Air Traffic Control Using Virtual Stationary Automata

by

Matthew D. Brown

B.S., Massachusetts Institute of Technology (2006)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

© Matthew D. Brown, MMVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
June 8, 2007

Certified by . . . .
Nancy Lynch
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Air Traffic Control Using Virtual Stationary Automata

by

Matthew D. Brown

Submitted to the Department of Electrical Engineering and Computer Science
on June 8, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

As air travel has become an essential part of modern life, the air traffic control system has become strained and overworked. This problem is occurring because the capacity of the current air traffic control system is severely limited by the capabilities of its human operators. Therefore, if we are to increase the capacity of the air traffic control system, then we must develop new automated systems for air traffic control.

In my thesis, I take a distributed approach to automated air traffic control. I use a wireless ad-hoc network to simulate a layer of Virtual Stationary Automata, or VSAs, which are virtual machines located at fixed locations in space. These VSAs can then be used to help coordinate the aircraft in the air traffic control system.

I model the air traffic control system as a directed graph, showing how the continuous real world air traffic can be abstracted into a discrete graph representation. Using this graph representation, I provide two algorithms to perform safe and efficient air traffic control and prove their effectiveness.

Thesis Supervisor: Nancy Lynch
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I have received an amazing amount of help in various forms from innumerable people during the writing of this thesis and throughout my time at MIT, and would like to recognize a small number of the individuals who helped make it all happen.

First, I would like to thank Nancy Lynch, who, aside from introducing me to distributed algorithms during a summer UROP years ago, has been dedicated to excellence in every step of the thesis writing process. Without her help and insight, this work would have lost so much of its technical rigor and clarity, and for that I am immensely appreciative.

I would also like to thank the rest of the Theory of Distributed Systems group that I have had the pleasure of working with these past couple of years. I would especially like to thank Tina Nolte for her extraordinary help throughout the process of the development of my algorithms in this thesis, and for giving me a solid base of research to model this thesis upon. Also, Seth Gilbert and Calvin Newport were extremely helpful while working on the VN Simulator, and without that work this thesis never would have happened.

My parents and family have been extremely supportive of me, and without that support and the strong value placed on education that they have instilled in me since childhood, I never would have been able to get this far.

And finally, I would like to thank Elizabeth for her constant emotional support, and for always being the one I can talk to about anything and everything.

Contents

1 Introduction **8**

1.1 Outline of the Thesis 9

2 Background **11**

2.1 Air Traffic Control 11

2.2 Distributed Algorithms and Virtual Stationary Automata 15

2.3 VSAs and Air Traffic Control 19

2.4 Chapter Summary 20

2.5 Global Constants 21

3 The Physical Network Layer **22**

3.1 Architecture of the Network Layer 22

3.2 Aircraft - The Physical Nodes 23

3.3 Communications - The Bcast Service 25

3.4 RealWorld 29

3.5 The Physical Layer 32

3.6 Chapter Summary 32

4 The Continuous VSA Layer **34**

4.1 Regions 35

4.2 Architecture of the VSA Layer 36

4.3 Clients 39

4.4 Virtual Stationary Automata 40

4.5 Intraregional Communication - Bcast 41

4.6 Interregional Communication - VtoVcast . . . 42
4.7 RealWorld . . . 44
4.8 The Continuous VSA Layer . . . 45
4.9 Emulation of the VSA Layer . . . 45
4.10 Chapter Summary . . . 46

5 The Known Path Model **48**
5.1 Architecture of the Known Path VSA Layer . . . 49
5.2 Graph Representation and Regions . . . 51
5.3 GraphRealWorld . . . 53
5.4 The Known Path VSA Layer . . . 56
5.5 Bounding Areas . . . 61
5.6 Arranging the Bounding Areas . . . 62
5.7 Emulating the Known Path Layer with the Continuous Layer . . . 67
5.8 Chapter Summary . . . 81

6 The FIFO Algorithm **83**
6.1 Problem Specification . . . 84
6.2 Capabilities of the Known Path VSA Layer . . . 84
6.3 Algorithm . . . 85
6.4 Proofs of Safety and Progress . . . 91
6.5 Chapter Summary . . . 95

7 The Heuristic Priority Algorithm **96**
7.1 Efficiency in a Free-Flight System . . . 97
7.2 The Time to Arrival Heuristic . . . 97
7.3 Algorithm . . . 99
7.4 Chapter Summary . . . 100

8 Conclusions **102**
8.1 Contributions . . . 102
8.2 Future Work . . . 103

8.3 Closing Comments 105

9 Appendix **107**

9.1 References 107

List of Figures

3-1 Architecture of the Network Layer for $I = \{1, 2, ..., n\}$ 23
3-2 TIOA Code for RealWorld 30

4-1 Architecture of the VSA Layer, for $I = \{1, 2, ..., n\}$ and $J = \{1, 2, ..., k\}$ 37
4-2 TIOA Signature for Clients and VSAs in the VSA Layer 38

5-1 Architecture of the Known Path VSA Layer, for $I = \{1, 2, ..., n\}$, $J = \{1, 2, ...k\}$ 50
5-2 A Constraint on the Layout of Vertices and Regions 52
5-3 TIOA Code for GraphRealWorld 54
5-4 Cross Sections of Bounding Area ba and Inner Bounding Area iba 64
5-5 Bounding Boxes b_1 and b_2 Meeting at Bounding Cylinder c_3 66
5-6 Emulation of the Known Path VSA Layer, for $I = \{1, 2, ...n\}$, $J = \{1, 2, ..., k\}$ 68
5-7 TIOA Code for Translation Automaton CD_i, for $i \in I$ 70
5-8 TIOA Code for RealWorld with Graph Constraints 72

6-1 TIOA State for VSA vn_j for $j \in J$ 86
6-2 TIOA Actions for VSA vn_j for $j \in J$ 88
6-3 TIOA Code for Client Automaton a_i for $i \in I$ 90

7-1 TIOA State for VSA vn_j for $j \in J$ 99
7-2 TIOA Actions for VSA vn_j for $j \in J$ 101

Chapter 1

Introduction

As air travel has become an essential part of modern life, the air traffic control system has become strained and overworked. Frequency and routing of flights are often limited by the capabilities of the modern air traffic control system, and the problem is getting worse every year. Compounding the problem, demand for a new class of very light aircraft threatens to further clog the system, rendering it ineffective.

The most significant reason for this capacity problem is the fact that ATC has not evolved as air travel has grown. While the tools and systems have improved with modern technology, the system is still limited by the capabilities of the air traffic controllers that operate it. Therefore, in order to adapt the air traffic control system for the demands of the coming years, we must begin to devise automated systems for air traffic control.

My approach, detailed in this thesis, uses distributed algorithms to allow aircraft to control themselves, without the need for a fixed air traffic control infrastructure. There are many advantages to this approach: the computing and communications hardware are already installed on the aircraft; no expensive new infrastructure needs to be deployed for the system to work; and the system can be deployed in locations where it is either too costly or simply impossible to install static infrastructure. That said, without static infrastructure, many difficulties and complexities arise, which must be accounted for in the interest of safety.

At a high level, my method is to utilize a wireless *ad-hoc* network to implement a number of Virtual Stationary Automata, a system which simulates reliable virtual machines at specific locations in space. The aircraft's computation and communication ability can be used to simulate

these virtual machines reliably. Using these virtual machines, distributed solutions to problems become much easier, as I can rely on these virtual machines as a point of communication, data storage, and decision making.

1.1 Outline of the Thesis

I will begin in Chapter 2 by discussing the background of the research. First, I will summarize some of the current practices in air traffic control at a relatively high level so that the reader can be properly introduced to the issues at hand. Next, I will introduce the theoretical concept of Virtual Stationary Automata (VSAs), and the relevant work done in the field. After that, I will connect the two topics, explaining how the concept of Virtual Stationary Automata can be used to help create an algorithm for air traffic control.

After completing discussion on the background of the research, I will present a series of models for the network layer of the system. In Chapter 3, I will specify the capabilities of the aircraft and the capabilities of the wireless ad-hoc network which will be run on the aircraft. Then, in Chapter 4, I will specify the VSA layer, a virtual networking layer that runs on top of the physical network. Throughout these two chapters, I will make appropriate assumptions which will allow us to get to the core of the problem and avoid being concerned with extraneous details of the operation of the network layer and emulation of the VSA layer.

I will then provide a model for the system called the Known Path Model in Chapter 5. The model will represent a *limited free-flight system* of air traffic control. This means that there are a number of predetermined paths through airspace that must be followed, but the pilot can independently choose which of these paths to follow. I will begin by providing a modified VSA layer which contains a representation of this model. Then, I will describe the physical world that the system will run in, and explain how the physical world is abstracted into a graph representation to represent the known paths.

After that, I will specify two algorithms for air traffic control using the Known Path Model. This first, presented in Chapter 6, will be called the *FIFO Algorithm*, named for its handling of conflicts using a queueing system. It will focus on keeping the aircraft safely separated from one another. The second, presented in Chapter 7, will be called the *Heuristic Priority Algorithm*, and it replaces the simple queueing system with a method for heuristically prioritizing aircraft when

conflicts arise. It will focus on efficiently moving the aircraft to their destinations while ensuring that the separation requirement still holds.

Finally, I will briefly discuss where this research leaves us, and make some suggestions on future avenues of research that would be interesting, ending with my conclusions about the algorithms provided within.

Chapter 2

Background

In this chapter, I will discuss the background of this research into a distributed algorithmic approach to air traffic control. I will begin by giving a short primer on the current system of air traffic control, how it works, and some new developments in the area. I will then proceed to discuss ad-hoc networks, some recent research, and how the research led to the formalization of the Virtual Stationary Automaton. Finally, I will present some previous work with Virtual Stationary Automata, showing how the concept can be applied to my goal of designing an air traffic control system.

2.1 Air Traffic Control

In order to understand the problem domain, a brief history of the current system of air traffic control (ATC) will be helpful. The majority of the information in this section on the history of air traffic control, and the current air traffic control system can be attributed to either texts on air traffic control, such as Nolan [9], or to general information encyclopedias [10,11].

In the earliest days of flight (before the 1930s), air travel was primarily performed during the day under clear visibility conditions. Because the traffic was so visible and there was so little of it there was no need for an organized system of air traffic control; safety could be left up to the pilots. When airplane instrumentation evolved to allow safe control of flight without having good visibility, a new set of rules for flight called instrument flight rules (IFR) were developed. This introduced the problem that pilots were able to fly without being able to see other aircraft

in the sky. As IFR flight became more common, and pilots started flying in increasingly hostile conditions, the need for a formal system of air traffic control became obvious.

2.1.1 Structure of the ATC System

Although the technology has evolved significantly in the past few decades, the system of air traffic control currently employed in the United States and around the world is quite similar to the system implemented in the mid 1960s. The airspace is divided up into a number of parts each of which is independently controlled. A plane flying from one part to another is passed off between the authority in one part to that of the next part so its route can be continued safely. We examine the way that the US airspace is divided up as a start of our discussion of ATC.

The United States airspace is divided up into twenty-one zones called *centers*. Each center is run by an Air Route Traffic Control Center (ARTCC), which manages the airspace of the entire center except for small areas of airspace around airports. Each center is then divided further into a number of *sectors* comprising most of the airspace in the center, and a small number of special portions of airspace around some airports called Terminal Radar Approach Control (TRACON) airspace. The TRACON airspaces are monitored by special facilities, while the sectors are monitored by the controllers in the ARTCC.

Within each ARTCC, aircraft in each sector are monitored and separated by a controller responsible for that individual sector. That controller instructs each pilot how to remain safely separated (either horizontally, vertically, or both) from other aircraft in the area. If congestion in the area gets too bad, airplanes can be forced into a holding pattern, staying in a sector until the congestion subsides and the air traffic controller allows the pilots to continue along their route.

2.1.2 En-Route Control

One of the most conceptually difficult matters of air traffic control is the handoff of a plane between controllers when a plane moves from one sector to another. How do the controllers seamlessly transfer a plane's information from one to another, ensuring that there will be no problems?

Each plane has a *flight progress strip* which includes all the necessary data for tracking the flight. These strips are either on paper or electronic, and they get passed between controllers,

and between ARTCCs, as the flight progresses. Adjacent ARTCCs have *Letters of Agreement* with one another, dictating appropriate altitudes and locations for planes to fly in when passing between the centers. Within each ARTCC, controllers of adjacent sectors also have agreements similar to the Letters of Agreement between the ARTCCs. But when special circumstances arise, it is up to the controllers of each sector to maintain proper separation when a plane transitions between them.

Other than the transitions, the *en-route control* of an aircraft, or the control that occurs while flying between airports, is relatively straightforward. Controllers keep the planes in established airways, depending on the origin and destination of the flight, and make sure the plane is using the correct altitude, heading, and velocity to remain separated from other aircraft in their sector. This separation must conform to the FAA's minimum of (under normal conditions) 1000 feet vertical separation or 3 nautical mile horizontal separation. Changes occur sometimes due to bad weather or turbulence in which case the pilot may request changes in altitude or course, and the controller must make sure that these changes will not result in any of the rules of separation being violated. Weather and traffic can also cause the controller to redirect or hold an aircraft in a specific sector, although this holding is much more commonly done in the area around the destination airport.

2.1.3 Terminal Control

The other half of the air traffic control equation is called *terminal control*, in which the takeoff and landing patterns of the aircraft are determined. These are the functions performed by the special portions of airspace at either the TRACON facilities or local airports' tower operators. This terminal control also covers control of the runways, taxiways, and terminals: including both planes and airport ground vehicles. This takes place in TRACON facilities, using radar and the other electronic monitoring technologies that are in place at the airport. Terminal control also takes place in airports' control towers, which use electronic tools as well as visual identification of aircraft and vehicles to control the local area.

2.1.4 Looking Ahead

As I stated earlier, the system that was in place nearly forty years ago is very similar to the system that is used today. This is problematic for a number of reasons. First of all, the sheer volume of air traffic has increased so much in that time that the old methods may not be as useful to provide safe control of the increased number of aircraft that are now operational. Second, the technology level has increased rapidly in that time, and the current system is having a hard time catching up. While many new technologies have been added to the ATC system in the past forty years, the overall workflow is very much the same, and it may be a limiting factor to the technologies' integration. For example, the recently developed Traffic Collision Avoidance System (TCAS), which alerts pilots to possible collisions and instructs them to adjust course accordingly, has caused a small number of in-air collisions or near misses when the TCAS and the local air traffic controller gave pilots conflicting instructions.

From an efficiency perspective, the rigidity of the current system wastes a lot of resources; certain airways might be reserved for a certain pattern of traffic that is not occurring at that moment, but could maybe be used for a different pattern of traffic to relieve temporary congestion problems. Limitations of human controllers have led to a very rigid framework, in order to make the traffic control problem manageable for controllers.

In an effort to help alleviate these limitations, institutions such as NASA, working with the FCC[14], are developing systems to return the air travel community to a system of *free flight*, allowing pilots to control the course of their aircraft without the need for en-route ATC. Using the new systems being developed for discovering nearby aircraft, avoiding mid-air collisions under IFR conditions is now possible. Using the current system, a lot of our airspace is wasted due to a lack of controllers for all sectors. Free flight could allow pilots to utilize the entire airspace, but may cause serious problems in heavily congested areas around major flight routes.

Advances in computing and technology, though, can now allow individual aircraft to coordinate with other aircraft to perform air traffic control in a decentralized manner. By developing a new system to ensure air safety, the human errors involved can be minimized and the volume of air traffic supported can be increased significantly while maintaining or improving safety. A hybrid system, somewhere between the free flight system and the current ATC system, seems to be the most likely candidate for use.

2.2 Distributed Algorithms and Virtual Stationary Automata

Now that we have an understanding of the air traffic control system that is currently in place, the methods for developing an algorithm to control air traffic can be explained in detail. While there is a large amount of research going on in the creation of a number of new next-generation air traffic control systems, this research will take a different stance in that it will be primarily a distributed algorithm, one that needs little or no centralized control, and that could theoretically be run using only hardware similar to that already installed in modern aircraft.

2.2.1 Wireless Ad-hoc Networks

The current system of air traffic control and its communications features comprise what is called a *basestation-type* wireless network. While communication between an aircraft and the ATC center is done through a wireless signal, the infrastructure of the network (the ATC centers) is static. The task of coordinating the large number of aircraft in the area becomes (at a high level) an aircraft asking the center for instructions, the center analyzing the data that is available to it in order to make the best decision, and the center replying to the aircraft. This system has a great benefit in the static nature of the center: the aircraft always know where and how to reach each center as they travel, and the center will always be there to respond to the aircraft. That benefit can also be a detriment, though, when we realize that this single static ATC center can become a bottleneck or a single point of failure if problems were to arise.

Therefore, we will consider the case where this is no static physical infrastructure in our communications network. Without fixed infrastructure, we are left with a large number of mobile physical nodes (aircraft), which may be entering and exiting sectors constantly and entering and leaving communications range with other aircraft without significant warning. This type of network topology comprises a wireless *ad-hoc* network, one that gets created by the combined capabilities of the clients in range of one another.

An ad-hoc network has a complementary set of problems to those of the basestation-type network: there is no static node to interact with. For example, a plane flying into a sector controlled by an ad-hoc network can see a completely different set of network connections every time it enters the network.

The problem becomes one of determining how each client can get the information it needs

and safely coordinate with other clients, but without the benefit of a reliable controller to interact with. This problem is an important concern in research into distributed computing over ad-hoc networks.

2.2.2 Earlier Work in Ad-hoc Networks

To solve the problem, we can consult a great deal of recent previous work in the area. A naive approach to coordinating over a wireless ad-hoc network could attempt to treat communication similarly to a wired static ethernet, discovering the other computers which could be reached by the communication link, and communicating until the link was broken due to movement or failure. It would be impossible to tell when a link was likely to go down, as the wireless ad-hoc network contained no information about the relative locations of the clients to one another.

By adding the use of geographical information into the system, though, the highly dynamic nature of the network becomes more predictable. If each client has the ability to determine its location through a service such as the Global Positioning Satellite (GPS) system, the local network topology becomes easy to examine at any time. A theoretical client on this network could use the geographical information to make important decisions about whether another client is likely to be in communications range in the near future, or coordinate behavior with other clients based on their locations. Under the assumption that clients have access to this geographical information, the possibility arises for reliable algorithms to coordinate clients on an ad-hoc network.

An interesting problem in distributed computing is to provide a long-range routing service over an ad-hoc network, in order to transmit data wirelessly further than could be done over a single link. Algorithms such as GeoCast [5] and GOAFR [4] both use the location of the clients to provide a mobile routing solution. Each client can determine which of the others in communication range is most likely to be able to continue the transmission of a message to a given location, and could therefore pass the message to that client, who would continue to pass it until it either reached its destination or the destination is determined to be unreachable.

Another problem is more local: storing data in a certain geographical location so that any clients who come within range will have access to it. Storing the data on an individual client could not in itself work, as that client may leave the area where the data needs to remain. A

proposed solution to this problem by Dolev, Gilbert, Lynch, Shvartsman, and Welch is called Geoquorums [1]. As the name suggests, the algorithm uses geographical information to achieve agreement between a number of clients: specifically, to solve the aforementioned problem of storing atomic data reliably in a specific location without any static infrastructure to store the data on.

The key idea of Geoquorums is that it uses an emulated virtual machine to store this data. Clients on the network are therefore able to communicate with a single static machine in a specific location, and do not have to worry about the dynamic nature of the ad-hoc network. The algorithm was not originally very reliable, as once the data storage point failed, clients were never able to recover the lost data. This problem has since been rectified by Tulone [18] using a slightly changed implementation and set of mobility constraints to guarantee data availability.

Work has also been done by Dolev et. al. [15] on Virtual Mobile Nodes, which were given the ability to change position along preplanned trajectories. Virtual Mobile Nodes were formalized as simple *Input/Output Automata* (or I/O Automata) [16, 17] and they could take an input, process that input, and provide an output. While they still had a virtual location in space, they were mobile, not fixed as Virtual Stationary Automata would be.

2.2.3 Virtual Stationary Automata

As previously stated, I use Virtual Stationary Automata in this thesis, an idea which drew influence from the work mentioned in the last section. Virtual Stationary Automata [3,7,8] are fixed in space as in the Geoquorums work, can react and process information like the Virtual Mobile Nodes, but are given access to not only a GPS service, but a real-time clock service as well.

A VSA is a Timed I/O Automaton [19], with access to a real-time clock and a local broadcast service allowing it to communicate with clients in its region and VSAs in the neighboring regions. The mobile physical nodes in an ad-hoc network emulate this machine, but how does this emulation of the VSA work?

The idea behind the VSA is to separate the total area of the network into some number of *regions*. Within each region is a Virtual Stationary Automaton (also known as VSA or virtual node). For simplicity, we can consider the center of the region to be the point that contains the VSA. We choose the size of the region such that under normal operation, any physical node

within the region will be able to communicate with any other physical node within that or a neighboring region.

The physical nodes within each region do the work to emulate the VSA within the region. A physical node is elected as *leader* to act as the node that actually sends all communication from the VSA of that region, and to handle the arrival of new physical nodes. To achieve fault-tolerance in the maintenance of the VSA state, the state of the VSA is replicated at some number of physical nodes in the region.

All communication within a region is done through broadcasts, allowing all physical nodes in the region to receive the message. If the VSA needs to send a message, the leader performs that send function. When the leader leaves the region, a new leader is elected to take its place, and since the other physical nodes have seen the same messages as the leader, no state should be lost. When a new physical node enters the region, the leader sends the state of the VSA to it so it can start emulating the machine at the same point as the rest of the physical nodes.

The VSA has become an effective framework for writing distributed algorithms. A number of papers have been written building on the work done in formulating the VSA to perform a wide variety of coordination tasks [7,8,12,13]. One algorithm uses VSAs to coordinate the motion of mobile devices [7], instructing clients to go where they are needed to fill in an area on a curve in space. Another set of algorithms [8] solves the problem of routing messages between clients over long distances using a multitiered approach. First, a service is created to route a message to a certain location in space. Second, a service for locating clients in the global network is created, using a VSA at the client's home location to keep track of the client's current position. Using those two services, a client-to-client message routing service is created, allowing messages to be routed over long distances without the use of static infrastructure.

Some work has also been done to expand the VSA out of the theoretical realm [12, 13]. A framework for running a restricted version of VSAs has been written in Python and run on mobile devices. Programs for both the client and the VSA can be plugged into the system, allowing it to perform virtually any task that a non-timed, reaction-based VSA system can do. A simulation of a traffic light in space was implemented and tested, using the mobile devices to display the state of the light. This, in a preliminary way, is one of the main inspirations for the upcoming work in air traffic control. While there are currently no plans to implement the proposed ATC system,

the fact that it is possible in the future certainly is interesting.

2.3 VSAs and Air Traffic Control

In the previous section, I attempted to give some background on the concept of Virtual Stationary Automata, but I have not yet discussed exactly why the concepts behind the VSA fit the problem of air traffic control so well. By connecting the problem of air traffic control to the theory of VSAs, it becomes clear why this research is interesting from two perspectives: the perspective of someone looking for a novel method of air traffic control, and the perspective of an algorithms researcher looking for interesting new ways to use VSAs.

The specific problem that I am solving is the modeling of en-route air traffic control using VSAs. The problem of terminal control is separate, and will not be covered. While this will be a theoretical treatment of the problem, the goal is to make reasonable assumptions about the system, keeping it similar to the real world. I will also justify a number of simplifying assumptions, which should allow readers from both theoretical and programming backgrounds to understand the algorithms presented.

While the current air traffic control system has been around for a long time, a distributed approach such as this has only been conceived of recently, for a number of reasons:

- Until recently, the ability to have enough computing power to run an air traffic control algorithm locally on all aircraft was not possible. While large jet liners have had significant computation power for years, these large aircraft are not the only aircraft in the sky. A complete air traffic control system must also work with smaller, single-engine aircraft. One could easily implement this system on modern laptops or small devices stored in even the smallest single-engine aircraft.

- The invention and refinement of the GPS system has allowed aircraft to have detailed and accurate information about their position without using ground-based radar. If the radar of an ATC center were necessary to position the airplane, there would be little reason to distribute control of the air traffic - the infrastructure would be necessary for the operation of the system. But since each aircraft now has the ability to determine its own position, we can think about the possibility of distributing ATC work to aircraft themselves.

- There may not have been appropriate methods for handling the complexity of ATC in a distributed manner. The concept of VSAs is new, and the difficulty of creating a safe distributed algorithm for ATC may have been too difficult without such a concept.

- The air travel system has expanded greatly over the past few years, and with the prospect of cheaper, smaller, personal aircraft, it will expand even further in the near future. Therefore, an examination of the ATC system and how to improve upon it is extremely important and relevant right now.

Why should a VSA system be used instead of a new infrastructure-based approach, or a different type of distributed algorithm? The main benefit is that it allows us to think about a decentralized air traffic control system as if it were a basestation-type system. Once we have set up a stable VSA layer, we can treat the distributed problem of ATC as if it was a centralized problem, which is much easier.

If we wanted to use infrastructure-based automated air traffic control, we could still use the VSA system as a backup. We can allow the ground-based infrastructure to function as the leader, simulating a Virtual Stationary Automaton at the same location. In the event of a failure in the infrastructure, the aircraft can seamlessly take over the work, simulating the VSA as if nothing happened to the infrastructure. This eliminates the single point of failure at the infrastructure point.

