A Component-Based Three-Layer Autonomy Architecture for Unmanned Aerial Vehicles

by

Sean D. Bassett, BASc, University of Ottawa

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for
the degree of Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
September 2008

Your file Votre référence
ISBN: 978-0-494-47505-8
Our file Notre référence
ISBN: 978-0-494-47505-8

The undersigned recommend to the
Faculty of Graduate Studies and Research
acceptance of the thesis

A Component-Based Three-Layer Autonomy Architecture for Unmanned Aerial Vehicles

Submitted by Sean D. Bassett, BASc, University of Ottawa
in partial fulfillment of the requirements for
the degree of Master of Applied Science in Electrical Engineering

---

Thesis Supervisor
Dr. Trevor Pearce

Chair, Department of Systems and Computer Engineering
Dr. Victor Aitken

Abstract

This thesis describes the Component-Based Three-Layer Autonomy (CB3A) architecture for controlling a UAV. CB3A is a new interpretation of a reactive hybrid three-layer architecture in which each layer contains one or more components that communicate using publish/subscribe messaging. The State-Based Event-Driven (SBED) sequencer is a fundamental component of CB3A, and is used to control autonomous behaviour specified using a UML 2 behavioural state machine. In addition, the Orbit Manoeuvre, a fallback mechanism for fixed-wing UAV obstacle avoidance, is presented. As part of the research for this thesis, the UAVSim Testbed was conceived and created to serve as a simulation environment for UAV research and development.

Acknowledgements

I would like to thank my supervisor, Dr. Trevor Pearce, for his guidance and direction in this process. Often, I found that by sitting and discussing my thoughts with him, I would come away with a clearer idea of what it was that I was trying to figure out.

Table of Contents

List of Figures

List of Abbreviations

COTS	- Commercial Off-the-Shelf
CB3A	- Component-Based Three-Layer Autonomy
FAA	- Federal Aviation Administration
FSM	- Finite State Machine
HLA	- High Level Architecture
MV	- Machine Vision
OAZ	- Obstacle Avoidance Zone
OFZ	- Obstacle-Free Zone
RTI	- Run-Time Infrastructure
SBED	- State-Based Event-Driven
SDE	- Simulation Driven Engineering
TC	- Transport Canada
TDL	- Task Description Language
UAV	- Unmanned Aerial Vehicle
UML	- Unified Modelling Language

Chapter 1

Introduction

There are many ongoing studies into developing autonomous Unmanned Aerial Vehicles (UAVs). The applications are desirable to many agencies. Certainly the military has a great interest in developing UAVs for gathering intelligence [1]. Other potential applications include border patrol, forest fire monitoring [2], environmental surveys, law enforcement, disaster relief, and agricultural use (crop spraying) [3].

The goal in developing an autonomous UAV system is to replace the role of a human controller. Modern control theory provides the necessary automation to fly the system, as well as perform automated takeoffs and landings. This is evidenced by the inclusion of these functions on commercially available autopilots [4, 5]. System autonomy is concerned with mission management which includes operating mission essential equipment, navigating around obstacles, changing course to deal with adverse weather, taking precautionary measures due to degrading system health, and dealing with emergencies due to system malfunctions.

Initial approaches to autonomy, based on a Sense-Plan-Act (SPA) paradigm were highly deliberative in nature [6]. They inspired research in artificial intelligence planning algorithms which were fundamental to the approach. Unfortunately, the inherently slow nature of these planning algorithms had a detrimental effect on robot performance.

In response to the performance limitations of SPA, a different, reactive, approach was proposed [7]. The Subsumption architecture replaced the planning algorithms of SPA with a library of premade plans. Stimuli from the environment were used to trigger execution of the appropriate plan. This approach initially showed promise. By not relying on performance-robbing classical planning algorithms, robots using Subsumption could perform simple tasks much quicker than before. However, the purely reactive approach did not allow for specifying complex intelligent behaviour.

Deliberative and reactive approaches both had their benefits and shortcomings, so it is not surprising that the two approaches were combined into a hybrid architecture. The resulting so-called three-layer architecture puts deliberative procedures on one layer, and reactive ones on the second one. The third layer is reserved for low level tasks such as control loops [8]. The execution of tasks associated with a system's autonomous behaviour is managed by the reactive procedures. This has led to research into different approaches specific to managing and implementing the sequencing of tasks.

The architecture for an autonomous UAV must be chosen carefully. Ultimately, UAVs will need to be certified by governing bodies, such as Transport Canada, and the Federal Aviation Administration (FAA) before they are allowed to share the skies with other aircraft. Reports by these agencies cite that a sense-and-avoid capability will be a key factor required for an autonomous UAV to operate safely [9, 10].

In pursuing a viable sense-and-avoid capability, research into obstacle avoidance for UAVs is an important field. Although the field of robotics offers many lessons, obstacle avoidance for UAVs has the potential to be a much greater challenge, especially when considering fixed-wing UAVs. Ground-based and rotorcraft vehicles have the

advantage that they can stop or reverse course, but fixed-wing UAVs must maintain a minimum forward speed to stay in the sky.

Developing an autonomous UAV requires the collaboration of many contributors. It involves the development of autonomy, airframe, avionics, propulsion, sensors, and mission specific subsystems. Autonomous UAV development has not yet matured, so some subsystems, in particular those related to obstacle detection, will require significant amounts of research and development. Simulation will be a valuable tool for developing autonomy software. It will provide a simulated world in which the autonomy software can operate. This will allow for testing its behaviour without the cost, time, or risk of performing tests with the actual airframe, until many of the bugs have been worked out. Furthermore, simulation will allow development to proceed on subsystems that rely on other as-of-yet undeveloped subsystems, by simulating the behaviour of the missing parts.

The primary goal of this research is to develop a system architecture that will support the interaction of multiple subsystems and that has a means of specifying autonomous behaviour in a clear and concise manner. The autonomous behaviour should include a means of static obstacle avoidance for a fixed-wing UAV.

The value of this research is enhanced by the fact that it took place in concert with a project to develop an actual autonomous UAV. The goal of that project, the GeoSurv II, is to develop an autonomous fixed-wing UAV that will be used to perform aerial geophysical surveys. Using a multidisciplinary effort, teams are working on designing the airframe, propulsion, avionics, and obstacle detection systems.

Chapter 2 provides background knowledge pertaining to this research including the three-layered architecture for autonomy, component-based systems, publish/subscribe messaging, state-based modeling, and the GeoSurv II project.

Chapter 3 provides a more in-depth review of current research in autonomy architecture, simulation environments for autonomy research, and current approaches to obstacle avoidance for autonomous vehicles.

Chapter 4 presents the claims of the thesis, discusses current limitations, and describes the scope of research.

Chapter 5 describes the CB3A architecture and the UAVSim Testbed simulation environment.

Chapter 6 describes the SBED sequencer and its integration in the CB3A architecture.

Chapter 7 describes how obstacle avoidance is enabled using the CB3A architecture and the SBED sequencer for a fixed-wing UAV

Appendix A contains example code for component wrapper classes that are referred to in Chapter 5.

Chapter 2

Background

This chapter presents background information related to the research. First, the three-layer autonomy architecture is introduced, as it serves as the basis for the proposed autonomy architecture. Component-based systems and publish/subscribe messaging are also covered as they are fundamental to the proposed architecture. An overview of finite state machines and UML 2 behavioural state machines is provided as it forms the basis for the implementation of the autonomy sequencer layer. Finally, the GeoSurv II UAV project is discussed.

2.1 Three-Layer Autonomy Architecture

The earliest approaches to developing autonomous robot systems were simple extensions of control theory. In control theory, sensors gather information on the system's state, and feed that information into a control function, which then calculates the required outputs to achieve or maintain a desired state. The simple sense-plan-act (SPA) [6] approach to autonomy, shown in Figure 2.1, replaces the control function with an artificial intelligence planning algorithm. Sensor information is gathered and passed to the planner, which generates plans that drive system actuators.

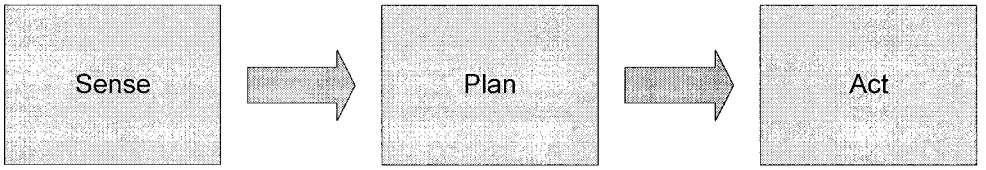

Figure 2.1: Sense-Plan-Act Approach

Given the current state and the desired 'goal' state, the planner searches through all possible sequences of actions to determine which actions will best help the autonomous system arrive at its desired goal state. It can be likened to a game of chess where the autonomous system's goal is to get to a state where its opponent is in checkmate. Based on the current board position, the autonomous system considers all possible moves, and countermoves by the opponent to determine what its next move will be. Due to the computational complexity of searching through such a large state space, search algorithms tend to be slow. As a result, research in artificial intelligence examined ways to speed up searches by reducing the size of the state space being searched. This is done by limiting how far to look ahead or by not looking at moves that are a waste of time (e.g., moving back to a location that was already occupied). Other ways to reduce search time include the use of heuristics (rules of thumb) that sacrifice the guarantee of finding an optimal solution, in return for finding a reasonable solution in significantly less time.

One of the advantages of the SPA approach is its adaptive nature. The actions of the system do not need to be specified at design time. Instead, the system, through the use of its planning algorithm, determines its own course of action. Due to this ability to independently resolve what to do, the SPA approach is classified as a deliberative approach.

Unfortunately, SPA does not perform well with all autonomous systems. What works well with a static chess board is less ideally suited for a robot travelling at twenty kilometres an hour. Even with advances in artificial intelligence, planning algorithms are relatively slow. With quick moving autonomous systems in dynamic environments, the formulated plan could end up being based on start and goal states that are no longer valid. This state invalidation could be due to the autonomous system and/or an external entity changing its position during the time the planner takes to devise the required sequence of actions. As a result, the abilities of SPA-based autonomous systems to operate in a dynamic environment become limited.

The Subsumption architecture proposed by Brooks [7] is a different approach to autonomous system design. Unlike the SPA approach, the Subsumption architecture relies on defining reactive behaviour to outside stimuli. There is no need to maintain a world model as the world acts as it own model. System behaviour is modelled as tasks that are defined using finite state machines. New behaviour can be added to a system without disturbing previously defined behaviours, and existing system behaviour can be extended by adding higher level behaviour responses that subsume lower ones.

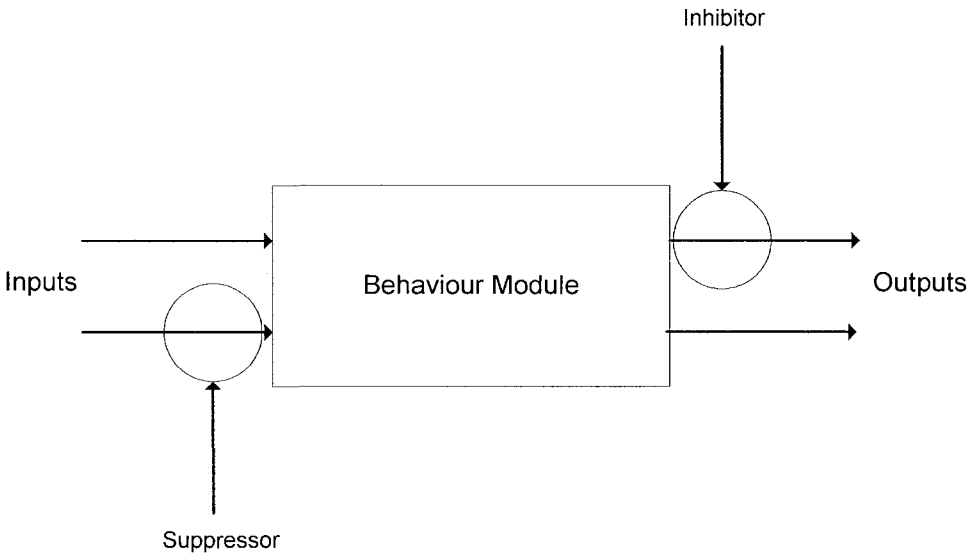

Figure 2.2: Subsumption Behaviour Module

Figure 2.2 shows a high level diagram of a behaviour module in the Subsumption architecture. A behaviour module contains a finite state machine that defines reactionary responses to input signals as well as generated output signals from other modules. Behaviour modules are chained together to define the entire autonomous system – one module's output becoming another's input. The suppressor and inhibitor signals are used by higher level behaviour modules to override input or output signals. Therefore, higher level behaviour modules can override lower ones as required.

An example of a behaviour module could be one that performs an *Avoid* behaviour that, when given the input of an obstacle position, generates the necessary output to avoid the obstacle. Another module that performs a *Turn* behaviour takes the output of the *Avoid* module as an input and generates the output to command a motor to generate the physical manoeuvre. A higher level behaviour module that performs a *Jump*

behaviour could inhibit the output from the *Turn* module if it was preferable to jump over the obstacle instead of turning away.

The essence of the Subsumption approach is that complex behaviours should not be implemented by a complex system, but instead as a combination of simple responses to a complex environment. In a way it tries to emulate how insects can respond intelligently to their environment without the need for high-level thought. The manner by which the Subsumption architecture works by reacting to its surroundings classifies it as a reactive approach to autonomy.

Initial implementations of the Subsumption architecture were very successful. Robots were able to perform simple tasks with a speed of execution previously unattainable using deliberative approaches. However, due to the lack of storing any world model information, there was a limit on the complexity of tasks they could perform. For example, a robot trying to find its way out of a maze could manoeuvre quickly without hitting the walls. However, if the robot came to a dead end, requiring it to reverse course, there was the chance that it could end up repeatedly searching ground that had already been covered. This was due to the fact that it did not store any knowledge of where it had already been.

The combination of deliberative and reactive approaches led to hybrid approaches to autonomous systems. The intent was to combine the intelligent behaviour of a deliberative approach with the speed of execution of a reactive approach. The most common is a three-layer architecture reviewed by Gat [8]. As the name suggests, the three layered architecture consists of three layers of functionality, illustrated in Figure 2.3.

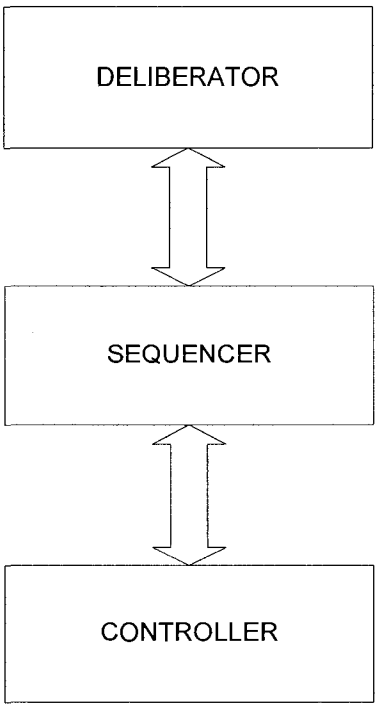

Figure 2.3: Three Layer Architecture

Each layer in the architecture represents a different class of functionality within the autonomous system. The characteristics of each layer are outlined below:

Deliberator Layer: The slowest executing functions reside at this level. The planning algorithms that were part of the plan stage in the deliberative SPA approach would reside here. Examples of the algorithms in this layer include state search algorithms which consider the current state and future states to determine the next course of action.

Sequencer Layer: Functions in this layer do not need to respond as quickly as those at the Controller layer, but they must still have short response times. The role of functions at this level is to control the behaviour of the autonomous system by selecting actions to be performed at the Controller level.

Controller Layer: Low level control functions reside in this layer. These functions have either no reliance on state or rely on current state only. These functions require extremely quick response times. This is the layer in which real time control loops reside.

In the three-layer architecture, each layer is temporally independent of the other. The highest layer will tend to operate the slowest while the lowest layer will tend to operate the quickest. The advantage of separating functionality based on temporal requirements is that tasks which take longer to complete, such as artificial intelligence planning algorithms, do not impede the performance of lower level tasks that need quick response times to function correctly.

The three-layer architecture approach is an abstract concept and its implementations can vary greatly. One of the biggest differentiating factors whether control originates from the highest layer (a deliberative approach), or from the middle layer (a reactive approach). Also, implementations of the middle Sequencer layer vary greatly. Further discussion on these points is covered in Chapter 3.

2.2 Component-Based Systems

Component-based system development is based on decomposing a system into functional components. The characteristics of a component-based system, as defined by Heineman [11], are:

- A component is an independent entity that can be deployed without modification and conforms to a specified standard,
- A component model defines interaction and composition standards, and
- A component model implementation is required to support the execution of components.

The main goal for component-based systems is to encourage reusability. This is accomplished by emphasizing that components act as independent entities. A component will tend be more reusable if it has high cohesion and low coupling. The level of cohesion is a measure of how focused a component's behaviour is. The level of coupling is a measure of how much the component depends on other components to function.

The component model implementation enables the components to interact with each other. It provides a specification standard for defining how a component provides access to its behaviour, typically through an interface. The component model also specifies how other components access the interface.

There are many well-known component models including the Microsoft Component Object Model (COM+) [12], Sun Microsystems Enterprise JavaBeans [13], and the Common Object Request Broker Architecture (CORBA) [14]. Furthermore,

work had been done to develop component models to support real-time systems. The Real-time CORBA specification [15] extends CORBA with real-time services such as fixed priority scheduling, priority banded connections, resource management, and thread pool configuration.

2.3 Publish/Subscribe Messaging

The motivation of using publish/subscribe messaging is to allow components to communicate without needing to know about each other's existence. Instead of specifying the source or receiver(s) of a message, all that needs to be specified is the type of message being sent or received. The publish/subscribe messaging paradigm derives its name from the mechanism by which this is done. Components that are sources of information declare themselves as publishers of a specific message type. Components that are receivers of information declare themselves as subscribers to a specific message type. A component may publish and/or subscribe to many different message types.

Publish/subscribe messaging is often implemented using middleware software. As suggested by its name, the middleware sits 'in the middle' of all software components and manages communications amongst them. Whenever a component publishes a message, the middleware ensures that the message is distributed to all of the components that are subscribers to it. Middleware often simplifies distributing components on different machines, as it manages all of the low-level details required to pass information from one machine to another. This includes any necessary data marshalling, which can include standardizing data across different platforms.

