected within the search area of a Search Objective that has the circle area type and has been specified as a Search-and-Analyze Objective. The application production will add a new Objective only if the Entity is of the required type and if no other Objective in the list has the Entity as Objective; otherwise, no Objective will be added. The new Objective has the following four relevant properties: the *objective-type* is set to 1, the *objective-tag* is set to the appropriate value (current Objective count + 1), the *position* is set to the current Entity position, the *target* is set to the value of the Entity tag property.

The *add-attack-box* operator is the operator that adds Objectives during a Search-and-Attack Objective that is using the box area type (e.g. the parallel track search pattern). Its proposal production verifies that an Entity is detected within the search area of a Search Objective that has the box area type and has been specified as a Search-and-Attack Objective. The application production will add a new Objective only if the Entity is of the required type and if no other Objective in the list has the Entity as Objective; otherwise, no Objective will be added. The new Objective has the following four relevant properties: the *objective-type* is set to 2, the *objective-tag* is set to the appropriate value (current Objective count + 1), the *position* is set to the current Entity position, the *target* is set to the value of the Entity tag property.

The *add-attack-circle* operator is the operator that adds Objectives during a Search-and-Attack Objective that is using the circle area type (e.g. the expanding diamond spiral search pattern). Its proposal production verifies that an Entity is detected within the search area of a Search Objective that has the circle area type and has been specified as a Search-and-Attack Objective. The application production will add a new Objective only if the Entity is of the required type and if no other Objective in the list has the Entity as Objective; otherwise, no Objective will be added. The new Objective has the following four relevant properties: the *objective-type* is set to 2, the *objective-tag* is set to the appropriate value (current Objective count + 1), the *position* is set to the current Entity position, the *target* is set to the value of the Entity tag property.

While in theory there is no limitation to the number of Objectives that could be added in this way, as previously stated in the current implementation of SAMMS the maximum number of Objectives is limited to ten. Therefore, the operators will be inhibited from firing if the Objective list that is the output of the *modify-objectives* state includes already ten Objectives, or whenever the *add-objectives* state will have added enough Objectives so that the limit is reached.

5.3 MMA Test scenarios

In Section 4.3, seven test scenarios for the Planner Agent were introduced. Most of these scenarios presented several possible variations, normally introducing additional Objectives or changing specific conditions, so as to test specific Planner algorithms. Testing of the Mission Manager Agent will be based on the same scenarios, however dedicated scenario variations will be introduced in order to test specific algorithms.

In this section, the scenario variations developed in order to test MMA functionality will be described. In most cases, the description will keep the original scenario definition from Section 4.3 as a reference. Only scenarios 1, 2, 3 and 4 will be used.

5.3.1 Scenario 1, variation d

The original scenario 1 (described in Section 4.3.1) is the simplest scenario and was primarily intended as an early test of SAMMS functionality. The scenario originally came with three variations (1a, 1b and 1c), but only variation 1c was effectively used during the testing campaign (see Section 4.4.1).

For the MMA testing campaign, a new scenario variation, called variation 1d, has been introduced with the aim of testing three of the algorithms described earlier: *mpa-ignore* (Section 5.2.2), *check-threats* and *check-target-position* (both in Section 5.2.5).

Figure 5.6 shows the updated map for scenario 1; comparing it to Figure 4.11 in Chapter 4, two differences can be noted. Firstly, a new Entity (Entity 3) is present; this Entity has a threat level of 3 and an area of effect of 3000 m, and is located so that Objective 3 is placed within its area of effect. Consequently, the Entity will trigger either the *mpa-ignore* algorithm or the *check-threats* algorithm (depending on the execution priority of Objective 3). Secondly, Entity 1 (the target for Objective 2) is moving; at a specified time, its position is updated, thus triggering the *check-target-position* algorithm.

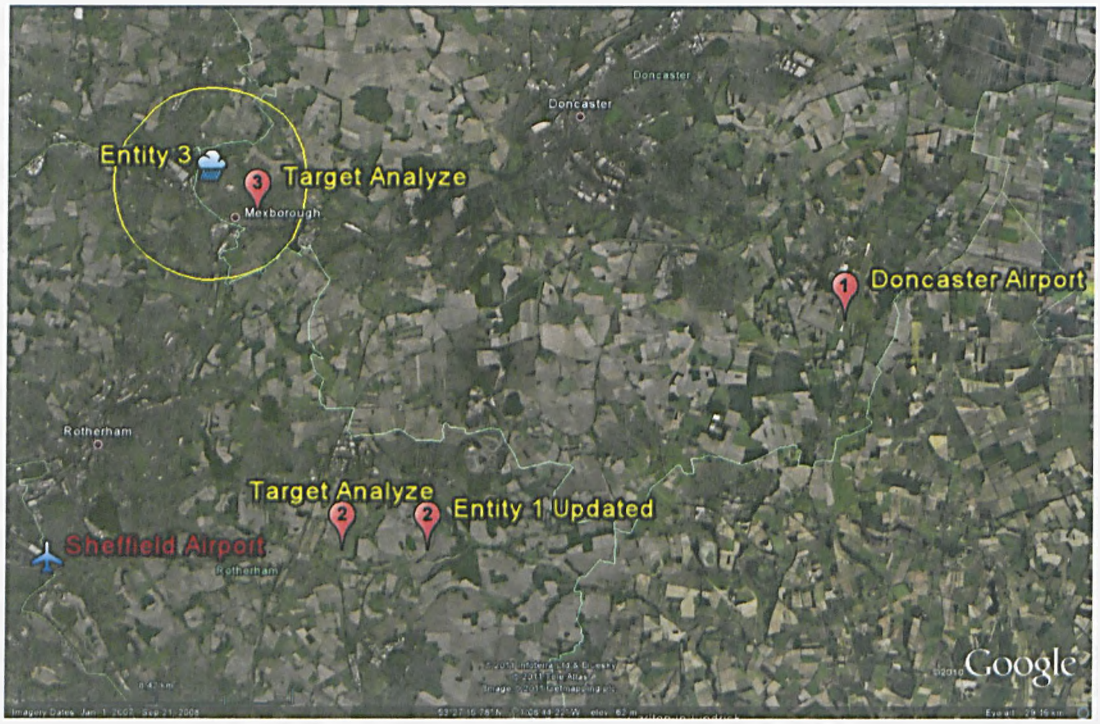


Figure 5.6. Scenario 1 map, updated for variation 1d

Table 5.8 shows details regarding the Objectives. Two columns of data are reserved for Objective 2, with the first indicating the original Objective position and the second indicating the new position of Entity 1; notice however that the updated Objective will be an output of the MMA rather than an input from the UAV operator. Entity 3 has the properties shown in Table 5.9.

Table 5.8. Details of Scenario 1 locations and Objectives, updated for variation 1d

Starting	ID	Property	Transit	Target-Analyze	Target-Analyze	Target-Analyze
Park: 53°23'42"N 1°22'51"W	1	Type	5	1	1	1
	2	ID	1	2	2	3
	3	Position	53°27'58"N 1°00'32"W	53°24'05"N 1°14'33"W	53°24'05"N 1°12'11"W	53°29'39"N 1°17'39"W
Runway: 53°23'37"N 1°22'48"W	4	Priority	0, 5	0, 5	0, 5	0, 2 (4)
	5	Duty	/	/	/	/
	6	Area Type	/	/	/	/
Runway Heading: 280	7	Variable 1	13	/	/	/
	8	Box-Corner	53°28'20"N 1°00'34"W	/	/	/
Runway Altitude: 71	9	Variable 2	20	/	/	/
	10	Target	/	1	1	2

Scenario variation 1d is dynamic: at the beginning of the mission, the three Objectives are given, Entity 1 has its original position and Entity 3 is not present. At time 150 sec, thus soon after the take-off sequence and while performing Action 6 (travelling towards Objective 3), Entity 3 is added to the Entity list and the position of Entity 1 is updated. This results in the generation of an updated flight plan. The execution priority property of Objective 3 is particularly important in determining the behaviour of the MMA, and will be varied in order to show the different courses of action that are possible.

5.3.2 Scenario 2, variations d and e

The original scenario 2 (described in Section 4.3.2) is a scenario that was mainly intended to test the search pattern algorithms. Additionally, it was used for early tests of the Planner algorithms that are meant to ensure that time priorities are respected. Three scenario variations (2a, 2b and 2c) were introduced, among these variations 2a and 2c were used during the testing campaign.

Two new scenario variations, called variations 2d and 2e, have been introduced for the MMA testing campaign. Variation 2d is aimed at testing one of the possible solutions to fuel problems that can be implemented by the MMA; both the *search-reduction-box* and *search-reduction-circle* algorithms (see Section 5.2.4) will be tested. Variation 2e is instead aimed at testing the *add-objectives* algorithms (see Section 5.2.6).

Table 5.9. Details of Entity 3

ID	Property	Entity 3
1	Entity Type	4
2	Entity Tag	3
3	Entity Position	53°30'12''N 1°18'21''W
4	Movement Info	0, 0
5	Entity Behaviour	2
6	Threat Level	3
7	Area of Effect	3000
8	Stance	/

Figure 5.7 shows the updated map for scenario 2; comparing it to Figure 4.12 in Chapter 4, the new Entities that are used in variation 2e can be noted. For both variation 2d and variation 2e, the Objectives are set as in scenario variation 2b; details are provided in Table 5.10.

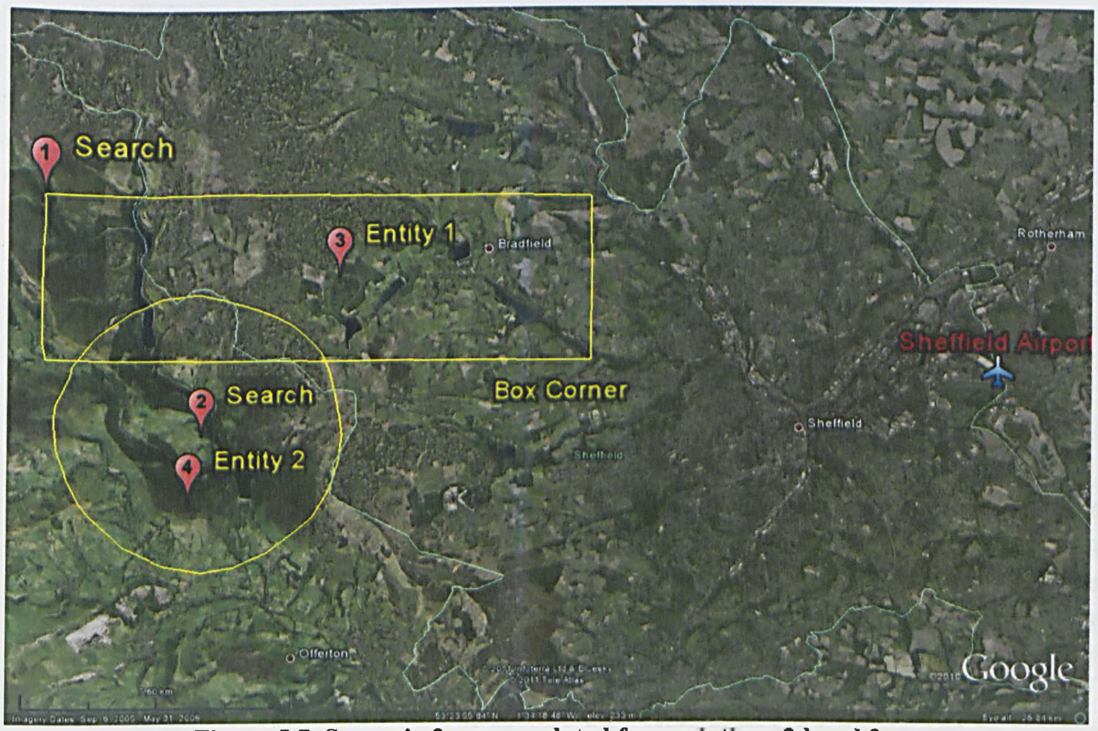


Figure 5.7. Scenario 2 map, updated for variations 2d and 2e

Variation 2d is static; the amount of on-board fuel is modified in order to trigger the *search-reduction-box* and *search-reduction-circle* operators. This amount has to be carefully tailored, since SAMMS will first try to address a fuel issue by using the *reduce-speed* algorithm in the Planner (see Section 4.2.10). If this is unsuccessful, the

Table 5.10. Details of Scenario 2 locations and Objectives

Starting	ID	Property	Search 1	Search 2
Park: 53°23'42"N 1°22'51"W	1	Type	4	4
	2	ID	1	2
	3	Position	53°26'36"N 1°47'07"W	53°22'55"N 1°43'12"W
Runway: 53°23'37"N 1°22'48"W	4	Priority	3500, 5	0, 5
	5	Duty	/	/
	6	Area Type	1	2
Runway Heading: 280	7	Accuracy	1200	700
	8	Box-Corner	53°24'02"N 1°33'14"W	/
Runway Altitude: 71	9	Radius	/	4000
	10	Target	/	/

MMA will intervene; the *check-fuel* operator in the *modify-objectives* state will fire, and the solve-*fuel* substate will opt for changing the search accuracy values of the Search Objectives. However, should the on-board fuel be reduced by an excessive amount, the change would not be sufficient and then an Objective would be removed.

Table 5.11. Details of Entities 1 and 2 in for scenario variation 2e

ID	Property	Entity 1	Entity 2
1	Entity Type	2	2
2	Entity Tag	1	2
3	Entity Position	53°25'21"N 1°39'41"W	53°21'55"N 1°43'28"W
4	Movement Info	0, 0	0, 0
5	Entity Behaviour	2	2
6	Threat Level	1	1
7	Area of Effect	0	0
8	Stance	/	/

Variation 2e is dynamic; the mission starts with the usual two Objectives, however Objective 1 is set as a Search-and-Analyze Objective, while Objective 2 is set as a Search-and-Attack Objective. While performing Objective 1, Entity 1 is added to the Entity list and consequently a new Analyze Objective will be added (Objective 3). Then, while performing Objective 2, Entity 2 is added to the Entity list and consequently a new Attack Objective will be added (Objective 4). The details regarding Entities 3 and 4 are provided in Table 5.11.

5.3.3 Scenario 3, variations f, g and h

The original scenario 3 (described in Section 4.3.3) is a scenario of medium complexity mostly aimed at testing the behaviour of the Planner in situations where several Objectives present a time priority. During the Planner testing campaign, some variations of scenario 3 were introduced in order to test specific algorithms (the *mission-path-adjust* and *reduce-speed* algorithms, see Sections 4.2.8 and 4.2.10), resulting in a total of five scenario variations.

During the MMA testing campaign, scenario 3 will be used to test the *check-priority* operator of the *check-objective* state and one of the possible solutions to fuel problems (the cancellation of an Objective). Three separate variations will be identified.

Figure 5.8 shows the positions of the Objectives for scenario 3; this is unchanged from the map in Figure 4.13 in Chapter 4. In all new variations, the Objectives will be the ones corresponding to scenario variation 3a; details are reproduced in Table 5.12. In order to shows the various possible outcomes, the amount of on-board fuel will be changed.

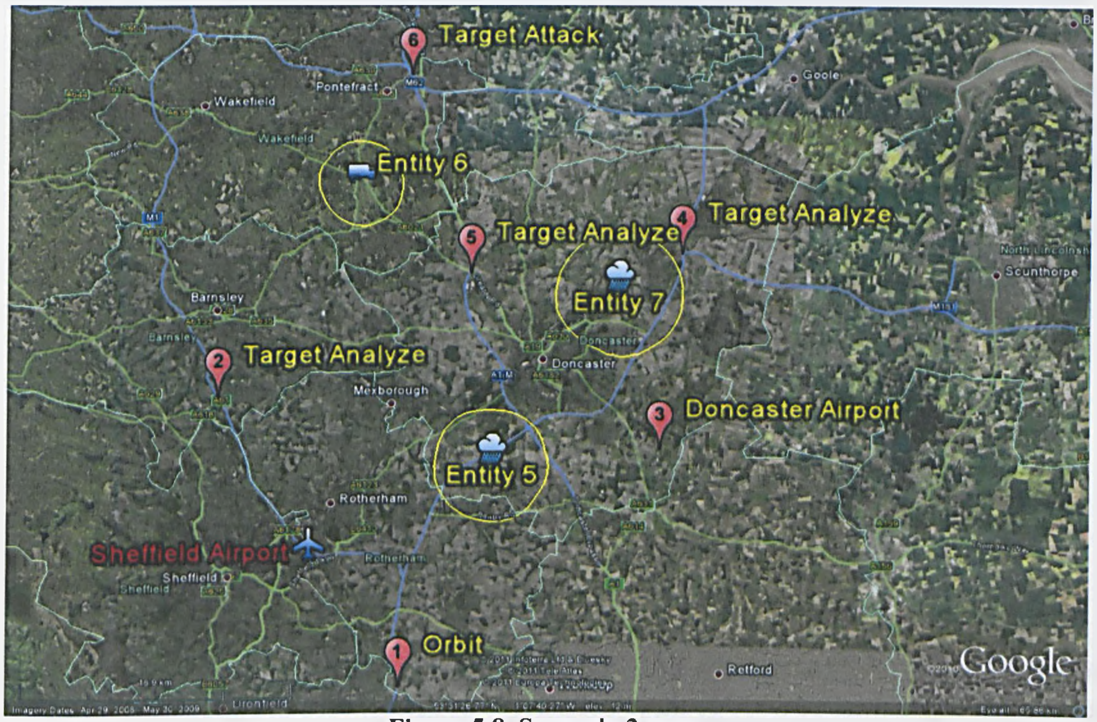


Figure 5.8. Scenario 3 map

In scenario variation 3f, the amount of fuel is set to a value that will trigger the *reduce-speed* algorithm in the Planner, which will be sufficient to ensure completion of the mission. Therefore the MMA will not intervene in this sense, however due to the reduction in flight speed the time priority for Objective 5 will not be respected and therefore the *check-priority* operator will fire in the *check-objective* substate.

In scenario variation 3g, the amount of fuel is set to a value that will trigger the cancellation of an Objective because of the lack of fuel; Objectives all have the same execution priority, so the farthest Objective from the starting airport will be cancelled.

Table 5.12. Details of Scenario 3 Objectives

ID	Property	Orbit	Analyze	Analyze	Analyze	Attack
1	Type	3	1	1	1	2
2	ID	1	2	4	5	6
3	Position	53°18'52"N 1°16'56"W	53°29'54"N 1°28'43"W	53°35'29"N 0°59'14"W	53°34'40"N 1°12'41"W	53°42'15"N 1°16'41"W
4	Priority	-1, 5	0 , 5	2500, 5	1500, 5	0, 5 (4)
5	Duty	1	/	/	/	/
6	Area Type	/	/	/	/	/
7	Altitude	800	/	/	/	/
8	Box-Corner	/	/	/	/	/
9	Time	600	/	/	/	/
10	Target	/	1	2	3	4

In scenario variation 3h, the amount of fuel is set to a value that will trigger the cancellation of an Objective because of the lack of fuel; the execution priority of Objective 6 is lowered, thus causing it to be cancelled.

5.3.4 Scenario 4, variation d

The original scenario 4 (described in Section 4.3.4) is a highly complex scenario designed to test the Planner in situations where a large number of Objectives are

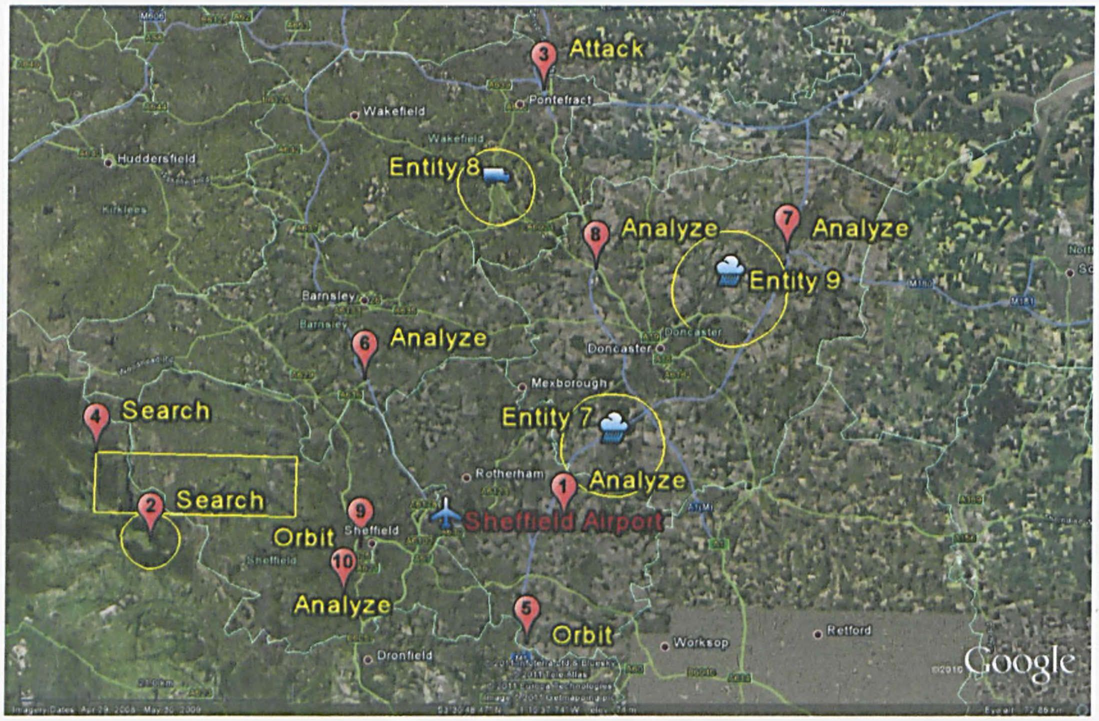


Figure 5.9. Scenario 4 map

present. Dangerous Entities are also introduced to test the behaviour of the mission-path-adjust algorithm. A total of three scenario variations were introduced.

Table 5.13. Details of Scenario 4 Objectives

ID	Property	Analyze	Search	Attack	Search
1	Type	1	4	2	4
2	ID	1	2	3	4
3	Position	53°25'05"N 1°14'33"W	53°22'55"N 1°43'12"W	53°42'15"N 1°16'41"W	53°26'36"N 1°47'07"W
4	Priority	0, 5	0, 5	0, 5	0, 5
5	Duty	/	/	/	/
6	Area Type	/	2	/	1
7	Variable 1	/	1000	/	2500
8	Box-Corner	/	/	/	53°24'02"N 1°33'14"W
9	Variable 2	/	2250	/	/
10	Target	1	/	5	/

ID	Property	Orbit	Analyze	Analyze	Analyze
1	Type	3	1	1	1
2	ID	5	6	7	8
3	Position	53°18'52"N 1°16'56"W	53°29'54"N 1°28'43"W	53°35'29"N 0°59'14"W	53°34'40"N 1°12'41"W
4	Priority	0, 5	0, 5	0, 5	0, 5
5	Duty	1	/	/	/
6	Area Type	/	/	/	/
7	Variable 1	500	/	/	/
8	Box-Corner	/	/	/	/
9	Variable 2	1000	/	/	/
10	Target	/	2	3	4

During the MMA testing campaign, scenario variation 4d will be used to test the *check-payload* operator of the *check-objective* state. The inability to perform a specific Objective type will be introduced and consequently Objectives of that type will be cancelled. The scenario map and Objective details are provided in Figure 5.9 and Table 5.13, which are basically unchanged from Figure 4.14 and Table 4.18 in Chapter 4.

5.4 Results of MMA Tests

As already stated several times, the Mission Manager Agent cannot be tested on its own; more precisely, it would be impractical to do so, since the full set of inputs would have to be simulated. As was shown in Section 5.1.1, the MMA needs up to 2158 input variables, including a full flight plan as developed by the Planner Agent.

However, the MMA operates in a very similar manner to the Planner; most importantly, while some feedback from the Execution Agent is needed, it is not strictly necessary to operate the whole SAMMS architecture in order to test the MMA.

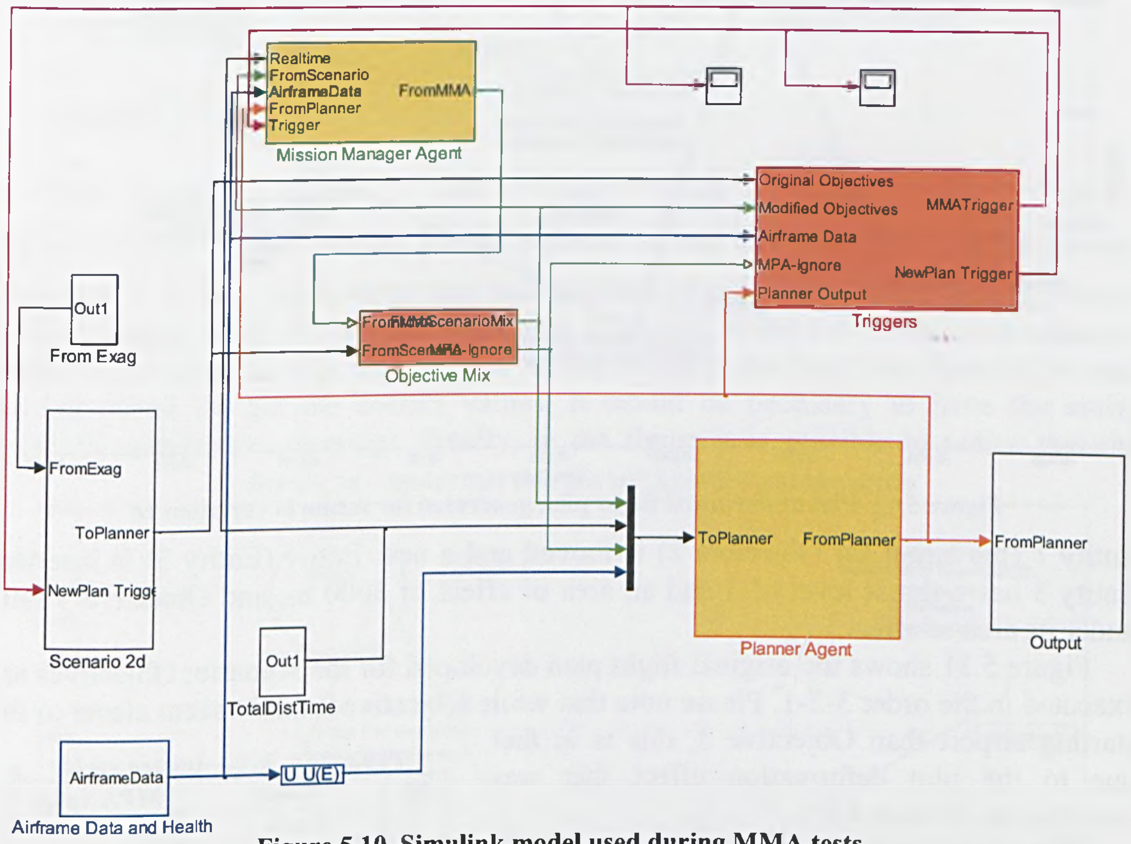


Figure 5.10. Simulink model used during MMA tests

It was then decided to perform MMA tests using a subset of the entire SAMMS architecture. In the simulations that will be shown in this section, the Planner and the MMA are fully integrated; the Objective Mix and New Plan Trigger function are used, however the UAV and autopilot models are excluded as well as the Execution Agent. Figure 5.10 shows the Simulink block diagram that was used to conduct these tests; the diagram can be compared with Figure 3.4 in Chapter 3, which shows instead the block diagram for the entire SAMMS architecture.

The output of the MMA is mainly constituted by the set of modified Objectives; rather than showing them using tables, the results will be shown graphically by showing how the flight plans developed by the Planner change as a consequence of the intervention of the MMA. The same plotting function used during Planner tests, called

Plan_Descripter, will be used; thus, the plots adopt the same conventions that were used for the Planner (especially regarding the Action labels, see Section 4.4). In most cases, the effect of MMA intervention will be shown by comparing original flight plans with the ones that are generated after the intervention. An exception to this is represented by the MPA-Ignore output, which will instead be directly listed.

5.4.1 Results of scenario variation 1d

In scenario variation 1d, the UAV starts the mission with 3 Objectives: a Transit Objective and two Analyze Objectives. Shortly after the take-off sequence is completed and SAMMS has entered main-mission mode, two changes are applied to the Entity list;

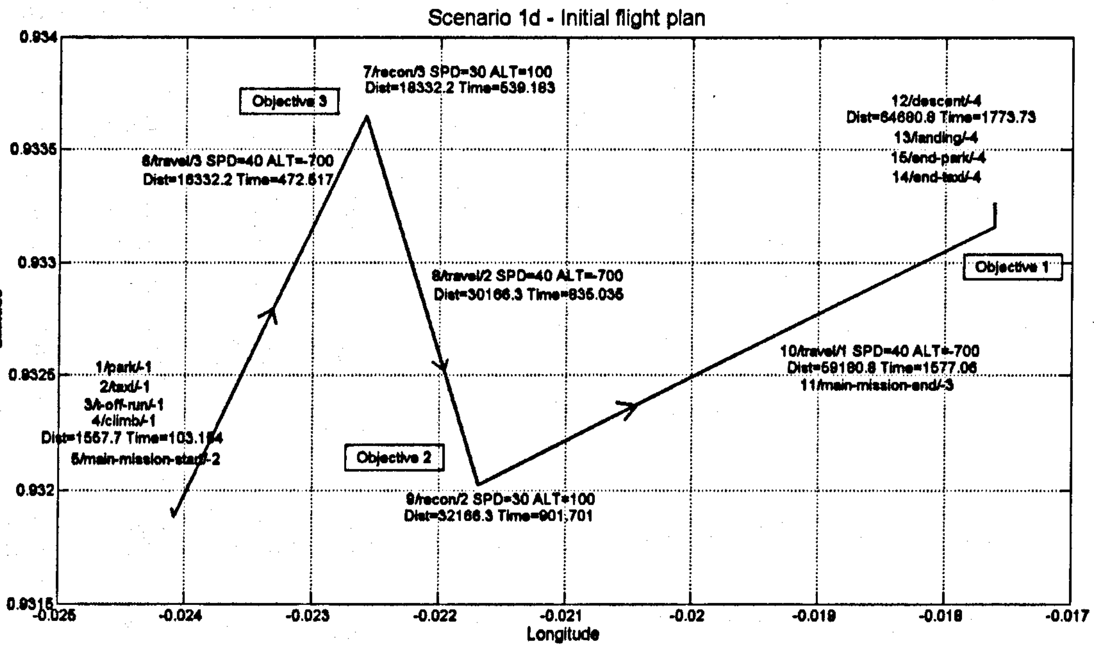


Figure 5.11. Plot of the initial flight plan generated for scenario variation 1d

Entity 1 (the target for Objective 2) is moved and a new Entity (Entity 3) is inserted. Entity 3 has a threat level of 3 and an area of effect of 3000 m, and Objective 3 falls under its area of effect.

Figure 5.11 shows the original flight plan developed for the scenario; Objectives are executed in the order 3-2-1. Please note that while Objective 2 might seem closer to the starting airport than Objective 3, this is in fact due to the plot deformation effect that was described in Section 4.4.

When the changes to the Entity list are made, the updated flight plans will vary depending on the execution priority of Objective 3. Figure 5.12 shows the flight plan obtained by setting an execution priority of 3 or higher for Objective 3. In this case, Objective 3 is not cancelled, since its execution priority level is not lower than the threat level of Entity 3. Instead, the MPA-Ignore state computes that the *mission-path-adjust* algorithm is to be ignored for Entity 3; Table 5.14 shows the *mpa-ignore* output vector, where the value 1 related to Entity 3 can be highlighted. Since the re-planning event occurs shortly after the beginning of the mission, the Execution Agent is not committed

Table 5.14. MPA-Ignore vector

ID	Entity	MPA value
1	Entity 1	0
2	Entity 2	0
3	Entity 3	1
4	Entity 4	0
5	Entity 5	0
6	Entity 6	0
7	Entity 7	0
8	Entity 8	0
9	Entity 9	0
10	Entity 10	0

to finish the current Objective (Objective 3). Thus, the original Travel Action towards the Objective (Action 6) is aborted, however, the new plan in fact still expects the UAV to head towards Objective 3 so the UAV will not divert from its course towards

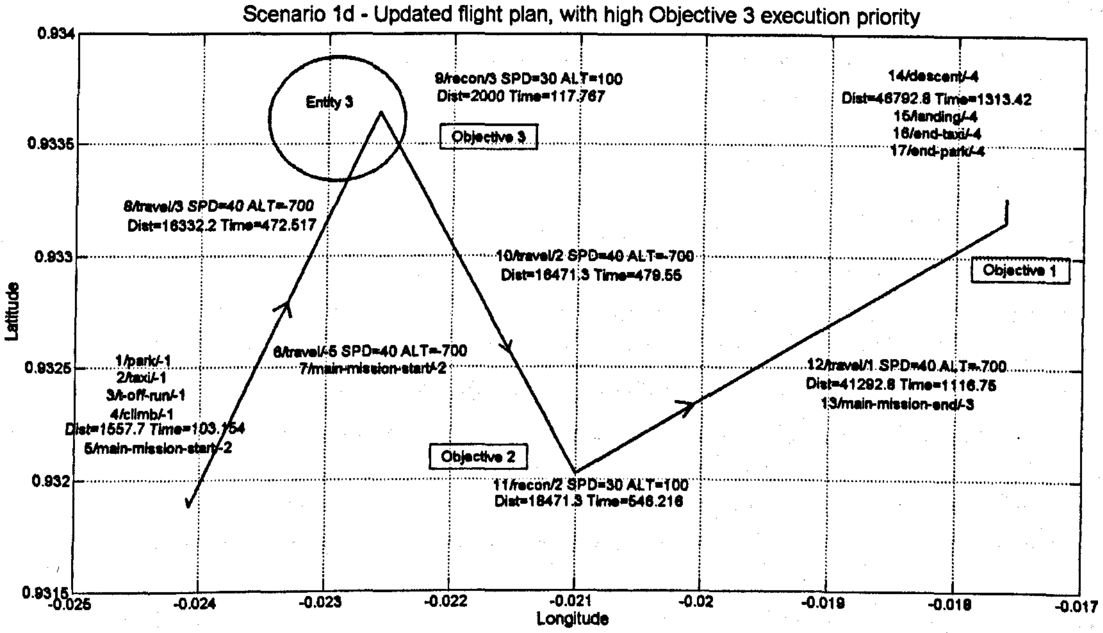


Figure 5.12. Updated flight plan for scenario variation 1d, with high Objective 3 execution priority

Objective 3. In fact, the distance and time estimates for the new Travel Action (Action 8) are the same as for the original flight plan. However, in the test the distance and time estimates are reset starting from Action 9; this is due to the way these simulation tests are conducted (to get the correct values, it would be necessary to have the entire SAMMS architecture running). Finally, in the figure it is possible to notice that the

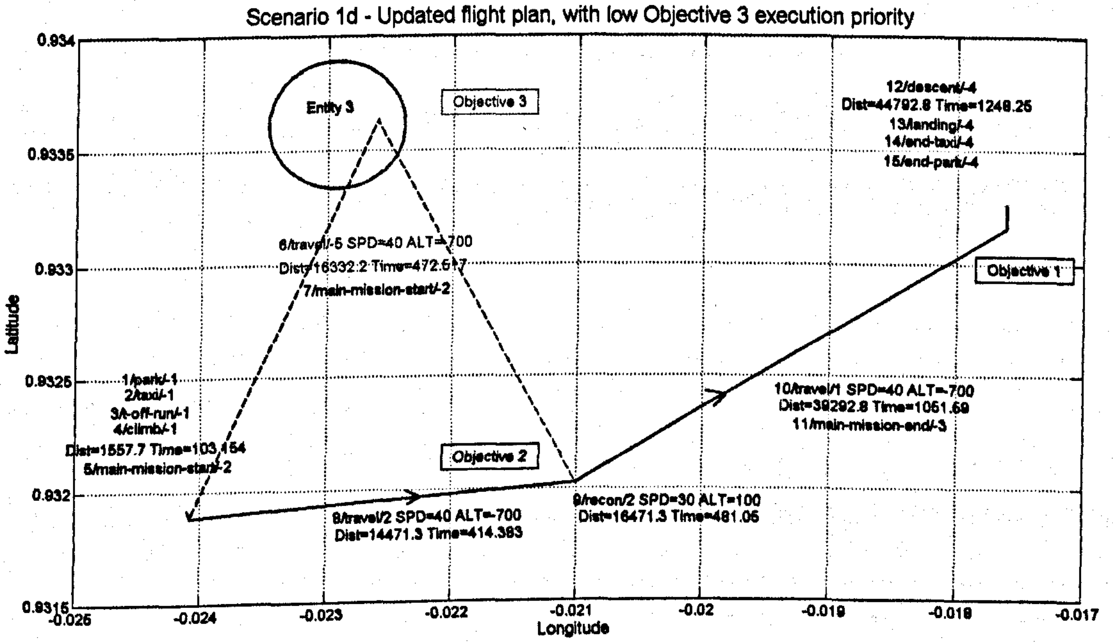


Figure 5.13. Updated flight plan for scenario variation 1d, with low Objective 3 execution priority

check-target-position operator (see Section 5.2.5) has correctly performed its function by automatically updating Objective 2 with the newly available Entity position.

Figure 5.13 shows instead the flight plan which is obtained by setting Objective 3 with an execution priority lower than 3. In this case, the *check-threats* operator (see

Section 5.2.5) decides correctly to cancel Objective 3 because of the newly detected threat. This time, Action 6 will be aborted and the UAV will divert directly towards Objective 2. In the figure it is possible to see as a dashed line the part of the plan that is deleted because of this change. Distance and time estimates are obviously completely different from the original ones. Finally, it is possible to notice that the *check-target-position* operator has correctly performed its function; its operation is not influenced by the decision regarding Objective 3.

5.4.2 Results of scenario variation 2d

In scenario variation 2d, the UAV starts the mission with 2 Search Objectives; however, it does not have a sufficient amount of on-board fuel to complete the mission, even after reducing flight speed. Consequently, the *search-reduction-box* and *search-reduction-circle* operators from the *solve-fuel* substate (see Section 5.2.4) are triggered.

Figure 5.14 shows the flight plan that is initially generated by the Planner; the flight plan involves a reduction of flight speed that is caused by four iterations of the *reduce-speed* algorithm (roughly 36%), but this is still not sufficient to ensure that the mission will be completed: the available amount of fuel is 18 kg, and the total fuel consumption estimate is 24.69 kg.

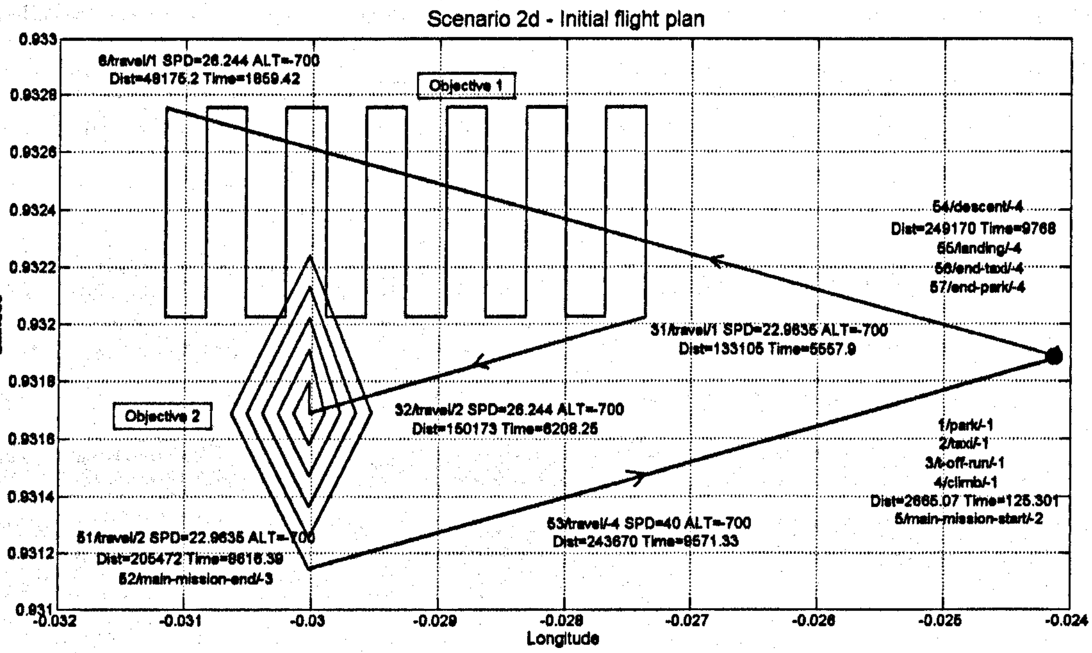


Figure 5.14. Plot of the initial flight plan generated for scenario variation 2d

The *check-fuel* operator in the *modify-objectives* state detects this and triggers the execution of the *solve-fuel* substate; since Search Objectives are present, a reduction of the distance covered performing these Objectives is preferred over the cancellation of an Objective. Objective 1 is chosen for the first reduction attempt, since it is the Objective during which the longest distance is covered. The change is actuated by the *search-reduction-box* operator. The resulting updated flight plan is shown in Figure 5.15. The Search Accuracy property for Objective 1 has been modified, from the original 1200 m to the new value of 2548.65 m; while the original search pattern involved 13 vertical legs, the search pattern is now flown using 7 vertical legs. The Planner still tries unsuccessfully to reduce flight speed so that the mission can be completed, as can be noted by the commanded flight speeds; however the expected fuel consumption is still too high, with a value of 22.81 kg.

The MMA will also check the second generated flight plan; the *check-fuel* operator and the *solve-fuel* substate will be again triggered, however this time it is Objective 2 that will be modified. The change is actuated by the *search-reduction-circle* operator.

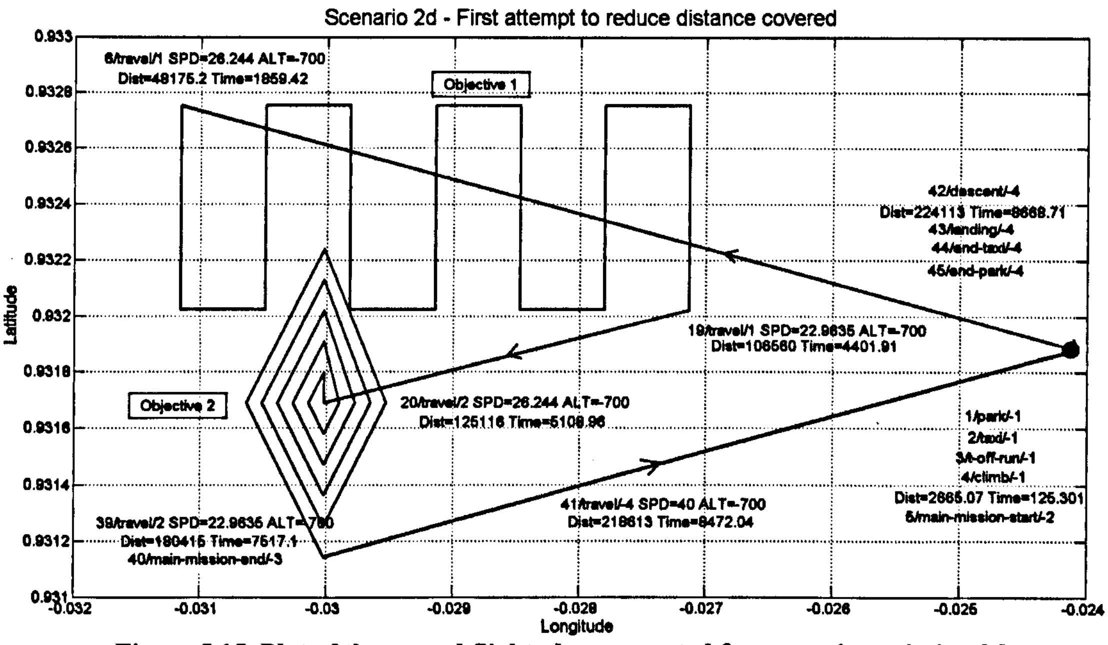


Figure 5.15. Plot of the second flight plan generated for scenario variation 2d

The resulting updated flight plan is shown in Figure 5.16. The Search Accuracy property for Objective 2 has been modified, from the original 700 m to the new value of 1337.77 m; the updated search pattern includes 7 legs instead of the 19 legs of the original one. A speed reduction is still needed in order to complete the mission, but this time the *reduce-speed* algorithm is iterated only two times; the total fuel consumption estimate is now 17.27 kg.

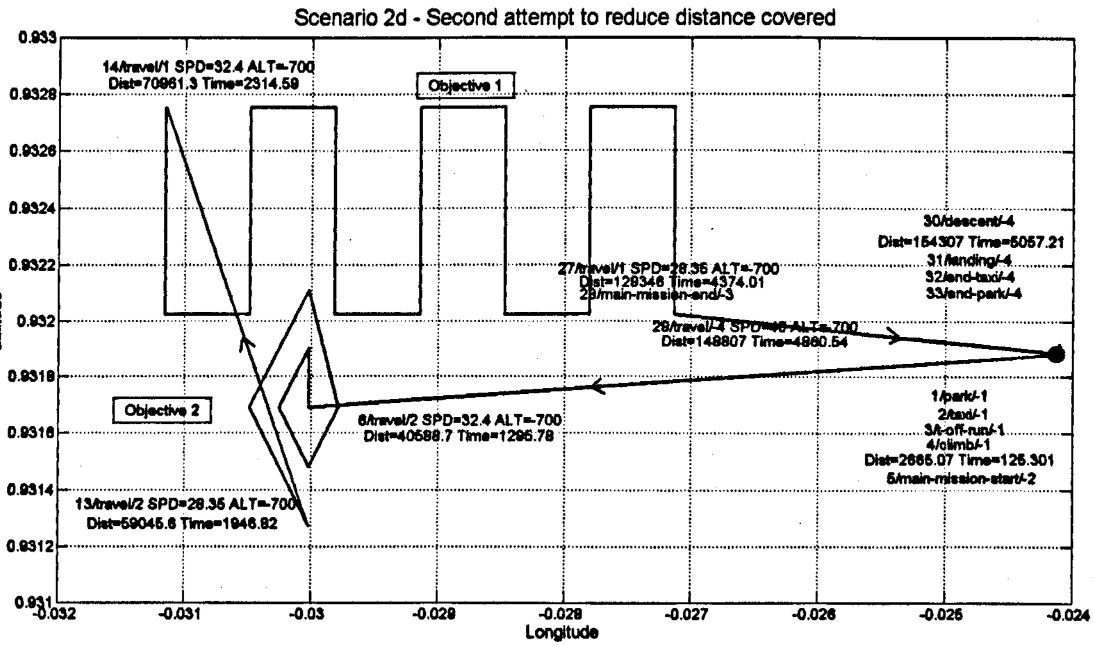


Figure 5.16. Plot of the third flight plan generated for scenario variation 2d

It can also be noted that the order of execution of the Objectives is swapped; because of the reduction in covered distance, time estimates now show that it will be possible to complete Objective 1 within the time priority even if Objective 2 is executed before it.

This can be considered an example of how the flight plan generation process within SAMMS is very dynamic and will adapt to constantly changing conditions. Naturally, the order change might be avoided by setting a different time priority value for Objective 1.

5.4.3 Results of scenario variation 2e

Scenario variation 2e presents the same Objectives as variation 2d, however in this case fuel is not an issue. The main difference is the fact that both Objectives are not simple Searches; Objective 1 is defined as a Search-and-Analyze Objective, while Objective 2 is defined as a Search-and-Attack Objective. The scenario evolves dynamically, showing that the *add-objectives* state (see Section 5.2.6) performs its function by adding new Objectives when Entities are detected within the Search area.

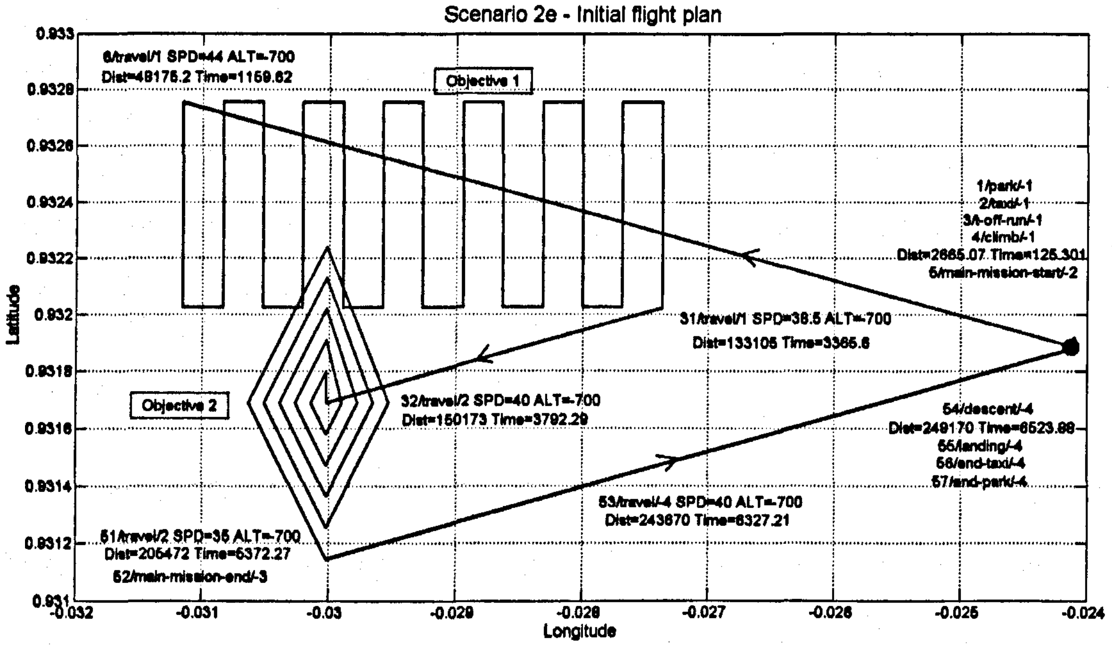


Figure 5.17. Plot of the initial flight plan generated for scenario variation 2e

Figure 5.17 shows the flight plan that is initially generated by the Planner; this may seem very similar to the original flight plan for scenario variation 2d, however the absence of fuel issues results in a very different speed profile. In fact, in order to respect the time priority for Objective 1, a 10% speed increase is planned for Actions that happen before the completion of Objective 1 (resulting from the *increase-speed* algorithm in the Planner, see Section 4.2.10).