Finally, it removes the human factor that limits the capacity of air traffic control, while adding to the scalability of the system. The free flight aspect of the system allows pilots to navigate around weather and congested airspace as they see fit. The VSA system can scale with the increasing size of the air traffic control system, adding new routes without the problem of instructing human controllers on how to use them.

Therefore, by electing to use this VSA system for the air traffic control, we could be keeping all the benefits of reliable infrastructure while improving the scalability of the air traffic control system, letting it grow with the air travel industry's needs.

2.4 Chapter Summary

In this chapter I have:

- Reviewed the literature of the current air traffic control system, and explained some previous work done in the field of distributed algorithms.
- Introduced the reader to the concept of Virtual Stationary Automata, explained what has been done with them, and outlined how they are used.
- Explained why the problem of air traffic control fits into the framework of Virtual Stationary Automata, and why I use it in my algorithms for air traffic control.

2.5 Global Constants

2.5.1 Overview

Before I begin discussing the first model for the Physical Network Layer, I will define a number of global constants that will be used throughout the thesis.

2.5.2 Definitions

- The finite set I represents the set of names that can be used to identify an aircraft.
- The finite set J represents the set of names that can be used to identify a region.
- The function $nbrs : J \rightarrow 2^J$ returns, on input j, the set of j's neighbors. It is required that for all $j, j' \in J$, if $j' \in nbrs(j)$, then $j \in nbrs(j')$.
- The data type Msg is our finite message alphabet.
- The connected deployment space $R \subseteq \mathbb{R}^3$ represents the airspace in the system.

Chapter 3

The Physical Network Layer

This chapter is concerned with the networking capabilities that are required for my air traffic control algorithms. In it, I present the Physical Network Layer (or just Network Layer), which is the lowest layer of abstraction in the air traffic control system. The Network Layer is comprised of the aircraft, functioning as physical nodes; a communications service; and a representation of the real world that the system operates in.

3.1 Architecture of the Network Layer

The network layer is comprised of the set $A = \{a_1, a_2, ..., a_n\}$, with $n \in \mathbb{Z}^+$, of aircraft - timed input-output automata [19] that function as our physical nodes; a timed input-output automaton $RealWorld$, which represents the physical world that the aircraft operate in; and the timed input-output automaton $Bcast$, which represents the broadcast medium for the network layer.

A physical node a_i has an input $\mathsf{GPSinput}(loc)_i$ from $RealWorld$, with $loc \in \mathbb{R}^3$, which functions as the aircraft's GPS oracle. A physical node a_i also has an output action $\mathsf{bcast}(msg)_i$ which outputs a message to $Bcast$, and $\mathsf{brcv}(msg)_i$ which receives a message string msg from $Bcast$.

An output from a_i, $\mathsf{move}(p, t)_i$ to $RealWorld$ also exists, which will be used to allow the software on a_i to control what occurs in $RealWorld$. The action will be used to control a_i's representation in $RealWorld$, moving it to p within time t.

The remainder of the chapter is concerned with specifying the details of each component of the Network Layer, and putting them together as the formal definition for the Network Layer. I

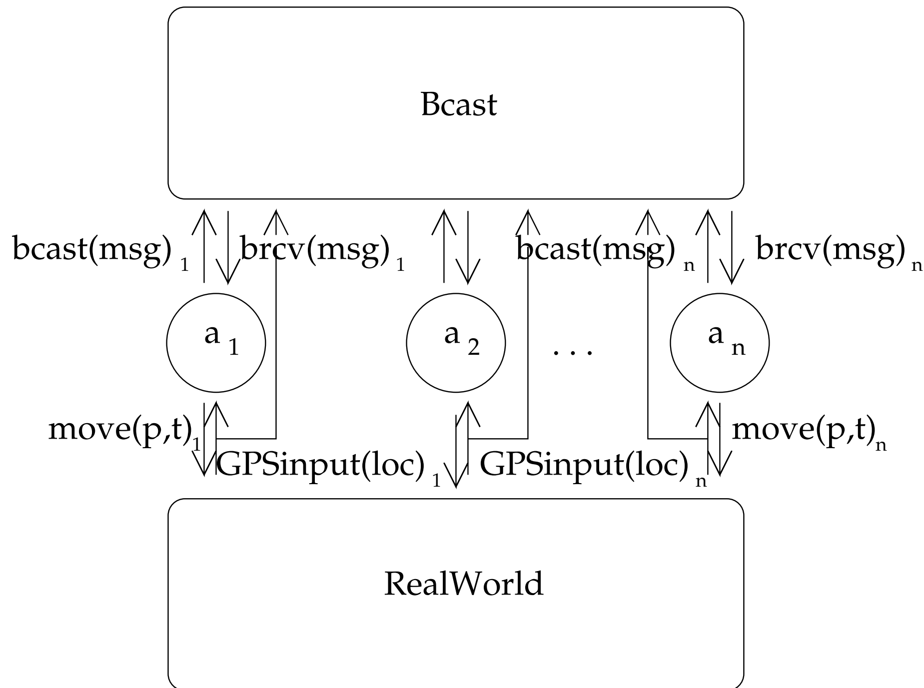


Figure 3-1: Architecture of the Network Layer for $I = \{1, 2, ..., n\}$

begin each section with an overview of the component, including some motivation for decisions I make within. Then, formal definitions for each component are specified at the end of each section.

3.2 Aircraft - The Physical Nodes

3.2.1 Overview

Aircraft serve as the *physical nodes* in our system. Each is equipped with a wireless communication device with the ability to *broadcast* and *receive* messages. As they are aircraft, I must assume that their communication is reliable, and that spontaneous hardware failures happen very rarely. That said, one goal is to design a system that is able to recover from these failures when they do happen.

While spontaneous failures are assumed to be rare, the network topology is constantly changing. Since all the physical nodes are traveling, the set of physical nodes that any one physical node is in contact with may change dramatically between two points in time. Therefore, our

model must not rely on two specific physical nodes being in communication for any large period of time.

In addition to a communication device, our physical nodes have some computational power. In our simulations, a simple desktop computer has more than enough power to run the algorithms that we will be using. For the purposes of the model, though, we will formalize their computational abilities in terms of abstract machines.

Therefore, I formalize the physical nodes as *timed input-output automata* [19]. They have the ability to *receive* inputs, corresponding to the receipt of a message through the wireless network. They also have the ability to *output*, broadcasting their output over the wireless network. In addition, physical nodes have access to a *GPS oracle*, which provides each node with its own location information as it moves throughout space.

Aircraft also have an internal real-time clock, which we will assume to be synchronized with the clocks of other physical nodes in the system. The problem of clock synchronization is not trivial, and there is significant research into methods of clock synchronization over wireless networks. That said, the use of a GPS service includes the use of an accurate clock for positioning, and since the algorithms in this thesis are not sensitive to small inconsistencies in time, for simplification we assume the aircrafts' clocks are synchronized.

3.2.2 Definitions

The type Msg is our finite message alphabet.

The set I represents the aircraft names that can be used to identify a unique aircraft.

For each $i \in I$, a_i represents a physical node in the Network Layer. The set of all such a_i is called A. Each $a_i \in A$ is a timed input-output automaton with the following actions:

- The output $\mathsf{bcast}(msg)_i$ action outputs a message $msg \in Msg$ to $Bcast$.
- The input $\mathsf{brcv}(msg)_i$ action receives a message $msg \in Msg$ from $Bcast$.
- The input $\mathsf{GPSinput}(loc)_i$ action receives i's location $loc \in \mathbb{R}^3$ from $RealWorld$.
- The output $\mathsf{move}(p,t)_i$ action to $RealWorld$, with $t \in \mathbb{R} \geq 0$ and $p \in \mathbb{R}^3$, which will be used to control i, moving it to p within time t.

For all messages $msg \in Msg$ output by a_i in a $\mathsf{bcast}(msg)_i$ action, msg is a globally unique message. This can be achieved by requiring that all $msg \in Msg$ include a field for the sender of the message $msg.sender = i$ and a field for the time of the message $msg.time = now$. The actual message sent is simply msg.

In addition to these actions, a_i may have other actions which must be internal, which are unspecified here.

The state of a_i includes:

- $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.
- $loc \in \mathbb{R}^3$, which is updated on each GPSinput.

In addition to these variables, a_i may have other state variables. Its transitions and trajectories are unspecified, except as noted above.

3.3 Communications - The Bcast Service

3.3.1 Overview

As previously stated, the aircraft will be operating on a wireless ad-hoc network; there is no fixed infrastructure present to provide a reliable point of communication. A physical node on the network has only the ability to *broadcast* a message to all other physical nodes, of which some are able to *receive* the message, but those out of communications range neither receive the message nor any indication that they have missed a message. The physical nodes have no ability to send a message to a single desired recipient, nor determine what other physical nodes are in range.

I represent this communication network with the timed input-output automaton $Bcast$ which receives messages from physical nodes and outputs those same messages to all physical nodes in communications range.

Communications Range

The range of a physical node's communications capability is fixed at a radius r, a distance in 3-space. No other physical nodes outside that range will receive broadcasts made by the phys-

ical node. Many complications arise from this assumption in a physical system that we, for the purposes of this model, can safely ignore, for these reasons:

- First, the communications range of different physical nodes will vary considerably based on the power of their transmitter. This is not a concern in this system, as we assume aircraft are all equipped with similar, standardized communications equipment.

- Second, the range may be affected by physical objects in the way, or interference from other equipment. Again, considering the space which we expect our network to be operating in (thousands of feet above the ground), obstructions will not occur, and interference is prevented due to aircraft's use of dedicated frequencies for communication.

- Third, problems with signal strength arise in real systems, as some physical nodes may receive corrupt or incomplete data at the edge of communication range. We assume that our assumed radius of communications r is such that within that range, anyone who receives a broadcast receives the complete message. While signal strength outside of the radius may vary, our algorithms will not depend on such messages, as you will see in the subsequent chapters.

Therefore, we are safe to assume that our broadcasts occur in a fixed radius r from the physical node.

Communications Reliability and Delay

When a physical node broadcasts a message, all other physical nodes within range will receive the message, and all messages are received in the order which they are broadcast. While these may seem like bold claims or oversimplifications, they are justified ones.

On perfect reliability, this is a thesis about the abstraction of the Virtual Stationary Automaton and their use in developing an algorithm for free flight air traffic control. This is not intended to be a thesis about network reliability. There are many methods for performing communications reliability: our implementation of the VSA emulator uses an absolute ordering on messages along with rebroadcasts and state synchronization to ensure that each physical node within range receives all messages. This is not the only way to provide reliability, and an imple-

mentor may wish to choose a different one. Therefore, for the purposes of this thesis, I assume that the reliability problem has been solved at a lower level.

On perfect ordering, I turn back to our implementation of the VSA emulator [12, 13], which contains a built-in service to provide message ordering. There are a number of ways to implement message ordering, and that is again, up to an implementor and beyond the scope of this thesis.

I will not assume that messages are delivered instantaneously. Due to the rebroadcasts that may be required to ensure reliability and ordering, it may take a non-trivial amount of time to get a message from one physical node to another in range. That said, our bound on the reception range allows us to assume a high probability of delivery which we can use to define an upper bound on the message delay. We will call this bound d, and we can assume that any message broadcast by a physical node will reach all other physical nodes in range within that time d.

There are two reasons this system will support such a bound:

- While broadcasts may occur quite often, the upper bound on range allows us to assume retransmissions are rare. Therefore, we can assume that most of the time, the network is not saturated, and no physical node will be required to wait a long time to transmit.

- There are no multiple-hop connections being made. Every message is a broadcast directly from the sender of the message. As a broadcast can be made only when there are no transmissions occurring already, the message will never be queued and will be delivered at the speed of the transmission medium immediately when sent.

Therefore, the claims of reliable message delivery and bounded transmission delay are justified for the purposes of this thesis. While some of the justifications of the claims I have made are not fully explained at this point, they should be made clear as the algorithm is presented.

3.3.2 Definitions

A timed input-output automaton $Bcast(d, r)$ represents the communications network in the Network Layer.

The parameters of $Bcast$ are:

- $d \in \mathbb{R}$, the upper bound on message delay between inputting a message and outputting it to the appropriate physical nodes.

- $r \in \mathbb{R}$, the communications range.

For each $i \in I$, $Bcast$ has the following actions:

- The input $\mathsf{bcast}(msg)_i$ action, which receives a message $msg \in Msg$ from a_i.

- The output $\mathsf{brcv}(msg)_i$ action, which sends a message $msg \in Msg$ to a_i.

- The input $\mathsf{GPSinput}(loc)_i$ action receives i's location $loc \in \mathbb{R}^3$ from $RealWorld$.

$Bcast$ contains a state variable $locs$, which is an array of locations, indexed by $i \in I$, such that $locs[i] \in \mathbb{R}^3$ gets updated upon each $\mathsf{GPSinput}(loc)_i$ action.

The behavior of $Bcast$ is as follows. Upon inputting $\mathsf{bcast}(msg)_i$ from a_i, within time d, $Bcast$ will output $\mathsf{brcv}(msg)_j$ to each a_j such that a_i and a_j are within r distance of each other according to $locs$ at the time the bcast was received by $Bcast$. Note that this means a_i will always receive its own broadcast.

Other requirements for the behavior of $Bcast$ include, for any three $i, j, k \in I$ where i, j, k may or may not be equal:

- Message Ordering: If a $\mathsf{bcast}(msg1)_i$ event precedes a $\mathsf{bcast}(msg2)_j$ event, and if $\mathsf{brcv}(msg1)_k$ and $\mathsf{brcv}(msg2)_k$ both occur, then the $\mathsf{brcv}(msg1)_k$ event precedes the $\mathsf{brcv}(msg2)_k$ event.

- Message Integrity: Each output $\mathsf{brcv}(msg)_j$ must have been preceded by an input action $\mathsf{bcast}(msg)_i$ such that msg in the two actions is the same and the $\mathsf{brcv}(msg)_j$ occurs at most d time after the $\mathsf{bcast}(msg)_j$ action.

- Message Uniqueness: No two $\mathsf{brcv}(msg)_i$ events occur for the same msg and the same i.

In addition to these variables, $Bcast$ may have other state variables. Its transitions and trajectories are unspecified, except as noted above.

3.4 RealWorld

3.4.1 Overview

The physical world in the air traffic control system will be represented in the Network Layer by the timed input-output automaton $RealWorld$. The main goal of $RealWorld$ is to keep track of the physical locations of the aircraft in the system, and to use that information to act as a GPS oracle to the physical nodes in A.

Airspace

The airspace for the air traffic control system is some contiguous subset of three-dimensional space. The airspace is represented by a connected deployment space $R \subseteq \mathbb{R}^3$. Aircraft are able to move freely in R, and we assume that they either never leave R, or if they do, then we are no longer concerned with them.

GPS Service

For each aircraft name $i \in I$ in the state of $RealWorld$, there is a corresponding physical node $a_i \in A$ for which $RealWorld$ outputs periodic GPS updates. These updates are performed by the output $\mathsf{GPSinput}(loc)_i$ action, which is an input to both a_i and $Bcast$.

These outputs are periodic with period ϵ. The period ϵ is chosen to be small enough that my algorithms can easily tolerate any slight inaccuracy in the location of a_i. This is intended to closely resemble the behavior of real GPS devices, which update their location extremely frequently.

3.4.2 Definitions

The connected deployment space of the system is $R \subseteq \mathbb{R}^3$.

A timed input-output automaton $RealWorld(\epsilon, v_max)$ represents the physical state of the real world in the Physical Layer. The parameter $\epsilon \in \mathbb{R} \geq 0$ is the GPS update period, and is assumed to be small. The parameter $v_max \in \mathbb{R}^+$ is the maximum velocity the aircraft can achieve.

```
   Constants:                                                  Actions:
2    ε, the GPS update frequency                                 Output GPSinput(loc)_i                              42
          ∈ ℝ ≥ 0                                                Precondition:
4    v_max , the maximum aircraft velocity                         now = kε ∧                                        44
          ∈ ℝ+                                                     GPSdone[i, k] = false ∧
6                                                                  loc = aircraft[i].loc                             46
   Signature:                                                    Effect:
8    Output GPSinput(loc)_i , i ∈ I, loc ∈ R                       GPSdone[i, k] ← true                              48
     Input move(p, t)_i , p ∈ ℝ^3, t ∈ ℝ ≥ 0
10   Internal finishMove_i , i ∈ I                               Input move(p, t)_i                                  50
     Internal Takeoff_i , i ∈ I                                  Effect:
12   Internal Landing_i , i ∈ I                                    if (dist(aircraft[i].loc, p) ≤ v_-max · t ∧       52
                                                                       aircraft[i].moving = false ∧
14 State:                                                              choose b ∈ {true, false}) then                54
     GPSdone, an array of booleans for each i ∈ I                    aircraft[i].moving ← true
16        and for k ∈ {0, 1, 2, 3...}                                aircraft[i].movingtime ← now + t                56
          initialized to false                                       aircraft[i].movingto ← p
18   aircraft, an array of type AircraftRecord                                                                       58
        for each aircraft[i], initialized to:                    Internal finishMove_i
20      aircraft[i].loc ← loc_NULL                               Precondition:                                       60
        aircraft[i].moving ← false                                 aircraft[i].loc = aircraft[i].movingto ∧
22      aircraft[i].movingtime ← 0                                 aircraft[i].moving = true                         62
        aircraft[i].movingto ← loc_NULL                          Effect:
24   now : ℝ                                                       aircraft[i].moving ← false                        64

26 Trajectories:                                                 Internal Takeoff_i                                  66
     evolves                                                     Precondition:
28     d(now) = 1                                                  now = aircraft[i].t_-takeoff ∧                    68
       for each i ∈ I, with a[i] = aircraft[i]:                    aircraft[i].loc = loc_-NULL
30       if (a[i].loc = loc_-NULL) then                          Effect:                                             70
           constant a[i].loc                                       aircraft[i].loc ← aircraft[i].loc_-takeoff
32       else                                                                                                        72
           0 < |d(a[i].loc)| ≤ v_max                             Internal Landing_i
34   invariant                                                   Precondition:                                       74
       if (a[i].moving = true) then                                aircraft[i].loc = aircraft[i].loc_-landing
36       dist(a[i].loc, a[i].movingto) ≤                         Effect:                                             76
           v_max(a[i].movingtime − now)                            aircraft[i].loc ← loc_-NULL
38   stops when
       any precondition is satisfied
```

Figure 3-2: TIOA Code for RealWorld

The constant location $loc_NULL \in \mathbb{R}^3 - R$ is some location outside the deployment space which is used to initialize the location of aircraft before they enter the system.

The data type $AircraftRecord$ is a record with fields:

- $t_takeoff \in \mathbb{R} \geq 0$, the time the aircraft enters the system.
- $loc_takeoff \in \mathbb{R}^3$, the location at which the aircraft enters the system.
- $loc_landing \in \mathbb{R}^3$, the location at which the aircraft leaves the system.
- $loc \in \mathbb{R}^3$, the current location of the aircraft.
- $moving$, a boolean representing whether or not this aircraft is being controlled by a move action.
- $movingtime \in \mathbb{R} \geq 0$, the deadline for finishing a move action.
- $movingto \in R$, the location in R that the aircraft is moving to.

$RealWorld$ also has a single output action, which is periodic with period ϵ, as well as an input action:

- The output GPSinput$(loc)_i$ action, where $loc = aircraft[i].loc$, and the output is received by a_i and $Bcast$. This output occurs for each $i \in I$ at $now = 0, \epsilon, 2\epsilon, ...$
- The input move$(p, t)_i$ action from a_i, with $t \in \mathbb{R} \geq 0$ and $p \in \mathbb{R}^3$, which will be used to control i, moving it to p within time t. If a subsequent move$(p', t')_i$ occurs while the aircraft is moving, the subsequent action is ignored.

  The path that i takes is nondeterministic, and nondeterministic flight resumes after reaching p. If the move action would force i to exceed the maximum velocity v_max, the action is ignored. The action can also be declared to be invalid nondeterministically.

The state of $RealWorld$ includes:

- $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.

- An array of $AircraftRecord$s, $aircraft$, indexed by $i \in I$. For each of these $aircraft[i]$, $aircraft[i].t_takeoff$, $aircraft[i].loc_takeoff$, $aircraft[i].loc_landing$ are set in the initial state of $RealWorld$. The values of $aircraft[i].loc_takeoff$ and $aircraft[i].loc_landing$ must be in R. The current location, $aircraft[i].loc$ is initialized to loc_NULL.

The most important behavior of $RealWorld$ is the motion of the aircraft. Upon initialization of the automaton, for each $i \in I$, the value of $aircraft[i].loc$ is set to the constant loc_NULL.

At $now = aircraft[i].t_takeoff$ an internal $\mathsf{Takeoff}_i$ action occurs, and $aircraft[i].loc$'s value changes to $aircraft[i].loc_takeoff$. The value of $aircraft[i].loc$ then changes nondeterministically with a continuous, bounded derivative in $\mathbb{R}^3$, such that $|\frac{d(aircraft[i].loc)}{dt}| \leq v_max$.

Upon receiving a valid $\mathsf{move}(p,t)_i$ action, $aircraft[i].moving$ is set to $true$, $aircraft[i].movingto$ is set to p, and $aircraft[i].movingtime$ is set to $now + t$. The aircraft i then moves to p within time t, resuming its normal, nodeterministic movement after reaching p.

When $aircraft[i].loc = aircraft[i].loc_landing$, we consider i to have left the system, and an internal $\mathsf{Landing}_i$ action occurs. This action changes $aircraft[i].loc$ to loc_NULL and the trajectory which changes the value of $aircraft[i].loc$ halts. The value of $aircraft[i].loc$ does not change again.

3.5 The Physical Layer

I define the Physical Layer of our system, $PLayer(R, \epsilon, v_max, d, r)$, depicted in Figure 3-1 at the beginning of this chapter, to be the composition of these components: a single $Bcast(d, r)$ TIOA, a single $RealWorld(\epsilon, v_max)$ TIOA, and n physical node TIOAs which comprise the set A.

3.6 Chapter Summary

In this chapter I have:

- Enumerated the capabilities of the aircraft that will serve as physical nodes in our air traffic control system.
- Discussed the communications service $Bcast$, while making simplifying assumptions about its behavior.

- Described the $RealWorld$ automaton, which models the physical state and motion of the world that the system operates in.

- Defined the Physical Network Layer, the lowest level layer in the air traffic control system.

Chapter 4

The Continuous VSA Layer

The motivation behind the Continuous VSA Layer, or just the VSA Layer, is to use the volatile mobile network described in the previous section to simulate a number of virtual stationary automata, or VSAs, at various locations in space. We can think of these VSAs as *virtual machines* which are able to perform computations and store data, and are located in a stable location. Therefore, the VSA Layer provides us a stable platform for implementing applications that can rely on the locations of the VSAs.

In reality, of course, these VSAs are not present in the world, but instead they are emulated by a number of nearby physical nodes. This emulation is a protocol to ensure that the VSA remains active during both small and large changes in the network topology.

A number of previous models for the VSA have been previously developed. A theoretical in-depth treatment has been described in ”Timed Virtual Stationary Automata for Mobile Networks” [3], and more recently, a Python-based implementation has been described in ”The Virtual Node Layer: A Programming Abstraction for Wireless Sensor Networks.” [13]. Other papers involving algorithms using the VSA also provide appropriate models for the VSA Layer [7, 8].

The VSA Layer will be specified in this chapter, including the architecture and the interactions of the layer. I will then describe some of the details of the VSA Layer's emulation, but this discussion will forgo some of the complex theoretical concerns of this emulation. Additional details of the emulation, if desired by the reader, can be found in Dolev, Gilbert, Lahiani, Lynch, and Nolte [3].

Aside from the section on emulation, this chapter is concerned with specifying the details

of each component of the VSA Layer, and putting them together as the formal definition for the VSA Layer. I begin each section with an overview of the component, including some motivation for decisions I make within. Then, formal definitions for each component are specified at the end of each section.

4.1 Regions

4.1.1 Overview

In order for the VSA Layer to work, the area that our network inhabits must be divided up into *regions*. A single VSA is contained within each region. Each region also has a set of *neighbors*, which are the nearby regions that the VSA in the region is able to communicate with.

The size of the regions and the set of neighbors for a region are chosen in order to satisfy the requirement that no matter where in a region a client is, its broadcast is able to reach every other client in the same region, and every other client in the region's neighbors. In general, this might mean that one region could be directly adjacent to another, yet not be considered a neighbor, if there are parts of the two regions that are not able to communicate with each other.

We choose the size and layout of the regions in our system to satisfy three requirements:

1. Each region's border is shared with another region (there are no regions that have a border that does not have another region on the other side of it) unless that region is on the edge of the system.

2. Every region's neighbors are exactly the regions that share a border with it.

3. The maximal distance between any point in any region and any point in one of its neighbors is less than or equal to a communications range r parameter for the Continuous VSA Layer, $CVLayer(d, r, \epsilon, v_max)$.

Adhering to these requirements will result in our space being partitioned into regions, within which a physical node is able to communicate with all other physical nodes in both its own region and neighboring regions.

4.1.2 Definitions

The parameter $r \in \mathbb{R}^+$ is the communications range within the layer.