2.4 State Modelling

An important concept in modelling system behaviour is the state machine. Traditionally, system behaviour was described using a finite state machine (FSM). An FSM describes the system as being in any one of a number of states. The machine can transition from one state to another upon receiving a stimulus. Outputs may also be generated by state machines. A Moore state machine [16] generates an output based on the state it enters, whereas a Mealy state machine [17] generates an output based on the current state and the stimuli. Figure 2.4 shows an example of a Mealy state machine. For example, if the system is in the initial state S_0 and receives an input of *1*, it will transition to state S_1 and output a *1*. If the system receives an input of *1* while in state S_1, it will remain in state S_1 and output a *0*.

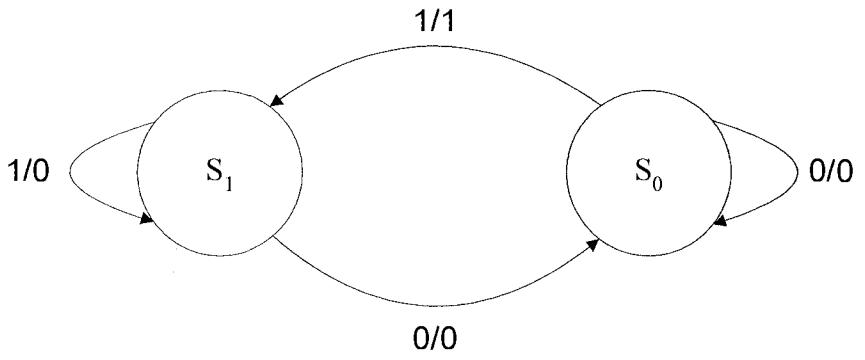

Figure 2.4: Mealy State Machine

While Mealy and Moore state machines can be used to model simple behaviour, Harel exposed several limitations with them [18]. Large systems with extensive behaviour require a large number of states which will result in large unwieldy state machines with many transitions. This counters the purpose of state machines as a means to clearly describe system behaviour. State machines also have no means of handling concurrent behaviour where different parts of the system can be in different states.

As a solution to the observed limitations, Harel introduced statecharts [18]. To handle complexity, statecharts allow for hierarchical decomposition of states. Furthermore, transitions in statecharts are more powerful, as they include the ability to transitions to previous states, and specify guard conditions. Statecharts also support orthogonal states in which high-level composite states can contain concurrent sub-states.

The value of using Harel statecharts is that they provide a well-defined formal semantics to describe system behaviour. This ensures a precise understanding of exactly how the system should behave. Well-defined semantics also enable the use of automatic code generation tools and state-based testing tools to validate system behaviour [19].

The power and usefulness of Harel statecharts was influential in the Unified Modelling Language (UML) approach to modeling object behaviour. In UML 2, statecharts are called behavioural state machines [20]. The next section highlights some of the key syntactic elements of behavioural state machines. It is not meant to serve as an exhaustive coverage of all of the syntax of UML state machines. Instead it covers the syntax as required for the thesis. A detailed overview of UML state machine syntax can be found in literature by Douglass [21] and the Object Management Group [20].

2.4.1 UML 2 Behavioural State Machine Syntax

UML 2 behavioural state machines include a number of syntax elements. Figure 2.5 shows some of the syntactic elements used in this thesis. The boxes with slightly rounded corners represent states that are labelled with the name of the state they represent.

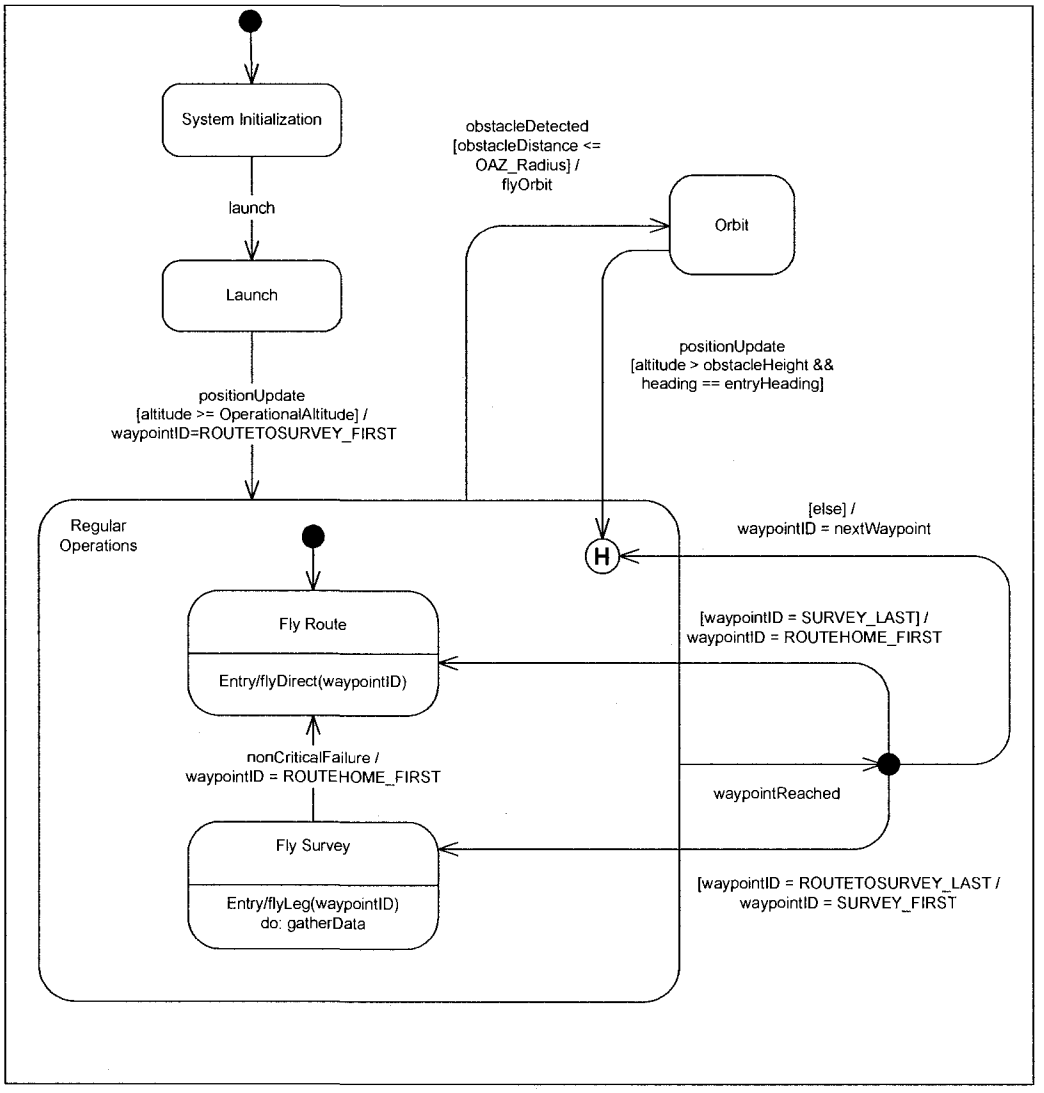

Figure 2.5: UML 2 Behavioural State Machine

Transitions are represented by arrows directed from one state to another and labelled with the name of the trigger that causes the transition. Transitions may also include guard conditions and actions as described below. A black dot with a transition extending from it is the symbol used for identifying an initial state. As shown Figure 2.5, receiving the *launch* trigger while in initial *System Initialization* state will cause the object to transition to the *Launch* state.

A guard condition is an expression (syntactically denoted by surrounding it using square brackets) that can be added to a transition. The expression *[altitude >= OperationalAltitude]* in the transition out of the *Launch* state is an example of a guard condition. The guard condition must evaluate to true for the transition to occur.

A composite state is a state that consists of two or more sub-states. In Figure 2.5, *Regular Operations* is a composite state with two sub-states, *Fly Route* and *Fly Survey*.

A junction point allows a transition to be made to one of many states based on a condition. The *waypointReached* trigger causes a transition out of the *Regular Operations* state to a junction point which continues to the *Fly Route, Fly Survey*, or history state depending on the value of *waypointID*.

The history state, denoted by a circle with an *H* in it, can be used with composite states to enable a transition into a previous state. In Figure 2.5, the first transition to the *Regular Operations* composite state from the *Launch* state will be to the *Fly Route sub-*state. However, if the object then transitions to the *Orbit* state, upon returning to the *Regular Operations* composite state the object will return to whichever of the two sub-states it was in previously.

Actions are operations that occur instantaneously and activities are operations that take more than an instant amount of time. Actions can only be specified to occur during a transition or upon entering or leaving a state. Activities can be specified to occur while in a state. In Figure 2.5, during the transition from the *Regular Operations* state to the *Orbit* state, the *flyOrbit* action takes place. Whenever the object enters the *Fly Route* state, the *flyDirect* action takes place. While in the *Fly Survey* state, the *gatherData* activity takes place. Upon transitioning out of the *Fly Survey* state, the *gatherData* activity stops.

2.5 The GeoSurv II UAV Project

A large motivating factor for this research is the GeoSurv II UAV project being undertaken at Carleton University. The goal of the project is to build an autonomous fixed-wing UAV to perform aerial geophysical surveys.

Traditionally, aerial data-gathering surveys have been flown using human-controlled aircraft. With the cost of fuel on the rise, significant cost savings could be realized if the same mission were to be flown by a much smaller unmanned aircraft. Aerial surveys are typically flown over a large distance (sometimes referred to as over-the-horizon operation), which precludes the ability to control them using radio frequency transmitters. While satellite communications are an option, the cost of maintaining a direct continuous link would negate the potential cost savings of using a UAV. Furthermore, there is always the possibility that remote communications could be

compromised. As such, a critical requirement of the GeoSurv II UAV project is that it is able perform its mission autonomously [22].

A typical survey mission by the GeoSurv II would involve two operators driving to a suitable launch location and assembling the UAV. The UAV is launched using a catapult-style launch system. The UAV would then fly to the survey area and upon reaching that location would commence flying the survey and recording data. The resolution of the data-gathering equipment requires the UAV to fly close to the ground at speeds between 60 and 100 knots. This necessitates a robust obstacle detection and avoidance system. Periodically during the flight, the UAV sends transmissions back to the ground station that include updates on its position, velocity, fuel status, and system health. After completing the survey, the UAV flies back to its launch location and performs a parachute descent recovery [22].

The GeoSurv II UAV is an ambitious project with stringent requirements for reliable autonomous behaviour and assured obstacle detection and avoidance. Success will depend on reliable autonomy software that incorporates proven strategies for fixed-wing UAV obstacle avoidance.

Chapter 3

State of the Art

A review of the state of the art that influenced this research is outlined in this chapter. The topics discussed are autonomous system architecture, simulating autonomous systems, and autonomous obstacle avoidance.

3.1 Autonomous System Architecture

While the hybrid three-layer architecture introduced in Chapter 2 provides an abstract foundation for autonomous system design, researchers have devised different interpretations of the approach. This section describes many of the proposed variations.

3.1.1 Deliberative Hybrid Architecture

A deliberative hybrid architecture can be described as a version of the three-layer architecture where the Deliberator layer is in control. This control comes from the Deliberator layer calling the Sequencer layer to perform tasks. Examples of this type of architecture include 3T, CLARAty, and agent-based autonomy, which are discussed in this section.

3.1.1.1 3T Architecture

Bonasso et al. adapt the three-layer architecture in their 3T (three tier) approach [23], as illustrated in Figure 3.1. The system is assigned a number of goals, which the highest Deliberation tier uses to formulate a plan that consists of tasks for the system to complete. The plan activates tasks by selecting their corresponding Reactive Action Package (RAP) which describes the details of executing the task. The Sequencing tier executes the selected RAP. All elements of a task are executed by activating actions in the Reactive Skills tier. The Reactive Skills tier interacts directly with the system's actuators and sensors to perform the required actions.

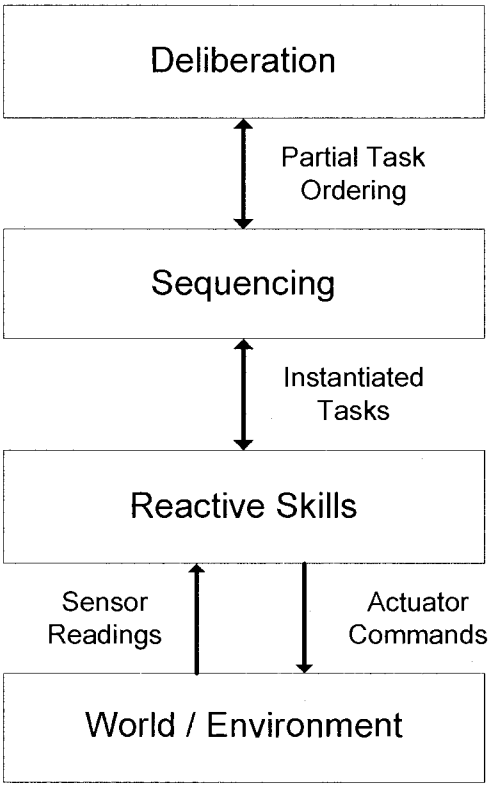

Figure 3.1: 3T Intelligent Control Architecture

3.1.1.2 CLARAty

The Coupled Layer Architecture for Robotic Autonomy (CLARAty) [24], shown in Figure 3.2, is a project undertaken by NASA to develop reusable robotic software. CLARAty is a deliberative hybrid architecture, based on the three-layer approach with some slight modifications. The approach has a lower Functional layer similar to the Controller layer; however the two highest layers are combined into a single Decision layer. Also, there are different levels of 'intelligence' within each layer. Whereas the conventional three-layer architecture approach defines each layer with a set level of abstraction, the CLARAty approach allows each layer to have different levels of granularity. As shown in Figure 3.2, the Decision layer has a high level Planner and mid level Executive. In the Functional layer, system behaviour is defined at high and low levels of control.

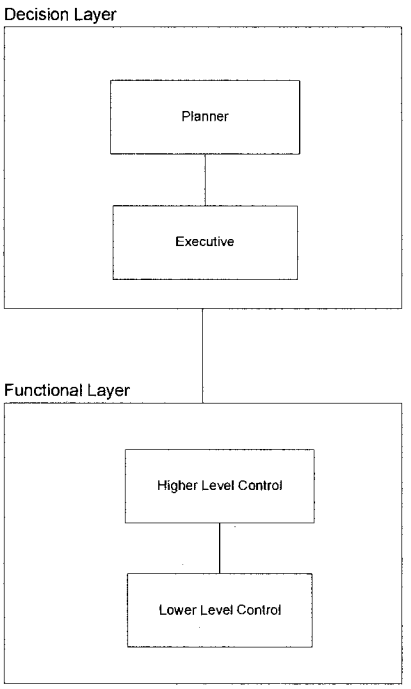

Figure 3.2: CLARAty Architecture

The Decision layer in CLARAty is described by its own architecture [25] called CLEaR (Closed Loop Execution and Recovery). The approach is similar to the 3T architecture in that it uses a planner to generate tasks to be carried out by the executive. The planning is carried out by the CASPER [26] planner. CASPER (Continuous Activity, Scheduling, Planning, Execution, and Replanning) is an iterative planner that uses goals and current state to generate a list of activities to be performed by the Executive. The Executive is implemented using TDL (Task Description Language), a sequencing language that is discussed in Section 3.2. CASPER updates its plans based on information obtained from monitoring current state and execution status.

3.1.1.2 Agent-Based Autonomy

Another means of implementing a deliberative hybrid approach to autonomy is to use an agent-based approach. An agent is defined as a system that stores information and uses the information, as well as a knowledge base, in determining its next course of action. Research into using the Jack Intelligent Agents programming language is described by Karim et al. [27]. Jack uses the Belief Desire Intention (BDI) framework to try and emulate rational human thought. Beliefs represent the internal database of information known by the agent, desires represent its goals, and intentions are the plans the agent uses to achieve those goals. The BDI framework is a form of plan generation that tries to match current beliefs with intentions to achieve the defined goals.

```
+arrivedAtWaypoint(X):surveyStartPoint(X) <-
                                !gatherData();
                                !flySurveyRoute().
```

Figure 3.3: Behaviour specified in BDI using AgentSpeak

Figure 3.3 shows a simple example of AgentSpeak [28], a language developed to define BDI agent behaviour. The example shows some of the grammar that can be used to define BDI plans. The '+' symbol indicates the occurrence of an event. The plan shown in Figure 3.3 deals with an *arrivedAtWaypoint* event. The expression after the colon is evaluated to attempt to match it with a known belief. In the example, the expression checks if the waypoint that was arrived at, denoted by the variable *X* is the waypoint that signifies the start location of the survey. If the expression can be matched, the expressions after the <- are executed. In the example, two new goals are created: One goal to gather survey data, and another to fly the survey route.

Once plans are formed, a selection mechanism is used to select which plan to execute. Plans are then executed by specifying tasks to be handled at the Sequencer layer.

3.1.2 Reactive Hybrid Architecture

A reactive hybrid architecture can be described as a version of the three-layer architecture where the Sequencer layer is in control. The Sequencer layer accomplishes tasks by calling on the services of procedures at the Controller and Deliberator layers. Examples of reactive hybrid architecture follow.

3.1.2.1 ATLANTIS Architecture

The ATLANTIS (A Three-Layer Architecture for Navigating Through Intricate Situations) architecture by Gat [29] is shown in Figure 3.4, and is very similar to the 3T architecture. As shown in Figure 3.4, each communication between layers falls within a specified frequency of execution. The biggest differentiating factor with ATLANTIS when compared to the 3T architecture is that the Sequencer layer controls all execution. In the 3T architecture, the Deliberation layer formulates plans that call specific Sequencer tasks, whereas in ATLANTIS, the Sequencer layer manages tasks that call on procedures in the Deliberator as required to use their planning capabilities. While the Sequencer continues with task execution, the Deliberator procedures calculate responses which are provided to the Sequencer layer upon completion.