A first re-planning event is triggered when Entity 1 is inserted into the Entity list (in more realistic terms, when Entity 1 is detected). Figure 5.18 shows the updated flight plan. The Execution Agent is committed to complete Objective 1, so the first part of the plan is not modified (it is possible to notice that the distance and time estimates do not vary).

Table 5.15. Details of Objectives created by the MMA

ID	Property	Objective 3	Objective 4
1	Type	1	2
2	ID	3	4
3	Position	53°25'21"N 1°39'41"W	53°21'55"N 1°43'28"W
4	Priority	0, 5	0, 5
5	Duty	/	/
6	Area Type	/	/
7	Accuracy	/	/
8	Box-Corner	/	/
9	Radius	/	/
10	Target	1	2

Because of the nature of the simulation tests, distance and time estimates are reset after the second main-mission-start Action (the one added to signal the beginning of the updated flight plan). The MMA has created a new Objective (to be more precise, it is the *add-analyze-box* operator that fires), which is assigned the ID tag value of 3. Details

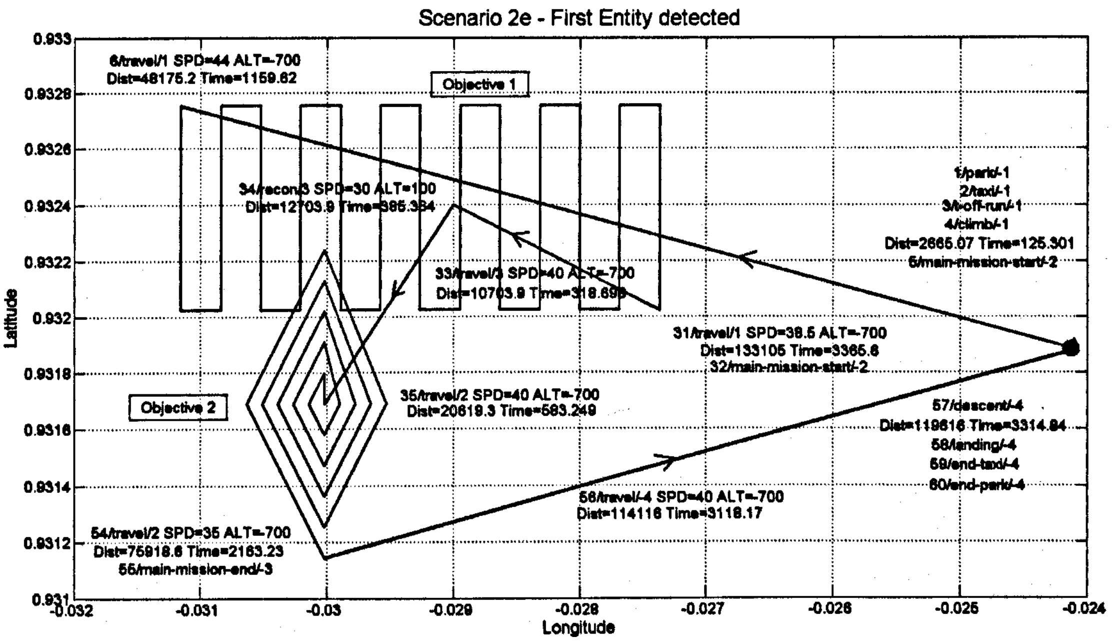


Figure 5.18. Plot of the first updated flight plan for scenario variation 2e

regarding the newly created Objective 3 are shown in Table 5.15; since Objective 1 is defined as a Search-and-Analyze Objective, Objective 3 is of the Target-Analyze type. The new Objective 3 is closer to the final position reached during Objective 1, so the Planner decides to execute it before Objective 2.

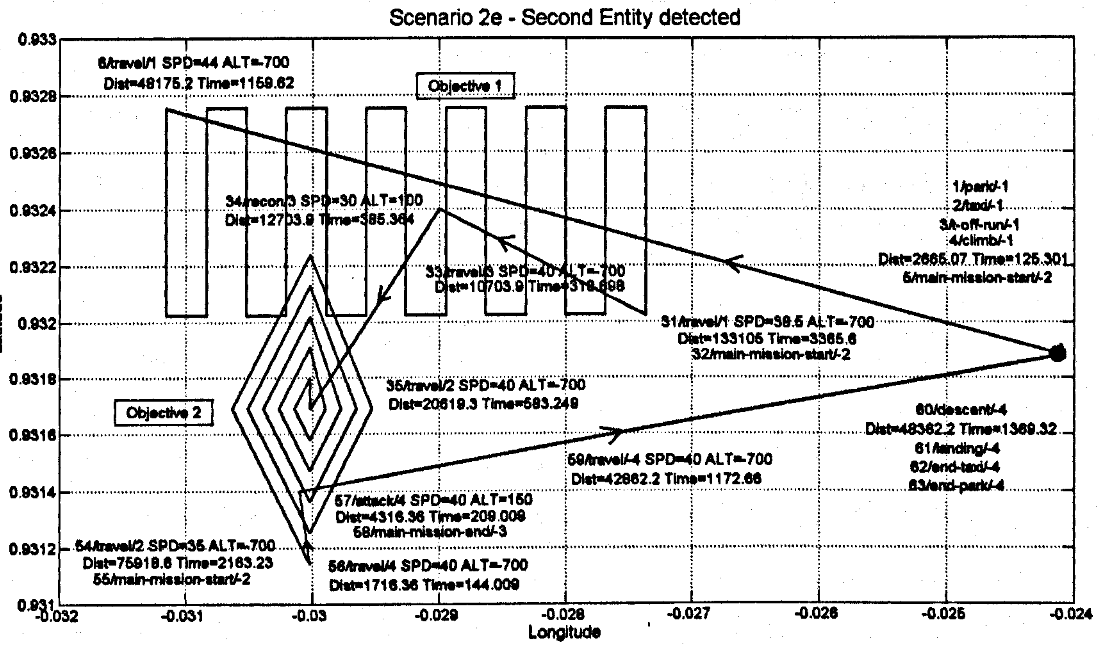


Figure 5.19. Plot of the second updated flight plan for scenario variation 2e

The second re-planning event is triggered when Entity 2 is detected. Figure 5.19 shows the updated flight plan. Again, the Execution Agent is committed to complete Objective 2, so the plan is not modified until the completion of Objective 2. Just as before, a new Objective (Objective 4) is added (to be more precise, it is the *add-attack-*

circle operator that fires); unlike Objective 3, this is a Target-Attack Objective. Details regarding the newly added Objective are also shown in Table 5.15. Since the Exag is committed to complete Objective 2, the updated flight plan only involves Objective 4.

5.4.4 Results of scenario variations 3f, 3g and 3h

Scenario variation 3f is in fact just the same as variation 3e that was used during the Planner testing campaign. In this case, it is intended to demonstrate the functionality of the *check-priority* operator in the *check-objective* substate of the Mission Manager Agent.

Six Objectives are defined and the amount of fuel available to the UAV is set at 24 kg. This causes the *reduce-speed* operator in the Planner (see Section 4.2.10) to reduce flight speed by 19%, in order to save enough fuel to complete the mission. The resulting flight plan is shown in Figure 5.20, where it is possible to see that the time estimate for the completion of Objective 5 is 1622.36 sec. Objective 5 has a time priority value of 1500 sec; while using the normal flight speeds the priority would be respected (from Figure 4.23 in Chapter 4 it is possible to see that the time estimate would be 1428.11 sec), with the reduced flight speeds it is not. Since the lack of fuel problem is obviously considered more important, SAMMS accepts the fact that the time priority for Objective 5 will not be respected. This is formalized by the MMA; whilst the *check-objective* substate for Objective 5 is active, the *check-priority* operator fires and modifies the time priority value for Objective 5. In the scenario, Objective 5 has its time priority modified from the value of 1500 sec to the value of 1682.36 sec (obtained as its corresponding time estimate, 1622.36 sec, increased by a fixed amount of 60 sec). The user interface should be designed so that this change will be communicated to the UAV operator.

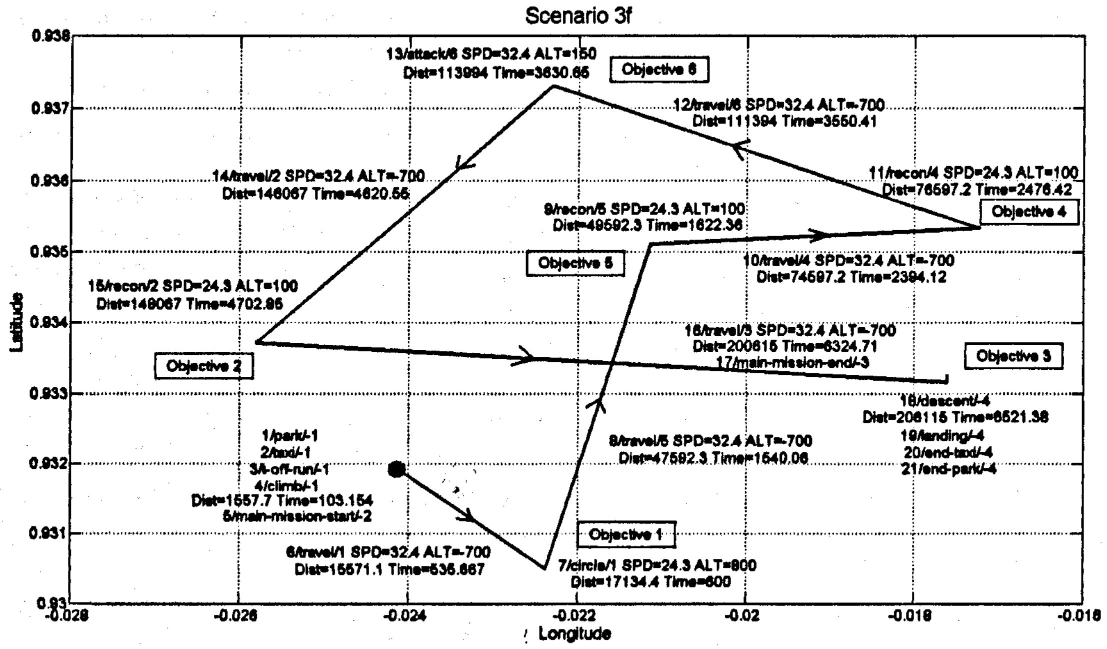


Figure 5.20. Plot of the flight plan generated for scenario variation 3f

Scenario variation 3g presents the same Objectives, but involves an even more limited amount of available fuel. Having 18 kg of fuel at its disposal, the UAV cannot complete the mission even after several iterations of the *reduce-speed* algorithm. Consequently, the *check-fuel* operator in the *modify-objectives* state activates the *solve-fuel* substate; since no Search Objectives are present, the decision to cancel an Objective is made. All Objectives have the same execution priority, thus the cancelled Objective is

Objective 4, which is the farthest from the starting airport. The resulting flight plan is shown in Figure 5.21, where dashed lines represent the cancelled Travel Actions from the original flight plan. In order to complete the mission, the Planner still needs to

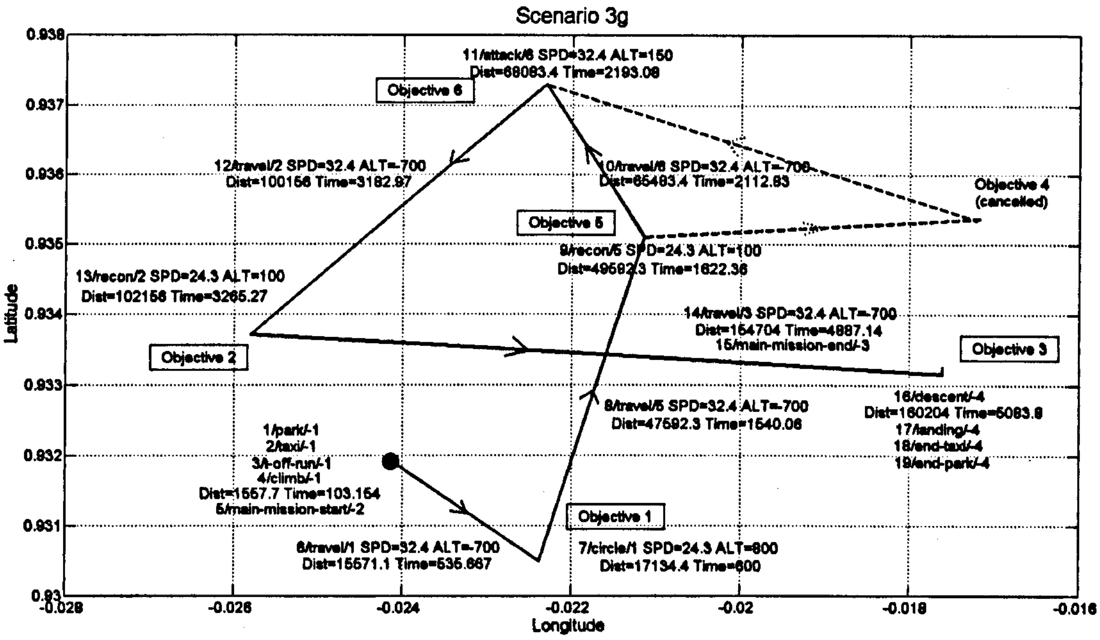


Figure 5.21. Plot of the flight plan generated for scenario variation 3g

reduce speed by 19%; the total estimated fuel consumption is now 17.86 kg. Note that the *priority-check* operator fires again for Objective 5, since its time priority is again not respected.

Scenario variation 3h presents a very similar situation, however in this case the execution priority for Objective 6 is changed from a value of 5 to a value of 4. The

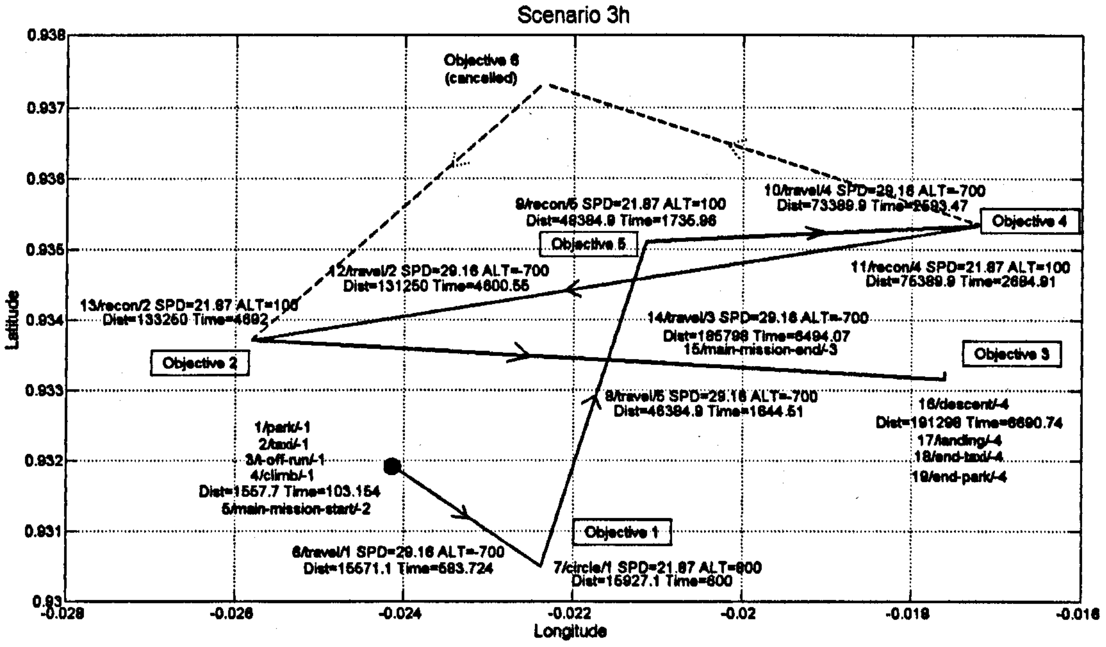


Figure 5.22. Plot of the flight plan generated for scenario variation 3h

amount of fuel is set to 19.2 kg. As for variation 3g, the *solve-fuel* substate is entered, but in this case Objective 6 is chosen to be cancelled since it has a lower execution priority. The resulting flight plan is shown in Figure 5.22, where dashed lines represent the cancelled Travel Actions from the original flight plan. A reduction of flight speed by 27% (three iterations of the *reduce-speed* algorithm) is needed to ensure that the plan will be completed with the current amount of fuel; with these speed values, the fuel consumption estimate is 18.65 kg. The *priority-check* operator fires again, this time for both Objective 5 and Objective 4; the new values assigned are 1922.17 sec for Objective 5 and 2976.57 sec for Objective 4.

5.4.5 Results of scenario variation 4d

Scenario variation 4d is based on variation 4a; it uses the same set of Objectives and Entities, however payload failures are introduced, causing the MMA to cancel Objective types that require the specific type of payload. It is meant to test the functionality of the *check-payload* operator in the *check-objective* substate (see Section 5.2.5). During testing, several failure types will be introduced and the corresponding flight plans plotted. Please note that due to the scenario complexity, some of the labels that describe Actions in the flight plan have been omitted, in order to avoid confusion. Entities 7, 8 and 9 are dangerous and will trigger the *mission-path-adjust* algorithm in all of the presented cases; flight routes that would result by excluding the algorithm are plotted using dashed lines.

Figure 5.23 shows the unmodified flight plan that is obtained when no payload failures are introduced; it can be noticed that this is the same as in Figure 4.29 in Chapter, which shows the flight plan for scenario variation 4a, obtained during the Planner testing campaign. Objectives are executed in the order 5-1-7-8-3-6-2-4.

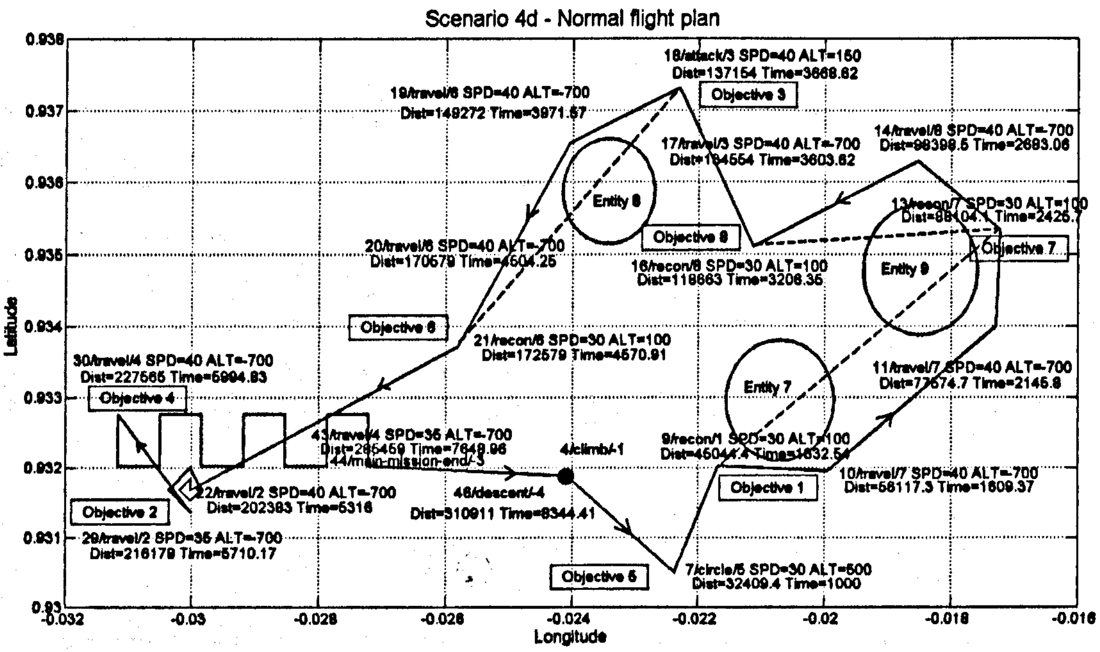


Figure 5.23. Plot of the unmodified flight plan generated for scenario variation 4d

Figure 5.24 shows the flight plan obtained by introducing a failure in the payload which is used to perform Target-Analyze Objectives. Of the eight available Objectives, four are of the Target-Analyze type; these are Objectives 1, 6, 7 and 8. The payload failure causes the *check-payload* operator to be triggered when the *check-objective* substate for each of these Objectives is entered. The Objectives are cancelled and the new resulting flight plan involves only four remaining Objectives; these are executed in the order 5-3-2-4.

Figure 5.25 shows the flight plan obtained by introducing a failure in the payload which is used to perform Target-Attack Objectives. Of the eight available Objectives, only one (Objective 3) is of the Target-Attack type. The *check-payload* operator correctly cancels Objective 3 and the new resulting flight plan involves seven remaining Objectives; these are executed in the order 5-1-8-7-6-2-4.

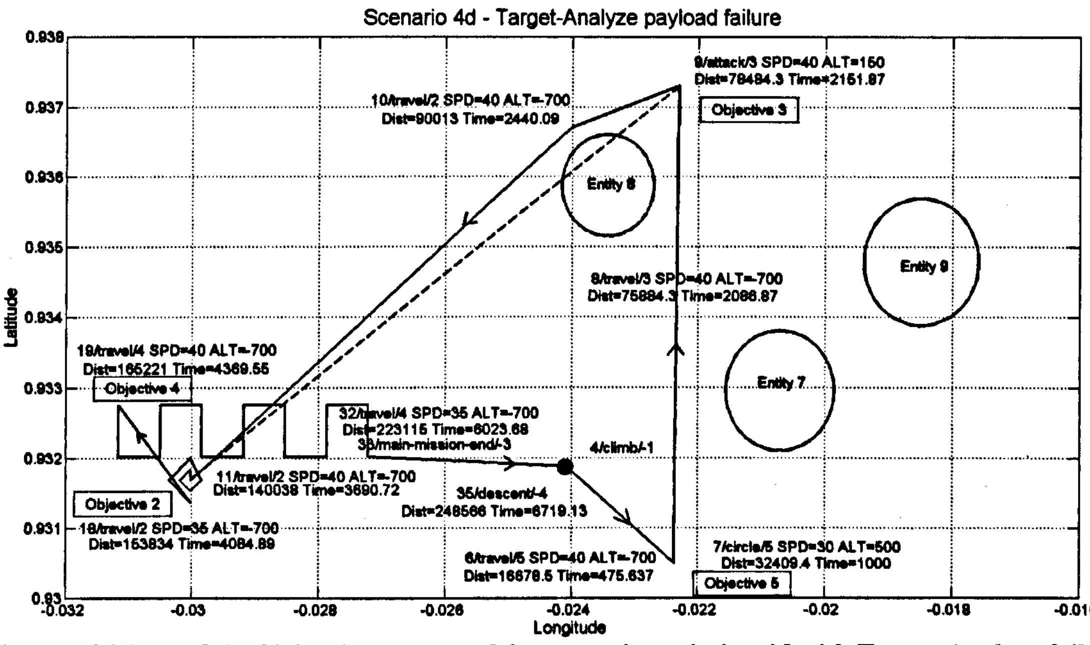


Figure 5.24. Plot of the flight plan generated for scenario variation 4d with Target-Analyze failure

Figure 5.26 shows the flight plan obtained by introducing a failure in the payload which is used to perform Search Objectives. Of the eight available Objectives, two are of the Search type; these are Objectives 2 and 4. The *check-payload* operator correctly cancels Objective 2 and Objective 4 and the new resulting flight plan involves the six remaining Objectives; these are executed in the order 5-1-7-8-3-6.

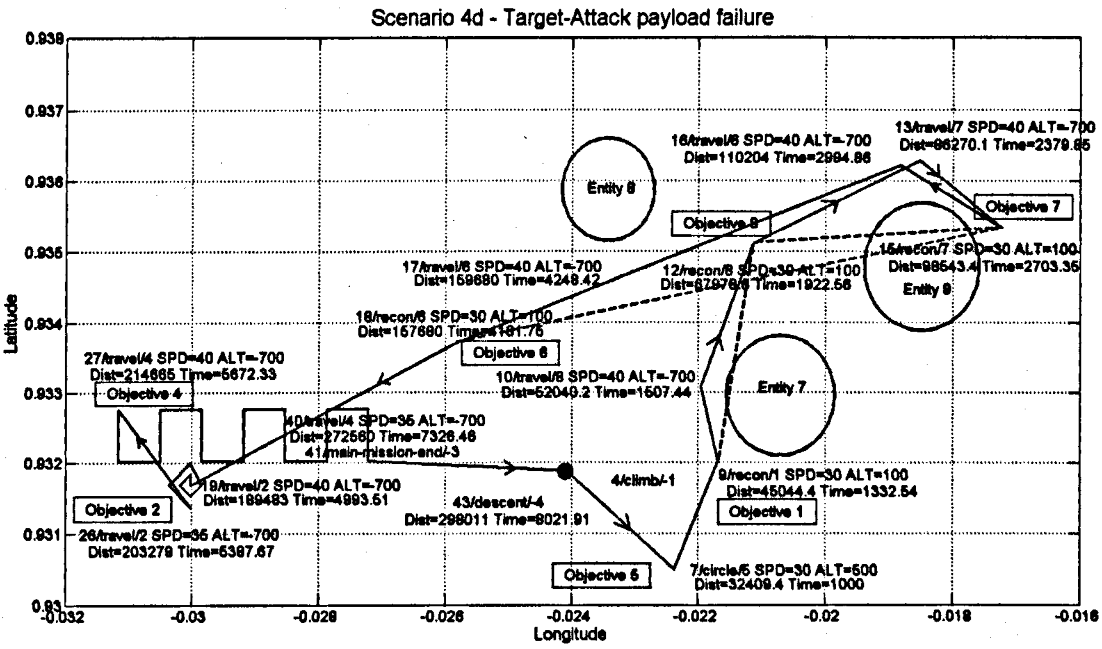


Figure 5.25. Plot of the flight plan generated for scenario variation 4d with Target-Attack failure

Figure 5.27 shows the flight plan obtained by introducing a failure in the payload which is used to perform Duty 1 during Orbit Objectives (unlike other Objective types, Orbit Objectives might use different types of payload while loitering, so related payload failures are separated depending on this). Of the eight available Objectives, only one

(Objective 5) is of the Orbit type; its Duty 1 is set to 1. The *check-payload* operator correctly cancels Objective 5 and the new resulting flight plan involves seven remaining

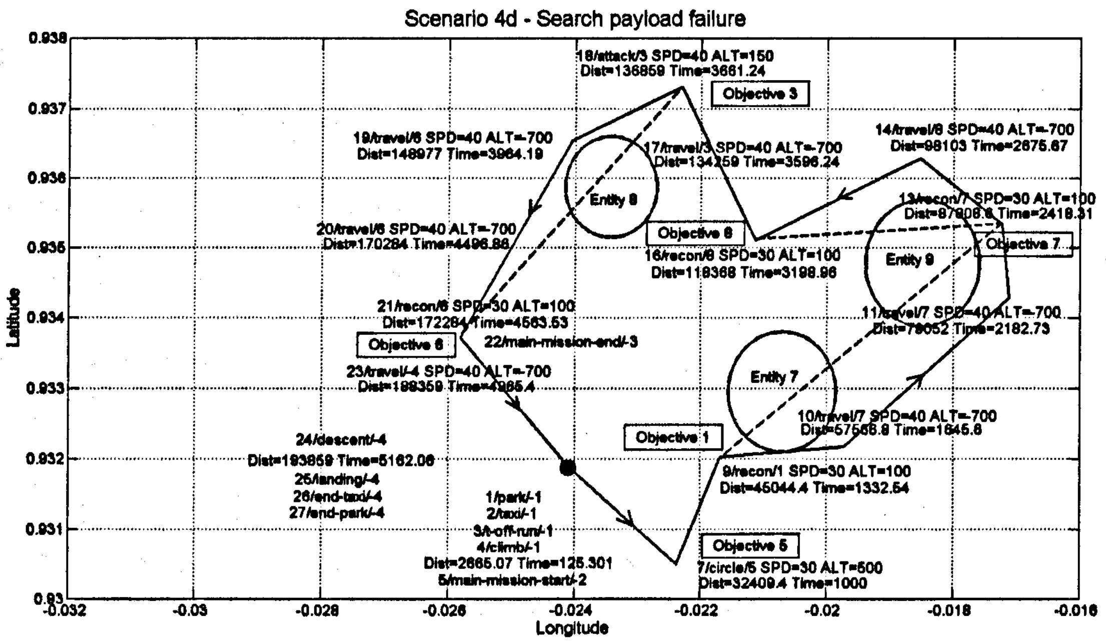


Figure 5.26. Plot of the flight plan generated for scenario variation 4d with Search failure

Objectives; these are executed in the order 1-7-8-3-6-2-4. Had the Duty value for Objective been set to 2, a failure to the payload needed to perform Orbit Duty 1 would

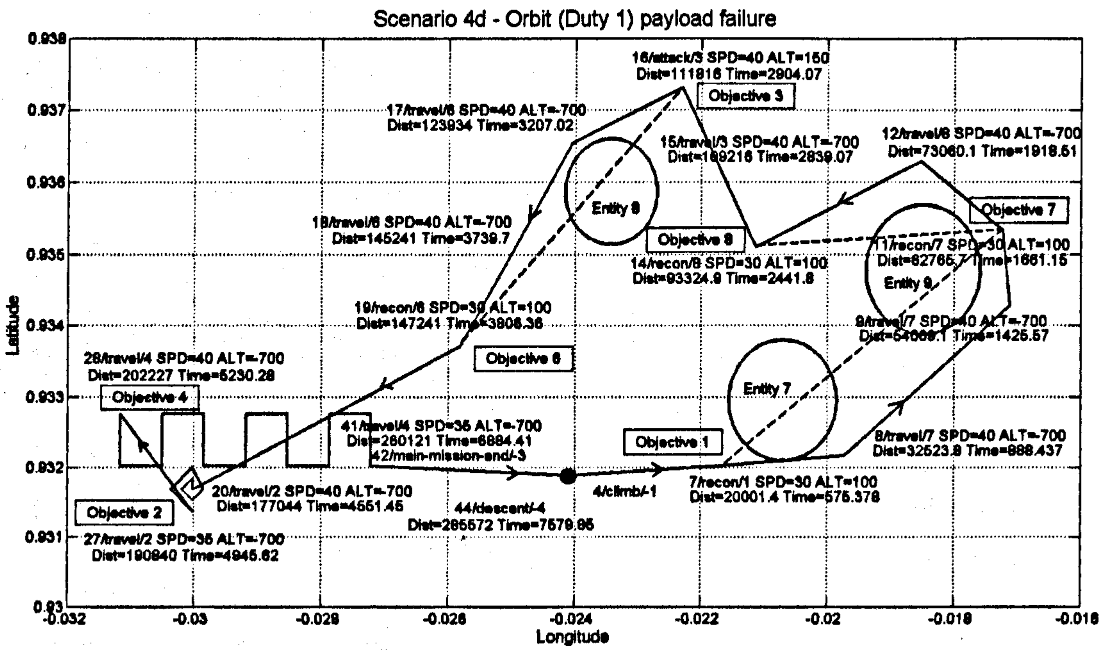


Figure 5.27. Plot of the flight plan generated for scenario variation 4d with Orbit Duty 1 failure

have not influenced it (the MMA would not have cancelled it).

5.5 Concluding remarks

In this chapter, the Mission Manager Agent was thoroughly described and tested. The Mission Manager Agent operates as an overarching layer for the Planner Agent, implementing high-level functionality that allows to dynamically change mission Objectives when the conditions demand it.

The functionality of the Mission Manager Agent could have been embedded directly into the Planner Agent, but it was deliberately separated so as to allow a choice regarding its use. This is because autonomously changing mission Objectives, which is the capability that characterizes the MMA, might not be desirable for certain applications or under particular usage conditions. More generally, it was deemed interesting to study how two separate, but similarly structured, Soar agents could be designed to interact constructively. The Planner Agent can fully operate without the Mission Manager Agent, applying a simple change to the SAMMS Simulink model.

The structure of this Chapter followed closely the structure of Chapter 4. The first part of the chapter focused on the description of the Mission Manager Agent from a theoretical and technical point of view. Having defined the functions it is expected to perform in previous chapters (and particularly Chapter 3), implementation details were outlined here. The Input/Output interface was described first, followed by a thorough description of the algorithms embedded in the agent and of how they are implemented in terms of Soar coding.

The second part of the chapter was dedicated to the practical simulation tests that were executed on the agent in order to validate it. The Mission Manager Agent needs input from the Planner Agent to work, so a dedicated simulation architecture (a subset of the full SAMMS architecture) was developed. Starting from the test scenarios defined in Chapter 4 for the Planner Agent testing campaign, several scenario variations were introduced, each specifically designed to test particular MMA algorithms. Results of MMA operation are demonstrated by showing the effects it has on the flight plans generated by the Planner Agent. The flight plans are displayed using graphical plots that are obtained in the same manner as for the Planner.

The next chapter will be dedicated to the final component of the SAMMS architecture: the Execution Agent. While it is a Soar agent, this is profoundly different from the Planner Agent and the Mission Manager Agent. To test it, the entire SAMMS architecture will be used, since it requires both the Planner Agent and the UAV model to be present to work properly. Thus, while the chapter will be mainly dedicated to the Execution Agent, it will also present the opportunity to test the entire SAMMS architecture as a system, also verifying the integration of its components.

6. The Execution Agent

In Chapter 4 and Chapter 5, the Planner Agent and Mission Manager Agent components of SAMMS were described and tested. Combining the Objectives assigned by a UAV operator and the available situational awareness, these two agents can develop a complex flight plan that will execute a mission, updating it as the mission is in progress. The flight plan is defined as an ordered sequence of Actions, as outlined in Section 3.1.3.

However, an Action is an abstraction that is meaningful from a mission management point of view but cannot be immediately translated into actual UAV actuator commands. To perform this "translation" function, a third agent is needed, acting as a middle layer between the high-level planning functions and the low-level UAV control algorithms (e.g. the autopilots). This third agent is called the Execution Agent (or Exag), since it is the agent that enables execution of a flight plan developed by the Planner.

It is important to highlight that the Exag is functionally very different from the Planner and the MMA. In particular, the timescale for the agents' operation is radically different: while the Planner and the MMA operate asynchronously and do not necessitate immediate response, the Exag must interface continuously with the lower level functions and check sensor data to verify the current situation and make short-term decisions. As a consequence, the Exag is very different from the other agents from a Soar point of view; while the Planner and the MMA shared many Soar structural and coding concepts (as could be seen in Sections 4.2 and 5.2), the Exag presents an entirely different basic structure.

In this chapter, the Execution Agent of the SAMMS architecture will be thoroughly described and tested. As a consequence of the differences just highlighted, the layout for the chapter will be slightly different from the previous two chapters. The chapter will still be divided into two main parts, with the first part focusing on the description of the agent and the second part focusing on tests performed to validate it. The second part in particular will be different, mostly because the nature of simulations is very different. The Exag will be tested together with the entire low-level control function set (the autopilot from Section 3.5), verifying the effect of control inputs on the UAV model described in Section 3.4. Thus the simulations will test the behaviour of the UAV, showing flight trajectories and low-level control input histories. Two main types of simulations can be performed; in the first type, the focus is placed on how the Exag converts Actions into actual commands, showing flight trajectories obtained for each different Action type. In the second type of tests, the Exag is used to execute flight plans developed by the Planner; a selection of the test scenarios described in Chapters 4 and 5 is used to validate functionality not only of the Exag but also of the entire SAMMS architecture. Results of the first type of tests are part of this chapter; results of the second type of tests are instead part of Chapter 7.

This chapter is divided into four sections. In Section 6.1, the Input/Output (I/O) interface for the Exag is described; while this is based on the interface depicted in Section 2.8, details regarding the actual data being transferred are given here. Section 6.2 introduces the MMA from a descriptive point of view; the agent is thoroughly analyzed by looking at the actual Soar code and the way it is organized; differences with the Planner Agent and the MMA are highlighted. Section 6.3 is dedicated to presenting the testing campaign that was performed to validate the Exag. In Section 6.4, flight trajectories obtained from each Action type will be visualized. Flight data is extracted from the scenarios, highlighting critical segments of the flight trajectories. Results from the testing campaign are also presented in Chapter 7, which is introduced in the concluding remarks section of this chapter.

6.1 Exag I/O Interface

Like the Planner Agent and the Mission Manager Agent, the Execution Agent requires a dedicated Soar/Simulink interface, developed on the basis of the one described in Section 2.8. In this section, details regarding this interface will be provided.

The considerations presented in Section 4.1 as an introduction to the description of the Soar/Simulink interface for the Planner Agent are also valid for the Exag. The large amount of data exchanged between the Exag and the SAMMS Simulink architecture is organized into pre-defined structures. As for the Planner Agent (and any Soar agent), the I/O process is managed through two dedicated parts of working memory, the *input-link* and the *output-link*. These structures will now be thoroughly described.

9.1.1 Input

Inputs for the Planner Agent are constituted by 4 different main categories, as in Table 6.1.

Table 6.1. Main level of input-link structure for the Execution Agent.

Code	Name	Values	Description
1	Planner Metadata	2	Data for the take-off and approach phases of the mission
2	Flight Plan	100	All Objectives and their properties
3	Real-time Data	4	All Entities and their properties
4	Flight Parameters	7	Feedback from Execution Agent; Total Distance and Time

The Planner Metadata part of the input structure contains general information produced by the Planner regarding the current flight plan. There are two separate variables: *Number of Actions* is a count of the total number of Actions that form the current flight plan, while *Cycles Valid* indicates the number of Soar cycles over which the current flight plan has been valid (as calculated in the *modify-plan* state, described in Section 4.2). Both variables are represented by integer values.

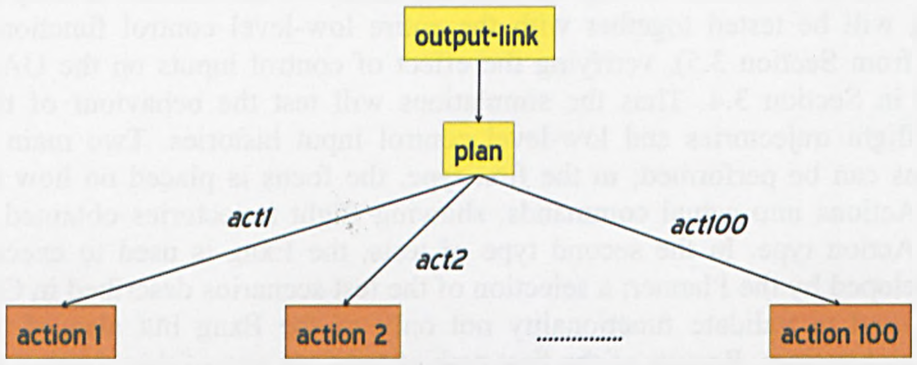


Figure 6.1. Organization of flight plan input of the Execution Agent, coming from the Planner.

The Flight Plan part of the input structure contains the entire current flight plan, as generated by the Planner Agent and first described in Section 4.1.2. Thus, in practice this part of the input-link is structured as in Figure 6.1. While in theory SAMMS should have no limitation in the number of Actions it can handle, a limit of 100 Actions was effected (also as a consequence of the limitation to 10 Objectives). Each Action has its own data structure, just as defined in Section 3.1. Table 6.2 summarizes this, also adding details about data types. With 100 Actions with 13 properties each, this part of the output structure amounts for a maximum of 1300 separate values.

Table 6.2. Action input structure.

Code	Type	Definition	Type
2.x.1	Action Type	As defined in Table 3.6	integer
2.x.2	Sequence	Sequence number for the Action	integer
2.x.3.1	Start Position	Latitude of initial position (used for some Action types)	float
2.x.3.2		Longitude of initial position (used for some Action types)	float
2.x.4.1	Position	Latitude of Action reference position	float
2.x.4.2		Longitude of Action reference position	float
2.x.5	Time	Time property of Action (usually time limit)	float
2.x.6	Heading	Bearing to be kept for certain Action Types	float
2.x.7	Altitude	UAV Altitude specified for Action	float
2.x.8	Duty	Duty type for Circle Actions, otherwise parent Objective type	integer
2.x.9	Speed	UAV Speed for Action	float
2.x.10	Target	Defines a specific target for Recon and Attack	integer
2.x.11	Objective	Parent Objective ID tag	integer

The Real-time Data part of the input structure contains data that is generated by the UAV model at the current stage of development of SAMMS and that would be generated by on-board sensors on an actual UAV. This includes information regarding the aircraft position, attitude and speed. Information is divided into four main types, as can be seen in Table 6.3. For each variable, an example of what type of sensor may provide the information is shown; for example, the Global Positioning System (GPS) or an Inertial Measurement Unit (IMU). The Real-time Data input structure amounts for a total of 11 separate values.

Table 6.3. Real-time Data input structure.

Code	Type	Definition	Type
3.1	Mission Time	Time in seconds since beginning of mission – from clock	float
3.2.1	Position	Latitude coordinate – from GPS or IMU	float
3.2.2		Longitude coordinate – from GPS or IMU	float
3.2.3		Altitude on sea-level – from GPS, IMU or barometer	float
3.2.4		Altitude on ground-level – from radio-altimeter	float
3.2.5		Altitude of ground – from database	float
3.3.1	Attitude	Pitch attitude angle – from gyro	float
3.3.2		Roll attitude angle – from gyro	float
3.3.3		Yaw attitude angle (heading) – from gyro or compass	float
3.4.1	Speed	True Airspeed – derived from IAS	float
3.4.2		Vertical Speed – from IMU	float

The Flight Parameters input structure contains information regarding the aircraft capabilities and status. In fact, this is the entirety of the data that is provided by the Airframe Data and Health block in the Simulink simulation environment (see Section 3.3 and Table 3.9). Table 6.4 displays all variables that are part of this input structure, together with their relative data types. Unlike the Planner Agent, the Exag needs the full set of Airframe Data and Health variables; this is because the information contained includes data which describes the aircraft performance and capabilities, which must be used during generation of low-level commands. The variables are divided into 7 categories and amount for a total of 54 separate values.

Summarising, the input structure for the Execution Agent is constituted by a total of 1367 variables, organized in the manner that was described in this subsection.

Table 6.4. Flight Parameters input structure

Code	Name	Definition	Type
4.1.1	Control Status	Status of pitch control system	float
4.1.2		Status of roll control system	float
4.1.3		Status of yaw control system	float
4.2.1	Speeds	Maximum speed	float
4.2.2		Cruise speed	float
4.2.3		Minimum (stall) speed	float
4.2.4		Orbit speed	float
4.2.5		Take-off rotation speed	float
4.2.6		Landing speed	float
4.2.7		Ground Manoeuvre speed	float
4.2.8		Ground Movement speed	float
4.2.9		Current status	float
4.2.10		Current speed limitation	float
4.2.11		Search speed	float
4.2.12		Analyze speed	float
4.2.13		Attack speed	float
4.3.1	Altitudes	Maximum altitude	float
4.3.2		Cruise altitude	float
4.3.3		Minimum ground altitude	float
4.3.4		Climb turn altitude	float
4.3.5		Descent altitude	float
4.3.6		Pre-land altitude	float
4.3.7		Flare altitude	float
4.3.8		Current altitude limitation	float
4.3.9		Search altitude	float
4.3.10		Analyze altitude	float
4.3.11		Attack altitude	float
4.4.1	Distances and angles	Recon distance	float
4.4.2		Attack distance	float
4.4.3		Circle distance	float
4.4.4		Descent distance	float
4.4.5		Pre-land distance	float
4.4.6		Take-off angle	float
4.4.7		Climb angle	float
4.4.8		Flare angle	float
4.5.1	Failures	Loss of Thrust Control	integer
4.5.2		Structural Damage	integer
4.5.3		Loss of Power	integer
4.5.4		Loss of GS Connection	integer
4.5.5		Loss of GPS	integer
4.5.6		Loss of IMU	integer
4.5.7		Loss of Autopilot	integer
4.6.1	Payload Failures	Failed Analyze Target	integer
4.6.2		Failed Attack Target	integer
4.6.3		Failed Search	integer
4.6.4		Failed Orbit (Duty 1)	integer
4.6.5		Failed Orbit (Duty 2)	integer
4.6.6		Failed Orbit (Duty 3)	integer
4.7.1	Fuel and range	Maximum Fuel	float
4.7.2		Remaining Fuel	float
4.7.3		Consumption constant	float
4.7.4		Minimum speed adjust	float
4.7.5		Maximum speed adjust	float
4.7.6		Fuel system status	float

9.1.2 Output

The output-link of the Execution Agent is structured into three main categories, as can be seen in Table 6.5.

The Performed Actions part of the output structure includes feedback that the Exag must send to the Planner and MMA. There are three separate integer variables. The *Current Action* variable indicates the sequence number of the Action within the flight plan that is currently being performed. The *Committed-to-Objective* variable indicates whether the Exag is committed to complete the Objective which is the parent Objective for the current Action. The Current Action and Committed-to-Objective variables are needed by the Planner during a re-planning event in order to properly update the previous flight plan. The *No-Plan Count* variable indicates the number of agent cycles that occur without a flight plan being provided by the Planner; this should normally be zero, and is useful in terms of fault detection.

Table 6.5. Main level of output-link structure for the Execution Agent.

Code	Name	Values	Description
1	Performed Actions	3	Data regarding the general status of the Exag
2	Current Action	11	Properties of the Action currently being performed
3	Commands	4	Commands directed to low-level control function

The Current Action part of the output structure reproduces the properties of the Action that is currently being performed by the Exag. For example, if the Current Action variable in the Performed Actions structure has the value 13, the properties of Action 13 of the flight plan will be sent as output. Action properties are as in Table 6.2, so this part of the output structure amounts for a total of 13 different variables.

Table 6.6. Commands output structure of the Execution Agent

Code	Type	Definition	Description
3.1.1	Direct commands	Direct Speed	Speed target for the Autopilot in Direct mode
3.1.2		Direct Pitch	Pitch target for the Autopilot in Direct mode
3.1.3		Direct Yaw	Yaw (bearing) target for the Autopilot in Direct mode
3.1.4		Direct Roll	Roll target for the Autopilot in Direct mode (unused)
3.1.5		Brakes	Activation of brakes (airbrakes and ground brakes)
3.2.1	Auto commands	Auto Speed	Speed target for the Autopilot in Auto mode
3.2.2		Auto Altitude	Altitude target for the Autopilot in Auto mode
3.2.3		Initial Position	Initial position of a segment in Auto mode – 2 values
3.2.4		End Position	Final position of a segment in Auto mode -2 values
3.3		Direct/Auto	Switch between Direct and Auto modes
3.4.1	Payload Commands	Power-On	Instructs Payload Management to arm payload
3.4.2		Use Recon	Instructs Payload Management to use Recon payload
3.4.3		Use Attack	Instructs Payload Management to use Attack payload
3.5		Mission Time	Sends back mission time input for cross-check

The Commands part of the output structure contains all instructions that the Execution Agent sends to the low-level control algorithms (e.g. the autopilot described in Section 3.5). The structure is divided into Direct mode and Auto mode sections, and also includes commands that are ideally directed at the Payload Management function (although at the current stage of development a Payload Management function has not been implemented). This part of the output structure amounts for a total of 16 different output variables.

Summarising, the output structure for the Execution Agent is constituted by a total of 32 variables, organized in the manner that was described in this subsection.

6.2 Exag algorithms and Soar code

As previously stated, as a Soar agent the Execution Agent is very different from the Planner Agent and the Mission Manager Agent. This is mostly due to the different timescales on which they operate: while the Planner and the MMA spend most of the time waiting to be triggered, the Exag is constantly operating and is only inactive when no flight plan from the Planner is available.

Figure 6.2 shows the hierarchy of states within the Execution Agent. While this might immediately look similar to the hierarchies of the Planner and the MMA, shown in Figure 4.3 and Figure 5.2, in fact it is radically different. The Exag has two main substates, called *action* and *no-plan*, just as the Planner has the *generate-plan* and *modify-plan* substates and the MMA has the *check-plan* and *wait-for-changes* substates. However, for the Planner and the MMA, the *generate-plan* and *check-plan* substates respectively are only entered momentarily; the normal operating states for these agents are the *modify-plan* and *wait-for-changes* substates. Instead, the Execution Agent enters the *no-plan* substate only when no flight plan is available, which should only happen at system initialization and eventually during re-planning events; the Exag will normally operate in the *action* substate.

