University of Manchester

Computer Science

Third Year Project Report

Air Traffic Control in Perfect Developer

Author: **Anjaney Menon** *Supervisor:* **Dr Richard Banach**

April 30, 2019

Abstract

As air traffic grows, so does the need for more robust and more efficient air traffic control. One way to handle this growth is by leveraging technology. Pioneering steps have been made towards automating air traffic control, especially in recent times. Although this has led to significant improvements, air traffic control is still, in many ways, hindered by human factors. This is because decisions are still mostly made by humans, therefore, unpredictable mistakes being made still persist.

To better understand the complexities involved in automating air traffic control, this project aims to build an application that is able to simulate an automated en route air traffic control system. Since the application will be simulating safety-critical software, it must be absolutely correct. To achieve this, the backend of the application has been built using a tool called Perfect Developer. Perfect Developer is a tool that can be used to built software that is mathematically proven to be correct. This makes it a very suitable choice for this project.

The verification results for the backend showed that, except for just one class, all other classes in the backend were completely correct. The class that could not be proven to be correct was tested many times to ensure that it functioned as expected.

Experiments were conducts to determine the effectiveness of the preventive measures that were assigned to randomly generated airplanes, with varying constraints on the randomness of the altitudes that these airplanes could have when entering the 'controlled' region. Although the application was not perfect, it was able to prevent all collisions from occurring. Furthermore, the application was also able to ensure that all airplanes were sufficiently separated from other airplanes during most of their journeys.

Acknowledgements

I would firstly like to thank my supervisor, Dr Richard Banach, for his continued and unwavering support and guidance throughout the course of my final year of undergraduate studies.

I would also like to thank my parents for constantly encouraging me, providing me with moral support, and for proof reading my report several times.

Contents

Chapter 1

Introduction

This chapter first explains the significance of undertaking this project. It then moves on to discussing the main aims and objectives for the project. Finally, an overview of the structure of the report has been presented towards the end of the chapter.

1.1 Motivation

Why does air traffic control (ATC) exist? Global air traffic has been doubling every 15 years since 1985 [12]. Therefore, it became unfeasible to hold pilots solely responsible for maintaining enough separation between every airplane in the sky and expedite the smooth flow of air traffic. To handle this challenge, the first national air traffic control centre was opened in the United States (Newark, New Jersey) in 1935 [35]. Since then, air traffic control has evolved into a complex hierarchical system [21]. Core responsibilities have, however, remained the same: to avoid collisions, and to facilitate the efficient and smooth flow of air traffic [7].

Air traffic control has, from the time it was conceived, been marred by problems arising due to inherent human limitations. Therefore, it is also subject to many of the challenges that other fields face with regard to its human workforce, such as the persistent probability of human errors, and consequent inefficiency. However, the job of an air traffic controller, unlike many others, particularly involves high-risk and safety-critical decisions being made; therefore, it has to be designed aiming at nil tolerance for mistakes and inefficiency. Numerous incidents have occurred directly attributable to

mistakes made by air traffic controllers. One of the most striking instances of this was the 1976 Zagreb mid-air collision [22]. The collision led to the death of 176 passengers and stands as the deadliest mid-air collision till today [3]. Furthermore, in recent times, there have also been shortages of air traffic controllers due to strikes, job-induced stress etc [6, 4].

To overcome shortages and human-induced errors, significant steps have been made towards automating air traffic control systems. For instance, in the US, the Federal Aviation Administration has implemented the NextGen project, which is an ongoing project to aid controllers in efficiently managing the expected growth in air traffic [10, 27]. Air traffic control systems, however, will remain largely manually controlled systems in the foreseeable future [16, 29].

The aim of this project is to better understand the complexities involved in fully automating air traffic control, and for determining thereby if it is possible for an application to take on the role of an air traffic controller entirely.