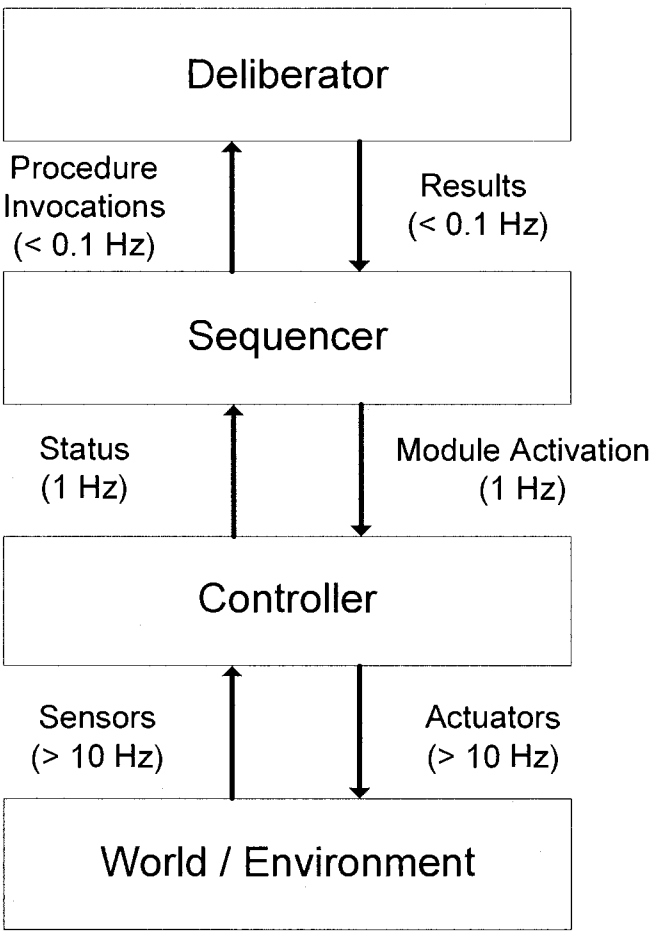

Figure 3.4: ATLANTIS Control Architecture

3.1.2.2 Apex – Reactive Planner

Apex is a reactive planner, developed by Freed et. al. [30] that uses Procedure Control Language (PCL) to specify system behaviour. It has been used in the NASA Autonomous Rotorcraft Project (ARP) project.

The methodology of Apex is to model human behaviour by trying to emulate human cognitive thought. Missions are modelled as goals, and there is a library of stored

partial plans that can be used to meet those goals. The system considers the current goals dynamically and works to match plans that will ultimately lead to the desired outcome.

Although Apex is modelled as a planner it differs from the classical planners used in a deliberative approach. Classical planners work by constructing new plans, whereas Apex uses a library of stored partial plans. This limits the flexibility with which Apex can deal with a situation, however the advantage is that since plans are already formulated a lot less computation needs to occur. Plans in Apex are specified using PCL (Procedure Control Language).

3.1.2.3 WITAS – Reactive Concentric

The Wallenburg Laboratory for Information Technology and Autonomous Systems (WITAS) [31] use a Reactive Centric hybrid of the three-layer autonomy architecture in their design of an autonomous rotorcraft UAV. The architecture, as shown in Figure 3.5 is controlled by the middle Reaction layer which calls on services from the Deliberation and Control layers. The approach differs somewhat from a true reactive hybrid because it still allows the Deliberation layer to have a limited amount of control. Missions may be generated at the Deliberation layer using a planner, which can call on the Reaction layer to start executing a task [31]. If a task at the Reaction layer initiates a Deliberation planning service, then the planner can respond by initiating a task of its choosing.

Another unique characteristic of the Reactive Centric architecture is the notion of layer overlap as indicated in Figure 3.5. The Reaction layer consists of multiple hierarchical task components called Task Procedures (TPs). TPs that call high

Deliberation services lie in the region defined as Hybrid Deliberation. TPs that deal with system control lie in the region defined as Hybrid Control. TPs are covered in further detail in section 3.2. The distinguishing feature of this approach is the blending of the sequencing language into the higher and lower layers. Many TPs can be operating concurrently and at different temporal rates depending on their task. For example, a TP in the Hybrid Deliberation region could call a path planning algorithm and wait for the result, thus operating at a slow temporal rate. Another TP could issue control commands to the aircraft and operate at a fast temporal rate.

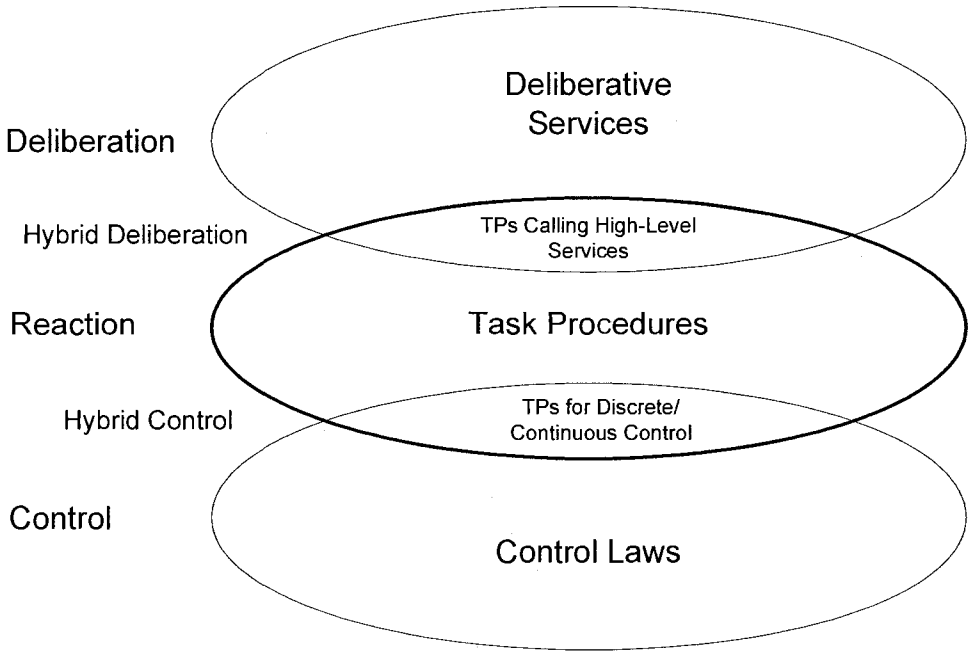

Figure 3.5: WITAS Reactive Concentric Architecture

3.2 Sequencing Languages

The Sequencer layer is fundamental to the three-layer architecture approach and is responsible for managing the execution of tasks. In a deliberative hybrid approach, the Sequencer layer is called by high level planning algorithms to perform tasks by calling upon services in the Controller layer. With a reactive hybrid approach, the Sequencer layer controls system execution by sequencing the execution of multiple tasks related to the system's autonomous behaviour. Research has examined better ways of defining the sequencing role. This section covers different sequencing languages that have been developed.

3.2.1 Reactive Action Packages

Firby proposes the Reactive Action Packages (RAP) approach to sequencing [32]. The RAP execution system was originally developed to enable the sequencing for a deliberative hybrid approach. A RAP is a program that describes how to carry out a task. A task's execution is accomplished by a combination of primitive actions and invoking other tasks described by other RAPs. Planners at the Deliberator layer generate abstract plans which the RAP system sequences into primitive actions carried out by the robot controller.

3.2.2 Task Description Language

The Task Description Language (TDL) is a task-oriented approach to sequencing that was developed by Simmons et. al. at Carnegie Mellon University [33]. TDL is an extension of C++ that is used to simplify the specification of autonomous system control. System behaviour is defined using task trees. Each node in the tree has an action associated with it. The action can be an external action, or it can involve spawning more children nodes which divide the task into further subtasks.

There are two types of nodes in a task tree. Goals, represented by circular nodes, represent high-level tasks made up of other subtasks represented by its child nodes. Commands, represented by rectangular nodes, represent executable behaviours.

Figure 3.6 illustrates the notion of defining autonomous behaviour using a task tree. The given example defines the high-level behaviour for an unmanned aerial vehicle performing an aerial survey as three goals (fly to survey site, fly the survey, and fly home). Each goal has its own commands at the leaf nodes of the tree.

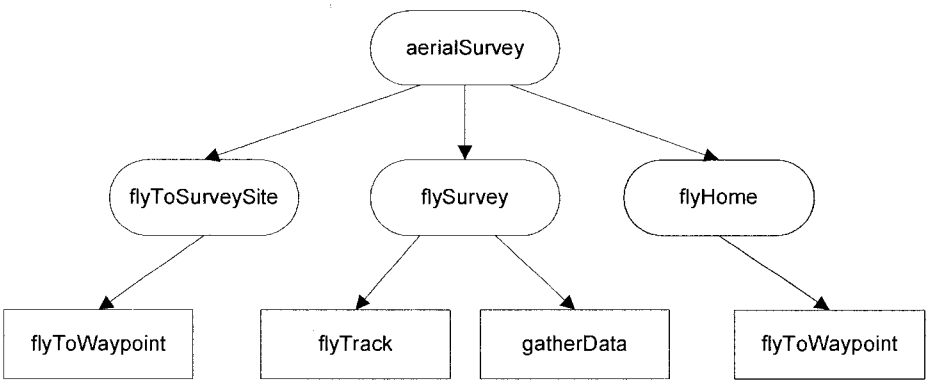

Figure 3.6: Task tree for an aerial survey

A task tree does not represent a static structure, but instead is generated dynamically when TDL code is executed. The Task Description Language is used to encode the rules used to generate the task trees and includes special keywords that can be used to cast restrictions on the execution of tasks. These restrictions can include sequential or parallel execution, or the requirement for specific conditions to be met before task execution can begin. For example in the task tree in Figure 3.6, TDL could specify that the second row of task goals execute in sequential order from left to right, and that the two commands under flySurvey execute in parallel. Simmons' motivation for developing TDL was to develop a high-level language for specifying autonomous system behaviour. The use of high-level keywords absolves the management of sequential and concurrent activity from the user.

3.2.3 WITAS Task Procedures

Task Procedures (TPs) are used by the WITAS UAV system for sequencing execution [31]. The concept of TPs is similar to that of RAPs and TDL in that they represent a breakdown of larger tasks into smaller tasks. TPs can also invoke the creation of other TPs as required. As part of the WITAS UAV architecture, the TPs are implemented as independent CORBA objects that communicate with each other using CORBA event channels. TPs are defined using an XML-based code mark-up language called the Task Specification Language (TSL). A TP specification includes CORBA-specific initialization parameters as well as task behaviour defined using a state automaton. The

Flight Command Language (FCL) was developed for use by hybrid controller TPs to issue commands to the UAV.

3.3 Autonomous System Simulation Architecture

Johnson et al. [34] specify that in order to develop an autonomous UAV, a simulation architecture should have the ability to:

- test all custom software more rigorously than possible in a safe flying environment,
- test all hardware and software in real-time,
- rehearse all mission procedures and flight plans,
- visualize recorded flight data,
- reconstruct flight events after the fact,
- be used on location at the flight test site, and
- validate flight test data.

The following section covers different approaches used for autonomous system simulation. Differences lie in how elements of the target environment are adapted to be tested in the simulation environment.

3.3.1 Approaches to Simulation

The Wallenburg Laboratory for Information Technology and Autonomous Systems (WITAS) uses a modular simulation platform [31]. The simulation platform consists of a simulation server, object state simulator, and a renderer. The simulation server registers simulation modules and acts as an intermediary of data exchange between the autonomous system architecture and simulated actuators and sensors. The object state simulator simulates the physical behaviour of vehicles or other physical devices in the real world. The renderer generates images based on the system's location to be used to simulate visual sensors.

The NASA Ames' Mission Simulation Facility (MSF) has created a simulation framework for testing autonomy software for planetary exploration vehicles. One of the main features of their simulation framework is that it is High Level Architecture (HLA) compliant. This simplifies distributing the simulation across many platforms. Pisanich et al. [35] state that the intention of the framework is to provide greater access to researchers wanting to develop autonomy-related algorithms. Often times, limitations in budget, time, and expertise preclude researchers from developing their own software for experimentation. By having the framework, a simulation can be assembled rather quickly. The MSF is a library of readily accessible simulation modules whose functions include modeling terrain, robotic kinematics, instruments, and mission resources.

The Real-time Interactive Prototype Technology Integration/Development Environment (RIPTIDE) is a simulation environment developed by the NASA Ames Research Centre for real-time control system simulation [36]. RIPTIDE can be used to simulate autonomous control systems in a dynamic real-time environment using a high-

fidelity vehicle model. In order to simulate processes in the RIPTIDE environment, the controlling software processes must be adapted to communicate with each other using shared memory.

The Georgia Tech GTMax UAV project uses a simulation tool that enables hardware-in-the-loop testing [37]. The simulation tool includes an aircraft model and sensor models. In order to use control software in the simulation, the code from the onboard computer is compiled into the simulation tool. In addition, the onboard computer may be connected directly to the simulation computer. In the latter case, the simulation is controlled by the actual flight control hardware which interacts with the simulated aircraft and sensors in the simulation tool.

3.3.2 Simulation Driven Development

In contrast to the approaches mentioned in the previous section, work has been done examining the potential for simulation to play a larger role in the development process. The Modeling and Simulation-Driven Engineering (MSDE) approach advocates using discrete-event simulation architecture as the final target architecture for a system [38]. With such an approach, system components begin as simulation models that are incrementally refined and replaced with the actual hardware components. Doing so eliminates the need to abandon components outright when moving to a target platform. The benefits of this are shortened development time, increased reliability, and reduced risk [39].

3.4 Obstacle Avoidance

One of the biggest risks faced by moving autonomous systems is the presence of obstacles. All forms of autonomous vehicles must include a degree of obstacle avoidance to prevent them from damaging themselves or others due to collision. Autonomous obstacle avoidance is also a concern with human-controlled UAVs because there is no guarantee that the radio frequencies used to control these systems could not be compromised. To deal with such an occurrence, many UAVs have pre-programmed flight manoeuvres to return to a pre-designated landing area (lost-link scenario). However, as these UAVs are not currently equipped to deal with obstacles, air traffic controllers need to halt all other aircraft traffic if this happens.

A lot of research has been done on ground-based obstacle detection and avoidance. While fixed-wing aircraft have unique characteristics to consider when devising obstacle avoidance algorithms, techniques from ground-based avoidance can be applied. The following sections outline different obstacle avoidance techniques including reinforcement learning, planning, and reactive approaches.

3.4.1 Reinforcement Learning

Stanford University's winning entry in the 2005 DARPA Grand Challenge is described by Thurn et al. [40]. Their approach to obstacle avoidance employs reinforcement learning and two primary sources of detection – laser terrain mapping and computer vision terrain analysis.

Laser terrain mapping is used for short and medium range obstacle detection. Five scanning lasers, mounted abreast on the roof of the vehicle are employed to provide a three dimensional view of the terrain in front of the vehicle 45 degrees to each side. Each laser provides 181 range measurements at 0.5 degree increments to the ground ahead. These measurements are converted to three dimensional coordinates relative to the vehicle's estimated position. Each of the five lasers are mounted at different scan angles so that combined they covered the ground out to 30 metres away from the vehicle. A classification algorithm is used to group similar points based on their height coordinates and to classify the ground as being one of three types: driveable, occupied, or unknown. The result is the accurate detection of terrain type out to a range of 22 metres.

Laser terrain mapping is highly accurate, but due to its limited detection range, a second means of detection is needed for obstacles located further away. The second means of detection is a colour camera which is used to try and extend the detection range out to 70 metres ahead of the vehicle. Regions in the image from the camera are compared to the range classification information from the laser range detectors. The laser range detectors can provide accurate ground classification for the nearest 22 metres in front of the vehicle, and these classifications are used as a form of reinforcement learning to train the camera algorithm the colour of the terrain that each of the three classifications (driveable, occupied, unknown) represent. The camera algorithm attempts to match regions between the 22 metre and 70 metre point with the known classifications based on their image alone, to predict whether the terrain is passable or not. The algorithm constantly uses the first 22 metres to make its determination, therefore the premise is that

the road will appear the same if the terrain changes and the matching colours will change as well. Due to the fact that the camera algorithm is less accurate than the laser detection, Thurn et al. adapt the speed of the vehicle depending on the camera information. When the camera algorithm detects passable terrain, the vehicle moves ahead at full speed. However, when the camera algorithm detects noise and is unsure about classifying passable terrain, the vehicle slows down to 25 miles per hour to rely on laser range detection information alone. Slowing the vehicle acts as a fallback mechanism to deal with uncertainty in long-range obstacle detection.

3.4.2 Planning Obstacle Avoidance

Another approach to obstacle avoidance is path planning. Many path planning algorithms are known to be computationally complex and would not return the necessary results within a reasonable amount of time. However, the path planning proposed by Saunders et al. [41] uses the Rapidly-Expanding Random Tree (RRT) algorithm [43] to find an acceptable route through known obstacles. The RRT algorithm is based on exact Voronoi diagram computation and can be used for multi-dimensional obstacle avoidance. Pseudo code for the RRT algorithm is shown in Figure 3.8.

1. Pick x_{rand} in R^3
2. Determine the node x_{near} in the tree that is nearest x_{rand}
3. Move an incremental distance from x_{near} toward x_{rand}, resulting in a new state x_{extend}
4. Search along the segment x_{near} to x_{extend} for collisions with known obstacles
5. If no collision is found, add the new node (x_{extend}) and the edge (x_{near} to x_{extend}) to the tree
6. Test to see if x_{extend} can be connected directly to x_{goal}
7. If so, add the valid path from x_{start} to x_{goal}
8. Go to step 1

Figure 3.7: Pseudo Code for the Modified RRT Algorithm [43]

The modified RRT algorithm is given a start and end waypoint. It picks a random point in the search space between the two points. It finds the closest node (not including the end waypoint) and adds a new waypoint that is a distance D from the closest node and in the direction of the random point. The algorithm then checks for a collision with an obstacle along the generated line between the closest node and the new waypoint. If a collision is found, the waypoint is discarded; otherwise it is added to the list of nodes. Upon adding the new node, the distance from the new waypoint to the end waypoint is also checked. If the distance is less than D, then a path from the new waypoint to the end waypoint is created and the path through the obstacles is complete.

3.4.3 Reactive Obstacle Avoidance

The purpose of reactive obstacle avoidance is to provide an immediate response upon detecting an obstacle. If the vehicle is unable to stop, as is the case with a fixed-wing UAV, there may not be enough time to wait for a response from a planning algorithm.