The set J represents the region names, as defined in the global constants of Section 2.5.

The function $region : R \rightarrow J$ maps a point in R to its region name. Each point in R has exactly one region name associated with it. A ***region*** is a subset S of R such that for some $j \in J$, $S = \{p \in R | region(p) = j\}$. I assume that each region is connected.

The function $inRegion : J, \mathbb{R}^3 \rightarrow true, false$ is a predicate that returns $true$ if a point in $\mathbb{R}^3$ is in the appropriate region in J, and $false$ otherwise. This function satisfies the condition that $region(p) = j \leftrightarrow inRegion(j,p) = true$.

The function $nbrs : J \rightarrow 2^J$ returns, on input j, the set of j's neighbors. It is required that for all $j, j' \in J$, if $j' \in nbrs(j)$, then $j \in nbrs(j')$.

The following constraints are required for all regions $j \in J$:

- Every region's neighbors are exactly the regions that share a border with it.
- For every point p such that $region(p) = j$, and all p' such that $region(p') \in \{j\} \cup nbrs(j)$, $|p - p'| \leq r$.

4.2 Architecture of the VSA Layer

The VSA Layer $CVLayer(d, r, \epsilon, v_max)$ contains two sets of timed input-output automata[19]: the *VSAs*, fixed in a specific location in space, and the ***clients***, which are mobile. Also, like the physical network layer, the VSA layer has a timed input-output automaton which represents the real world and another timed input-output automaton which represents broadcast communications. In the VSA Layer, though, the broadcast communications automaton only allows a client to send messages to the VSA in its region, and receive messages from that same VSA. It does not allow for direct client-to-client communications. The VSA Layer also adds a timed input-output automaton to represent the communication between neighboring VSAs.

In many ways, the VSA layer looks similar to the network layer described in the previous chapter. The VSA layer includes the set $A = \{a_1, a_2, ..., a_n\}$ of aircraft - timed input-output automata that function as our clients; a timed input-output automaton $RealWorld$, which rep-

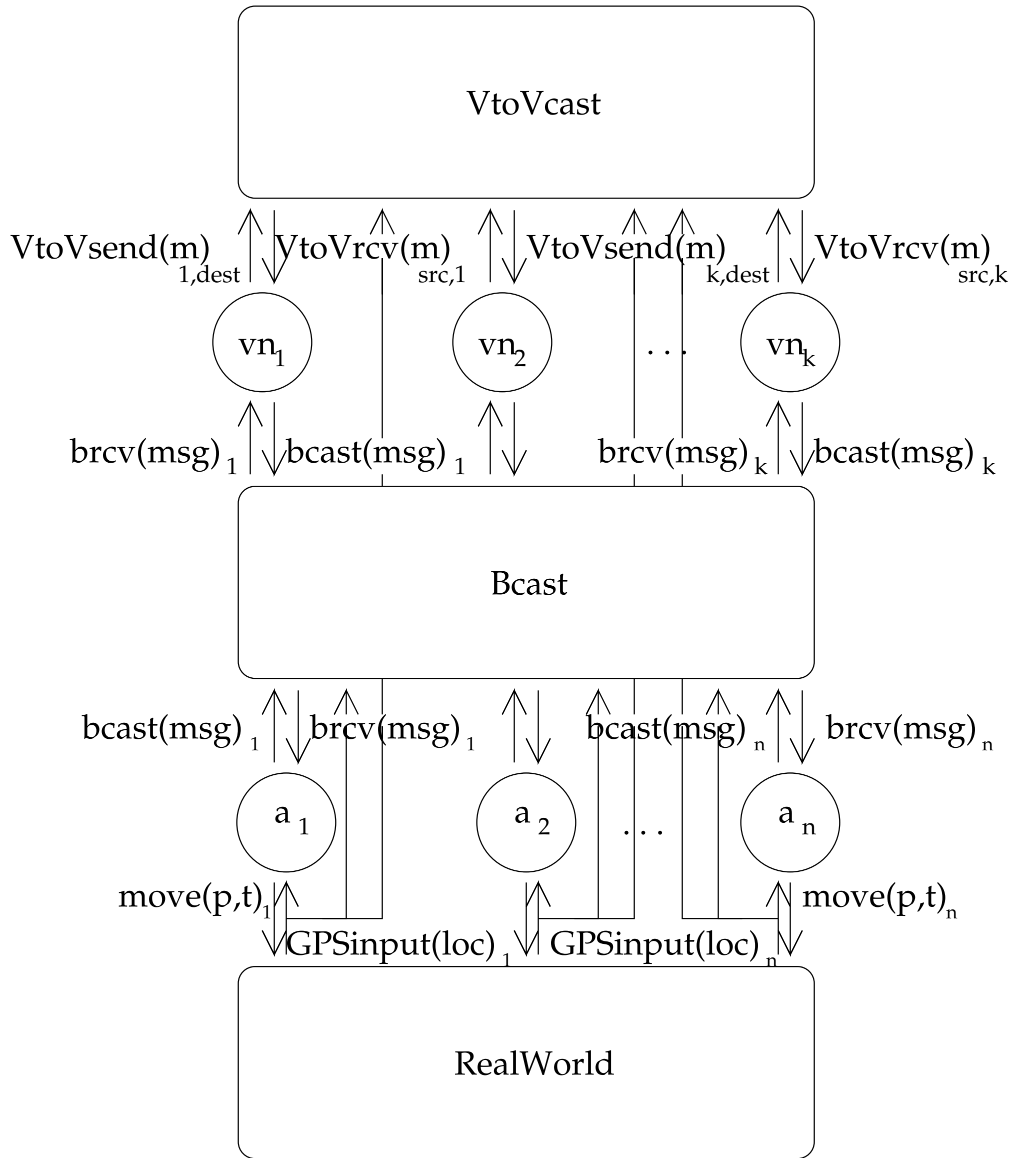


Figure 4-1: Architecture of the VSA Layer, for $I = \{1, 2, ..., n\}$ and $J = \{1, 2, ..., k\}$

Client Automaton a_i, for $i \in I$	Virtual Stationary Automaton vn_j, for $j \in J$
Signature:	**Signature**:
Input brcv$(msg)_i$, $msg \in Msg$	**Input** brcv$(msg)_j$, $msg \in Msg$
Input GPSinput$(loc)_i$, $loc \in R$	**Input** VtoVrcv$(msg)_{src,j}$, $msg \in Msg$
Output bcast$(msg)_i$, $msg \in Msg$	**Output** bcast$(msg)_j$, $msg \in Msg$
Output move$(p,t)_i$, $t \in R^{\geq 0}$, $p \in R$	**Output** VtoVsend$(msg)_{j,dest}$, $msg \in Msg$
State:	**State**:
$loc \in R$, the client's current location in space	$now : \mathbb{R}$, the current real time
$now : \mathbb{R}$, the current real time	

Figure 4-2: TIOA Signature for Clients and VSAs in the VSA Layer

resents the physical world that the aircraft operate in; and the timed input-output automaton $Bcast$, which represents the broadcast medium for clients and VSAs.

In the VSA layer, we also have a set $VN = \{vn_1, vn_2, ..., vn_k\}$ of Virtual Stationary Automata, which are also timed input-output automata. In addition, a timed input-output automaton $VtoVcast$ represents the communications service between neighboring VSAs.

A client a_i has an input GPSinput$(loc)_i$ from $RealWorld$ which functions as the aircraft's GPS oracle. The input occurs periodically with some small period ϵ.

A client a_i also has an output action bcast$(msg)_i$ which outputs a message to $Bcast$, and brcv$(msg)_i$ which receives a message from $Bcast$. The recipient of the message does not receive information on who the sender of the message was.

A VSA vn_j has an output action bcast$(msg)_j$ which outputs a message to $Bcast$, and brcv$(msg)_j$ which receives a message from $Bcast$.

The VSA also has an output action VtoVsend$(m)_{j,dest}$ which outputs a message to $VtoVcast$ with destination VSA vn_{dest}. Also, the VSA has an input action VtoVrcv$(m)_{src,j}$ which receives a message from $VtoVcast$ from source VSA vn_{src}. Note that vn_{dest} and vn_{src} must be neighbors of vn_j for $VtoVcast$ to output these actions to their destination.

4.3 Clients

4.3.1 Overview

Clients are the mobile processes in the VSA Layer. They correspond very closely to the physical nodes in the Network Layer, with similar capabilities and actions. A client can broadcast and receive messages, has access to a GPS oracle, and has an internal real-time clock which is synchronized with the other clients and Virtual Stationary Automata.

In the VSA Layer though, all outgoing messages from a client will be received only by a VSA. No messages are delivered to other clients. Conversely, a client only receives messages from a VSA, not from other clients.

4.3.2 Definitions

The set I represents the aircraft names that can be used to identify a unique aircraft.

For each $i \in I$, in the VSA Layer a_i represents a client. The set of all such a_i is called A. Each $a_i \in A$ is a timed input-output automaton with the following actions:

- The output $\mathsf{bcast}(msg)_i$ action outputs a message $msg \in Msg$ to $Bcast$.
- The input $\mathsf{brcv}(msg)_i$ action receives a message $msg \in Msg$ from $Bcast$.
- The input $\mathsf{GPSinput}(loc)_i$ action receives i's location $loc \in \mathbb{R}^3$ from $RealWorld$.
- The output $\mathsf{move}(p,t)_i$ action to $RealWorld$, with $t \in \mathbb{R} \geq 0$ and $p \in \mathbb{R}^3$, which will be used to control i, moving it to p within time t.

In addition to these actions, a_i may have other actions which must be internal, which are unspecified here.

The state of a_i includes:

- $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.
- $loc \in \mathbb{R}^3$, which is updated on each GPSinput.

In addition to these variables, a_i may have other state variables. Its transitions and trajectories are unspecified, except as noted above.

4.4 Virtual Stationary Automata

4.4.1 Overview

Virtual Stationary Automata are the primary new addition to the VSA Layer. They are machines fixed in specific locations in space, with the ability to send and receive messages, store data, and perform computations. They also have access to internal real-time clocks which are synchronized with the clients and other Virtual Stationary Automata.

As they are located at specific locations in space, VSAs are also associated with specific regions. Each VSA is associated with the region that contains its location. A VSA can broadcast messages to clients and receive messages from clients in the same region. A VSA is also able to send and receive messages from VSAs in neighboring regions.

4.4.2 Definitions

Recall, from earlier in the chapter, the set J, which represents the region names that can be used to identify the unique VSA in each region.

For each $j \in J$, vn_j represents a Virtual Stationary Automaton in the VSA Layer. The set of all such vn_j is called VN.

Each $vn_j \in VN$ is a timed input-output automaton with the following actions:

- The output $\mathsf{bcast}(msg)_j$ action outputs a message $msg \in Msg$ to $Bcast$.
- The output $\mathsf{VtoVsend}(m)_{j,dest}$ action outputs a message $m \in Msg$ to $VtoVcast$ intended for destination vn_{dest}.
- The input $\mathsf{brcv}(msg)_j$ action receives a message $msg \in Msg$ from $Bcast$.
- The input $\mathsf{VtoVrcv}(m)_{src,j}$ action receives a message $m \in Msg$ from $VtoVcast$ which was originally sent by source vn_{src}.
- The input $\mathsf{VtoVrcv}(m)_{\emptyset,j}$ action, with $m \in \{STARTUP, SHUTDOWN\}$, receives a special startup or shutdown message from $VtoVcast$.

In addition to these actions, vn_j may have other actions which must be internal, which are unspecified here.

The state of vn_j includes a real time clock, $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.

In addition to this variable, vn_j may have other state variables.

When vn_j receives a VtoVrcv$(SHUTDOWN)_{\emptyset,j}$ input, it stops all outputs. All state variables except for now are set to null, and no trajectories change values of any variables except for now. The automaton ignores all inputs until a VtoVrcv$(STARTUP)_{\emptyset,j}$ input is received, at which point all state variables except now are reset to their initial values, and trajectories resume their changes to variables, as if the VSA were just initialized.

Its transitions and trajectories are unspecified, except as noted above.

4.5 Intraregional Communication - Bcast

4.5.1 Overview

A client is able to communicate only with a VSA in its region, not with other clients in the region, clients in neighboring regions, or VSAs in neighboring regions. It communicates with the VSA through the $Bcast$ channel. Conversely, a VSA is able to communicate with all clients in its region through broadcasting messages on the $Bcast$ channel.

The $Bcast$ automaton filters messages sent through it to prevent disallowed communications. The parameter d represents message delay in the VSA layer's $Bcast$.

4.5.2 Definitions

The timed input-output automaton $Bcast(d)$ represents the intraregional communications network in the VSA Layer. For each client a_i or VSA vn_j, with $i \in I$ and $j \in J$, $Bcast$ has the following actions:

- The input bcast$(msg)_i$ action, which receives a message $msg \in Msg$ from a_i and the input bcast$(msg)_j$ action, which receives a message $msg \in Msg$ from vn_j.
- The output brcv$(msg)_i$ action, which sends a message $msg \in Msg$ to a_i and the output brcv$(msg)_j$ action, which sends a message $msg \in Msg$ to vn_j.
- The input GPSinput$(loc)_i$ from $RealWorld$, with $loc \in \mathbb{R}^3$.

The parameters of $Bcast$ are:

- d, the message delay between inputting a message and outputting it to the appropriate clients or VSA.

$Bcast$ contains a state variable $locs$, which is an array of locations in R, indexed by $i \in I$.

The behavior of $Bcast$ is as follows:

Upon inputting $\mathsf{bcast}(msg)_i$ from client a_i, within time d, $Bcast$ will output $\mathsf{brcv}(msg)_j$ to the VSA vn_j which is associated with the region j such that at the time $Bcast$ received the bcast input, $region(locs[i]) = j$. No other outputs will occur.

Upon inputting $\mathsf{bcast}(msg)_j$ from VSA vn_j, within time d, $Bcast$ will output $\mathsf{brcv}(msg)_k$ to all clients a_k such that at the time $Bcast$ received the bcast input, $region(locs[k]) = j$. No outputs will occur to clients outside of region j.

Upon inputting $\mathsf{GPSinput}(loc)_i$ from $RealWorld$, $locs[i] \in \mathbb{R}^3$ in $Bcast$ gets updated to loc.

Other requirements for the behavior of $Bcast$ include:

- Message Ordering: If a $\mathsf{bcast}(msg1)_i$ event precedes a $\mathsf{bcast}(msg2)_j$ event, and if $\mathsf{brcv}(msg1)_k$ and $\mathsf{brcv}(msg2)_k$ both occur, then the $\mathsf{brcv}(msg1)_k$ event precedes the $\mathsf{brcv}(msg2)_k$ event.

- Message Integrity: Each output $\mathsf{brcv}(msg)_j$ must have been preceded by an input action $\mathsf{bcast}(msg)_i$ such that msg in the two actions is the same and the $\mathsf{brcv}(msg)_j$ occurs at most d time after the $\mathsf{bcast}(msg)_i$ action.

- Message Uniqueness: No two $\mathsf{brcv}(msg)_i$ events occur for the same msg and the same i.

4.6 Interregional Communication - VtoVcast

4.6.1 Overview

A VSA is able to communicate with any specific neighboring VSA through the $VtoVcast$ channel. This channel enables point-to-point communication between VSAs, as opposed to the broadcast communication performed by $Bcast$. While the VSA sends a message to a specific neighboring VSA, it has no way to know if that VSA is active or not, and it receives no specific acknowledgment that the message has been received.

The $VtoVcast$ automaton passes messages sent through it to the appropriate VSA. Like the $Bcast$ channel, a message delay parameter d is present in $VtoVcast$. Due to the requirements of the network layer and the region layout, if a neighboring VSA is active, the message will always be received.

The $VtoVcast$ automaton also starts up each VSA when a client enters that VSA's region, and shuts the VSA down when all clients leave that region. It keeps a state variable $status$, with an entry for each VSA representing whether that VSA is active (has clients in its region), or inactive (has no clients in its region).

4.6.2 Definitions

The timed input-output automaton $VtoVcast(d)$ represents the interregional communications network in the VSA Layer. For VSA vn_i and VSA vn_j, with $i, j \in J$, and $j \in nbrs(i)$, $VtoVcast$ has the following actions:

- The input $\mathsf{VtoVsend}(m)_{i,j}$ action receives a message $m \in Msg$ from vn_i intended for vn_j.
- The output $\mathsf{VtoVrcv}(m)_{j,i}$ action outputs a message $m \in Msg$ to vn_i that was originally sent by vn_j.

Other actions in $VtoVcast$ include:

- The output $\mathsf{VtoVrcv}(m)_{\emptyset,j}$ action outputs a message $m \in \{STARTUP, SHUTDOWN\}$ to vn_j. This form of the action is not subject to the Message Integrity or Message Uniqueness requirements to be given later in this section.
- The input $\mathsf{GPSinput}(loc)_i$ from $RealWorld$, with $loc \in \mathbb{R}^3$, for each $i \in I$.

The parameters of $VtoVcast$ are:

- d, the message delay between inputting a message and outputting it to the appropriate VSA.

$VtoVcast$ contains a state variable $locs$, which is an array of locations, indexed by $i \in I$. This variable gets updated upon each $\mathsf{GPSinput}(loc)_i$ input, by setting $locs[i]$ to loc.

$VtoVcast$ also contains a state variable $status$, which is an array indexed by $j \in J$ with each element in the set $\{ACTIVE, INACTIVE\}$. This array is initialized to $INACTIVE$ for all j, and is updated upon VtoVrcv$(STARTUP)_{\emptyset,j}$ and VtoVrcv$(SHUTDOWN)_{\emptyset,j}$ actions.

The behavior of $VtoVcast$ is as follows. Upon inputting VtoVsend$(m)_{i,j}$ from VSA vn_i, within time d, $VtoVcast$ will output VtoVrcv$(m)_{i,j}$ to vn_j if and only if $status[j] = ACTIVE$ at the time of the VtoVsend input.

Also, for all $j \in J$, whenever $status[j] = INACTIVE$ and there is some $i \in I$ such that $region(locs[i]) = j$, $VtoVcast$ will, within time d, output a VtoVrcv$(STARTUP)_{\emptyset,j}$ to vn_j and set the value of $status[j]$ to $ACTIVE$.

Similarly, whenever $status[j] = ACTIVE$ and for all $i \in I$, $inRegion(j, locs[i]) = false$, $VtoVcast$ will, within time d, output a VtoVrcv$(SHUTDOWN)_{\emptyset,j}$ to vn_j and set the value of $status[j]$ to $INACTIVE$.

Other requirements for the behavior of $VtoVcast$ include:

- Message Ordering: If a VtoVsend$(msg1)_{i,k}$ event precedes a VtoVsend$(msg2)_{j,k}$ event, and if VtoVrcv$(msg1)_{i,k}$ and VtoVrcv$(msg2)_{j,k}$ outputs both occur, then the VtoVrcv$(msg1)_{i,k}$ event precedes the VtoVrcv$(msg2)_{j,k}$ event.

- Message Integrity: Each output VtoVrcv$(msg)_{i,j}$ must have been preceded by an input action VtoVsend$(msg)_{i,j}$ such that msg in the two actions is the same and the VtoVrcv$(msg)_{i,j}$ occurs at most d time after the VtoVsend$(msg)_{i,j}$ action.

- Message Uniqueness: No two VtoVrcv$(msg)_{j,i}$ events occur for the same msg and the same i.

4.7 RealWorld

4.7.1 Overview

The $RealWorld$ timed input-output automaton is the same in the VSA Layer as it is in the Network Layer.

4.8 The Continuous VSA Layer

I define the Continuous VSA Layer of our system, $CVLayer(d, r, \epsilon, v_max)$ as pictured in Figure 4-1 at the beginning of the chapter, to be the composition of these components: a single $Bcast(d)$ TIOA, a single $VtoVcast(d)$ TIOA, a single $RealWorld(\epsilon, v_max)$ TIOA, the client TIOAs a_i for $i \in I$ comprising the set A, and the Virtual Stationary Automata TIOAs vn_j for $j \in J$ comprising the set VN.

4.9 Emulation of the VSA Layer

The VSA Layer is emulated by the physical layer of physical nodes and the wireless network that they operate. This section will specify some of the details of how this emulation is performed. As stated earlier, more details on the emulation of VSAs may be found in Dolev, Gilbert, Lahiani, Lynch, and Nolte [3].

4.9.1 Emulation of the Clients

Emulation of clients is trivial. Each physical node corresponds to a client. A physical node uses its networking capabilities to emulate the communications channels specified in the VSA layer by broadcasting messages for the client and delivering to the client only messages the client is able to receive. The physical node's real-time clock and GPS input messages are simply delivered to the client.

We can therefore think of each physical node a_i as a composition of the corresponding client and some part of the emulation of a VSA.

4.9.2 Emulation of the VSAs

Again, this thesis is not concerned with implementation-level details, but the emulation of stable Virtual Stationary Automata using physical nodes is not trivial, and therefore requires some discussion.

The emulation of the VSA is carried out by the physical nodes in the VSA's region. A local copy of the state of the VSA is kept on some number of physical nodes (possibly all of them) that

are in the VSA's region. While this may seem wasteful, it is very important for the VSA to not lose information due to network volatility, and if many physical nodes have a replicated copy of the VSA, as long as one of them remains, the emulation can continue.

We designate one physical node to be the *leader*. This physical node is in charge of giving any new physical nodes that enter the region a copy of the VSA state so they can maintain their local copy when new messages are received. While we assume our messaging is reliable, the leader can also provide message ordering and consistency for the other physical nodes in the region, making sure that every physical node has the same copy of the VSA state.

We can elect this leader in a number of ways, ranging from the simple (the first physical node is the first leader, and then some random physical node becomes leader when the old one leaves) to the more intelligent (choosing a leader based on position and motion, so that we can minimize the number of times leadership needs to be transfered). In any situation when the leader leaves the region or fails, any other physical node in the region is able to resume the emulation with minimal interruption. By keeping track messages that are received while waiting for a new leader to be appointed we can ensure that even with no leader present, the VSA's behavior will be able to be emulated by some physical node. Therefore, aside for some short startup time, we can assume that the VSA in a region will be active unless there are no physical nodes in the region to emulate it. This process is emulated with a specific leader election subroutine running on the physical nodes.

The clock in a VSA is also maintained by the physical nodes. Since we assume the clocks of each physical node are synchronized, we can allow the leader to use his clock as the VSA's clock. When there is no leader for some period of time, a catch-up mechanism synchronizes the clocks and deals with actions that occur during the period without a leader.

While there is much more to be said about emulating a VSA, further details are not extremely important to understanding the algorithms in this thesis. Interested readers are advised to consult the works cited in the beginning of this chapter [3, 7, 8, 13].

4.10 Chapter Summary

In this chapter I have:

- Specified how our deployment space is divided up into regions, specifying the way to identify those regions.
- Enumerated and defined the capabilities of the clients and the VSAs in the VSA Layer.
- Presented a new version of the communications service $Bcast$, as well as a new virtual communications service $VtoVcast$.
- Explained how the VSA layer is emulated by the physical nodes.
- Provided the appropriate background to understand how the algorithms in the following chapters work.

Chapter 5

The Known Path Model

Air Traffic Control is a real world system which behaves in a continuous manner. For the purposes of performing the duties of ATC algorithmically, I would rather think of the world using a more discrete representation. Therefore, the goal of this chapter is to abstract the continuous motion model from the previous chapters with a discrete graph model that will be used in the following chapters' algorithms.

Let us assume that we have access to a map of allowable air routes, which we can use to navigate each aircraft through each sector. This is an assumption based on the real world ATC system: using Letters of Agreement between sectors and predefined airways that are used within each sector, we have available to us a map of *paths* connecting all airports in the system.

I model the real world as this map in two steps. The first step is to partition areas of airspace into *bounding boxes* and *bounding cylinders*. Partitioning the airspace into bounded areas will allow us to require aircraft to remain within a specific boundary. If the bounding areas are separated appropriately, the problem of keeping aircraft safely separated reduces to the problem of making sure only one aircraft is within a bounding area at any given time.

The second step is to arrange the bounding areas into the predetermined map of air routes. Since the boundaries allow us to ignore the exact location of an aircraft, representing the aircraft's position by only the boundary it is within, we can arrange the boundaries into a connected, directed graph. Then I can represent an aircraft's location in space by a single edge in the graph, and can use graph-theoretical algorithms on the VSAs to implement the air traffic control system.

But first, I present a modified version of the generic VSA layer described in the last chapter, which I call the Known Path VSA Layer $KnownPathVL(G, \epsilon, d, t_advance)$. While most aspects of the layer remain the same, I replace the $RealWorld$ automaton with a $GraphRealWorld$ automaton which represents the real world with aircraft on edges of a graph instead of aircraft moving continuously through space.

I then present a method for using the given map of air routes to abstract $RealWorld$ into $GraphRealWorld$. Using that, I prove that, under a set of assumptions, we can use $RealWorld$ to emulate $GraphRealWorld$, and that doing so allows us to safely consider $GraphRealWorld$ as the representation of the system.

5.1 Architecture of the Known Path VSA Layer

5.1.1 Overview

The architecture that I will be using for my air traffic control algorithms is nearly identical to the Continuous VSA Layer described in Chapter 4. Each component, while sharing the name of its counterpart in the VSA Layer, has been modified slightly with changed inputs and different data types for some state variables. As an overview, the layer is changed in these ways:

1. I replace the $RealWorld$ automaton with a $GraphRealWorld$ automaton, the specifics of which are presented in Section 5.3.