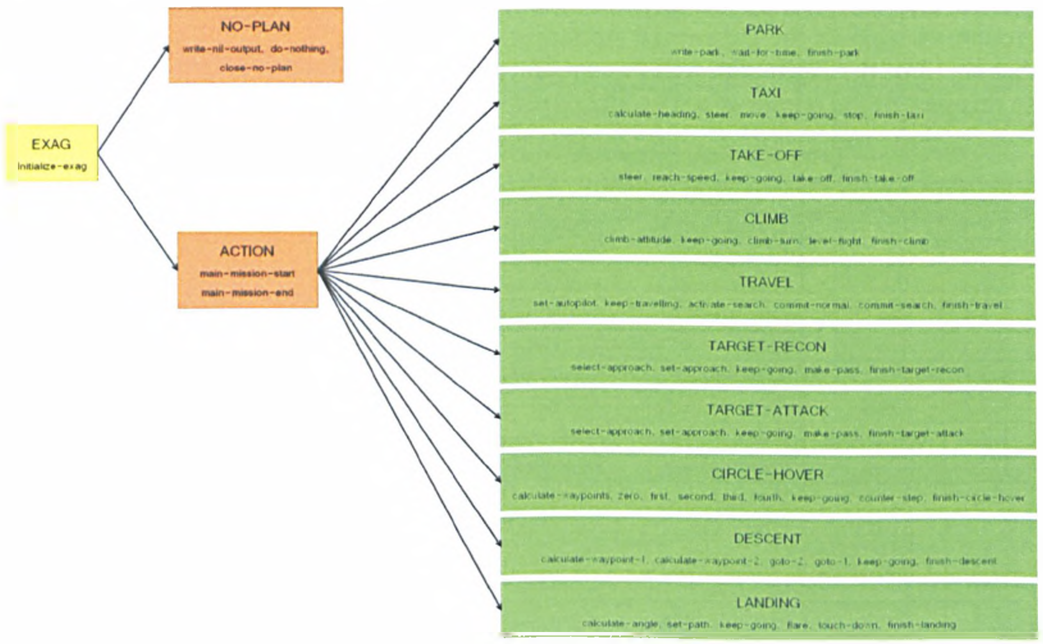

EXAG
initialize-exag
NO-PLAN
write-nil-output, do-nothing, close-no-plan
ACTION
main-mission-start
main-mission-end
PARK
write-park, wait-for-time, finish-park
TAXI
calculate-heading, steer, move, keep-going, stop, finish-taxi
TAKE-OFF
steer, reach-speed, keep-going, take-off, finish-take-off
CLIMB
climb-altitude, keep-going, climb-turn, level-flight, finish-climb
TRAVEL
set-autopilot, keep-travelling, activate-search, commit-normal, commit-search, finish-travel
TARGET-RECON
select-approach, set-approach, keep-going, make-pass, finish-target-recon
TARGET-ATTACK
select-approach, set-approach, keep-going, make-pass, finish-target-attack
CIRCLE-HOVER
calculate-waypoints, zero, first, second, third, fourth, keep-going, counter-step, finish-circle-hover
DESCENT
calculate-waypoint-1, calculate-waypoint-2, goto-2, goto-1, keep-going, finish-descent
LANDING
calculate-angle, set-path, keep-going, flare, touch-down, finish-landing

Figure 6.2. Execution Agent structural overview.

The *action* substate has 10 further substates, each corresponding to a particular type of Action. Before showing details regarding these substates, it is necessary to explain the main principles of operation of the Execution Agent.

The most important working memory element (WME) for the Exag is the *current action counter* that is placed in the top level state. This is the WME that univocally identifies which Action of the current flight plan is being accomplished; its value is sent back to the Planner (as the Current Action variable seen in Section 6.1.2). The counter is initialized at value 1 (thus indicating that the agent should immediately seek to perform the Park Action) and then its value is increased by 1 each time an Action of the flight plan is performed. Thus, the Execution Agent goes sequentially from Action 1 in the flight plan (which will always be a Park Action) to the last Action in the flight plan (whose sequence number is indicated by the *Number of Actions* variable of the *Planner Metadata* input structure, as seen in Section 6.1.1). Note that the *counter* value can never be decreased during a mission, so an already performed Action cannot be cancelled within the plans generated by the Planner; the algorithms in the *old-plan* state in the Planner, described in Section 4.2.1, guarantee this.

When a flight plan is available, the Exag enters the *action* state and then looks at the *current action counter*; the Action in the flight plan whose *sequence* property is equal to the *counter* value will be chosen, and a further substate entered depending on the Action type. For example, at initialization the *counter* has value 1; then Action 1 from the flight plan will be chosen, and since this is bound to be a Park action, the *park* substate will be entered. When the Park Action is deemed to be completed, the *park* substate will be closed and the *counter* value increased to 2, thus a new substate corresponding to Action 2 in the flight plan will be entered (since Action 2 is always a Taxi Action, this will be a *taxi* substate). This cycle is continued until the final Park Action of the flight plan is reached.

At system initialization, no flight plan will be available until the Planner Agent will have generated it; in this situation, the Execution Agent enters the *no-plan* substate, which will be detailed in Section 6.2.1. In this state, the Exag is in stasis, waiting for a valid flight plan; the *current action counter* value is kept, and is sent back to the Planner. The *no-plan* state may also be entered during re-planning events; this only happens in cases where the time used by the Planner to generate a flight plan is greater than the Execution Agent cycling time. It is to be noted that during the simulations that will be described in the second part of this chapter (Sections 6.3, 6.4 and 6.5), no instance of this was detected. At present, when the Exag is in the *no-plan* substate no command is sent to the low-level control functions; in order to improve the fault tolerance of SAMMS, the *no-plan* substate might be modified to command the UAV to maintain its current position, so that the UAV would loiter at the current position, waiting for instructions from the UAV operator, in the event of a failure of the Planner Agent.

It can be noted that there are no substates corresponding to the main-mission-start and main-mission-end Action types. Since these are "virtual" Actions, that are part of a SAMMS flight plan (to separate flight phases) but correspond to no real act of the UAV, a dedicated substate to execute them is not needed. Two operators (one per Action type) in the *action* state are sufficient; these detect the fact that the *current action counter* value corresponds to a main-mission-start/main-mission-end Action, and subsequently directly increase the *counter* value.

6.2.1 No-plan

The *no-plan* substate of the Execution Agent is entered only when the Planner Agent has not generated a valid flight plan. This is signalled by the fact that the *Number of Actions* variable in the *input-link* does not have a value greater than zero. At present, the no-plan substate simply "freezes" the Exag, in the sense that no command is sent to the low-level control functions. In future versions of SAMMS, this might be modified to improve fault tolerance of the system. For example, the *no-plan* substate might instruct the UAV to loiter at the current position if it is in flight; since the absence of a flight plan implies a Planner failure, the Exag might wait for instructions from the UAV operator and then eventually perform a return-to-base manoeuvre.

The *no-plan* state performs its functions using three operators; these are called *write-nil-output*, *do-nothing* and *close-no-plan*. The *write-nil-output* operator is executed immediately after the *no-plan* state is entered and writes directly to the *output-link*. The only meaningful variables in this case are *Current Action* and *No-Plan Count*, which are set respectively to the value of the *current action counter* in the top state and to a zero value. The rest of the *output-link* structure is filled with nil values.

The *do-nothing* operator is executed iteratively after the *write-nil-output* operator. It simply updates the *No-Plan Count* output value, increasing the value by one at each cycle; thus the *No-Plan Count* variable represents a count of the number of agent cycles during which the Exag remained in the *no-plan* state.

When a flight plan from the Planner is finally available, the *close-no-plan* operator fires; this resets the internal value of *No-Plan Count* to zero and cancels all augmentations in the *output-link*, thus closing the *no-plan* state and preparing the agent so that the *action* state might be entered.

6.2.2 Park

The *park* substate is entered at the beginning and at the end of a mission. In both cases, the aircraft is on ground and the substate corresponds to the UAV simply waiting at a parking position. At the beginning of the mission, the *park* substate waits until the specified mission start time, while at the end of the mission the *park* substate waits until the entire SAMMS system is deactivated (no time limit is specified).

Three operators are used within the *park* substate: *write-park*, *wait-for-time* and *finish-park*. The *write-park* operator fires immediately after the *park* substate has been entered and writes a full set of values to the *output-link*. The *Performed Actions* part of the output structure is set with the *Current Action* value equal to the *current action counter* value (thus either 1 or the value of the *Number of Actions* input, which indicates the final Park Action), the *Committed-to-Objective* value set to 0 to indicate no commitment to the current Objective (this is particularly significant during the take-off phase, as the UAV will instead be committed to complete the phase once the Taxi Action begins) and the *No-Plan Count* variable set to zero (this is not relevant). The *Current Action* part of the output structure contains the details regarding the Park Action being performed (see Section 4.2.2 for a description of meaningful properties). The *Commands* part of the output structure is mostly set to nil values, thus instructing the UAV to remain parked at minimum power (systems will likely already be active); the only meaningful values are the *Brakes* variable (item 3.1.5 in Table 6.6), which is set to 1 to indicate that UAV ground brakes should be active, and the *Mission Time* variable, which is set to the value received from the *input-link*.

The *wait-for-time* operator fires iteratively after the write-park operator; its only task is to update the *Mission Time* variable, so as to allow continuous cycling of the agent. The operator has an indifferent preference, so that when the *finish-park* operator fires it will be chosen over *wait-for-time*.

The *finish-park* operator fires when the *Mission Time* value in the *input-link* (item 3.1 in Table 6.3) exceeds the *time* property of the current Action; in the case of the initial Park Action in the flight plan, this value will be equal to the *Mission Start Time* (item 1.2 in Table 4.2), while in the case of the final Park Action this value will be undefined (thus the operator will in fact never fire for the final Park Action). The *finish-park* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *park* substate to close, and a new substate in the *action* state will be entered.

6.2.3 Taxi

The *taxi* substate corresponds to the Taxi Action type, and is entered when the UAV has to move on the ground, usually from the initial parking location to the designated take-off runway, or from the final landing position to a designated parking area. In any case, the Action is characterized by a starting position and a destination position; how the UAV navigates between them depends on the complexity of the algorithm implementing the Action.

The starting and destination position of a Taxi Action are its main properties and are decided directly by the Planner when generating a flight plan; however, the starting position is always updated at by the Execution Agent when starting to execute the Action, so as ensure that the UAV will be manoeuvred correctly even if the actual starting location does not coincide with the planned one. Normally, in any SAMMS flight plan two Taxi Actions will be present; one leading the UAV from the initial parking position to the runway position (where take-off may begin), and one leading the UAV from the final landing position (position where the UAV stops after landing) to the final parking position.

At the current development stage of SAMMS, the *taxi* substate has been implemented in the simplest possible manner; as can be seen in Figure 6.3, the UAV will move in a straight line from the starting position to the destination position. Since normally a fixed-wing aircraft cannot be steered on-ground if it is not moving, the taxi algorithm takes account of the manoeuvring space that is needed to steer the UAV in the desired heading. This is achieved by dividing the manoeuvre into two parts.

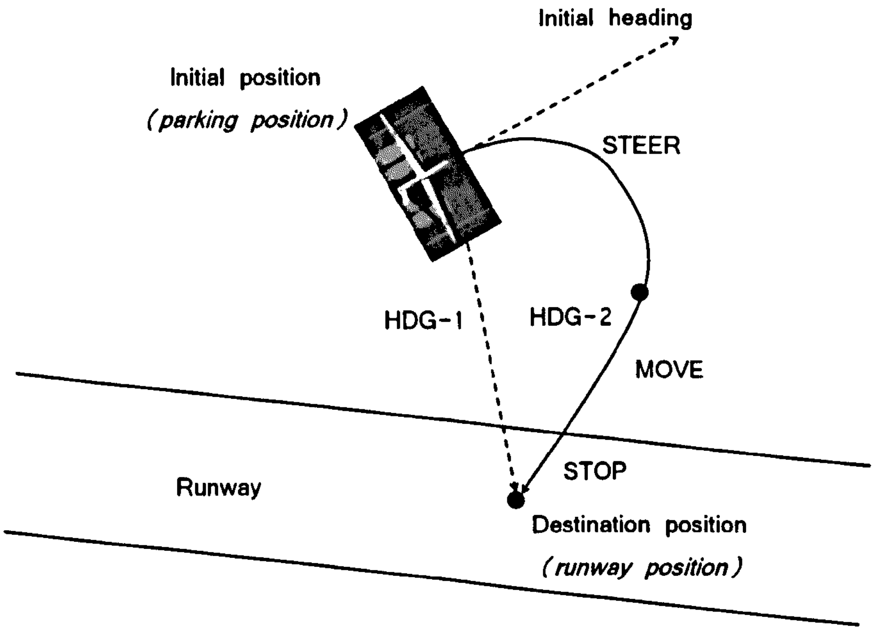


Figure 6.3. Scheme of Taxi manoeuvre in SAMMS.

This taxi manoeuvre is clearly not ideal, however a more advanced taxi algorithm was deemed out-of-scope with respect to the project goals. Taxiing is one of the most dangerous operations for commercial aircraft, especially when operating in busy airports (FAA, 2003). For this reason, it is one of the few phases of flight that has not yet been automated; this is because a taxiing aircraft must not only follow the paved ways leading to the runway, but also take account of all traffic around it, including both other aircraft and the ground vehicles of airport crew. In any case, an automated system would require a large database containing information regarding the taxiing ways at airports; such a database is not publicly available (if it is available at all). Implementation of a more complex taxi algorithm is a possible improvement path for SAMMS; a first step would be including the capability to navigate to the destination position using taxiing runways from a provided map (which could be extracted from an external database). A second improvement step would bring the capability to take account of traffic and communicate with Air Traffic Control.

The taxi substate is implemented by six operators: *calculate-heading*, *steer*, *move*, *keep-going*, *stop*, *finish-taxi*. The *calculate-heading* operator fires immediately after the *taxi* substate is entered. This calculates the direct heading that would bring the UAV from that taxi starting position to the taxi destination position. In Figure 6.3, this heading is indicated as HDG-1; it is possible to see that often this heading will not be maintained since the UAV cannot steer without moving and must then undertake a curved path.

The *steer* operator is used to make the first part of the taxi manoeuvre; rather than bringing the UAV towards the destination position, this operator is aimed at steering it towards the position (or at least in a heading which is not too far from it). In practice, the operator writes all variables to the *output-link*, so as to give appropriate commands. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Taxi Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the take-off and approach phases should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Taxi Action. The *Commands* part of the output structure is set as can be seen in Table 6.7; the Direct mode of the autopilot is used (see Section 3.5 for details), with the *Direct Speed* value set to the Ground Manoeuvre Speed (item 4.2.7 in Table 6.4), the *Direct Yaw* value set to the heading HDG-1 (calculated by the calculate-heading operator), the *Brakes* value set to 0 (brakes inactive), the *Direct/Auto* value set to 0 (autopilot in Direct mode) and the *Mission Time* value taken from the *input-link*. The *steer* operator fires only once, at the beginning of the first part of the taxi manoeuvre; it is followed by iterative firings of the *keep-going* operator, which will be described later. Following the commands given by the *steer* operator, the UAV accomplishes the first part of the taxi manoeuvre, which in Figure 6.3 is represented by the arc that connects the starting position to the point where the *move* operator fires. Steering is accomplished at the Ground Manoeuvre Speed, which is a pre-defined velocity that will allow the UAV to properly steer while on ground without travelling large distances. This is opposed to the Ground Movement Speed, which is a faster speed used by the *move* operator to perform the straight part of the taxi manoeuvre. Please note that while on ground the Direct Yaw command is not actuated by the heading-hold autopilot of Section 3.5.4, but by a dedicated ground-steer loop.

Table 6.7. Commands output for the Taxi Action

Type	Definition	Value
Direct commands	Direct Speed	Ground Manoeuvre
	Direct Pitch	-
	Direct Yaw	HDG-1
	Direct Roll	-
	Brakes	0
Auto commands	Auto Speed	-
	Auto Altitude	-
	Initial Position	-
	End Position	-
	Direct/Auto	0 (Direct)
Payload Commands	Power-On	-
	Use Recon	-
	Use Attack	-
	Mission Time	From *input-link*

The *move* operator fires when the actual heading of the UAV (from the *Real-time Data* input structure) is close to the desired HDG-1 heading (at present, the threshold for the operator to fire is set at 3 deg). At this point, the UAV will not be exactly steered towards the destination location (because of the manoeuvring space needed), but it will at least be roughly pointed towards it. The *move* operator calculates a new heading (HDG-2) that will bring the UAV towards the destination position. Then it updates three values in the *output-link*: the *Mission Time* (which must be updated at any agent cycle), the *Direct Yaw* value in *Commands* (which is set to the newly calculated heading, HDG-2) and the *Direct Speed* value (which is set to the Ground Movement Speed, item 4.2.8 in Table 6.4). The *move* operator fires only once, giving commands that will accomplish the second part of the taxi manoeuvre; just as the *steer* operator, it is followed by iterative firings of the *keep-going* operator.

The *keep-going* operator has the same function of the *wait-for-time* operator in the *park* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a manoeuvre is completed.

The *stop* operator fires when the UAV position is sufficiently close to the desired destination position; at present, the threshold for the operator to fire is set at 10 m. The *stop* operator applies commands that will stop the UAV at the current position. It updates three values in the *output-link*: the *Mission Time*, the *Direct Speed* value (which is set to zero) and the *Brakes* value (set to 1 to activate brakes). Like the *steer* and *move* operators, the *stop* operator fires once and is followed by the *keep-going* operator.

calculate-heading
steer
keep-going
keep-going
....
keep-going
move
keep-going
....
keep-going
stop
keep-going
....
keep-going
finish-taxi

Figure 6.4. Sequence of operators in the Taxi substate

The *finish-taxi* operator fires after the *stop* operator when the UAV is completely stopped. This means that the UAV must have reached the destination position and stopped there. The *finish-taxi* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *taxi* substate to close, and a new substate in the *action* state will be entered.

Figure 6.4 shows an example of the sequence in which operators should be expected to fire within the *taxi* substate. Please note the "linking" role that is performed by the *keep-going* operator.

6.2.4 Take-off

The *take-off* substate corresponds to the Take-off Action type, and is entered when the UAV is positioned close to its planned take-off position (usually, the centre of a runway). During the Action, the UAV is steered in the runway direction, then accelerates to the take-off speed and finally assumes a pitch attitude such that a positive vertical speed may be obtained.

The most relevant Action property for this Action type is the *heading* property, which tells the UAV the heading it should maintain in order to accelerate correctly along the runway. The Take-off Action is executed only once during a mission. Figure 6.5 shows a typical scheme for the take-off manoeuvre.

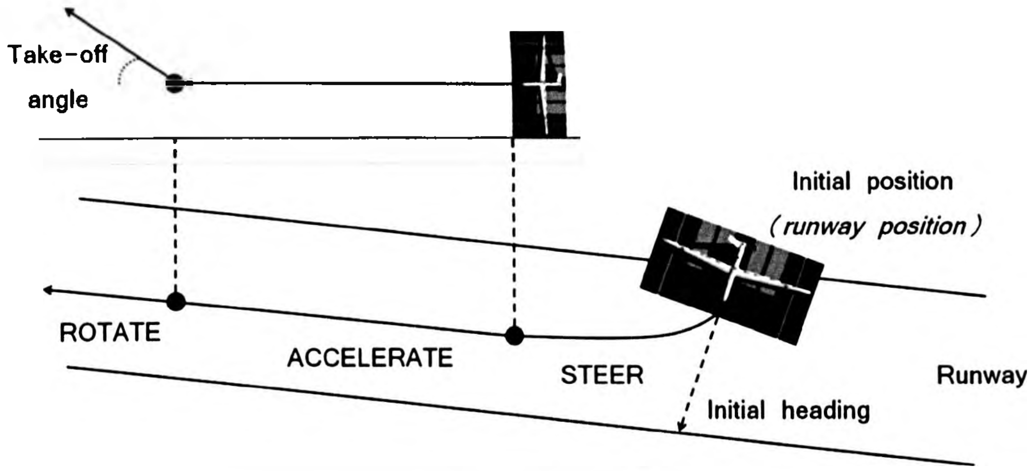


Figure 6.5. Scheme of Take-off manoeuvre in SAMMS.

The *take-off* substate is implemented by five operators: *steer*, *keep-going*, *accelerate*, *rotate* and *finish-take-off*. The *steer* operator fires immediately after the state is entered and sets commands that will align the UAV with the runway. In practice, the operator writes all variables to the *output-link*, so as to give appropriate commands. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Take-off Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the take-off phase should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Take-off Action. The *Commands* part of the output structure is set as can be seen in Table 6.8; the Direct mode of the autopilot is used (see Section 3.5 for details), with the *Direct Speed* value set to the Ground Manoeuvre Speed (item 4.2.7 in Table 6.4), the *Direct Yaw* value set to the Runway Heading value (*heading* property of the Action), the *Brakes* value set to 0 (brakes inactive), the *Direct/Auto* value set to 0 (autopilot in Direct mode) and the *Mission Time* value taken from the *input-link*. The *steer* operator fires only once, at the beginning of the take-off manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *accelerate* operator fires. Please note that while on ground the *Direct Yaw* command is not actuated by the heading-hold autopilot of Section 3.5.4, but by a dedicated ground-steer loop.

Table 6.8. Commands output for the Take-off Action

Type	Definition	Value
Direct commands	Direct Speed	Ground Manoeuvre
	Direct Pitch	-
	Direct Yaw	Runway Heading
	Direct Roll	-
	Brakes	0
Auto commands	Auto Speed	-
	Auto Altitude	-
	Initial Position	-
	End Position	-
	Direct/Auto	0 (Direct)
Payload Commands	Power-On	-
	Use Recon	-
	Use Attack	-
	Mission Time	From *input-link*

The *keep-going* operator has the same function of the *keep-going* operator in the *taxi* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a manoeuvre is completed.

The *accelerate* operator fires when the UAV is correctly aligned with the runway; that means that the actual heading value from the Real-time Data input structure (item 3.3.3 in Table 6.3) is almost equal to the indicated runway heading; at present, the threshold for the operator to fire is set at 0.5 deg. The operator changes the *Mission Time* (updating it as normal) and *Direct Speed* values; *Direct Speed* is set to the Maximum Speed value (item 4.2.1 in Table 6.4), thus ensuring that the UAV will be operating with the maximum thrust setting (as normal during a take-off). Like the *steer* operator, the *accelerate* operator fires once and is followed by the *keep-going* operator.

The *rotate* operator fires when the UAV has reached the designated Take-off Rotation Speed (item 4.2.5 in Table 6.4), which is the speed at which the aircraft should pitch up in order to establish a positive rate of climb. The operator changes the *Mission Time* (updating it as normal) and *Direct Pitch* values; *Direct Pitch* is set to the Take-off Angle value (item 4.4.6 in Table 6.4), which is a pre-determined angle that will ensure the maximum possible rate-of-climb during take-off (maximum thrust is still being commanded). Like the *steer* and *accelerate* operators, the *rotate* operator fires once and is followed by the *keep-going* operator.

The *finish-take-off* operator fires after the *rotate* operator when the UAV has cleared a pre-defined altitude above the ground; normally, a take-off is considered completed when the aircraft is 50 ft (or 15 m) above the ground level. The *finish-take-off* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *take-off* substate to close, and a new substate in the *action* state will be entered.

The sequence of operators in the *take-off* substate is similar to that shown in figure 6.4; in the comparison, the *steer* operator in *taxi* is equivalent to the *steer* operator in *take-off*, the *move* operator in *taxi* is equivalent to the *accelerate* operator in *take-off* and the *stop* operator in *taxi* is equivalent to the *rotate* operator in *take-off*. The *keep-going* operators perform the same function in both substates.

6.2.5 Climb

The *climb* substate corresponds to the Climb Action type, and is entered after the UAV has taken off and cleared a pre-defined altitude above ground (normally 50 ft or 15 m). During the Climb Action, the UAV is rapidly gaining height, until a specified altitude is reached; the first part of the climb is accomplished while maintaining the orientation of the runway where take-off occurred, the second part is instead accomplished flying towards the first Objective in the flight plan. When the target altitude is reached, the aircraft is levelled off.

The most relevant Action property for this Action type is the *altitude* property, which tells the UAV the altitude that should be reached while climbing; when this altitude is reached, the aircraft is levelled off and main mission is entered (which means that the UAV will begin executing mission Objectives). The *heading* property is also used, indicating the orientation of the runway where take-off occurred. The Climb Action is executed only once during a mission.

The *climb* substate is implemented by five operators: *climb-attitude*, *keep-going*, *climb-turn*, *level-flight* and *finish-climb*. The *climb-attitude* operator fires immediately after the state is entered and sets commands that will put the UAV into a safe attitude with a positive rate of climb. Maximum thrust is used during the entire climb manoeuvre, and the UAV is controlled using the Direct autopilot mode (rather than the Auto mode). The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Climb Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the take-off phase should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Climb Action. The *Commands* part of the output structure is set as can be seen in Table 6.9; the Direct mode of the autopilot is used, with the *Direct Speed* value set to the Maximum Speed (item 4.2.1 in Table 6.4), the *Direct Pitch* value set to the Climb Angle value (item 4.4.7 in Table 6.4), the *Direct Yaw* value set to the Runway Heading value (*heading* property of the Action), the *Brakes* value set to 0 (brakes inactive), the *Direct/Auto* value set to 0 (autopilot in Direct mode) and the *Mission Time* value taken from the *input-link*. The Climb Angle is a steep pitch angle setting (but normally not as steep as the Take-off Angle) that will allow the UAV to rapidly gain altitude without sacrificing speed. The *climb-attitude* operator fires only once, at the beginning of the climb manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *climb-turn* operator fires.

Table 6.9. Commands output for the Climb Action

Type	Definition	Value
Direct commands	Direct Speed	Maximum
	Direct Pitch	Climb Angle
	Direct Yaw	Runway Heading
	Direct Roll	-
	Brakes	0
Auto commands	Auto Speed	-
	Auto Altitude	-
	Initial Position	-
	End Position	-
	Direct/Auto	0 (Direct)
Payload Commands	Power-On	-
	Use Recon	-
	Use Attack	-
	Mission Time	From *input-link*

The *keep-going* operator has the same function of the *keep-going* operator in the *taxi* and *take-off* substates: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *climb-turn* operator fires when the Climb-Turn Altitude (item 4.3.4 in Table 6.4) has been reached. This altitude will normally be lower than the Mission Start Altitude which is the main target for the Climb Action. At the Climb-Turn Altitude, the UAV is freed from the constraint of maintaining the original runway heading, so it can manoeuvre towards the first Objective while still climbing. The operator changes the *Mission Time* (updating it as normal) and *Direct Yaw* values; a desired heading, which is meant to bring the UAV towards the first Objective, is calculated during the operator proposal phase, then *Direct Yaw* is set to this heading. Like the *climb-attitude* operator, the *climb-turn* operator fires once and is followed by the *keep-going* operator.

The *level-flight* operator fires when the Mission Start Altitude (defined by the UAV operator and contained by the altitude property of the Climb Action) has been reached. At this altitude, the UAV is considered to have safely concluded the take-off phase of the mission, thus it can begin the main mission (during which it will execute Objectives). A level flight condition is acquired before the switch to main mission mode, so as to ensure a smooth switch. The *level-flight* operator changes the *Mission Time* (updating it as normal), *Direct Speed* and *Direct Pitch* values; *Direct Speed* is set to the Cruise Speed value (item 4.2.2 in Table 6.4) and *Direct Pitch* is set to a value which ensures an almost level flight (this depends on the UAV, it is currently set to 2 deg). Like the *climb-attitude* and *climb-turn* operators, the *level-flight* operator fires once and is followed by the *keep-going* operator.

The *finish-climb* operator fires after the *level-flight* operator when the UAV has reached the Mission Start Altitude and levelled off at a pitch angle of 2 deg. The *finish-climb* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *climb* substate to close, and a new substate in the *action* state will be entered.

The sequence of operators in the *climb* substate is similar to that shown in figure 6.4; in the comparison, the *steer* operator in *taxi* is equivalent to the *climb-attitude* operator in *climb*, the *move* operator in *taxi* is equivalent to the *climb-turn* operator in *climb* and the *stop* operator in *taxi* is equivalent to the *level-flight* operator in *climb*. The *keep-going* operators perform the same function in both substates.

6.2.6 Travel

The *travel* substate corresponds to the Travel Action type, which is the most common Action type. Each Objective type is translated into a set of Actions within the actions-definition state of the Planner (see Section 4.2.5); for all Objective types, at least one Travel Action will be included, and Search Objectives are entirely composed by Travel Actions. During the Travel Action, the UAV cruises from its current position to a destination position, travelling at pre-defined altitudes and speeds.

Relevant Action properties for a Travel Action include the *Start Position*, *Position*, *Altitude* and *Speed* properties. The UAV is commanded to travel from the *Start Position* to the *Position* along a great circle route, flying at *Altitude* and at *Speed.* Note that the *Start Position* indicated in the Action is the planned one, but in practice the actual position of the UAV is used to calculate the heading that is needed to guide the UAV along the great circle route.

In navigation, two main types of routes between points on a spherical surface can be defined: great circle routes (or orthodromes) and rhumb lines (or loxodromes). On a sphere, a great circle is defined as the intersection of the sphere and a plane which passes through the centre point of the sphere; this means that the diameter of the great circle coincides with the diameter of the sphere, and that a great circle is the largest circle that can be drawn on a sphere. A great circle route is a route that connects two points and is part of a great circle; significantly, the great circle route is demonstrated to be the shortest possible route between two points on a sphere. The Earth reference system is traced using the rotation axis as a base (meridians are great circles for which the rotation axis is a diameter, and parallels are always perpendicular to the rotation axis), and within this the heading of a moving object is defined as the angle between the movement direction and a meridian. It is possible to see that, consequently, the heading to follow a great circle route is constantly changing, unless the great circle route is in fact a meridian.

This is in stark contrast with rhumb lines, which are defined as routes with a constant heading. Travelling along a rhumb line usually means travelling a longer distance compared to an equivalent great circle route; however, with its constant heading the rhumb line is very easy to follow from a control point of view, while the constantly changing heading typical of a great circle route demands continuous adjustments during navigation. The difference between the two route types can be seen in Figure 6.6. The figure compares the two routes in two different map projections; the gnomonic projection shows great circle routes as a straight line, while the Mercator projection shows rhumb lines as a straight line.

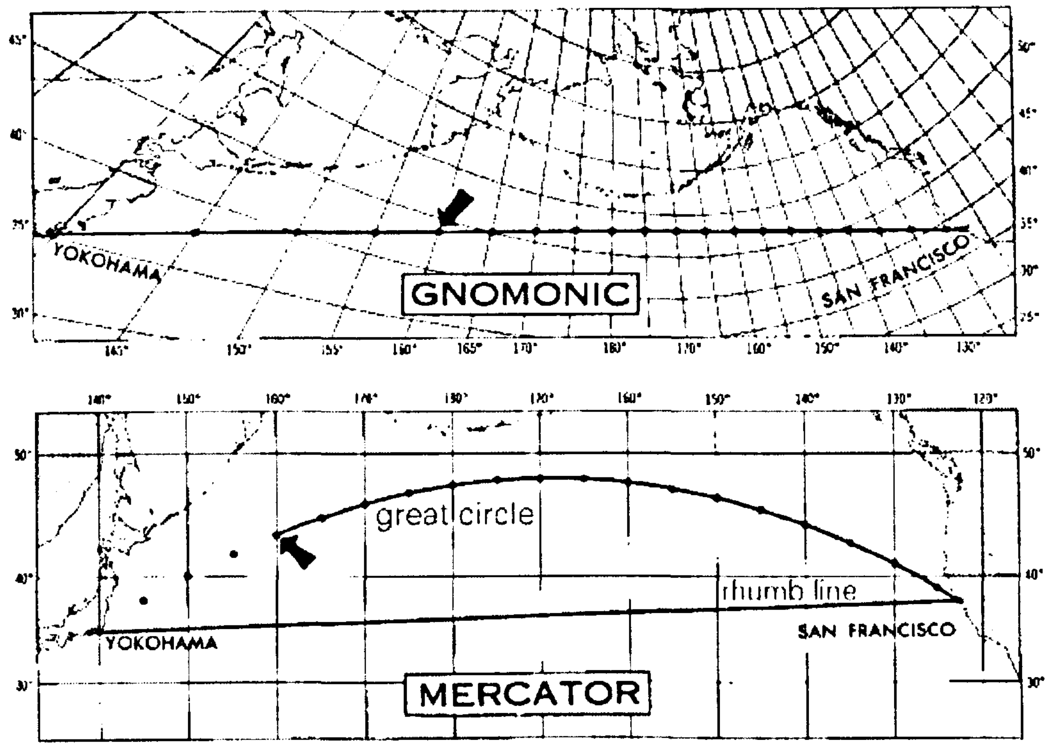


Figure 6.6. Great circle routes and rhumb lines.

Because of the large radius of Earth, the difference between a great circle route and a rhumb line is very small when small distances are involved. The difference in distance is substantial over large distances; for example, for the San Francisco – Yokohama routes shown in Figure 6.6, the great circle distance is 4517 miles, while the rhumb distance is 4723 miles. However a great circle route is unpractical to follow without an automatic system, and in navigation great circle routes are usually approximated by a series of rhumb lines. For the purposes of SAMMS, it was decided to implement a navigation algorithm that purely follows great circle routes; the Navigation subsystem of the Autopilot (see Section 3.5.5) continuously calculates the required heading that will allow to reach the destination position from the current UAV position.

It is to be noted that this means that Travel Actions are implemented using the Auto mode of the autopilot (see Section 3.5); this is in contrast to the other Action types described (Park, Taxi, Take-off, Climb). In fact, all Action types related to the take-off and approach phases (Park, Taxi, Take-off, Climb, Landing) use the Direct autopilot mode, while all Action type related to the main mission phase (Travel, Target-Recon, Target-Attack, Circle) use the Auto mode. An exception to this is represented by the Descent Action, which is part of the approach phase but uses the Auto mode.

The *travel* substate is implemented by six operators: *set-autopilot*, *keep-travelling*, *activate-search*, *commit-normal*, *commit-search* and *finish-travel*. The *set-autopilot* operator fires immediately after the state is entered and sets commands that will direct the UAV towards the destination position. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Travel Action *sequence* number, the *Committed-to-Objective* value is set to 0 at the beginning of the Action, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Travel Action. The *Commands* part of the output structure is set as can be seen in Table 6.10; the Auto mode of the autopilot is used, with the *Auto Speed* value set to the Speed property of the Action, the *Auto Altitude* value set to the Altitude property of the Action, the *Initial Position* value set to the Start Position property of the Action, the *End Position* value set to the Position property of the Action, the *Direct/Auto* value set to 1 (autopilot in Auto mode) and the *Mission Time* value taken from the *input-link*. The *set-autopilot* operator fires only once, at the beginning of the Travel Action; it is followed by iterative firings of the *keep-travelling* operator.

Table 6.10. Commands output for the Travel Action

Type	Definition	Value
Direct commands	Direct Speed	-
	Direct Pitch	-
	Direct Yaw	-
	Direct Roll	-
	Brakes	-
Auto commands	Auto Speed	Action Speed
	Auto Altitude	Action Altitude
	Initial Position	Start Position
	End Position	Position
	Direct/Auto	1 (Auto)
Payload Commands	Power-On	0 (off)
	Use Recon	0 (off)
	Use Attack	0 (off)
	Mission Time	From *input-link*

The *keep-travelling* operator has the same function of the *keep-going* operator in the *taxi*, *take-off* and *climb* substates: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *activate-search* operator is used to activate appropriate payload while performing a Search Objective. It fires after the set-autopilot operator when the current Action is detected to be part of a Search Objective and consequently changes the corresponding output variables. It updates both the *Mission Time* output (updating it as normal) and the *Power-On* output (item 4.4.1 in Table 6.6), which goes from a value of 0 to a value of 1 (at the current stage of implementation of SAMMS, payload is an external component that is managed using binary switches for activation and de-activation). It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed. Like the *set-autopilot* operator, the *activate-search* operator fires once and is followed by the *keep-travelling* operator.

The *commit-normal* operator is used to update the Committed-to-Objective output of the Exag. Most Objective types (all apart from Search Objectives) consist in a Travel Action that brings the UAV to a position where some act must be performed, plus an Action that performs the act (the Travel Action might be split in several segments by the *mission-path-adjustment* algorithm). It is clear that once the UAV is close to the Objective position, this should be pursued even in the occurrence of a re-planning event; this is to avoid cases in which an Objective has almost been reached but is then aborted because of a re-planning event. The decision regarding commitment to an Objective must be made by the Exag; it is immediate for Actions related to the take-off and approach phases (which should not be interrupted, thus always involve commitment to the current Objective), and also for the Target-Recon, Target-Attack and Circle Actions (Target-Recon and Target-Attack are always committed, since the Objective is almost concluded, while Circle is never committed, since loitering must be interruptible). However, the decision is very complex for Travel Actions, since it potentially involves a large amount of variables (distance already covered, validity of the Objective, distances to other Objectives, etc.). At present, a very simple strategy has been implemented within the Exag: the agent becomes committed to the current Objective once half the distance towards the destination has been covered. Note this is not valid for Travel Actions that originate from a Search Objective. An elaboration calculates the distance between the *Start Position* and *Position*; another elaboration (called *travel*elaborate*distance*) calculates the distance between the current UAV position (from the *Real-time Data* input structure) and the *Position*. The *commit-normal* operator fires when the current distance from the destination becomes less than half the initial distance. The operator changes the *Mission Time* (updating it as normal) and *Committed-to-Objective* values; Committed-to-Objective is changed from a 0 value to a 1 value, to indicate commitment to the current Objective. Like the *set-autopilot* operator, the *commit-normal* operator fires once and is followed by the *keep-travelling* operator.

The *commit-normal* operator does not work on Travel Actions that originate from a Search Objective; for these Actions, the *commit-search* operator is used. This operator implements a different decision strategy; for Search Objectives, the Exag becomes committed once a certain number of the Search segments have been covered. In practice, a set of elaborations extracts from the flight plan the *sequence* number of the first and last Actions that constitute the Search; these values are used to compute a *sequence* number threshold. When the *sequence* number of the current Action exceeds the threshold, the Exag becomes committed to the current Objective; at present, the threshold is set to be at the middle of the search. Unlike the *commit-normal* operator, the *commit-search* operator fires immediately after the *set-autopilot* operator. The operator changes the *Mission Time* (updating it as normal) and *Committed-to-Objective* values; Committed-to-Objective is changed from a 0 value to a 1 value, to indicate commitment to the current Objective. Like the *commit-normal* operator, the *commit-search* operator fires once and is followed by the *keep-travelling* operator.

The *finish-travel* operator fires after all other operators in the substate, when the UAV has reached the destination Position. Since the UAV is flying, the operator is set to fire when the distance calculated by the *travel*elaborate*distance* elaboration goes below a pre-defined threshold; this threshold must be greater that the distance flown by the UAV between two Exag cycles (it is currently set to 150 m). The *finish-travel* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *travel* substate to close, and a new substate in the *action* state will be entered.

set-autopilot
(activate-search)
(commit-search)
keep-travelling
....
keep-travelling
(commit-normal)
keep-travelling
....
keep-travelling
finish-travel

Figure 6.7. Sequence of operators in the Travel substate

Figure 6.7 shows an example of the sequence in which operators should be expected to fire within the *travel* substate.

6.2.7 Target-Recon and Target-Attack

The *target-recon* and *target-attack* substates correspond to the Target-Recon and Target-Attack Action types. During these Actions, the UAV is manoeuvred to perform a pass over a target, so that the appropriate payload can be used. Both Action types are accomplished using the same pre-defined manoeuvre; however, this manoeuvre is defined by a set of parameters, which can be different between the two Action types. For this reason, this subsection will focus on describing the *target-recon* substate and Action; all statements will be directly applicable to the *target-attack* substate and Action, with differences consisting of different parameters and different naming.

The main properties of a Target-Recon Action are the *Position* property and the *Target* property. The *Target* property is in fact more important, since it will allow to update the *Position* value with the most recent *Position* information from the Entity list. Another very relevant property for Target-Recon is related to the following Action in the flight plan; the *Position* property of the Action that immediately follows the Target-Recon Action in the flight plan is used to determine the manoeuvre to be accomplished.

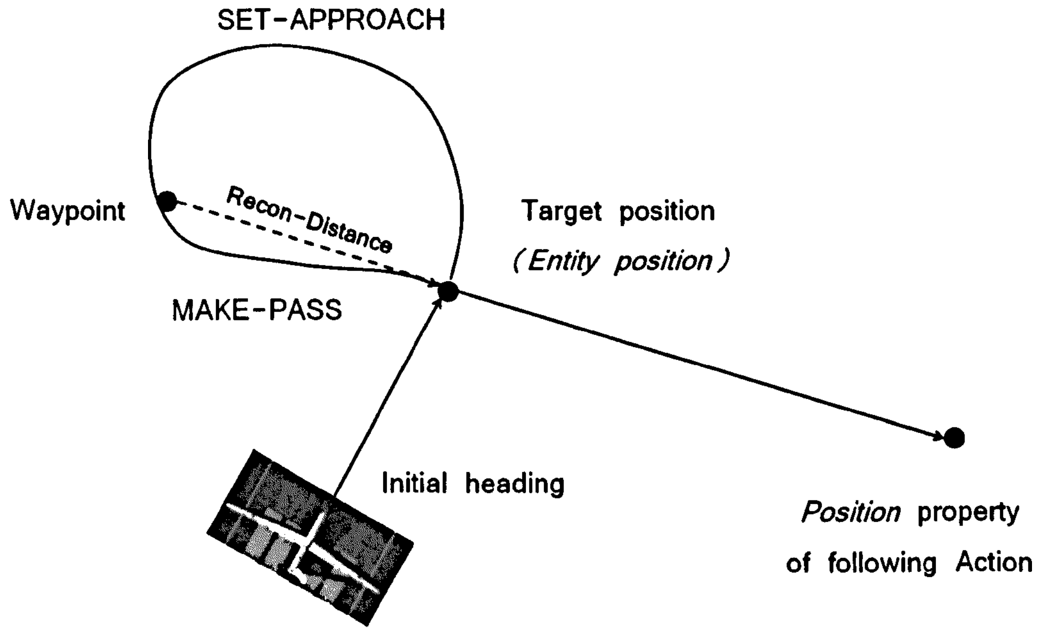


Figure 6.8. Scheme of Target-Recon manoeuvre in SAMMS.

The Target-Recon Action is accomplished using the manoeuvre that can be seen in Figure 6.8. The manoeuvre consists of two segments: first the UAV is moved towards a waypoint, then it flies back towards the target position. Because of how the Exag works, the UAV arrives above the target flying at the cruise altitude; the UAV flies over the target, then starts turning towards the waypoint, then performs another turn when the waypoint is reached. Normally, a Target-Recon Action will be accomplished at a lower altitude compared to the cruise altitude; thus the UAV will be descending throughout the manoeuvre, until it reaches the designated altitude. The waypoint position is calculated on the basis of two elements: the *Recon Distance* value (item 4.4.1 in Table 6.4) and the *Position* property of the following Action in the flight plan. The waypoint is placed along the continuation of the great circle that connects the target *Position* and the following Action *Position*, at *Recon Distance* from the target *Position*. The resulting manoeuvre is largely dependent on three factors: not only the *Recon Distance* value and the *Position* property of the following Action, but also the initial heading of the UAV. It is be noted that the UAV might need to fly more complex trajectories (for example, running two circles) in order to reach the waypoint correctly (that is, getting close enough to it and at the correct altitude); this depends on the actual UAV manoeuvrability and on the low-level control system performance.

The *target-recon* substate is implemented by five operators: *select-approach*, *set-approach*, *keep-going*, *make-pass* and *finish-target-recon*. The *select-approach* operator fires immediately after the state is entered and calculates the position of the waypoint used for the manoeuvre. Based on the Target *Position*, the *Position* property of the following Action and the *Recon Distance* value, the waypoint coordinates are calculated and stored into a dedicated working memory element.

The *set-approach* operator fires immediately after *select-approach* and gives commands to guide the UAV towards the calculated waypoint. The Auto mode of the autopilot is used, so only the waypoint coordinates are needed. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Target-Recon Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the manoeuvre should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Target-Recon Action. The *Commands* part of the output structure is set as can be seen in Table 6.11; the Auto mode of the autopilot is used, with the *Auto Speed* value set to the value of the Speed property of the Action (usually Analyze Speed, item 4.2.12 in Table 6.4), the *Auto Altitude* value set to the value of the Altitude property of the Action (usually Analyze Altitude, item 4.3.10 in Table 6.4), the *Initial Position* value set to the *Position* of the Action *Target*, the *End Position* value set to the waypoint calculated by the *select-approach* operator, the *Direct/Auto* value set to 1 (autopilot in Auto mode) and the *Mission Time* value taken from the *input-link*. To indicate that the UAV should prepare for activation of the Recon payload, the *Power-On* payload command is set to 1. The *set-approach* operator fires only once, at the beginning of the target-recon manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *make-pass* operator fires.

Table 6.11. Commands output for the Target-Recon Action

Type	Definition	Value
Direct commands	Direct Speed	-
	Direct Pitch	-
	Direct Yaw	-
	Direct Roll	-
	Brakes	-
Auto commands	Auto Speed	Speed property
	Auto Altitude	Altitude property
	Initial Position	Target Position
	End Position	Waypoint
	Direct/Auto	1 (Auto)
Payload Commands	Power-On	1 (on)
	Use Recon	0 (off)
	Use Attack	0 (off)
	Mission Time	From *input-link*

The *keep-going* operator has the same function of the *keep-travelling* operator in the *travel* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *make-pass* operator fires when the UAV is in proximity of the waypoint calculated by the *select-approach* operator. An elaboration is used to calculate the distance between the UAV's current detected position (taken from the Real-time Data input structure) and the waypoint position; when this distance goes below a specified threshold, the operator fires and consequently manoeuvres the UAV towards the target. The threshold is currently set at 100 m. The *make-pass* operator changes the *Mission Time* (updating it as normal), *Initial Position*, *End Position* and *Use-Recon* values. *Initial Position* is set to the waypoint position value, *End Position* is set to the *Position* of the Action *Target* and *Use Recon* is set to 1 (indicating activation of the Recon payload). Like the *set-approach* operator, the *make-pass* operator fires once and is followed by the *keep-going* operator.

The *finish-target-recon* operator fires after all other operators in the substate, when the UAV has reached again the target position (this time flying at the Recon values for Speed and Altitude). An elaboration is used to calculate the distance between the UAV's current detected position (taken from the Real-time Data input structure) and the target position; when this distance goes below a specified threshold (the same used for the make-pass operator), the operator fires and consequently terminates the Action. The *finish-target-recon* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *target-recon* substate to close, and a new substate in the *action* state will be entered.

It is to be noted that a Target-Recon Action will normally be followed by a Travel Action, so typically the UAV will pass over the target, circle around it while descending, pass again over it (in a flight configuration more suitable to perform its task) and then continue on to the next Objective, climbing back up to the normal cruise configuration. The operator firing sequence will usually be *select-approach/set-approach/make-pass/finish-target-recon*, with the *keep-going* operator firing iteratively between *set-approach* and *make-pass* and between *make-pass* and *finish-target-recon.*

The entire description of the *target-recon* substate is valid for the *target-attack* substate, apart from some small differences in parameters and naming. The *target-attack* substate uses the Target-Attack distance (item 4.4.2 in Table 6.4) instead of the Target-Recon distance; also, the *Speed* and *Altitude* properties of a Target-Attack Action will normally be set to the *Attack Speed* and *Attack Altitude* values (items 4.2.13 and 4.3.11 in Table 6.4), although the decision is made by the Planner and passed to the Exag as an Action property. Most operators of the *target-attack* substate keep the name and rules defined for the their *target-recon* counterpart; the only differences regard the *make-pass* operator, which activates the Use Attack payload command instead of Use Recon, and the *finish-target-attack* operator, which is named differently but performs the same functions of *finish-target-recon.*

6.2.8 Circle

The *circle* substate corresponds to the Circle Action type, which consists in a manoeuvre that will allow the UAV to loiter about a specified position. For a rotary-wing aircraft, the Circle Action would translate to a Hover Action, but a fixed-wing UAV must continuously move when loitering. Loitering is achieved by flying an approximated circular path around the desired position. The UAV might perform tasks while loitering (for example, acting as a communications relay or collecting sensor data), and will normally be flying at low speeds so as to minimize fuel consumption. The Circle Action is considered finished after a specified time.

The most relevant Action property for the Circle Action type is the *Time* property, which specifies the time which the UAV should spend loitering at the desired position. This acts as a time limit: when the specified time has been reached, the UAV can move on to the next Objective. It is to be noted that if no Time Priority is specified for the Objective, the Planner will not consider how much time is spent loitering; if the position is reached just before the time specified by the Time property, the UAV will loiter there for a very short time. Specifying a Time Priority will ensure that the UAV spends the specified times at the position; for example, if the Time Priority is set to 1200 sec and the Time property is set to 1800 sec, the Planner will make sure that the UAV spends the time between 1200 sec and 1800 sec at the position (possibly arriving earlier). Without a Time Priority, the UAV might arrive at the position at time 1790 sec and stay there for only 10 sec. The *Position* and *Duty* properties are also important, since they specify the position which the UAV should loiter about and the task it should perform at the position.