An obstacle avoidance strategy for a fixed-wing UAV is presented by Saunders et al. [41]. Obstacles are classified into two categories: a priori known and pop-up obstacles. Known obstacles are obtained from terrain maps whereas pop-up obstacles are detected by on-board obstacle detection sensors. An algorithm to avoid pop-up obstacles is presented that is based on using one light-weight laser range finder.

The reactive avoidance algorithm relies on a single laser range finder that points directly out to the front of the UAV. The laser can detect the distance of an obstacle but not its size or height. Therefore, the obstacle is modelled as a cylinder with a radius R equal to the minimum turning radius of the UAV. The value of R depends on the current speed and performance characteristics of the UAV. A new waypoint is plotted a distance of R away from the obstacle that it is perpendicular to the original flight path. The sequence is shown in Figure 3.7. If the obstacle has a radius wider than R, then the laser will detect it again after the UAV has changed course. In this case another obstacle with radius R is plotted in the new location and a new waypoint is inserted to avoid the obstacle by a wider path. The process repeats until the path is wide enough to go around the obstacle.

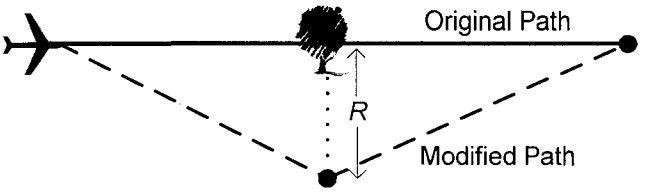

Figure 3.8: Reactive Obstacle Avoidance

It is proven by Griffiths et al. [42] that in using the reactive obstacle avoidance algorithm, the UAV is guaranteed to avoid the obstacle as long as the UAV begins turning at a minimum distance of D away from the obstacle where the value of D is defined as:

$$D > \left(\frac{8 + 2\sqrt{6}}{2\sqrt{3}} \right) R$$

Unlike planning algorithms, the reactive obstacle avoidance algorithm proposed by Griffiths et al. only bases its avoidance manoeuvre on a single obstacle, however it will generate an immediate response.

Chapter 4

The Thesis

4.1 Statement of Thesis

The thesis of this research is that autonomic capability for a fixed-wing UAV can be built using a generic three-layer architecture, in which a central layer resolves conflicts from competing sources and includes a fallback safety mechanism for obstacle avoidance.

4.2 Analysis

As described in Chapter 3, current approaches to autonomous system development are based predominantly on a deliberative hybrid or reactive hybrid approach.

With the deliberative approach described in Section 3.1.1, control occurs at the deliberative layer using either classical planning or an agent-based approach. Rules are used by the system to formulate its own plans to respond to events, rather than relying on behaviour that is specified at design-time. The limitation with this approach is reduced predictability. Without adhering to a specific sequence of actions, an unforeseen combination of events could result in unexpected behaviour.

Limitations with the reactive approach exist in the current methods of sequencing which, as described in Section 3.2, rely on a task-oriented approach. Tasks are dynamically activated at execution time resulting in multiple simultaneously executing

tasks. Each task functions by sending control commands to other subsystems. When more than one task tries to command the same subsystem, conflicts can occur. In certain circumstances, one task may need to take precedence over another, but often a simple task priority system will not suffice. There could be circumstances where a task's priority might depend on the current state of the system. Another limitation is that current sequencers employ custom languages to define autonomous behaviour. This could be a hindrance to UAV certification for operation in civil airspace. Without a standardized means of describing the autonomous behaviour, there could be misinterpretations of the semantics leading to improper conclusions being drawn about the design.

Obstacle avoidance represents a significant challenge for autonomous UAV development. As described in Section 3.4, successful obstacle avoidance relies on the accuracy of obstacle detection sensors and the algorithms that use this information to plan a route around them. Limitations exist with the detection of obstacles at longer ranges, and the fact that planning algorithms do not have a guaranteed response time. A reactive approach to obstacle avoidance for a fixed-wing UAV will act if a planner has not yet responded, however it only considers the position of a single obstacle in planning the route. Wheeled and rotorcraft autonomous systems can deal with these limitations by implementing fallback mechanisms such as reducing their speed or stopping to rely on shorter range obstacle detection or to give more time for avoidance planners to respond. On the contrary, a fixed-wing UAV must maintain a minimum forward speed to generate lift. Therefore, a fixed-wing UAV requires a fallback mechanism that allows it to maintain forward motion.

Research to address these limitations would make valuable contributions to the state of the art. A reactive hybrid variation of the three-layer autonomy architecture that has a central sequencer could resolve conflicts from competing tasks. Specifying the sequencer using a standard modeling language such as UML 2 behavioural state machines would eliminate the need for a custom specification language and could simplify the certification of UAV software. Furthermore, integrating a fallback reaction into the sequencer would provide an additional safety mechanism for cases where the planners cannot respond fast enough to avoid obstacles.

4.3 Scope

The scope of the thesis is focused on developing an architecture for implementing autonomous behaviour in a fixed-wing UAV. The result of this effort is the CB3A architecture and the SBED sequencer. The UAVSim Testbed was also conceived to aid in their development and evaluation. Only simulated components are used in the architecture – future work will need to address adding hardware components. The thesis does not evaluate the performance of the middleware used to enable the architecture as that work is being done by another researcher. The scope of the thesis also includes developing a fallback mechanism for avoiding static obstacles. Dynamic obstacle avoidance represents an important area of research for future work.

Chapter 5

The CB3A Architecture

This chapter describes the Component-Based Three-Layer Autonomy (CB3A) Architecture. CB3A is based on a reactive hybrid three-layer architecture approach. As a reactive approach, all behaviour is specified at design time with predetermined plans to react to external events.

5.1 CB3A Architecture

In CB3A, the UAV system is decomposed into functional components. Each component is further classified as belonging either at the Deliberator, Sequencer, or Controller layers. This notion is illustrated in Figure 5.1 which shows a typical set of components associated with an autonomous UAV system. The Deliberator layer contains planning and monitoring components. The Controller layer contains low-level control components such as the autopilot, emergency parachute recovery system, and sensor controllers. The *SBED Sequencer* is the only component that resides at the Sequencer layer.

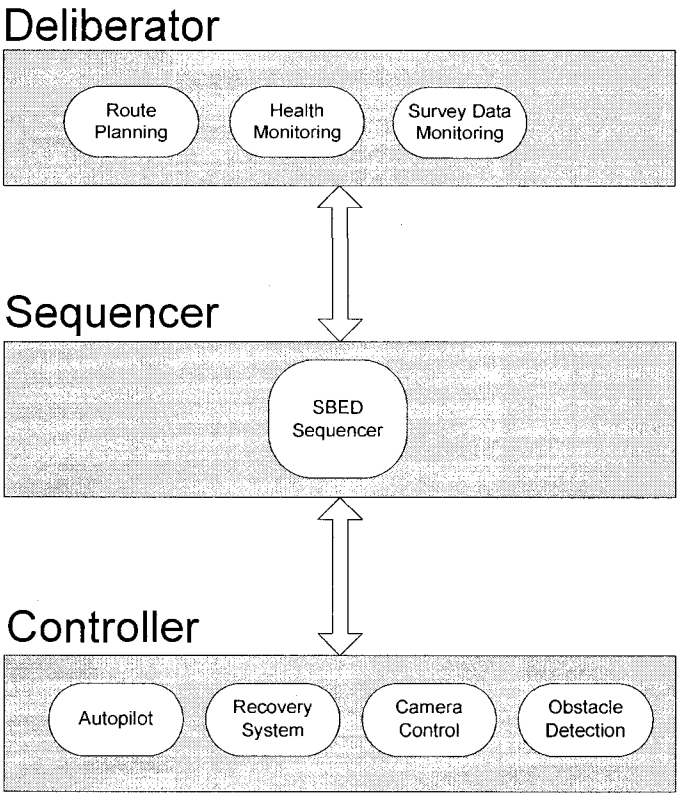

Figure 5.1: Component-Based Three-Layer Autonomy

Figure 5.2 shows a refined view of the CB3A architecture as a component diagram which includes subsystems of the GeoSurv II UAV residing on the three layers. Further information about the components' functionality and the reason for their placement in a specific layer follows.

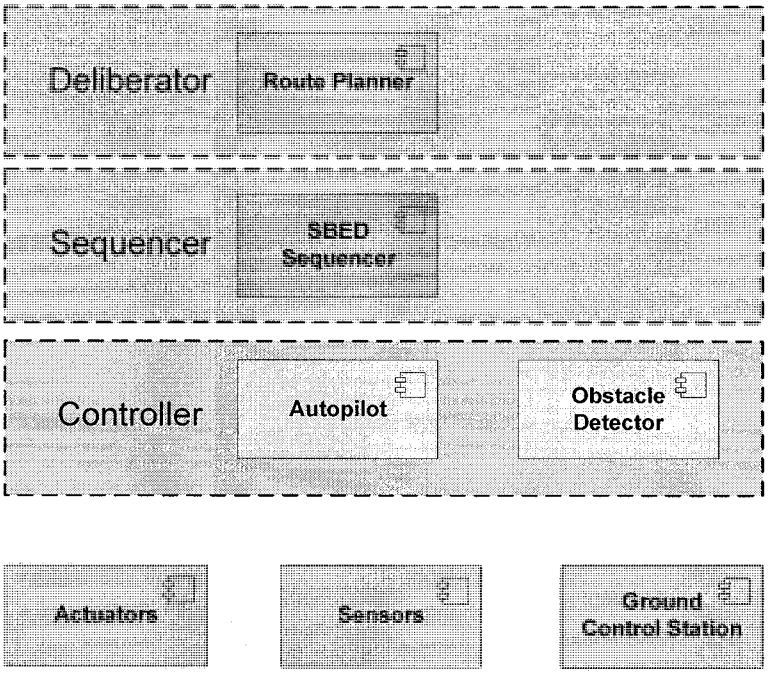

Figure 5.2: GeoSurv II implemented using CB3A

The *Autopilot* executes as a low-level control loop to keep the plane in stable flight. Therefore, it resides in the lowest Controller layer.

The *Obstacle Detector* analyzes information from the obstacle detection sensors on the aircraft. It employs obstacle detection algorithms to determine the presence of static obstacles within the UAV's obstacle detection range.

The *Route Planner* monitors the aircraft's current route and determines if there is a risk of encountering a known obstacle on the predicted flight path. If a known obstacle is located close to the flight path, the route planner will either plot a route around the obstacle or increase the flight altitude as required. The *Route Planner* utilizes an obstacle avoidance planning algorithm and therefore resides on the highest Deliberator layer.

The *SBED Sequencer* is the only component that resides in the middle Sequencer layer. The *SBED Sequencer* is responsible for centralized control of the system's next course of action based on consideration of all input from all other processes. The *SBED Sequencer* initiates actions by generating response commands. According to the three-layer architecture, it must generate a response command within a predetermined time frame (e.g. less than one second) [29]. Chapter 6 covers details of the implementation of the *SBED Sequencer*.

The three other components in Figure 5.2 are required for input and output interaction with the aircraft and human operator. Therefore, they do not reside in a specific temporal layer. The *Sensors* provide information about the aircraft and world state. The *Actuators* are used to control the aircraft and data gathering equipment. The *Ground Control Station* is used by the human operator to communicate with the UAV as required. While the UAV performs its mission autonomously, the *Ground Control Station* allows for periodic monitoring of the UAV's status and issuing override commands.

5.1.1 CB3A Component Requirements

Components in the CB3A architecture communicate with each other by using publish/subscribe messaging. The use of this messaging paradigm helps to increase the architecture's flexibility as explained in Section 5.3. All components in the CB3A architecture must adhere to the following requirements:

- the component must include a middleware wrapper class, which allows the component to operate using a chosen middleware configuration;
- all messages between components must be done using publish/subscribe messaging; and
- all messages between components must conform to a specified protocol which includes the expected response and/or action by the component upon receiving a message.

An important part of the component requirements is that it includes a middleware wrapper to isolate the component from the specifics of the middleware. The advantage of using this approach is greater portability which is discussed in Section 5.3. The component wrapper must implement the *Wrapper* interface shown in Figure 5.3.

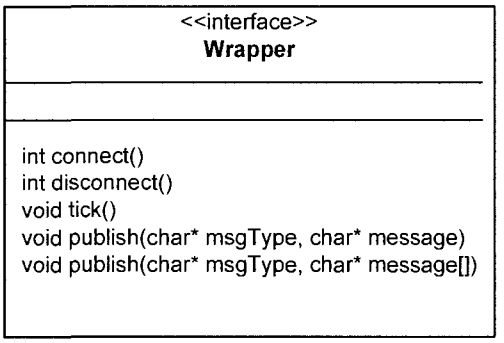

Figure 5.3: Wrapper interface

The methods that must be implemented by a component's wrapper class are described below:

int connect()

This method establishes a connection to the middleware. It notifies the middleware which messages will be published and which messages will be subscribed to. Subscribed messages must be linked to callback methods that are called when messages are received. This method returns a value of 1 if the connection is successful, and a value of 0 otherwise.

int disconnect()

This method disconnects from the middleware and returns a value of 1 if the disconnection is successful, and a value of 0 otherwise.

void tick()

This method must be called periodically to trigger the middleware's evocation of callbacks when messages are received.

void publish(char* msgType, char* message)

This method publishes a message with a single attribute. ***msgType*** specifies the type of message and ***message*** specifies the attribute to publish.

void publish(char* msgType, char* message[])

This method publishes a message with a multiple attributes. ***msgType*** specifies the type of message and ***message[]*** specifies the attributes to publish.

5.2 Evaluating the CB3A Architecture

An autonomous UAV system embodied by the CB3A architecture represents a collection of interacting subsystem components that may be developed independently. The autonomous behaviour relies on many messages being exchange between these components which points to the need for a means to test the system. In order to test the system, many inputs need to be emulated. These include inputs from internal sensors as well as input from external effects such as the interaction of the system with its environment. Furthermore, it is possible that there may be the need to develop a specific subsystem before the other subsystems with which it interacts exist. Lastly, the nature of autonomous systems is that they perform with little or no human interaction which means their behaviour must be exercised thoroughly. All of these factors point to the need for a complete system simulation environment.

The requirement for a system simulation has led to the development of the UAVSim Testbed environment. This environment realizes the CB3A architecture with simulated components. Whereas, the intention is for the system to be gradually evolved from simulated components based on initial concepts into real components, the UAVSim Testbed can be used as part of the development process of a UAV such as the GeoSurv II or it can be used to research new ideas on a single component.

Figure 5.4 shows the components of the GeoSurv II embodied by the CB3A architecture as implemented in the UAVSim Testbed. The diagram is similar to the mapping of the GeoSurv II subsystems to components in the CB3A architecture in Figure 5.2. The *Aircraft Model* and *World Model* components have been added to simulate the aircraft and real world respectively. The *Aircraft Model* represents the real aircraft and its inherent physical characteristics (aerodynamic behaviour) as it travels through the air. The *World Model* represents the outside world and includes terrain, weather, and external forces (e.g. gravity) acting on the aircraft. In Figure 5.3 the *Aircraft Model*, *World Model*, *Actuators*, and *Sensors* are contained within the *X-Plane* component, a COTS flight simulator that has been extended to function as part of the CB3A architecture.

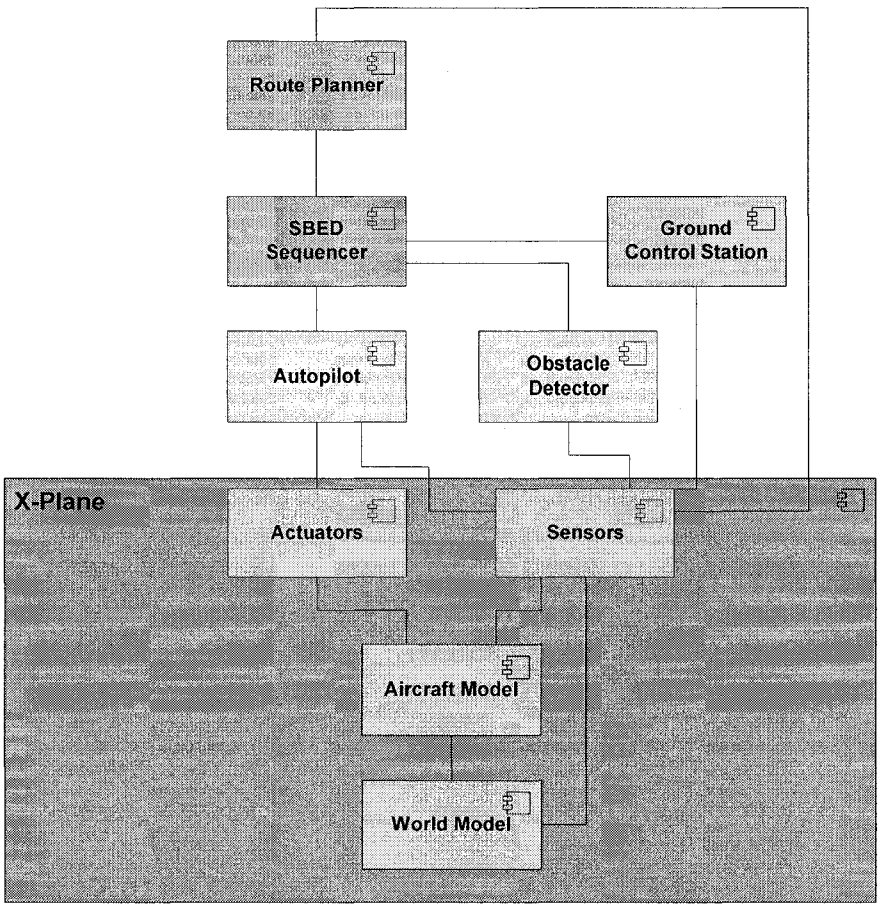

Figure 5.4: GeoSurv II implemented using CB3A in the UAVSim Testbed

Inter-component communications in CB3A in the UAVSim Testbed is done using the Carnegie Mellon University (CMU) Inter Process Communications (IPC) middleware [44]. One of the primary reasons for selecting IPC is its simplicity. IPC can be used to send and receive messages as simple strings. A goal for the CB3A architecture is to facilitate switching middleware implementations if desired. Basing the initial implementation on a simple messaging scheme should make it easier to switch to a middleware with more stringent message structure requirements. The wrapper class's

responsibility is to format the string data as required by the new middleware. Another advantage of the CMU IPC is that it is free and requires no license to operate. This is advantageous for fostering an open research environment that can be used without restriction.