2. A client a_i now has an input $\mathsf{GPSinput}(j, e)_i$ from $GraphRealWorld$, where $j \in J$ and $e \in E$, where E is the set of edges of the graph which will be described in this chapter. This input replaces the $\mathsf{GPSinput}(loc)_i$ input from $RealWorld$ that was described in Chapter 4. The $\mathsf{GPSinput}(j, e)_i$ output from $GraphRealWorld$ still is input by a_i, $Bcast$, and $VtoVcast$ as in Chapter 4.

3. In all automata in which locations of clients are stored, the old representation of a location as a point in $\mathbb{R}^3$ is replaced by the representation of a location as a region and an edge. This allows the communications automata to determine the region of a client directly from the stored GPS input, instead of mapping the client's location to a region, as was done in the previous layers.

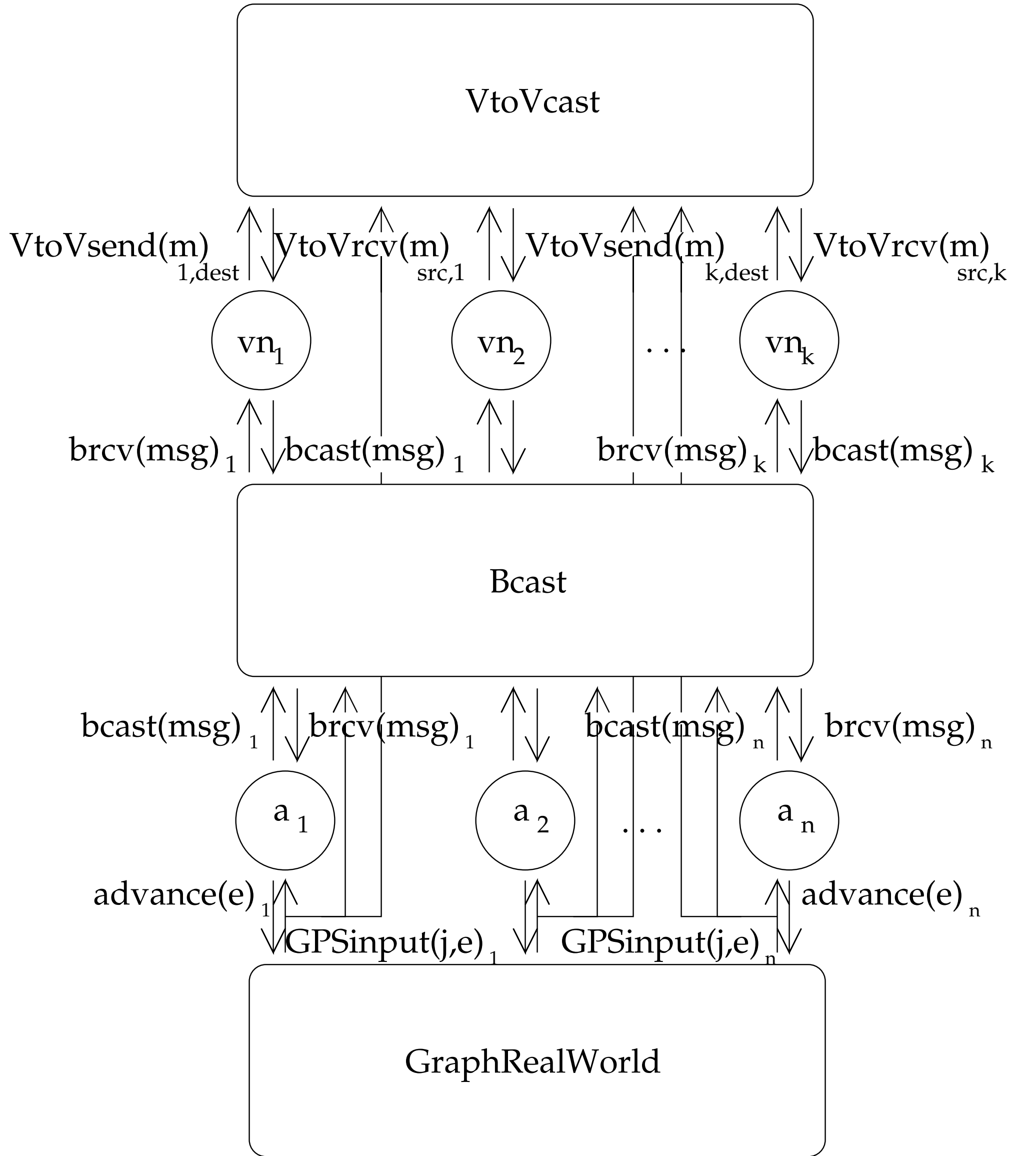


Figure 5-1: Architecture of the Known Path VSA Layer, for $I = \{1, 2, ..., n\}$, $J = \{1, 2, ...k\}$

4. The input move$(p, t)_i$ action from a_i to $RealWorld$ is replaced with an advance$(e)_i$ action to $GraphRealWorld$, with e being an edge in the graph representation. The functionality is similar, in that a move action moves an aircraft to a point $p \in R$, while an advance action moves an aircraft to $e \in E$.

Parameters of the layer are the graph representation $G = (V, E)$, the GPS update period ϵ, the message delay d, and the time required for an advance action $t_advance$.

While no additional outputs are added to $GraphRealWorld$, we would like the clients and VSAs to have access to the actual graph representation of the real world, so that they may determine the layout of the edges and vertices. I therefore provide a number of functions that allow clients and VSAs to access the graph representation. I include this representation as a parameter of the Known Path VSA Layer.

5.2 Graph Representation and Regions

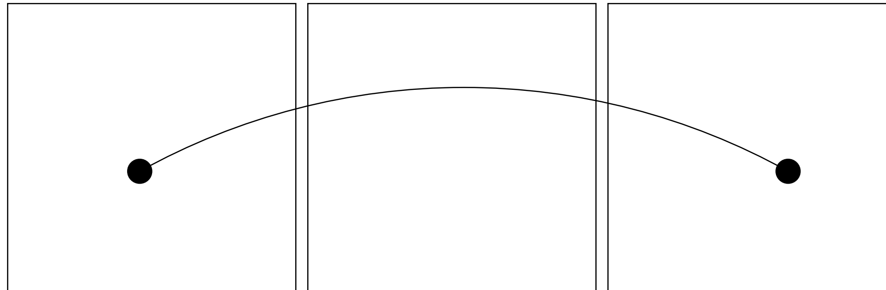
5.2.1 Overview

The first step in specifying $GraphRealWorld$ is to specify the graph representation that the system will use. This representation will be accessible by all automata in the Known Path VSA Layer, as it is a parameter of the layer.

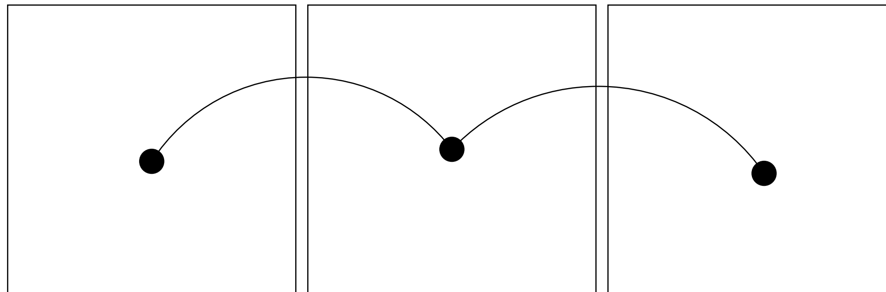
We model the map of routes as a directed graph $G = (V, E)$. A subset of vertices $T \subseteq V$ represents the terminal vertices, source and exit points for aircraft in the system. These vertices are connected by directed edges in E to other vertices in V. The graph is strongly connected, so each vertex is reachable from any other vertex in the graph.

The definition of *regions* must also be changed in order to use this graph representation as the representation of the world in $GraphRealWorld$. In Chapter 4, I defined a region as a connected subset of R. For the Known Path VSA Layer, a region is a subset of V that may not be connected. Later in this chapter, you will see more of how this definition connects to the continuous definition of regions. These regions also have a set of neighbors, and draw region names from the set J.

Using this definition of regions, we can place a constraint on the graph's layout. This constraint is that for any two vertices v and v' connected by an edge $e = (v, v') \in E$, the two vertices



(a) Three regions and two vertices that are incompatible with the model.



(b) Three regions and three vertices that are compatible with the model.

Figure 5-2: A Constraint on the Layout of Vertices and Regions

are either in the same region, or in neighboring regions. For a graphical representation of this constraint, see Figure 5-2, which shows an invalid placement in 5-2(a), and a valid placement in 5-2(b).

This constraint is important for our air traffic control algorithms due to the model for communication between clients and VSAs. Since a VSA can only communicate with a neighboring VSA, it is imperative that an edge's endpoints are in neighboring regions. You will see more on why this is needed while discussing the FIFO Algorithm in Chapter 6.

5.2.2 Definitions

We define the graph representation of the system to be the strongly connected directed graph $G = (V, E)$.

The source and exit points of the graph are the set of terminal vertices $T \subseteq V$.

We also define a number of functions exposing information about the graph:

- $getRegion : V \rightarrow J$, mapping a vertex $v \in V$ to its region name $j \in J$.

- $getRegions : E \rightarrow 2^J$, mapping an edge $(v, v') \in E$ to $\{j, j'\}$ with $j, j' \in J$ such that $getRegion(v) = j$ and $getRegion(v') = j'$. This function is constrained so that for all $(v, v') \in E$, $getRegion(v') \in \{getRegion(v)\} \cup nbrs(getRegion(v))$.

- $vertexAfter : E \rightarrow V$, mapping an edge $(v, v') \in E$ to its target vertex v'.

5.3 GraphRealWorld

5.3.1 Overview

Now that the graph representation has been defined, I can use that to define the timed input-output automaton $GraphRealWorld$, our new discrete model for the physical behavior of the system.

5.3.2 Definitions

A timed input-output automaton $GraphRealWorld(\epsilon, t_advance)$ represents the discretized physical state of the real world in the Known Path VSA Layer. Code for this TIOA is presented in Figure 5-3.

The parameter $\epsilon \in \mathbb{R} \geq 0$ is the GPS update period, it is assumed to be small. The parameter $t_\delta \in \mathbb{R} \geq 0$ is the maximum transition time that an aircraft takes to move from one edge to the next.

The constant e_NULL represents a dummy edge outside the set E. It is the initial location of an aircraft before it enters the system.

The constant e_DONE represents a dummy edge outside the set E. It is the location of an aircraft after it has left the system.

The constant r_NULL represents a dummy region outside the set J. It is the initial region of an aircraft before it enters the system.

The data type $GraphAircraftRecord$ is a record with fields:

- $t_takeoff \in \mathbb{R} \geq 0$, the time the aircraft enters the system.

- $v_takeoff \in V$, the vertex at which the aircraft enters the system.

```
Constants:
  ε , the GPS update frequency
        ∈ ℝ ≥ 0
  t_advance , the time between an advance action
        and the corresponding state update
        ∈ ℝ ≥ 0

Signature:
  Output GPSinput(j, e)_i , i ∈ I, j ∈ J, e ∈ E
  Input advance(e)_i , i ∈ I, e ∈ E
  Internal doAdvance_i , i ∈ I
  Internal Takeoff(e)_i , i ∈ I, e ∈ E
  Internal Landing_i , i ∈ I

State:
  GPSdone, an array of booleans for each i ∈ I
        and for k ∈ {0, 1, 2, 3...}
        initialized to false
  aircraft, an array of type GraphAircraftRecord
      for each aircraft[i], initialized to:
      aircraft[i].e ← e_NULL
      aircraft[i].region ← j_NULL
      aircraft[i].auth ← false
      aircraft[i].authtime ← 0
      aircraft[i].authedge ← e_NULL
  now : ℝ

Trajectories:
  evolves
    d(now) = 1
    constant aircraft, GPSdone
  stops when
    ((aircraft[i].e = (v, aircraft[i].v_landing) ∨
      aircraft[i].auth = true) ∧
     now = aircraft[i].authtime]) ∨
    (now = kε ∧ GPSdone[i, k] = false)

Actions:
  Output GPSinput(j, e)_i
  Precondition:
    now = kε ∧
    GPSdone[i, k] = false ∧
    j = aircraft[i].region ∧
    e = aircraft[i].e
  Effect:
    GPSdone[i, k] ← true

  Input advance(e)_i
  Effect:
    (v, v′) ← e
    if (vertexAfter(aircraft[i].e) = v ∧
        aircraft[i].auth = false) then
      aircraft[i].auth ← true
      aircraft[i].authtime ← now + t_advance
      aircraft[i].authedge ← e

  Internal doAdvance_i
  Precondition:
    now ∈ [aircraft[i].authtime - t_advance,
            aircraft[i].authtime] ∧
    aircraft[i].auth = true
  Effect:
    aircraft[i].auth ← false
    aircraft[i].authtime ← now + t_advance
    aircraft[i].e ← aircraft[i].authedge
    tempv ← vertexAfter(aircraft[i].e)
    aircraft[i].region ← getRegion(tempv)
    aircraft[i].authedge ← e_NULL

  Internal Takeoff(e)_i
  Precondition:
    e = (aircraft[i].v_-takeoff, v) ∧
    now ≥ aircraft[i].t_-takeoff ∧
    aircraft[i].e = e_NULL ∧
    ∀k, aircraft[k].e ≠ e
  Effect:
    aircraft[i].e ← e
    aircraft[i].region ← getRegion(v)

  Internal Landing_i
  Precondition:
    aircraft[i].e = (v, aircraft[i].v_landing) ∧
    now ∈ [aircraft[i].authtime - t_advance,
            aircraft[i].authtime]
  Effect:
    aircraft[i].e ← e_DONE
```

Figure 5-3: TIOA Code for GraphRealWorld

- $v_landing \in V$, the vertex at which the aircraft leaves the system.

- $e \in E \cup \{e_NULL, e_DONE\}$, the current location of the aircraft.

- $region \in J \cup \{j_NULL\}$, the current region of the aircraft.

- $auth$, a boolean representing whether the control software allows the aircraft to move at this time.

- $authtime \in \mathbb{R}$, the time an authorization ends.

- $authedge \in E \cup \{e_NULL, e_DONE\}$, the edge that the aircraft was authorized to move to.

$GraphRealWorld$ has a single output action, which is periodic with period ϵ, as well as a single input action:

- The output GPSinput$(j, e)_i$ action, where $j = aircraft[i].region$ and $e = aircraft[i].e$. The output is received by a_i, $Bcast$, and $VtoVcast$.

- The input advance$(e)_i$ action from a_i, with $e \in E$, which represents the air traffic control software telling $GraphRealWorld$ that $aircraft[i]$ is allowed to make a transition to e.

This output occurs for each $i \in I$ at $now = 0, \epsilon, 2\epsilon, ...$

The state of $GraphRealWorld$ includes:

- $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.

- $aircraft$, an array of $GraphAircraftRecord$s, indexed by $i \in I$. For each of these $aircraft[i]$, $aircraft[i].t_takeoff$, $aircraft[i].v_takeoff$, $aircraft[i].v_landing$ are set in the initial state of $GraphRealWorld$. The values of $aircraft[i].v_takeoff$ and $aircraft[i].v_landing$ must be in T. The current location, $aircraft[i].e$ is initialized to e_NULL, the current region, $aircraft[i].region$ is initialized to j_NULL, and $aircraft[i].auth$ is initialized to $false$.

The most important behavior of $GraphRealWorld$ is the motion of the aircraft. Upon initialization of the automaton, for each $i \in I$, the value of $aircraft[i].e$ is set to e_NULL and the value of $aircraft[i].region$ is set to j_NULL.

At $now = aircraft[i].t_takeoff$ an internal $\mathsf{Takeoff}(e)_i$ action occurs, the value of $aircraft[i].e$ changes to $e = (aircraft[i].v_takeoff, v) \in E$, and $aircraft[i].region$'s value changes to the value of $region(v)$.

Upon $GraphRealWorld$ receiving an $\mathsf{advance}(e)_i$ input from a_i, $aircraft[i].auth$ will be set to true. Then, within $t_advance$, the values of $aircraft[i].e$, $aircraft[i].region$, and $aircraft[i].auth$ make a transition if $vertexAfter(aircraft[i].e) = v$ for $(v, v') = e$:

- $aircraft[i].e = (v, v') \in E$ changes to $aircraft[i].e = (v', v'') = e$.
- $aircraft[i].region = j \in J$ changes to $aircraft[i].region \in \{j\} \cup nbrs(j)$.
- $aircraft[i].auth = true$ changes to $aircraft[i].auth = false$.

When $aircraft[i].e = (v, aircraft[i].v_landing)$, within $t_advance$ we consider i to have left the system, and an internal $\mathsf{Landing}_i$ action occurs. This action changes $aircraft[i].e$ to e_DONE and the value does not change again.

5.4 The Known Path VSA Layer

5.4.1 Overview

While much of the Continuous VSA Layer from Chapter 4 is left unchanged in the Known Path VSA Layer, small changes are necessary to each component in the layer. Since most of these changes simply involve changing the data type of a variable containing a location in R to one containing an edge in E or a region in J, significant discussion is not necessary, but the definitions are included in this section for completeness. At the beginning of each set of definitions, I explain the changes that were made to that component. At the end of the section, I formally define the Known Path VSA Layer.

5.4.2 Client Definitions

The changes to the clients are minor. Instead of receiving a GPSinput containing a location in 3-space, a client receives one containing its region and the edge that it is on. Where it would store its location, the client now stores $currentRegion$ and $currentEdge$.

For each $i \in I$, in the Known Path VSA Layer a_i represents a client. The set of all such a_i is called A. Each $a_i \in A$ is a timed input-output automaton with the following actions:

- The output bcast$(msg)_i$ action outputs a message $msg \in Msg$ to $Bcast$.
- The input brcv$(msg)_i$ action receives a message $msg \in Msg$ from $Bcast$.
- The input GPSinput$(j, e)_i$ action receives i's region $j \in J$ and current edge $e \in E$ from $GraphRealWorld$.
- The output advance$(e)_i$ action to $GraphRealWorld$, with $e \in E$ will be used to control i, moving it to e within time $t_advance$.

In addition to these actions, a_i may have other actions which must be internal, which are unspecified here.

The state of a_i includes:

- $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.
- $currentRegion \in J$ and $currentEdge \in E$, which are updated on each GPSinput.

In addition to these variables, a_i may have other state variables. Its transitions and trajectories are unspecified, except as noted above.

5.4.3 VSA Definitions

No changes are made to the VSAs between Chapters 4 and 5.

Recall the set J, which represents the region names that can be used to identify the unique VSA in each region.

For each $j \in J$, vn_j represents a Virtual Stationary Automaton in the Known Path VSA Layer. The set of all such vn_j is called VN.

Each $vn_j \in VN$ is a timed input-output automaton with the following actions:

- The output bcast$(msg)_j$ action outputs a message $msg \in Msg$ to $Bcast$.

- The output $\mathsf{VtoVsend}(m)_{j,dest}$ action outputs a message $m \in Msg$ to $VtoVcast$ intended for destination vn_{dest}.
- The input $\mathsf{brcv}(msg)_j$ action receives a message $msg \in Msg$ from $Bcast$.
- The input $\mathsf{VtoVrcv}(m)_{src,j}$ action receives a message $m \in Msg$ from $VtoVcast$ which was originally sent by source vn_{src}.
- The input $\mathsf{VtoVrcv}(m)_{\emptyset,j}$ action, with $m \in \{STARTUP, SHUTDOWN\}$, receives a special startup or shutdown message from $VtoVcast$.

In addition to these actions, vn_j may have other actions which must be internal, which are unspecified here.

The state of vn_j includes a real time clock, $now \in \mathbb{R} \geq 0$, which is initialized to 0 and increases with rate 1.

In addition to this variable, vn_j may have other state variables.

When vn_j receives a $\mathsf{VtoVrcv}(SHUTDOWN)_{\emptyset,j}$ input, it stops all outputs. All state variables except for now are set to null, and no trajectories change values of any variables except for now. The automaton ignores all inputs until a $\mathsf{VtoVrcv}(STARTUP)_{\emptyset,j}$ input is received, at which point all state variables except now are reset to their initial values, and trajectories resume their changes to variables, as if the VSA were just initialized.

Its transitions and trajectories are unspecified, except as noted above.

5.4.4 Bcast Definitions

In $Bcast$, I change the data type of a couple of components. Instead of receiving a $\mathsf{GPSinput}_i$ containing a location in 3-space, $Bcast$ receives one containing i's region and the edge that i is on. It stores the regions for each i in the array reg instead of storing locations in the array $locs$.

The timed input-output automaton $Bcast(d)$ represents the intraregional communications network in the Known Path VSA Layer. For each client a_i or VSA vn_j, with $i \in I$ and $j \in J$, $Bcast$ has the following actions:

- The input $\mathsf{bcast}(msg)_i$ action, which receives a message $msg \in Msg$ from a_i and the input $\mathsf{bcast}(msg)_j$ action, which receives a message $msg \in Msg$ from vn_j.

- The output brcv$(msg)_i$ action, which sends a message $msg \in Msg$ to a_i and the output brcv$(msg)_j$ action, which sends a message $msg \in Msg$ to vn_j.
- The input GPSinput$(j, e)_i$ from $RealWorld$, with $j \in J$ and $e \in E$.

The parameters of $Bcast$ are:

- d, the message delay between inputting a message and outputting it to the appropriate clients or VSA.

$Bcast$ contains a state variable reg, which is an array of regions in J, indexed by $i \in I$.

The behavior of $Bcast$ is as follows:

Upon inputting bcast$(msg)_i$ from client a_i, within time d, $Bcast$ will output brcv$(msg)_j$ to the VSA vn_j which is associated with the region j such that at the time $Bcast$ received the bcast input, $reg[i] = j$. No other outputs will occur.

Upon inputting bcast$(msg)_j$ from VSA vn_j, within time d, $Bcast$ will output brcv$(msg)_k$ to all clients a_k such that at the time $Bcast$ received the bcast input, $reg[k] = j$. No outputs will occur to clients outside of region j.

Upon inputting GPSinput$(j, e)_i$ from $RealWorld$, $reg[i]$ gets updated to j.

Other requirements for the behavior of $Bcast$ include:

- Message Ordering: If a bcast$(msg1)_i$ event precedes a bcast$(msg2)_j$ event, and if brcv$(msg1)_k$ and brcv$(msg2)_k$ both occur, then the brcv$(msg1)_k$ event precedes the brcv$(msg2)_k$ event.
- Message Integrity: Each output brcv$(msg)_j$ must have been preceded by an input action bcast$(msg)_i$ such that msg in the two actions is the same and the brcv$(msg)_j$ occurs at most d time after the bcast$(msg)_i$ action.
- Message Uniqueness: No two brcv$(msg)_i$ events occur for the same msg and the same i.

5.4.5 VtoVcast Definitions

In $VtoVcast$, I change the data type of a couple of components. Instead of receiving a GPSinput$_i$ containing a location in 3-space, $VtoVcast$ receives one containing i's region and the edge that i is on. It stores the regions for each i in the array reg instead of storing locations in the array $locs$.

The timed input-output automaton $VtoVcast(d)$ represents the interregional communications network in the Known Path VSA Layer. For VSA vn_i and VSA vn_j, with $i, j \in J$, and $j \in nbrs(i)$ $VtoVcast$ has the following actions:

- The input $\mathsf{VtoVsend}(m)_{i,j}$ action receives a message $m \in Msg$ from vn_i intended for vn_j.
- The output $\mathsf{VtoVrcv}(m)_{j,i}$ action outputs a message $m \in Msg$ to vn_i that was originally sent by vn_j.

Other actions in $VtoVcast$ include:

- The output $\mathsf{VtoVrcv}(m)_{\emptyset,j}$ action outputs a message $m \in \{STARTUP, SHUTDOWN\}$ to vn_j. This form of the action is not subject to the Message Integrity or Message Uniqueness requirements to be given later in this section.
- The input $\mathsf{GPSinput}(j, e)_i$ from $RealWorld$, with $j \in J$ and $e \in E$, for each $i \in I$.

The parameters of $VtoVcast$ are:

- d, the message delay between inputting a message and outputting it to the appropriate VSA.

$VtoVcast$ contains a state variable reg, which is an array of regions in J, indexed by $i \in I$. This variable gets updated upon each $\mathsf{GPSinput}(j, e)_i$ input, by setting $reg[i]$ to j.

$VtoVcast$ also contains a state variable $status$, which is an array indexed by $j \in J$ with each element in the set $\{ACTIVE, INACTIVE\}$. This array is initialized to $INACTIVE$ for all j, and is updated upon $\mathsf{VtoVrcv}(STARTUP)_{\emptyset,j}$ and $\mathsf{VtoVrcv}(SHUTDOWN)_{\emptyset,j}$ actions.

The behavior of $VtoVcast$ is as follows. Upon inputting $\mathsf{VtoVsend}(m)_{i,j}$ from VSA vn_i, within time d, $VtoVcast$ will output $\mathsf{VtoVrcv}(m)_{i,j}$ to vn_j if and only if $status[j] = ACTIVE$ at the time of the VtoVsend input.