1.2 Objectives

The purpose of this project is to implement an application that would attempt to simulate an automated air traffic control system. This would involve proposing algorithms to predict and avoid potential collisions between airplanes. Logically, it would also be necessary for functionalities of the application to be formally verified. Furthermore, the application would also have to be feature-rich, in that it would have to allow the user to start simulations involving scenarios of his/her own choosing. It would also have to be very user-friendly and permit users to easily and quickly interact with the simulation. Lastly, the application must also provide numerical data on how well it has performed in avoiding potentially dangerous situations.

1.3 Report Structure

Chapter 2 focuses on providing a brief overview of how air traffic control works.

Chapter 3 puts forth the requirements that the application must meet (both functional and non-functional). It then presents a high-level descrip-

tion of the structure of the application. The chapter then ends with an overview of the mathematics behind one of the most important features of the application.

Chapter 4 starts with an overview of the main tools used for the creation of the application. It then discusses how some of the major features were implemented.

Chapter 5 provides information regarding the correctness of the application. It also evaluates the performance of the application.

Chapter 6 reflects on the outcome of the project by discussing what was learned, and what could have been done better.

Chapter 2

Background

This chapter begins by presenting fundamental information regarding air traffic control. It then moves on to explaining the various divisions of an air traffic control system. Following this, the chapter presents information on waypoints, an extremely important concept in air traffic control. Finally, the chapter ends with an overview of commercial airplanes.

The United States (US) has one of the busiest air spaces among all countries [17]. Their huge volume of air traffic calls for one of the most complex and robust air traffic control systems in the world to manage and oversee its airspace efficiently. Therefore, in this chapter, information regarding air traffic control has been largely based on practises in the US.

2.1 Introduction to Air Traffic Control

Air traffic control is a service provided by air traffic controllers, and involves directing airplanes both on the ground and in the air [7]. An air traffic controller's aim is to strike a delicate balance that ensures safety for all airplanes that they are controlling, while at the same time aiming to maximize efficiency as well [7].

To manage air traffic better, the airspace is often divided into regions. These regions are further divided into smaller regions, thus forming a hierarchy. For instance, as we can see from Figure 2.1, the US airspace is divided into **zones**, and each zone is further divided into **sectors** [21]. Zones are also divided into **T**erminal **R**adar **A**pproach **Con**trol (**TRACON**) airspaces, wherein each TRACON contains one or more airports, each having their

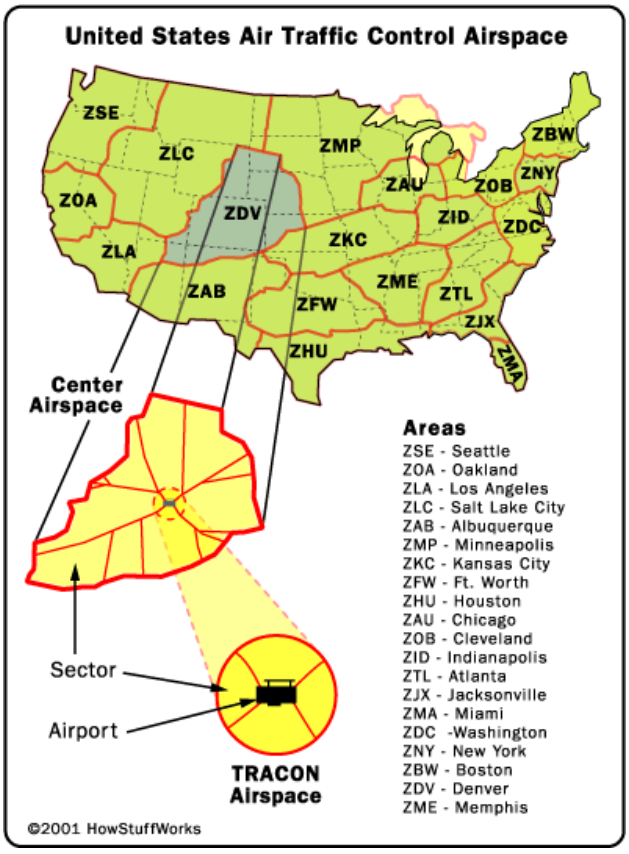


Figure 2.1: United States airspace hierarchy [21]

own airspace to oversee [21]. Every single region has a division of the air traffic control system that controls all airplanes that fly within it. Once an airplane moves out of a region, responsibility for it is delegated to the division that handles the region that the airplane is entering.