The CB3A architecture was evaluated by executing a prototypal mission of the GeoSurv II UAV in the UAVSim Testbed. As shown in Figure 5.5, the execution of the mission can be visualized in the COTS X-Plane flight simulator in real-time 3-dimensional or 2-dimensional views. A log of execution of state transitions in the SBED sequencer is also recorded to review mission execution.

The mission executed by CB3A is based on a subset of the requirements in the GeoSurv II Systems Requirement Document [22]. It includes the autonomous execution of an initial takeoff; flight to the survey start point based on a specified flight plan; flying legs of the survey according to the plan; returning back to the start point; and performing a parachute recovery. The overhead view in Figure 5.5 shows the UAV flying a survey leg that has obstacles placed in different configurations which the UAV must autonomously avoid.

All autonomous behaviour is controlled by the SBED sequencer and is discussed in Chapter 6. Further explanation on how obstacle avoidance is enabled within the CB3A architecture is covered in Chapter 7.

As exercised in the UAVSim Testbed, the CB3A architecture was effective at controlling the interaction between the UAV subsystem components in a real-time simulation to successfully complete the mission.

Figure 5.5: 3D and 2D views using X-Plane in the UAVSim Testbed

5.3 CB3A Architecture Characteristics

The component-based design of the CB3A architecture using publish/subscribe messaging helps bestow the architecture with greater flexibility and portability.

Flexibility is important in the development process as it makes it easier to switch parts of the system. A truly flexible system allows for any component to be replaced by another that can perform the same task without affecting the rest of the system. The CB3A architecture offers flexibility due to its use of independent components. A component's implementation can be switched as required whether it is to upgrade its behaviour or switch to a completely different implementation. The use of publish/subscribe makes the task of communicating between components independent of the components themselves. To enable this, any candidate component must conform to the defined inter-component messaging protocol. In other words, the component must respond to specified messages with specific behaviour.

The CB3A architecture's flexibility was validated during the development of the autonomy software. The sequencer component underwent many changes during the research process. The first implementation of the sequencer was implemented using the Task Description Language (TDL) by Reid Simmons [33]. However, trying to implement the autonomy with TDL led to the discovery of issues which led to the requirement of a different approach. These limitations and the solution, the SBED sequencer, are discussed in Chapter 6. When the decision was made to change from using TDL to using the SBED sequencer, only the sequencer component needed to be

changed due to the flexibility of the simulation architecture. All of the other simulation components remained the same, and the simulation performed as expected.

The goal for portable software is to enable it to run on as many platforms as possible. The CB3A architecture accomplishes this by using component wrapper classes. The wrapper classes decouple the middleware implementation from the component, which allows for easier migration to other middleware.

There are several advantages in being able to change middleware implementations as required. At the software development stage, there may be the desire to run on a specific middleware. One example is in porting the software component into an HLA-compliant simulation in which case an HLA wrapper can be used.

Portability can also be valuable for the deployment of the autonomy software to the target UAV. At this stage, middleware selection may be restricted due to licensing or performance considerations. Performance testing could indicate that a particular middleware is better suited for the chosen hardware configuration. In such a circumstance, the autonomy software component can be deployed onto the selected middleware by changing the wrapper class implementations.

The CB3A architecture's portability was validated by changing the middleware implementation for two of the components of the CB3A architecture in the UAVSim Testbed. These components were the *X-Plane* flight simulator and the *Autopilot*. The simulated *Autopilot* was designed with the ability to accept text commands from the command line interface to control the aircraft so porting these two components to a different middleware was sufficient to run a simulation. These components had their wrapper classes changed to operate with the MÄK Run-Time Infrastructure (RTI)

middleware [45].[1] MÄK Technologies provides a trial version or their RTI that can be used in federations of up to two federates.[2] For reference, Appendix B contains code examples that highlight the differences between the *CMU IPC Wrapper* and *MÄK RTI Wrapper* class methods. The transmission of messages between components using the CMU IPC middleware is very different from those using the MÄK RTI. In CMU IPC a message can be transmitted as a string so sending a message is simply a matter of calling its publish method (*IPC_publishData*) with the message type and message. Using the MÄK RTI is a more involved process. Each message type is defined as an object with the fields of the message as its attributes. Furthermore, all of these message objects need to be defined in a Federation Object Model (FOM).[3] Figure 5.6 shows an example how a *GpsSensor* message object is defined in the FOM.

When the *GpsSensor* issues an update message, the *MÄK RTI Wrapper* publish method updates the *GpsSensor* object attributes and calls the update method of the *RTIAmbassador*.[4] Despite the fact that the wrapper classes and middleware implementations operate very different in how the pass messages, the underlying components operate the same. The only change made with the components is specifying which middleware wrapper to use.

[1] The RTI is a middleware for the High Level Architecture (HLA), a simulation framework developed by the United States Department of Defense [46].
[2] These terms relate to the HLA. A federate is an HLA-compliant simulation component and a federation is a simulation consisting of multiple federates.
[3] The FOM is used in the HLA to describe the objects that are shared amongst federates in the federation.
[4] The RTIAmbassador acts as an interface to the RTI for a federate.

```
<?xml version="1.0"?>
<!DOCTYPE objectModel SYSTEM "HLA.dtd">
<objectModel
    DTDversion="1516.2"
    name="UAVSimTestbed"
    type="FOM"
    version="1.1"
    date="2008/3/24"
    purpose="UAVSim FOM"
    sponsor="Bassett"
    <objects>
        <objectClass name="HLAobjectRoot">
            <attribute name="HLAprivilegeToDeleteObject"/>
            <objectClass name="GPSSENSOR">
                <attribute name="Longitude"/>
                <attribute name="Latitude"/>
                <attribute name="Altitude"/>
                <attribute name="Heading"/>
                <attribute name="GroundSpeed"/>
                <attribute name="Track"/>
            </objectClass>
        </objectClass>
    </objects>
</objectModel>
```

Figure 5.6: GpsSensor object defined in the FOM

Chapter 6

The SBED Sequencer

This chapter presents the SBED sequencer, an integral part of the CB3A architecture that is responsible for controlling all autonomous behaviour. SBED improves on previous sequencers by centralizing control to deal with competing tasks. Furthermore, SBED uses UML 2, a standardized modeling language specify autonomous behaviour.

6.1 SBED Overview

SBED is a state-based event-driven sequencer. Autonomous behaviour is modelled as predefined responses to events depending on what state the UAV is in. Previous approaches to sequencing employ a task-oriented approach that models behaviour as a series of independent tasks that must be completed as part of the autonomous system's mission. In a deliberative hybrid architecture, control of the system resides at the Deliberative layer. Planning algorithms generate dynamic plans which call on tasks at the Sequencer layer. In a reactive hybrid architecture, control of the system resides at the Sequencer layer, which is responsible for completing predefined tasks. A task functions by creating other subtasks or instructing other subsystems to carry out actions. Tasks operate independently of one another, and if two or more tasks try to command the same

subsystem, conflicts may occur. For example, if two tasks need to control the autopilot, they may send it conflicting commands.

The problem with task-oriented sequencing was identified during initial research when employing a sequencer using the Task Description Language [33]. In the behaviour specification, one task was responsible for navigating the UAV along survey legs by commanding the autopilot where to fly. A second task, responsible for avoiding obstacles, would wait for an obstacle to be detected and if one was discovered, it would send the autopilot commands to fly around it. A conflict would occur if an obstacle was detected close to the end of a survey leg. In avoiding an obstacle, the UAV could fly close enough to the leg's end waypoint that the navigation task would instruct the autopilot to turn towards the next leg, interfering with the obstacle avoidance task. Prioritizing tasks was not a viable solution as there were times during the mission that task priority needed to change.

The SBED sequencer solves the problem of competing tasks by embodying all behaviour within a single central state machine. This is used to ensure that the appropriate command is based on the current mission state of the UAV. The benefits of the approach were validated as the problems encountered with the TDL-based sequencer were eliminated when the SBED sequencer was employed. To mitigate the potential complexity that could arise from specifying all behaviour in a single state machine, a UML 2 Behavioural State Machine is used as it contains syntactic elements that can be used to reduce model complexity. The use of a standardized language to define behaviour has other potential advantages. COTS software modelling tools can be used to design and test UML 2 behavioural state machines.

6.2 SBED in the CB3A Architecture

The SBED sequencer resides as a component on the Sequencer layer within CB3A. As with other components in CB3A, SBED uses publish/subscribe messaging to interact with other subsystems. SBED is notified of events that drive its behaviour by subscribing to messages from other components. SBED directs other components in CB3A to perform actions or ongoing activities by publishing command messages. Actions may also be performed internally in SBED but only if they can be completed within the temporal response time required at the Sequencer layer. The following summary of design guidelines applies when specifying autonomous behaviour in the SBED sequencer:

- incoming messages inform SBED of events that are used to trigger state transitions;
- actions may be internal or external to the sequencer;
- ongoing activities may only be external to the sequencer;
- the completion of all internal actions must complete within the Sequencer layer's temporal response time;
- external actions and activities are accomplished by publishing command messages to other components in the CB3A architecture;

6.3 Evaluating the SBED Sequencer

In order to verify the effectiveness of the SBED sequencer, it was used to control autonomous behaviour for a prototypal mission of the GeoSurv II UAV in the UAVSim Testbed. The specification of the autonomous behaviour using a UML 2 behavioural state machine is shown in Figure 6.1. An overview of the syntactic elements is covered in Section 2.4.

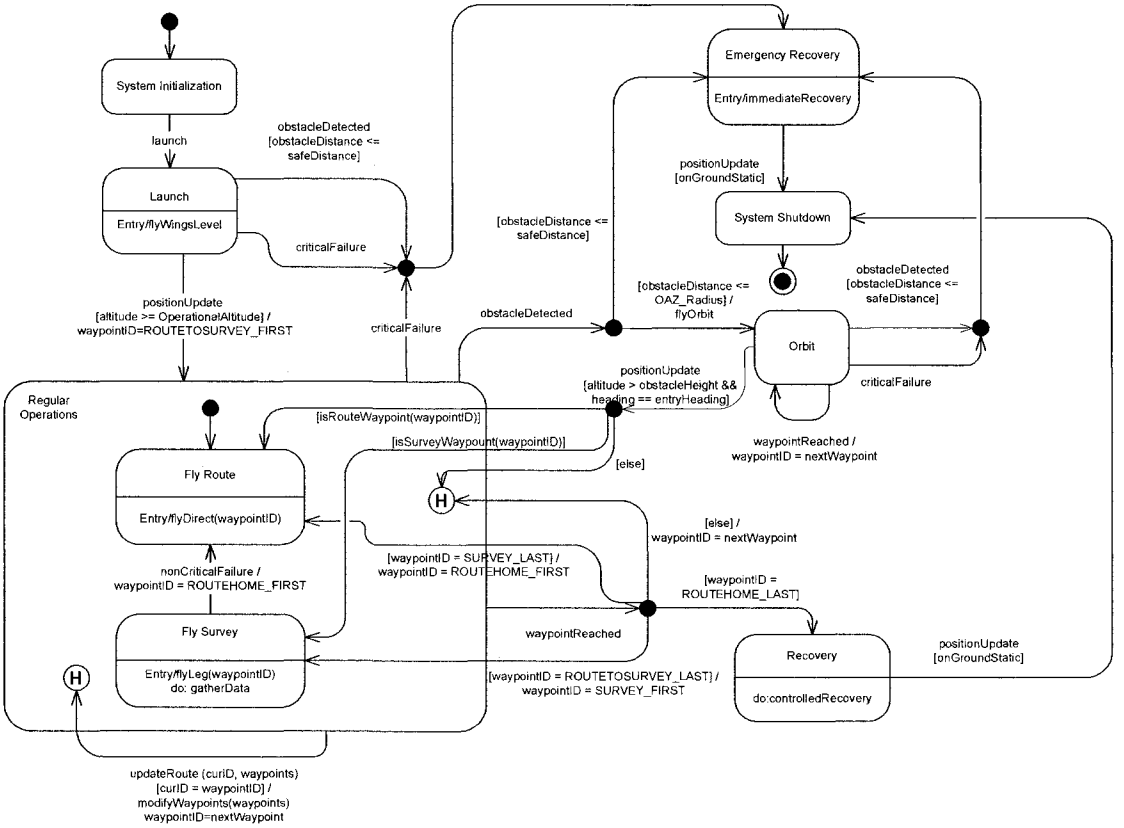

Figure 6.1: GeoSurv II Prototypal Mission Autonomous Behaviour

Each of the states in the autonomous behaviour specification is described below:

***System Initialization*:** This initial state is entered upon the UAV being powered up. In this state the UAV initializes and calibrates its sensors. It also performs a series of self tests to ensure all systems are operating properly.

Launch: The UAV enters this state upon being launched. During the initial climb, the UAV maintains a wings level attitude[5] until it reaches the minimum altitude required for safe manoeuvring.

Regular Operations: This state is a composite state that the UAV is in while performing regular flight. It includes the two sub-states, *Fly Route* and *Fly Survey*, described below.

Fly Route: In this state the UAV follows a flight plan of waypoints to reach its destination. The flight plan includes the route to the survey and the route back home to the recovery point. All obstacle detection sensors are active however no survey sensors are active in this state. The route planner can update the route by inserting waypoints if required to avoid obstacles.

[5] Wings level refers to a mode in the autopilot where it holds the current heading. This is used to ensure that the UAV does not perform any aggressive maneuvers while it is dangerously close to the ground during the launch.

Fly Survey: Within this state, the aircraft performs an aerial survey by flying a route prescribed by waypoints. While in this state the *gatherData* activity takes place which consists of recording geophysical information. The aircraft must fly at a low altitude to gather the survey data, so it is during this state that the greatest risk from obstacles on the ground exists.

Orbit: Upon entering this state, the aircraft enters the orbit manoeuvre by issuing the autopilot the *FlyOrbit* command. This state is a critical part of the obstacle avoidance strategy discussed in Chapter 7, and acts as a fallback mechanism if the route planner can not safely determine a route around obstacles.

Recovery: Upon entering this state, the UAV begins a controlled recovery. It must fly past the desired landing zone to estimate wind speed and direction. The aircraft will then make another pass to a desired location for carrying out the recovery. At the desired location, the UAV will shut down its engine and deploy the parachute to carry out a descent under canopy.

Emergency Recovery: This state is entered only when a critical system failure has occurred that precludes the ability to fly to a known safe zone to carry out a recovery. The aircraft will carry out an immediate recovery by shutting down its engine and deploying the parachute. The intention is to reduce damage by having it descend slowly to the earth.

System Shutdown: This is the final state that upon being entered manages the shutdown of all of the UAV systems.

As part of the mission specification, mission parameters such as survey location, survey legs, and routes to and from the survey are specified in a flight plan of waypoints. Each waypoint consists of an identifier, a destination latitude and longitude, altitude and speed. Survey leg waypoints also include a starting latitude and longitude, so that the leg becomes a straight track between the start and end points. The UAV will fly at the designated altitude and speed en route to the waypoint. Changing a mission involves changing the set of waypoints that comprise the mission flight plan.

The SBED sequencer handles competing mission tasks by having all behaviour explicitly stated in one central state machine. For example, in the GeoSurv II prototypal mission, flying a survey leg and performing the orbit manoeuvre to avoid obstacles are competing goals since one goal tries to stay as close to the track as possible to collect survey data, while the other will depart the track to avoid obstacles. SBED handles these competing tasks through its state-based approach. The UAV will either be in the *Fly Survey* state or the *Orbit* state and its behaviour is explicitly defined depending on what state it is in. If while in the *Orbit* state, the *waypointReached* message is received, SBED updates the flight plan, but continues doing the orbit manoeuvre. Once the obstacle is safely avoided, SBED instructs the autopilot to carry on flying to the next waypoint.

Figure 6.2 shows an overhead view of a scenario executed in the UAVSim Testbed. It depicts the UAV flying a leg in a survey that contains three sets of obstacles. The dotted line represents the desired track. As obstacles are detected, the route planner

uses the *updateRoute* message to insert waypoints to avoid the obstacles as specified in the behavioural state machine. As shown in the overhead view, SBED uses waypoints from the route planner to avoid the first two sets of obstacles. Upon avoiding the second set of obstacles, SBED directs the UAV to fly back on the initial track. In this case, the route planner does not respond with a path around the obstacles fast enough and SBED transitions to the *Orbit* state and commands the autopilot to fly the orbit manoeuvre (the orbit manoeuvre is covered in detail in Chapter 7). Once the UAV gains sufficient altitude in the orbit manoeuvre to fly over the obstacles, SBED switches back to the *Fly Survey* state which resumes the survey track.