Also, for all $j \in J$, whenever $status[j] = INACTIVE$ and there is some $i \in I$ such that $reg[i] = j$, $VtoVcast$ will, within time d, output a $\mathsf{VtoVrcv}(STARTUP)_{\emptyset,j}$ to vn_j and set the value of $status[j]$ to $ACTIVE$.

Similarly, whenever $status[j] = ACTIVE$ and for all $i \in I$, $reg[i] \neq j$, $VtoVcast$ will, within time d, output a $\mathsf{VtoVrcv}(SHUTDOWN)_{\emptyset,j}$ to vn_j and set the value of $status[j]$ to $INACTIVE$.

Other requirements for the behavior of $VtoVcast$ include:

- Message Ordering: If a VtoVsend$(msg1)_{i,k}$ event precedes a VtoVsend$(msg2)_{j,k}$ event, and if VtoVrcv$(msg1)_{i,k}$ and VtoVrcv$(msg2)_{j,k}$ outputs both occur, then the VtoVrcv$(msg1)_{i,k}$ event precedes the VtoVrcv$(msg2)_{j,k}$ event.

- Message Integrity: Each output VtoVrcv$(msg)_{i,j}$ must have been preceded by an input action VtoVsend$(msg)_{i,j}$ such that msg in the two actions is the same and the VtoVrcv$(msg)_{i,j}$ occurs at most d time after the VtoVsend$(msg)_{i,j}$ action.

- Message Uniqueness: No two VtoVrcv$(msg)_{j,i}$ events occur for the same msg and the same i.

5.4.6 Known Path VSA Layer Definition

I define the Known Path VSA Layer, or $KnownPathVL(G, \epsilon, d, t_advance)$ to be the composition of a $VtoVcast(d)$ TIOA, a $Bcast(d)$ TIOA, the TIOAs in VN for the VSAs, the client TIOAs in A, and a $GraphRealWorld(\epsilon, t_advance)$ TIOA. Parameters of the layer are the graph representation $G = (V, E)$, the GPS update period ϵ, the message delay d, and the time required for an advance action $t_advance$.

5.5 Bounding Areas

5.5.1 Overview

Now that we have two models of the physical movement of the system, it is necessary to show how I can use the $RealWorld$ model, representing the physical continuous motion of aircraft, to emulate the $GraphRealWorld$ model, representing the discrete motion of aircraft between edges in a graph.

To do this, constraints can be put on the $RealWorld$ model in order to represent our graph of safe paths. These constraints will be in the form of *bounding boxes* and *bounding cylinders* that can be used to partition R into a number of separate flight areas.

The bounding boxes and cylinders can be as large or as small as needed, with one simple requirement: that all aircraft can remain within any box or cylinder indefinitely. Since the motion

of aircraft is continuous and the velocity is non-zero, this means that all aircraft must be able to loop around within all bounding areas. Since those details (such as aircraft turning radius) are extraneous to this discussion, we assume the requirement is satisfied.

The goal of the next section is to arrange the bounding areas into a graph-like structure that we can use to emulate $GraphRealWorld$ with $RealWorld$, while ensuring they are arranged in a way that preserves the above separation requirements.

5.5.2 Definitions

I define the set $B = (b_1, b_2, ..., b_{|E|})$, where each b_i is a *bounding box*, a connected subset of R. While we call the elements of B boxes for convenience, they may not actually be in the shape of rectangular prisms.

I define the set $C = (c_1, c_2 ..., c_{|V|})$, where each c_i is a *bounding cylinder*, also a connected subset of R. Each c_i is defined by (r, h, loc) which represents a cylinder of space centered at $loc \in R$ with radius r and height h. The cylinder is oriented vertically, so that the projection of c_i onto the xy-plane is a circle.

5.6 Arranging the Bounding Areas

5.6.1 Overview

We can arrange these bounding boxes and cylinders in R to create our desired "map of allowable air routes" by arranging them in a graph-like structure. The next section will discuss this in detail, but for now let us turn to the safety requirements we wish to enforce in the system.

Recall that in the real-world ATC system there is a separation requirement of three miles lateral distance or 1000 feet altitude. Let us generalize these requirements, letting d_{vsep} represent the minimum vertical separation distance (the minimum difference in altitudes that is considered safe), and letting d_{hsep} represent the minimum horizontal separation distance.

These separation distances for aircraft in $RealWorld$ become the basis for how we separated our bounding areas. If we restrict aircraft to fly within bounding areas in an appropriate way and separate the bounding areas in a way that spaces them out by enough distance, we can use the bounding areas to ensure safety in the air traffic control system.

d_{hsep}	the minimum lateral separation between two aircraft
d_{vsep}	the minimum vertical separation between two aircraft

Table 5.1: Parameters for the minimum safe separation distances in $RealWorld$

Since we have sets of bounding areas B, C of the deployment space R and a condition for their separation, our next step in emulating $GraphRealWorld$ using $RealWorld$ is to organize the bounding areas into a collection of separated routes that aircraft can travel on. We do this by showing how the directed graph described in Section 5.2 can be represented in 3-space using bounding areas.

In order to ensure separation, I specify subsets of each bounding area as *inner bounding areas*, which are separated from the outer edges of their own bounding areas by d_{hsep} horizontally and d_{vsep} vertically.

5.6.2 Definitions

For each vertex $v \in V$, with $getRegion(v) \in J$, let $loc(v)$ be a point in R such that $region(loc(v)) = getRegion(v)$. Then, place a bounding cylinder $c \in C$ such that c is centered at $loc(v)$. For all points p such that $p \in c$, $region(p) = getRegion(v)$. The exact process of choosing the radius r and the height h is unspecified, but in 5.6.3 I will explain why choosing them should be possible to accomplish.

Now, for each outgoing edge $e = (v, v')$ from v, place a bounding box $b \in B$ connecting a portion of the outer surface of the cylinder c centered at $loc(v)$ with a portion of the outer surface of the cylinder c' centered at $loc(v')$. b does not need to be a rectangular prism, but must be connected.

Finally, for each bounding area $ba \in B \cup C$, the ***inner bounding area*** $iba \subseteq ba$ is a set of points $p_1 \in ba$ such that the distance between p_1 and the surface of ba is at least d_{hsep} horizontally and d_{vsep} vertically.

I also specify a number of functions that connect the graph representation G, the bounding area construction, and the continuous deployment space R:

- $loc : V \rightarrow R$, maps a vertex in V to its location in R.

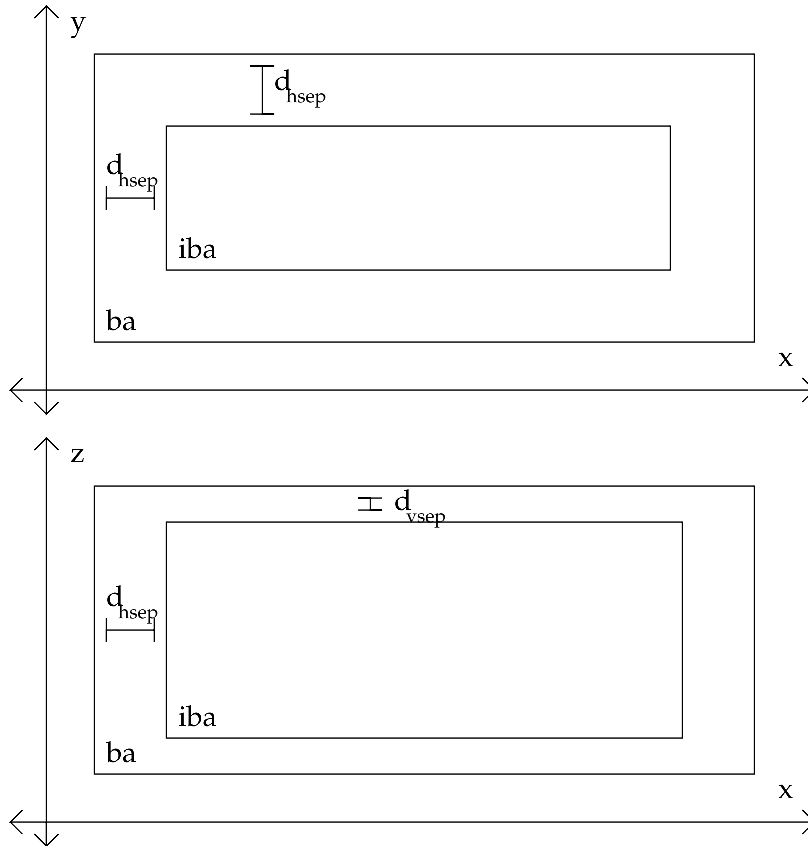


Figure 5-4: Cross Sections of Bounding Area ba and Inner Bounding Area iba

- $vertexToCyl : V \to C$, maps a vertex in V to its corresponding bounding cylinder $c \in C$. It is required that $loc(v) \in c$.
- $cylToVertex : C \to V$, $vertexToCyl(v) = c \leftrightarrow cylToVertex(c) = v$ for $v \in V$, $c \in C$.
- $edgeToBox : E \to B$, maps an edge in E to its corresponding bounding box $b \in B$.
- $boxToEdge : B \to E$, $edgeToBox(e) = b \leftrightarrow boxToEdge(b) = e$ for $e \in E$, $b \in B$.
- $objectAt : R \to V \cup E$, maps a location in R to the unique vertex $v \in V$ or edge $e \in E$ that corresponds to the bounding area at that location, or, if that location is outside of any bounding area, maps the location to the dummy edge e_NULL.
- $inner : B \cup C \to 2^R$, maps a bounding area in B or C to the subset of that bounding area which comprises the inner bounding area.
- $outer : B \cup C \to 2^R$, maps a bounding area in B or C to the subset of that bounding area which comprises the outer bounding area.

The bounding areas must satisfy the size requirement that for all bounding areas $ba \in B \cup C$, $inner(ba) \neq \emptyset$ and an aircraft's motion can be contained within $inner(ba)$ for an indefinitely long period of time.

The bounding areas must also satisfy the following four separation requirements:

1. For any points p_1 and p_2 in any two bounding boxes b_1 and b_2 respectively, where $b_1 \neq b_2$, p_1 and p_2 are separated by either d_{hsep} horizontally, d_{vsep} vertically, or both.
2. For any points p_1 and p_2 in any two bounding cylinders c_1 and c_2 respectively, where $c_1 \neq c_2$, p_1 and p_2 are separated by either d_{hsep} horizontally, d_{vsep} vertically, or both.
3. For any points p_1 and p_2 such that $p_1 \in b_1$ and $p_2 \in c_2$ where b_1 is a bounding box in B and c_2 is a bounding cylinder in C, we let $e = objectAt(p_1)$ and $v = objectAt(p_2)$. If $e \neq (v, v')$ and $e \neq (v', v)$ for some $v' \in V$, then p_1 and p_2 are separated by either d_{hsep} horizontally, d_{vsep} vertically, or both.
4. For any bounding box $b_1 \in B$ and bounding cylinder $c_2 \in C$, we let $e = boxToEdge(b_1)$ and $v = cylToVertex(c_2)$. If $e = (v, v')$ or $e = (v', v)$ for some $v' \in V$, then b_1 and c_2 meet at the border of the two bounding areas. Points in b_1 need not be separated from points in c_2.

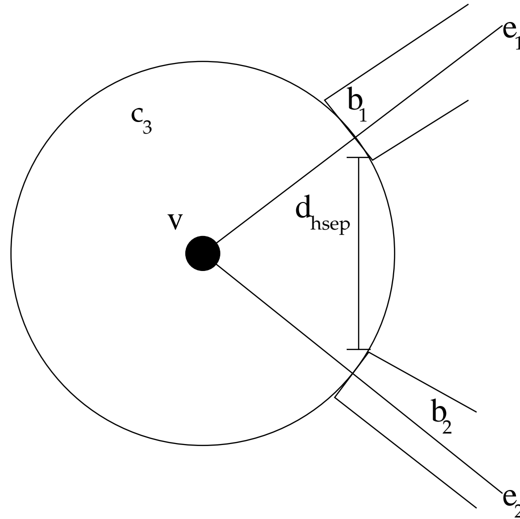


Figure 5-5: Bounding Boxes b_1 and b_2 Meeting at Bounding Cylinder c_3

5.6.3 Discussion and Justifications

While the separation between boxes and other boxes, and cylinders and other cylinders is quite easy to visualize, it is quite difficult to understand what these requirements mean for each cylinder that multiple edges border. An example of this situation is depicted in Figure 5-4. Assuming the two boxes are not sufficiently vertically separated, they are placed around the cylinder so that all points are horizontally separated. As you can see, this figure shows condition 4, that a bounding box such as b_1 with a cylinder such as c_3 at its endpoint is not separated from that cylinder, but is separated from all other boxes such as b_2.

A natural and difficult question now arises - is it even possible to lay out such a configuration while satisfying all separation requirements? First, I will justify this by saying in the *real* air traffic control system, a similar layout already exists. Specific altitudes and areas in the airspace are designated for certain routes, and putting them all together creates a similar layout.

Second, the fact that the required separation distances are so small compared to the large size of each region implies that it should be possible to accomplish such a layout. The size of the regions are bounded above by the communications radius in the physical layer, but this radius is extremely large compared to the separation distances. For standard sizes and parameter values, there is the vertical space for nearly thirty boxes to be incident on the same cylinder before even having to increase the cylinder's radius to accommodate more edges horizontally. In

most configurations of the layers, the size of each region will be much larger than the separation requirements, making such a placement much more likely to be possible.

The exact arrangement of such a placement would be difficult to perform algorithmically due to the massive size of the problem space. There are an infinite number of ways to arrange bounding areas for a given R and G, and for a system with small regions, a dense graph with many vertices, and large separation requirements, an arrangement may not even be possible. But for a region size, graph, and separation requirements that are similar to the real world ATC requirements, a proper arrangement is likely to be possible.

For those reasons, we can assume that such an arrangement exists, and that our defined sets satisfy all defined requirements.

5.7 Emulating the Known Path Layer with the Continuous Layer

5.7.1 Overview

Now that the bounding area construction is clear, we can show that the Known Path VSA Layer can be emulated by the Continuous VSA Layer. Much of this process is trivial, though, since there are no differences in the VSAs between the layers, and the differences between $Bcast$, $VtoVcast$, and the clients of the two layers are minor.

In $Bcast$ and $VtoVcast$, the only change that must be made is the changed data type of the GPSinput action, from the Continuous VSA Layer's locations in 3-space to the Known Path VSA Layer's region and edge. But within the code of both $Bcast$ and $VtoVcast$, the locations stored in the Continuous VSA Layer are merely translated into a corresponding region to determine whether or not $Bcast$ or $VtoVcast$ delivers a message. In the Known Path VSA Layer, the input still contains each region, giving both automata the exact same functionality as before.

The interesting and more difficult component, though, is the emulation of $GraphRealWorld$ using $RealWorld$, and that is what we must discuss in order to prove that the emulation can be done correctly.

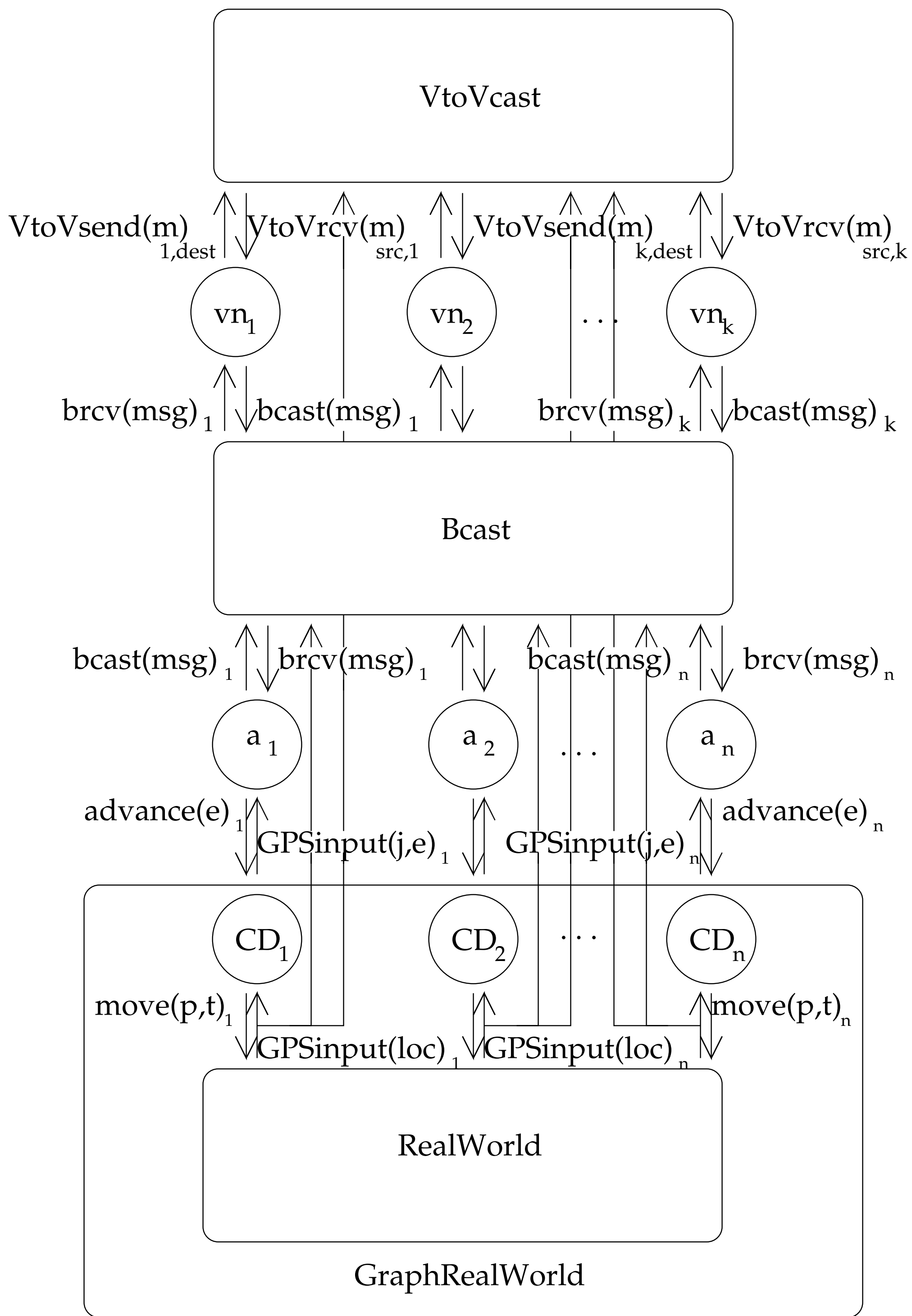


Figure 5-6: Emulation of the Known Path VSA Layer, for $I = \{1, 2, ...n\}$, $J = \{1, 2, ..., k\}$

5.7.2 Translation Automata

In order to show the emulation of the Known Path VSA Layer using the Continuous VSA Layer, we must show that $GraphRealWorld$ can be emulated by $RealWorld$. In order to do so, I create ***translation automata*** that translate continuous outputs from $RealWorld$ into the discrete outputs that $GraphRealWorld$ would provide, as well as translating the discrete inputs expected by $GraphRealWorld$ into continuous inputs expected by $RealWorld$.

5.7.3 Definitions

For each $i \in I$, let CD_i be a timed input-output automaton, for which the set of CD_i for all i function as our *translation automata*.

The translation automaton CD_i, for each $i \in I$ has the following actions:

- The input action $\mathsf{GPSinput}(loc)_i$, for $loc \in R$, which inputs the continuous GPS location from $RealWorld$.
- The output action $\mathsf{GPSinput}(j, e)_i$, for $j \in J$ and $e \in E$, which outputs the discrete GPS location to a_i..
- The input action $\mathsf{advance}(e)_i$, for $e \in E$, which inputs a control action from a_i to move $aircraft[i].e$ to e.
- The output action $\mathsf{move}(p, t)_i$, for $t \in \mathbb{R}^+$ and $p \in R$, which outputs a control action to $RealWorld$ to move $aircraft[i].loc$ to point p within time t.

CD_i also stores a state variable $location \in R$, which is the most recently seen location of aircraft i in $RealWorld$. The variable is initialized to loc_NULL. We enforce an invariant on $location$, and after changing from loc_NULL, the value of $location$ will always be inside some inner bounding box, the most recent inner bounding box that i was in.

The behavior of CD_i is as follows. Upon inputting $\mathsf{GPSinput}(loc)_i$, first, CD_i tests whether $objectAt(location) \in E$ and $loc = loc_NULL$. If this test is true (the aircraft has landed), CD_i sets $GPSlanded$ to $true$ in order to immediately output a $\mathsf{GPSinput}(j, e)_i$ action for $j = region(location)$ and $e = e_DONE$.

```
Constants:
  t_advance , the time between an advance action
        and the corresponding state update
        ∈ ℝ ≥ 0

Signature:
  Input GPSinput(loc)_i , loc ∈ R
  Output GPSinput(j, e)_i , j ∈ J,
            e ∈ E ∪ { e_NULL, e_DONE }
  Input advance(e)_i , e ∈ E
  Output move(p, t)_i , p ∈ R, t ∈ ℝ ≥ 0

State:
  location, ∈ R, i′s most recently input
      location which is located inside an
      inner bounding box,
      initialized to loc_NULL
  GPStodo, a boolean, initialized to false
  GPSlanded, a boolean, initialized to false
  move, a boolean, initialized to false
  moveto, an edge in E ∪ {e_NULL, e_DONE}
  now : ℝ


Trajectories:
  evolves
    d(now) = 1
    constant location, GPStodo
      move, moveto
  stops when
    any precondition is satisfied

Actions:
  Input GPSinput(loc)_i
  Effect:
    if (location ∈ b for some b ∈ B ∧
     loc = loc_NULL) then
      GPSlanded ← true
    else
      for b ∈ B:
        if (loc ∈ inner(b)) then
          location ← loc
      GPStodo ← true

  Output GPSinput(j, e)_i
  Precondition:
    GPSlanded = true ∧
    j = region(location) ∧
    e = e_DONE
  Effect:
    GPSlanded ← false

  Output GPSinput(j, e)_i
  Precondition:
    GPStodo = true ∧
    j = region(location) ∧
    e = objectAt(location)
  Effect:
    GPStodo ← false

  Input advance(e)_i
  Effect:
    v ← vertexAfter(objectAt(location))
    if (e = (v, v′) for some v′ ∈ V) then
      move ← true
      moveto ← e

  Output move(p, t)_i
  Precondition:
    move = true ∧ t = t_advance ∧
    p ∈ (inner(edgeToBox(moveto)) ∩
      region(vertexAfter(moveto)))
  Effect:
    move ← false
```

Figure 5-7: TIOA Code for Translation Automaton CD_i, for $i \in I$

If the above test is false, CD_i tests whether, for some $e' \in E$, if $loc \in inner(edgeToBox(e'))$. If that test is true, CD_i sets $location$ to loc. Performing this test before updating $location$ ensures that our desired invariant is true, and $objectAt(location)$ is always an edge in E. CD_i then sets $GPStodo$ to $true$ (whether or not $location$ was updated) in order to immediately output a GPSinput$(j, e)_i$ action, with $j = region(location)$ and $e = objectAt(location)$.

Upon inputting advance$(e)_i$, CD_i lets $v = vertexAfter(objectAt(location))$, which is always defined because of the invariant on $location$'s value, and tests whether $e = (v, v')$ for some $v' \in V$. If that test fails, the advance input is invalid, and CD_i does nothing. Otherwise the advance input is valid, and CD_i sets $move$ to $true$ and $moveto = e$ in order to immediately output a move$(p, t)_i$ action, with $t = t_advance$, and $p \in (inner(edgeToBox(e)) \cap region(vertexAfter(e)))$, so that the aircraft moves to a location inside the inner bounding box corresponding to e which is also in the region of e's target vertex.

5.7.4 Emulation

Now we prove that, using the translation automata, an execution of $RealWorld$ can emulate an execution of $GraphRealWorld$. In order to do this, though, I must restrict the motion allowed within $RealWorld$ so that the locations which aircraft are allowed to travel in $RealWorld$ correspond to the bounding areas $B \cup C$. Using that assumption, we can prove Theorem 5.1 about the emulation of $GraphRealWorld$ with $RealWorld$.

I amend the behavior of $RealWorld$ to adhere to a number of constraints based on the graph representation. I have amended the code of $RealWorld$ in Figure 5-8 to further specify the movement of the aircraft with these assumed constraints:

1. The connected deployment space $R = (\bigcup_{j=1}^{|E|} b_j) \cup (\bigcup_{k=1}^{|V|} c_k)$. That is, the union over all bounding areas encompasses the entire deployment space for the Continuous VSA Layer.

2. For all $i \in I$, in $RealWorld$ the value of $aircraft[i].loc_takeoff = p$ is restricted to values of p such that $objectAt(p) = (v, v') \in E$ for some vertex $v \in T$ in the set of terminal vertices, and that $p \in inner(edgeToBox(objectAt(p)))$. That is, all aircraft enter the system inside the inner bounding box of some edge outgoing from a terminal vertex.