2.2 Air Traffic Control Divisions

There are four main air traffic control divisions [8]:

1. Air Traffic Control System Command Centre (ATCSCC)
2. Air Traffic Control Centres (ATCC)

3. Terminal Radar Approach Control (TRACON)

4. Air Traffic Control Towers (ATCT)

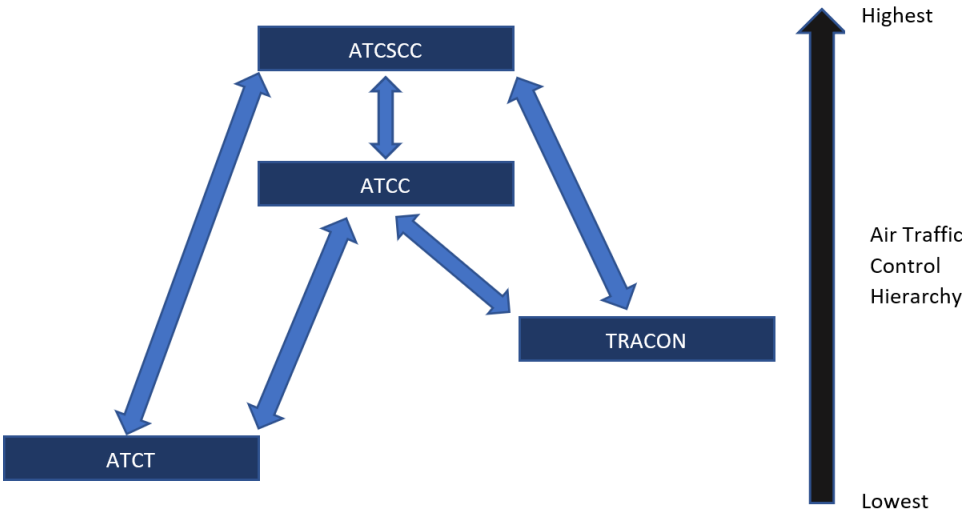


Figure 2.2: Air traffic control hierarchy (blue arrows represent bi-directional communication channels)

2.2.1 Air Traffic Control System Command Centre

The Air Traffic Control System Command Centre (ATCSCC) is the highest on the air traffic control system hierarchy and communicates directly with all other divisions (see Figure 2.2) [8, 21]. It oversees controlling all air traffic within zones that are experiencing bad weather, congested airspace or runway closures [9].

2.2.2 Air Traffic Control Centres

There is one Air Traffic Control Centre (ATCC) for each zone [8]. They manage all sectors within their zone, except TRACON airspace and local airport airspace [8]. As can be seen from Figure 2.3, ATCCs control airplanes that are en route to their destination airport [21, 7]. While flying in

ATCC controlled airspace, it is the responsibility of en route air traffic controllers (controllers that work in ATCCs) to ensure that all airplanes within sectors controlled by them maintain enough spacing between each other [7]. Currently, FAA laws have a different set of separation rules for various situations; however, it is generally the case that airplanes must maintain a vertical separation of approximately 300m, or a horizontal separation of approximately 9km (see Figure 2.4) [32, 31]. If these conditions are not met, the airplanes are said to be conflicting [26]. En route air traffic controllers are required to predict if/when airplanes could be conflicting and provide instructions to airplanes to travel at a different altitude or in some cases, alter their directions to avoid the potential conflict [21]. Sometimes, ATCCs control TRACON airspace as well. Therefore, ATCCs communicate both with TRACON controllers and ATCT controllers (see Figure 2.2).