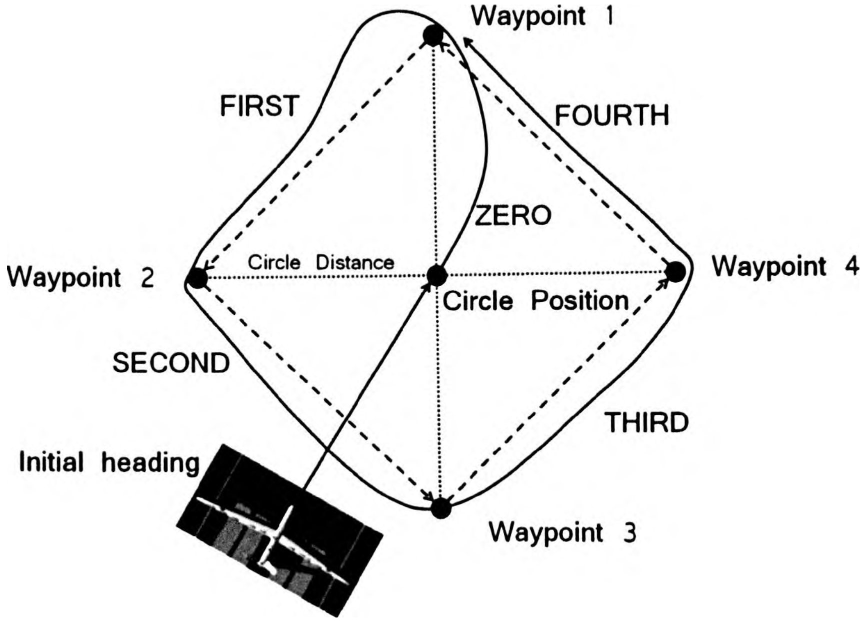


Figure 6.9. Scheme of Circle manoeuvre in SAMMS.

The Circle Action is accomplished using the manoeuvre that can be seen in Figure 6.9. The manoeuvre consists of a series of waypoints which are cycled while loitering. Four waypoints are defined, positioned at the *Circle Distance* (item 4.4.3 in Table 6.4) from the Circle Position along the cardinal directions (North, West, South, East). The UAV first reaches the *Circle Position*, then it navigates towards waypoint 1, and then towards waypoints 2, 3 and 4 in this sequence. The UAV then navigates again to waypoint 1, continuing the cycle until the time limit for the Circle Action is reached. The resulting trajectory is not circular, but considering the UAV manoeuvrability it can be considered a good approximation when the Circle Distance value is small. This distance should be defined to be as small as possible, while still allowing the UAV to manoeuvre correctly between the waypoints (since the UAV will normally be flying at low speeds, latero-directional manoeuvrability is higher).

The *circle* substate is implemented by nine operators: *calculate-waypoints*, *zero*, *first*, *second*, *third*, *fourth*, *keep-going*, *counter-step* and *finish-circle*. The *calculate-waypoints* operator fires immediately after the state is entered and calculates the position of the four waypoints needed for the manoeuvre. These are obtained from the desired *Circle Position* by adding the *Circle Distance* in the four cardinal directions. The calculated waypoints are stored into dedicated working memory elements.

The *zero* operator fires immediately after *calculate-waypoints* and gives commands to guide the UAV towards the first waypoint. The Auto mode of the autopilot is used, so only the waypoint coordinates are needed. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Circle Action *sequence* number, the *Committed-to-Objective* value is set to 0 since the manoeuvre can be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Circle Action. The *Commands* part of the output structure is set as can be seen in Table 6.12; the Auto mode of the autopilot is used, with the *Auto Speed* value set to the value of the Speed property of the Action (usually Orbit Speed, item 4.2.4 in Table 6.4), the *Auto Altitude* value set to the value of the Altitude property of the Action (this is commanded by the UAV operator when specifying the Orbit Objective from which the Circle Action originates), the *Initial Position* value set to the Action *Position*, the *End Position* value set to the first waypoint calculated by the *calculate-waypoints* operator, the *Direct/Auto* value set to 1 (autopilot in Auto mode) and the *Mission Time* value taken from the *input-link*. To indicate that the UAV might be executing a task while loitering, the *Power-On* payload command is set to 1. The *zero* operator fires only once, at the beginning of the circle manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *first* operator fires.

Table 6.12. Commands output for the Circle Action

Type	Definition	Value
Direct commands	Direct Speed	-
	Direct Pitch	-
	Direct Yaw	-
	Direct Roll	-
	Brakes	-
Auto commands	Auto Speed	Speed property
	Auto Altitude	Altitude property
	Initial Position	Circle Position
	End Position	Waypoint 1
	Direct/Auto	1 (Auto)
Payload Commands	Power-On	1 (on)
	Use Recon	0 (off)
	Use Attack	0 (off)
	Mission Time	From *input-link*

The *first* operator fires when the *counter-step* operator detects that the current destination waypoint (which will always be waypoint 1 when the *first* operator is firing) is being reached, giving commands to guide the UAV towards waypoint 2. The *first* operator changes the *Mission Time* (updating it as normal), *Initial Position* and *End Position* values. *Initial Position* is set to the position of waypoint 1; *End Position* is set to the position of waypoint 2.

The *second*, *third* and *fourth* operators perform the same functions of the *first* operator, but for different waypoints; *second* sets commands for the segment between waypoint 2 and waypoint 3, *third* sets commands for the segment between waypoint 3 and waypoint 4, *fourth* sets commands for the segment between waypoint 4 and waypoint 1. It is to be noted that unlike the *zero* operator, all these operators fire repeatedly while the Action is being accomplished; they are preceded by the *counter-step* operator and followed by the *keep-going* operator.

The *keep-going* operator has the same function of the *keep-travelling* operator in the *travel* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *counter-step* is used to manage the execution of the *first*, *second*, *third* and *fourth* operators. The operator uses a counter to know which segment of the manoeuvre is being flown. An elaboration is used to calculate the distance between the UAV's current detected position (taken from the Real-time Data input structure) and the current destination waypoint position; when this distance goes below a specified threshold, the operator fires and consequently manoeuvres the UAV towards the target. The threshold is currently set at 80 m; since the UAV typically performs a Circle Action at low speeds, its manoeuvrability is improved and the threshold can be set at lower values. When the *counter-step* operator fires, it increases the counter value by one. This will in turn cause the firing of one among the *first*, *second*, *third* and *fourth* operators; these operators always fire in this sequence (but with the *keep-going* and *counter-step* operators firing between them). It is to be noted that the *counter-step* operator does not write on the *output-link*, thus the *first*, *second*, *third* and *fourth* operators fire immediately after *counter-step* (during the same agent cycle).

The *finish-circle* operator fires after all other operators in the substate, when the *Mission Time* value from the *input-link* equals the value specified by the *Time* property of the Circle Action. The *finish-circle* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *circle* substate to close, and a new substate in the *action* state will be entered.

calculate-waypoints
zero
keep-going
...
counter-step
first
keep-going
...
counter-step
second
keep-going
...
counter-step
third
keep-going
...
counter-step
fourth
keep-going
...
counter-step
first
keep-going
...
finish-circle

Figure 6.10. Sequence of operators in the Circle substate

Figure 6.10 shows an example of the sequence in which operators should be expected to fire within the *circle* substate.

6.2.9 Descent

The *descent* substate corresponds to the Descent Action type, during which the UAV is manoeuvred from a position at cruise altitude to a position from which a landing can be accomplished. This means that the UAV has to be taken to a low altitude above ground and aligned with the landing runway. In practice, the manoeuvre is accomplished by flying along a set of waypoints, which are determined on the basis of pre-defined parameters. It is to be noted that, unlike all other Action types belonging to the take-off or approach phase, the Auto mode of the Autopilot is used. The Descent Action is concluded (and immediately followed by a Landing Action) when the last waypoint is reached. The Descent and Landing Actions as they are implemented within SAMMS do not make use of the Instrument Landing System (ILS) that is used on commercial aircraft, since this relies on ground-based transmitters.

The most relevant Action properties for the Descent Action type are the P*osition*, *Altitude* and *Heading* properties. These specify the characteristics of the landing runway, respectively the ideal touch-down point, the altitude of ground and the runway heading. These properties are fused with information from the Flight Parameters input structure in order to calculate the waypoints which define the approach trajectory.

Figure 6.11 shows the vertical and lateral profiles for the Descent Action. The manoeuvre assumes that the UAV will first fly above the desired landing location, then use two waypoints to perform a correct approach. Both waypoints are aligned with the runway; their position is found at a pre-determined distance from the ideal touch-down position along the runway axis, specifically Pre-Land Distance (item 4.4.5 in Table 6.4) for Waypoint 1 and Descent Distance (item 4.4.4 in Table 6.4) for Waypoint 2; their altitude is calculated adding pre-determined altitude values to the known runway altitude (Altitude property of the Descent Action), specifically Pre-Land Altitude (item 4.3.6 in Table 6.4) for Waypoint 1 and Descent Altitude (item 4.3.5 in Table 6.4) for Waypoint 2. The current values for these parameters are: Descent Distance 2000 m, Descent Altitude 200 m, Pre-Land Distance 500 m, Pre-Land Altitude 80 m. The Exag first guides the UAV from the Runway Position (which is where the Action is begun) to Waypoint 2; during this part of the manoeuvre, the main goal is to lose UAV altitude and speed, so that a correct approach can then be established. After reaching Waypoint 2, the Exag guides the UAV towards Waypoint 1; during this phase, the main goal is to align the UAV with the runway. The use of two waypoints is necessary to ensure that Waypoint 1, from which the Landing manoeuvre will begin, is reached with a sufficiently small angle with respect to the runway heading. The Descent Action is terminated when the UAV reaches Waypoint 1.

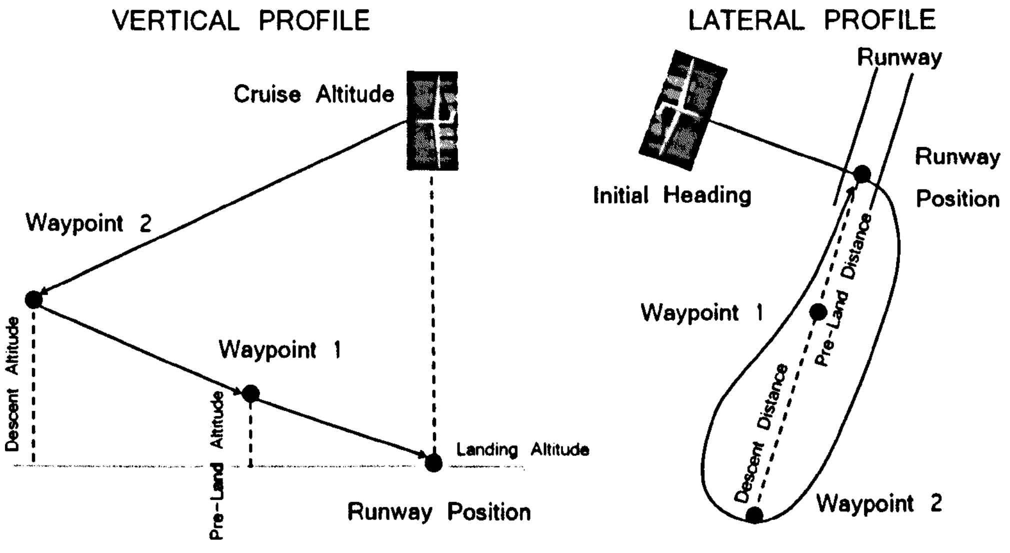


Figure 6.11. Scheme of Descent manoeuvre in SAMMS.

The *descent* substate is implemented by six operators: *calculate-waypoint-1*, *calculate-waypoint-2*, *goto-2*, *goto-1*, *keep-going* and *finish-descent*. The *calculate-waypoint-1* operator fires immediately after the state is entered and calculates the position of Waypoint 1. Based on the *Position*, *Altitude* and *Heading* properties of the Descent Action, the waypoint coordinates are calculated by adding the *Pre-Land Distance* and *Pre-Land Altitude* values. Waypoint 1 coordinates are calculated and stored into a dedicated working memory element.

The *calculate-waypoint-2* operator fires immediately after the *calculate-waypoint-1* operator and calculates the position of Waypoint 2. Based on the *Position, Altitude* and *Heading* properties of the Descent Action, the waypoint coordinates are calculated by adding the D*escent Distance* and *Descent Altitude* values. Waypoint 2 coordinates are calculated and stored into a dedicated working memory element.

The *goto-2* operator fires immediately after *calculate-waypoint-2* and gives commands to guide the UAV towards Waypoint 2. The Auto mode of the autopilot is used, so only the waypoint coordinates are needed. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Descent Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the manoeuvre should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Descent Action. The *Commands* part of the output structure is set as can be seen in Table 6.13; the Auto mode of the autopilot is used, with the *Auto Speed* value set to the Orbit Speed value (item 4.2.4 in Table 6.4), the *Auto Altitude* value set to the Waypoint 2 altitude, the *Initial Position* value set to the *Position* property of the Action (the runway position), the *End Position* value set to Waypoint 2 position, the *Direct/Auto* value set to 1 (autopilot in Auto mode) and the *Mission Time* value taken from the *input-link*. The *goto-2* operator fires only once, at the beginning of the descent manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *goto-1* operator fires.

Table 6.13. Commands output for the Descent Action

Type	Definition	Value
Direct commands	Direct Speed	-
	Direct Pitch	-
	Direct Yaw	-
	Direct Roll	-
	Brakes	-
Auto commands	Auto Speed	Orbit Speed
	Auto Altitude	Waypoint 2
	Initial Position	Runway Position
	End Position	Waypoint 2
	Direct/Auto	1 (Auto)
Payload Commands	Power-On	-
	Use Recon	-
	Use Attack	-
	Mission Time	From *input-link*

The *goto-1* operator fires when the UAV is in proximity of Waypoint 2 and directs the UAV towards Waypoint 1. An elaboration is used to calculate the distance between the UAV's current detected position (taken from the Real-time Data input structure) and the Waypoint 2 position; when this distance goes below a specified threshold, the operator fires and consequently manoeuvres the UAV towards Waypoint 1. The threshold is currently set at 100 m. The *goto-1* operator changes the *Mission Time* (updating it as normal), *Initial Position*, *End Position* and *Auto Altitude* values. *Initial Position* is set to Waypoint 2 position, *End Position* is set to Waypoint 1 position and *Auto Altitude* is set to Waypoint 1 altitude. Like the *goto-2* operator, the *goto-1* operator fires once and is followed by the *keep-going* operator.

The *keep-going* operator has the same function of the *keep-travelling* operator in the *travel* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *finish-descent* operator fires after all other operators in the substate, when the UAV has reached Waypoint 1. An elaboration is used to calculate the distance between the UAV's current detected position (taken from the Real-time Data input structure) and Waypoint 1 position; when this distance goes below a specified threshold, the operator fires and consequently terminates the Action. The *finish-descent* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *descent* substate to close, and a new substate in the *action* state will be entered.

The sequence of operators in the *descent* substate is simple. At first the *calculate-waypoint-1*, *calculate-waypoint-2* and *goto-2* operators fire sequentially. They are then followed by iterative firings of *keep-going*, until the *goto-1* operator fires. Then again the *keep-going* operator will fire iteratively, until the *finish-descent* operator closes the substate.

6.2.10 Landing

The *landing* substate corresponds to the Landing Action type, during which the UAV accomplishes the final part of the approach phase. The Action begins with the UAV already almost aligned with the runway; its altitude and speed are such that a correct landing can be achieved. Unlike the Descent Action, the Direct mode of the Autopilot is used. The UAV first assumes a descending pitch attitude and then performs a flare when almost on-ground. The Landing Action is concluded when the UAV has completely stopped after landing on the desired runway.

The most relevant Action properties for the Landing Action type are the *Position*, *Altitude* and *Heading* properties. These specify the characteristics of the landing runway, respectively the ideal touch-down point, the altitude of ground and the runway heading. Using the Pre-Land Distance and Pre-Land Altitude values (items 4.4.5 and 4.3.6 in Table 6.4), an ideal approach angle is calculated; the UAV is then manoeuvred so as to maintain this angle from the initial position of the Action (which should be Waypoint 1 from the Descent Action) to the ideal touch-down position.

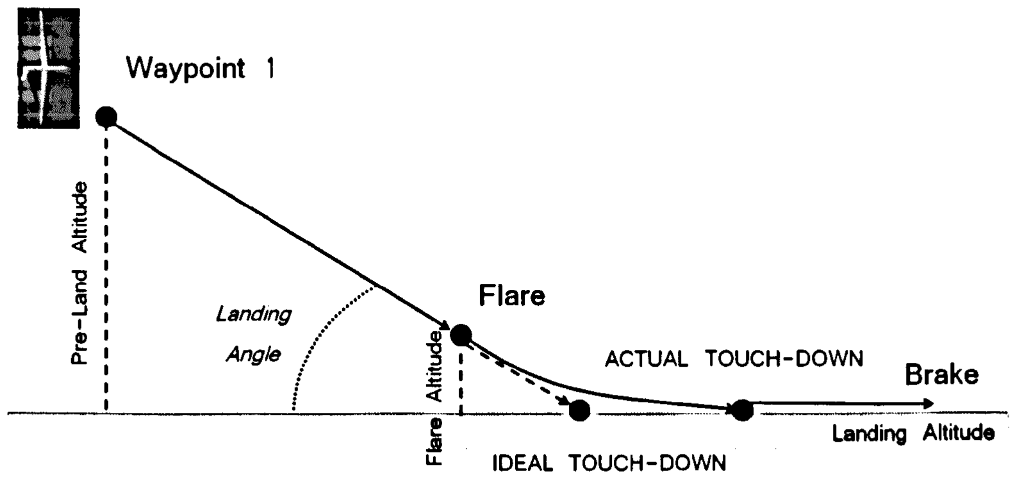


Figure 6.12. Scheme of Landing manoeuvre in SAMMS.

Figure 6.12 shows the vertical profile for the Landing Action. The manoeuvre assumes that the UAV has correctly reached Waypoint 1 during the Descent Action. From Waypoint 1, the UAV enters a descending path with a pre-defined angle, which is the *Landing Angle* in the figure and is obtained from the Pre-Land Distance and Pre-Land Altitude values. The UAV is maintained aligned with the runway and descending at the Landing Angle until the Flare Altitude is reached; at this altitude, the Exag gives commands that will cause the UAV to reduce the rate of descent and assume the correct landing attitude. The flare manoeuvre brings the UAV into contact with the ground; after contact with the ground has been ensured, full braking action is taken, until the UAV is completely stopped, terminating the Landing Action.

The *landing* substate is implemented by six operators: *calculate-angle*, *set-path*, *keep-going*, *flare*, *touch-down* and *finish-landing*. The *calculate-angle* operator fires immediately after the state is entered and calculates the *Landing Angle* which should be maintained by the UAV in order to land at the ideal touch-down position from Waypoint 1. The *Landing Angle* is directly calculated from the *Pre-Land Distance* and *Pre-Land Altitude* values and stored into a dedicated working memory element. Note that technically this is a ramp angle γ rather than a pitch angle θ; since the pitch-hold autopilot maintains a desired pitch angle, the UAV angle of attack α should be known to give the correct command ($\theta = \gamma + \alpha$). On-board calculation of the angle of attack requires a combination of sensors which might not be available; however, the angle of attack can be estimated to be small for a light UAV in a landing configuration, so at present the *Landing Angle* value is directly fed into the pitch-hold autopilot.

The *set-path* operator fires immediately after *calculate-angle* and gives commands to guide the UAV along the final descending path from Descent Waypoint 1 to the ideal touch-down position. The Direct mode of the autopilot is used. The *Performed Actions* and *Current Action* parts of the output structure are standard; the *Current Action* value is taken from the Landing Action *sequence* number, the *Committed-to-Objective* value is set to 1 since the manoeuvre should not be interrupted by re-planning events, the *No-Plan Count* is set to zero and the *Current Action* structure is filled with the details regarding the Landing Action. The *Commands* part of the output structure is set as can be seen in Table 6.14; the Direct mode of the autopilot is used, with the *Direct Speed* value set to the Landing Speed value (item 4.2.6 in Table 6.4), the *Direct Pitch* value set to the *Landing Angle* value, the *Direct Yaw* value set to the *Runway Heading* (the *Heading* property of the Action), the *Brakes* value set to 0 (brakes inactive), the *Direct/Auto* value set to 0 (autopilot in Direct mode) and the *Mission Time* value taken from the *input-link*. The *set-path* operator fires only once, at the beginning of the landing manoeuvre; it is followed by iterative firings of the *keep-going* operator, until the *flare* operator fires.

Table 6.14. Commands output for the Landing Action

Type	Definition	Value
Direct commands	Direct Speed	Landing Speed
	Direct Pitch	Landing Angle
	Direct Yaw	Runway Heading
	Direct Roll	-
	Brakes	0
Auto commands	Auto Speed	-
	Auto Altitude	-
	Initial Position	-
	End Position	-
	Direct/Auto	0 (Direct)
Payload Commands	Power-On	-
	Use Recon	-
	Use Attack	-
	Mission Time	From *input-link*

The *keep-going* operator has the same function of the *keep-travelling* operator in the *travel* substate: it executes repeatedly to allow continuous cycling of the agent, updating the *Mission Time* output. It works as the link between other operators, since it is while this operator is firing that the UAV is actually performing manoeuvres; the other operators fire when a part of the manoeuvre is completed.

The *flare* operator fires when the UAV's altitude above the ground becomes lower than the Flare Altitude (item 4.3.7 in Table 6.4). At this point the UAV must begin the flare manoeuvre, during which it will acquire the correct landing attitude (typically a slightly positive pitch angle) and reduce its rate of descent. The Flare Altitude is currently set at 10 m. The *flare* operator changes the *Mission Time* (updating it as normal), *Direct Speed* and *Direct Pitch* values. *Direct Speed* is set to half the Landing Speed value (resulting in a null thrust command) and *Direct Pitch* is set to the *Flare Angle* value (item 4.4.8 in Table 6.4), which will normally be slightly positive. Like the *set-path* operator, the *flare* operator fires once and is followed by the *keep-going* operator.

The *touch-down* operator fires when the UAV's altitude above the ground becomes almost zero (if weight-on-wheel sensors are available, these would be used as the condition to fire the operator). At this point the UAV is on the ground and must decelerate, so thrust must be null and brakes must be activated. The *touch-down* operator changes the *Mission Time* (updating it as normal), *Direct Speed*, *Direct Pitch* and *Brakes* values. *Direct Speed* is set to zero, *Direct Pitch* is set to zero and *Brakes* is set to 1, to indicate activation of the brakes. Note that the *Direct Yaw* command is not changed, however it must be applied by ground steer commands rather than the normal heading-hold loop. Like the *set-path* operator, the *touch-down* operator fires once and is followed by the *keep-going* operator.

The *finish-landing* operator fires after all other operators in the substate, when the UAV has completely stopped after the landing. The *finish-landing* operator performs two tasks: first, it cancels all working memory elements from the *output-link*, so that it will be empty when the next Action is being performed; secondly, it increases the value of the *current action counter* in the top state by 1. This will in turn cause the *landing* substate to close, and a new substate in the *action* state will be entered.

The sequence of operators in the *landing* substate is similar to that shown in figure 6.4 for the *taxi* substate; in the comparison, the *calculate-heading* operator in *taxi* is equivalent to the *calculate-angle* operator in *landing*, the *steer* operator in *taxi* is equivalent to the *set-path* operator in *landing*, the *move* operator in *taxi* is equivalent to the *flare* operator in *landing* and the *stop* operator in *taxi* is equivalent to the *touch-down* operator in *landing*. The *keep-going* operators perform the same function in both substates.

6.3 Exag testing strategy

The testing campaign conducted on the Execution Agent of the SAMMS architecture is also aimed at testing the functionality of the entire architecture. The full range of components, as described in Section 3.2, will be used.

Figure 6.13 shows the SAMMS architecture overview: summarizing, scenario information is fed to the Planner Agent and Mission Manager Agent, which derive a flight plan fusing scenario information with real-world information. Real-world information is provided by the UAV model; the combination of scenario information and real-world information is used by the New-Plan Trigger function to trigger the generation of new plans. The generated flight plan is converted by the Execution Agent into low-level commands, which can be applied by the Autopilot and Payload Management systems. These in turn feed into the UAV model, which consequently updates real world information, closing the cycle.

The Planner Agent and the Mission Manager Agent (and their relationship) have been thoroughly treated in Chapters 4 and 5. The Execution Agent was described in the first two sections of this chapter, but its functionality must still be proven. In order to work, the Exag needs a full flight plan as input, so it was decided that Exag tests should be carried out together with the Planner. The MMA is not necessary for the Planner/Exag combination to work (SAMMS is designed to allow the exclusion of the MMA), however its presence will not impact testing of the Exag, since the only consequence of MMA intervention is a change in the flight plan, and the Exag should be able to execute any plan which is derived by the Planner.

During the Execution Agent testing campaign, the entire SAMMS architecture will be tested using the scenarios which were used for the Planner Agent and Mission Manager Agent testing campaigns (described in Chapters 4 and 5). While during these campaigns the flight plan generation aspect was highlighted, for the Exag campaign the application of flight plans will be highlighted. Through the UAV model, actual flight trajectories will be calculated, and these will be used to demonstrate that the system is capable of correctly guiding the simulated UAV model.

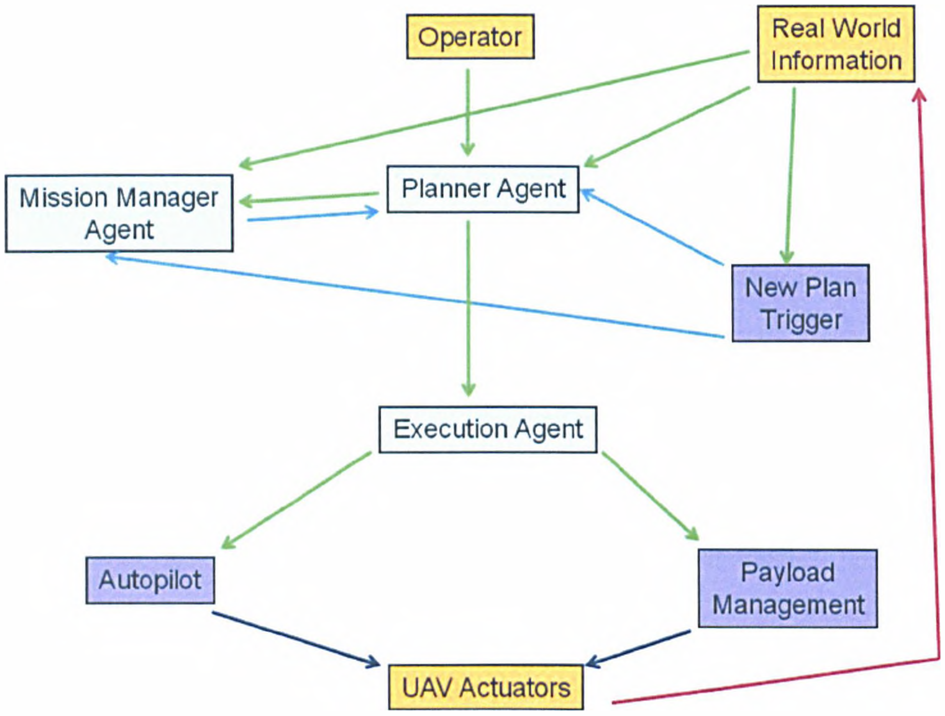


Figure 6.13. Architecture overview.

Since the goal of the testing campaign is not only to demonstrate functionality of the Exag, but also of entire SAMMS architecture, two main types of tests will be accomplished (in fact, it is more precise to talk about two different methodologies for test data analysis). The first type is dedicated at validating the Execution Agent; for each Action type which can be converted by the Exag into low-level commands, a simulated trajectory will be shown. This type of test is further described in Section 6.3.1 and will be shown in Section 6.4. The second type of tests is dedicated at validating the entire SAMMS architecture; in this case, the scenarios which were used during the Planner and MMA testing campaigns (described in Section 4.3 and 5.3) will be applied to the whole architecture, verifying how the Exag and the low-level control systems accomplish the generated flight plans. This type of tests is further described in Section 6.3.2 and will be shown in Chapter 7.

Combining these two types of tests and methodologies for test data analysis, it is hoped to demonstrate that the SAMMS architecture is capable of generating flight plans, executing them and update them when needed, all without external supervision.

6.3.1 Visualization of Action trajectories

The main task of the Execution Agent is to translate high-level mission management abstractions such as Actions into the low-level practical commands that can be issued to an autopilot system. To verify that this is done correctly, a part of the testing campaign was focused on highlighting how each Action was carried out, thus verifying not only the functionality of the Exag or of the low-level commands, but also of the two integrated components.

In practice, these tests are carried out using the same scenarios that will be used in the other test type; however, the focus will be placed on single Actions. For each Action type, a simulated trajectory will be shown, together with the plots of meaningful state variables (such as the UAV speed and attitude). These plots will be correlated with the Execution Agent description, highlighting when the various agent operators fire and what their effect is.

The tested Action types are: Taxi, Take-off, Climb, Travel, Target-Recon, Target-Attack, Circle, Descent and Landing.

6.3.2 Choice of scenarios

Since the Execution Agent cannot be tested on its own (because this would require simulating a full flight plan, which is unpractical), the Exag testing campaign uses the entire SAMMS architecture.

The Planner Agent and the Mission Manager Agent were extensively tested in Chapters 4 and 5; because of the large amount of input variables, it is practically impossible to test every possible input combination. Instead, a set of meaningful scenarios was prepared, and the flight plans obtained for these scenarios were displayed.

For the Exag testing campaign, all of the scenarios which were previously used during the Planner and MMA testing campaigns (see Section 4.3 and 5.3) will be applied to the entire SAMMS architecture. It will then be possible to obtain simulated flight trajectories and state variable plots.

Many of the scenarios included several variations that were designed to test particular algorithms within the Planner or MMA; while all of these variations were used during the testing campaign, only a selection will be actually portrayed in this thesis. This is to avoid repetition, since most scenario variations only present minor differences.

The selection of scenario variations is aimed at demonstrating the widest array of SAMMS capabilities. Demonstrated scenarios include a variation of each scenario, excluding Scenario 6 which involves very large distances and consequently would result in very long simulation times. In particular, the results for the following scenario variations will be shown: scenario variation 1c (see Section 4.4.1), scenario variation 2e (see Section 5.4.2), scenario variation 3c (see Section 4.4.3), scenario variation 4c (see Section 4.4.4), scenario variation 5e (see Section 4.4.5) and scenario 7 (see Section 4.4.7). Most of these are chosen among the Planner testing campaign, however one scenario (variation 2d) is taken from the MMA testing campaign. For each of these scenario variations, the entire mission is simulated and the corresponding flight trajectory and state variable plots displayed.

6.4 Action type visualization

Figure 6.14 shows the Simulink block diagram that was used for the Exag testing campaign, both for the visualization of Action trajectories (displayed in this section) and for the simulation of entire scenarios (displayed in Section 6.5).

For the purposes of visualization of Action trajectories, the data obtained from these simulations will be plotted in several graphs; typically, for every Action type the following plots will be shown:

- Two-dimensional trajectory plot (North/East coordinates)
- Three-dimensional trajectory plot (North/East/Altitude coordinates)
- Time/Altitude plot
- Time/Speed plot
- Time/Heading plot

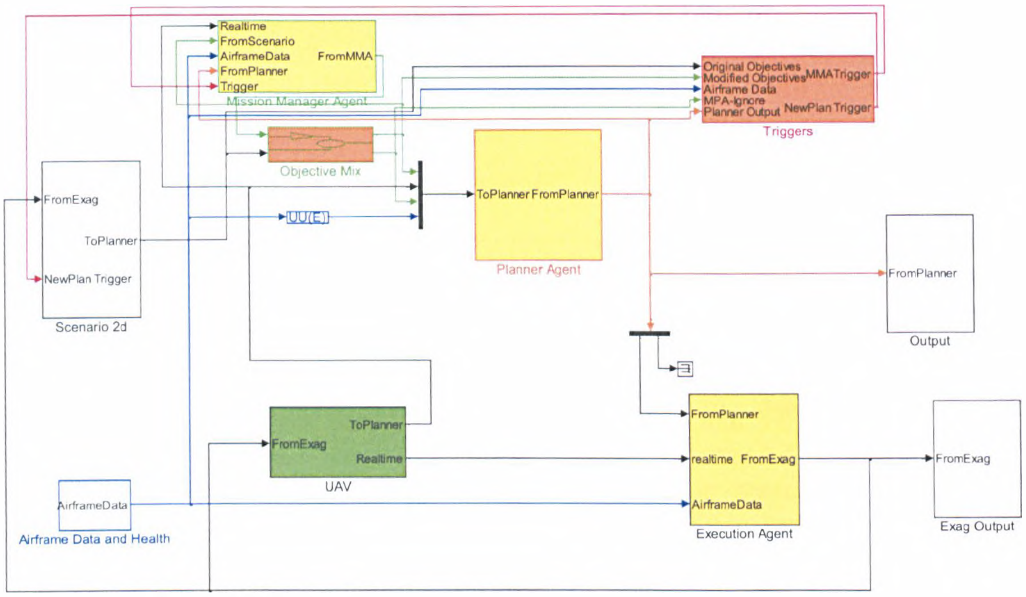


Figure 6.14. SAMMS top level Simulink diagram, used for the Exag testing campaign.

Through these, it will be possible to demonstrate that the flight trajectory accomplished by SAMMS for each Action type is realistic. Depending on the particular Action type, certain plots might not be needed, additional ones might be necessary and specific information will have to be provided.

6.4.1 Taxi

The Taxi Action type is used to move the UAV while it is still on the ground; its Exag implementation is described in Section 6.2.3.

The visualized data is obtained from a scenario where the UAV is taking off from Sheffield Airport; hence, the initial *Parking Position* is set at coordinates [53°23'42''N - 1°22'51''W]. These coordinates constitute the origin of the North/East reference system used in the UAV model (the one described in Section 3.4).

Thus, in the North/East reference system, the *Parking Position* is situated at coordinates [0, 0]; the *Runway Position* is instead at coordinates [53°23'37''N - 1°22'48''W],

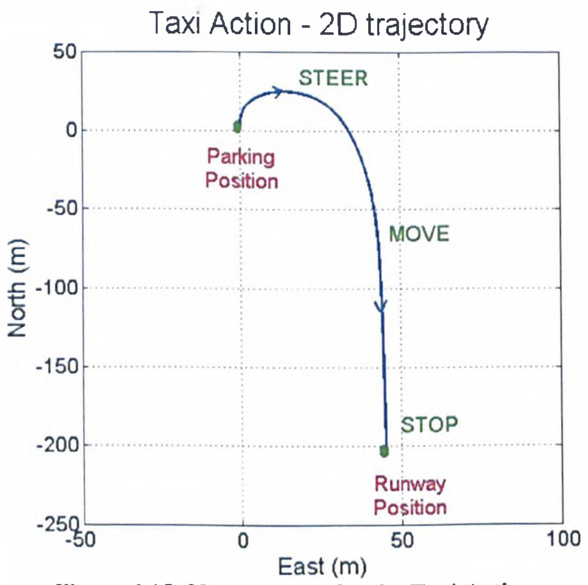


Figure 6.15. 2D trajectory for the Taxi Action

which corresponds to coordinates [-204, 45] in the North/East reference system (these are expressed in metres).

For the Taxi Action type, it is not necessary to show the 3D trajectory and Time/Altitude plots. Figure 6.15 shows the 2D trajectory covered by the UAV during the Taxi Action; the UAV starts with a northerly heading so it has to perform a long steering manoeuvre (*steer* operator) since the Runway Position is in the South-South-East direction. Once a correct heading has been achieved, the *move* operator fires, increasing movement speed until the *Runway Position* is reached; at this position, the *stop* operator stops the UAV in preparation for Take-off.

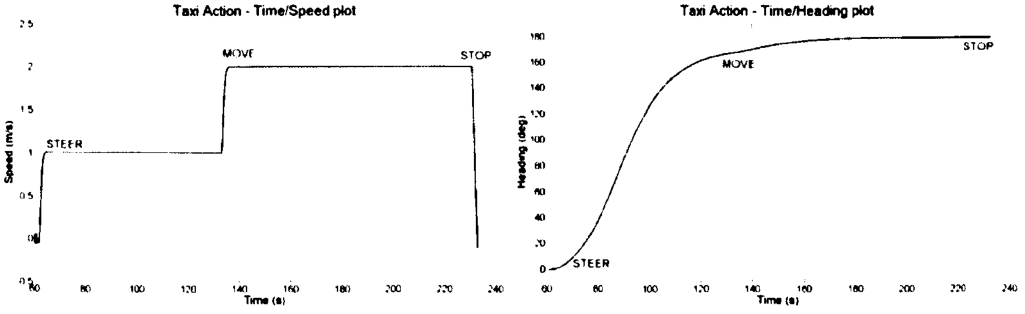


Figure 6.16. Time/Speed and Time/Heading plots for the Taxi Action.

Figure 6.16 shows the Time/Speed and Time/Heading plots for the Taxi Action. It is possible to notice that the *steer* operator moves the UAV at 1 m/s, while the *move* operator moves it at 2 m/s. Also, the heading changes rapidly during the *steer* manoeuvre and is only slightly adjusted during the *move* manoeuvre; on the plot, it is possible to see the adjustment in heading calculation that happens when the *move* operator fires as a slight change in the steepness of the heading plot.

6.4.2 Take-off

The Take-off Action type is used to get the UAV airborne starting from its initial grounded condition; its Exag implementation is described in Section 6.2.4.

The visualized data represents the continuation of the mission from which the Taxi Action data was obtained (see Section 6.4.1). Hence, the UAV starts from the *Runway Position* (coordinates [53°23'37"N - 1°22'48"W] or [-204, 45] in the North/East reference system). The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42"N - 1°22'51"W].

Figure 6.17 shows the 2D trajectory covered by the UAV during the Take-off Action; during the Taxi Action, the UAV reached the Runway Position with a heading different from the Runway Heading, so it has to perform a steering manoeuvre (*steer* operator). Once the Runway Heading has been achieved, the *accelerate* operator fires, keeping the UAV on the runway and increasing speed until the *Take-off Speed* is reached; at this speed, the *rotate* operator fires pitching the UAV up so as to gain a positive rate of climb.

During take-off, the UAV model must transition from the ground condition to the airborne condition; because of the simplifications applied to the model, this transition is not seamless. In particular, two problems are present: firstly, proper modelling of the grounded condition would require the calculation of an entirely different set of equations of motion; secondly, the aerodynamics model is not precise for low-speed high-angle-of-attack conditions (e.g. take-off and landing). These limitations are common to all simplified aircraft models: accurate modelling of ground operations and ground-level aerodynamics can only be achieved at considerable cost, thus it is only done by aircraft builders (who will have the most accurate aerodynamics data available) and by the developers of advanced simulation platforms (for example, for commercial pilot training). For the purposes of this thesis and the SAMMS architecture, such

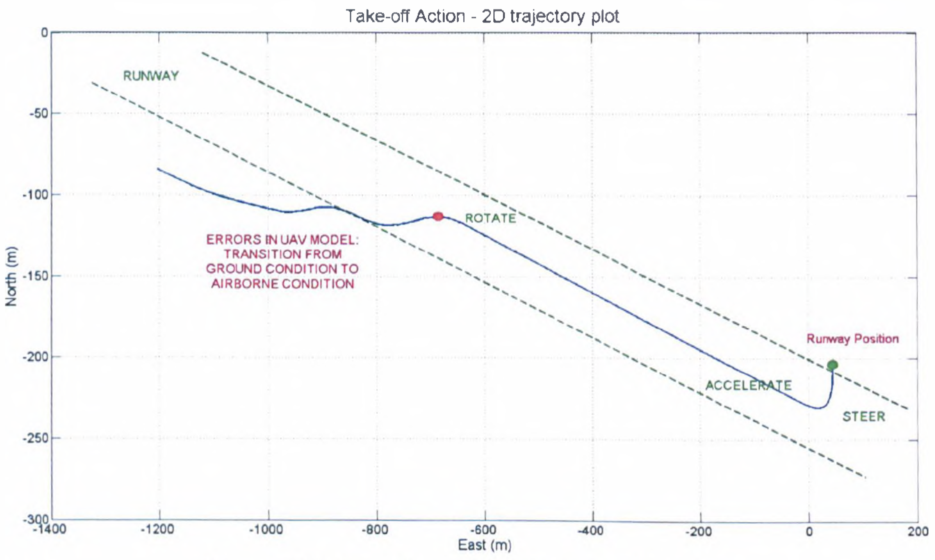


Figure 6.17. 2D trajectory for the Take-off Action.

advanced simulation was not readily available and was deemed unnecessary to demonstrate the system. Therefore, simulation errors during the take-off and landing phases are accepted, as long as they do not impact the rest of the simulation; as will be

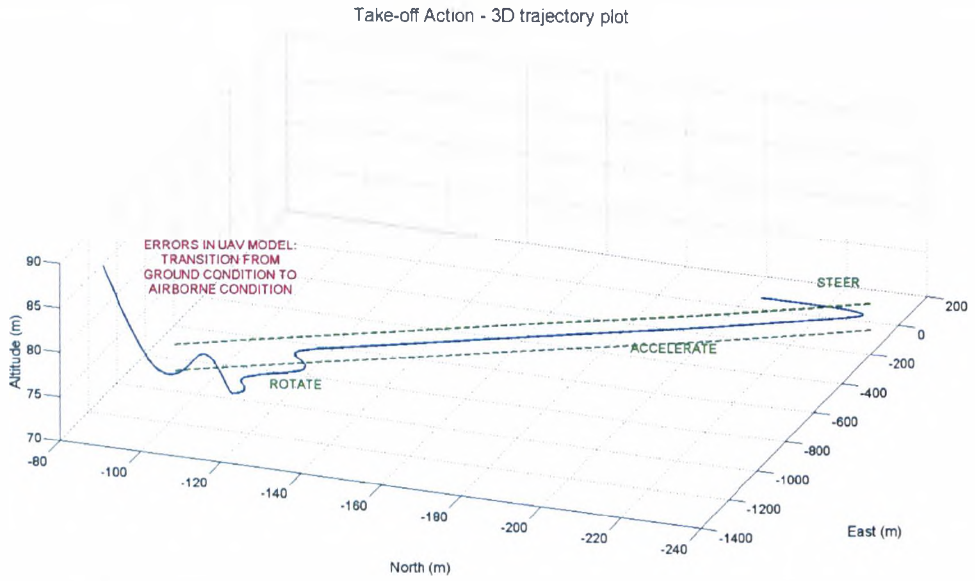


Figure 6.18. 3D trajectory for the Take-off Action.

seen in the following sections, the UAV model is capable of recovering from the errors introduced during take-off. However, the errors are highly visible in the graphical plots shown for the Take-off Action, and particularly in Figures 6.17, 6.18 and 6.19.

Figure 6.18 shows the 3D trajectory covered by the UAV during the Take-off Action; the trajectory remains on the ground level until the position where the *rotate* operator fires and the UAV acquires a positive rate of climb. It is possible to see that the UAV model errors heavily affect this phase; the UAV is subject to heavy oscillations on both the directional and the longitudinal axis. This can also be observed in Figure 6.17, where the directional oscillations are highlighted.

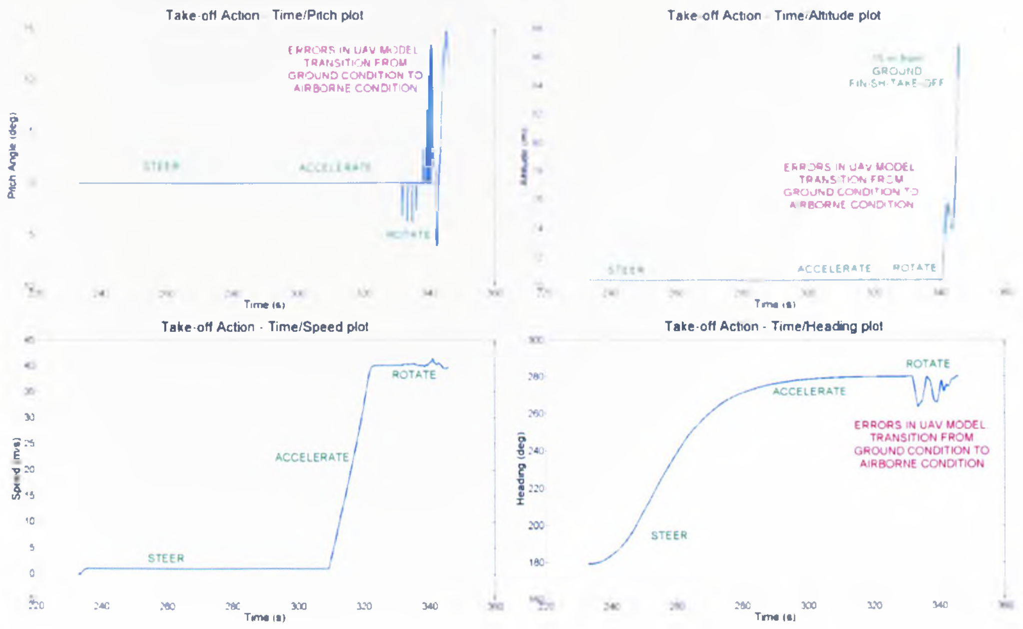


Figure 6.19. State variable plots for the Take-off Action.

Figure 6.19 shows four plots of state variables during the Take-off Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle plot is the one for which the UAV model errors are more evident; as soon as the UAV detaches from the ground (which can happen before the *rotate* manoeuvre), heavy pitch angle oscillations can be noticed. However, it is possible to see that the oscillations are eventually cancelled; the final decrease in pitch angle which can be seen in the plot is in fact due to the activation of the Climb Action, during which the commanded pitch angle is inferior to the take-off pitch angle. The Time/Altitude plot is useful in showing that the Action is terminated when 15 m from the ground are reached. The Time/Speed plot is not affected heavily by the UAV model errors; the UAV steers at 1 m/s then accelerates until 40 m/s, which is the Take-off Speed. The Time/Heading shows that the Runway Heading is kept until rotation, at which point the UAV model errors can be noticed.

6.4.3 Climb

The Climb Action type is used to take the UAV from the 15 m from ground altitude to a safe altitude where the main mission can start; its Exag implementation is described in Section 6.2.5. It is to be noted that while the main mission is executed using the Auto mode of the autopilot, the Climb Action still makes use of the Direct mode.

The visualized data represents the continuation of the mission from which the data for the Taxi and Take-off Actions was obtained (see Sections 6.4.1 and 6.4.2). Hence, the UAV starts from the position reached at the conclusion of the Take-off Action. The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23’42’’N - 1°22’51’’W].

Figure 6.20 shows the 2D trajectory covered by the UAV during the Climb Action; the first operator to fire is *climb-attitude*. Normally, the UAV will keep the Runway

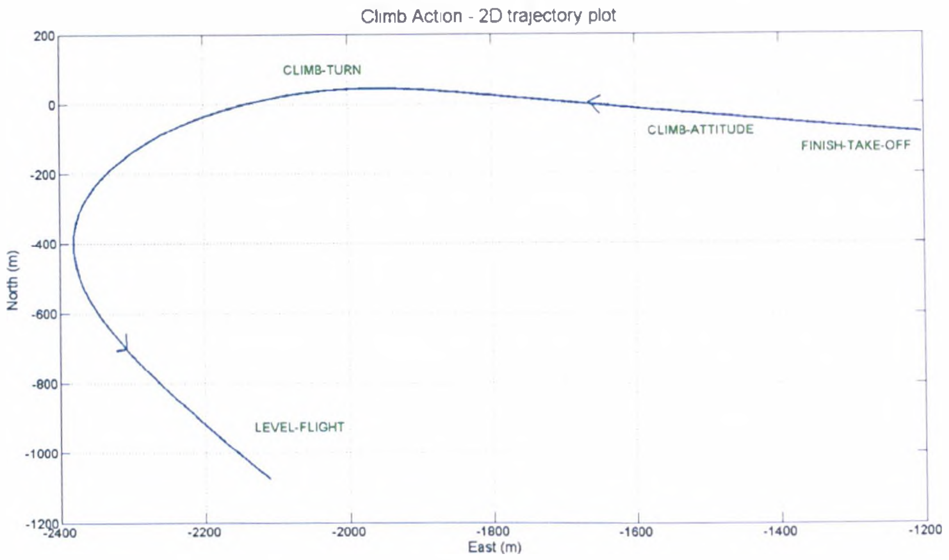


Figure 6.20. 2D trajectory for the Climb Action.