3. For all $i \in I$, in $RealWorld$ the value of $aircraft[i].loc_landing = p$ is restricted to values of

```
Constants:
  ε , the GPS update frequency ∈ ℝ ≥ 0
  v_max , the maximum aircraft velocity ∈ ℝ+
  t_advance , the time between an advance action
        and the corresponding state update
        ∈ ℝ ≥ 0

Signature:
  Output GPSinput(loc)_i , i ∈ I, loc ∈ R
  Input move(p, t)_i , p ∈ ℝ^3, t ∈ ℝ ≥ 0
  Internal finishMove_i , i ∈ I
  Internal Takeoff_i , i ∈ I
  Internal Landing_i , i ∈ I
  Internal updateLastBox_i , i ∈ I

State:
  GPSdone, an array of booleans for each i ∈ I and
        for k ∈ ℤ ≥ 0 initialized to false
  aircraft, an array of type AircraftRecord
      for each aircraft[i], initialized to:
      aircraft[i].loc ← loc_NULL
      aircraft[i].lastbox ← loc_NULL
      aircraft[i].moving ← false
      aircraft[i].movingtime ← 0
      aircraft[i].movingto ← loc_NULL
      aircraft[i].movingfrom ← loc_NULL
      aircraft[i].loc_takeoff ∈ inner(b) for v ∈ T
        and b = edgeToBox((v, v')), (v, v') ∈ E
      aircraft[i].loc_landing ∈ inner(b) for v' ∈ T
        and b = edgeToBox((v, v')), (v, v') ∈ E,
  now : ℝ

Trajectories:
  evolves
    d(now) = 1
    for each i ∈ I, with a[i] = aircraft[i]:
      if (aircraft[i].loc = loc_NULL) then
        constant a[i].loc
      else
        0 < |d(a[i].loc)| ≤ v_max
  invariant
    if (a[i].moving = true) then
      dist(a[i].loc, a[i].movingto) ≤
          v_max(a[i].movingtime − now)
      if (objectAt(a[i].movingfrom) = e ∈ E ∧
          objectAt(a[i].movingto) = e' ∈ E ∧
          vertexAfter(e) = v such that (v, v') = e')
        then a[i].loc ∈ edgeToBox(e)∪
                edgeToBox(e') ∪ vertexToCyl(v)
    else a[i].loc ∈ inner(b) such that
          b = edgeToBox(objectAt(a[i].loc))
  stops when
    any precondition is satisfied

Actions:
  Output GPSinput(loc)_i
  Precondition:
    now = kε ∧
    GPSdone[i, k] = false ∧
    loc = aircraft[i].loc
  Effect:
    GPSdone[i, k] ← true

  Input move(p, t)_i
  Effect:
    if ((dist(aircraft[i].loc, p) ≤ v_max · t) ∧
        (aircraft[i].moving = false) ∧
        (objectAt(p) = (v, v') ∈ E such that v =
          vertexAfter(objectAt(aircraft[i].loc))))
        then
      aircraft[i].moving ← true
      aircraft[i].movingtime ← now + t
      aircraft[i].movingto ← p
      aircraft[i].movingfrom ← aircraft[i].loc

  Internal finishMove_i
  Precondition:
    aircraft[i].loc = aircraft[i].movingto ∧
    aircraft[i].moving = true
  Effect:
    aircraft[i].moving ← false
    if (aircraft[i].loc_landing ∈
        edgeToBox(objectAt(aircraft[i].loc)) then
      aircraft[i].moving ← true
      aircraft[i].movingtime ← now + t_advance
      aircraft[i].movingto ← loc_landing
      aircraft[i].movingfrom ← aircraft[i].loc

  Internal Takeoff_i
  Precondition:
    now = aircraft[i].t_takeoff ∧
    aircraft[i].loc = loc_NULL
  Effect:
    aircraft[i].loc ← aircraft[i].loc_takeoff

  Internal Landing_i
  Precondition:
    aircraft[i].loc = aircraft[i].loc_landing
  Effect:
    aircraft[i].loc ← loc_NULL

  Internal updateLastBox_i
  Precondition:
    aircraft[i].loc ∈ inner(b) such that
    (b ∈ B ∧ aircraft[i].lastbox ∉ inner(b))
  Effect:
    aircraft[i].lastbox ← aircraft[i].loc
```

Figure 5-8: TIOA Code for RealWorld with Graph Constraints

p such that $objectAt(p) = (v, v') \in E$ for some vertex $v' \in T$ in the set of terminal vertices, and that $p \in inner(edgeToBox(objectAt(p)))$. That is, all aircraft leave the system at some point inside the inner bounding box of an edge incident on a terminal vertex.

Also, once $objectAt(aircraft[i].loc) = objectAt(aircraft[i].loc_landing)$, $aircraft[i].loc$ will move to $loc_landing$ within time $t_advance$. In other words, once an aircraft has reached the bounding box before its landing vertex, it will land within $t_advance$ time.

4. If $aircraft[i].loc$ is inside a bounding box $b \in B$ with $boxToEdge(b) = e \in E$, with the value of $aircraft[i].moving = false$, the value of $aircraft[i].loc$ is restricted to remain within the inner bounding box $inner(b)$.

   When a $\mathsf{move}(p, t)_i$ action occurs, we first to check if it is valid. A valid move action requires that the distance between $aircraft[i].loc$ and p is not too large to be traversed in t_max time, that $objectAt(p) = e \in E$ such that $objectAt(aircraft[i].loc)$ either is an edge incident on e's source vertex, is that source vertex itself, or is the outer bounding box of some edge with a source vertex the same as e's.

   If the action is valid, then let $b = objectAt(aircraft[i].loc)$, $(v, v') = e' = objectAt(p)$, let $c = vertexToCyl(v)$, and let $b' = edgeToBox(e')$. Then, until $aircraft[i].loc$ reaches p, $aircraft[i].loc$ is restricted to remain within $b \cup c \cup b'$, at which point it must remain within $inner(b')$ until a subsequent input $\mathsf{move}(p', t')_i$.

   If a second valid move occurs before $aircraft[i].loc$ reaches p, the original request is aborted and i moves to the new location. For this second move to be valid, the same conditions apply as above.

   In summary, each aircraft's nondeterministic motion must remain inside an inner bounding box until told to move to another bounding box. Then, it must move through a bounding cylinder into another inner bounding box.

Theorem 5.1

Let system S be the composition of $RealWorld$ and the CD_i automata for all $i \in I$. For every execution α_r of S, there exists an execution α_g of $GraphRealWorld$ such that:

- α_r and α_g have the same actions occur at the same times at the external boundary of $GraphRealWorld$.

- There is a correspondence between states of S and states of $GraphRealWorld$ which includes the fact that at all times [1], for aircraft i in $GraphRealWorld$:

  - If $GRW.aircraft[i].e = e$ for some $e \in E$, and $GRW.aircraft[i].auth = false$, then $e = objectAt(RW.aircraft[i].loc)$, $RW.aircraft[i].loc \in inner(edgeToBox(e))$, and the value of $RW.aircraft[i].moving = false$, as well as the converse. That is, an aircraft on an edge in $GraphRealWorld$ corresponds to that aircraft being inside that edge's inner bounding box in $RealWorld$.
  - If $aircraft[i].auth = true$ and $aircraft[i].authedge = e$ in $GraphRealWorld$, then $aircraft[i].moving = true$ and $e = objectAt(aircraft[i].movingto)$ in $RealWorld$, as well as the converse. That is, an aircraft being authorized to pass through a vertex in $GraphRealWorld$ corresponds to that aircraft's motion being able to pass through that vertex's bounding cylinder in $RealWorld$.

Assuming the above restrictions, if we compose $RealWorld$ and the CD_i automata for all $i \in I$, the behavior of that composition emulates the behavior of $GraphRealWorld$, as seen by the other components in the Known Path VSA Layer, and the state of $GraphRealWorld$ corresponds as above to the state of $RealWorld$.

Proof

Our primary goal is to show that the state of $GraphRealWorld$ and the state of S correspond during all trajectories and after all actions through these relations for all $i \in I$:

- $GRW.now = RW.now$

- If $CD_i.GPStodo = false$ then, for all $k \in \mathbb{Z} \geq 0$:

  - $GRW.GPSdone[i, k] = RW.GPSdone[i, k]$

[1] Saying *at all times* is not exactly true. Between multiple actions that occur at the same time, the states may not always correspond. More formally, we are saying that for each finite prefix of execution α_r, we can create some finite prefix of α_g with the same size (in time) in which the state correspondence is satisfied at the end. Moreover, successive extensions to the prefix of α_r should result in extensions of α_g of the same size.

- $GRW.aircraft[i].t_takeoff = RW.aircraft[i].t_takeoff$
- $GRW.aircraft[i].v_takeoff = v|(v, v') = objectAt(RW.aircraft[i].loc_takeoff)$
- $GRW.aircraft[i].v_landing = v|(v', v) = objectAt(RW.aircraft[i].loc_landing)$
- $GRW.aircraft[i].e = objectAt(RW.aircraft[i].lastbox)$
- $GRW.aircraft[i].region = region(RW.aircraft[i].loc)$
- If $CD_i.move = false$ then:
  - $GRW.aircraft[i].auth = RW.aircraft[i].moving$
  - $GRW.aircraft[i].authedge = objectAt(RW.aircraft[i].movingto)$
  - $GRW.aircraft[i].authtime = RW.aircraft[i].movingtime$

Now, let us fix execution α_r of S. The state of $GraphRealWorld$ in α_g will then have initial state which corresponds to the initial state of S as follows:

- For all $i \in I, k \in \mathbb{Z} \geq 0$, $RW.GPSdone[i, k] = GRW.GPSdone[i, k] = false$.
- $RW.now = GRW.now = 0$
- For all $i \in I$:
  - $RW.aircraft[i].loc = RW.aircraft[i].lastbox = loc_NULL$
    $GRW.aircraft[i].e = objectAt(loc_NULL) = e_NULL$.
  - $region(RW.aircraft[i].loc) = GRW.aircraft[i].region = j_NULL$
  - $RW.aircraft[i].loc_takeoff = p \in R$ (as constrained by Assumption 2).
    $GRW.aircraft[i].v_takeoff = v|(v, v') = objectAt(p)$
  - $RW.aircraft[i].loc_landing = p \in R$ (as constrained by Assumption 3).
    $GRW.aircraft[i].v_landing = v|(v', v) = objectAt(p)$
  - $RW.aircraft[i].moving = GRW.aircraft[i].auth = false$
  - $RW.aircraft[i].movingto = loc_NULL$
    $GRW.aircraft[i].authedge = objectAt(loc_NULL) = e_NULL$

– $RW.aircraft[i].movingtime = GRW.aircraft[i].authtime = 0$

This initial state of S in execution α_r therefore corresponds to $GraphRealWorld$'s initial state in α_g with the required relations. Now I show that in any step of the execution of α_r, we can construct a step of α_g such that the required state correspondence is maintained.

Assume the correspondence has held through step n of α_r and α_g. Now, given the possibilities for step $n+1$ in α_r, I construct all possibilities of step $n+1$ in α_g, and show that α_g can be constructed so that the state correspondence still holds:

- Say that step $n+1$ involves a passage of time in which no actions occur in α_r and for some $i \in I$, $aircraft[i].moving = false$. The value of now increases by some amount, and the value of each $aircraft[i].loc$ changes at some rate, yet remains within an inner bounding box. In α_g, the value of now can increase by the same amount, and for the same i, $aircraft[i].auth = false$ since the value corresponds to the value of $aircraft[i].moving$ in $RealWorld$. Since $aircraft[i].loc$ in $RealWorld$ never changes bounding boxes, we know that $aircraft[i].lastbox$ remains the same and in $GraphRealWorld$, $aircraft[i].e$ does not change, keeping the correspondence intact.

  If no actions occur in α_r but for some $i \in I$, $aircraft[i].moving = true$, that aircraft may leave its bounding box on its way to some other bounding area. That aircraft does not yet reach $aircraft[i].movingto$, as doing so would trigger a finishMove action. In α_g, the value of $aircraft[i].auth$ is $true$, but a doAdvance does not occur, so i does not yet advance onto $aircraft[i].authedge$, keeping the correspondence intact.

- Say that at step $n+1$ of α_r, $RealWorld$ outputs a GPSinput$(loc)_i$ action to CD_i. This step corresponds to no change in α_g. The equality of $GRW.GPSdone$ and $RW.GPSdone$ does not remain, as the output from $RealWorld$ is input by CD_i and the flag $CD_i.GPStodo$ is set to $true$.

- Say that at step $n+1$ of α_r, CD_i outputs a GPSinput$(j,e)_i$ action which was translated from a GPSinput$(loc)_i$ action as specified in Section 5.7.2. This sets $CD_i.GPStodo$ to $false$. Then, in α_g, $GraphRealWorld$ outputs a corresponding GPSinput$(j,e)_i$ action. Since the values of now correspond, the value of now in $GraphRealWorld$ must be equal to $k\epsilon$ for

some k, since the values of $aircraft$ correspond, j and e in the two outputs must be the same, and since the values of $GPSdone$ were equal until the preceding GPSinput$(loc)_i$ occurred, $GraphRealWorld$ must have $GPSdone[i,k] = false$. Therefore, it will be enabled to perform this action. After the action, both automata have set $GPSdone[i,k] = true$, and $CD_i.GPStodo = false$, so the state correspondence holds after step $n+1$.

- Say that at step $n+1$ of α_r, CD_i receives an input advance$(e)_i$ action from a_i. If the input is valid, this action sets $CD_i.move$ to $true$ in order to output a translated move action. Let us assume that the input is valid. The corresponding α_g has a advance$(e)_i$ input to $GraphRealWorld$. It sets $aircraft[i].auth$ to $true$, $aircraft[i].authedge$ to e, and sets $aircraft[i].authtime$ to $now + t_advance$. After this, the states of those variables no longer correspond with $RealWorld$, but $CD_i.move$ is $true$ as required.

- Say that at step $n+1$ of α_r, CD_i outputs a move$(p,t)_i$ action as specified in Section 5.7.2. $RealWorld$ receives this move$(p,t)_i$, let us assume it is valid. It sets $aircraft[i].moving$ to $true$, $aircraft[i].movingto$ to p, $aircraft[i].movingtime$ to $now + t$, where $t = t_advance$ as specified by CD_i. Also, $CD_i.move$ is set to $true$. Since the $aircraft[i].moving$ variables were updated to correspond with the $aircraft[i].auth$ variables in $GraphRealWorld$, the correspondence holds as required.

  If the advance$(e)_i$ action was invalid because it violates the graph requirements, in α_r, CD_i will not pass it through to $RealWorld$, and in α_g, $GraphRealWorld$ will also have ignored the action. If the action was invalid in $RealWorld$ because $aircraft[i].moving = true$, then we know that $aircraft[i].auth = true$ in $GraphRealWorld$, allowing $GraphRealWorld$ to ignore the action as well.

- Say that at step $n+1$ of α_r, an internal finishMove$_i$ occurs. Due to the motion invariant of $RealWorld$, and the fact that all move actions to $RealWorld$ from CD_i have $t = t_advance$, this must have occurred at most $t_advance$ time after such a move$_i$ action. As specified by finishMove$_i$, $aircraft[i].loc = aircraft[i].movingto$, which, by the specifications of CD_i is within the inner bounding box of the edge e from the original advance$(e)_i$ input that was received by S. Therefore, an updateLastBox$_i$ must occur, updating $aircraft[i].lastbox$ to some location in $inner(edgeToBox(e))$. The value of $aircraft[i].moving$ is set to $false$

and its associated variables get reset as specified.

In step $n+1$ of α_g, an internal $\mathsf{doAdvance}_i$ action occurs. We know this is possible, because now must be at most $t_advance$ time after an $\mathsf{advance}(e)_i$ action and $aircraft[i].auth$ must be true, as it is equal to $aircraft[i].moving$ in $RealWorld$. Therefore, $aircraft[i].e$ gets updated to e, which is equal to the updated $objectAt(aircraft[i].lastbox)$, keeping them equal. The value of $aircraft[i].auth$ is also set to $false$ with its associated variables, keeping them in correspondence with the $aircraft[i].moving$ variables in $RealWorld$.

- Say that at step $n+1$ of α_r, a $\mathsf{Takeoff}_i$ action occurs in $RealWorld$. Since the variables correspond as above, a $\mathsf{Takeoff}(e)_i$ action occurs, with $objectAt(RW.aircraft[i].loc_takeoff) = e$, in execution α_g of $GraphRealWorld$, and the values of $aircraft[i].e$ and $aircraft[i].region$ are updated to correspond.

- Say that at step $n+1$ of α_r, a $\mathsf{Landing}_i$ action occurs. This must have been preceded by time $t_advance$ by a move_i action which put $aircraft[i].loc$ into the inner bounding box which $aircraft[i].loc_landing$ is also in. The value of $aircraft[i].loc$ is then set to loc_NULL. In α_g, a $\mathsf{Landing}_i$ occurs in $GraphRealWorld$, which can occur since it must have been preceded by an $\mathsf{advance}_i$ action which caused the move_i action in α_r. The value of $aircraft[i].e$ is updated to correspond.

Since every step in α_r of S has a directly corresponding step in α_g of $GraphRealWorld$ that preserves the state correspondence above, throughout the two executions, the state of S corresponds to the state of $GraphRealWorld$. Since every step in S is able to occur in $GraphRealWorld$, S is able to emulate $GraphRealWorld$.

Q.E.D.

Now, using Theorem 5.1, I prove a corollary.

Corollary 5.2

Let system S, as described in Theorem 5.1, composed with the Continuous VSA Layer components $Bcast$ and $VtoVcast$ be called system S'.

For every execution α_r of system S' composed with the Known Path VSA Layer clients and VSAs, there exists an execution α_g of $KnownPathVL$ such that:

- Let any two actions with the same name, but possibly with different parameters, be called ***similar actions***. In α_r and α_g, ignoring the translation actions between $RealWorld$ and the CD_i automata in α_r, similar actions will occur at the same times in the two executions.

- There is a correspondence between states of S and states of $GraphRealWorld$ which includes the fact that at all times [2], for aircraft i in $GraphRealWorld$:

  – If $aircraft[i].e = e$ for some $e \in E$, and $aircraft[i].auth = false$ in $GraphRealWorld$, then $e = objectAt(aircraft[i].loc)$ and $aircraft[i].moving = false$ in $RealWorld$, as well as the converse.

  – If $aircraft[i].auth = true$ and $aircraft[i].authedge = e$ in $GraphRealWorld$, then $aircraft[i].moving = true$ and $e = objectAt(aircraft[i].movingto)$ in $RealWorld$, as well as the converse.

Proof

By Theorem 5.1, we know that if inputs to S and $GraphRealWorld$ occur at the same times, then corresponding outputs from both S and $GraphRealWorld$ occur at the same times and states of S and $GraphRealWorld$ correspond as above.

I now show that actions in $CVL.Bcast$ correspond to actions in $KPVL.Bcast$, and that actions in $CVL.VtoVcast$ correspond to actions in $KPVL.VtoVCast$, where CVL is the Continuous VSA Layer, and $KPVL$ is the Known Path VSA Layer.

This correspondence, though, is trivial. $\mathsf{GPSinput}(loc)_i$ actions occur at the same time as $\mathsf{GPSinput}(j, e)_i$ actions, and $region(loc) = j$, as shown in Theorem 5.1. Since the Continuous $Bcast$ and $VtoVcast$ use this location information only to determine $region(loc)$, there is a directly corresponding state between the Continuous and Known Path versions of the automata. Since the actions both automata take depend solely on this corresponding state and inputs from $RealWorld$ or $GraphRealWorld$, their actions correspond as well.

Therefore, system S' and $KnownPathVL$ have corresponding executions α_r and α_g such that actions in both occur at the same time, and states in S' correspond to states of $KnownPathVL$.

Q.E.D.

[2] See note 1. The same inexactness in saying *at all times* applies here as well.

5.7.5 Safety

Now we can prove that, as arranged, during corresponding executions of the Continuous VSA Layer and the Known Path VSA Layer, that keeping aircraft separated in the Known Path VSA Layer implies that aircraft are separated by the separation requirements from Section 5.6.1.

Recall that the separation requirement is that no two aircraft ever get within d_{hsep} of each other laterally or d_{vsep} of each other vertically. If we require the motion in $RealWorld$ to follow the requirements enumerated in 5.7.4., we can ensure safety and prove this theorem.

Theorem 5.3

Let α_r and α_g be corresponding executions, according to the correspondence given in Corollary 5.2, of the Continuous VSA Layer and the Known Path VSA Layer respectively. Assume that, in α_g, for all $i, k \in I$:

- In $GraphRealWorld$, $aircraft[i].e \neq aircraft[k].e$ unless both are e_NULL or e_DONE, throughout the execution α_g. That is, no two aircraft ever occupy the same edge.
- In $GraphRealWorld$, for each vertex $v \in V$, if $aircraft[i].authedge = (v, v')$ then for all k, $aircraft[k].authedge \neq (v, v'')$ throughout the execution α_g. That is, no two aircraft are authorized to pass through the same vertex at the same time.

Then, on corresponding execution α_r of $RealWorld$, for $i, k \in I$, the values of $aircraft[i].loc$ and $aircraft[k].loc$ are always separated by d_{hsep} horizontally, d_{vsep} vertically, or both, throughout the execution.

Proof

First, by Theorem 5.1, we know that aircraft occupying the same edge in $GraphRealWorld$ corresponds to those same aircraft being in the same bounding box in $RealWorld$. Similarly, we know that aircraft being authorized to pass through the same vertex in $GraphRealWorld$ corresponds to those aircraft (possibly) being in the same bounding cylinder in $RealWorld$.

Therefore, I can rephrase the theorem as: if no two aircraft are in the same bounding area in $RealWorld$, all aircraft are separated by d_{hsep} horizontally, d_{vsep} vertically, or both.

By the bounding area construction, we know that all aircraft in $ba \in B \cup C$ must remain within the ***inner bounding area*** $inner(ba) = iba$ except while passing through a vertex. By the definition of bounding areas, we know that an aircraft in a bounding box is separated from all other aircraft in all other bounding boxes. Now we show that an aircraft in a bounding cylinder is separated from all aircraft in bounding boxes with a shared boundary.

Since we assume from the theorem statement that no two aircraft attempt to pass through a vertex at the same time, it follows that at any shared boundary between bounding areas ba_1 and ba_2 in which $aircraft[i].loc \in ba_1$ and $aircraft[k].loc \in ba_2$, only one aircraft is in the outer bounding area, say ba_1, and the other aircraft is in the inner bounding area $inner(ba_2)$. Since all points in the inner bounding area $inner(ba_2)$ are separated from all boundaries of ba_2 by d_{hsep} horizontally and d_{vsep} vertically, $aircraft[i].loc$ and $aircraft[k].loc$ must also be separated by d_{hsep} horizontally and d_{vsep} vertically.

Therefore, if no two aircraft occupy the same edge or attempt to pass through the same vertex at the same time in $GraphRealWorld$ on execution α_g, each aircraft is separated by d_{hsep} horizontally, d_{vsep} vertically, or both, from all other aircraft in $RealWorld$ on corresponding execution α_r.

Q.E.D.

5.8 Chapter Summary

The physical system has therefore been constrained so that aircraft are able to move from their $loc_takeoff$ to their $loc_landing$ by simply moving through the bounding cylinders and boxes. Since we have arranged the bounding areas appropriately, if we assume that no two aircraft ever occupy the same bounding area, it follows that the separation requirements are satisfied. Therefore, while the aircraft move continuously throughout space, we are able to discretize their movement as transitions from one bounding area to an adjacent one.

This discrete interpretation of the aircraft's movement makes our goal much simpler, as instead of caring exactly where each aircraft is, we can consider any aircraft within a bounding box b_k to be on the corresponding edge $e \in E$. This is exactly what is desired, as we can now use a graph-theoretical approach to solving the open problems in the air traffic control system, that is, ensuring that no two aircraft occupy the same bounding area, and efficiently moving aircraft

from their origin to their destination.

To summarize, In this chapter I have:

- Defined a discrete model for the physical behavior of the air traffic control system.
- Developed a system of constraints for the continuous system called bounding areas, which can be used to organize the space into a number of paths.
- Abstracted the paths of bounding areas into a graph representation for the air traffic control system that can be used in graph-theoretical algorithms for performing the functions of air traffic control.

For the next two chapters, it will be useful to briefly reiterate the model that was developed in this chapter. While the physical model of the system involves aircraft that move continuously through space, our use and arrangement of bounding areas allows us, for the purposes of the algorithm, to consider the model of a directed graph as a model for the system.

The directed graph contains a number of vertices, of which some correspond to the entrance and exit points of the air traffic control system (terminal control areas), and some simply function as a point where two or more edges meet. An edge in the graph represents a partition of airspace, each edge separated from all other edges, that our planes can use to travel between all vertices. We consider an aircraft to be on an edge if and only if it is within the corresponding bounding box. Aircraft move nondeterministically from their entrance vertex to their exit vertex along edges.

Remember that while at a lower level layer, aircraft move continuously through space and have locations in 3-space, I have proven that fact is safely abstracted away by the Known Path VSA Layer. We can consider the graph to represent the system and wish to satisfy one important requirement in order to prove safety: each bounding area must always contain zero or one aircraft. If this is satisfied, then the aircraft can fly freely within that bounding area with no adverse effect on safety, so ensuring the satisfaction of that requirement becomes a primary goal for the air traffic control algorithms.