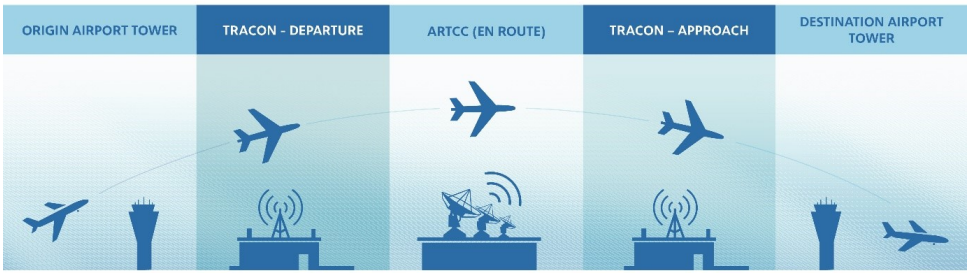


Figure 2.3: Flight profile of an average commercial airplane [38]

2.2.3 Terminal Radar Approach Control(TRACON)

TRACON airspaces are circular regions with diameters of about 80.5 km [21]. TRACON airspace usually contains multiple airports and form the link between these airports and ATCC controlled sectors. Controllers that oversees TRACON airspace handle airplanes that are departing or approaching an airport within their airspace (see Figure 2.3) [8].

2.2.4 Air Traffic Control Towers

These are the towers that all of us are familiar with and often see at airports. Controllers in ATCTs are responsible for ensuring that airplanes safely take-off and land [21]. They also oversee all airplanes on the ground [21]. The

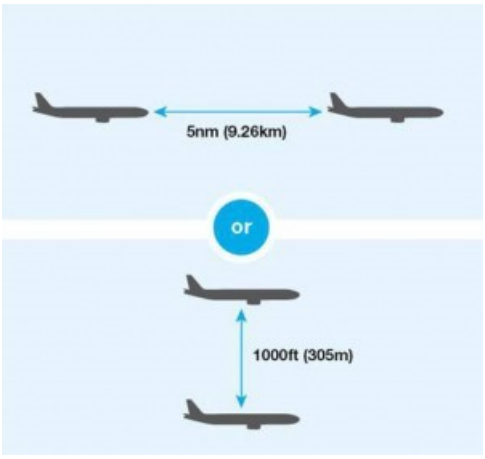


Figure 2.4: Separation rules [33]

circular airspace that they handle usually has a diameter of approximately 16km [21].

2.3 Waypoints

How are air traffic controllers able to pinpoint exactly where an airplane is, at any point in time? Air traffic controllers must have accurate information about where airplanes are, and in what direction they are headed. To achieve the required level of accuracy, commercial airplanes are required to travel through predetermined routes [5]. Like nodes in graphs, these routes are made up of points with very specific longitudes and latitudes called waypoints [5]. These waypoints are connected by straight paths called airways [30, 13]. Waypoints are often beacons that help pilots of each airplane know the exact coordinates of their airplane [5]. However, especially at sea, it is usually the case that waypoints are not physical objects at all [5]. There is no specific minimum distance between waypoints; however, they are often close to each other in TRACON airspaces, and much further apart in sectors.

2.4 An Overview of Commercial Aircraft

Commercial aircraft are used to transport both passengers and goods between airports. Most aircraft, and indeed most commercial aircraft, are

equipped with airborne collision avoidance systems [14, 11]. These collision avoidance systems detect if there are any airplanes that are dangerously close, and automatically change the airplane's altitude to reduce the risk of collision [11].

Commercial airplanes usually have a cruising speed of approximately 800km/hr to 1000km/hr [18]. Commercial airplanes try to avoid sharp changes to altitudes, so that passengers in the airplane are not disturbed. Therefore, commercial airplanes usually descend and elevate with an angle of just three degrees [39].