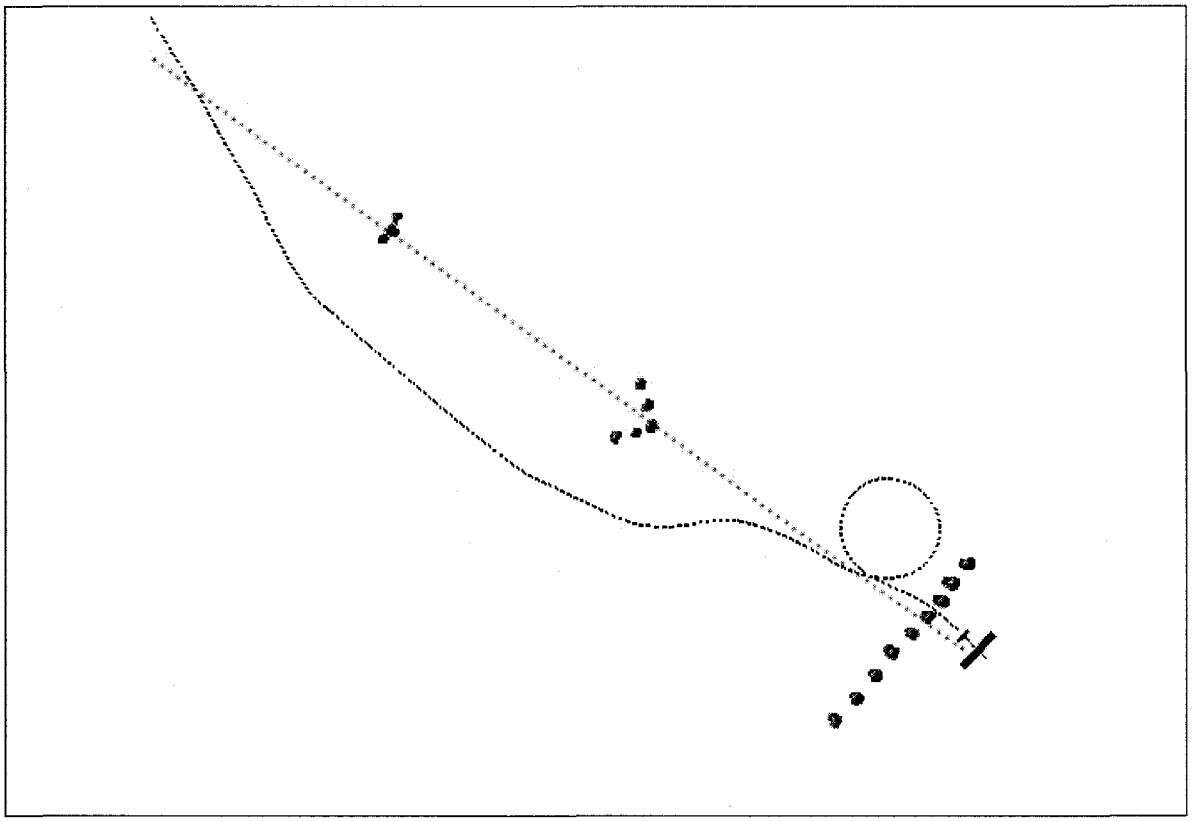

Figure 6.2: SBED controlling competing tasks

Chapter 7

Enabling Obstacle Avoidance in CB3A

The ability to avoid obstacles is a critical factor for an autonomous UAV flying at close to the ground. The thesis focuses on static obstacles which include buildings, towers, trees, and power lines. This chapter discusses how obstacle avoidance for static obstacles is achieved within the CB3A architecture and SBED sequencer.

7.1 Static Obstacle Avoidance Overview

The sequence of events in an obstacle avoidance scenario can be divided into three steps: Firstly, detecting an obstacle, secondly, determining a safe route around the obstacle, and thirdly, controlling the aircraft to fly the selected route. Sensing an obstacle is reliant on obstacle detection sensors such as scanning lasers and video cameras mounted on the aircraft. The information from these sensors must be processed to determine whether an obstacle has been detected in which case a message with information about the obstacle is sent to an obstacle avoidance algorithm. The obstacle avoidance algorithm considers this information to determine if the current flight path of the aircraft must be altered. If a change in route is required, the aircraft's autopilot must be directed accordingly.

The level of success that a UAV will have in avoiding an obstacle is limited by its knowledge of the obstacle's presence. In general, the presence of an obstacle can be determined from previous knowledge or by in-flight detection.

Previous knowledge of an obstacle's location can come from a map on which the obstacle's position is recorded. The information on the map can be loaded before the mission begins or can be dynamically updated during the mission. Updates can be made by marking the position of an obstacle that has been detected, or by receiving communications from another UAV or the ground station.

In-flight obstacle detection can be achieved by a variety of obstacle detection sensors with have their own strengths and weaknesses. Machine vision detection relies on cameras that are mounted in a stereoscopic configuration so that the differences in images can be compared to determine the obstacle's range using trigonometry. This process attempts to mimic the natural depth range abilities of animals with stereoscopic vision. Due to the detection being dependent on image quality, performance can be limited due to the effects of weather or glare. Also, the type of terrain plays a factor – if the obstacles do not contrast with their surroundings they will be much harder to detect. Scanning lasers have shorter detection ranges but have more accurate distance measurement and high reliability rates that are less affected by environmental factors. Scanning lasers now exist that are as small as 15 pounds are that are able to detect obstacles with 99.5 percent certainty up to 2 kilometres away [47].

To enable obstacle detection within the CB3A architecture a *Route Planner* and *Obstacle Detector* component have been employed.

The *Route Planner* resides on the Deliberator layer of the architecture and actively compares the current trajectory of the aircraft with an internal obstacle database. If it determines that the UAV is at risk of encountering an obstacle, it suggests modifications to the route being flown to avoid the obstacle. The suggestion is sent to the SBED sequencer, which ultimately decides whether or not to modify the route. This ensures that SBED remains the sole controller of all autonomous behaviour. Depending on the time it takes for the *Route Planner* to generate a new route, the suggestions may no longer be relevant if SBED has already taken other actions.

One or more *Obstacle Detector* components use data from the aircraft's obstacle detection sensors to detect unmapped obstacles that present a potential threat. Upon detecting an obstacle, they publish an o*bstacleDetected* message that is received by the *SBED Sequencer* and *Route Planner* components.

The route flown by the aircraft is specified by using waypoints. To enable the *Route Planner* to make temporary alterations to the route without changing the underlying flight plan, different waypoint types are used. A *permanent* waypoint is used to identify the route to be followed as specified in the flight plan. A *temporary* waypoint may be inserted by the *Route Planner* in order to alter the current flight path to fly over or around obstacles.

7.2 Route Planning for Static Obstacle Avoidance

The *Route Planner* component must subscribe to a number of messages to function properly. Messages from *Obstacle Detector* components allow it to update its internal database of obstacle locations. Flight commands from the *SBED sequencer* component keep it informed about which waypoint the UAV is currently flying towards. Update messages from the *Autopilot* component keep it informed on the current position of the UAV. Whenever the *Route Planner* receives an update on the UAV's current position it performs the following algorithm:

a) Plot a line from the current position to the current destination waypoint.
b) If there are any obstacles along the route, but their heights are low enough that the aircraft has time to climb higher to fly over all of them, determine a new intermediate waypoint to fly at a higher altitude over the obstacles.
c) If any of the obstacles' heights necessitate flying around them, use an obstacle avoidance planning algorithm to plot a route around the obstacle to determine a set of intermediate waypoints that avoids the obstacles.

For evaluation purposes, the modified Rapidly-Exploring Random Tree (RRT) [41] algorithm was used as an obstacle avoidance planning algorithm in the *Route Planner* component. If the *Route Planner* determines the need for intermediate waypoints it publishes an *updateRoute* message for the *SBED sequencer*. The message includes the

set of waypoints and the identifier of the waypoint that is currently flown by the UAV. This acts as a mechanism for SBED to confirm that the new waypoints are coherent with its current intentions. If the waypoint identifier does not match the one it currently flying towards, then SBED will disregard the suggested waypoint insertions.

7.3 Reactive Static Obstacle Avoidance

One of the characteristics of components on the Deliberator layer is that they do not have a guaranteed response time. Therefore, if an obstacle is not detected until the UAV is very close to it, there is no guarantee that the *Route Planner* component will be able to generate a plan to avoid it in an adequate amount of time. In other autonomous vehicle applications, the system's movement can be halted to wait until the planner generates a response; however this is not an option with a fixed-wing UAV, which necessitates a new fallback mechanism, the orbit manoeuvre.

7.3.1 The Orbit Manoeuvre

The principle behind the orbit manoeuvre is to keep the UAV away from static obstacles at a minimum distance such that it always has enough room to turn around safely. Figure 7.1 defines two zones of interest to the orbit manoeuvre. The first zone, called the Obstacle Free Zone (OFZ) is the area around the aircraft which must be kept free of obstacles. Section 7.3.2 defines the required radius of this zone (R_{OFZ}) which is equal to two times the current minimum turning radius of the UAV. In order to ensure that the

OFZ remains obstacle-free, the UAV must employ an obstacle detection sensor which can positively identify obstacles before they enter the OFZ. To deal with the delay between the time it takes to detect an obstacle and the time it takes to generate a response, another zone is defined that includes a safety margin that extends beyond the OFZ. This zone, also shown in Figure 7.1, is called the obstacle avoidance zone (OAZ) and has a radius R_{OAZ} which is equal to R_{OFZ} plus the required safety margin. The size of the safety margin is dependent upon the response time of the system which is dependent on system performance and the UAV's current ground speed.

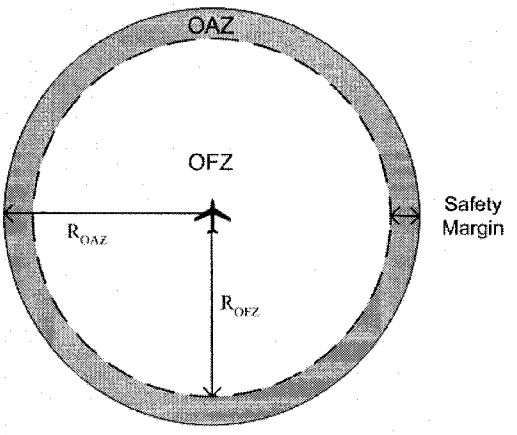

Figure 7.1: OFZ and OAZ defined

7.3.2 Defining the Size of the OFZ

The reasoning behind the required size of the obstacle-free zone (OFZ) is examined in this section. Figure 7.2 shows a UAV flying on a straight flight path. The dashed line represents the OFZ which has a radius of two times the minimum turning radius of the UAV. The OAZ safety margin is represented by the shaded region outside the OFZ and its dimensions account for the distance the aircraft can travel between the time an obstacle is detected and the time the orbit manoeuvre is activated.

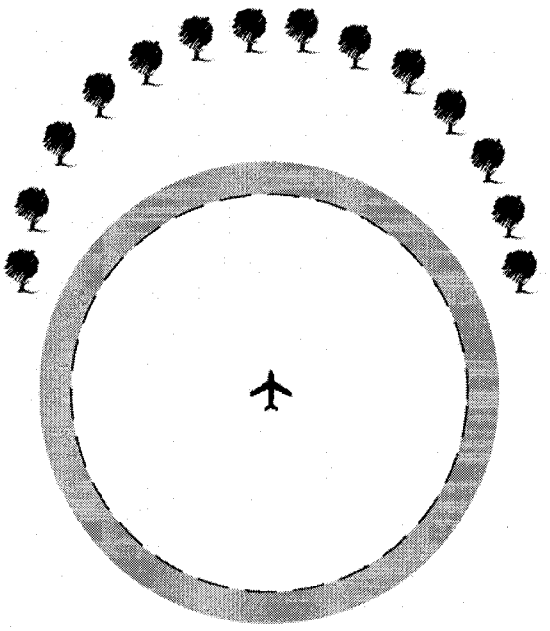

Figure 7.2: UAV flying in a straight path towards a group of static obstacles

Figure 7.3 shows the UAV reaching a point where an obstacle has just entered the OAZ. The *SBED Sequencer* determines that the obstacle is in the OAZ, and immediately transitions to the *Orbit* state. Upon entering the *Orbit* state, the *SBED Sequencer* initiates the orbit manoeuvre.

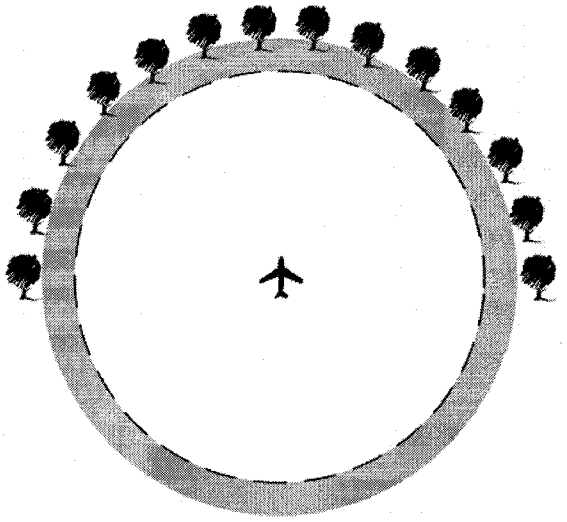

Figure 7.3: An obstacle has penetrated the obstacle avoidance zone

Figure 7.4 shows the path that the aircraft will take while executing the orbit manoeuvre. If the aircraft turns at its minimum turning radius, then the diameter of the orbit will be equal to two times its minimum turning radius. As long as this region, which lies in the OFZ, has no obstacles in it, then the UAV will be safe. Therefore the UAV must enter the orbit manoeuvre prior to any obstacles entering the OFZ. The UAV climbs higher while in the orbit manoeuvre until it reaches an altitude greater than the height of the obstacle, at which point it can proceed safely in the original direction of flight.

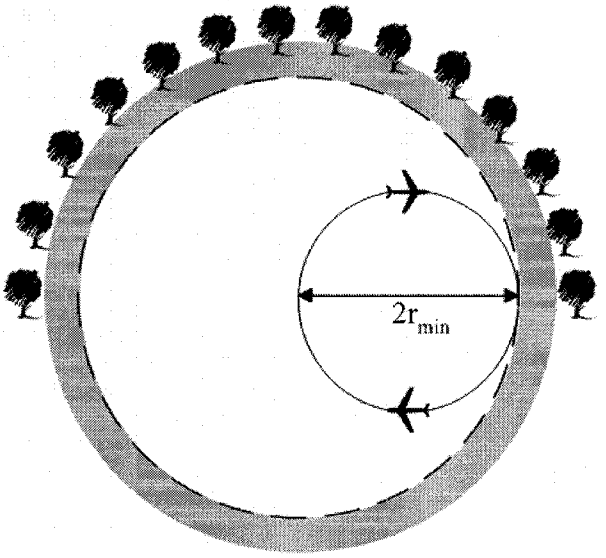

Figure 7.4: Flight path of the Orbit Manoeuvre

The orbit manoeuvre is based on a conservative approach to obstacle avoidance due to the challenges of fixed-wing UAV obstacle avoidance and the limited range of obstacle detection. Figure 7.5 illustrates an example of one of the potential manifestations due to limited range of obstacle detection that the orbit manoeuvre addresses. In Figure 7.5.a (a), if the UAV does not employ the orbit manoeuvre, there is a chance that it may choose to proceed through the gap in the obstacles as the way appears clear. However, once the UAV gets closer to the gap, as shown in Figure 7.5 (b) a new obstacle may be detected that blocks the way through the gap. By this time even if the UAV were to turn away in either direction, at best it could only follow one of the routes shown on the overlain circles and would not avoid hitting an obstacle. By employing the orbit manoeuvre, the UAV avoids this situation by entering the orbit manoeuvre to climb before proceeding ahead when any of the obstacles enter the OAZ.

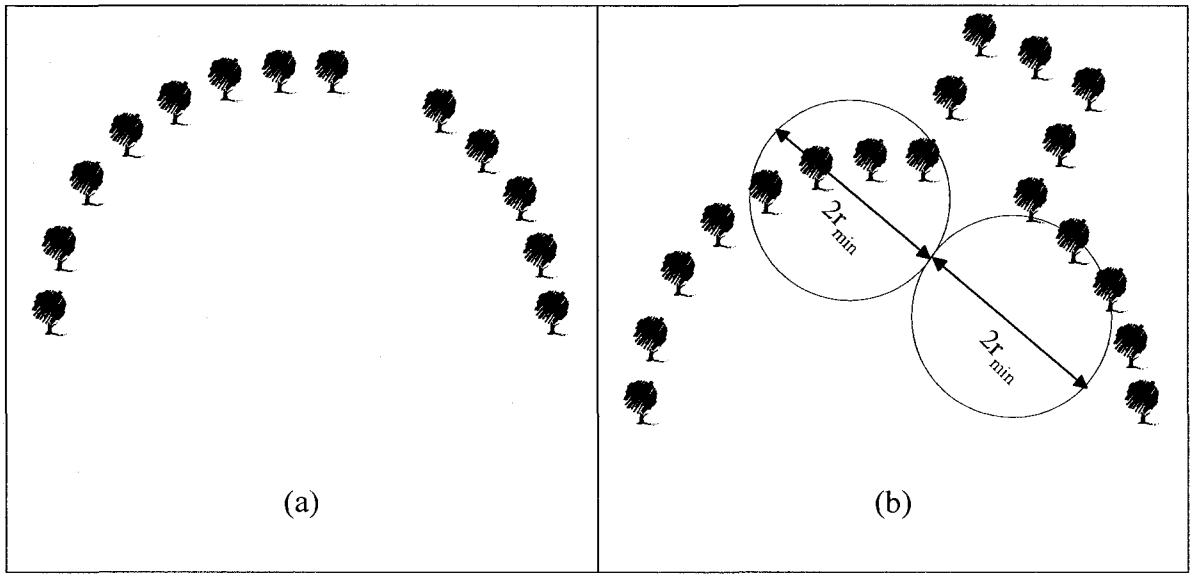

Figure 7.5: Problems due to limited range of obstacle detection

It can be shown that it is sufficient to have an obstacle detection sensor that can cover the 180 degree semi-circular area to the front of the UAV to ensure that the OFZ remains free from static obstacles. The semi-circular area of the OFZ is known to be obstacle-free because it lies in the region covered by the obstacle detection sensor. However, it needs to be shown that the area behind the UAV will remain clear of obstacles no matter what flight path the UAV takes. Figure 7.6 (a), shows the procession of the OFZ as the UAV flies in a straight line. The shaded area shows the coverage achieved by the obstacle detection sensor. Due to the flight path of the UAV, no obstacles will be missed and the OFZ is known to be obstacle-free.