Heading during the Take-off Action, and this heading is kept for the first part of the Climb Action. The UAV is set with a preset pitch attitude that will allow it to achieve a positive rate of climb (usually only slightly lower than the maximum achievable rate of climb). The Climb Action is mostly performed using maximum thrust; this is reduced only when the *level-flight* operator fires. At a specified altitude, the *climb-turn* operator

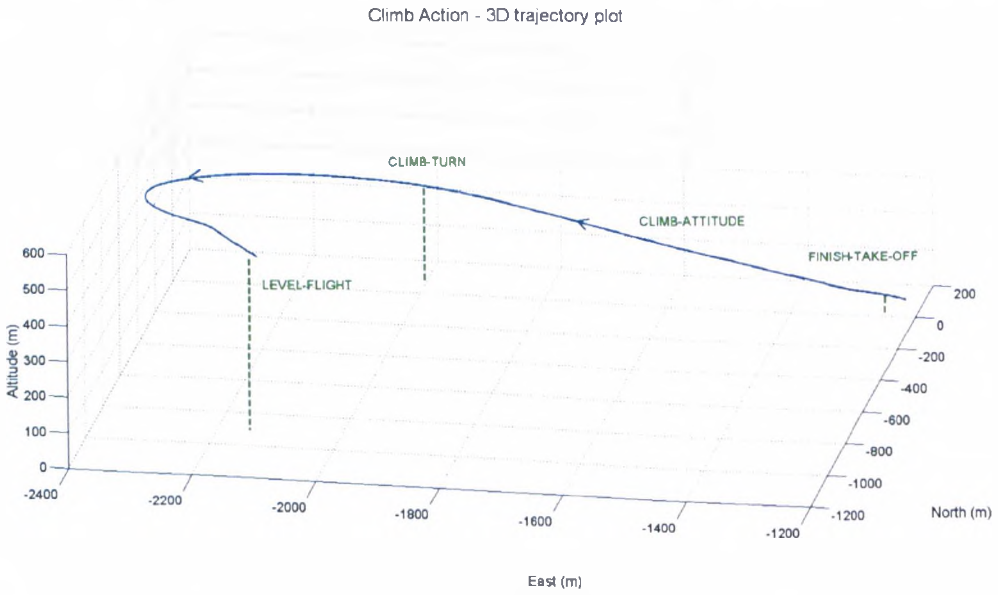


Figure 6.21. 3D trajectory for the Climb Action.

fires and directs the UAV towards the position of the first Objective that is scheduled in the flight plan; the resulting turn is clearly visible in the plot. The UAV finally reaches the Mission Start Altitude, where the *level-flight* operator fires.

Figure 6.21 shows the 3D trajectory plot for the Climb Action; this is particularly significant since the change in altitude is the ultimate goal for the Action. It is possible to see that the ascending attitude is kept throughout the Action, until the *level-flight* operator fires. This operator puts the UAV into an almost horizontal path; since the Direct mode of the autopilot is still in use, the altitude-hold loop is not used and a truly horizontal trajectory cannot be guaranteed.

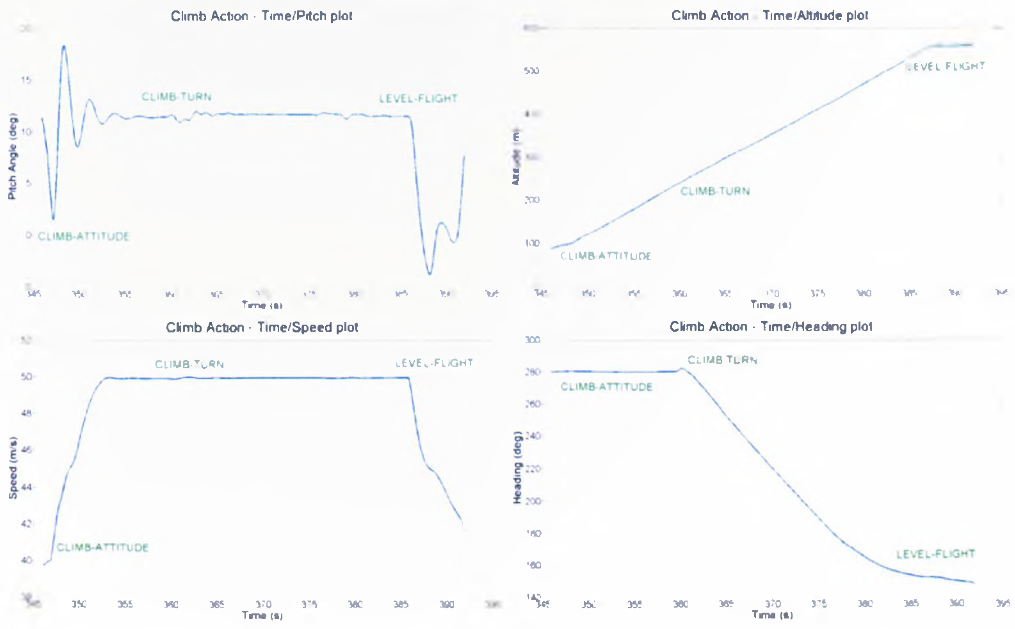


Figure 6.22. State variable plots for the Climb Action.

Figure 6.22 shows four plots of relevant state variables for the Climb Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle shows the errors caused by the ground to airborne transition gradually disappearing; the pitch angle is initially oscillating, but the oscillations are damped after roughly 10 sec. Since the Direct mode of the autopilot is used, a constant pitch angle is demanded; in the simulations, this is set to 0.2 rad, roughly equivalent to 11.5 deg. The pitch angle is influenced by UAV manoeuvres and it is possible to notice the effect of a turn (which pitches the UAV down). The level-flight operator still uses the Direct autopilot mode, however this seems an area for improvement; because of the drastic change in aircraft attitude, an oscillation is induced. It is to be noted that at this stage the UAV is transitioning for the Direct mode of the autopilot to the Auto mode, which uses smoother control laws. The Time/Altitude plot shows that a linear climb profile is maintained throughout the manoeuvre. The Time/Speed plot shows that the UAV model is capable of maintaining the maximum speed of 50 m/s despite the steep climb; should the UAV lack the power to do so, the actual UAV speed would not equal the Maximum Speed, but the Speed-Hold loop set with a Maximum Speed target would ensure that maximum thrust is used. The UAV is slowed down to Cruise Speed (40 m/s) by the *level-flight* operator. The Time/Heading plot shows that the UAV maintains the original runway heading until the climb-turn operator fires, then it performs a turn towards the first Objective.

6.4.4 Travel

The Travel Action type is the most common Action type; it is used to move the UAV from its current position to a destination position. This is normally carried out at the optimal cruise conditions (cruise altitude and cruise speed) along a great-circle route connecting the current position and the destination position. Any UAV movement can be expressed as a series of Travel Actions. Details about the Exag implementation of Travel Action can be found in Section 6.2.6. It is to be noted that while the Taxi, Take-off and Climb Actions are executed using the Direct mode of the autopilot, the Travel Action makes use of the Auto mode.

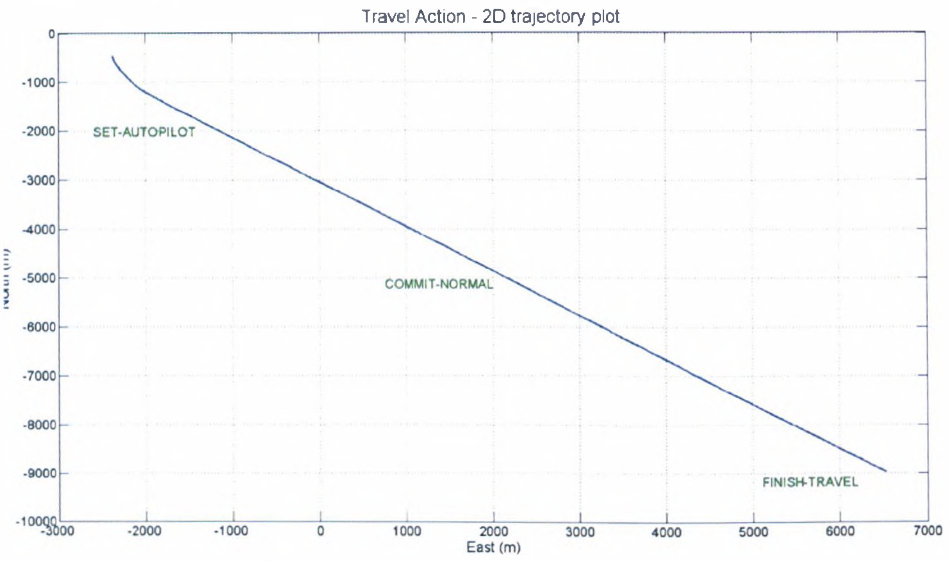


Figure 6.23. 2D trajectory for the Travel Action.

The visualized data represents the continuation of the mission from which the data for the Taxi, Take-off and Climb Actions was obtained (see Sections 6.4.1, 6.4.2 and 6.4.3). Hence, the UAV starts from the position reached at the conclusion of the Climb Action. The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

Figure 6.23 shows the 2D trajectory covered by the UAV during a Travel Action; it is important to note that while Actions related to the take-off phase are executed only once during a mission, normally several different Travel Actions will be executed during a mission, each originating and ending at different positions. The first operator to fire is *set-autopilot*; the Travel Action is started with the UAV exiting the Climb Action. With standard Climb parameters, the UAV will be flying towards the first Objective at the Mission Start Altitude. In the plot, the final phases of the Climb Action are also shown; the transition to the Travel Action is seamless, with the autopilot adjusting the route towards the first Objective and climbing to Cruise Altitude. The UAV then travels in a straight path towards its destination. After half of the planned travel distance has been covered, the *commit-normal* operator fires; thus, the current Objective will be completed even if a re-planning event occurs. When the destination position is reached, the *finish-travel* operator fires and terminates the Action.

Figure 6.24 shows the 3D trajectory plot for the same Travel Action; this is very similar to the 2D plot, since most of the Action occurs at the cruise conditions. The initial part of the Action can be noted, since it is possible to see that the UAV achieves level flight at the end of the Climb Action then starts climbing again until the Cruise Altitude is reached.

Figure 6.25 shows four plots of relevant state variables for the Travel Action: the Time/Altitude plot, the Time/Speed plot, the Time/Roll Angle plot and the Time/Heading plot. In the Time/Altitude plot, it is possible to see how the UAV exits

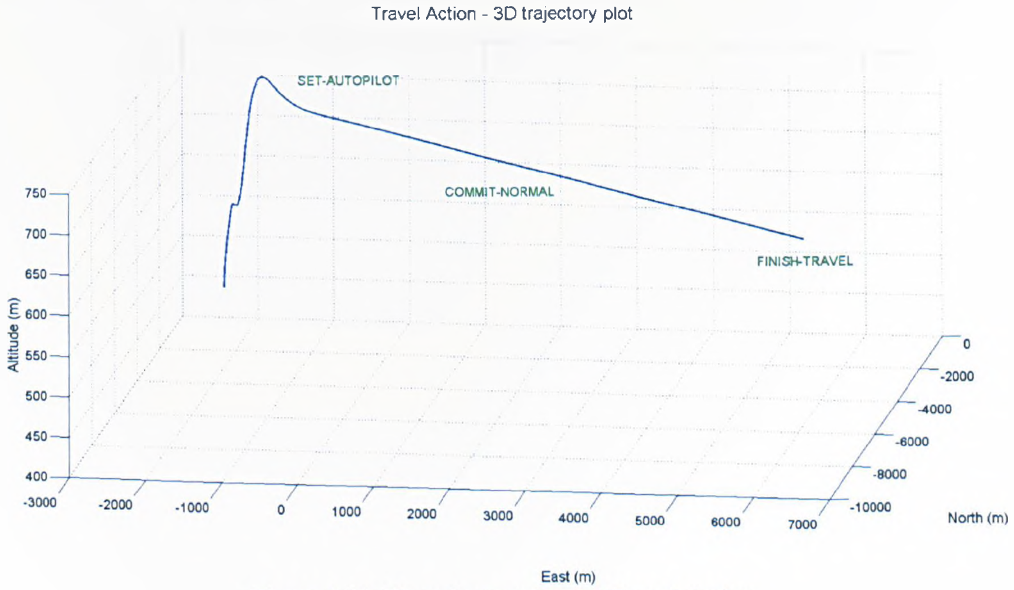


Figure 6.24. 3D trajectory for the Travel Action.

from the Climb Action by achieving level flight then starts climbing again so as to reach Cruise Speed. The Time/Speed plot shows how the Climb Action is performed at 50 m/s (Maximum Speed) while the Travel Action is performed at 40 m/s. The Time/Roll Angle plot is shown to provide an example of how the UAV achieves directional

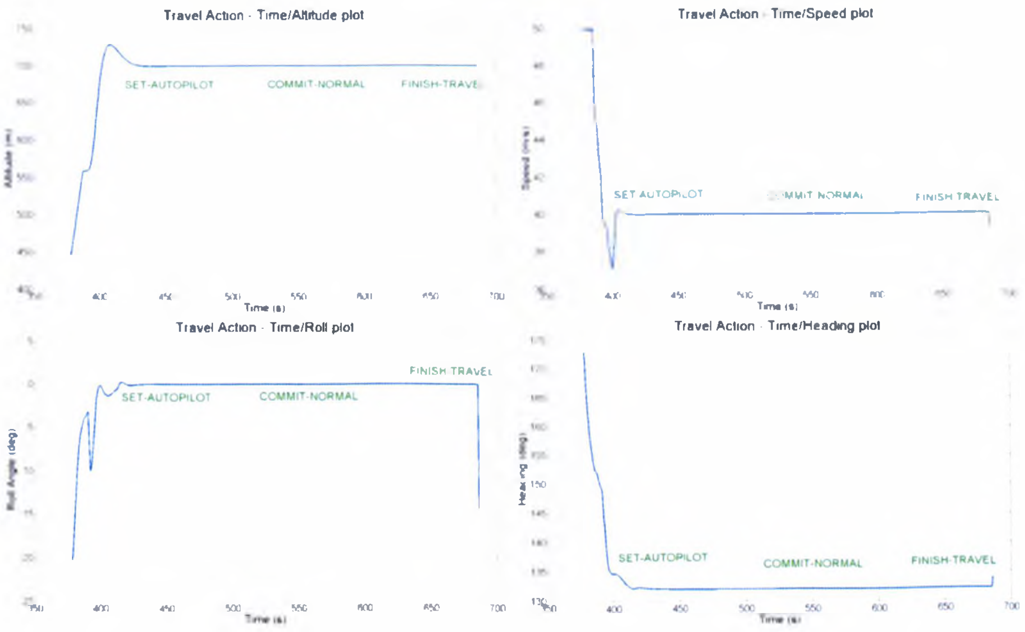


Figure 6.25. State variable plots for the Travel Action.

navigation. This can be linked to the Time/Heading plot: the UAV changes heading by commanding a roll angle (see details of roll-hold and heading-hold loops in Section 3.5). Due to the short distance covered during this Travel Action, it is not possible to notice that a great circle route is being used. Should longer distances be travelled, it would be possible to see that the heading is constantly changing (unless travelling along a meridian, see Section 6.2.6).

6.4.5 Target-Recon

The Target-Recon Action type is used to manoeuvre the UAV above a target so that a reconnaissance payload can be activated. Its Exag implementation is described in Section 6.2.7; a parameterized manoeuvre is accomplished. The parameters are usually set so as to bring the UAV close to the ground and perform a pass directly above the target. The Target-Recon Action makes use of the Auto mode of the autopilot.

The visualized data is taken from a Target-Recon Action performed during the same simulated mission from which the data for the Taxi, Take-off, Climb and Travel Actions was taken. The UAV starts from the position reached at the conclusion of a Travel Action (not the one shown in Section 6.4.4). The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

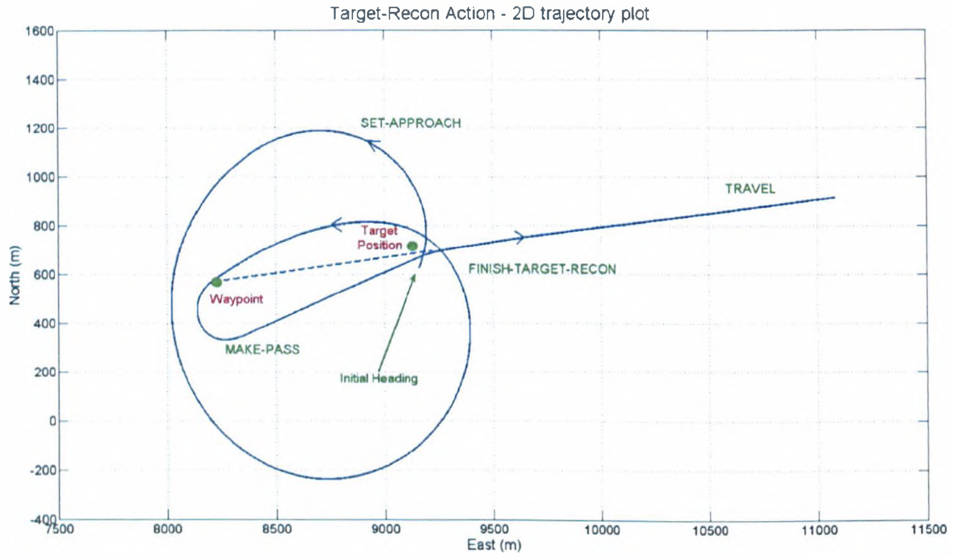


Figure 6.26. 2D trajectory for the Target-Recon Action.

Figure 6.26 shows the 2D trajectory covered by the UAV during the Target-Recon Action; in this case, the target is located at position [53°25'05''N - 1°14'33''W], corresponding to North/East coordinates [680, 9185]. The Target-Recon Action starts with the UAV arriving from the previous Objective; the first operators to fire are *select-approach* and *set-approach*. The *select-approach* operator calculates the position of the *Waypoint*, which is set at the Recon Distance from the Target Position, in the direction opposite from the direction corresponding to the next Objective in the flight plan. This distance is currently set at 1000 m, and the Waypoint is at coordinates [538, 8195], which can be easily demonstrated to be at 1000 m from the Target Position. The *set-approach* operator commands the UAV to reach the Waypoint and the desired Recon Altitude, which is set to 100 m above the ground. Due to the steep descent, during which the UAV gathers high speed, the UAV is unable to perform a turn tight enough to pass over the Waypoint at the first try; therefore, the UAV trajectory involves two circles being flown during the manoeuvre. At the second try, the Waypoint is reached at a low speed and at the correct altitude, the *make-pass* operator fires and commands the UAV to travel towards the Target Position. When the Target Position is finally reached, the *finish-target-recon* operator fires and terminates the Action; meanwhile the payload will have been performing its task. In all the plots related to the Target-Recon Action, the initial part of the following Travel Action is shown; on the 2D trajectory plot, it is possible to see how the UAV passes over the Target and then continues directly towards the next Objective.

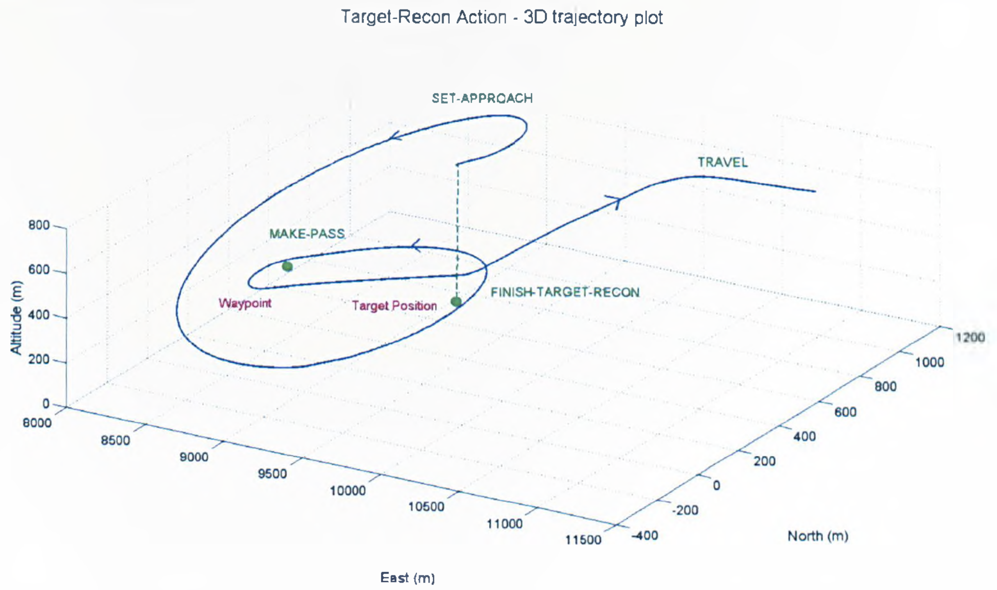


Figure 6.27. 3D trajectory for the Target-Recon Action.

Figure 6.27 shows the 3D trajectory plot for the Target-Recon Action. Comparing it to Figure 6.26, it is possible to notice the descending path adopted to reach the Recon Altitude. Also, after the Target-Recon Action is completed a Travel Action begins and this results into an ascending path that brings the UAV back to the Cruise Altitude.

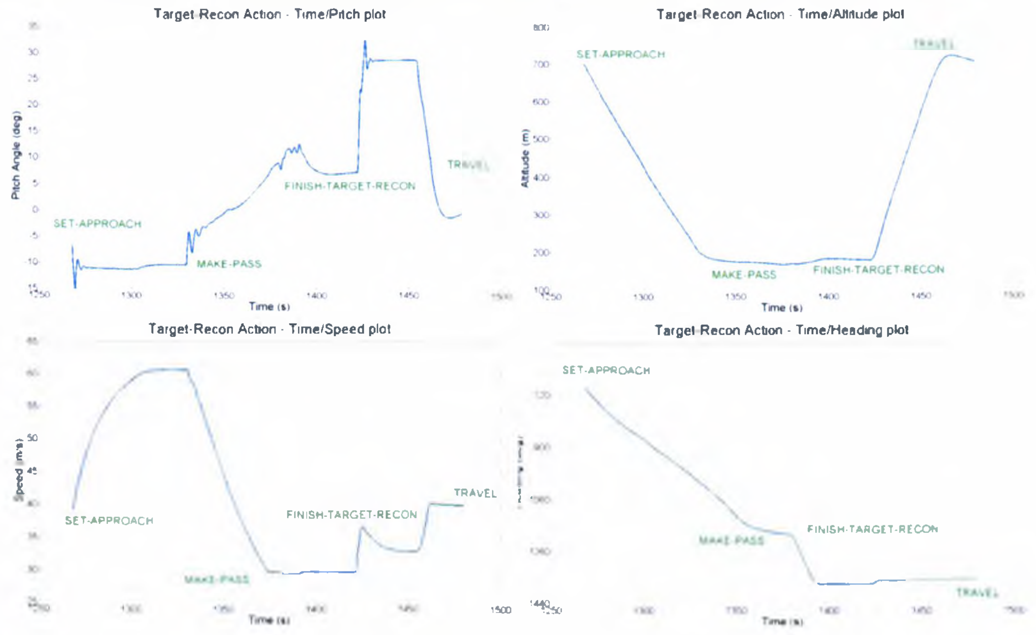


Figure 6.28. State variable plots for the Target-Recon Action.

Figure 6.28 shows four plots of relevant state variables for the Target-Recon Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the

Time/Heading plot. From these, it is possible to notice that the UAV descends adopting a steep negative pitch angle; this results in a Speed increase, up to 60 m/s (the altitude-hold loop integrates a speed limiter algorithm, see Section 3.5.2). Due to the high speed, the UAV cannot perform tight manoeuvres while descending, hence the need to cover two "circles". When the UAV has reached the Recon Altitude (185 m = 100 m above + 85 m ground level), speed lowers and manoeuvrability is increased. The *Waypoint* is correctly reached and then the actual pass over the target can be performed. The UAV then climbs back to the Cruise Altitude while performing the following Travel Action; the climb is performed at a steep pitch angle which results in the UAV being unable to maintain its desired speed (which is the Cruise Speed) even though Maximum Thrust is used.

6.4.6 Target-Attack

The Target-Attack Action type is used to manoeuvre the UAV above a target so that a weapon payload can be delivered. Its Exag implementation is described in Section 6.2.7; the manoeuvre is very similar to the Target-Recon manoeuvre, however different parameters are used. While the description of the Target-Attack Action directly referred to the Target-Recon Action description, a different set of plots from the Target-Recon ones will be shown in this section in order to demonstrate how the different parameters affect the manoeuvre. The Target-Attack Action makes use of the Auto mode of the autopilot.

The visualized data is taken from a Target-Attack Action performed during the same simulated mission from which the data for the Taxi, Take-off, Climb and Travel Actions was taken. The UAV starts from the position reached at the conclusion of a Travel Action (not the one shown in Section 6.4.4). The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

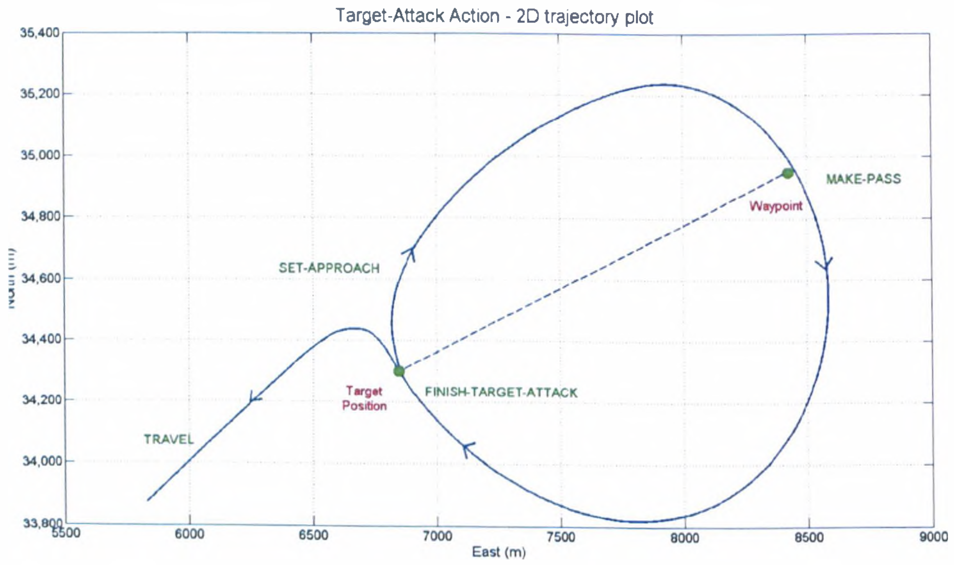


Figure 6.29. 2D trajectory for the Target-Attack Action.

Figure 6.29 shows the 2D trajectory covered by the UAV during the Target-Attack Action; in this case, the target is located at position [53°42'15''N - 1°16'41''W], corresponding to North/East coordinates [34300, 6853]. The Target-Attack Action starts with the UAV arriving from the previous Objective; the first operators to fire are *select-*

approach and *set-approach*. The *select-approach* operator calculates the position of the *Waypoint*, which is set at the Attack Distance from the Target Position, in the direction opposite from the direction corresponding to the next Objective in the flight plan. This

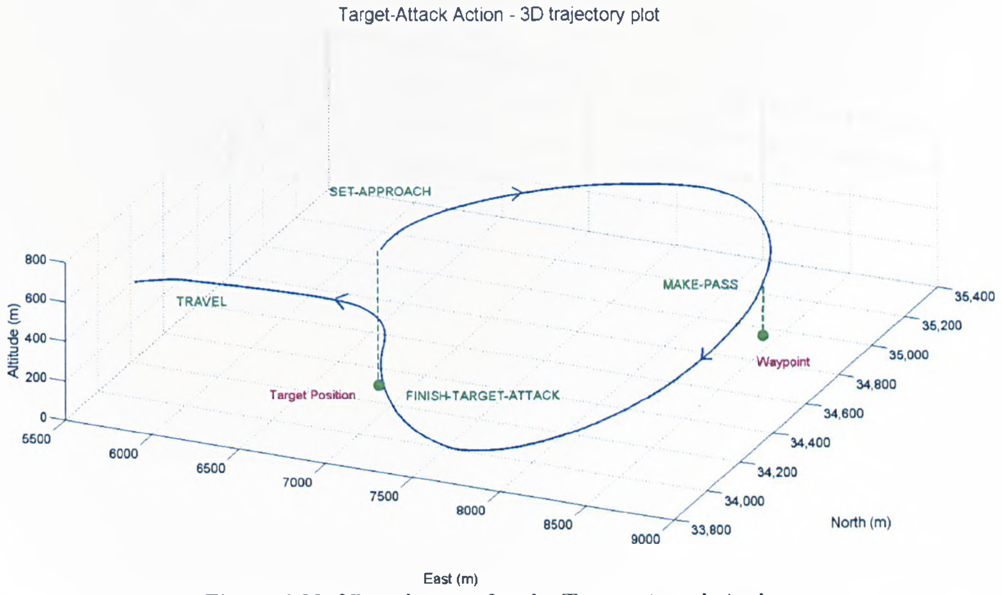


Figure 6.30. 3D trajectory for the Target-Attack Action.

distance is currently set at 1700 m, and the Waypoint is at coordinates [34955, 8422], which can be easily demonstrated to be at 1700 m from the Target Position. The *set-approach* operator commands the UAV to reach the Waypoint and the desired Attack Altitude, which is set to 150 m above the ground. In contrast with the Target-Recon Action shown in Section 6.4.5, the UAV is able to pass over the Waypoint at the first

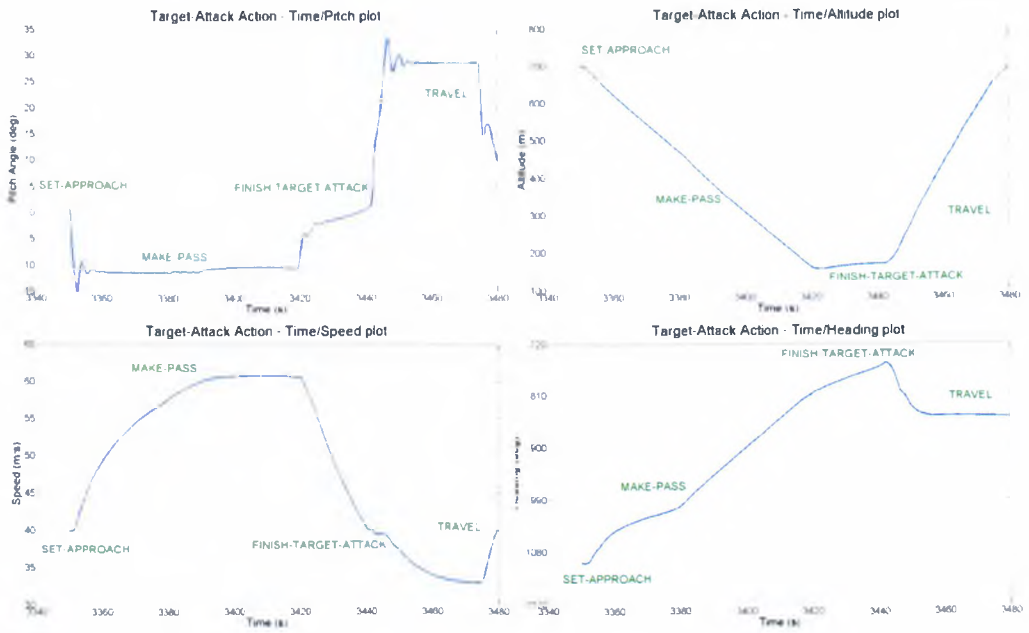


Figure 6.31. State variable plots for the Target-Attack Action.

try; the *make-pass* operator fires and commands the UAV to reach the Target Position. However, the desired altitude is not yet reached at this point, thus flight speed is still high (because the UAV is descending) and the turn towards the target is not tight; the manoeuvre is accomplished correctly, however a significant turn towards the next Objective is required at the end of the Action. When the Target Position is finally reached, the *finish-target-attack* operator fires and terminates the Action; meanwhile the payload will have been performing its task. In all the plots related to the Target-Attack Action, the initial part of the following Travel Action is shown; on the 2D trajectory plot, it is possible to see how the UAV passes over the Target and then turns towards the next Objective.

Figure 6.30 shows the 3D trajectory plot for the Target-Attack Action, which clearly shows the smooth descent path covered by the UAV during a Target-Attack Action. After the Target-Attack Action is completed, a Travel Action begins and this results in an ascending path that brings the UAV back to the Cruise Altitude.

Figure 6.31 shows four plots of relevant state variables for the Target-Attack Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. From these, it is possible to notice that the UAV descends adopting a steep negative pitch angle; this results in a Speed increase, up to 60 m/s (the altitude-hold loop integrates a speed limiter algorithm, see Section 3.5.2). Due to the high speed, the UAV cannot perform tight manoeuvres while descending. When the UAV has reached the Attack Altitude (175 m = 150 m above + 25 m ground level), speed lowers and manoeuvrability is increased. The *Waypoint* is correctly reached and then the actual pass over the target can be performed. The UAV then climbs back to the Cruise Altitude while performing the following Travel Action; the climb is performed at a steep pitch angle which results in the UAV being unable to maintain its desired speed (which is the Cruise Speed) even though Maximum Thrust is used.

6.4.7 Circle

The Circle Action type is used to have the UAV loiter above a desired position waiting for a specified time; certain task types might be accomplished while loitering (for example, the UAV might act as a communication relay). The Exag implementation

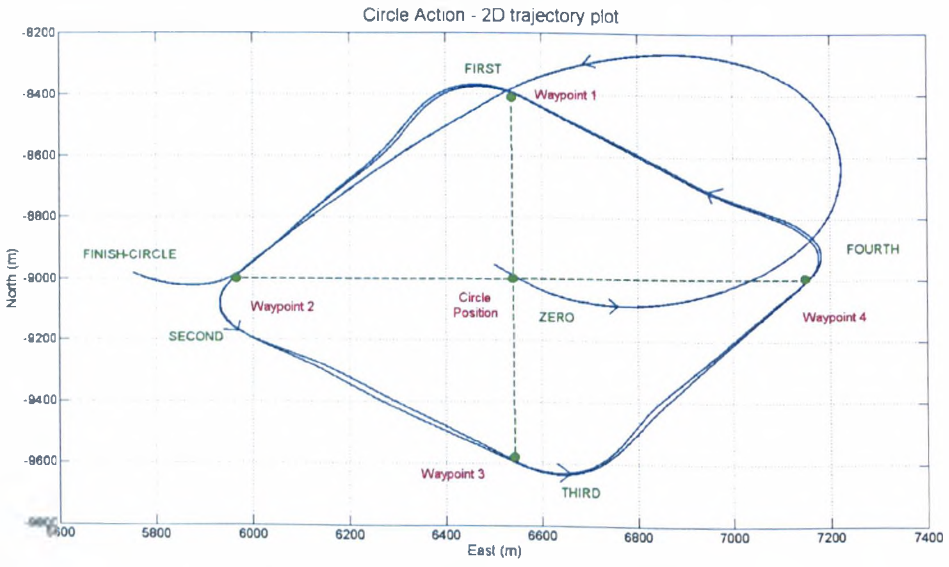


Figure 6.32. 2D trajectory for the Circle Action.

of the Circle Action is described in Section 6.2.8. For rotary-wing aircraft, the equivalent to the Circle Action would be a Hover Action. The Circle Action involves flying iteratively through a series of four waypoints at an altitude which is normally specified by the UAV operator. The manoeuvre is usually accomplished at a flight speed which will minimize fuel consumption. The Circle Action makes use of the Auto mode of the autopilot.

The visualized data is taken from a Circle Action performed during the same simulated mission from which the data for the Taxi, Take-off, Climb and Travel Actions was taken. The UAV starts from the position reached at the conclusion of a Travel Action (in fact, the one shown in Section 6.4.4). The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

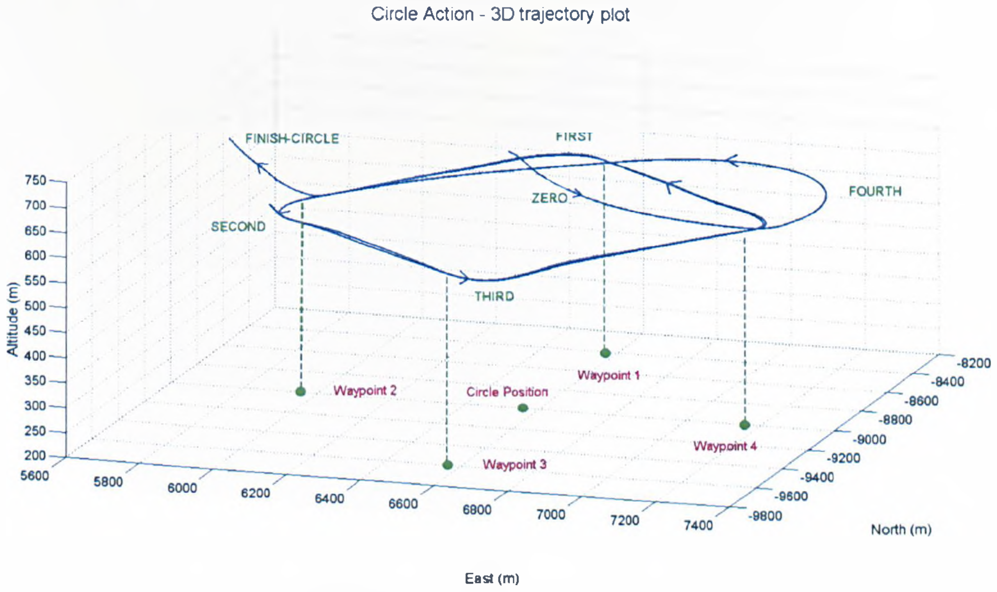


Figure 6.33. 3D trajectory for the Circle Action.

Figure 6.32 shows the 2D trajectory covered by the UAV during the Circle Action; in this case, the desired position for loitering is located at coordinates [53°18'52''N - 1°16'56''W], corresponding to North/East coordinates [-9000, 6525]. The Circle Action starts with the UAV arriving from the previous Travel Action; the first operators to fire are *calculate-waypoints* and *zero*. The *calculate-waypoints* operator calculates the position of the four waypoints, which are set at the Orbit Distance from the desired Circle Position in the four cardinal directions. Hence, *Waypoint 1* has coordinates [-8400, 6525], *Waypoint 2* has coordinates [-9000, 5925], *Waypoint 3* has coordinates [-9600, 6525] and *Waypoint 4* has coordinates [-9000, 7125]. The *zero* operator commands the UAV to reach Waypoint 1 and the desired Circle Altitude, which is set to the value required by the UAV operator (which is 500 m above the ground). Since the UAV has to descend from the Cruise Altitude, it gathers speed and cannot turn tightly towards Waypoint 1. When Waypoint 1 is reached, the *first* operator fires and directs the UAV towards Waypoint 2; the *first*, *second*, *third* and *fourth* operators then cycle iteratively until the specified time limit (1000 sec) is reached. When the time limit is reached, the *finish-circle* operator fires and terminates the Action; the next Action will be a Travel Action towards the next Objective (or towards the landing airport).

Figure 6.33 shows the 3D trajectory plot covered by the UAV during the Circle Action. It is possible to notice how the UAV descends to the desired Circle Altitude from the Cruise Altitude, and how this altitude is kept throughout the Action. When the Circle Action is terminated, it is followed by a Travel Action, and consequently the UAV starts climbing back to the Cruise Altitude.

Figure 6.34 shows four plots of relevant state variables for the Circle Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. From these, it is possible to notice that the UAV first descends to the desired altitude while travelling towards Waypoint 1, then it stabilizes the loitering

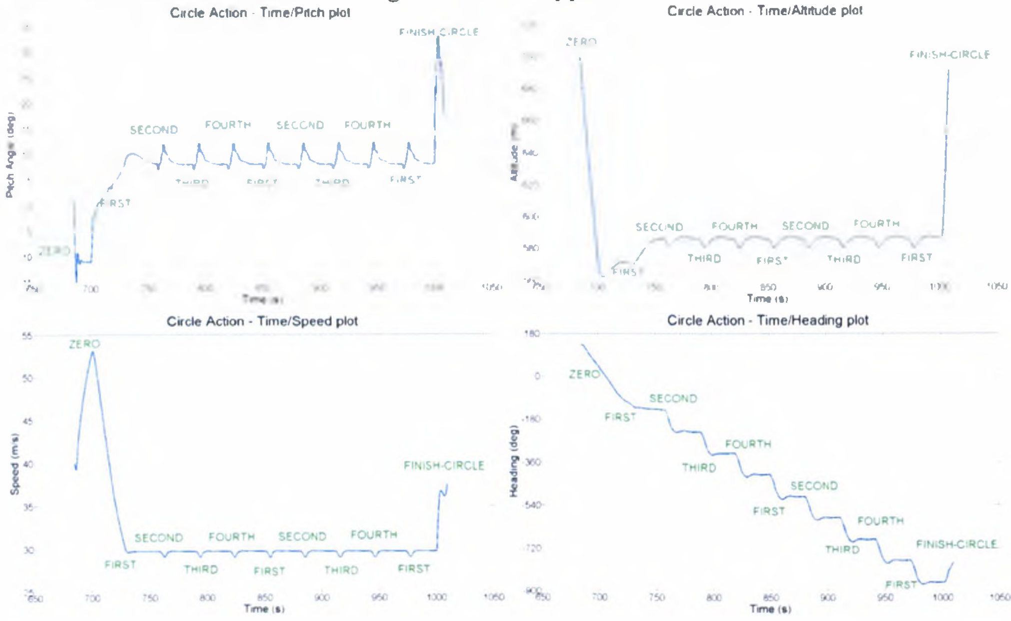


Figure 6.34. State variable plots for the Circle Action.

pattern flying in a regular manner. Each time a waypoint is reached, a 90 deg (more in practice, due to waypoint overshooting) left turn is put in practice. It is possible to notice that the flight of the UAV is affected by the turns; at each turn, the Pitch Angle decreases (as natural for a rolling fixed-wing aircraft), causing a loss of altitude, which is then countered by an increase in the Pitch Angle, causing in turn a decrease in speed. Flight is stabilized again when the turn is completed. The Time/Heading plot shows that two full iterations of the circling pattern are covered before the time limit is reached.

6.4.8 Descent

The Descent Action type is used to prepare the UAV for landing, taking it to a low altitude and aligning it with the runway where the UAV should land; its Exag implementation is described in Section 6.2.9. The Action is accomplished by flying through two separate waypoints, both aligned with the runway and set at different altitudes. Because the UAV is descending, flight speed can be rather high, despite the thrust being set to idle for most of the Action. The Descent Action makes use of the Auto mode of the autopilot.

The visualized data is taken from a Descent Action performed during the same simulated mission from which the data for the Taxi, Take-off, Climb and Travel Actions was taken. The UAV is landing at the same airport from where it took off, in the opposite direction of the runway. The Descent Action starts from the position reached at the conclusion of a Travel Action, which is added to the flight plan by the *return* operator (see Section 4.2.6). The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

Figure 6.35 shows the 2D trajectory covered by the UAV during the Descent Action; the Action starts when the desired landing location is reached at the cruise conditions (altitude and speed). The descent path is defined by two waypoints which are calculated by the *calculate-waypoint-1* and *calculate-waypoint-2* operators. These waypoints are placed along the runway axis at preset distances and altitude from the desired landing location. In this scenario, Waypoint 2 is placed at 2000 m distance and 200 m altitude

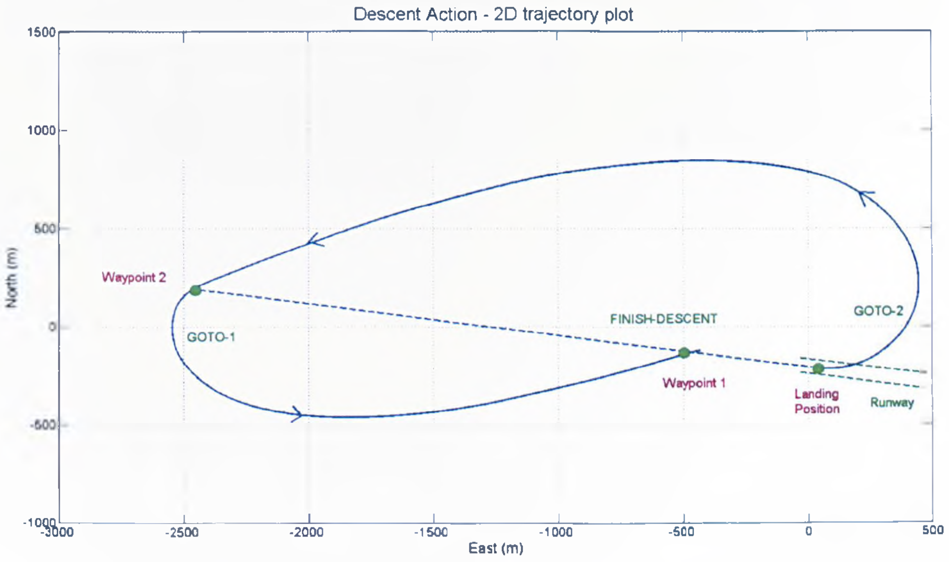


Figure 6.35. 2D trajectory for the Descent Action.

from the Landing Position; Waypoint 1 is placed at 500 m distance and 80 m altitude from the Landing Position. The *goto-2* operator commands the UAV to reach Waypoint 2 (note that Waypoint 2 is reached before Waypoint 1). Speed is set at the Orbit Speed value (item 4.2.4 in Table 6.4). The *goto-1* operator fires when Waypoint 2 is reached

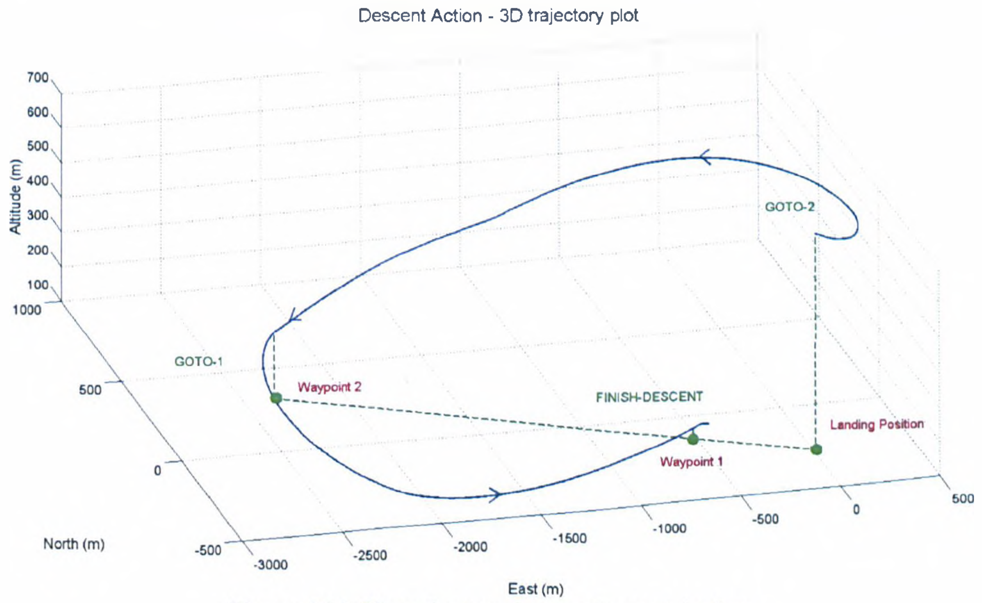


Figure 6.36. 3D trajectory for the Descent Action.

and directs the UAV towards Waypoint 1. During a consistent part of the Action, the UAV is descending rapidly and gathers speed, so it performs high-radius turns. When the desired altitudes have been reached, the UAV can slow down and prepare for landing. Waypoint 1 is reached at the correct altitude, at low speed and with a small intercept angle relative to the runway heading, so it should be possible to smoothly turn the UAV in the runway direction. The *finish-descent* operator fires when Waypoint 1 is reached, terminating the Action and thus causing the ensuing Landing Action to be performed.

Figure 6.36 shows the 3D trajectory plot covered by the UAV during the Descent Action. It is possible to notice the smooth descending path undertaken by the UAV and how Waypoint 1 is reached with an attitude that should allow a correct landing.

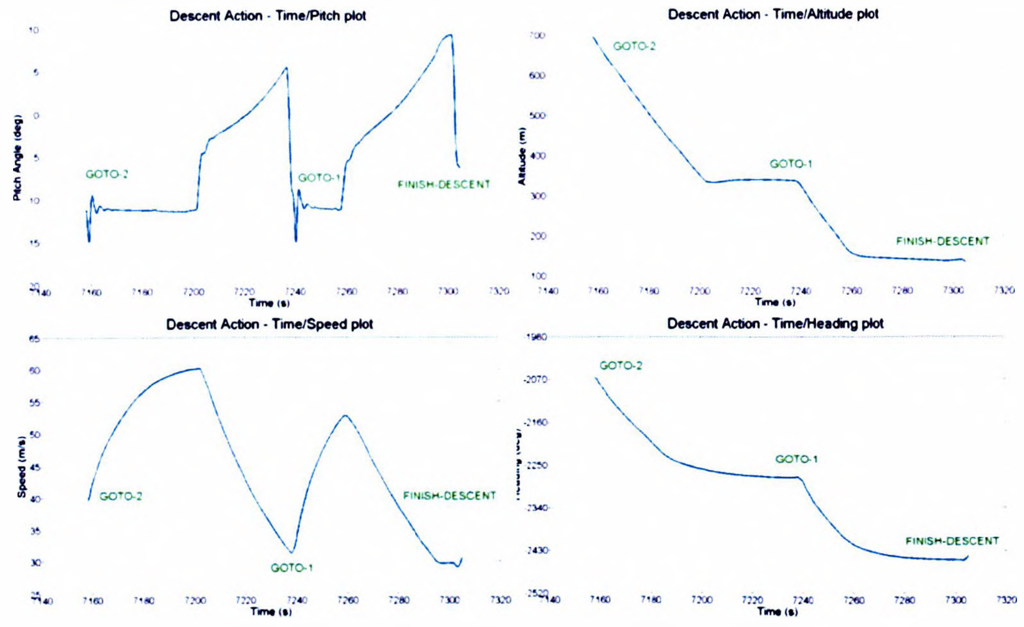


Figure 6.37. State variable plots for the Descent Action.