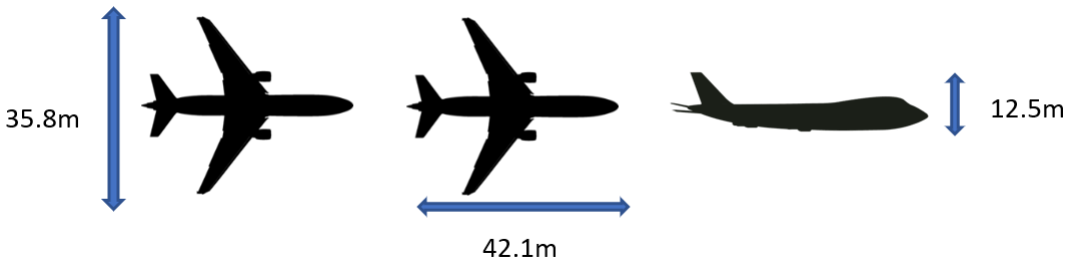


Figure 2.5: Dimensions of the Boeing 737-900 (wingspan: 35.8m, length: 42.1m and height: 12.5)

Commercial aircraft usually come in different sizes, ranging from the Dornier 328 [19], with a wingspan of 21m to the Airbus A380, with a wingspan of 80m [20]. The most popular commercial aircraft (based on number of units produced) is the Boeing 737(Figure 2.5 shows its dimensions) [2].

Chapter 3

Design

This chapter focuses on the design of the automated air traffic control system. It first discusses the scope of the project. Based on the scope, a set of functional and non-functional requirements are established. Following this, a very high-level description of how the simulation runs will be presented. The chapter will then move on to provide details about the architectural design of the system. Finally, a description of the mathematics behind some of the most important features that are part of the application will be provided towards the end.

3.1 Project Scope

As we saw in the previous chapter, air traffic control systems are immensely complex. Automating every division as a third-year project is not a viable solution. Therefore, although automating the four divisions would be exciting areas of research, this research project will focus on one important subset of work done by controllers in Air Traffic Control Centres (ATCC). Specifically, it will focus on the duties of en route air traffic controllers. The primary goal of the application will be to ensure that no airplane within the controlled region is conflicting with another airplane. ATCC seemed to be the most interesting division to automate, requiring significant ingenuity when it came to designing algorithms and even when using well-known algorithms. Additionally, it requires a thorough knowledge of vector geometry and basic calculus. Lastly, while cruising in sectors, the altitude of airplanes remain constant unless they must change it because of potential conflicts in

the future. Their speeds remain constant for the most part as well. Therefore, airplanes are more predictable during this part of their journey, which makes it easier to simulate.

3.2 Project Requirements

Requirements for this project were separated into functional and non-functional requirements. Functional requirements are those that specify what the application should be able to do, while non-functional requirements are those that can be used to assess the functionalities of the application. These requirements can then be further divided and grouped together based on their priority (low, high and medium).

Grouping the requirements made the development process more structured and robust. This consequently made implementing the application easier. Furthermore, this also allowed me to develop multiple versions of the application, where initial versions satisfied the high priority functional requirements, and subsequent versions gradually satisfied the rest until all of them were fulfilled. Doing things this way also provided a degree of security, because if something did not go as planned, a functional application would still be available for submission.

3.2.1 Functional Requirements

High Priority

Adding airplanes: A form must be available for the user to specify when an airplane will enter the controlled region, the altitude it will have when entering, and the route that it will take.

Collision/Conflict detection: The application must be able to detect potential conflicts or collisions between airplanes.

Assigning preventive measures: The application should be able to assign measures (specifically manoeuvres involving changes in altitude) to airplanes that are in danger of being in a conflicted status in the future.

Removing Airplanes: Airplanes that have collided or have exited the controlled region must be completely removed from the simulation.

Simulations timesteps: Timesteps of the simulation must equate to one second in the simulation environment.

Medium Priority

Adding waypoints: The user must have the option to include waypoints within the controlled region. These waypoints must be uniquely identifiable, and they must also have a certain minimum distance between each other.