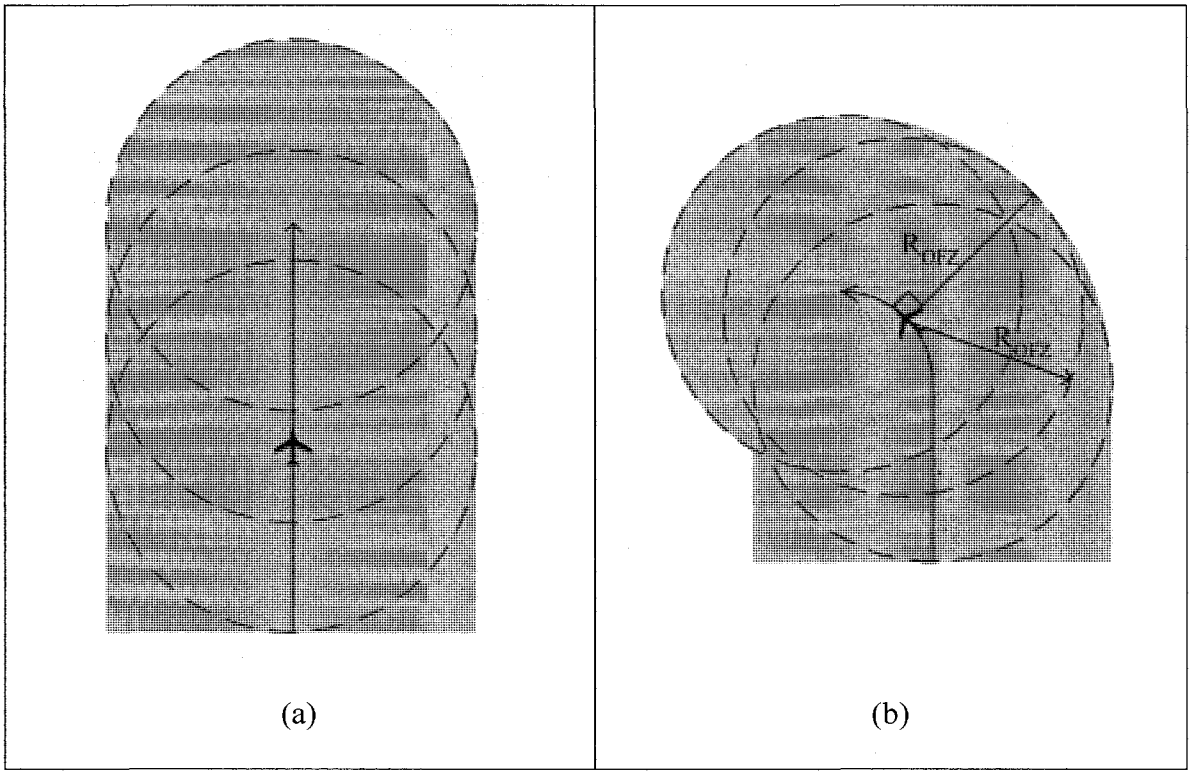

Figure 7.6: Ensuring the OFZ clear

Figure 7.2 (b) shows that even if the UAV turns 90 degrees to the left at its maximum turning radius the OFZ will remain obstacle-free. In order for the OFZ to remain obstacle-free, there must be no point in the OFZ that is unobserved during flight. In other words there must be no point within the radius of the OFZ (R_{OFZ}) from the flight path that has not been observed by the 180 degree forward-looking sensor. The shaded area shows the coverage by the obstacle detections sensor area as the UAV completes the manoeuvre. At any point on the flight path, the distance to the edge of the shaded area along a line perpendicular to the flight path is equal to R_{OFZ}. Therefore, no unobserved point to the rear of the aircraft will ever be less than R_{OFZ} away and the OFZ is known to be free from static obstacles.

7.4 Obstacle Avoidance using the SBED Sequencer

This section outlines how the SBED sequencer is used to define obstacle avoidance as autonomous behaviour. Figure 7.7 details a subset of the UAV's autonomous behaviour concerned with obstacle avoidance.

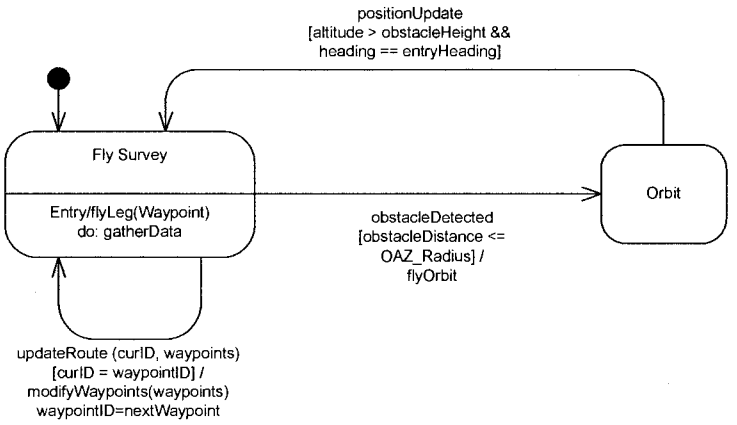

Figure 7.7: Defining obstacle avoidance in the SBED sequencer

The transitions are explained below:

obstacleDetected: The *obstacleDetected* message is sent from the *Obstacle Detector* component when a static obstacle has been detected and includes the obstacle's distance direction, and height. The guard condition checks if the detected obstacle is within the OAZ. If so, *SBED* transitions to the *Orbit* state and publishes the *flyOrbit* command for the *Autopilot*.

positionUpdate: The *positionUpdate* message is received periodically from the *Autopilot* and includes the UAV's current altitude and heading. Once the UAV's altitude exceeds that of the obstacle and the heading matches the entry heading of the orbit manoeuvre, *SBED* transitions back to the *Fly Survey* state. Upon entering the *Fly Survey* state, *SBED* publishes the *flyLeg* command to the *Autopilot* to return to the previous course.

updateRoute: The *updateRoute* message is sent by the *Route Planner* if it wants to add temporary waypoints to the route as part of an obstacle avoidance plan. *SBED* confirms that the updated route is coherent with its current destination by comparing the *Route Planner's* intended destination (*curID*) with the actual destination (*waypointID).* If they match, *SBED* adds the temporary waypoints, re-enters the *Fly Survey* state, and publishes the *flyLeg* command for the autopilot to fly to the temporary waypoint.

Scenarios executed in the UAVSim Testbed were used to confirm that the SBED sequencer as part of the CB3A architecture could effectively control obstacle avoidance. Figure 7.8 shows an example of a survey leg being flown while reacting to three groups of obstacles. The dotted line represents the desired survey track. As seen in Figure 7.8 the UAV successfully planned a route around the obstacles and then returned to the desired track after reaching the end of them.

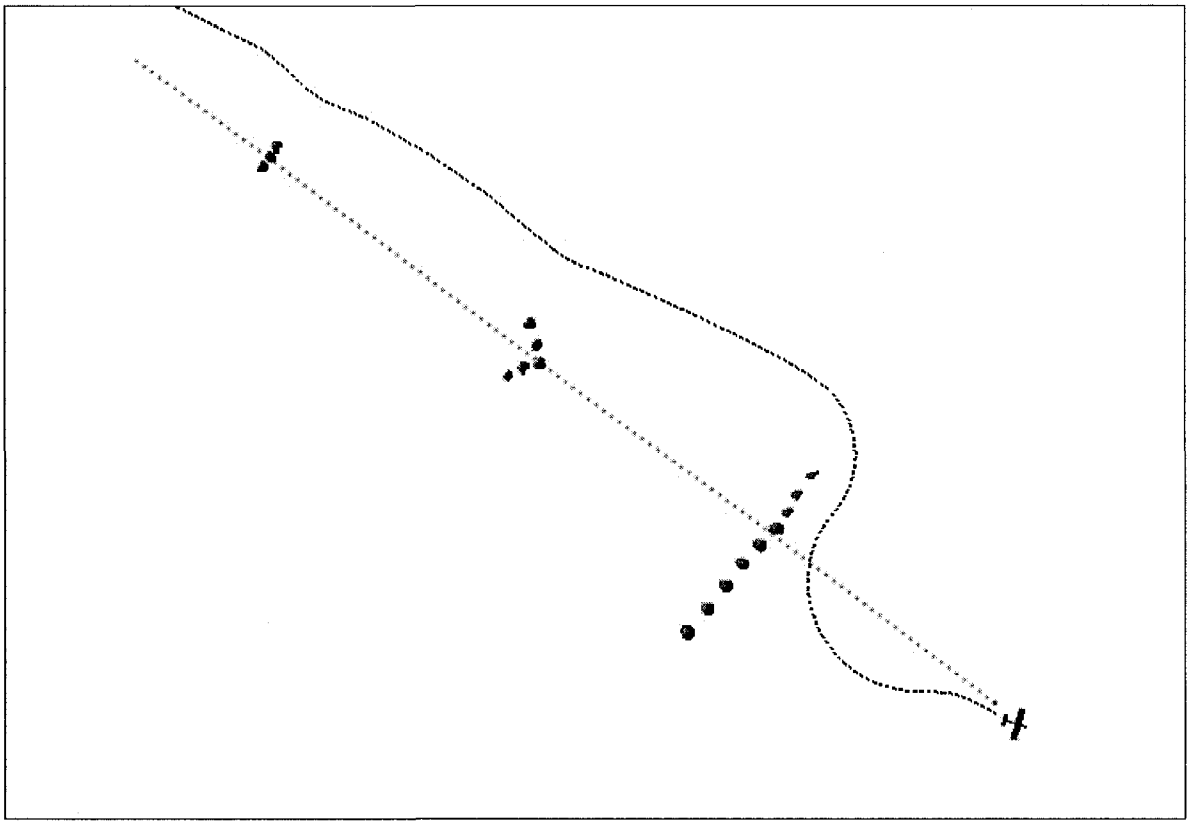

Figure 7.8: UAV avoiding three groups of obstacles on a survey leg

The UAVSim Testbed was also used to demonstrate how the orbit manoeuvre acts as a fallback mechanism for obstacle avoidance. In Figure 7.9, the UAV was flown towards a set of obstacles in an inverted-V formation with the orbit manoeuvre disabled. The UAV initially determines that there is a clear route between the obstacles at the tips of the V, because the obstacles beyond the tips cannot yet be detected. As the UAV proceeds ahead, it detects the second row of obstacles but because there is still enough space to fly between them, it proceeds ahead. Eventually, the UAV detects the obstacles that form the bottom of the V, but by this time it is too late to turn around and the UAV cannot avoid crashing.

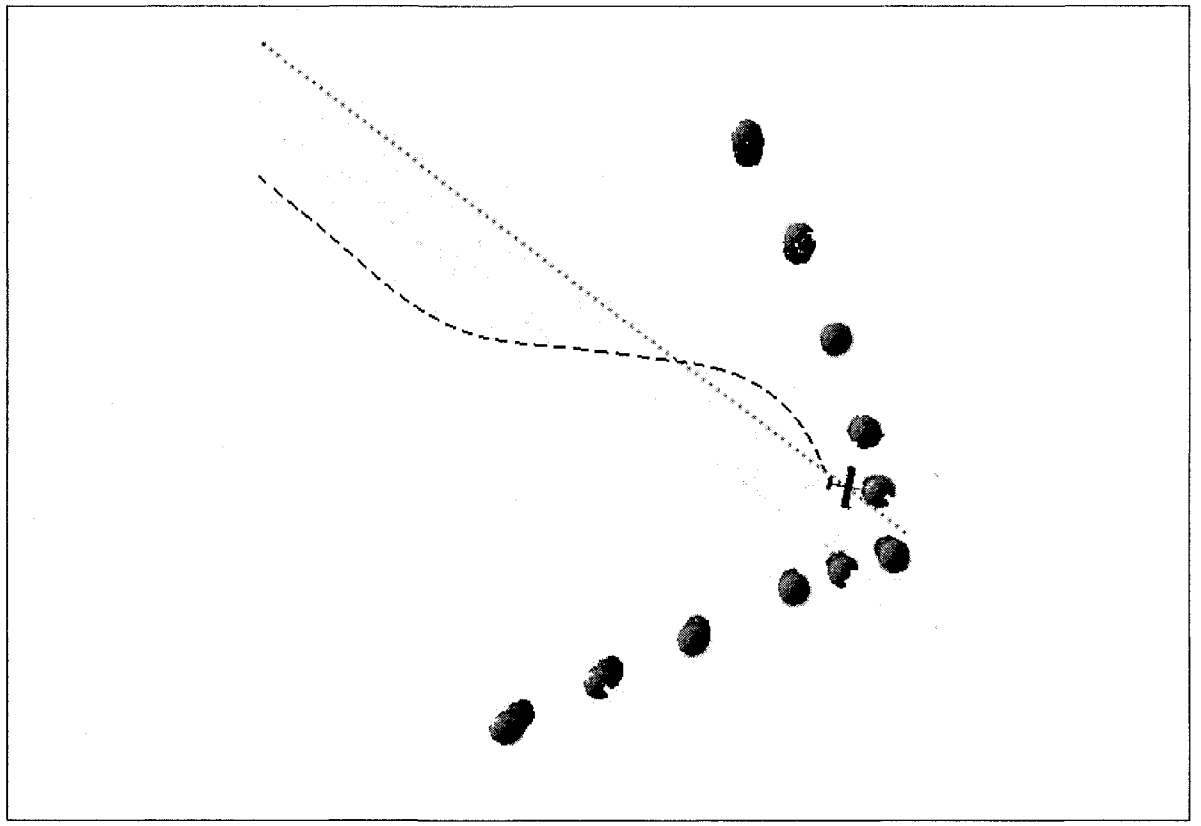

Figure 7.9: UAV with orbit manoeuvre disabled crashing into an inverted-V obstacle formation

Figure 7.10 shows how the UAV handles the same scenario when the orbit manoeuvre is enabled. At first the UAV determines that it will fly between the tips of the obstacles as before. However, once the obstacle avoidance zone (OAZ) is penetrated, SBED activates the orbit manoeuvre. The UAV climbs in the orbit manoeuvre until it is higher than the obstacles and then proceeds ahead on the original track.

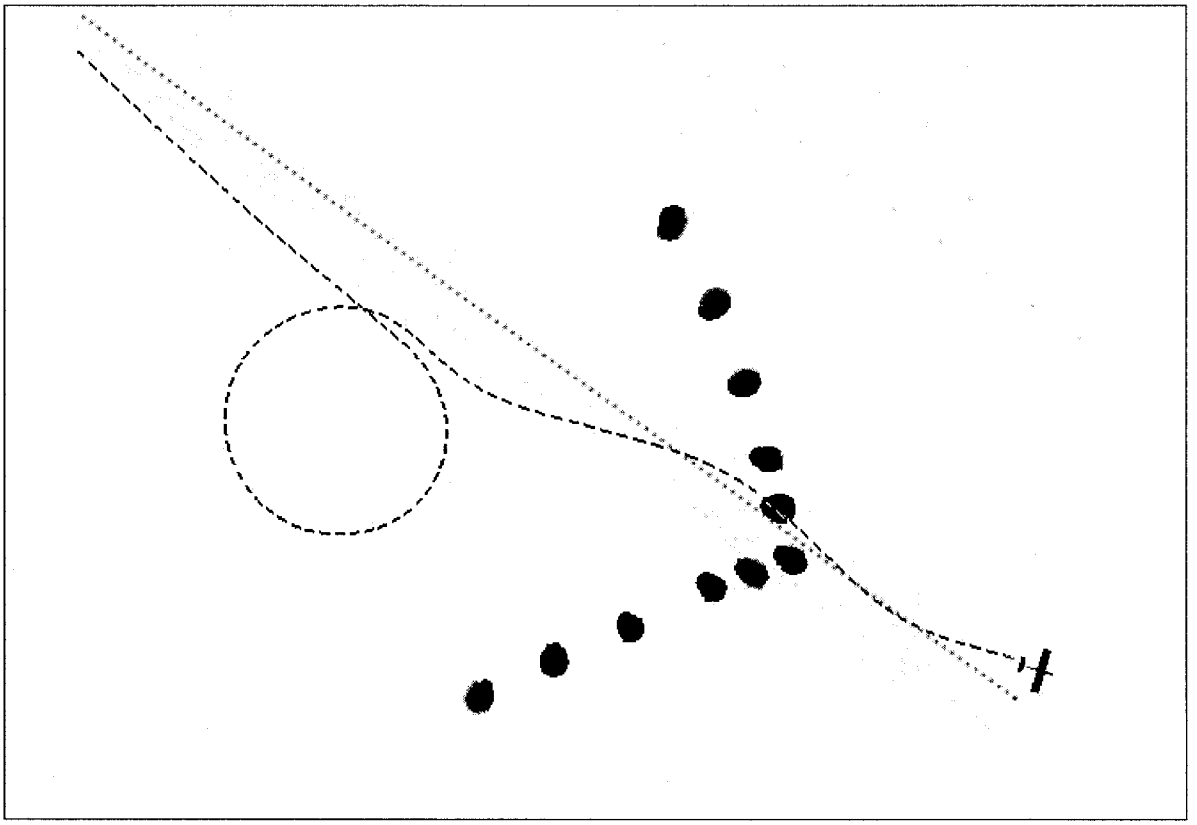

Figure 7.10: UAV with orbit manoeuvre enabled flying over the inverted-V obstacle formation

Chapter 8

Conclusions and Future Work

The claims in the thesis statement were validated by implementing the CB3A architecture and SBED sequencer in the UAVSim Testbed, a simulation-based environment. This chapter summarizes conclusions made about the research, outlines the contributions, and presents suggestions for future work for research in autonomous UAV development.

8.1 Conclusions

The following conclusions were made as a result of this work:

1. A novel reactive-hybrid approach to the generic three-layer architecture for fixed-wing UAV autonomy can be organized around a central sequencer layer that resolves control conflicts from competing sources. As described in Section 5.2, the CB3A architecture embodies a set of UAV subsystems and was demonstrated to successfully execute a prototypical mission in a simulated environment.
2. The SBED sequencer realizes the sequencer layer in the CB3A architecture, and provides the effective resolution of competing autonomous behaviours. As shown in Section 6.3, the SBED sequencer was able to resolve conflicts in plans suggested by the survey path planner and the obstacle avoidance path planner.

3. The orbit manoeuvre can be integrated into the sequencer to provide a safe fallback technique for fixed-wing UAVs to avoid obstacles. As described in Section 7.4, the orbit manoeuvre was integrated into the SBED sequencer and was shown to successfully avoid collisions with obstacles in a simulation environment.
4. Using a UML 2 behavioural state machine to specify autonomous behaviour in the SBED sequencer is an effective means of specifying autonomous behaviour. As shown in Sections 6.3, the sequencing of autonomous behaviour for a prototypical mission of the GeoSurv II was specified as a UML 2 behavioural state machine and then demonstrated to successfully execute the mission in a simulation environment.
5. The CB3A architecture exhibits flexibility and portability, which are both favourable system architecture traits. As demonstrated in Section 5.3, the sequencer component was replaced with an alternate and two of the components were ported to a different middleware.
6. The UAVSim Testbed will be useful as a simulation environment for further research on the GeoSurv II project. It was used to develop and evaluate the work for this thesis. The simulated components form the basis for a library of simulation components to be used and expanded upon for future work.

8.2 Contributions

The thesis makes the following contributions to knowledge:

1. Demonstration that the CB3A architecture can be used effectively to direct and control a fixed-wing UAV autonomously as described in Chapter 5.