Figure 6.37 shows four plots of relevant state variables for the Descent Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle plot shows that a steep descent path is used; when desired altitudes are reached, the UAV is levelled and then must gradually increase pitch in order to maintain latitude while speed is decreasing. The Time/Altitude shows that the altitudes of Waypoint 1 and 2 are reached before the waypoints themselves; this could be improved by decreasing the steepness of the descent path. The Time/Speed plot shows that the UAV achieves very high speeds during descent, even though idle thrust is used; it is to be noted that speed stabilizes at the desired Orbit Speed value before reaching Waypoint 1. The Time/Heading plot clearly shows the two parts of the Descent Action, during which the two waypoints are reached.

6.4.9 Landing

The Landing Action type is used to land the UAV; the UAV has been approaching a runway and it must control its descent rate while keeping alignment with the runway. It is to be noted that actual automatic landing systems use ground-based radio transmitters in order to give a reference signal to the landing aircraft; in the SAMMS landing implementation, the presence of a standard Instrument Landing System (ILS) is not assumed, and the algorithm is designed to perform a landing only on the basis of limited information regarding the runway. The Exag implementation of the Landing Action is described in Section 6.2.10. Unlike the Descent Action which precedes it, the Landing Action makes use of the Direct mode of the autopilot.

The visualized data is taken from a Landing Action performed during the same simulated mission from which the data for the Taxi, Take-off, Climb and Travel Actions was taken. The UAV is landing at the same airport from where it took off, in the opposite direction of the runway. The Landing Action starts from the position reached at the conclusion of the Descent Action, e.g. Descent Waypoint 1. To highlight how the Descent and Landing Actions are integrated, the final part of the preceding Descent Action is shown in all the plots. The origin of the North/East reference system is still situated at the *Parking Position* of coordinates [53°23'42''N - 1°22'51''W].

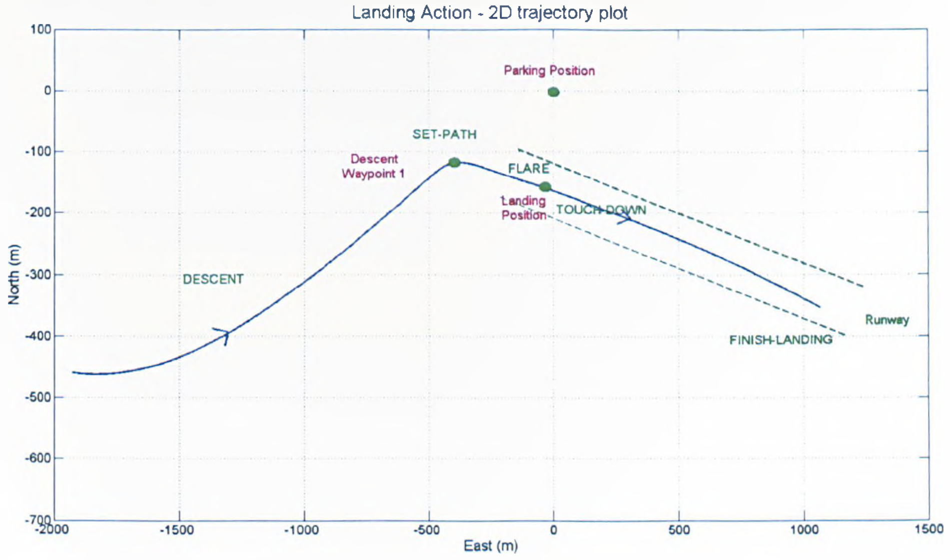


Figure 6.38. 2D trajectory for the Landing Action.

Figure 6.38 shows the 2D trajectory covered by the UAV during the Landing Action; the Action starts when Waypoint 1 is reached during the Descent Action. The first operators to fire are *calculate-angle* and *set-path*; the former calculates the pitch angle that should be maintained to land at the ideal touch-down position (the Landing Position) starting from Waypoint 1, the latter sets the commands so that the UAV will

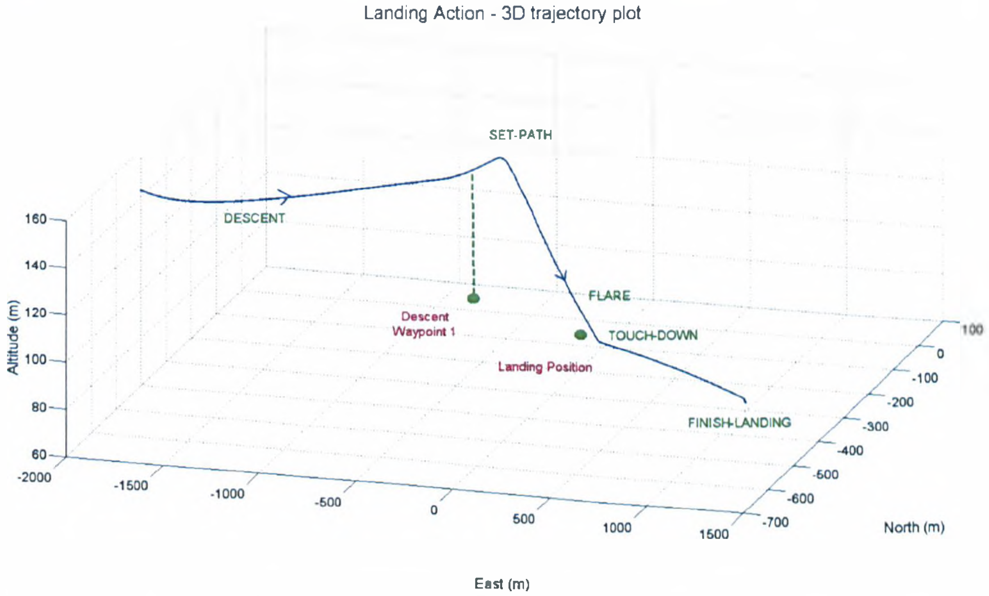


Figure 6.39. 3D trajectory for the Landing Action.

steer directly into the runway direction and assume the required ramp angle. The UAV then assumes a descending attitude which is maintained until very close to the ground, at which point the *flare* and *touch-down* operators fire in sequence. During this phase, lateral-directional control must keep the UAV within the runway axis, but is much easier to accomplish than longitudinal control. When securely on-ground, the UAV will slow down to a halt; in this phase, ground brakes are used, as well as the steering wheel to achieve directional control. The Action terminates when the UAV has fully stopped.

Figure 6.39 shows the 3D trajectory plot covered by the UAV during the Landing Action. It is possible to notice the descending path undertaken by the UAV; please note the scale of the plot, which is used to highlight the various phases of landing but sees the different axes having different scales. Consequently, the actual descending path is not as steep as it would look from the plot.

Figure 6.40 shows four plots of relevant state variables for the Landing Action: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle plot clearly shows the adopted descending path and the flare manoeuvre; however it is possible to notice that the UAV model inaccuracies that were already seen during the Take-off Action are again impacting the simulation. Again, the transition from the airborne to the ground condition is not seamless due to the simplifications in the model, and consequently some state variable calculations are affected. The Time/Speed plot also shows these errors; in this case, the UAV is unable to achieve a full stop, because the calculation of aerodynamic and gravitational forces is affected. The Time/Altitude and Time/Heading plots are not evidently affected by the errors; this is because they are derived from the kinematic equations rather than the equations of motion (see Section 3.4). The plots show that the UAV can successfully maintain the commanded approach profile and the correct runway heading.

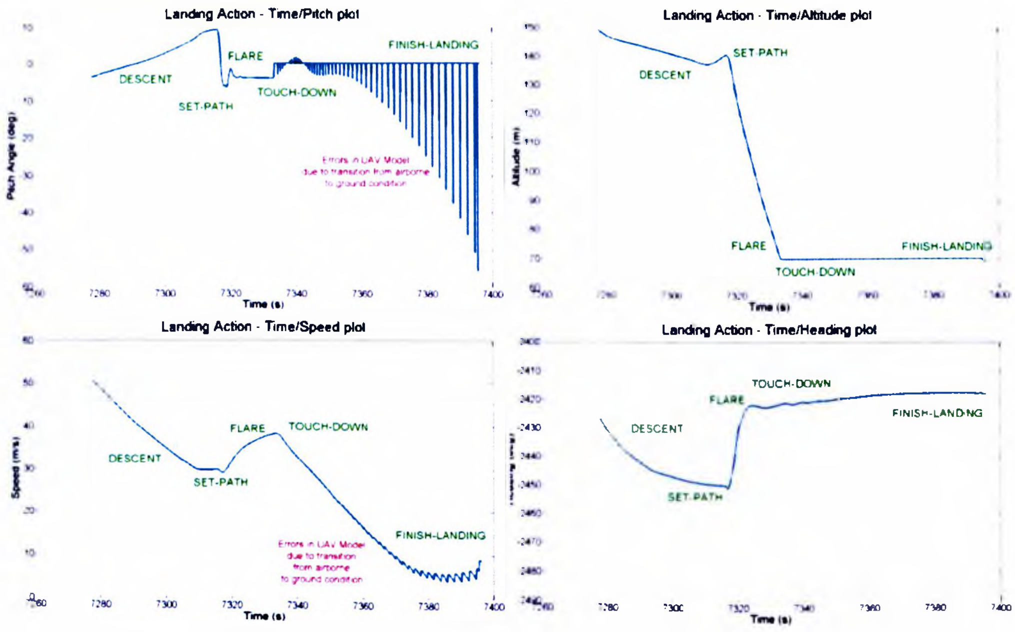


Figure 6.40. State variable plots for the Landing Action.

The Landing Action is followed by a Taxi Action which will bring the UAV to its Parking Position. This is entirely similar to the Taxi Action shown in Section 6.4.1, although naturally with different starting and ending points inverted (the starting point will be the actual position where the UAV achieves a full stop, while the ending position will be the Parking Position). When the Parking Position is reached, the final Park Action begins and the mission is concluded.

6.5 Concluding remarks

In this chapter, the final component of the SAMMS architecture (the Execution Agent) was described and tested. Testing required the use of the entire architecture; the simulation architecture used for the Exag testing is also used for the testing of SAMMS as a system. However the focus is very different: while testing the Exag, the focus is on single Actions and how they are implemented; when testing the entire architecture, the focus is instead placed on how an entire mission is carried out.

The first part of the chapter was dedicated to the description of the Execution Agent. First, the I/O interface was thoroughly analyzed, detailing the inputs and output needed by the agent to perform its functions. The Execution Agent was then described; the Exag structure focuses on the execution of Actions types, so for each Action type an accurate description of the Soar operators and production rules that implement it was provided.

The second part of the chapter focuses on the testing campaign for the Execution Agent. Two types of tests are introduced. The first type of tests is focused at testing how the Execution Agent converts each Action type into a series of commands that can be actuated by the UAV; by focusing on separate Actions, it is possible to verify that the flight trajectories commanded for each Action type are realistic and feasible. The second type of tests, focused on verifying the behaviour of the Execution Agent (and of the SAMMS architecture in general) during entire missions, is only presented.

The results from the second type of tests will be presented in the next chapter. Scenarios from the Planner Agent and Mission Manager Agent testing campaigns will be chosen and their execution verified. While all scenarios presented in Chapters 4 and 5 have been in fact tested, only six of these will be presented in the thesis.

Since the entire SAMMS architecture will be operative without restrictions, the next chapter also represents the apex of this work, summarizing everything that was discussed up to this point.

7. Final Results

In Chapters 4, 5 and 6 the Soar-based Autonomous Mission Management System was thoroughly described and extensively tested by focusing on the main components: the Planner Agent, the Mission Manager Agent and the Execution Agent. In this chapter, the focus will be instead placed on the entire architecture: coherent operation of the three Soar agents together with the low-level control functions will be demonstrated.

A similar test configuration was used for the tests presented in Section 6.4; in these tests, the focus was placed on a thorough analysis of the execution of each Action type. The test presented in this chapter will instead focus on demonstrating the execution of entire missions, allowing to see how Actions that form a flight plan are executed in a sequence.

These tests represent the sum of the entire project: in executing them, the goal is not to demonstrate specific capabilities, but instead to prove the feasibility of SAMMS as a system, and thus its ability to manage and execute a UAV mission at the same time.

As stated previously, all of the scenario variations introduced during the Planner Agent and Mission Manager Agent testing campaign have been simulated with the full SAMMS architecture. However, in this thesis only a selection of these will be presented. The selected scenarios are chosen to demonstrate the widest possible array of functionality and include scenario variations from both the Planner and the MMA testing campaigns.

The plots used to visualize results obtained from the simulations are similar to those used in Section 6.4. In particular, the following plot types will be used:

- Two-dimensional trajectory plot (North/East coordinates)
- Three-dimensional trajectory plot (North/East/Altitude coordinates)
- Time/Altitude plot
- Time/Speed plot
- Time/Heading plot
- Time/Pitch Angle plot
- Time/Roll Angle plot

The flight plans for each scenario are available in Sections 4.4 and 5.4, however these flight plans will be reproduced in this section to allow the immediate comparison between the planned and the actual flight trajectory.

The results for six scenarios will be shown; separate sections are dedicated to each scenario, so the chapter is divided into six sections. Section 7.1 shows the results for scenario variation 1c; in Section 7.2, the results for scenario variation 2e are shown. Section 7.3 shows the results for scenario variation 3c; in Section 7.4, the results for scenario variation 4c are shown. Section 7.5 shows the results for scenario variation 5e; in Section 7.6, the results for scenario 7 are shown.

7.1 Scenario variation 1c

Scenario 1 is the simplest scenario and was used to demonstrate the basic functionality of SAMMS. Four variations were defined, with two variations being presented in this thesis. Scenario variation *1c* is used during the Planner testing campaign (see Section 4.4.1); scenario variation *1d* is instead used during the MMA testing campaign (see Section 5.4.1). For the Execution Agent testing campaign, all four variations were in fact tested, but only the results for scenario variation *1c* will be shown in the thesis.

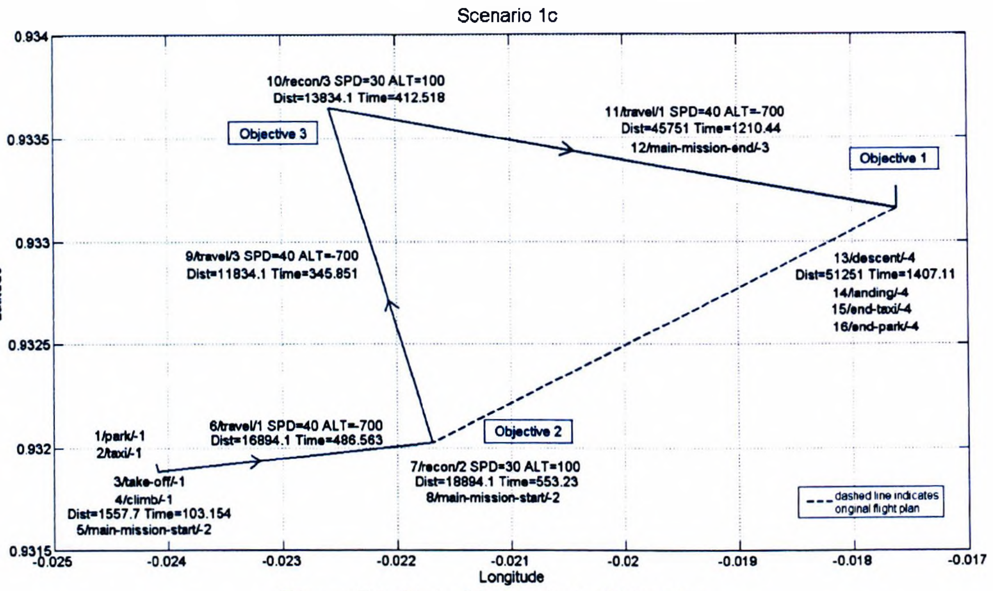


Figure 7.1. Plot of Scenario 1c flight plan.

Scenario variation 1c involves three Objectives: Objective 1 is a Transit Objective, while Objectives 2 and 3 are Target-Analyze Objectives. The mission starts with the UAV located at Sheffield Airport, with only Objectives 1 and 2 assigned. The first generated flight plan expects the UAV to accomplish Objective 2 then fly towards Objective 1 and land there. While Objective 2 is being carried out, Objective 3 is added to the Objective list; a new flight plan is generated, expecting the UAV to complete

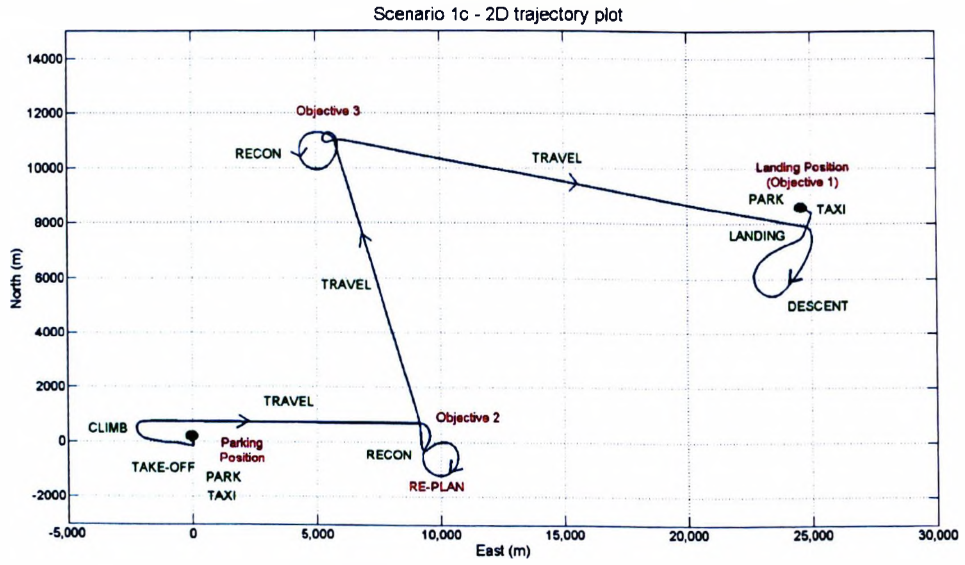


Figure 7.2. 2D flight trajectory for Scenario variation 1c.

Objective, perform Objective 3 and then finally fly towards Objective 1 to land there. Figure 7.1 shows the updated flight plan for the mission. It is important to note that the re-planning event occurs while performing Action 7 (the Target-Recon Action part of Objective 2), which corresponds to time 700 sec.

Figure 7.2 shows the 2D trajectory plot obtained from the simulation of scenario variation 1c. On the plot, the take-off, landing and Objective positions are highlighted. As can be seen in Figure 6.41, the flight plan consists of 16 Actions; the corresponding part of the flight trajectory is indicated in the plot of Figure 6.42 (apart from Main-

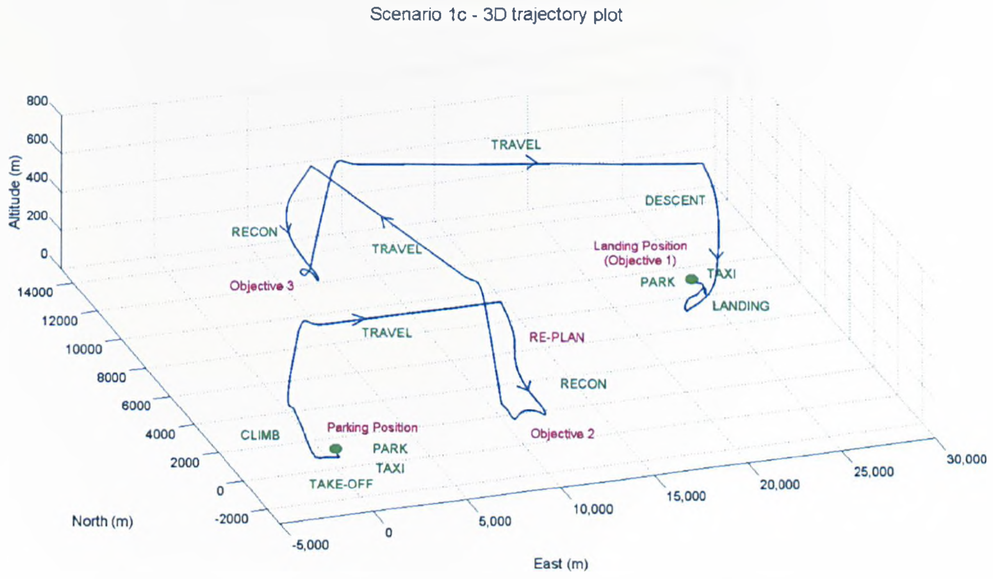


Figure 7.3. 3D flight trajectory for Scenario variation 1c.

Mission-Start and Main-Mission-End Actions, which are instantaneous. It is also possible to notice the particular trajectory adopted to accomplish Objective 2; this is due to the re-planning event which occurs while performing Action 7. Action 7 is a Target-Recon Action, and thus a waypoint is selected to prepare the UAV for the pass over the target (see Section 6.2.7); the re-planning event occurs before the waypoint is reached,

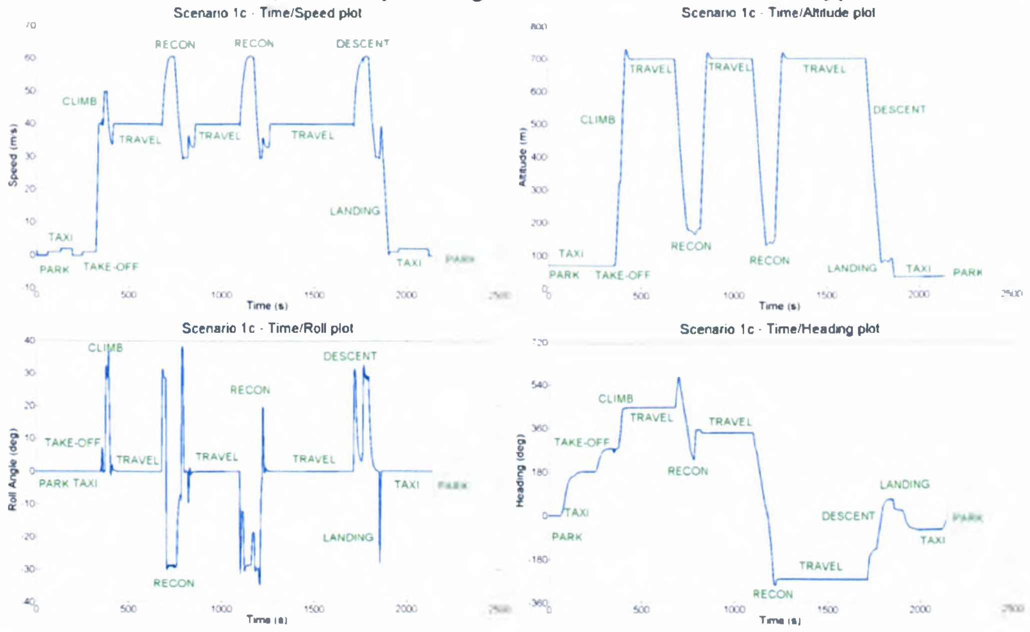


Figure 7.4. State variable plots for Scenario variation 1c.

and since the UAV has a new destination, a different waypoint is chosen. Hence, the UAV adjust the manoeuvre turning into a different direction. From the plot, it is also possible to see that the Target-Recon manoeuvre for Objective 3 requires two turns around the target to be accomplished, and that the Descent manoeuvre is accomplished smoothly.

Figure 7.3 shows the corresponding 3D trajectory plot, from which it is possible to see that the take-off, target-recon and descent manoeuvres all require consistent altitude changes.

Figure 7.4 shows four plots of relevant state variables for the scenario: the Time/Speed plot, the Time/Altitude plot, the Time/Roll Angle plot and the Time/Heading plot. From these plots, it is possible to see how the Actions that compose a flight plan are fused together while being executed by the Exag. Comparing the plots in Section 6.4 with these, each Action type can be identified at the time when it is executed. Due to the time scale of these plots (as opposed to the time scale of those in Section 6.4), the simulation errors during take-off can hardly be noticed; in this particular case, simulation errors during landing are not present and the mission is fully completed, with the execution of the final Taxi and Park Actions.

7.2 Scenario variation 2e

Scenario 2 is a scenario that is mainly aimed at demonstrating the search patterns implemented within SAMMS. Five variations were defined, with four variations being presented in this thesis. Scenario variations *2a* and *2c* are used during the Planner testing campaign (see Section 4.4.2); scenario variations *2d* and *2e* are instead used during the MMA testing campaign (see Sections 5.4.2 and 5.4.3). For the Execution Agent testing campaign, all five variations were in fact tested, but only the results for scenario variation *2e* will be shown in the thesis.

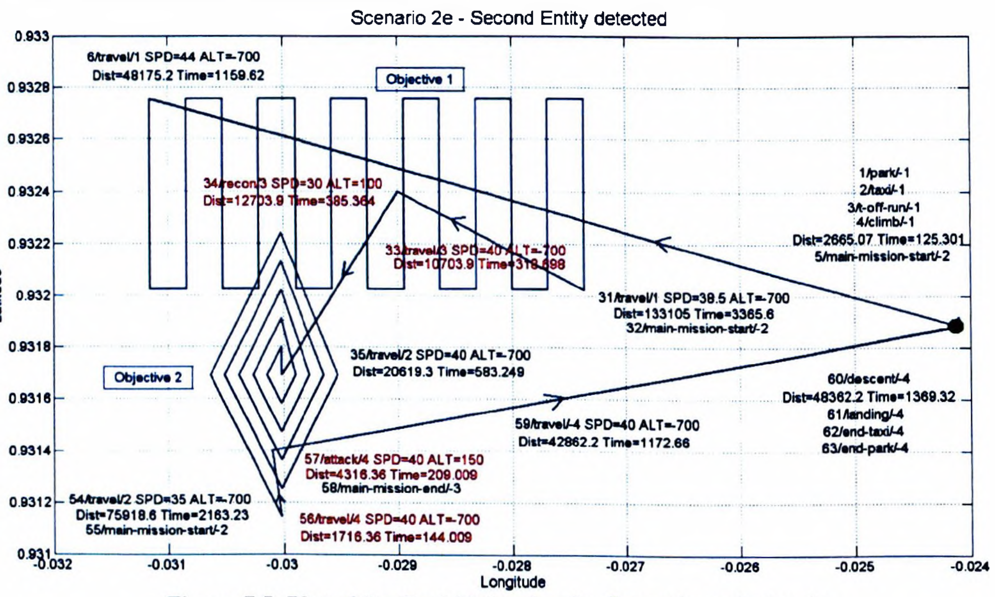


Figure 7.5. Plot of the final flight plan for Scenario variation 2e

Scenario variation 2e involves two Search Objectives: however, Objective 1 is specified as a Search-and-Analyze Objective, while Objective 2 is specified as a Search-and-Attack Objective. This means that while these Searches are in progress, if a new Entity of the desired type is detected within the search area the MMA will automatically add a new Target-Analyze (or Target-Attack) Objective to the Objective list. The mission starts with the UAV located at Sheffield Airport, which is also the designed landing airport. The first generated flight plan expects the UAV to accomplish

Objective 1 (a Search of type *box*) then Objective 2 (a Search of type *circle*). At time 2100 sec, while Action 21 is being performed, the first new Entity is injected into the Entity List; consequently, a Target-Analyze Objective (Objective 3) is added by the MMA, and the Planner generates a new flight plan which includes this Objective. Then at time 3950 sec, while Action 46 is being executed, the second new Entity is injected into the Entity list; consequently, a Target-Attack Objective (Objective 4) is added by the MMA, and the Planner generates a new flight plan which includes this Objective. Figure 7.5 shows the final flight plan for the mission; Actions added because of the Objectives added by the MMA are highlighted in red.

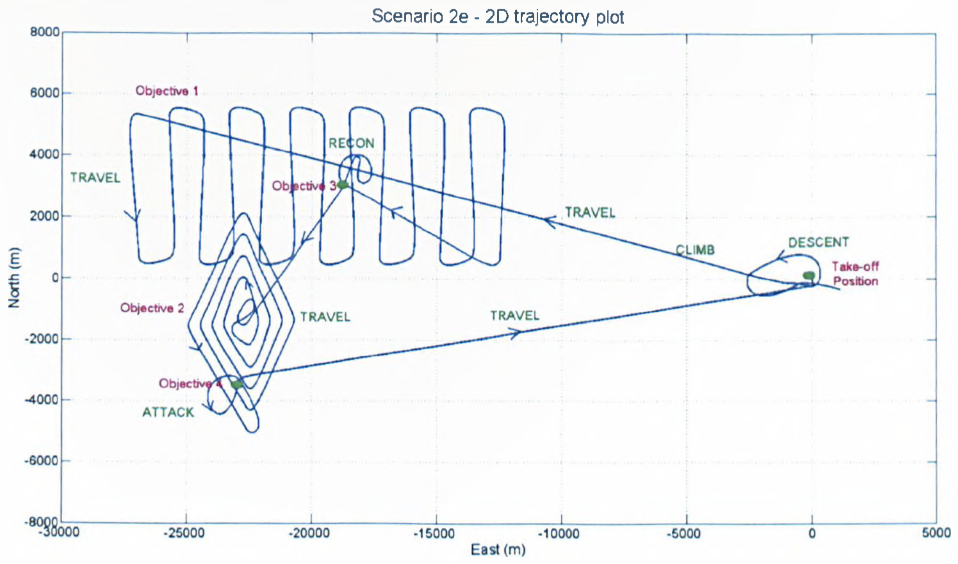


Figure 7.6. 2D flight trajectory for Scenario variation 2e.

Figure 7.6 shows the 2D trajectory plot obtained from the simulation of scenario variation 2e. Highlighted features include the take-off (and landing) position, the Search Objectives and the position of new Objectives added by the MMA. The final flight plan

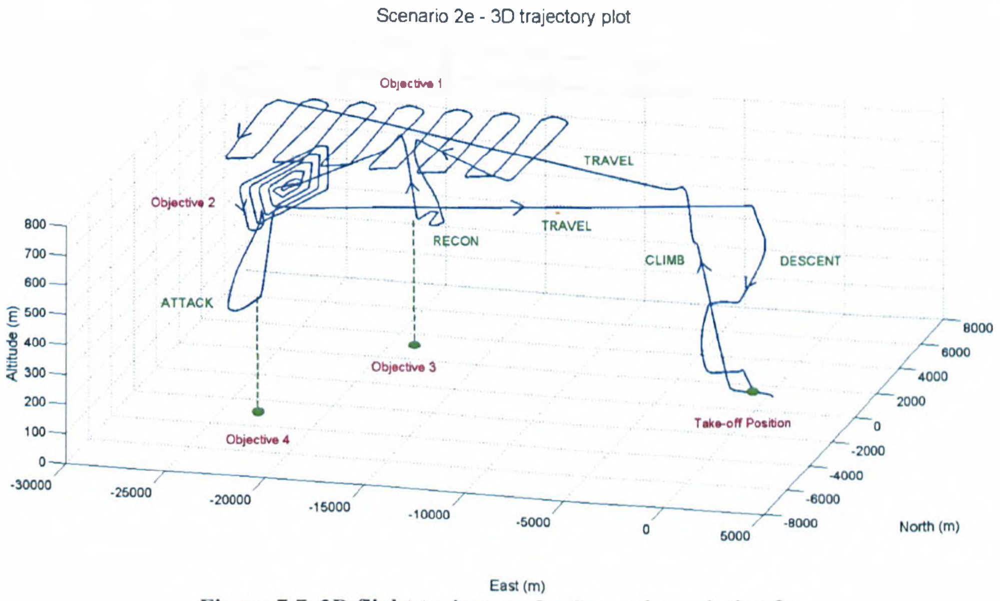


Figure 7.7. 3D flight trajectory for Scenario variation 2e.

consists of as many as 63 Actions; not all of these are indicated on the plot. Action labels are in fact limited to the Climb and Descent Actions, the Recon and Attack Actions corresponding to Objectives 3 and 4, and the Travel Actions which bring the UAV from the starting airport to the commence start point and from Objective to the landing airport. The altitude of the ground is higher than in other scenarios, thus the Recon and Attack Actions do not bring the UAV lower than 400 m above sea level. Some particular aspects of the manoeuvres can be noted: first, the Recon Action for Objective 3 is executed with a 360 degree spiralling descent phase followed by a low-altitude/low-speed phase with tight turns; secondly, Actions 35 (which brings the UAV from the Objective 3 position to the centre of Objective 2 search area) and 36 (which is the first leg of Search Objective 2) intersect with a large relative angle and consequently the manoeuvre is not performed smoothly (notice however that the search area is still covered); finally, the final Attack Action is performed seamlessly.

Figure 7.7 shows the corresponding 3D trajectory plot, from which it is possible to notice how the flight trajectory evolves along the altitude axis. The spiralling trajectories adopted during the Recon, Attack and Descent Actions can be noted.

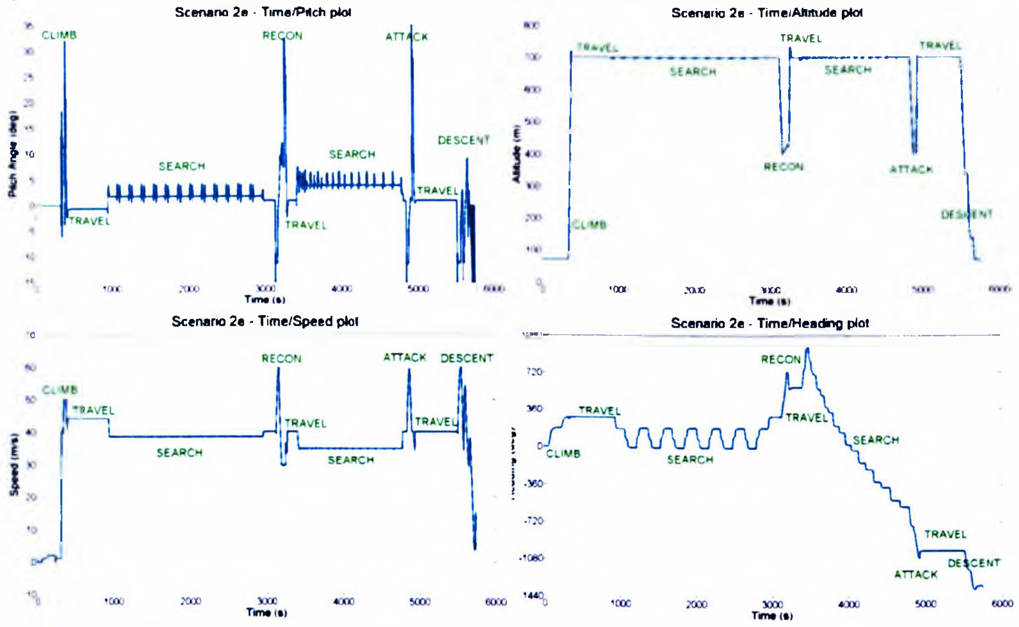


Figure 7.8. State variable plots for Scenario variation 2e.

Figure 7.8 shows four plots of relevant state variables for the scenario: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle plot shows how Pitch must be constantly adjusted in order to maintain altitude, in particular during turns (there are many tight turns performed during a Search) and when the speed is changed (Pitch must be higher if Speed is lower). The Time/Altitude plot shows that Cruise Altitude is maintained for most of the time (Search Objectives might be set with a different altitude, but in the scenario the Cruise Altitude was used). The Time/Speed plot shows that Objective 1 is executed at a higher speed as planned; due to the time priority of 3500 sec assigned to it, speed is increased by 10% from nominal values until the Objective is completed. It can be noted that the time estimates which can be seen in the flight plan are in fact conservative; the plan expects Travel Action 6 to be completed at time 1159.62 sec, while it is in fact completed before time 1000 sec; while the Search is expected to be completed at time 3365.6 sec, while it is in fact completed at about time 3000 sec. The need for conservative estimates is explained by the fact that it is not possible to take acceleration/deceleration times and manoeuvring space into account while estimating the distances and times needed to cover a flight plan; it is then preferable to overestimate them. Finally, on the Time/Heading plot, it is possible to notice the different nature of the search patterns. Due to the time scale of these plots (as opposed to the time scale of those in Section 6.4), the simulation errors during take-off can hardly be noticed; the simulation errors during landing are more visible, especially in the Time/Pitch Angle and Time/Speed plots.

7.3 Scenario variation 3c

Scenario 3 is a scenario that is mainly aimed at demonstrating the effect that time priorities have on the flight plan generation within SAMMS. Several Objectives are present and many are assigned a time priority value; the resulting flight plan is impacted by these time priorities, and a re-planning event can be triggered by changing the time priorities. Eight variations were defined, with all variations being presented in this thesis. Scenario variations *3a* through *3e* are used during the Planner testing campaign (see Section 4.4.3); scenario variations *3f* through *3h* are instead used during the MMA testing campaign (see Section 5.4.4). For the Execution Agent testing campaign, all eight variations were in fact tested, but only the results for scenario variation *3c* will be shown in the thesis.

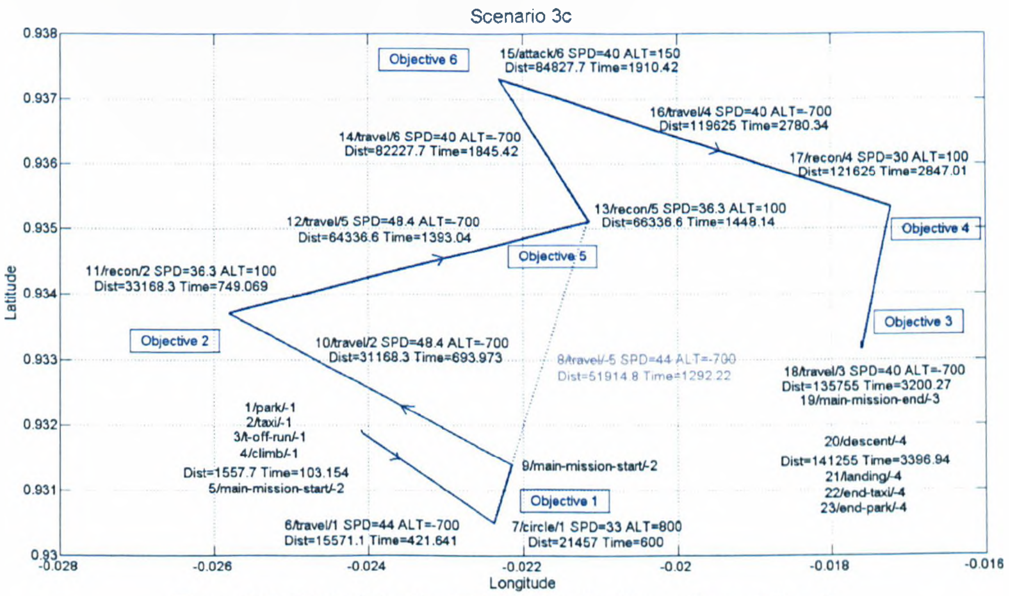


Figure 7.9. Plot of the final flight plan for Scenario variation 3c

Scenario variation 3c involves six Objectives: an Orbit Objective (Objective 1), three Target-Analyze Objectives (Objectives 2, 4 and 5), a Target-Attack Objective (Objective 6) and a Transit Objective (Objective 3); Objective 1 has an immediate time priority, Objective 4 has a time priority of 2900 sec, Objective 5 has a time priority of 1900 sec. Please note that the time priorities and the time limit for Objective 1 have been increased by 400 sec compared to the values shown during the Planner testing campaign; this is because the take-off sequence is longer to execute than originally expected (particularly true if the UAV has to manoeuvre a lot while on the ground). The initial flight plan expects the Objectives to be performed in the following order: 1-5-4-6-2-3. However, the priorities are changed while performing Action 8 (time 1070 sec, Objective 1 completed); Objective is assigned an immediate time priority, the priority for Objective 4 is revoked and a time priority of 2900 sec is assigned to Objective 6.

The new resulting flight plan expects the order of execution 1-2-5-6-4-3, as can be seen in Figure 7.9. The re-planning event causes a diversion from the original course; Objective 5 was being pursued and is momentarily abandoned (Travel Action 8 is left in the flight plan and indicated as an unsuccessful Action), since the Execution Agent is not committed to the Objective when the re-planning event occurs.

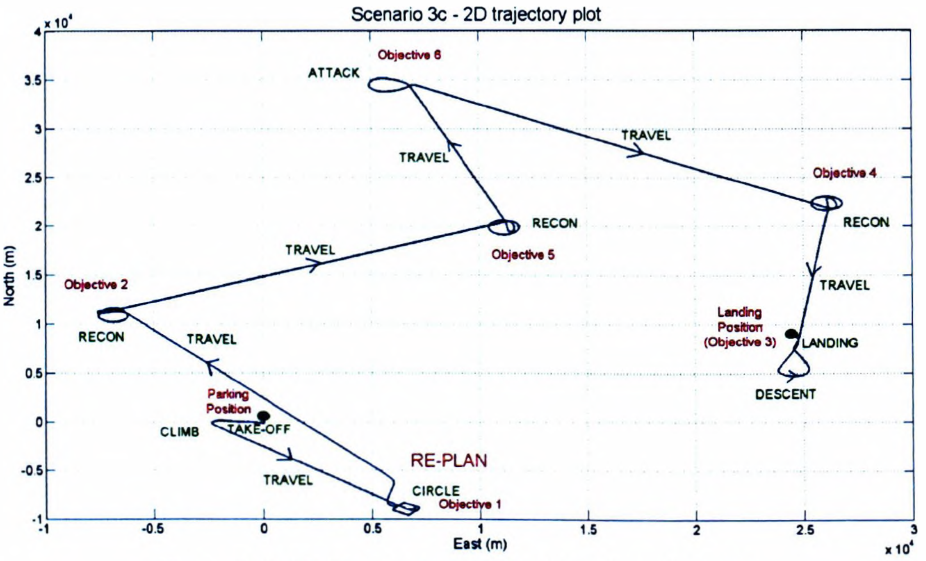


Figure 7.10. 2D flight trajectory for Scenario variation 3c.

Figure 7.10 shows the 2D trajectory plot obtained from the simulation of scenario variation 3c. Highlighted features include the take-off and landing positions, the position of Objectives and the UAV position when the re-planning event occurs. The final flight plan consists of 23 Actions, almost all of which are labelled on the plot. It is

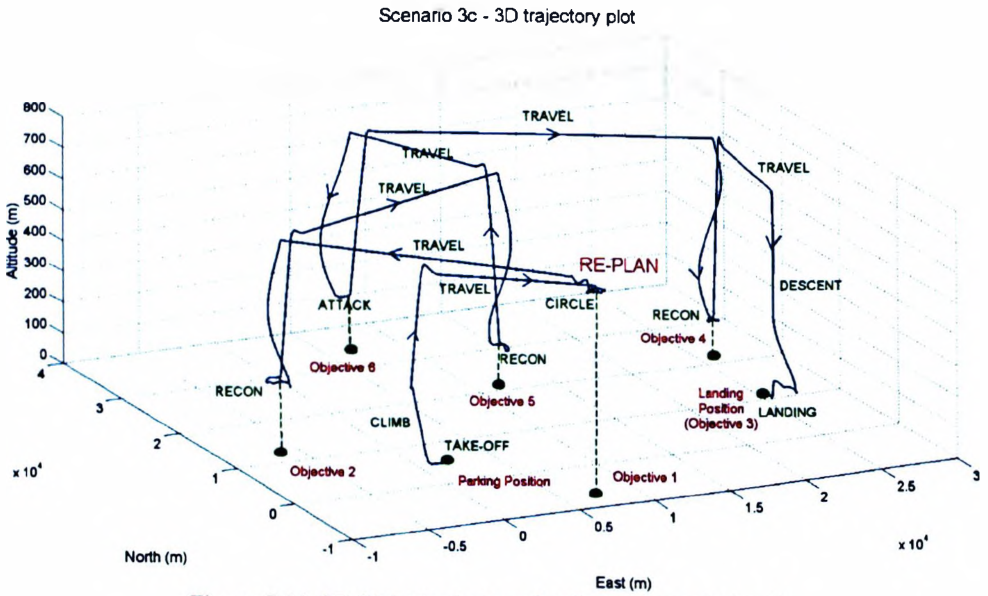


Figure 7.11. 3D flight trajectory for Scenario variation 3c.

possible to notice the flight trajectories that the UAV undertakes when performing different Actions; the Circle Action corresponds to slow tight turns, Recon and Attack Actions are executed with a spiralling trajectory (which is different for each Action, since it depends on several contextual factors) and the Descent manoeuvre is affected by the speed increase achieved during descent.

Figure 7.11 shows the corresponding 3D trajectory plot, from which it is possible to notice how the flight trajectory evolves along the altitude axis. The spiralling trajectories adopted during the Recon, Attack and Descent Actions can be noted, as well as the loitering pattern adopted during the Circle Action.

Figure 7.12 shows four plots of relevant state variables for the scenario: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. From these plots, it is possible to highlight some generic aspects of the Execution Agent and how it executes Actions:

- Circle Actions are performed at low speed (30 m/s), consequently the manoeuvrability of the UAV is high and tight turns can be accomplished
- the first part of Recon and Attack Actions usually involves descending to the desired altitude, which causes high speed and low manoeuvrability; when the altitude is reached, speed is stabilized and the trajectory can be adjusted
- the spiralling pattern assumed during Recon and Attack Actions is different for each Action from a trajectory point of view, but the Pitch Angle, Speed and Altitude profiles are in fact very similar
- since for Recon and Attack Actions an altitude above the ground is specified, the actual altitude (above sea level) reached during these Actions varies (100 m above ground for Recon Actions and 150 m above ground for Attack Actions)

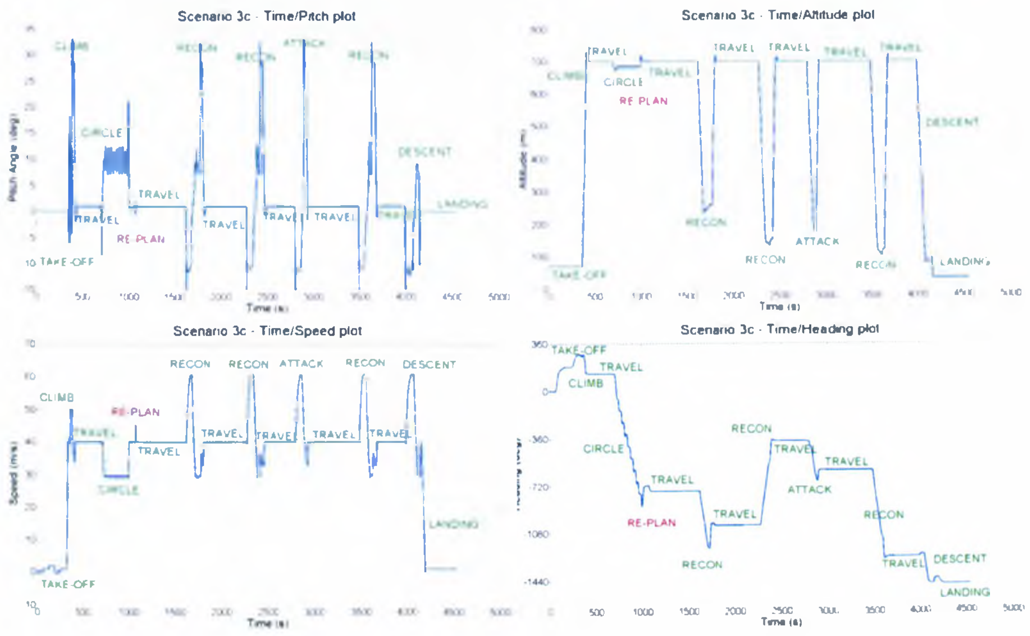


Figure 7.12. State variable plots for Scenario variation 3c.

The re-planning event is also noticeable. The change in time priorities is scheduled to happen at time 1070 sec; at this time, an updated flight plan is generated and, since the Exag is not committed to the current Objective (Objective 5), it has to turn towards another Objective (Objective 2). The plots show that the re-planning event occurs without significant issues. Pitch and speed fluctuations can be observed; these are caused by the rapid turn that the UAV performs once the plan is changed (from Figure

6.50, it is possible to see that this is a roughly 90 deg left turn). Comparing the flight plan in Figure 6.49 with the simulation results, it is again possible to notice that the flight plan time estimates are conservative (please note that in the flight plan plot the re-planning event causes a reset of the total time, so time estimates for Action 10 and following are to be intended after the re-planning occurred).

7.4 Scenario variation 4c

Scenario 4 is a scenario that is mainly aimed at demonstrating SAMMS' ability to deal with a large number of Objectives; furthermore, the presence of dangerous Entities triggers the mission-path-adjust algorithm (see Section 4.2.8). Four scenario variations were defined, with all variations being presented in this thesis. Scenario variations *4a*, 4b and *4c* are used during the Planner testing campaign (see Section 4.4.4); scenario variations *4d* is instead used during the MMA testing campaign (see Section 5.4.5). During the Execution Agent testing campaign, all four variations were in fact tested, but only the results for scenario variation *4c* will be shown in the thesis.