Altering routes: The user must be able to alter the routes of all airplane within the controlled region during the simulation. The new route must be a sequence of waypoints with the point of exit being unchanged.

Random airplanes: The user must have the option to automatically create airplanes in the controlled region. The user must, to some extent, also have control over the degree of randomness.

Low Priority

Controlled region size: The user must have the option to select the dimensions of the controlled region that the system will be overseeing.

Automatic route finder: The application must be able to automatically find an alternate route for an airplane. If no waypoints exist in the controlled region, then alternate routes will not be found.

3.2.2 Non-functional Requirements

High Priority

The backend of the application must be correct: The backend of the application will represent most of the simulation model. As the application is attempting to simulate an automated air traffic control system, which is a safety-critical application, it is necessary for the backend to be formally verified.

Airplanes and waypoints are easy to identify: Users must be able to easily identify airplanes and waypoints in the controlled region. This implies that all airplanes and waypoints must be associated with a unique ID.

Airplane information can be easily deduced: Every airplane's altitude, position, direction, route, status, and direction must be readily and easily available to users. Some of this information must be visually retrievable, in that users must be able to obtain information about airplanes by just looking at the controlled region, for e.g. obtain information about an airplane's path or its status (whether it is conflicting or not).

Medium Priority

Waypoints must be quick and easy to add: All waypoints must maintain a certain minimum distance between each other. Therefore, when adding waypoints to the controlled region, these valid locations must be easy to identify.

No operation must slow down the simulation: The timesteps that make up a simulation must each take approximately the same period of time. This implies that no operation must visually lengthen a timestep from the perspective of the user.

Low Priority

Parallelly interacting with the application: The application must allow users to interact with the simulation while the simulation is running. For instance, when a user is selecting a new path for an airplane, the simulation must continue running. This makes the simulation feel more realistic.

The simulation must be of a professional standard (or close to it): The application must include features that are commonly present in other simulation software, such as pausing the simulation and changing the speed of the simulation. Moreover, information regarding the state of the simulation must also be readily available, for instance, the number of collisions that have occurred, and the average amount of time that an airplane has been in a conflicted status.

3.3 The Simulation

Based on the requirements, it seemed most logical to have two modes (see Figure 3.1). The first mode will require users to manually add airplanes into the region. Users will be required to input the airplane's speed, altitude, and route. The user will then have the option to add any number of airplanes as he/she desires, while the simulation is running.

The second mode will require users to select how many airplanes will always be in the controlled region. Users will also be required to select the degree of randomness for the altitudes that airplanes can have when entering the controlled region. Once these constraints have been provided, the simulation will randomly generate a fixed number of airplanes within the controlled region, each having a random altitude that falls within the

boundaries that were specified by the user. This feature proved to be very useful when evaluating the performance of the collision avoidance feature of the application.

Now that the simulation has been initialized, the user can run it. The simulation cycle for both modes represents one timestep in the simulation environment. The user will have the option to interact freely with the simulation, while the simulation runs; however, changes that are made during a timestep will only be "implemented" in the next timestep.