2. Demonstration that the SBED sequencer in the CB3A architecture can resolve conflicts between competing autonomous behaviours using control specified in a UML 2 Behavioural State Machine as described in Chapter 6.

3. Demonstration that an orbit manoeuvre provides a safe fallback technique for obstacle avoidance in an autonomous fixed-wing UAV as described in Chapter 7.

8.3 Future Work

There are a number of suggestions made in the thesis regarding future work. These suggestions are summarized below:

- Research the ability of the CB3A architecture to support simulation-driven development in the UAVSim Testbed by using it as the simulation and target architecture.

- Replace simulated components in the CB3A architecture with actual hardware components.
- Develop performance metrics to analyse the performance of the CB3A to determine target hardware requirements.
- Investigate the integration of obstacle avoidance algorithms into the CB3A architecture to deal with moving obstacles.
- Continued research on improving methods of static and obstacle detection.

References

[1] H. Stocker. Autonomous Intelligent Systems: Opportunities and Needs for the CF/DND. Technical Memorandum, Defence R & D Canada, July 2003.

[2] D. W. Casbeer, R. W. Beard, T. W. McLain, L. Sai-Ming, and R. K. Mehra. Forest Fire Monitoring With Multiple Small UAVs. In *Proceedings of the American Control Conference, 2005,* pages 3530-3535, Provo, UT, USA, June 2005.

[3] Z. Sarris. Survey of UAV Applications in Civil Markets. In *Proceedings of the 9th Mediterranean Conference on Control and Automation*, Dubrovnik, Croatia, June 2001.

[4] MicroPilot. MP2028g Miniature UAV Autopilot. Viewed 20 July 2008, http://www.micropilot.com/PDF%20Files/broch_MP2028g.pdf/

[5] Procerus Technologies. Kestrel Autopilot System. Viewed 20 July 2008, http://www.procerusuav.com/Downloads/DataSheets/Kestrel_2.2x.pdf

[6] N. J. Nilsson. *Principles of Artificial Intelligence*, Morgan Kaufmann, San Francisco, CA, USA 1980.

[7] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal on Robotics and Automation.* Vol RA-2, no. 1. March 1986.

[8] E. Gat. On Three-Layer Architectures. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195-210, MIT Press, Cambridge, MA, 1998.

[9] U.S. Government Accountability Office. Unmanned Aircraft Systems: Federal Actions Needed to Ensure Safety and Expand Their Potential Uses Within the National Airspace System. Report to Congressional Requesters, May 2008.

[10] Transport Canada. Unmanned Air Vehicle Working Group. Final Report, September 2007.

[11] G. T. Heineman, and W. T. Councill. *Component-based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[12] Microsoft. COM: Component Object Model Technologies. Viewed 20 July 2008 http://www.microsoft.com/com/default.mspx

[13] Sun Microsystems. Enterprise JavaBeans Technology. Viewed 20 July 2008, http://java.sun.com/products/ejb/

[14] Object Management Group. CORBA/IIOP specifications. Viewed 20 July 2008, http://www.omg.org/technology/documents/corba_spec_catalog.htm

[15] Object Management Group. Real-time CORBA Specification, Version 1.2. Viewed 20 July 2008, http://www.ois.com/images/stories/ois/real-time corba specification 2005-01-04 jan 202005.pdf

[16] E. F. Moore. Gedanken-experiments on Sequential Machines. *Automata Studies (Annals of Mathematical Studies, no 34)*, pages 129-153, Princeton, NJ, USA, 1956.

[17] G. H. Mealy. A Method to Synthesizing Sequential Circuits. *Bell System Technical J*, pages 1045-1079, 1955.

[18] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, pages 231-274, Amsterdam, the Netherlands, June 1987.

[19] B. Meadowcroft. A Review of Statecharts. Viewed 20 July 2008, http://www.benmeadowcroft.com/reports/statechart/

[20] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2.*, November, 2007.

[21] B. P. Douglass. *Real Time UML: Advances in the UML for Real-Time Systems.* Addison-Wesley, Third Edition, 2004.

[22] T. James. *GeoSurv II UAV Systems Requirement Document Revision E*, Carleton University, Ottawa, ON, Canada, March 2008.

[23] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, Vol 9, pages 237-256, Houston, TX, USA, 1997.

[24] I. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. S. Kim, CLARAty: An Architecture for Reusable Robotic Software. In *Proceedings of the SPIE*, Vol 5083, pages 253-264, Orlando, FL, USA, April 2003.

[25] F. Fisher, T. Estlin, D. Gaines, S. Schaffer, C. Chouinard, and R. Knight. CLEaR: Closed Loop Execution and Recovery – A Framework for Unified Planning and Execution. *Technology and Science IND News*, Issue 16, pages 15-20, Pasadena, CA, USA, September 2002.

[26] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. In *Proceedings of the 5TH International Conference on Artificial Intelligence Planning and Scheduling,* Breckenridge, CO, USA, April 2000.

[27] S. Karim, C. Heinze, and S. Dunn. Agent-based Mission Management for a UAV. In *Proceedings of the 2004 Intelligent Sensors, Sensor Networks, and Information Processing Conference*, pages 481-486, Piscataway, NJ, USA, December 2004.

[28] A. S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42-55, Springer Verlag, Eindhoven, the Netherlands, 1996.

[29] E. Gat. Integrating Planning and Reaction in a Heterogeneous Asynchronous Architecture for Controlling Mobile Robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI, San Jose, CA, USA, 1992.

[30] M. Freed, P. Bonasso, K. M. Dalal, W. Fitzgerald, C. Frost, and R. Harris. An Architecture for Intelligent Management of Aerial Observation Missions. In *Proceedings of American Institute of Aeronautics and Astronautics "Infotech@Aerospace" Technical Conference*, Arlington, VA, USA, September 2005.

[31] P. Doherty, P. Haslum, F. Heintz, T. Merz, T. Persson, and B. Wingman. A Distributed Architecture for Intelligent Unmanned Aerial Vehicle Experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, Toulouse, France, 2004.

[32] R. J. Firby. Modularity Issues in Reactive Planning. In *Proceedings of the Third International Conference on AI Planning Systems,* pages 78-85, Edinburgh, Scotland, 1996.

[33] R. Simmons, and D. Apfelbaum. A Task Description Language for Robot Control. In *Proceedings of the Conference on Intelligent Robots and Systems,* Vancouver, BC, Canada, 1998.

[34] E. N. Johnson and D. P. Schrage. The Georgia Tech Unmanned Aerial Research Vehicle: GTMax. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Austin, TX, USA, 2003.

[35] G. Pisanich, L. Plice, C. Neukom, L. Fluckiger, M. Wagner. Mission Simulation Facility: Simulation Support for Autonomy Development. *42nd AIAA Aerospace Sciences Conference*, Reno, NV, USA, January 2004.

[36] M. H. Mansur, M. Frye, B. Mettler, and M. Montegut. Rapid Prototyping and Evaluation of Control System Design for Manned and Unmanned Applications. In *Proceedings of the American Helicopter Society 56th Annual Forum*, Virginia Beach, VA, USA, May 2000.

[37] E.N. Johnson and S. Mishra. Flight Simulation for the Development of an Experimental UAV. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference*, Monterey, CA, USA, 2002.

[38] G. Wainer and E. Glinsky. Model-Based Development of Embedded Systems with RT-CD++. In *Proceedings of the WIP session, IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, ON, Canada, 2004.

[39] T. W. Pearce, Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems. In *Proceedings of RTSS-WIP*, Cancun, Mexico, December 2003.

[40] S. Thurn, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, and P. Stang. Stanley: The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics*. Vol 23(9), pages 661-692, 2006.

[41] J. B. Saunders, B. Call, A. Curtis, R. W. Beard, and T. W. McLain. Static and dynamic obstacle avoidance in miniature air vehicles. In *AIAA Infotech at Aerospace*, paper no. AIAA-2005-6950, Arlington, VA, USA, 2005.

[42] S. Griffiths, J. Saunders, A. Curtis, B. Barber, T. McLain, and R. Beard. Maximizing Miniature Aerial Vehicles. *IEEE Robotics & Automation Magazine*, Vol. 13, pages 34-43, November 2006.

[43] S. M. LaValle, Rapidly-Exploring Random Trees: A New Tool for Path Planning. *Technical Report 98-11,* Iowa State University, Ames, IA, USA, October 1998.

[44] Carnegie Mellon University. Inter Process Communication (IPC). Viewed 20 July 2008, http://www.cs.cmu.edu/~ipc/

[45] MÄK Technologies. MÄK High Performance RTI. Viewed 20 July 2008, http://www.mak.com/products/rti.php

[46] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Modelling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules*, IEEE Std 1516-2000, New York, 2000.

[47] Goodrich Corporation. Laser Obstacle Awareness System (LOAS). Viewed 20 July 2008, http://www.sensors.goodrich.com/literature/lit_pdfs/4109_LOAS.pdf

Appendix A

Wrapper Method Implementations

This section contains code from the CMU IPC and MÄK RTI Wrapper classes that is referred to in Chapter 5. A component's wrapper class implementations is the only class that needs to be modified (other than specifying which wrapper to use in the component class) in order to port it to another middleware.

A.1 CMU IPC Wrapper Methods for Autopilot

```
int Autopilot_Wrapper_IPC::connect () {
      if (IPC_connect(componentName)!= IPC_OK)
            return 0;

      // declare message publications (turnMsg is a class variable)
      IPC_defineMsg(turnMsg.name, IPC_VARIABLE_LENGTH, turnMsg.type);

      // ...

      // declare message subscriptions (gpsSensorMsg and myName are
      // class variables)
      IPC_subscribe(gpsSensorMsg.name, gpsSensorMsgHandler, myName);

      return 1;
}

int Autopilot_Wrapper_IPC::disconnect() {
      if (IPC_disconnect()!= IPC_OK)
            return 0;

      return 1;
}

void Autopilot_Wrapper_IPC::tick () {
      IPC_listen(0);
}

void Autopilot_Wrapper_IPC::publish (char* msgType, char* message) {
      IPC_publishData(msgType, &message);
}

void Autopilot_Wrapper_IPC::publish (char* msgType, char* message[]) {
      IPC_publishData(msgType, message);
}
```

A.2 MÄK RTI Wrapper Methods for Autopilot

```
int Autopilot_Wrapper_MAKRTI::connect () {

    try {
        // Federate and Federation info
        std::string federationName("MAK_Autopilot");
        std::string federationFile("MAK_Autopilot.xml");
        std::string federateType("MAK_Autopilot");

        // Create the federation (rtiAmb is a class variable)
        createFedEx(rtiAmb, federationName, federationFile);

        // Join the federation (fedAmb a is class variable)
        joinFedEx(rtiAmb, &fedAmb, federateType, federationName);

        // declare message publications (turnMsg and myName are class
        // variables)
        publishAndRegisterObject(rtiAmb, turnMsg, myName);

        // ...

        // declare message subscriptions (gpsSensorMsg is a class
        // variable)
        subcribeToObject(rtiAmb, gpsSensorMsg);

    }
    catch (RTI::Exception& ex)
    {
        return 0;
    }

    return 1;
}

int Autopilot_Wrapper_MAKRTI::disconnect () {
    try {
        rtiAmb.resignFederationExecution(RTI::DELETE_OBJECTS);
    }
    catch (RTI::Exception& ex)
    {
        return 0;
    }

    return 1;
}

void Autopilot_Wrapper_MAKRTI::tick () {
    rtiAmb.tick(0.1, 0.5);
}

void Autopilot_Wrapper_MAKRTI::publish (char* msgType, char* message) {

    // object name is msgType + myName + processID
    std:string objectName = msgType;
    unsigned int objId = abs(getpid());
    std::stringstream pid;
    pid << objId;
    objectName += pid.str();

    // get reference to object
```

```
    theObjectHandle = rtiAmb.getObjectInstanceHandle
       (objectName.c_str());

    // get object attribute
    DtAttrNameHandleMap theAttrNameHandleMap =
        theObjectNameAttributesMap.find(objectName)->second;
    RTI::AttributeHandleValuePairSet *attrValue =
         RTI::AttributeSetFactory::create(theAttrNameHandleMap.size());

    // assign value to attribute
    DtAttrNameHandleMap::iterator iter = theAttrNameHandleMap.begin();
    std::string temp = message;
    attrValue->add(iter->second,temp.c_str(), temp.length()+1);

    // update attribute through rtiAmbassador
    std::stringstream ss;
    ss << "message";
    std::string tag(ss.str());

    rtiAmb.updateAttributeValues(
            theObjectHandle,
            *attrValue,
            tag.c_str());
}

void Autopilot_Wrapper_MAKRTI::publish (char* msgType, char* message[]) {

    // object name is msgType + processID
    std:string objectName = msgType;
    unsigned int objId = abs(getpid());
    std::stringstream pid;
    pid << objId;
    objectName += pid.str();

    // get reference to object
    theObjectHandle = rtiAmb.getObjectInstanceHandle
        (objectName.c_str());

    // get object attributes
    DtAttrNameHandleMap theAttrNameHandleMap =
        theObjectNameAttributesMap.find(objectName)->second;
    RTI::AttributeHandleValuePairSet *attrValues =
         RTI::AttributeSetFactory::create(theAttrNameHandleMap.size());

    // assign values to attributes
    int index = 0;
    for (DtAttrNameHandleMap::iterator iter =
          theAttrNameHandleMap.begin();
          iter != theAttrNameHandleMap.end();
          iter++)
    {
        std::string temp = message[index];
        attrValues->add(iter->second,temp.c_str(), temp.length()+1);
        index++;
    }

    // update attributes through rtiAmbassador
    std::stringstream ss;
    ss << "message";
    std::string tag(ss.str());

    rtiAmb.updateAttributeValues(
            theObjectHandle,
```

```
            *attrValues,
            tag.c_str());
}

// helper methods

void createFedEx(RTI::RTIambassador & rtiAmb,
                 string const& fedName,
                 string const& fedFile)
{
   try
   {
      rtiAmb.createFederationExecution(fedName.c_str(), fedFile.c_str());
   }
   catch(RTI::FederationExecutionAlreadyExists& ex)
   {
   }
   catch(RTI::Exception& ex)
   {
   }
   rtiAmb.tick(0.1, 0.2);
}

void joinFedEx(
   RTI::RTIambassador & rtiAmb, MyFederateAmbassador* fedAmb,
   string const& federateType, string const& federationName)
{
   bool joined=false;
   const int maxTry  = 10;
   int numTries = 0;

   while (!joined && numTries++ < maxTry)
   {
      try
      {
         rtiAmb.joinFederationExecution(federateType.c_str(),
            federationName.c_str(), fedAmb);
         joined = true;
      }
      catch(RTI::FederationExecutionDoesNotExist)
      {
      }
      catch(RTI::Exception& ex)
      {
      }
      rtiAmb.tick(0.1, 0.2);
   }
}

void publishAndRegisterObject(RTI::RTIambassador & rtiAmb, Message * message,
     char* myName)
{
   DtAttrNameHandleMap theAttrNameHandleMap;

   // Get the object class handle
   string theClassName = message->getName();
   try
   {
      theClassHandle = rtiAmb.getObjectClassHandle(
           theClassName.c_str());
           theAmbData.objectClassMap[theClassHandle] = theClassName;

      // Get attribute handles and construct name-handle map and
```

```
      // attribute set
      string attrName;
      vector <string> attributes = message->getAttributes();
      for (vector<string>::iterator iter = attributes.begin() ;
           iter != attributes.end() ;
           iter++)
       {
            attrName = *iter;
            theAttrNameHandleMap[attrName] =
                rtiAmb.getAttributeHandle(attrName.c_str(),
                theClassHandle);
       }

       // Construct an attribute handle set
       RTI::AttributeHandleSet* hSet =
           RTI::AttributeHandleSetFactory::create(
           theAttrNameHandleMap.size());
       for (DtAttrNameHandleMap::iterator iter =
           theAttrNameHandleMap.begin();
           iter != theAttrNameHandleMap.end();
           iter++)
       {
           hSet->add(iter->second);
       }

       // Publish
       rtiAmb.publishObjectClass(theClassHandle, *hSet);
       rtiAmb.tick(0.1, 0.2);

       // Register the object instance
       // objectName + myName + pid
       string objectName = message->getName() + string(myName);
       unsigned int objId = abs(getpid());
       stringstream pid;
       pid << objId;
       objectName += pid.str();
       theObjectHandle = rtiAmb.registerObjectInstance(theClassHandle,
       objectName.c_str());

       // Add name-handle to map
       theAmbData.objectInstanceMap[theObjectHandle] =
           rtiAmb.getObjectInstanceName(theObjectHandle);
       theObjectNameAttributesMap[objectName] =
            theAttrNameHandleMap;

       rtiAmb.tick(0.1, 0.2);
   }
   catch (RTI::Exception& ex)
   {
   }
}

void subscribeToObject(RTI::RTIambassador & rtiAmb, Message * message)
{
   DtAttrNameHandleMap theAttrNameHandleMap;

   // Get the object class handle
   string theClassName = message->getName();
   try
   {
      theClassHandle =
         rtiAmb.getObjectClassHandle(theClassName.c_str());
      theAmbData.objectClassMap[theClassHandle] = theClassName;
```

```
        // Get attribute handles and construct name-handle map and
        // attribute set
        string attrName;
        vector <string> attributes = message->getAttributes();
        for (vector<string>::iterator iter = attributes.begin() ;
             iter != attributes.end() ;
             iter++)
        {
              attrName = *iter;
              theAttrNameHandleMap[attrName] =
                  rtiAmb.getAttributeHandle(attrName.c_str(),
                  theClassHandle);
        }
        // Construct an attribute handle set
        RTI::AttributeHandleSet* hSet =
              RTI::AttributeHandleSetFactory::create(
              theAttrNameHandleMap.size());
        for (DtAttrNameHandleMap::iterator iter =
              theAttrNameHandleMap.begin();
              iter != theAttrNameHandleMap.end();
              iter++)
        {
              hSet->add(iter->second);
        }

        // Subscribe
        rtiAmb.subscribeObjectClassAttributes(theClassHandle, *hSet);
        rtiAmb.tick(0.1, 0.2);
    }
    catch (RTI::Exception& ex)
    {
    }
}
```