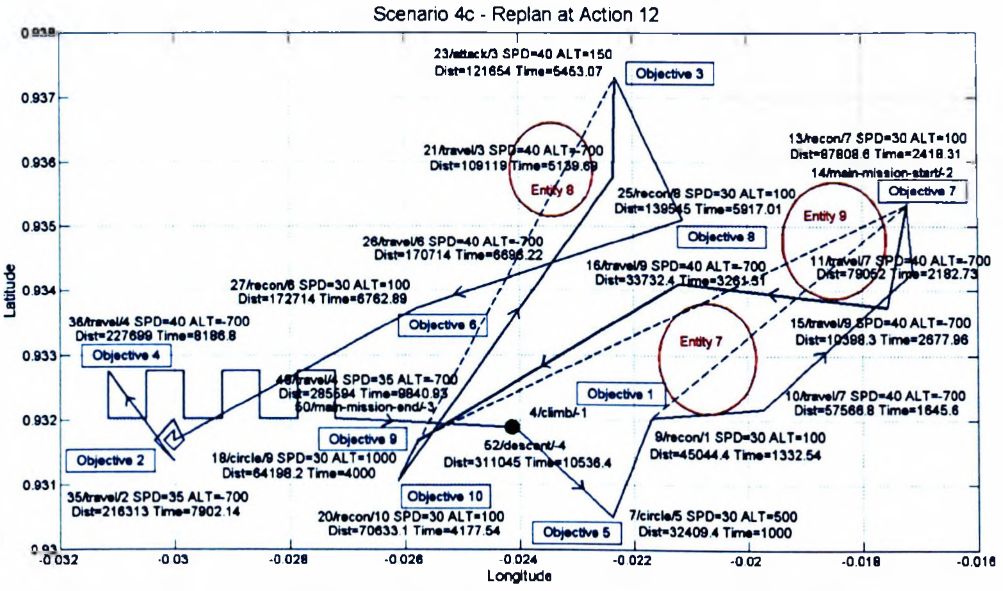


Figure 7.13. Plot of scenario 4c final flight plan, re-planning event at time 1700 sec (Action 12).

Scenario variation 4c involves ten Objectives: two Orbit Objectives (Objectives 5 and 9), five Target-Analyze Objectives (Objectives 1, 6, 7, 8 and 10), a Target-Attack Objective (Objective 3) and two Search Objectives (Objectives 2 and 4); the scenario starts with Objectives 1 through 8, none of which have a time priority. At a specified time, a re-planning event is triggered by the addition of two new Objectives and of a time priority for Objective 3. During the Planner testing campaign, the effects of varying the re-planning time (and consequently the Action being performed) were studied, demonstrating three separate cases (see Section 4.4.4); while all of these cases have been simulated, only one of the cases will be treated in this thesis. The re-planning event is scheduled to happen at time 1700 sec, while Action 12 is being performed (and with the Exag committed to complete its parent Objective, Objective 7); consequently, the time limit for Objective 9 is set to 4000 sec and the time priorities for Objectives 3 and 10 are set at 6000 sec and 4500 sec respectively (Objective 9 has an immediate time priority). The initial flight plan expects the Objectives to be performed in the order 5-1-7-8-3-6-2-4; the re-planning event occurs while performing Objective 7 (with commitment to complete it), and the updated flight plan follows the order 5-1-7-9-10-3-

8-6-2-4. Figure 7.13 shows the final flight plan; the scenario is meant to test the threat avoidance algorithm and its operation can be noted in the flight plan. The algorithm intervenes on three Travel Actions, splitting them into two or more segments; the Travel Action from Objective 1 to Objective 7 is split into three segments, the Travel Action from Objective 7 to Objective 9 is split into three segments and the Travel Action from Objective 10 to Objective 3 is split into two segments. The threat areas represented by the dangerous Entities are thus successfully avoided.

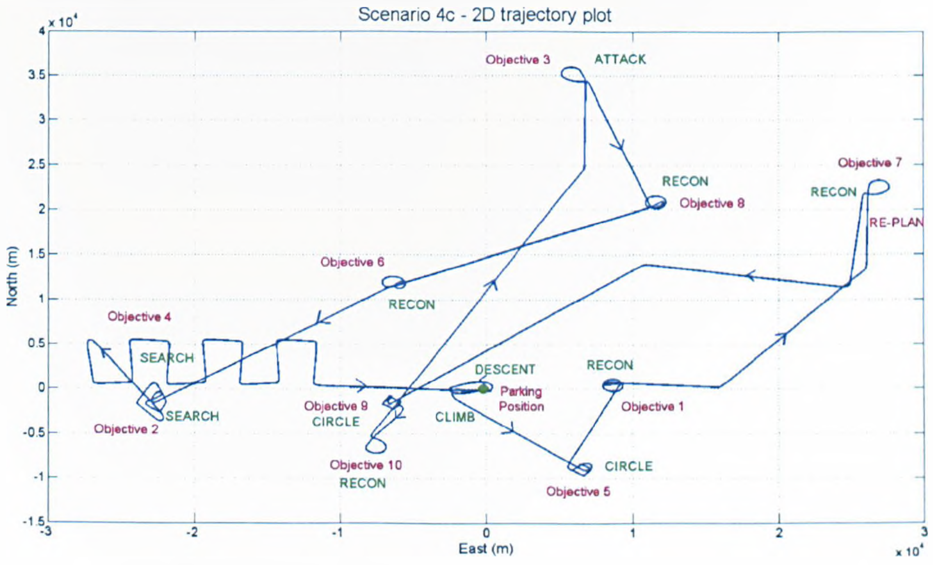


Figure 7.14. 2D flight trajectory for Scenario variation 4c.

Figure 7.14 shows the 2D trajectory plot obtained from the simulation of scenario variation 4c. Highlighted features include the take-off (and landing) position, the position of Objectives and the UAV position when the re-planning event occurs. The final flight plan consists of 55 Actions, so only the most relevant are labelled on the plot. It is possible to notice the flight trajectories that the UAV undertakes when performing different Actions; the Circle Action corresponds to slow tight turns, Recon

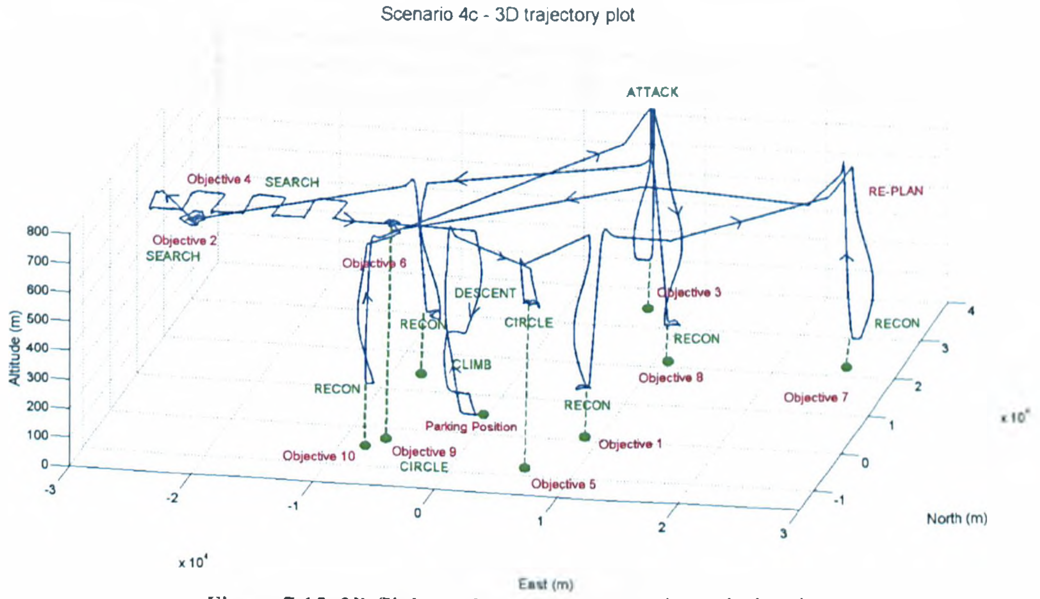


Figure 7.15. 3D flight trajectory for Scenario variation 4c.

and Attack Actions are executed with a spiralling trajectory (which is different for each Action, since it depends on several contextual factors) and the Descent manoeuvre is affected by the speed increase achieved during descent. Furthermore, the resulting flight trajectories due to the operation of the threat avoidance algorithm can be noted: the UAV simply continues flying at Cruise Altitude and Speed while turning towards the new waypoints.

Figure 7.15 shows the corresponding 3D trajectory plot. Due to the complexity of the mission, the plot can be confusing, however when compared with the 2D trajectory plot it allows to better understand the trajectory covered by the UAV during this simulated mission. The ground position of Objectives is noted in the plots for improved clarity.

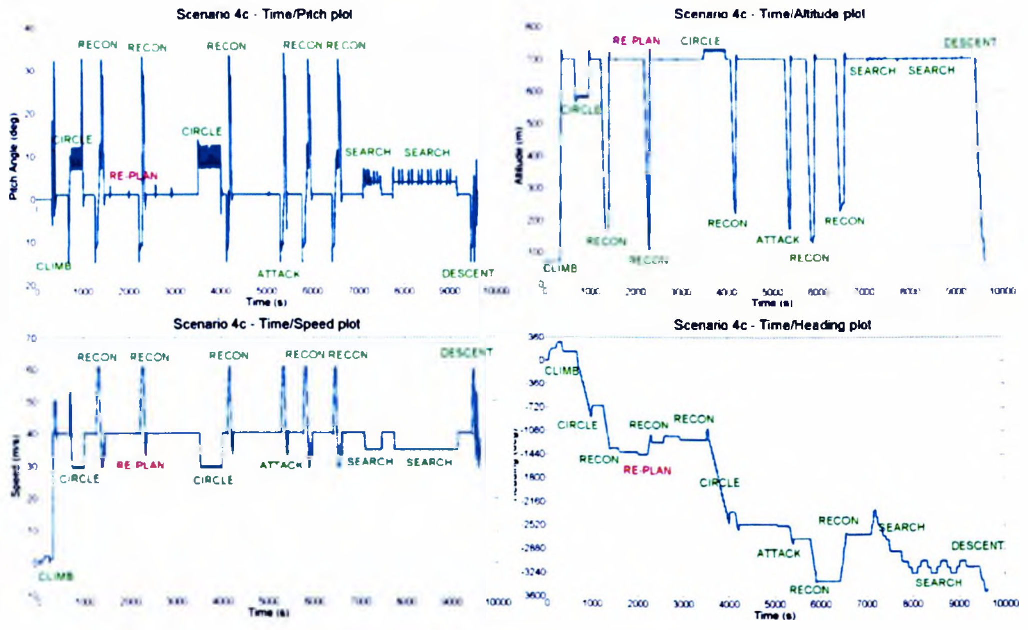


Figure 7.16. State variable plots for Scenario variation 4c.

Figure 7.16 shows four plots of relevant state variables for the scenario: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. Comparing these plots with the same type of plots regarding other scenarios, it is possible to notice that the increased number of Objectives does not impact the functionality of the Execution Agent and of SAMMS as a whole. The system maintains its ability to generate and update appropriate flight plan and then execute them with feasible trajectories. The flight plan time estimates are again revealed to be conservative, with a total time estimate of 10536 sec compared to the actual landing time of roughly 9600 sec. The fuel consumption model is not put to test during the Execution Agent testing campaign; without a more precise "real" fuel consumption model to compare, such a test is not meaningful. In general, the scenarios used during the Exag testing campaign always involve starting with the maximum amount of fuel on-board, and this fuel is deemed sufficient by the Planner to complete all of these scenarios.

7.5 Scenario variation 5c

Scenario 5 is a scenario that is mainly aimed at demonstrating the capability of SAMMS to deal with multiple re-planning events; in most other scenarios, a single re-planning event occurs, while in the most relevant scenario variation for Scenario 5 three separate events occur. Five scenario variations were defined, with the first four (variations *5a* through *5d*) representing the various steps through which the last variation (variation *5e*) passes. During scenario variation 5e, three re-planning events occur, and consequently four flight plan are generated: the initial one plus an updated plan for each re-planning event. For the Execution Agent testing campaign, all five variations were in fact tested, but only the results for scenario variation *5e* will be shown in the thesis.

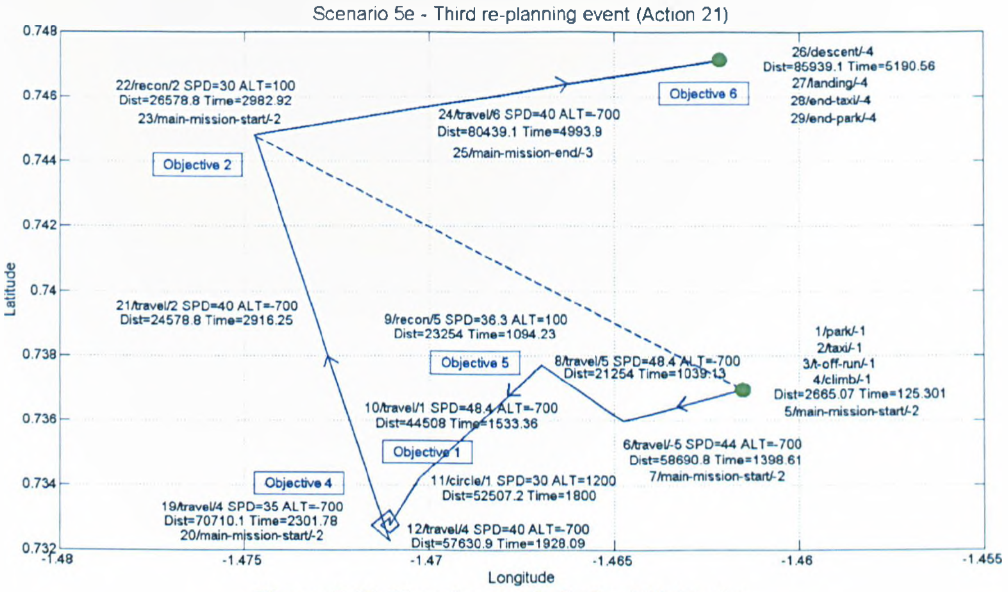


Figure 7.17. Plot of scenario 5e final flight plan.

Scenario variation 5e involves six Objectives, including an Orbit Objective (Objective 1), two Target-Analyze Objectives (Objectives 2 and 5), a Target-Attack Objective (Objective 3), a Search Objective (Objective 4) and a Transit Objective (Objective 6). The scenario starts with Objectives 1 through 4; Objective 1 has a time limit of 1800 sec and a time priority of 1500 sec, while Objective 4 has a time priority of 2500 sec. The initial flight plan involves the following order of execution for Objectives: 1-4-2-3. At time 500 sec, while performing Action 6 (that is, the first Travel Action, directed towards Objective 1), the first re-planning event is triggered by the addition of Objective 5 (with immediate priority); the Exag is not committed to complete Objective 1, so it diverts towards Objective 5, planning to continue flight plan execution with the order 5-1-4-2-3 (it is be noted that a flight speed increase is planned in order to ensure that time priorities are respected). Then at time 2200 sec, corresponding to Action 15 of the updated flight plan (the third leg of the search pattern for Objective 4), a new re-planning event is triggered by the removal of Objective 3 from the Objective list (please note that this removal is caused by the UAV operator and not by the MMA); in this case, the Exag is committed to complete Objective 4, and consequently the updated flight plan simply removes Objective without further consequences. At time 2800 sec, corresponding to Action 21 of the latest flight plan (while the UAV is flying from Objective 4 to Objective 2), the final re-planning event is triggered by the addition of Objective 6 to the Objective list; this in fact instructs the UAV to land at a different airport from the one where it took off. The final flight plan can be seen in Figure 7.17; the actual trajectory covered by the UAV will follow this flight plan. The scenario is set at different locations compared to the other scenarios:

rather than Sheffield airport, the UAV is taking off from Ann Arbor airport (Michigan, USA) and landing at the near Linden Price airport.

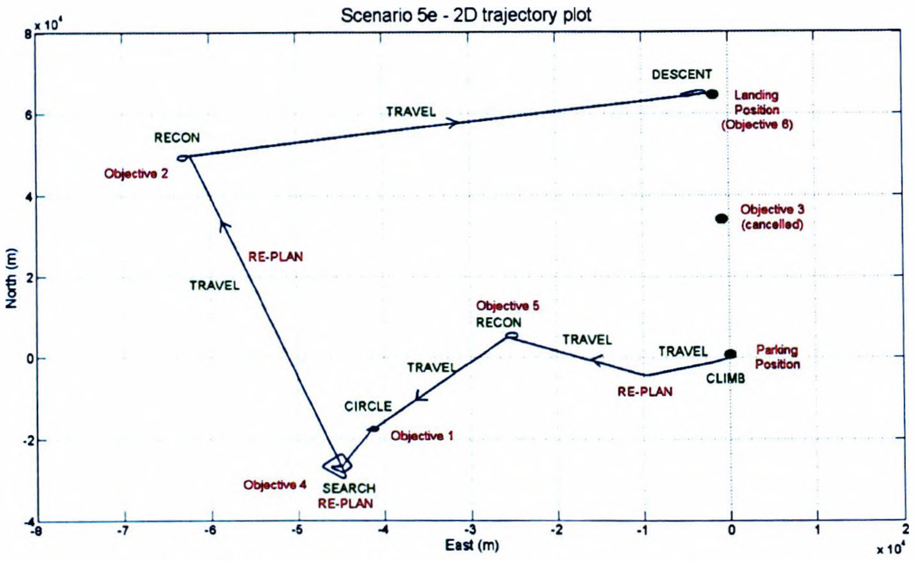


Figure 7.18. 2D flight trajectory for Scenario variation 5e.

Figure 7.18 shows the 2D trajectory plot obtained from the simulation of scenario variation 5e. Highlighted features include the take-off and landing positions, the position of Objectives and the UAV position when the re-planning events occur. The final flight plan consists of 29 Actions, and not all of these are labelled on the plot. Due to the slightly larger scale of the distances covered during this scenario, the trajectories

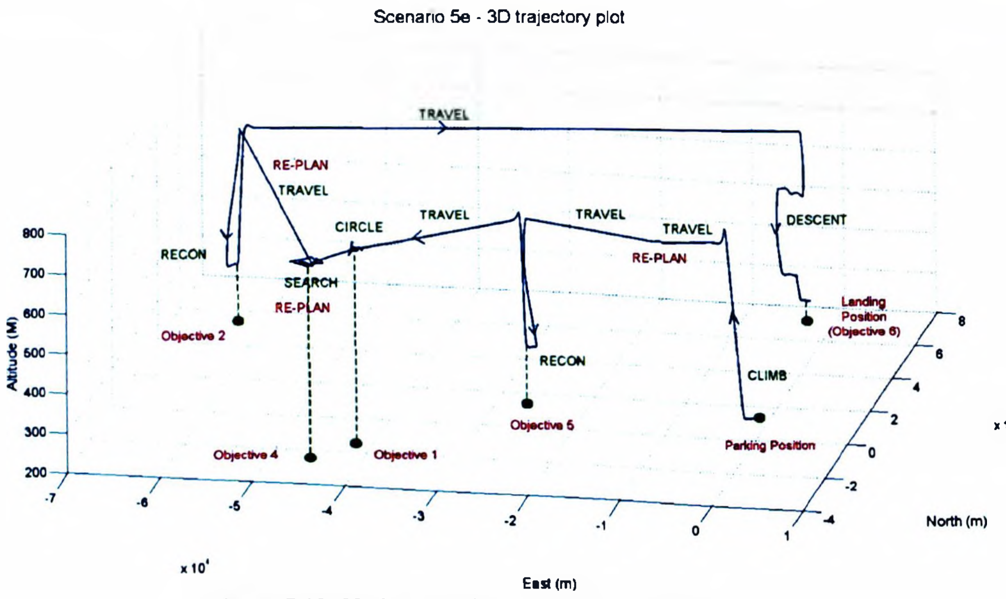


Figure 7.19. 3D flight trajectory for Scenario variation 5e.

regarding Recon, Attack and Circle Actions are not clearly visualized. It is instead possible to notice how the UAV deviates when re-planning without commitment to the current Objective, while no deviation is seen when re-planning with commitment. The scale of distances is still not large enough to highlight the fact that Travel Actions occur along great circle routes rather than loxodromes. The position of the cancelled Objective 3 is also noted in the plot.

Figure 7.19 shows the corresponding 3D trajectory plot. The plot shows clearly the trajectory undertaken by the UAV during the two Recon Actions and the Descent and Landing Action. The ground position of Objectives is noted in the plots for improved clarity.

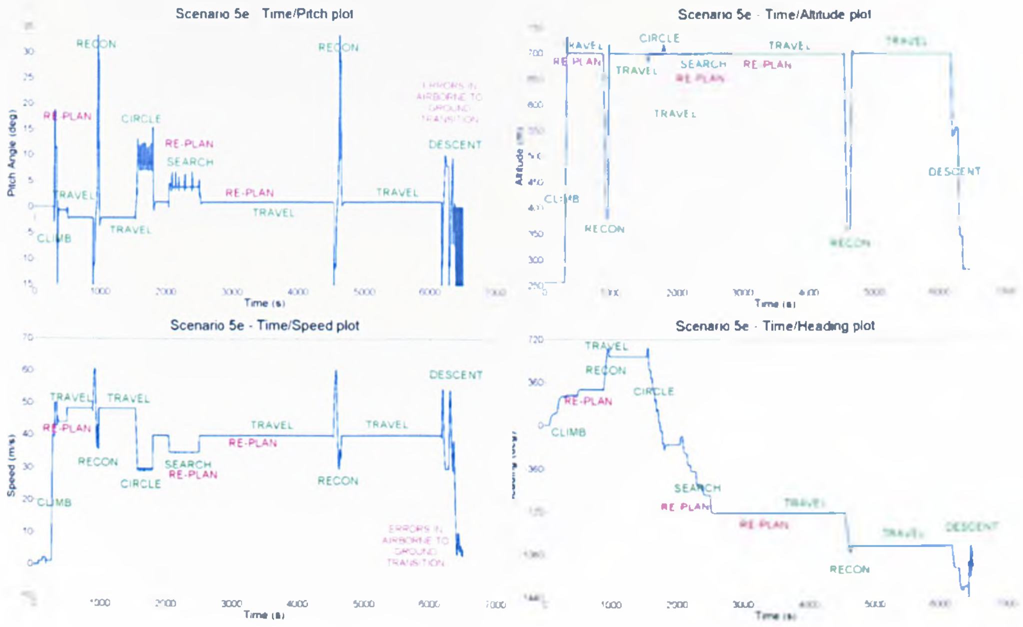


Figure 7.20. State variable plots for Scenario variation 5e.

Figure 7.20 shows four plots of relevant state variables for the scenario: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. Notable features that can be observed in the Time/Pitch Angle plot are the first re-planning event (because the event causes an increase in flight speed, the pitch angle to maintain altitude is reduced), the Circle Action and the Travel Actions related to the Search Objective (in both cases, it is possible to observe the behaviour of the UAV when performing several turns). It is also possible to notice the simulation errors regarding the airborne to ground transition. The Time/Altitude plot is straightforward; the Circle Action is performed at the Cruise Altitude of 700 m, and during it the autopilot must compensate the tight turns that are being executed; the only significant altitude changes occur during the Climb and Descent Action, and while the two Recon Actions are executed. The Time/Speed plot reveals an important characteristic of the flight plan; due to the time priority of Objective 1, the initial flight plan expects a 10% flight speed increase (up to 44 m/s) to ensure that the priority is respected. When Objective 5 is added and selected for immediate execution, the Planner further increases the flight speed to compensate this; flight speed until Objective 1 is reached is increased by 21%, which means that Travel Actions are performed at 48.4 m/s rather than the Cruise value of 40 m/s. It is also possible to notice that this increase is not sufficient to ensure that the time priority is respected, but significantly reduces the gap (Objective 5 is reached at time 1535 sec instead of the required 1500 sec). The rest of the flight plan is executed at normal speeds; it is possible to notice the simulation errors caused by the airborne to ground transition. In the Time/Heading plot, it is possible to notice the circular patterns undertaken by the UAV during the Circle Action and during the Travel Actions related to the Search Objective. Also, while the first re-planning event triggers a direction change, the second and third events do not (immediately) change the flying status of the UAV.

7.6 Scenario 7

Scenario 7 is a scenario that is not aimed at demonstrating specific SAMMS capabilities; its significance resides in the fact that this is an externally designed scenario, provided by an industrial research partner that did not have any knowledge of the SAMMS architecture and capabilities. The scenario does not present variations, does not involve any re-planning event and does not trigger intervention from the MMA. It was used during the Planner Agent test campaign (see Section 4.4.7) and also for the Execution Agent testing campaign.

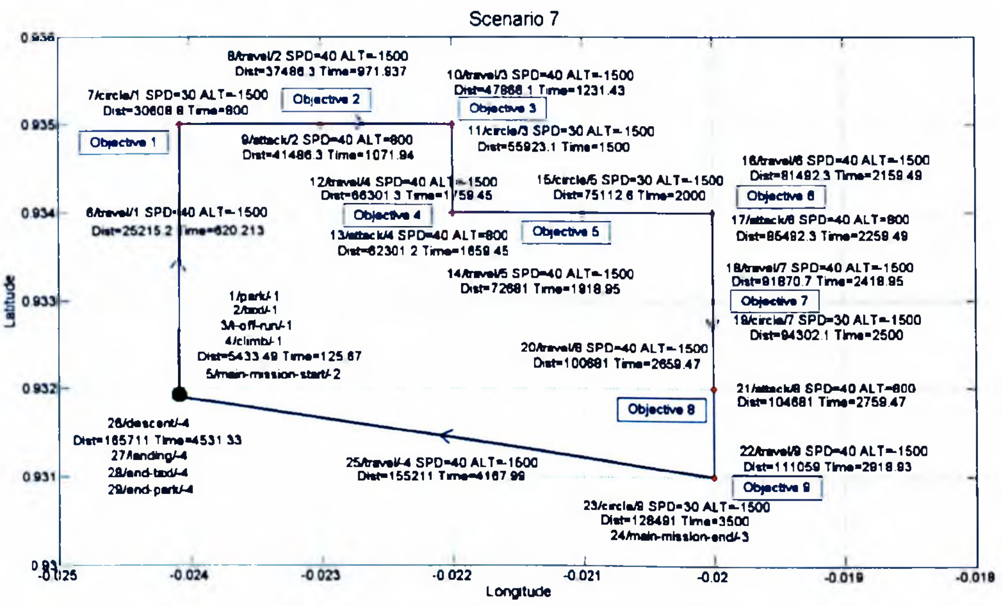


Figure 7.21. Plot of flight plan for scenario 7.

Scenario 7 involves nine Objectives, including five Orbit Objectives (Objectives 1, 3, 5 7 and 9) and four Target-Attack Objectives (Objectives 2, 4, 6 and 8). The Objectives are to be executed in the order 1-2-3-4-5-6-7-8-9 and basically reproduce the behaviour of a UAV that has to deliver four weapon payloads and each time loiters waiting for the desired attack time; the UAV is also scheduled to loiter prior to landing. The time limits for the Orbit Objectives are respectively 800 sec, 1500 sec, 2000 sec, 2500 sec and 3500 sec. Figure 7.21 shows the flight plan for the scenario; the UAV takes off from Sheffield airport and the targets of Attack Objectives are positioned at fixed locations, with the Orbit Objectives being placed close to them. The flight plan then sees the UAV landing back at Sheffield airport. The demanded altitude and speed profiles are also externally provided as part of the scenario, so the configuration parameters for SAMMS (the data contained in the Airframe Data and Health block, see Section 6.1.1 and Table 6.4) are accordingly modified (Cruise Altitude is set to 1500 m above sea level, Attack Altitude is set to 800 m above ground, Orbit altitude is set to be the same as Cruise Altitude, Attack Speed is set to 40 m/s as Cruise Speed, and Distance values are modified to allow for smoother trajectories).

Figure 7.22 shows the 2D trajectory plot obtained from the simulation of scenario 7. Highlighted features include the take-off (and landing) position and the position of

Objectives. The final flight plan consists of 29 Actions, and not all of these are labelled on the plot. Due to the different flight parameters used for this scenario, the trajectories regarding Attack Actions involve large turnarounds (the Attack Distance parameter is set to 2000 m instead of the usual 1300 m); instead, the Circle Action trajectories are tighter than usual (Circle Distance is set at 500 , instead of 600 m). These values are used because of the higher altitudes and large altitude changes required by the scenario altitude profile definition; without a longer space to descend, Attack Action would require several turnarounds to reach the correct altitude.

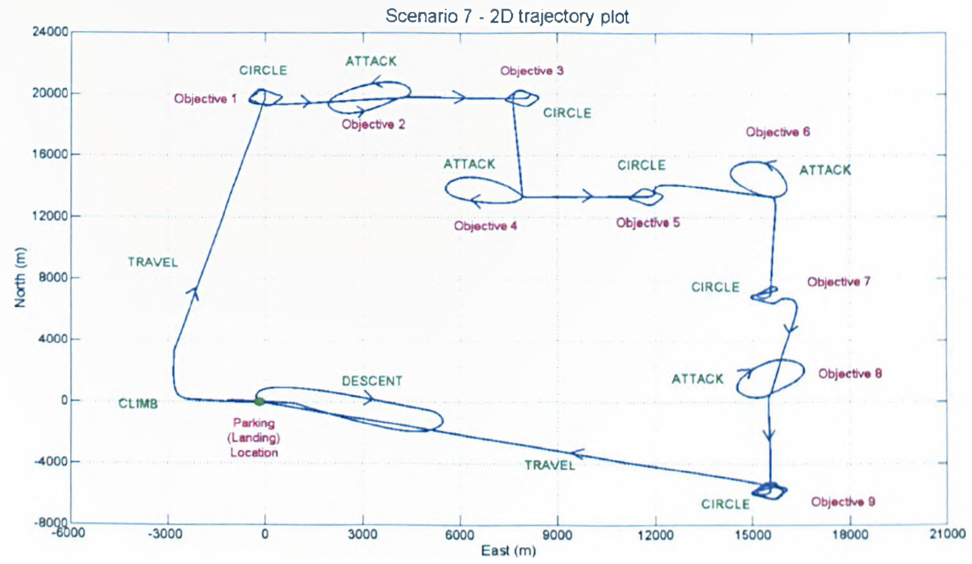


Figure 7.22. 2D flight trajectory for Scenario 7.

Figure 7.23 shows the corresponding 3D trajectory plot. In the plot, it is possible to notice the stretched Climb and Descent profiles (longer than in the other scenarios, since the Cruise Altitude is 1500 m rather than the usual 700 m). Also, the ground position of all Objectives is noted for improved clarity. Circle Action are carried out at the Cruise Altitude and are difficult to plot because of the small Circle Distance.

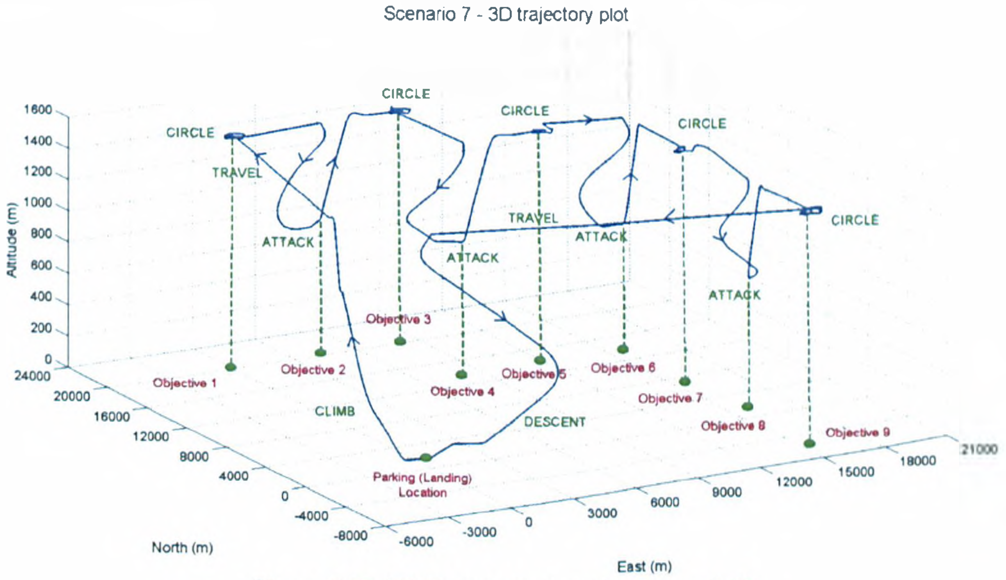


Figure 7.23. 3D flight trajectory for Scenario 7.

Figure 7.24 shows four plots of relevant state variables for the scenario: the Time/Pitch Angle plot, the Time/Altitude plot, the Time/Speed plot and the Time/Heading plot. The Time/Pitch Angle plot shows clearly the heavy compensation that is required from the Altitude-Hold autopilot in order to maintain altitude during turns; during Circle Action, the UAV turns several times, and each times this induces a disturbance which must be compensated. It is also possible to notice the simulation errors regarding the airborne to ground transition, which are present as in most other scenarios. The Time/Altitude plot is straightforward; it can be noted that the different altitude values compared to the other scenarios do not impact SAMMS' ability to plan and execute a flight plan. Travel and Circle Action are performed at 1500 m above sea level, while Attack Actions are performed at 800 m above the ground. The Time/Speed

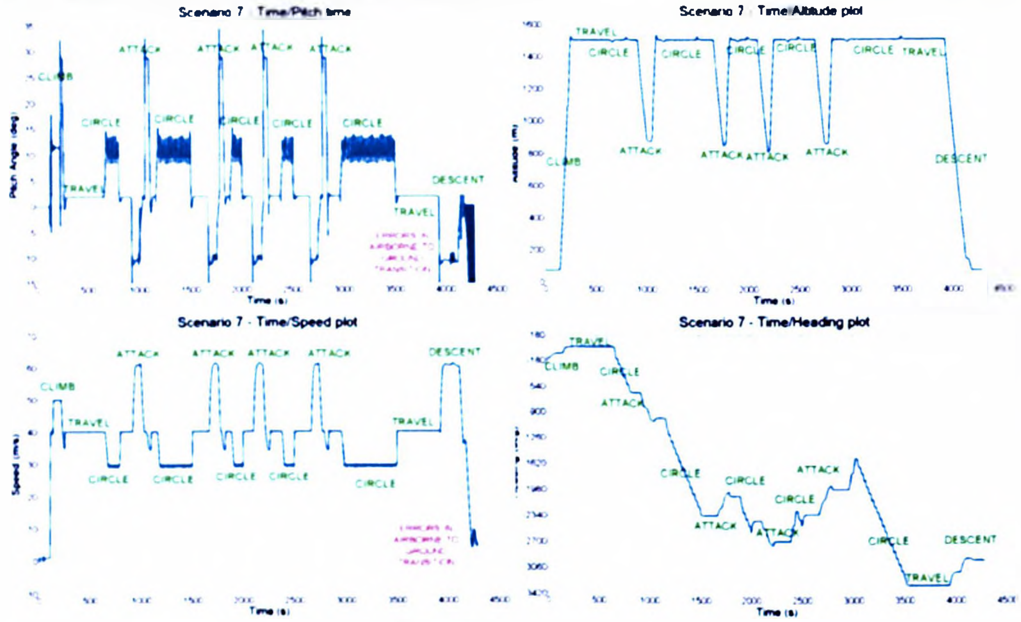


Figure 7.24. State variable plots for Scenario 7.

plot does not yield any surprises; the scenario does not involve time priorities, thus the nominal speed values are maintained. It is to be noted that Attack Actions should be performed at 40 m/s, but due to the large altitude change speed remains high throughout the Actions, until the UAV starts climbing back to the Cruise Altitude. When climbing back, the UAV is already performing a Travel Action; the demanded speed should be Cruise Speed (40 m/s), but due to the steep climb this speed cannot be kept until the Cruise Altitude is reached and the UAV resumes level flight. It is also possible to notice the simulation errors regarding the airborne to ground transition, which in fact cause the UAV to never be able to fully stop after ground contact is ensured. The Time/Heading plot is straightforward and reveals that the UAV has to perform several 360 turns, because of the scheduled loitering time during which it has to wait (corresponding to the Circle Actions). Finally, it is possible to notice that the time estimates for the flight plan are very close to the actual execution times. This precision is due to the nature of the mission: because several times the UAV is loitering waiting for the specified time to act, the errors in time estimates are not allowed to build up.

7.7 Concluding remarks

In this chapter, the final results of tests performed on the SAMMS architecture were shown. Unlike the separate testing campaigns for each agent, presented in Chapters 4, 5 and 6, these tests occurred with the entire architecture functioning and involved the simulation of entire missions. Such tests basically summarize the entire work presented in this thesis, since they bring forward not only the functionality of each SAMMS components, but also the effective integration between the components.

The tests are carried out by executing entire missions based on the test scenarios used during the Planner Agent and Mission Manager Agent testing campaign. A set of six scenario variations was chosen, with the intention of demonstrating the largest possible array of capabilities. For each presented scenario, graphical plots of the flight trajectories and of some state variables are provided, accompanied by an analysis of the results.

This chapter concludes the descriptive part of the thesis. The next chapter will present conclusions that can be derived from the thesis and introduce possible future work that could be performed on SAMMS.

8. Conclusions and Future Work

Throughout this thesis, the dominating subject has been autonomy: all of the presented work is aimed at developing and demonstrating the capability of a software system (SAMMS) to manage and perform a mission with clearly defined goals without human supervision.

The project was characterized by several decisions:

- in contrast with most of the on-going research work, SAMMS was not intended to focus on the management and control of multiple UAVs; instead, the project was scoped to achieve a level of autonomy between levels 4 and 5 of the Clough classification
- the approach used for development was to be the integration of cognitive intelligent agents (based on the Soar architecture) with more traditional control algorithms (particularly autopilots)
- rather than demonstrating specific capabilities such as payload and sensor management, SAMMS is meant to provide a foundation for the integration of such capabilities into an overarching architecture
- the system was designed to be usable by personnel without specific training (e.g. only generic knowledge of the system performance is required, to avoid the assignment of mission goals outside the UAV capabilities)
- the system was meant to automatically update its flight plan during flight, according to newly gained knowledge
- while keeping SAMMS as generic as possible, the focus was to be placed on fixed-wing UAVs (even though many concepts are applicable to other vehicle types)

The Soar-based Autonomous Mission Management System (SAMMS) presented in this thesis was demonstrated to possess these characteristics: SAMMS is controlled by the User only at the highest level of control (assignment of mission objectives), is capable of developing a flight plan addressing most of the issues highlighted by the requirements of levels 4/5 of the Clough classification (fault mitigation, collision avoidance, etc.) and to update it in real-time (e.g. while the mission is in progress). It does so through the use of a multi-agent system based on three interacting Soar agents that are interfaced with low-level control algorithms that provide specific functionality (in particular the autopilot that directly controls the UAV). It is however to be stated that certain specific functions (such as fault detection and payload/sensor management) have not been developed; instead, the chosen approach was to describe how these systems can be integrated with the SAMMS architecture.

In this chapter, an analysis of the results presented throughout the thesis will be provided. The first part of the chapter (Section 8.1) focuses on the analysis of the work and highlights significant achievements of the project. The second part (Section 8.2) instead focuses on detailing possible improvements for SAMMS and on prospecting a feasible implementation strategy for the entire system.

8.1 Project summary and achievements

In this Section, SAMMS will be analyzed using the characteristics defined in the chapter introduction as a reference. These characteristics can be seen as generic requirements for the system and the analysis will focus on detailing how each characteristic is achieved (or how its achievement is planned).

8.1.1 Single-UAV autonomy

One of the stated goals for the SAMMS project was to achieve an autonomy level standing between levels 4 and 5 of the Clough classification. The classification clearly specifies the capabilities that a UAV must possess for each autonomy level.

Levels 0 and 1 are centred around basic flight capabilities; be it through direct piloting (level 0) or through pre-determined flight plans (level 1), the UAV has to be able to fly. This involves two main sets of capabilities: sensing capabilities (mostly flight data, such as UAV speed, attitude and position) and control capabilities (used to actuate pilot or autopilot commands).

Levels 2 and 3 are instead focused around failure management: the UAV has to be able to autonomously detect and react to faults within the UAV itself, be it with pre-programmed plans (level 2) or with a real-time generated response plan (level 3).

Level 4 introduces off-board awareness: the UAV has to be able to consider externally provided data regarding the current situation and use it to update its flight plan. Level 5 expands on this concept: information from on-board sensors must be fused with externally provided information, and the information must be used to improve and update the flight plan.

Within SAMMS, level 1 autonomy is achieved by the Execution Agent coupled with the low-level control functions (the autopilot). Using only these components of the system, it is possible to completely execute a pre-planned mission, when defined in the appropriate format (a series of Actions, as defined in Section 3.1). The results presented in Chapters 6 and 7 demonstrate that the Exag and the autopilot are capable of correctly managing the execution of a flight plan; the only concerns arise from the phases where the simulation presents errors (e.g. Take-off and Landing). For these phases, results are positive but cannot be considered definitive evidence that the UAV is correctly manoeuvred; further improvements might be needed, though this is unlikely. It is important to highlight that a key point to demonstrate was the ability of a Soar agent to react to the fast events that are typical of flight; the Execution Agent is particularly stressed in this sense, since it has to interact directly with the autopilot. Through the tests, it was proven to reliably command the autopilot during all flight phases.

Regarding failure management (for levels 2 and 3), at present SAMMS does not address it completely. Fault detection and evaluation are not currently implemented; it is expected that an external on-board system will able to perform the fault detection function, with an intermediate layer performing fault evaluation. Ultimately, fault evaluation can be seen as the ability to predict the effect that a fault will have on high-level UAV performance. This is translated into a set of performance parameters, which the high-level planning functions (the Planner Agent and the Mission Manager Agent for SAMMS) consider while developing flight plans. For example, a fuel leak would be evaluated as an increase in fuel consumption; while generating a flight plan, the Planner will take account of the reduced range caused by the leak, and update the plan accordingly. In this sense, within SAMMS failure management has been addressed at the system level, so as to include fault information in the flight plan generation process, but not specifically (the fault detection and evaluation subsystems are not developed). This is justified by the fact that such systems are largely platform-dependant: once a platform is selected for the implementation of SAMMS, its fault detection system will be available and the fault evaluation layer will be developed based on it.

The Planner Agent and the Mission Manager Agent are meant to fully accomplish levels 4 and 5 of autonomy. Off-board awareness is introduced within SAMMS through the concept of Entity: in practice, this is a format to provide all information regarding the external environment. Once formatted in this manner, SAMMS is capable to update the flight plan according to the provided data. Currently, the most important function performed on this is threat avoidance (which also includes collision avoidance). In this sense, SAMMS achieves level 4 of autonomy; however, level 5 would require the ability to predict collisions with moving Entities, and this is not currently implemented within SAMMS. Static threats are correctly avoided, but when the threats are dynamic the flight plan does not take into account expected movement. The Entity concept means that the origin of off-board information is irrelevant: an Entity will be treated in the same manner whether it is detected by an on-board sensor or provided by an external source. It is to be noted that a dedicated subsystem is needed in order to format sensor and external data into Entities; this is not currently implemented. The same type of subsystem could be used for both data sources, depending on the nature of the original data.

The flight plans generated by SAMMS are not optimal, but they are meant to possess desirable characteristics:

- time constraints can be placed on mission objectives, and the flight plan will try to respect them as possible
- the shortest path is searched for, within the limitations of the Nearest-Neighbour algorithm
- flight time and fuel consumption estimates are calculated, and the flight plan is updated according to these (ensuring that fuel is sufficient and that time constraints are respected)

Some of this functionality is expected at level 5 in the Clough classification, and the ability to deal with time constraints is not specified in the Clough classification, therefore constituting a significant improvement.

In general, when provided with appropriate failure management and sensor/data management subsystems, the SAMMS architecture has been demonstrated to exceed the requirements for level 4 of the Clough classification. Regarding level 5, the only unsatisfied part of the requirements is the one related to basic cooperation with other UAVs; in Clough's terms, the UAV should be able to "accomplish a group tactical plan as externally assigned and fly in formation in non-threat conditions". This is compensated by the ability to specify time constraints on mission objectives and flexibly adapt the flight plan according to these. It should also be noted that the Clough classification is not linear and relies on a generic definition of capabilities; as such, a certain level of uncertainty in defining the actual autonomy level can always be expected.

8.1.2 Development approach

The SAMMS architecture is developed using a Simulink model as the underlying platform over which Soar intelligent agents and additional control software are placed. Using appositely developed C++ code, Soar agents are executed within a Simulink model as S-functions; multiple Soar agents are used in the architecture, and these are interfaced with each other through Simulink model signals. The agents are also interfaced with additional components directly modelled in Simulink, specifically the Autopilot, the UAV model, the New-Plan Trigger function, the Objective Mix block and the functions for output visualization.

Results of the simulations prove that the Soar/Simulink approach is viable. Furthermore, it is possible to speculate that the computational requirements for SAMMS will not exceed the computational resources available on a typical PC/104 board. This was not proven but is derived from previous experiences. Prospective implementation will be detailed further in Section 8.2.

8.1.3 Integration of other software

An autonomous UAV with level 5 capabilities in the Clough scale must be able to autonomously manage failures, on-board sensors and payloads and on-board energy. Within SAMMS, these aspects have not been fully developed. Rather than on implementing the capabilities, the focus was placed on allowing to integrate externally developed systems into the architecture; the goal is to generate flight plans that take into account information provided by these systems and to ensure that the SAMMS software is capable of giving correct instructions to them.

Consistent research has been carried out on all of the mentioned subjects; most of this research is platform dependent, so it is difficult to integrate them at an early stage of development. Once a final target platform is established for the SAMMS architecture, it should be possible to find suitable externally provided systems that accomplish these functions and to integrate them with the SAMMS software through the appropriate interfaces.

Failure management is an activity where SAMMS is mostly receiving data from the dedicated subsystem. To integrate it with the flight plans being generated, this data has to be formatted according to specific requirements. In particular, the concept of "performance index" was used; the SAMMS agents do not need to know in detail the nature of a fault, but they rather need to know what the effects of the fault will be in terms of UAV performance. Thus, each detected fault is interpreted by reducing the value of related performance indices; for example, a fuel leak would result in an increase of fuel consumption, and the Planner would take this into account, ensuring that the UAV does not exceed its new (reduced) operational range. In general, simple algorithms derived from a Failure Modes Effects and Criticality Analysis (FMECA) database should be sufficient to perform the conversion from the output of a fault detection system to the performance indexes needed by SAMMS. The System-Level Prognosis component of the ASTRAEA work presented in Section 2.7 is an example of how this might be implemented.

Sensor and payload management is an activity which requires mutual interaction between SAMMS and the subsystem. Data captured by sensors (these are intended to be sensors for situational awareness, such as radar and electro-optical, rather than flight sensors, such as an altimeter or a gyro) and payloads must be formatted using the Entity construct so that it can be used by the Planner during the flight plan generation process. For example, if a new target is detected by the radar, a new Entity should be passed to SAMMS, including all known information regarding it (target type, position, movement, behaviour, etc.). On a real-world implementation of SAMMS, a new intermediate layer will be needed to format data provided by the sensors and payloads using the Entity construct; this is not expected to represent a major issue, but is strictly platform dependent (depending on the sensor or payload type). Notably, the Entity construct allows for seamless fusion of data derived from multiple sensors. SAMMS is not only receiving data from the sensors and payloads, but in certain cases must also instruct them regarding the current stage of the mission. For example, on a video gathering mission, SAMMS must activate the camera payload when the target is being flown upon, possibly also having to provide targeting information (e.g. radar detects the target position, SAMMS flies over it and instructs camera regarding its position). Summarizing, sensor and payload management is not trivial, but details can only be discussed when a platform has been selected; at present, the SAMMS architecture uses the Entity format for data gathering and has demonstrated limited ability to give commands directed to a payload management subsystem.

On some types of UAV, energy management might be a significant issue. On-board energy might be extracted directly from the engine, thus signifying a reduction in UAV capabilities when a high-powered payload is used. Such a reduction can be taken into account by SAMMS through the performance indices used for failure management. More complex energy management is currently the subject of research studies, but at present energy management is typically tasked to the pilots on manned aircraft; thus, the SAMMS architecture currently does not include specific parts that could be used to integrate an energy management subsystem, apart from the performance indexes used for failure management.

8.1.4 User interface and need for supervision

A SAMMS-based UAV is operated by assigning Objectives to it; the only other information that the operator should provide is related to take-off and landing, although this could be largely automated (for example, with a database of runways and the corresponding take-off and landing instructions, allowing the pilot to simply choose a runway for take-off and eventually landing).

At present, Objectives are assigned through a Simulink interface, specifically the *Scenario* block described in Section 3.3. While this is convenient at the simulation stage, an actual implementation of SAMMS will require the development of a dedicated interface, allowing for easy and rapid selection of Objectives and their properties. It is clear that depending on the Objective type, certain properties will have to be defined by the operator, while others might be assumed with default values and only modified by the operator if desired. For example, for a Target-Analyze Objective the operator must certainly define the Objective Type and Target properties (Objective Tag is automatically assigned); other relevant properties such as Time and Execution Priority are defined only if desired, with default values being assigned as 0 (no Time Priority) and 3 (Objective to be executed even if the mission could be compromised, but not at the risk of damaging the UAV).

Once Objectives are assigned, SAMMS can fully execute the mission without further operator intervention; the operator maintains the possibility to change mission Objectives throughout the flight, changing, adding or cancelling them as required. SAMMS will consequently update the flight plan to reflect the operator's decision. Without any supervision, the decisions taken by SAMMS will be entirely predictable within the defined mission parameters. It must be noted that SAMMS does not implement any learning capability, and therefore will only react to events that fall under the categories for which a response has been programmed. For example, SAMMS will always react to threats by avoiding them, but at present it does not take into account their movement information, so it will not avoid them considering their expected trajectory.