Chapter 6

The FIFO Algorithm

In the previous chapter, I abstracted our earlier continuous model for the motion of aircraft into a discrete directed graph model for use in my algorithms. Each aircraft can be considered to be on a single edge in the graph of air routes. The problem of ensuring safety in such a system, by Theorem 5.3, becomes the problem of ensuring only a single aircraft is on any edge or is cleared to pass through any vertex at any time.

To do this we employ the Known Path VSA layer discussed in Chapter 5. For each of the regions in J, a VSA controls the traffic in that region. When an aircraft i wishes to progress from its current edge to a new edge, it makes a request to do so from the region's VSA. In order for the movement to be safe, the VSA determines whether the desired edge is empty or not.

If the desired edge is empty, the VSA must determine whether it has given another aircraft clearance to pass through the same vertex, where each vertex functions as an intersection that aircraft must pass through to get from one edge to another. If no other aircraft has clearance, i is given clearance, and progresses from the current edge to the desired edge.

The simplicity of the algorithm is a direct result of using the Known Path VSA layer, and this chapter will show how simple it is to control a complex system with a relatively small algorithm when a VSA layer is used. In this chapter I will specify the problems that this algorithm will solve, provide the details of the algorithm in both TIOA pseudocode and detailed explanation, and prove that the algorithm solves the problems.

The algorithm is called the *FIFO Algorithm*, due to the fact that it resolves conflicts between multiple aircraft requesting clearance using a simple first-in-first-out queue. In the next chapter

I will propose an improvement upon the algorithm by resolving conflicts in a more intelligent manner.

6.1 Problem Specification

The problem that the FIFO Algorithm solves has two main parts.

The first part is safety, where we must ensure that no two aircraft violate the separation requirements of d_{hsep} minimum lateral separation and d_{vsep} minimum vertical separation. As shown in Theorem 5.3, our graph model simplifies this problem for us, allowing us to restate the problem in terms of the graph of air routes. The problem of safety can therefore be stated as: *no two aircraft may occupy the same edge or progress through the same vertex at any time.*

The second part is progress. While we make no requirements that the overall movement of the aircraft is efficient (as the nondeterministic movement of said aircraft makes that impossible), we would like to require that when there is a conflict between multiple aircraft requesting clearance, one of those aircraft will eventually be cleared. The problem of progress can therefore be stated as: *given a set of aircraft that wishes to move onto some empty edge e, one will be allowed to do so in bounded time.*

6.2 Capabilities of the Known Path VSA Layer

While the capabilities of the VSAs were discussed in depth in Chapter 4, recall these few short points about their capabilities which we will be concerned with when developing the FIFO Algorithm.

- A VSA is only active if there is at least one aircraft in its region. For our purposes, we assume the problems of starting it up and keeping it up are trivial, and therefore assume that if there is an aircraft in a region, the VSA associated with it is active and has not experienced failures, with the exception of a reasonable startup time, of the message delay d, as a single aircraft enters the region when there were previously none.
- Aircraft do not communicate directly with other aircraft. They have the ability to broadcast to $Bcast$, which will be received by the VSA in the aircraft's region through a brcv action. The

VSA can also communicate with all aircraft in its region with a bcast action, and the aircraft will receive them with a brcv action.

- A VSA can communicate with any neighboring VSA which is alive using a VtoVsend. The neighboring VSA will receive the message with VtoVrecv. We assume that a communication between two active VSAs cannot fail.

Before I present the algorithm, it would be helpful to recall some of the functions defined in the last chapter that will be used in the algorithm. All clients and VSAs have access to the graph representation of air routes $G = (V, E)$, and the functions $getRegion : V \rightarrow U$ mapping a vertex to the region that contains it, and $getRegions : E \rightarrow U$ which maps an edge to the one or two regions that contain it. The function $vertexAfter : E \rightarrow V$ maps an edge in E to the vertex that (directed) edge leads to.

6.3 Algorithm

6.3.1 Overview

The high level idea of the FIFO Algorithm is to use each VSA to keep track of the state of its region. A VSA stores information about which edge each aircraft in its region is on. The VSA also stores an array of queues, one for each vertex in its region, in order to determine in which order aircraft should be cleared to pass through each vertex. Each client periodically updates the VSA in its region with position updates, so that the VSA can keep its state up to date.

When an aircraft i wishes to progress to an adjacent edge e, a_i requests clearance from the VSA in its region vn_j. The VSA then puts i in a queue for the vertex between i's current edge and e. When i reaches the front of the queue, the VSA informs i's client a_i that the vertex is reserved for i, and that it can therefore safely advance to e.

Whenever an aircraft has reached the front of the queue, safely advanced, and updated the VSA to the aircraft's new position, the VSA removes that aircraft from the front of the queue, and moves the next aircraft, who has requested an edge that is now empty, to the front of the queue. The moving of an aircraft that wants to progress to an empty edge ensures progress is made, and aircraft that want to progress to an empty edge do not need to wait in the queue for aircraft that

```
   Constants:                                                   Signature:
2    t_wait, the time to wait before assuming a                     for v ∈ V, e ∈ E, i ∈ I,                              24
         neighboring VN has no active clients                       replyColor ∈ {red, yellow, green}
4                                                                 Output bcast(ColorReply⟨v, replyColor, i⟩)_j            26
   State:                                                         Output VtoVsend(ReqVN⟨v, j, i, e⟩)_{j,dest}
6    queues, an array, indexed by vertex for each                 Output VtoVsend(ReplyVN⟨v, replyColor, i⟩)_{j,dest}     28
         vertex in j, of queues, initially empty, which           Output VtoVsend(UpdateVN⟨e, i⟩)_{j,dest}
8        contain elements of type (i, e) |i ∈ I, e ∈ E            Input brcv(ColorReq⟨v, i, e⟩)_j                         30
     aircraftOn, an array of aircraft in I or ⊥,                  Input brcv(PositionUpdate⟨e, i⟩)_j
10       initially ⊥, indexed by                                  Input VtoVrcv(ReqVN⟨v, vn, i, e⟩)_{src,j}               32
         edges in E for all edges in j                            Input VtoVrcv(ReplyVN⟨v, replyColor, i⟩)_{src,j}
12   vnReqs, a list, initially empty, of outstanding              Input VtoVrcv(UpdateVN⟨e, i⟩)_{src,j}                   34
         requests of the form (v, i, time)                        Internal advanceQueue(v, i)_j
14       v ∈ V, i ∈ I, time ∈ ℝ ≥ 0                               Internal reqTimeout(v, i)_j                             36
     bcastQ, a list, initially empty, of messages of
16       the form (v, i, color) where v ∈ V, i ∈ I              Trajectories:                                             38
         color ∈ {red, yellow, green}                             evolves
18   vtovQ, a list, initially empty, of messages of the             d(now) = 1                                            40
         form (obj, i, type) where i ∈ I, v ∈ V, e ∈ E              constant bcastQ, vtovQ, vnReqs,
20       obj ∈ {v, e, ⊥}                                                queues, aircraftOn                                42
         type ∈ {red, yellow, green, REQ, UPD}                    stops when
22   now : ℝ, the current real time                                 Any precondition is satisfied                         44
```

Figure 6-1: TIOA State for VSA vn_j for $j \in J$

want to progress to an occupied edge. Safety is ensured by only allowing an aircraft to progress onto an empty edge, and requiring that all previous aircraft have completely cleared a vertex before allowing the next aircraft to progress through it.

Now, for both the VSA and the Client, the FIFO Algorithm can be described in detail.

6.3.2 The VSA State

On VSA $vn_j \in VN$, with $j \in J$, we store an array of queues called $queues$ indexed by each vertex v such that $getRegion(v) = j$. Each of these queues can be accessed with $queues[v]$, and each is initially empty when the execution begins.

Additionally, the VSA stores an array of aircraft in I called $aircraftOn$ for each edge e such that $j \in getRegions(e)$. For any edge e' that has no aircraft on it, $aircraftOn[e'] = \bot$. Each of these aircraft can be accessed with $aircraftOn[e]$, and for each edge in the array, the value is initialized to $\bot$.

These two arrays represent the overall state of the region according to the VSA: all aircraft in the region are represented in the $aircraftOn$ array, indexed by their edge in the graph, and

the queues of aircraft that wish to be cleared to pass through each vertex are represented in the $queues$ array.

A number of message queues are also stored in the VSA state. First is the $bcastQ$, which stores messages to be sent using a bcast action. There is also a $vtovQ$, which stores messages to be sent using a VtoVsend action.

Since the VSA does not know whether or not a neighboring VSA is alive, it also stores a list of outstanding requests in $vnReqs$. These represent messages that have been sent out through a VtoVsend, but have not yet been responded to via a corresponding VtoVrcv. The VSA contains access to a global constant t_{wait}, and when the VSA has gone t_{wait} time without receiving a response to a message kept in $vnReqs$, it will assume the neighboring VSA is not alive, and respond appropriately as described in the next subsection.

Finally, as specified earlier, the VSA has a real-time clock variable now.

6.3.3 The VSA Actions

The actions of the VSA are the most significant part of the algorithm. The VSA vn_j must keep track of where each aircraft in its region $j \in J$ is and the state of the queue for each vertex in j. It must also communicate with VSAs in neighboring regions and aircraft in its own region.

Whenever an aircraft i in the region moves from one edge to another edge e (after being cleared to do so by the VSA), a_i will send a PositionUpdate$< e, i >$ message, which the VSA will receive with a brcv action. When the VSA receives the message, it uses $getRegions(e)$ to determine what regions e is in. It updates $aircraftOn[e]$ to be i, and if e is in a second region, adds a message to $vtovQ$. Then, vn_j sends an update to that region's VSA, outputting a VtoVsend to the VSA vn_{dest} with an UpdateVN$< e, i >$ message and removing the message from the $vtovQ$. It also clears the state of $aircraftOn[e']$ for any edges $e' \neq e$ such that $aircraftOn[e'] = i$.

When vn_j receives an UpdateVN$< e, i >$ message on the VtoVrcv channel from a neighboring VSA, it updates the state of $aircraftOn[e]$ to i. While having the VSA know about incoming aircraft does not impact correctness to any significant degree, updating a neighboring VSA about an edge that has been cleared is quite important, so that progress can occur.

The other message from an aircraft that the VSA vn_j must handle is the ColorReq$< v, i, e >$ message which the VSA will receive on the brcv channel. The message includes the aircraft i

```
Actions:
  Input brcv(PositionUpdate⟨e, i⟩)_j
  Effect:
    for all e′ ∈ E such that j ∈ getRegions(e′)
      if aircraftOn[e′] = i
        aircraftOn[e′] ← ∅
        if |getRegions(e′)| = 2
          vtovQ.append((e′, ∅, UPD))
    for all u ∈ getRegions(e)
      if u = j then
        aircraftOn[e] ← i
      else
        vtovQ.append((e, i, UPD))

  Output VtoVsend(UpdateVN⟨e, i⟩)_{j,dest}
  Precondition:
    (e, i, UPD) ∈ vtovQ ∧
    dest = (getRegions(e) -j)
  Effect:
    vtovQ.remove((e, i, UPD))

  Input VtoVrcv(UpdateVN⟨e, i⟩)_{src,j}
  Effect:
    aircraftOn[e] ← i

  Input brcv(ColorReq⟨v, i, e⟩)_j
  Effect:
    if getRegion(v) = j then
      queues[v].append((i, e))
      if (front(queues[v]) = i ∧
        aircraftOn[e] = ∅ )then
        color = green
      else
        color = red
      bcastQ.append((v, i, color))
    else
      vnReqs.append((v, i, now))
      vtovQ.append((v, i, REQ))

  Output bcast(ColorReply⟨v, replyColor, i⟩)_j
  Precondition:
    (v, i, replyColor) ∈ bcastQ
  Effect:
    bcastQ.remove((v, i, replyColor))

  Output VtoVsend(ReqVN⟨v, j, i, e⟩)_{j,dest}
  Precondition:
    (v, i, e, REQ) ∈ vtovQ ∧
    dest = getRegion(v)
  Effect:
    vtovQ.remove((v, i, e, REQ))

  Input VtoVrcv(ReqVN⟨v, vn, i, e⟩)_{src,j}
  Effect:
    if getRegion(v) = j then
      queues[v].append((i, e))
      if (front(queues[v]) = i ∧
        aircraftOn[e] = ∅ )then
        color = green
      else
        color = red
      vtovQ.append((v, i, color))

  Output VtoVsend(ReplyVN⟨v, replyColor, i⟩)_{j,dest}
  Precondition:
    (v, i, replyColor) ∈ vtovQ ∧
    replyColor ∈ {red, yellow, green} ∧
    dest = getRegion(v)
  Effect:
    vtovQ.remove((v, i, replyColor))

  Input VtoVrcv(ReplyVN⟨v, replyColor, i⟩)_{src,j}
  Effect:
    vnReqs.remove((v, i, [now -t_wait, now]))
    bcastQ.append((v, i, replyColor))

  Internal advanceQueue(v, i)_j
  Precondition:
    getRegion(v) = j ∧
    front(queues[v]) = (i, e) ∧
    ∃ v′ ≠ v such that aircraftOn[(v, v′)] = i
  Effect:
    queues[v].remove((i, e))
    (i′, e′) ← earliest elt in queues[v] such that
              aircraftOn[e′] = ∅
    queues[v].moveToFront((i′, e′))

  Internal reqTimeout(v, i)_j
  Precondition:
    (v, i, now -t_wait) ∈ vnReqs
  Effect:
    vnReqs.remove((v, i, now -t_wait))
    bcastQ.append((v, i, yellow))
```

Figure 6-2: TIOA Actions for VSA vn_j for $j \in J$

which is making the request, the edge e that it wishes to move onto, and the vertex v that it must pass through to do so. Now, depending on whether the request is for a vertex in region j or not, there are two possible ways for vn_j to handle it.

If the region of v is j, then vn_j will be able to handle the request without a neighbor. It does so by first adding (i, e) to $queues[v]$. If (i, e) is at the front of the queue, the reply color is green, which means that i is clear to progress to e, otherwise the color is red, and the aircraft must hold on its current edge until cleared. Then, by adding an appropriate message to $bcastQ$, vn_j will output a ColorReply$< v, replyColor, i >$, where v is the vertex requested, i is the aircraft that requested it, and $replyColor$ is the color signifying whether or not i is clear to move through v.

If the region of v is not j, and is actually j', then the request is forwarded to vn_j's neighbor $vn_{j'}$, by having vn_j send a ReqVN$< v, n, i, e >$ message to vn_{dest} with $dest = j'$. The VSA vn_j stores a copy of that message in the $vnReqs$ list. The VSA vn_{dest} chooses a reply color in the same way that VN_n did in the preceding paragraph, and responds with a ReplyVN$< v, replyColor, i >$ message. Upon receiving such a message, vn_j then forwards the $replyColor$ to the appropriate aircraft just as if it was determined by vn_j and not by vn_{dest}, and removes its copy of the message from $vnReqs$.

Finally, the VSA has two internal procedures. When a message kept in $vnReqs$ has gone more than t_{wait} without a reply, the reqTimeout procedure will remove it from $vnReqs$ and reply to the aircraft i with the yellow $replyColor$, meaning that the neighboring VSA is not alive, and that i should wait until it enters the neighboring region to request clearance. When the aircraft i that has clearance to pass through v has moved onto the requested edge e, the queue must be updated with the advanceQueue procedure, which dequeues (i, e), advancing the next request to pass through a vertex onto a currently empty edge onto the front of the queue.

6.3.4 The Client State and Actions

The client program running on the aircraft a_i is simpler than the code running on the VSAs. It needs to keep track of what region i is in and what edge i is on, storing them in $currentRegion$ and $currentEdge$ respectively. When the client starts, $currentRegion \leftarrow region(v_{takeoff})$, and $currentEdge \leftarrow (v_{takeoff}, v')$.

The client a_i keeps track of two edges related to the movement of the aircraft: $nextEdge$ rep-

```
Constants:
 2   t_retry, the time to retry a ColorReq after
             receiving a red or yellow response
 4   nextPath(e) :=
             choose e' ∈ {(v, v')|v = vertexAfter(e)}
 6           a nondeterministic function that
             returns the next edge that i wishes
 8           to move onto

10 Signature:
         for v ∈ V, e ∈ E
12       replyColor ∈ {red, yellow, green}
     Output bcast(ColorReq⟨v, i, e⟩)_i
14   Output bcast(PositionUpdate⟨e, i⟩)_i
     Output advance(e)_i
16   Input brcv(ColorReply⟨v, replyColor, i⟩)_i
     Input GPSinput(j, e)_i, j ∈ J
18   Internal progress_i

20 State:
     currentRegion, a region in J, initially j_NULL
22   currentEdge, an edge in E, initially e_NULL
     bcastPos, a boolean, set to true in order to
24           broadcast i's position, initially false
     nextEdge, an edge in E that i
26           wishes to move to, initially e_NULL
     destination, an edge in E that i
28           is currently moving to, initially e_NULL
     t_reply, the time ∈ ℝ when a_i last received a
30           ColorReply message, initially ∞
     now : ℝ, the current real time
32
   Trajectories:
34   evolves
       d(now) = 1
36     constant currentRegion, currentEdge,
             bcastPos, nextEdge, destination, t_reply
38   stops when
       any precondition is satisfied

   Actions:
     Input GPSinput(r, e)_i                                    42
     Effect:
       if (r ≠ currentRegion ∨ e ≠ currentEdge) then           44
         bcastPos = true
       currentRegion ← r                                       46
       currentEdge ← e
                                                               48
     Output bcast(PositionUpdate⟨e, i⟩)_i
     Precondition:                                             50
       bcastPos = true ∧
       e = currentEdge                                         52
     Effect:
       bcastPos = false                                        54

     Output advance(e)_i                                       56
     Precondition:
       e = destination ∧                                       58
       destination ≠ null ∧
       currentEdge ≠ destination                               60

     Internal progress_i                                       62
     Precondition:
       nextEdge = null ∧ currentEdge = destination             64
     Effect:
       destination ← null                                      66
       nextEdge ← nextPath(currentEdge)
       t_reply ← 0                                             68

     Output bcast(ColorReq⟨v, i, e⟩)_i                         70
     Precondition:
       (t_reply + t_retry ≤ now) ∧                             72
       v = vertexAfter(currentEdge) ∧
       e = nextEdge                                            74
     Effect:
       t_reply ← ∞                                             76

     Input brcv(ColorReply⟨e, replyColor, i⟩)_i                78
     Effect:
       if (replyColor = green) then                            80
         destination ← nextEdge
         nextEdge ← null                                       82
         t_reply ← ∞
       else                                                    84
         t_reply ← now
```

Figure 6-3: TIOA Code for Client Automaton a_i for $i \in I$

resents the next edge that a_i wants to move to, and $destination$ represents an edge that a_i is moving to, as it has already output an advance to do so. The nondeterministic movement of the aircraft is abstracted into the function $nextPath$, which will return the next edge that a_i wishes to move to.

Whenever the client receives a GPSInput$(j, e)_i$, the client a_i updates its $currentRegion$ and $currentEdge$ to j and e respectively, and sets $bcastPos$ to true, which will trigger a PositionUpdate$< e, i >$ message to be output through a bcast action, with $e = currentEdge$. The local VSA will receive this message and update a_i's position in its state.

The other actions the client takes are related to moving the aircraft from edge to edge. After the aircraft has progressed to a new edge, the internal procedure progress will be called, which will reset the state of $destination$, and use the nondeterministic $nextPath$ function to determine what the new $nextEdge$ should be. It also sets t_{reply} to 0, which will trigger a ColorReq message.

The ColorReq$< v, i, nextEdge >$ message requests clearance to pass through the vertex v onto $nextEdge$ from the VSA in $currentRegion$. The VSA should then reply with a ColorReply$< nextEdge, replyColor, i >$ message; if $replyColor$ is green, a_i sets its $destination \leftarrow nextEdge$, and outputs an advance$(e)_i$ action to $RealWorld$ with $e = destination$ which allows i to move to $destination$ in $RealWorld$. Otherwise, it sets $t_{reply} \leftarrow now$, waiting t_{retry} before trying another ColorReq.

Such a polling procedure is not necessary for correctness of the FIFO Algorithm given our assumptions of perfect reliability for the VSAs. We could simplify the algorithm by having a_i only request a vertex once, and having the VSA wait to respond until i has reached the front of the queue. Were the assumptions on VSA reliability to be weakened, polling the VSA and expecting an immediate response could be a useful indicator on whether the local VSA is alive or not.

6.4 Proofs of Safety and Progress

Returning our attention to the system requirements, there are two that we must prove that the algorithm satisfies: safety and progress.

Recall the safety requirement, which was derived in Theorem 5.3, that no two aircraft may

occupy the same edge or progress through the same vertex at any time. Using our algorithm, we can prove this theorem about safety:

Theorem 6.1

The FIFO Algorithm ensures that no two aircraft ever occupy the same edge or attempt to pass through the same vertex at the same time.

Proof

We know that the internal Takeoff action in $GraphRealWorld$ does not allow any aircraft to enter the system on an edge that is occupied by another aircraft. Therefore, the FIFO algorithm must simply ensure that if an advance$(e)_i$ action is output by a_i to $GraphRealWorld$ then, for all aircraft $i' \in I$, at the time of the advance$(e)_i$ action, all $aircraft[i'].e \neq e$.

As specified in line 53 of the code for client a_i, a advance output must be preceded by the setting of $destination$ to some edge, which can only occur, by line 76, after a ColorReply message is received with $replyColor =$ green.

The VSA vn_j, with $j \in J = a_i.currentRegion$ will only send such a message, as specified in line 40 of the code for VSA vn_j, as well as line 32 of the same code, if the aircraft $i \in I$ made a previous ColorReq, and i is in the front of the queue for the vertex $v \in V$ such that $e = (v, v')$ and $aircraftOn[e]$ is empty.

Since the VSA vn_j knows, through the specification of the PositionUpdate message, the location of every aircraft $i' \in I$ such that $aircraft[i'].region = j$ in $GraphRealWorld$, either its state is accurate, and the theorem is proven, or there actually is an aircraft i' such that $aircraft[i'].e = e$ in $GraphRealWorld$, and the update has not yet been broadcast to vn_j.

This second case, though, is impossible, since if such an update has not yet been broadcast, i' must have been cleared to move through v to advance to e, which contradicts the requirement in line 30 of the code for vn_j, that $front(queues[v]) = i$, and the ColorReply message with $replyColor =$ green could not have been sent to (and received by) a_i.

Therefore, i could only progress through v to e if both e is empty, and i is on the front of the VSA vn_j's queue for v, and the next aircraft will only be cleared to pass through v only after a_i has sent a PositionUpdate to vn_j which means i is through v and already on e.

Since the FIFO algorithm never allows an aircraft to progress onto a non-empty edge, and $GraphRealWorld$ does not allow an aircraft to start on an edge that is occupied, the algorithm ensures that no two aircraft will ever occupy the same edge. Since the FIFO algorithm only allows the aircraft on the front of a vertex's queue to pass through that vertex, and only allows the next aircraft to progress once it has received confirmation that the vertex is clear, it ensures that no two aircraft attempt to pass through the same vertex at the same time.

Q.E.D.

Now, I prove a corollary showing that the FIFO algorithm ensures safety in the continuous system as well.

Corollary 6.2

Assume that the Continuous VSA Layer can be correctly emulated by the Physical Network Layer such that the states of $RealWorld$ for the two are equal throughout the execution.

Then, the FIFO Algorithm guarantees that in $RealWorld$ of the Physical Network Layer, all aircraft are separated at all times by d_{hsep} horizontally, d_{vsep} vertically, or both.

Proof

We know, from Theorem 6.1, that the FIFO Algorithm ensures that no two aircraft ever occupy the same edge or attempt to pass through the same vertex at the same time in $GraphRealWorld$. This means, by Theorem 5.3, that in a corresponding execution of the Continuous VSA Layer, that in $RealWorld$ all aircraft are separated at all times by d_{hsep} horizontally, d_{vsep} vertically, or both. Since we assume at all times on a corresponding execution, the state of $RealWorld$ in the Physical Network Layer is equal to the state of $RealWorld$ in the Continuous VSA Layer, I conclude that in $RealWorld$ of the Physical Network Layer, all aircraft are separated at all times by d_{hsep} horizontally, d_{vsep} vertically, or both.

Q.E.D.

Now recall from Section 6.1 that our progress requirement is that given a set of aircraft that wish to move onto an an empty edge e (through vertex v), one will be able to do so in bounded time. Using the FIFO Algorithm, we can prove this theorem about progress:

Theorem 6.3

Given a set of aircraft $\{i_1, i_2, ..., i_n\}$ on incident edges to vertex v that wish to progress to empty edge e which is an outgoing edge from v, the FIFO Algorithm ensures that one of these aircraft will be able to progress onto edge e in bounded time.

Proof

The proof is relatively straightforward. Assume, in the worst case, that all outgoing edges are empty, and that the first aircraft, say i_1, that wishes to progress to e is behind the aircraft that wish to progress to all other edges in the queue, which is stored as $queues[v]$. Therefore, for each of the $|queues[v]| - 1$ aircraft before i_1, I calculate how long it will take for each to pass through the vertex v.

When the VSA vn_j in $j \in J = getRegion(v)$ has received the PositionUpdate saying that an aircraft has moved to the next edge, it will advance the queue to the next aircraft that wishes to progress through v onto an empty edge. It will take at most t_{retry} + d for the VSA to receive a request from that next aircraft, and that aircraft will receive the request in time d. Then, that aircraft will output a advance action to $GraphRealWorld$, which will move the aircraft to an empty edge in time $t_advance$. It will take at most ϵ time after that before the aircraft receives its next GPSinput, and will therefore send a PositionUpdate to vn_j' where $j' \in \{j\} \cup nbrs(j)$, which will arrive at vn_j in at most $2d$ time, and the process will repeat for the next aircraft.

Therefore, each aircraft before i_1 takes at most $4d + t_{retry} + t_advance + \epsilon$ time to clear the vertex v, and i_1 will move onto empty edge e within at most $|queues[v]|$ of these steps. The upper bound on i_1 moving to e is therefore $|queues[v]| * (4d + t_{retry} + t_advance + \epsilon)$.

Why is the worst case that all outgoing edges are empty? The algorithm ignores the request of any aircraft in the queue that wishes to progress to an occupied edge until that edge is empty. If all edges are empty, then all aircraft in the queue before i_1 will have to progress through the queue before i_1 is allowed. If some of the edges are occupied, i_1 could have to wait for fewer aircraft to progress.

The discussed case, where all outgoing edges are empty, is therefore the worst, and the above upper bound on i_1's advancing to e holds.

Q.E.D.

6.5 Chapter Summary

In this chapter I have:

- Presented an algorithm which allows the VSAs to control when aircraft move from one edge to another in our graph representation of the air traffic control system.
- Shown that the algorithm satisfies two important properties:
  - Safety, since no two aircraft will occupy the same edge or progress through the same vertex at any time.
  - Progress, since given a given a set of aircraft that wish to move onto an an empty edge, one will be able to do so in bounded time.

The algorithm is extensible, though, and I will continue using it in the next chapter in order to introduce a measure of efficiency to the algorithm's desired properties. Where in this chapter we use a simple queue to determine which aircraft is given clearance to pass through a vertex, in the next chapter I will discuss how to specialize this process to give preference to aircraft that need to be given clearance first.

Note that there is nothing special about using a first-in-first-out queue to ensure safety, so the algorithm could have decided which aircraft gets clearance randomly, nondeterministically, or using some other method with no impact on the safety of the system. We rely on this fact to know that the algorithm in the next chapter will also satisfy the safety requirements, as it merely changes the method to decide which aircraft gets clearance.

Chapter 7

The Heuristic Priority Algorithm

In the previous chapter, I presented an algorithm for air traffic control that focuses on safety, that is, ensuring that no two aircraft are in the bounding area corresponding to the same edge or vertex at the same time. I also proved that the algorithm has a desired progress property, that given a number of aircraft that wish to progress onto an empty edge, one will be able to do so in bounded time. While this means that each vertex will be as locally efficient as possible, no claims have been made about the efficiency of the system as a whole.

This chapter will make small modifications to the FIFO Algorithm that will result in a system that has improved global efficiency while maintaining the nondeterminism inherent in a free-flight air traffic control system. Absolute proofs of global efficiency will not be possible due to the possibility of aircraft making decisions that disrupt that efficiency. Nonetheless, we can still determine a *heuristic* for a best-effort improvement of global efficiency.

I call the modified algorithm the *Heuristic Priority Algorithm*, as it replaces the first-in-first-out queue at each vertex from the FIFO Algorithm with a *priority queue*. The front element in a priority queue is the element with the highest *priority*, which we will determine using a greedy heuristic.

This chapter will begin by discussing efficiency in the system, including why it is impossible to find the most efficient choices of aircraft to give clearance to in this free-flight system. I will then choose the heuristic that will be used, discussing a number of different options before deciding on a single one. Finally, I will show how it can be integrated into the FIFO Algorithm to provide better global efficiency in the system.

7.1 Efficiency in a Free-Flight System

In the current air traffic control system, entire routes are planned by controllers in order to get an aircraft from their origin to their destination with minimal interruption of progress. Due to this pre-planning, when factors such as weather or ground delays occur, entire routes must be changed, flights need to be cancelled, and passengers get inconvenienced. This is because the pre-planned system cannot react effortlessly to small changes in the same way that a free-flight system can.

While that benefit of a free-flight air traffic control system is important, the efficiency of the aircraft routes in a free-flight system may suffer. One problem with efficiency in this free-flight model of air traffic control is that in many situations, we have no idea where an aircraft wishes to go next. If we assume each aircraft uses a shortest-path algorithm, we could route them according to that assumption. If an aircraft deviates from that assumption, though, it could create far-reaching effects similar to the ones in the pre-planned system.

Additionally, since we are attempting to create a distributed air traffic control system, it becomes more difficult to intelligently decide how to move an aircraft when the state of its final destination is unknown to us. It is possible that clearing aircraft i to pass over Chicago before aircraft k will result in i having to wait longer in Los Angeles, while if we cleared k first, neither flight would encounter any traffic at all, moving directly to their destinations.

Therefore, it would be neither possible nor prudent to attempt to design an algorithm for which we can prove global efficiency properties in this free-flight system, and we must resort to a best-effort attempt to prioritize some aircraft over others using a local, greedy heuristic.

7.2 The Time to Arrival Heuristic

In order to add efficiency to our system, we must ask the question, ”What type of efficiency do we wish to enforce?” This is a difficult question, and there may be a number of good answers to it, but I would like to propose one heuristic that we can use in this section. I call this heuristic the *Time to Arrival Heuristic*.

Since we have no control over the path length of an aircraft (one may choose to take a longer path due to congestion, weather, or turbulence), enforcing efficiency of total distance travelled

would be impossible. Attempting to avoid traffic would be possible locally, but may result in undesirable global behavior, and again, the free-flight aircraft may not wish to cooperate with this type of guidance.

One property that is important in air traffic control (which we have not yet seen in our model), is a scheduled *arrival time*. The goal of air traffic controllers is not to get an aircraft to its destination as quickly as possible, but to get an aircraft to its destination as close before the scheduled arrival time as possible. Arriving early is of little benefit to an aircraft, as it must wait in terminal control until it has been cleared to land. Therefore, our efficiency goal should be *to get an aircraft* i *to its landing vertex* $v_landing$ *as closely to its arrival time as possible.*

The one time our air traffic control algorithm needs to make a decision that effects this goal is when multiple aircraft, say i and k, wish to progress through a vertex v to empty edges e_1 and e_2 (which could be the same edge) respectively. In this case, we want to prioritize the aircraft which has less extra time to get to its destination by its arrival time. In other words, we prioritize the aircraft for which the ratio of the distance remaining to its destination and the distance it can travel in the time remaining to reach its destination is greatest. I call this value the *Time to Arrival Heuristic*, or TTA:

$$TTA = \frac{dist(loc(vertexAfter(currentEdge)), loc(v_landing))}{(t_arrival - now)|\frac{\Delta(loc(vertexAfter(currentEdge)))}{\Delta(now)}|} \tag{7.1}$$

The numerator of this value is the straight-line distance from the aircraft's next vertex to its destination, and the denominator is the time remaining until the scheduled arrival time multiplied by the total change in distance over time. It should be clear that the numerator is a *lower* bound on the distance the aircraft must travel, as it must travel within the bounding areas, and cannot, in most cases, take a direct path to its destination.

Normally, the value of the function will be below 1, as the lower bound on the distance to be travelled should be less than the amount of distance that can be travelled in that time. Valued at or close to 1, the aircraft will be barely able to reach its destination by the arrival time at its current rate of travel, so it should be given priority clearance by the VSAs.

There are a number of benefits to this heuristic. First, it is easy and efficient for the clients to calculate its value and send it to the VSAs. Second, it does not require adding any new communications to the VSA code; I merely need to add the value of the TTA to update messages.

```
   Constants:                                              Signature:
2    t_wait, the time to wait before assuming a               for v ∈ V, e ∈ E, i ∈ I, tta ∈ ℝ                    28
          neighboring VN has no active clients                 replyColor ∈ {red, yellow, green}
4                                                          Output bcast(ColorReply⟨v, replyColor, i⟩)_j                30
   State:                                                  Output VtoVsend(ReqVN⟨v, j, i, e, tta⟩)_{j,dest}
6    queues, an array, indexed by vertex for each          Output VtoVsend(ReplyVN⟨v, replyColor, i⟩)_{j,dest}        32
        vertex in j, of priority queues, initially         Output VtoVsend(UpdateVN⟨e, i⟩)_{j,dest}
8       empty, which contain elements of type (i, e)       Input brcv(ColorReq⟨v, i⟩)_j                              34
        |i ∈ I, e ∈ E, prioritized by values in ℝ          Input brcv(PositionUpdate⟨e, i, tta⟩)_j
10   aircraftOn, an array of aircraft in I or ⊥,           Input VtoVrcv(ReqVN⟨v, vn, i, e, tta⟩)_{src,j}            36
        initially ⊥, indexed by                            Input VtoVrcv(ReplyVN⟨v, replyColor, i⟩)_{src,j}
12      edges in E for all edges in j                      Input VtoVrcv(UpdateVN⟨e, i⟩)_{src,j}                     38
     vnReqs, a list, initially empty, of outstanding       Internal advanceQueue(v, i)_j
14      requests of the form (v, i, time)                  Internal reqTimeout(v, i)_j                               40
        v ∈ V, i ∈ I, time ∈ ℝ ≥ 0
16   bcastQ, a list, initially empty, of messages of       Trajectories:                                             42
        the form (v, i, color) where v ∈ V, i ∈ I            evolves
18      color ∈ {red, yellow, green}                           d(now) = 1                                            44
     vtovQ, a list, initially empty, of messages of the        constant bcastQ, vtovQ, vnReqs,
20      form (obj, i, type) where i ∈ I, v ∈ V, e ∈ E              queues, aircraftOn, TTA                           46
        obj ∈ {v, e, ⊥}                                      stops when
22      type ∈ {red, yellow, green, REQ, UPD}                  Any precondition is satisfied                         48
     TTA, an array of real numbers indexed by
24      aircraft in I, initialized to 0
     now : ℝ the current real time
```

Figure 7-1: TIOA State for VSA vn_j for $j \in J$

Finally, it will satisfy the goal of getting aircraft to their destination as close to their arrival time as possible.

7.3 Algorithm

For this algorithm, we must add a small amount of information to the algorithm already developed in Chapter 6. The edits will be minor, though, and should look quite similar to the code you have seen already.

The arrival time $t_arrival$ is decided on by the client, and can change when needed. The client uses this to report its TTA to the VSA in its region, which then uses it to prioritize aircraft.

7.3.1 The VSA State

To implement the Time to Arrival Heuristic in the VSA state, we change the type of data stored in the array $queues$ from regular queues to *priority queues*. With a priority queue, instead of

inserting an object and retrieving it in a first-in-first-out manner, we insert an object with a given *priority*, and the front of the priority queue will be the object with the highest *priority*.The VSA also stores an array of reals, $TTA[i]$ indexed by $i \in I$, that gets updated upon PositionUpdate messages.

7.3.2 The VSA Actions

The VSA Actions need to be modified slightly to implement the Time to Arrival Heuristic. Specifically, when advanceQueue gets called to determine which aircraft is next in the priority queue, instead of simply looking at the front of the priority queue, all elements are removed, the TTA of each is looked up, and they are inserted into a new priority queue. This ensures that all priorities are kept current, so the highest priority aircraft is on the front.

Also, whenever position updates occur for any aircraft i, the $TTA[i]$ value is updated. When one VSA updates a neighboring VSA on an aircraft's position, it looks up that aircraft's TTA and sends it.

7.3.3 The Client State and Actions

The client must simply keep track of its desired arrival time $t_arrival$, calculate their TTA upon each PositionUpdate message sent to the VSA in $currentRegion$, and append the value of their TTA to that PositionUpdate message. In order to calculate its TTA, the client must also keep track of how far it has traveled since it entered the system.

7.4 Chapter Summary

In this section, I have:

- Discussed how efficiency can be measured in a free-flight air traffic control system.
- Created a heuristic called the Time to Arrival Heuristic, a measure of how much extra time the aircraft has to get to its destination before its arrival time.
- Modified the FIFO Algorithm by prioritizing the vertex queues, resulting in the Heuristic Priority Algorithm.

```
Actions:
  Input brcv(PositionUpdate⟨e, i, tta⟩)_j
  Effect:
    TTA[i] = tta
    for all e′ ∈ E such that j ∈ getRegions(e′)
      if aircraftOn[e′] = i
        aircraftOn[e′] ← ∅
        if |getRegions(e′)| = 2
          vtovQ.append((e′, ∅, UPD))
    for all u ∈ getRegions(e)
      if u = j then
        aircraftOn[e] ← i
      else
        vtovQ.append((e, i, UPD))

  Output VtoVsend(UpdateVN⟨e, i, tta⟩)_{j,dest}
  Precondition:
    (e, i, UPD) ∈ vtovQ ∧
    dest = (getRegions(e) -j) ∧
    tta = TTA[i]
  Effect:
    vtovQ.remove((e, i, UPD))

  Input VtoVrcv(UpdateVN⟨e, i, tta⟩)_{src,j}
  Effect:
    aircraftOn[e] ← i ∧
    TTA[i] ← tta

  Input brcv(ColorReq⟨v, i, e⟩)_j
  Effect:
    if getRegion(v) = j then
      queues[v].append((i, e), TTA[i])
      if (front(queues[v]) = i ∧
        aircraftOn[e] = ∅ )then
        color = green
      else
        color = red
      bcastQ.append((v, i, color))
    else
      vnReqs.append((v, i, now))
      vtovQ.append((v, i, REQ))

  Output bcast(ColorReply⟨v, replyColor, i⟩)_j
  Precondition:
    (v, i, replyColor) ∈ bcastQ
  Effect:
    bcastQ.remove((v, i, replyColor))

  Output VtoVsend(ReqVN⟨v, j, i, e, tta⟩)_{j,dest}
  Precondition:
    (v, i, e, REQ) ∈ vtovQ ∧
    dest = getRegion(v) ∧
    tta = TTA[i]
  Effect:
    vtovQ.remove((v, i, e, REQ))

  Input VtoVrcv(ReqVN⟨v, vn, i, e, tta⟩)_{src,j}
  Effect:
    if getRegion(v) = j then
      queues[v].append((i, e), tta)
      if (front(queues[v]) = i ∧
        aircraftOn[e] = ∅ )then
        color = green
      else
        color = red
      vtovQ.append((v, i, color))

  Output VtoVsend(ReplyVN⟨v, replyColor, i⟩)_{j,dest}
  Precondition:
    (v, i, replyColor) ∈ vtovQ ∧
    replyColor ∈ {red, yellow, green} ∧
    dest = getRegion(v)
  Effect:
    vtovQ.remove((v, i, replyColor))

  Input VtoVrcv(ReplyVN⟨v, replyColor, i⟩)_{src,j}
  Effect:
    vnReqs.remove((v, i, [now -t_wait, now]))
    bcastQ.append((v, i, replyColor))

  Internal advanceQueue(v, i)_j
  Precondition:
    getRegion(v) = j ∧
    front(queues[v]) = (i, e) ∧
    ∃ v′ ≠ v such that aircraftOn[(v, v′)] = i
  Effect:
    queues[v].remove((i, e))
    newPQ ← empty priority queue
    for each (i′, e′) ∈ queues[v]
      newPQ ← newPQ ∪ ((i′, e′), TTA[i′])
    queues[v] ← newPQ

  Internal reqTimeout(v, i)_j
  Precondition:
    (v, i, now -t_wait) ∈ vnReqs
  Effect:
    vnReqs.remove((v, i, now -t_wait))
    bcastQ.append((v, i, yellow))
```

Figure 7-2: TIOA Actions for VSA vn_j for $j \in J$

Chapter 8

Conclusions

In this thesis, I have modeled the air traffic control system in a way that allowed us to apply recent techniques in distributed algorithms to the problem. This chapter will conclude the thesis with some final remarks about the work that has been done and the related research that can follow from this paper.

I will begin by explaining the contributions to research in algorithmic air traffic control and distributed algorithms that this thesis has made. Knowing that, I will be able to suggest some further lines of research that are related to this work. Finally, I will conclude the thesis with some personal thoughts and remarks about the research.

8.1 Contributions

As I stated at the opening of the thesis, its contributions can be seen in two areas: research into the specific system of air traffic control, and research in distributed algorithms, namely, in Virtual Stationary Automata.

In the field of air traffic control, this thesis has:

- Discussed some benefits and drawbacks of a free-flight air traffic control system, including:
  - The difficulty in planning efficient routes for a free-flight system
  - The ability of a free-flight system to react to small changes in the system

- The measures of efficiency that one could be interested in when designing an algorithm to control a free-flight system.

- Developed a formal model that can be used in many applications of air traffic control, specifically the modeling of the continuous motion of aircraft in designated bounding areas as the discrete transition between different edges in a graph representation.

- Presented two algorithms for performing algorithmic air traffic control, and proved that they are safe.

And in the field of distributed algorithms, this thesis has:

- Explained the history of algorithms designed to run on ad-hoc wireless networks, including the ability to coordinate effectively based on location data received from a GPS oracle.

- Presented a generalizable method of discretizing a problem which originally involved continuous motion.

- Designed two new algorithms using Virtual Stationary Automata to provide a stable location for storage, communication, and computation in a wireless network with highly dynamic topology.

8.2 Future Work

In this section, I would like to suggest some future lines of research that this thesis has opened up.

8.2.1 Implementation and Simulation of ATC

As I stated previously, a simulator for the Virtual Stationary Automata layer was implemented in Python in 2006. An obvious next step for the air traffic control algorithms in this paper would be to implement them in Python, using the simulator to observe performance of the system.

In this case, instead of having a single heuristic to determine how aircraft get cleared through a vertex, we could simulate multiple heuristics, observing the performance of each of them. This

would give powerful evidence whether or not the efficiency arguments I give in Chapter 7 will hold in a real-world implementation of the Heuristic Priority Algorithm.

An impediment to this research, though, is the simplicity of the movement model implemented in the simulator. The simulator does not currently support the type of nondeterministic movement assumed of the aircraft in the Known Path Model, making simulation of the algorithms impossible. If the ability to simulate the movement of our aircraft was implemented in the simulator, it would be straightforward to implement and test the algorithms in this paper.

8.2.2 ATC Without Known Paths

Another line of research would be to throw out the graph representation, designing an algorithm to control free-flight air traffic using a continuous movement model. While the graph representation allows us to describe the system in a computationally efficient manner, it is somewhat restrictive in the freedom pilots have to plan their routes.

Instead, we could use VSAs for collision avoidance, keeping track of the locations and velocities of all aircraft in the region. When two aircraft get close and appear to be in danger of colliding, the VSA mediates the conflict by instructing both pilots how to avert the collision.

One problem with this is that our assumptions about the network layer, specifically the perfect reliability and bound on message delivery, would need to be removed for this application. In our clearance-based system, only one aircraft can be allowed to enter a bounding area at a time, and a simplified view of the message delay never causes problems with safety. In a system that uses VSAs for collision avoidance, message delay becomes very important, as a delayed or dropped message could result in a violation of separation requirements, or even a collision.

Another possible ATC algorithm would involve allowing bounding areas to be created as needed. When an aircraft enters a region, it creates and reserves a bounding area with the VSA, which it can use until the aircraft leaves the region. This is an interesting solution to the problem, but could be computationally intensive and difficult to implement.

8.2.3 Other Applications for VSAs

One important aspect of VSA theory is how generally applicable the idea is to a ad-hoc networking problem. We can use a VSA layer any time it is impossible or too expensive to use physical

infrastructure in a wireless network, but we would still want the ability to communicate with a stable location in space.

For example, a military application could be the coordination of nearby troops on a battlefield. It is often difficult or impossible to deploy infrastructure in an active battlefield, but the communication between multiple groups of soldiers is essential to success. VSAs could be used to coordinate locations of different groups of soldiers, or to collaborate on a strategy involving nearby groups of soldiers who would be unable to communicate directly.

Commercial applications of Virtual Stationary Automata could involve physical social networking, where individuals leave notes and data for others at specific locations in space. Other applications, such as collaboration between users at a conference, or coordination of medical and police personnel during a disaster, would also be great uses of VSAs.

While these applications for VSAs would be interesting lines of research, the use of the Known Path VSA Layer can make many applications much easier and more intuitive. We saw how simple a distributed air traffic control algorithm could be when the motion is discretized using the bounding areas method, and this sort of continuous-to-discrete conversion can be used in a variety of new applications.

8.3 Closing Comments

I'd like to close by commenting on the two main aspects of my research: air traffic control and Virtual Stationary Automata. While I have stated what I believe the contributions of this thesis are to both domains, it remains to be said what importance it has in each.

Concerning air traffic control, the importance of the research is difficult to say. Right now there are many competing ideas on where the future of air traffic control will be, and nobody has any perfect answers on what the results will be. In general, a distributed solution for air traffic control is not likely to be adapted for commercial air travel. There is already a great deal of infrastructure in place, and the difficulties of adapting commercial travel to free-flight may be too great, with too little benefit.

On the other hand, I believe a solution like this would be extremely effective for air traffic control of civil aviation. The development of fast, fuel-efficient, small aircraft will cause a dramatic increase in the numbers of civil aviators in upcoming years, and an inexpensive automated so-

lution will need to be created to accommodate them. If aviation reaches a point where a large number of citizens commute to work using very-light aircraft, a solution such as the one in this paper would be quite effective: it has similarities to streets and roads that people are already comfortable navigating, and it requires no expensive infrastructure to deploy or maintain.

The future of Virtual Stationary Automata, in my opinion, is extremely bright. As wireless networks become ubiquitous in our everyday lives, the ability to coordinate with nearby clients will be more and more important. Using Virtual Stationary Automata, the development of innovative applications for the ad-hoc wireless networks in everyday life will be quicker, easier, and less expensive. The deployment of such applications will become trivial as well.

In my opinion, a stable, efficient implementation of the VSA layer will have a huge number of uses in the networking world. The steps we have taken so far are still early, but will prove extremely important to the continued development of such a useful, widely-applicable idea.

Chapter 9

Appendix

9.1 References

1. S. Dolev, S. Gilbert, N. A. Lynch, A. Shvartsman, and J. L. Welch. Geoquorums: Implementing atomic memory in mobile ad-hoc networks. In *17th International Symposium on Distributed Computing (DISC)*, 2003.

2. S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Virtual mobile nodes for mobile ad hoc networks. In *18th International Symposium on Distributed Computing (DISC)*, pages 230-244, 2004.

3. S. Dolev, S. Gilbert, L. Lahiani, N. A. Lynch, and T. A. Nolte. Timed Virtual Stationary Automata for Mobile Networks. In *9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, December, 2005.

4. F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric Ad-Hoc Routing: Of Theory and Practice. *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

5. J. C. Navas and T. Imielinski. Geocast - geographic addressing and routing. *Proceedings of the 3rd MobiCom*, 1997.

6. J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris. A Scalable Location Service for Geographic Ad Hoc Routing. *Proceedings of Mobicom*, 2000.

7. N. Lynch, S. Mitra, and T. Nolte. Motion coordination using virtual nodes. In *44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC 2005)*. Seville, Spain, December, 2005.

8. S. Dolev, L. Lahiani, N. Lynch, and T. Nolte. Self-Stabilizing Mobile Node Location Management and Message Routing. *SSS 2005: Seventh International Symposium on Self-Stabilizing Systems*, October 2005.

9. M. S. Nolan. *Fundamentals of Air Traffic Control.* Third Edition. Pacific Grove, CA: Brooks/Cole Publishing Company, 1999.

10. "Air Traffic Control - Wikipedia", [Online Document], 2006,

    Available HTTP: http://en.wikipedia.org/wiki/Air_traffic_control.

11. C. C. Freudenrich, PhD. "How Air Traffic Control Works" [Online Document]

    Available HTTP: http://electronics.howstuffworks.com/air-traffic-control.htm.

12. M. Brown and M. Spindel. The Traffic Light Problem Using Reactive Virtual Stationary Automata. 2006. Unpublished.

13. M. Brown, S. Gilbert, N. Lynch, C. Newport, T. Nolte, and M. Spindel. The Virtual Node Layer: A Programming Abstraction for Wireless Sensor Networks. Proceedings of the the International Workshop on Wireless Sensor Network Architecture (WWSNA), Cambridge, MA, April, 2007.

14. J. Krozel. *Free Flight Research Issues and Literature Search.* Prepared for NASA Ames Research Center, Moffett Field, CA. September, 2000

15. S. Dolev, S. Gilbert, E. Schiller, A. Shvartsman, and J. Welch. Autonomous Virtual Mobile Nodes. *Technical Report MIT-LCS-TR-992*, MIT CSAIL, Cambridge, MA, June 2005.

16. N. Lynch. I/O Automata: A model for discrete event systems. *In 22nd Annual Conference on Information Sciences and Systems, pages 29-38*, Princeton University, Princeton, N.J., March 1988.

17. N. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pages 137-151*. Vancouver, British Columbia, Canada, August 1987

18. D. Tulone. "Is it possible to ensure strong data guarantees in highly mobile networks?" In *Proceedings of the 5th Annual Mediterranean Workshop of Ad hoc Networks (MedHoc)*, June 2006.

19. D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. "The Theory of Timed I/O Automata". *Synthesis Lectures on Computer Science.* Morgan Claypool Publishers, 2006.