The flight parameters defined in the Airframe Data and Health block described in Section 3.3 describe the performance of the UAV from a mission management point of view. SAMMS uses these parameters to compute distance, time and fuel estimates for the flight plans it generates. The estimates are then used to correct the flight plan when inconsistencies are detected. For example, if the operator assigns a mission which is estimated to require more fuel than currently on-board, SAMMS will adjust the flight plan accordingly (flight speed reduction or Objective cancellation), but then the mission will not be performed according to the chosen parameters. The same can happen if an unrealistic time priority is assigned: SAMMS will try (unsuccessfully) to respect it, and the flight plan will consequently be modified, possibly resulting in clearly non-optimal behaviour (such as taking a much longer route to cover the mission). In general, the UAV operator should possess generic knowledge regarding the UAV capabilities, in order to avoid such situations; apart from this, it can be stated that a SAMMS-based UAV will not need specific training to be operated.

8.1.5 Autonomous updating of flight plan

All flight plans generated by SAMMS are automatically constructed from the Objective list and the knowledge provided through the Entity list. When the system is initialized, a first plan is generated, then this plan is updated throughout the mission when needed.

A flight is updated by repeating the same plan generation process used for the first flight plan, however the new flight plan takes into account the parts of the previous flight plan that have already been executed. This is ensured by the *old-plan* state described in Section 4.2.1: Actions that have already been performed are copied directly into the new plan.

The process of updating a plan is entirely autonomous: no intervention from the UAV operator is needed (although the re-planning event might be triggered by the operator changing the Objective list).

The New-Plan Trigger function is very important in avoiding an excess of re-planning events; ideally, only major changes or a series of minor changes will trigger re-planning. For example, re-planning should not be triggered if a dangerous Entity changes its position slightly (less than 100 m), but it should if the position change is larger (the Entity area of effect might now be within the UAV's path). In fact, at present the plan generation process is triggered by single changes, as described in Section 3.3; the Planner Agent and the Mission Manager Agent are triggered separately, and the situation changes that cause triggering are different for each agent. It is to be noted that when a plan is generated, the situation is recorded, so that the comparison is done between the current situation and the situation of the time of plan generation. A possible improvement over this function will be discussed in Section 8.2.1.

8.1.6 Applicability of SAMMS architecture

During development of SAMMS, a choice was made to focus on building a system to control a fixed-wing UAV rather than another vehicle type. However, many of the concepts that are used within the architecture are not limited to fixed-wing UAVs and could be easily ported to other UAV types (rotorcraft, lighter-than-air) or even other vehicle types (Unmanned Marine Vehicles, Unmanned Ground Vehicles).

This is particularly true for the high-level components of the SAMMS architecture: the Planner Agent and the Mission Manager Agent could be easily adapted for other UAV types or for UMVs. To be adapted for a rotorcraft or lighter-than-air UAV, SAMMS would require some changes to the types of possible Actions (particularly during the take-off and landing phases, but generally the Action types are equivalent. Unmanned Marine Vehicles could be treated easily: in this case the take-off and landing phases are not necessary; an important distinction would be the one between surface

UMVs and underwater UMVs (or UUVs). For surface UMVs, planning would have to take into account that movement basically occurs only in two dimensions; for UUVs, depth is used instead of altitude, but conceptually control happens in a very similar manner to UAVs. Unmanned Ground Vehicles are inherently more complex, because of the added constraint of navigating the ground; for these, the mission management activity is not as relevant compared to the low-level control and navigation issues. Because of this, the applicability of SAMMS to UGVs cannot be ensured: even only using the planning functionality, the preponderance of low-level control could possibly result in conflicts between high-level planning functions and low-level controls.

In all cases, adaptation of SAMMS would require some changes to the Planner Agent and the Mission Manager Agent, and a complete rewriting of the Execution Agent and of low-level control algorithms. For example, for a rotorcraft UAV the Circle Action would better be renamed as a Hover Action; the autopilot would have to be able to perform hovering (which is not at all a trivial task, especially in windy environments) and the Execution Agent would need to instruct the autopilot correctly, likely engaging a dedicated autopilot mode instead of choosing four waypoints as is the case for fixed-wing UAVs.

In general, the concepts underlying SAMMS are generic enough that little adaptation would be needed to adapt the high-level planning functionality to other vehicle types. Low-level functionality is instead naturally platform-dependant, and specific algorithms would have to be developed (the basic structure of the Execution Agent could be re-used, but all Actions would have to be executed differently).

8.2 Future work

At the current stage of development, the Soar-based Autonomous Mission Management System only exists as a set of software agents which have been tested within a simulation environment. This is naturally far from ideal and in fact the system was designed with prospective implementation in mind.

In this section, possible future work that could be done on the SAMMS architecture will be detailed. Three main advancement routes will be covered: improvement of SAMMS capabilities, development of additional components and hardware implementation.

8.2.1 Improvement of SAMMS capabilities

All work regarding the SAMMS architecture was done as part of this PhD project, which is not part of a larger project. Consequently, during development several decisions had to be made in order to limit the amount of work required to reach a stage where meaningful conclusions might be extracted. Generally, these decisions involved simplifications of the project or limitations to the algorithms, especially when a more complex version of the system would not yield very meaningful results. For example, at present the parallel track search pattern (see Section 4.2.5) cannot be orientated: while this is technically feasible (requiring more complex equations but little more), it was not perceived to bring substantial improvement with respect to the scope of the project. However, it is to be noted that such functionality would likely be required from an actual system.

In this subsection, several possible improvements that could be applied to SAMMS but were left out at the current development stage will be detailed. There is a long list of possible improvements, which will be now described. The list does not include the development of architecture components which were overlooked at this stage, such as failure detection and evaluation and sensor/payload management. These will be addressed in Section 8.2.2.

Improvement of connection between Travel Actions and other Action types

The current implementation of the Execution Agent does not cover an ideal flight trajectory when performing Target-Recon, Target-Attack and Descent Actions. The biggest issue is the fact that at present these Actions are preceded by a Travel Action that will bring the UAV directly above the target (for Target-Recon and Target-Attack) or above the runway (for Descent). The UAV has then to turn around in order to reach the waypoints that are used to perform the actual manoeuvre, usually having to change

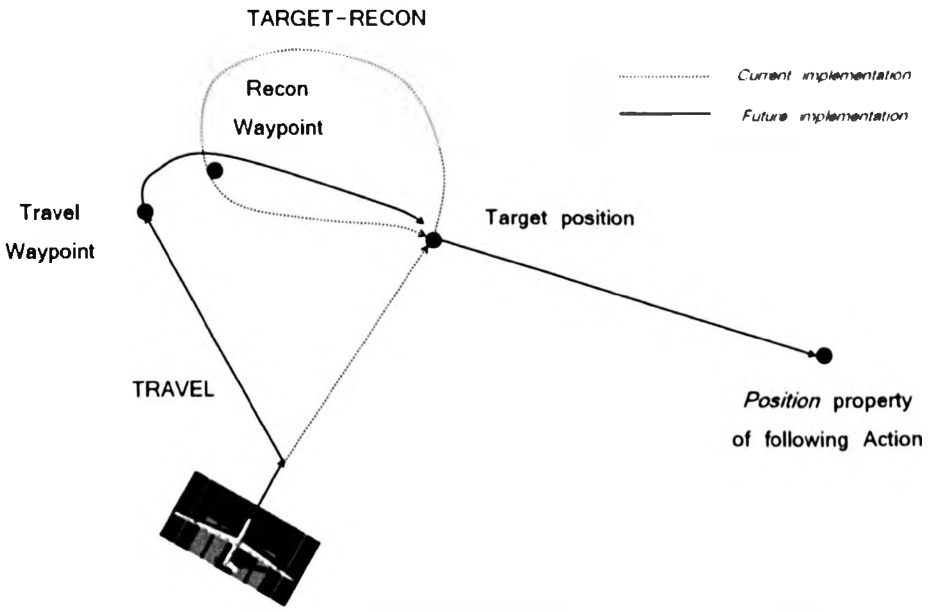


Figure 8.1. Example of planned Execution Agent improvement.

altitude at the same time. An improvement over this situation would be to execute the Travel Action preceding a Target-Recon/Target-Attack/Descent Action in a different manner; for example, at a predetermined distance from the Travel Action destination, SAMMS could turn the UAV towards a waypoint that will ensure a smooth trajectory when the Travel Action finishes and the other Action begins. Figure 8.1 shows an example of how this could be done for a Target-Recon Action; at some point during the Travel Action preceding the Target-Recon Action, the UAV would turn towards a first waypoint (also descending at the same time), and from then it would turn towards the Recon waypoint and then pass over the target. Similar adjustments could be made for Target-Attack and Descent Actions as well (they are not really necessary for Circle Actions). In fact this modification only requires changes to the Travel Action state in the Execution Agent.

Improvement of plan-sequencing with a better heuristic than NN-algorithm

When selecting the order in which Objectives should be executed, SAMMS uses the Nearest Neighbour algorithm to minimize the distance covered during the mission (albeit also taking time priorities into account). The NN algorithm was chosen because of its simplicity and its fast execution time; however the results it yields are not guaranteed to be optimal, and in certain cases its performance can be very poor (even giving the worst possible answer). Various other heuristics are available to solve what is essentially a Travelling Salesman Problem (for example, the Boruvka and Lin-Kernighan algorithms, see (Applegate et al, 2007)), so a possible improvement strategy for SAMMS is the inclusion of a better plan-sequencing algorithm. This algorithm should be easy to implement within Soar and to integrate with the algorithms that guarantee that time priorities are respected. The improvement would involve the modification of the *plan-sequencing* state in the Planner Agent.

Introduction of new Objective types that require specific flight trajectories

At present, five Objective types have been implemented within SAMMS; for an actual UAV, more Objective types could be required, differing in various respects from the ones already defined (especially regarding flight trajectory). This is also dependant on the specific UAV which is controlled by SAMMS; every platform could have highly specific uses that will require the definition of new Objective types. Defining a new Objective type should not be particularly troublesome: while requiring modifications to both the Planner Agent and the Execution Agent, the modularity of the Soar architecture and of the SAMMS system allow for straight-forward addition of new Objective types.

Improvement of take-off and landing sequences

The take-off and landing sequences (that is, not only the Take-off and Landing Actions, but also the related Taxi, Climb and Descent Actions) were implemented using algorithms that result in relatively simple flight trajectories. In actual flight, take-off and landing are the most dangerous phases and a large amount of legislation is present, determining many details regarding how an aircraft should perform these manoeuvres. While precise legislation is not yet available for UAVs, it can be expected to be when civilian use of UAVs will become common. Preparing for this, the take-off and landing sequence for SAMMS should be improved, in order to take account of such legislation and of the performance requirements it will impose. It is also to be noted that in certain cases UAVs do not take off from a runway, but are instead launched, either by hand or with a catapult, or released directly into the air. In general, the Execution Agent should be modified in order to take account of legislation and allow for the possibility of non-runway take-off and landing.

New search pattern types

Two types of search patterns are currently implemented within SAMMS: the classical parallel track pattern and an expanding diamond spiral pattern. This is not an exhaustive list and several more pattern types could be implemented: for example, the sector search pattern and the square spiral search pattern. This would involve adding the relative productions within the *actions-definitions* state, but does not otherwise require further modifications.

Possibility of orientation for search patterns

The possibility to orient a search pattern is particularly relevant for the parallel track pattern. At present, the pattern can only be executed with the "searching" legs running vertical (along a meridian), starting from the north-west corner and finishing at the south-east one. Adding the possibility to orient the search area is just a matter of added mathematical complexity when calculating the position of waypoints. For example, at present the second waypoint of a parallel track pattern is derived directly from the coordinates of the north-west and south-east corners; calling (lat_{NW} lon_{NW}) and (lat_{SE} lon_{SE}) the coordinates of the corners, the second waypoint has coordinates (lat_{SE} lon_{NW}).

For an oriented search pattern, this is not valid, and appropriate spherical geometry formulas have to be used; however, apart from the mathematical complexity, there is no issue preventing this capability. The change would only require improving some of the productions in the *actions-definition* state (see Section 4.2.5). It is to be noted that with orientation being a possibility, a possible flight distance reduction strategy would involve flipping a search area, for example starting from the SE corner instead of the NW one. In Figure 4.23, the case where Objective 1 is executed yields a longer covered distance; this could be avoided by flipping the search. Such a decision could be made by the MMA, as an additional algorithm within the *modify-objectives* and *check-objective* states (see Sections 5.2.3 and 5.2.5).

Possibility to interrupt and reprise searches

Search Objectives are usually the ones requiring the longest distances and times to be completed; despite this, at present after a search is begun, SAMMS will either complete it or have to restart it from scratch (if a re-planning event causes a diversion). The possibility to interrupt and reprise searches would certainly represent an improvement for the system; for example, if an Objective with immediate priority is entered while the UAV is performing a Search Objective over a large area, this Search could be interrupted to perform the other Objective and then reprised from the point where it was interrupted. This is even more relevant for Search-and-Analyze and Search-and-Attack Objectives: newly added Objectives could be executed immediately, especially since the UAV will very likely be already close to them (being within "detection" range). It is to be noted that the modifications needed for this functionality are not trivial: it would probably require modifications to both the Planner Agent and the Execution Agent.

Designation of a number of targets to be searched for

When a Search Objective is being carried out, the number of expected targets could be known; for example, a parallel track pattern might be used to search for four vehicles within the defined area. Once four targets of the required type have been identified, the Search could be considered complete and thus interrupted, yielding fuel consumption savings. This improvement should be easy to implement and would be a good complement to the ability to interrupt and reprise searches.

Capability to consider Entity movement for threat avoidance

As observed in Section 8.1.1, SAMMS currently does not consider the movement of Entities when performing its threat avoidance function (even if movement information is available). Also, it does not consider movement info when performing target-related missions; it will automatically update its destination to reflect target movement, but does not plan in advance. Such capability would evidently be desirable and thus represents one of the possible improvements for SAMMS. It is to be noted that implementation of such planning abilities is complex, however research work on similar problems has already been performed and related algorithms might be integrated into SAMMS. A major difference in the problem definition lies in the fact that the Entity might be actively trying to evade the UAV (or collide with it); this evidently introduces a further layer of complexity, compared to the situation in which an Entity is moving with constant speed and heading.

Improvement of the fuel consumption model

The fuel consumption model currently implemented within SAMMS derives fuel consumption estimates directly from planned speed and distance; altitude is not considered. This is a significant oversight: a climbing UAV will use much more fuel than a descending one flying at the same speed. The fuel consumption model should be improved to take altitude changes into account; this would require changes to the *estimations* state in the Planner Agent.

Improvement of the New-Plan Trigger function

The New-Plan Trigger function is currently implemented so as to trigger re-planning when pre-determined input variables change (possibly by a specific amount). A possible improvement over this function would be the adoption of a point-based system: each change in the situation is awarded a number of points, the points are then summed and when a threshold is reached re-planning is triggered. This would allow to still have single changes that cause re-planning, but also to consider re-planning when multiple minor changes are happening. Furthermore, the behaviour of SAMMS could be tailored depending on the mission needs by changing the threshold value: for example, a mission in a high-risk environment would involve a low threshold and consequently a larger number of re-planning events. This is an improvement that does not directly affect the Soar agents, as the New-Plan Trigger function is performed externally.

8.2.2 Development of additional components

Throughout the development of SAMMS, the availability of specific components such as a fault detection system or a payload management system was assumed; the system was implemented with pre-determined ways to integrate such components, but the components themselves were not developed.

In this subsection, ideas regarding the implementation of such systems will be outlined. Three subsystem types will be treated: fault detection, sensor/payload data processing and sensor/payload management.

Fault detection

The detection of failures within a system is non-trivial and has attracted (and continues to attract) a consistent amount of research. An even more complex task is the prognosis of faults (that is, the prediction of when a fault might arise and how it might evolve). At the state of the art, the combination of improved sensors, high available computational resources and statistical studies regarding faults and their characteristics resulted in the development of impressive fault detection systems, especially for safety-critical application.

For example, gas-turbine engines are equipped with a set of sensors and a digital controller; by fusing the information from the sensors with the knowledge gained through Failure Modes Effects and Criticality Analysis (FMECA) statistical studies, the controller can automatically react to dangerous running conditions (such as a compressor stall or a surge) and to specific fault types (such a broken compressor blade). When automatic reaction is not possible, the fault detection system can relay a warning to the pilot, who can then decide on a course of action. For example, if a fuel leak is detected, causing gradual degradation of the engine and that can only be fixed on the ground, the engine controller will warn the pilot, who will be in the position to choose the best mitigation plan. In some cases, the pilot could decide to abandon its mission and return to base; in other cases, he might decide to ignore the fault (perhaps because the mission can be completed before the fault becomes too dangerous. The decision is not only dependant on the failure situation, but also on the generic situation of the aircraft.

On an autonomous vehicle, this decision-making cannot be delegated to a pilot, but has to be performed by the vehicle controller itself. The SAMMS architecture is built to be able to make such decisions during its plan generation process. It does so by using a set of "performance indices" that represent an evaluation of the aircraft's ability to perform its mission. These performance indices are taken into account during the flight plan generation process; for example, an engine problem might result in a limitation to flight speed, while a fuel leak could result in the cancellation of an Objective which is too distant.

For a real-world implementation of SAMMS, it is safe to assume that, whatever the actual platform, it will possess some form of fault detection capability. The question is then: how can the output of this fault detection capability be converted into the performance indices used by SAMMS? Two possible strategies can be theorized.

The first strategy is to use a system similar to the one realized within the ASTRAEA project and described in Section 2.7. In particular, the System-Level Prognosis functionality described in 2.7.2 represents the type of functionality that should be developed. It is however to be noted that the System-Level Effects described there are discrete, while ideally performance indices should vary in a continuous manner. The main concept remains the same: build an algorithm that embeds some form of FMECA database (a simple look-up table, or something more complex such as a fuzzy-logic based system), so that a system-level prognosis can be derived from the fault detection knowledge.

A second possible strategy would involve the use of another Soar agent, which based on the knowledge extrapolated from a FMECA database would perform the same type of calculations performed by the "traditional" algorithm. At present and considering the ASTRAEA experience, it is difficult to point out specific advantages that would make this approach preferable, but the option has to be considered.

Sensor/payload data processing

Within SAMMS, only a handful of variables regarding the external environment is provided directly: this is mostly comprised of information related to the current flight condition of the UAV, such as flight speed, attitude, altitude and position. All other knowledge which might be relevant to SAMMS is available using the Entity format, as described in Section 3.1.2. For example, a mountainous range is represented by an Entity, classified as dangerous and thus avoided by SAMMS. A target is also represented by an Entity, and the relative position data can be used by SAMMS to perform Target-Analyze and Target-Attack Objectives.

In short, the Entity theoretical construct constitutes a way to unify the definition of specific aspects of the environment where the UAV is operating. Many types of objects can be classified as Entities, including other aircraft, ground vehicles, buildings, terrain features, weather areas, Air Traffic Control flight zones or generic threat areas; the Entity construct is designed to provide to SAMMS all information it needs in order to take into account the Entity during the flight generation process. For example, a mountain range is specified by its location and area of effect (currently only rectangular and circular areas can be defined), and will be indicated as a neutral Entity (it is not actively trying to damage the UAV) but very dangerous (a collision would be catastrophic). SAMMS will then plan accordingly by avoiding the area.

In the simulations presented in this thesis, the definition of Entities was part of the scenario definition. In an actual system based on SAMMS, Entities will have to be automatically generated from several sources. Interestingly, the use of the Entity format removes the need for data fusion at the highest level: Entities provided by different sources can be seamlessly mixed. It is to be noted however that data fusion might happen at lower level, as a single source might not be able to provide all data regarding an Entity and this might need to be integrated with data from another source.

There are several sources which could provide data to be formatted as Entities:

- geographic databases
- on-board sensors, such as radar and electro-optic sensors
- off-board sensors, relaying similar knowledge to that provided by on-board sensors but that is not within range at the moment
- radio communications, giving ATC instructions

For all of these, some means of automatic formatting must be envisioned.

As previously stated, SAMMS is informed of the presence of terrain features and buildings by assigning equivalent Entities. Because of their static nature, they need not to be detected in real-time, but might be provided from a previously developed database. The behaviour of the UAV with respect to them depends greatly on the UAV type. Low-flying UAVs will need to consider these Entities a threat throughout the mission, while high-altitude UAVs could only consider them a danger during the take-off and approach phases. The Entities might also constitute a target for an Analyze or Attack Objective, in which case deconfliction with their threat level is needed (e.g. if the Entity represents a worse threat than the execution priority for the related Objective, the Objective should be cancelled, and the mission-path-adjust algorithm should be inhibited otherwise).

On-board sensors of a UAV include dedicated navigational sensors (such as a weather radar) but might also include the actual UAV payload (as could be an electro-optic sensor). Depending on the system type, a variety of objects will be detectable: weather areas, other aircraft, ground vehicles, buildings and terrain features. For example, the UAV might be exploring an unmapped area and carry a payload that can determine in real-time the presence of specific terrain features and buildings; this information can be formatted as Entities and then be actively used by SAMMS in its flight planning. The conversion of the output of sensor systems to the Entity format is expected to be one of the greatest challenges towards an actual implementation of the SAMMS architecture. The biggest challenge is represented by the fact that this conversion is highly dependent on the specific sensor type: for each sensor type (and for each sensor model within each type), dedicated algorithms will have to be implemented. A more interesting challenge is the possibility to fuse data from multiple sensors in order to fully determine the properties of an Entity.

It is important to note that while the challenge of automatic conversion of on-board sensor data into the Entity format is not trivial, the same strategies can be applied to the data provided by off-board sensors, as the sensor types can be expected to be the same (with different scales). An important observation regarding off-board sensors is that sensor data processing might be carried out off-board (thus relaying an Entity list to the SAMMS-based UAV) or on-board (relaying direct off-board sensor output to the SAMMS-based UAV, to be converted into the Entity format by the on-board computer). Naturally the first option (off-board processing) is preferable, especially in the case of small UAVs which can be expected to possess limited computational resources.

Sensor/payload management

Sensors and payloads, whether they directly provide information to the UAV or not, might need to receive specific commands in order to perform their function. For example, radar might autonomously pick up the location of a target; however, to perform a Recon mission a picture would need to be taken, and the electro-optic sensor performing this function would need to be instructed regarding the target position.

Sensor and payload management is an essential activity for a UAV; if the payload is unable to perform its function, a mission will inevitably fail. However, it is too platform dependent to be developed at this stage of the project. As a generic demonstration of capability, the Execution Agent was fitted with very simple commands that are addressed to a payload; depending on the Action currently being performed, specific payload types are activated.

An actual payload management system will need to be integrated with the Execution Agent and possibly with energy management functionality. In general, the Execution Agent already possesses all information that could be required in order to give correct instructions to a payload management system, such as target position, current UAV situation and planned manoeuvres.

8.2.3 Hardware implementation

While the SAMMS architecture is not constrained to a specific UAV class (UAV performance is provided to SAMMS by a set of easily changeable parameters), it has been designed with application on small low-cost UAVs in mind. A key goal of the project has always been ensuring that the system could run on a PC/104 board running a real-time operating system such as QNX or VxWorks Tornado.

This phase of development has not been reached, so the following observations cannot currently be proved and are purely based on analysis of the simulations and from experience derived from other projects. The SAMMS architecture is currently executed within a Matlab R2006b environment with a 100 ms sample time; a dual-core laptop computer can run an entire simulation (including the computationally heavy UAV model and a visualization feature based on Microsoft Flight Simulator) faster than real-time.

The target platform for SAMMS is a single PC/104 board running either QNX or Tornado; using the Real-Time Workshop feature of Matlab, the intention is to automatically generate executable code from the Simulink model. This code would include the three Soar agents and the autopilot function, so as to avoid the need for an external autopilot. Depending on the avionics suite and the actual hardware, an additional PC/104 I/O board could be needed, but no additional components should be required. Using Common-Off-The-Shelf (COTS) hardware, such a configuration would weigh in the 200-500 grams range and cost about £ 400-800. This qualifies it for use within small UAVs with a maximum weight of 5-10 kg and costs in the £ 5000 range.

At present, the fact that such a system possesses sufficient computational resources to execute the SAMMS software cannot be guaranteed. However, past experiences such as in (Battipede et al, 2006), suggest that the generation of executable code from Simulink models allows for a dramatic increase of execution speed. The addition of Soar agents brings uncertainty, however these are in fact executed as Simulink S-functions (which do not pose an issue), and related literature (Long et al, 2007; Laird and Rosenbloom, 1990; Jones et al, 1999) does not evidence problems regarding the computational resources required by the Soar architecture. While this cannot be proven at present, the author is confident that a single PC/104 board will be sufficient to execute the SAMMS software respecting real-time requirements.

With this in mind, it is possible to plot a possible roadmap for an actual implementation of the SAMMS architecture, up to flight testing. The first activity is mainly related to SAMMS itself: apart from the possible improvements suggested in Section 8.2.1, the challenge is to prototype a "SAMMS box", or more precisely a PC/104 board running the executable code generated from the SAMMS model. At this stage, through appropriate interfacing it should be possible to still use the UAV model to test the system.

The second activity is related to the choice or development of an actual UAV platform. Many academic and industrial groups have recently developed low-cost UAVs, so it should not be difficult to find a suitable platform; in light of this, development from scratch of a new UAV is not a desirable option. Together with the definition of the UAV platform, the definition of the avionics suite must be ensured; a third-party UAV could already possess a full avionics suite or not, and careful thought must be placed on determining the sensory needs for the platform and how these should be addressed.

Once a hardware implementation of SAMMS and a viable UAV platform are ready, the third phase of development regards integration of the two systems. The avionics suite on-board the UAV must be modified to suit the SAMMS requirements, and the parameters determining the performance of SAMMS must be redefined. In particular, two sets of values need to be recalculated: the values of the Airframe Data and Health block (which define the UAV performance for SAMMS), and the values of autopilot gains (which need to be tailored to the specific configuration).

During this phase, it may also be necessary to develop intermediate layers for the Fault Detection and Sensor/Payload Management functions. Depending on the actual capabilities of the UAV platform, at least some of this functionality will likely be present and appropriate interfacing must be developed (as described in Section 8.2.2).

The fourth and final phase involves actual testing of the entire UAV. After initial ground tests (system activation, hardware-in-the-loop simulations), actual flight tests might be performed; it is to be noted that a difficult part of tests might be obtaining the required authorizations. This is especially true for out-of-sight tests, which would certainly be needed to prove the SAMMS capabilities but are generally treated severely by rules.

8.3 Concluding remarks

Throughout this thesis, the Soar-based Autonomous Mission Management System was demonstrated to be a viable software architecture for the development of autonomous UAVs.

Notable achievements of the project are:

- the development of a set of theoretical constructs that allow computational expression of a flight plan
- the demonstration of feasibility of an integrated Soar/Simulink architecture
- the implementation of various flight plan improvement strategies
- the development of a fully contained simulation environment, used to test the architecture in a realistic manner

The project has currently reached an advanced simulation stage, but significant work would have to be done to reach actual implementation. SAMMS is built to be adaptable to any fixed-wing UAV type (provided that sufficient computational resources are on-board), but a consistent integration phase is needed, especially since the development of specific functionality is directly connected to the choice of an actual target platform.

A SAMMS-based UAV could be used in many operational environments: the architecture is built for both military and civilian use, and to accommodate various UAV types. Due to focus on cost limitation, it can be seen as best suited for small civilian UAVs. In particular, the possibility of use by untrained personnel is appealing for civilian applications: a SAMMS-based UAV can be used by an operator which only possesses broad knowledge regarding its capabilities (and this is in fact to ensure that missions are executed, rather than for safety reasons).

In this sense, an interesting possible application field for SAMMS-based UAV is environmental monitoring. Many fields of research could benefit from the availability of a low-cost flying platform, that could replace satellite observation (which is costly, has limited availability and can lack sensor resolution due to the distances involved) and manned flight observation (which is essentially the same as UAV observation, but involves higher costs because of increased sizes). This type of application will constitute the ideal target for a future implementation of SAMMS.

References

Albus J., "*The NIST Real-time Control System (RCS): an approach to intelligent systems research*", Journal of Experimental and Theoretical Artificial Intelligence 9 (1997), pp. 157-174

Alexander J., "*Loxodromes: A Rhumb Way to Go*", pages 349-356, the Mathematics Magazine, Vol. 77, No. 5, December 2004

Ambrosia V., Wegener S., Brass J., Schoenung S., "*The UAV Western States Fire Mission: Concepts, Plans and Developmental Advancements*", AIAA 3rd "Unmanned Unlimited" Technical Conference, Workshop and Exhibit, 20 - 23 September 2004, Chicago, Illinois, AIAA 2004-6415

Anderson J., Bothell D., Byrne M., Douglass S., Lebiere C., Qin Y., "*An Integrated Theory of the Mind*" Psychological Review 2004, 11(4), pp. 1026-1060

Applegate D., Bixby R., Chvatal V., Cook W., "*The Traveling Salesman Problem: A Computational Study*", published by the Princeton University Press, 2007

Arkin R., Balch T., "*AuRA: Principles and Practice in Review*" Journal of Experimental & Theoretical Artificial Intelligence, Vol. 9, No. 2-3, April 1997, pp. 175 – 189

Battipede M., Gili P., Lando M., Gunetti P., "*Flight Control System Rapid Prototyping for the Remotely-Controlled Elettra-Twin-Flyer Airship*", Proceedings of the AIAA Modelling Simulation and Technologies Conference, Keystone, CO, 2006, AIAA-2006-6624

Bekey G., "*Autonomous Robots: From Biological Inspiration to Implementation and Control*", The MIT Press, 2005

Bellman R., "*Combinatorial Processes and Dynamic Programming*", in "Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics 10", American Mathematical Society, 1960

Boskovic J., Mehra R., "*An Integrated Fault Management System for Unmanned Aerial Vehicles*", 2nd AIAA Unmanned Unlimited Conference, 15-18 September 2003, San Diego, CA, AIAA 2003-6642

Boskovic J., Prasanth R., Mehra R., "*A Multi-Layer Control Architecture for Unmanned Aerial Vehicles*", Proceedings of the American Control Conference, 8-10 May 2002, Anchorage, AK

Bowditch N., "*The American Practical Navigator*", published by the Defense Mapping Agency Hydrographic Topographic Center (DMAHTC), 2002; first published in 1802

Boyd J., "*Destruction and Creation*", 1976
http://www.goalsys.com/books/documents/DESTRUCTION_AND_CREATION.pdf
[last accessed on 20/12/2010]

Boyd J., "*The Essence of Winning and Losing*", 1996
www.projectwhitehorse.com/pdfs/boyd/The%20Essence%20of%20Winning%20and%20Losing.pdf
[last accessed on 20/12/2010]

Brooks R., "*A Robot that Walks; Emergent Behavior from a Carefully Evolved Network*", IEEE International Conference on Robotics and Automation, Scottsdale, AZ, May 1989

Brooks R., "*Intelligence without reason*", Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), pages 569–595, Sydney, Australia, 1991

Bugajska M., Schultz A., Trafton J., Taylor M., Mintz F., "*A hybrid cognitive-reactive multiagent controller*", Proceedings of 2002 IEEE/RSJ, International conference on Intelligent Robots and Systems (IROS 2002), Switzerland

References

Campa G., "*Airlib*", http://www.mathworks.com/matlabcentral/fileexchange/3019-airlib, 2003 [last accessed on 12/01/2011]

Carbonell J., Knoblock C., Minton S., "*PRODIGY: An integrated architecture for planning and learning*", in "Architectures for intelligence", Hillsdale, NJ, Lawrence Erlbaum, 1990

Chandler P., Pachter M., "*Research Issues in Autonomous Control of Tactical UAVs*", Proceedings of the American Control Conference, Philapelphia, PA, 1998

Chandler P., Pachter M., Swaroop D., Fowler J., Howlett J., Rasmussen S., Schumacher C., Nygard K., "*Complexity in UAV Cooperative Control*", Proceedings of the American Control Conference, 8-10 May 2002, Anchorage, AK

Chantery E., Barbier M., Farges J., "*Mission Planning for Autonomous Aerial Vehicles*", ONERA, Toulouse, France, Systems Control and Flight Dynamics Department http://www.cert.fr/dcsd/cd/MEMBRES/magali/IFAC_IAV04.pdf, [last accessed on 20/12/2010]

Chien S., Sherwood R., Tran D., Cichy B., Rabideau G, Castano R., Davis A., Mandl D., Frye S, Trout S, Shulman S., Boyer D., "*Using Autonomy Flight Software to Improve Science Return on Earth Observing One*", Journal of Aerospace Computing, Information, and Computing, Vol. 2, April 2005

Clough B., "*Metrics, Schmetrics! How the Heck Do You Determine A UAVs Autonomy Anyway?*", PerMIS Conference Proceedings, 2002

Clough B., "*Unmanned Aerial Vehicles: Autonomous Control Challenges, a Researcher's Perspective*", Journal of Aerospace Computing, Information and Communication, Vol. 2, August 2005

Councill I., Haynes S., Ritter F., "*Explaining Soar: Analysis of Existing Tools and User Information Requirements*", Proceedings of the 5th International Conference on Cognitive Modelling, 2003

Cummings M., Bruni S., Mercier S., Mitchell P., "*Automation Architecture for Single Operator, Multiple UAV Command and Control*", The International Command and Control (C2) Journal, Volume 1, Number 2, 2007, Special Issue on Decision Support for Network-Centric Command and Control

DeGarmo M., Nelson G., "*Prospective Unmanned Aerial Vehicle Operations in the Future National Airspace System*", 4th Aviation Technology, Integration and Operations Conference, ATIO 2004

Doherty P., Granlund G., Kuchcinski K., Sandewall E., Nordberg K., Skarman E., and Wiklund J., "*The WITAS Unmanned Aerial Vehicle Project*", Proceedings of the 14th European Conference on Artificial Intelligence 2000 (ECAI-00), August 2000, Berlin

Doherty P., Kvarnström, J., Heintz, F. "*A temporal Logic-Based Planning and Execution Monitoring Framework for Unmanned Aircraft Systems*", Journal of Autonomous Agents and Multi-Agent Systems, Volume 19, Number 3, 2009

Drozeski G., Saha B., Vachtsevanos G., "*A Fault Detection and Reconfigurable Control Architecture for Unmanned Aerial Vehicles*", 2005 IEEE Aerospace Conference, Big Sky, MT

Federal Aviation Administration (FAA), "*Flight Crew Procedures during Taxi Operations*", Advisory Circular Number 120-74A, US Department of Transportation, 26/03/2003

Ferguson I., "*TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*", PhD thesis, Clare Hall, University of Cambridge, UK, November 1992 (Also available as Technical Report No. 273, University of Cambridge Computer Laboratory)

Ferry M., Tu Z., Stephens L., Prickett G., "*Towards true UAV Autonomy*", 2007 Information Decision and Control Conference, ©2007 IEEE

Fisher M., Wooldridge M., "*On the Formal Specification and Verification of Multi-Agent Systems*", International Journal of Cooperative Information Systems, 1997

Gancet J., Hattenberger G., Alami R., Lacroix S., "*Task planning and control for a multi-UAV system: architecture and algorithms*", International Conference on Intelligent Robots and Systems, IEEE 2005

Geiger B., Horn J., Delullo A., Long L., "*Optimal Path Planning of UAVs Using Direct Collocation with Nonlinear Programming*", AIAA Guidance Navigation and Control (GNC) Conference 2006, August 2006, AIAA 2006-6199

Gellert W., Gottwald S., Hellwich M., Kastner H., Kustner H., "*The VNR Concise Encyclopaedia of Mathematics*", 2nd edition, published by Van Nostrand Rheinhold, New York, 1989

Georgeff M., Lansky A., "*Reactive reasoning and planning*", Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), pages 677–682, Seattle, WA, 1987.

Goebel G., "*History of Unmanned Aerial Vehicles*", http://www.vectorsite.net/twuav.html, 2008 [last accessed on 20/12/2010]

Gottfredson L., "*Mainstream Science on Intelligence: An Editorial with 52 Signatories, History, and Bibliography*", Intelligence, Vol. 24, No. 1, 1997, pp. 13-23

Hanford S., Janrathitikarn O., Long L., "*Control of a Six-Legged Mobile Robot Using the Soar Cognitive Architecture*", Proceedings of the 46th AIAA Aerospace Sciences Meeting and Exhibit, 7 – 10 January 2008, Reno, NV, AIAA 2008-0878

Hanson D., Olney A., Pereira I., Zielke M., "*Upending the Uncanny Valley*", AAAI Conference Proceedings, 2005

Harris P., Swain B., Webb K., "*The Control and Monitoring System for the Adour 900*", Aircraft Engineering and Aerospace Technology, pages 545-556, Volume 72, Number 6, 2000

Haselager W., "*Robots, philosophy and the problems of autonomy*", pages 515-532 in Pragmatics and Cognition, Vol. 13, No. 3, 2005, published by John Benjamins Publishing Company, 2005

Higashino S., Kim J., Kuroyanagi S., Sakurai A., "*Hierarchical Flight Management and Control of Autonomous UAVs Based on Evolutionary Computation and Total Energy Concept*", 3rd AIAA Unmanned Unlimited Conference, 20-23 September 2004, Chicago, IL, AIAA 2004-6531

Howden N., Ronnquist R., Hodgson A., Lucas A., "*JACK Intelligent Agents: Summary of an Agent Infrastructure*", Proceedings of the 5th ACM International Conference on Autonomous Agents, 2001

Jacklin S., Schumann J., Gupta P., Richard M., Guenther K., Soares F., "*Development of Advanced Verification and Validation Procedures and Tools for the Certification of Learning Systems in Aerospace Applications*", American Institute of Aeronautics and Astronautics, InfoTech@Aerospace Conference 2005, Paper No. 2005-5476

Jang J., Tomlin C., "*Design and Implementation of a Low Cost, Hierarchical and Modular Avionics Architecture for the DragonFly UAVs*", Proceedings the 2002 AIAA Guidance Navigation and Control Conference, AIAA 2002-4465

Janrathitikarn O., Long L., "*Gait Control of a Six-Legged Robot on Unlevel Terrain Using a Cognitive Architecture*", 2008 IEEE Aerospace Conference, Big Sky, MT, March 2008

Jennings N., Wooldridge M., "*Applications of Intelligent Agents*", chapter 1 in "Agent Technology: Foundation, Applications and Markets", Springer, 1998

Johnson T., Sutherland A., Bush S., "*The TRAC Mission Manager Autonomous Control Executive*", 2001 IEEE Aerospace Conference, Big Sky, MT

References

Jones R., Laird J., Nielsen R., Coulter K., Kenny R., Koss F., "*Automated Intelligent Pilots for Combat Flight Simulation*", AI Magazine, Spring 1999, pp. 27-41

Jonker C., Treur J., "*Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness*", Proceedings of COMPOS '97, LNCS issue 1536, pp. 350-380, Springer-Verlag Berlin, 1998

Kalus A., Hirst A., "*Soar Agents for OOTW Mission Simulation*", 4th International Command and Control Research and Technology Symposium, Nasby Park, Sweden, 1998

Karim S., Heinze C., Dunn S., "*Agent-based Mission Management for a UAV*", Proceedings of the IEEE International Conference on Intelligent Sensors, Sensor Networks & Information Processing (ISSNIP), Melbourne, Australia, December 2004

Karim S., Heinze C., "*Experiences with the Design and Implementation of an Agent-based Autonomous UAV Controller*", AAMAS'05, July 25-29, 2005, Utrecht, Netherlands

Kim H., Shim D., "*A flight control system for aerial robots: algorithms and experiments*", Control Engineering Practice 11 (2003), published by Elsevier Science Ltd, pages 1389-1400

Kim H., Cho Y., Oh S., "*CAMUS: A middleware supporting context-aware services for network-based robots*", 2005 IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO2005)

Krause, S., "*Aircraft Safety: Accident Investigations, Analyses and Applications*", published by The McGraw-Hill Companies, 2003

Laird J., Newell A., Rosenbloom P., "*Soar: An Architecture for General Intelligence*", Artificial Intelligence, 1987, 33(3), pp. 1-64

Laird J., Rosenbloom P., "*Integrating Execution, Planning, and Learning in Soar for External Environments*", Proceedings of the 1990 AAAI Conference, 1990

Langley P., Cummings K., Shapiro D., "*Hierarchical skills and cognitive architectures*", Proceedings of the 26th annual conference of the Cognitive Science Society, pages 779–784, Chicago, IL, 2004

Langley P., Laird J., Rogers S., "*Cognitive architectures: Research issues and challenges*", Cognitive Systems Research 10, pages 141–160, 2009

Lehman J., Laird J., Rosenbloom P., "*A gentle Introduction to Soar, an Architecture for Human Cognition; 2006 update*", 2006

Li S., Boskovic J., Seereeram S., Prasanth R., Amin J., Mehra R., Beard R., Mclain T., "*Autonomous Hierachical Control of Multiple Unmanned Combat Air Vehicles (UCAVs)*", Proceedings of the American Control Conference, 8-10 May 2002, Anchorage, AK

Long L., Hanford S., "*Evaluating Cognitive Architectures for Intelligent and Autonomous Unmanned Vehicles*", Proceedings of the 2007 AAAI Workshop, 2007

Long L., Hanford S., Janrathitikarn O., Sinsley S., Miller J., "*A Review of Intelligent Systems Software for Autonomous Vehicles*", Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Security and Defense Applications (CISDA 2007)

Long L., "*The Critical Need for Software Engineering Education*", CrossTalk - The Journal of Defense Software Engineering, January 2008 issue

Lucas A., Van der Welden S., Heinze C., Karim S., "*Development and Flight Testing of an Intelligent, Autonomous UAV Capability*", 3rd AIAA Unmanned Unlimited Conference, 20-23 September 2004, Chicago, IL

Ludington B., Johnson E., Vachtsevanos G., "*Augmenting UAV Autonomy: Vision-Based Navigation and Target Tracking for Unmanned Aerial Vehicles*", IEEE Robotics and Automation Magazine, pages 63-71, September 2006

Matijevic M., "*Autonomous Navigation and the Sojourner Microrover*", Science, 17 April 1998: Vol. 280. no. 5362, pp. 454 – 455

McDermott J., Forgy C., "*Production system conflict resolution strategies*", In "Pattern Directed Inference Systems", Academic Press New York, 1978

Mehra R., Boskovic J., Li S., "*Autonomous Formation Flying of Multiple UCAVs under Communication Failure*", Position Location and Navigation Symposium, IEEE 2000

Miller J. et al., "*Intelligent Unmanned Air Vehicle Flight Systems*", American Institute of Aeronautics and Astronautics, InfoTech@Aerospace Conference 2005, Paper No. 2005-7081

Muller J., Pischel M., Thiel M., "*Modelling reactive behaviour in vertically layered agent architectures*", in Wooldridge M. and Jennings N., editors, "Intelligent Agents: Theories, Architectures, and Languages" (LNAI Volume 890), pages 261–276, Springer-Verlag Berlin, Germany, January 1995

Nehme C., Cummings M., Crandall J., "*A UAV Mission Hierarchy*", HAL2006-09, Research Report for the MIT Humans and Automation Laboratory, Cambridge, MA, 2006

Nehme C., Crandall J., Cummings M., "*An Operator Function Taxonomy for Unmanned Aerial Vehicle Missions*", 12th International Command and Control Research and Technology Symposium, Newport, RI, June, 2007

Nelson R., "*Flight Stability and Automatic Control*", published by the McGraw-Hill Book Company, 1989

Newell A., "*A Reasoning, problem solving and decision processes. The problem space as a fundamental category*", in "Attention and Performance VIII", Erlbaum, HIllsdale, NJ, 1980

Newell A., "*Unified Theories of Cognition*", Harvard University Press, Cambridge, 1990

Rathbun B., Kragelund S., Pongpunwattana A., Capozzi B., "*An Evolution-Based Path Planning Algorithm for Autonomous Motion of a UAV through Uncertain Environments*", Proceedings of the 21st Digital Avionics Systems Conference, IEEE 2002

Radio Technical Commission for Aeronatics (RTCA), "*DO-178B, Software Considerations in Airborne Systems and Equipment Certification*", published by RTCA Inc., 1992

Rauw M., "*The Flight Dynamics and Control Toolbox*", http://dutchroll.sourceforge.net/, 2003 [last accessed on 12/01/2011]

Rauw M., "*FDC 1.4 – A Simulink Toolbox for Flight Dynamics and Control Analysis*", draft version 7, 25 May 2005

Riseborough P., "*Automatic Take-Off and Landing Control for Small UAVs*", Proceedings of the 5th Asian Control Conference, 2004

Schaefer P. et al., "*Reliable Autonomous Control Technologies (ReACT) for Uninhabited Aerial Vehicles*", 2001 IEEE Aerospace Conference, Big Sky, MT

Sharkey N., "*Grounds for Discrimination: Autonomous Robot Weapons*", Royal United Services Institute (RUSI) Defence Systems, Vol. 11, No. 2, October 2008

References

Sholes E., "*Evolution of a UAV Autonomy Classification Taxonomy*", IEEE Aerospace Conference, 2007

Sinsley G., Long L., Niessner A., Horn J., "*Intelligent Systems Software for Unmanned Air Vehicles*", Proceedings of the 46th AIAA Aerospace Science Meeting, January 2008, Reno, NV, AIAA 2008-0871

Smith P., Willcox S., "*Systems Research for Practical Autonomy in Unmanned Air Vehicles*", AIAA Infotech@Aerospace Conference 2005, Arlington, VA, September 2005

Stevens B., Lewis F., "*Aircraft Control and Simulation*", a Wiley-Interscience publication, published by John Wiley and Sons, Inc., 1992

Stover J., Kumar R., "*A Behavior-based Architecture for the Design of Intelligent Controllers for Autonomous Systems*", IEEE International Symposium on Intelligent Control/Intelligent Systems and Semiotics, Cambridge, MA, Sept. 15-17, 1999

Sullivan D., Totah J., Wegener S., Enomoto F., Frost C., Kaneshige J., Frank J., "*Intelligent Mission Management for Uninhabited Aerial Vehicles*", Remote Sensing Applications of the Global Positioning System, Proceedings of SPIE, Volume 5661-121, November 2004, Honolulu, HI

Sun R., Merrill E., Peterson T., "*From implicit skills to explicit knowledge: A bottom–up model of skill learning*", Cognitive Science, 25, 203–244

Sun R., "*Theoretical status of computational cognitive modelling*", Cognitive Systems Research 10, pages 124–140, 2009

Tan Y., Zhu X., Zhao R., Zhang B., "*The Design and Implementation of Autonomous Mission Manager for Small UAVs*", International Conference on Control and Automation, 30 May – 1 Jun 2007, Guangzhou, China

Taylor G., Knudsen K., Holt L., "*Explaining Agent Behavior*", Proceedings of Behavior Representation in Modeling and Simulation (BRIMS) 2006

The Mathworks, "*Simulink. Overview of S-functions*", software documentation, 2006

Theunissen E., Goossens A., Bleeker O., Koeners G., "*UAV Mission Management Functions to Support Integration in a Strategic and Tactical ATC and C2 Environment*", Proceedings of the 2005 AIAA Modeling and Simulation Technologies Conference, 15-18 August 2005, San Francisco, CA

UAV Task Force, "*The Joint JAA/EUROCONTROL Initiative on UAVs*", UAV Task Force Final Report, 2004

United States Air Force (USAF), "*USAF Unmanned Aircraft Systems Flight Plan 2009-2047*", USAF Headquarters, Washington DC, 18 May 2009

Valenti M., Schouweenars T., Kuwata Y., Feron E., How J., "*Implementation of a Manned Vehicle – UAV System*", Proceedings of the 2004 AIAA Guidance Navigation and Control Conference, 16-19 August 2004, Providence, Rhode Island

Veres S., Molnar L., Lincoln N., Morice C, "*Autonomous Vehicle Control Systems: a Review of Decision Making*", IMechE Journal of Systems and Control, 2010, under revision

Veres S., "*Mission Capable Autonomous Control Systems in the Oceans, in the Air and in Space*", Brain-Inspired Information Technology, Studies in Computational Intelligence 2010, Volume 266/2010

Wegener S., Schoenung S., Totah J., Sullivan D., Frank J., Enomoto F., Frost C., Theodore C., "*UAV Autonomous Operations for Airborne Science Missions*", 3rd AIAA Unmanned Unlimited Conference, 20-23 September 2004, Chicago, IL

Wells A., Rodrigues C., "*The Regulatory Framework*", pages 1-23, chapter 1 from "*Commercial Aviation Safety*", published by The McGraw-Hill Companies, 2004

Wooldridge M., Jennings N., "*Intelligent Agents: Theory and Practice*", in Knowledge Engineering Review, Volume 10 number 2, June 1995, Cambridge University Press, 1995

Wooldridge M., "*Intelligent Agents*", chapter 1 in "Multi-Agent Systems: a modern approach to distributed artificial intelligence", the MIT Press, 1999

Wooldridge M., "*Introduction to Multi-agent Systems*", John Wiley & Sons, Inc., 2001

Wooldridge M., Ciancarini P., "*Agent-Oriented Software Engineering: the State of the Art*", chapter 1 in "Agent-Oriented Software Engineering AOSE 2000", Lecture Notes in Computer Science 1957, published by Springer-Verlag, 2001

Wong K., "*Active Message Logical Routing Verification An Approach to Automate Verification of Intelligent Agent Behaviors*", ACM SIGICE Bulletin, Volume 23, Number 2, October 1997

Wzorek, M., Doherty P., "*Reconfigurable Path Planning for an Autonomous Unmanned Aerial Vehicle*", Proceedings of the 16th International Conference on Automated Planning and Scheduling, 2006