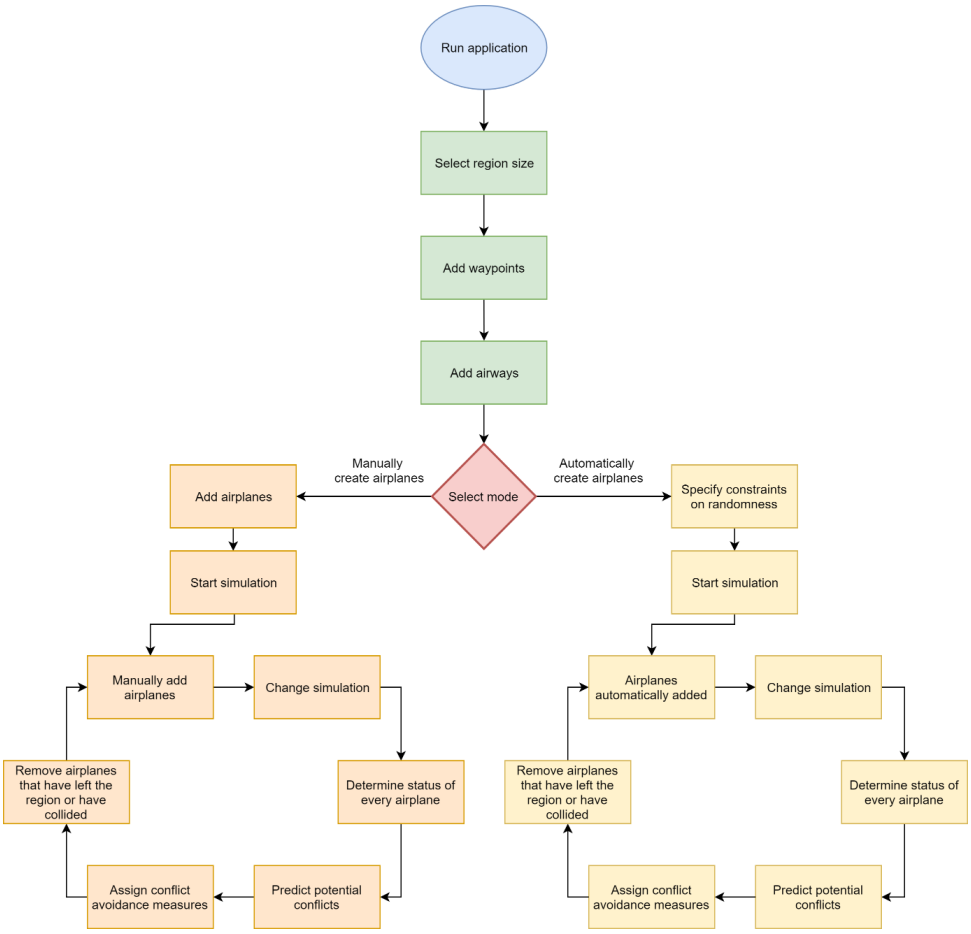


Figure 3.1: A high-level diagrammatic representation of how the simulation will run

3.4 System Architecture

This application consists of two main components (see Figure 3.2). The first component is the simulation model. The second component deals with representing the simulation model in a user-friendly manner and will also allow interactions with the simulation model.

The first component represents the backend of the application. It contains all the features required for the simulation to run. It is also responsible for automating the air traffic control system by attempting to avoid conflicts between every airplanes in the simulation. As was mentioned before, this component must be formally verified. This ensures that both the simulation and the conflict avoidance operations are robust and error-free. The backend will be implemented using a tool called Perfect Developer to achieve complete correctness.

The second component forms the frontend of the application. This component deals with the graphical user interface. It will comprise a visual representation of the simulation model along with everything else required for the user to effectively communicate with the simulation model. It also carries out functions that Perfect Developer is unable to carry out. The frontend has been implemented in Java, using the JavaFX library.

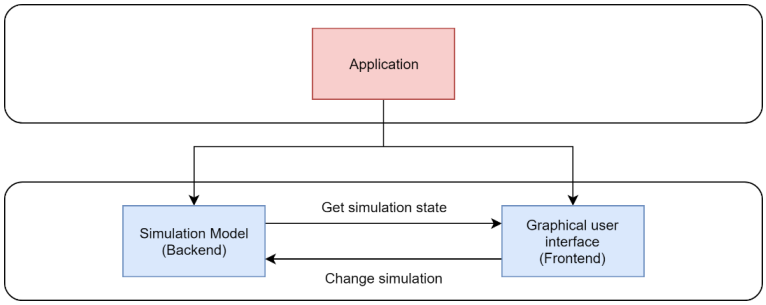


Figure 3.2: The two main components of the application

Now that the high-level structure has been established, we can move onto looking at the structure of the backend (see Figure 3.3).

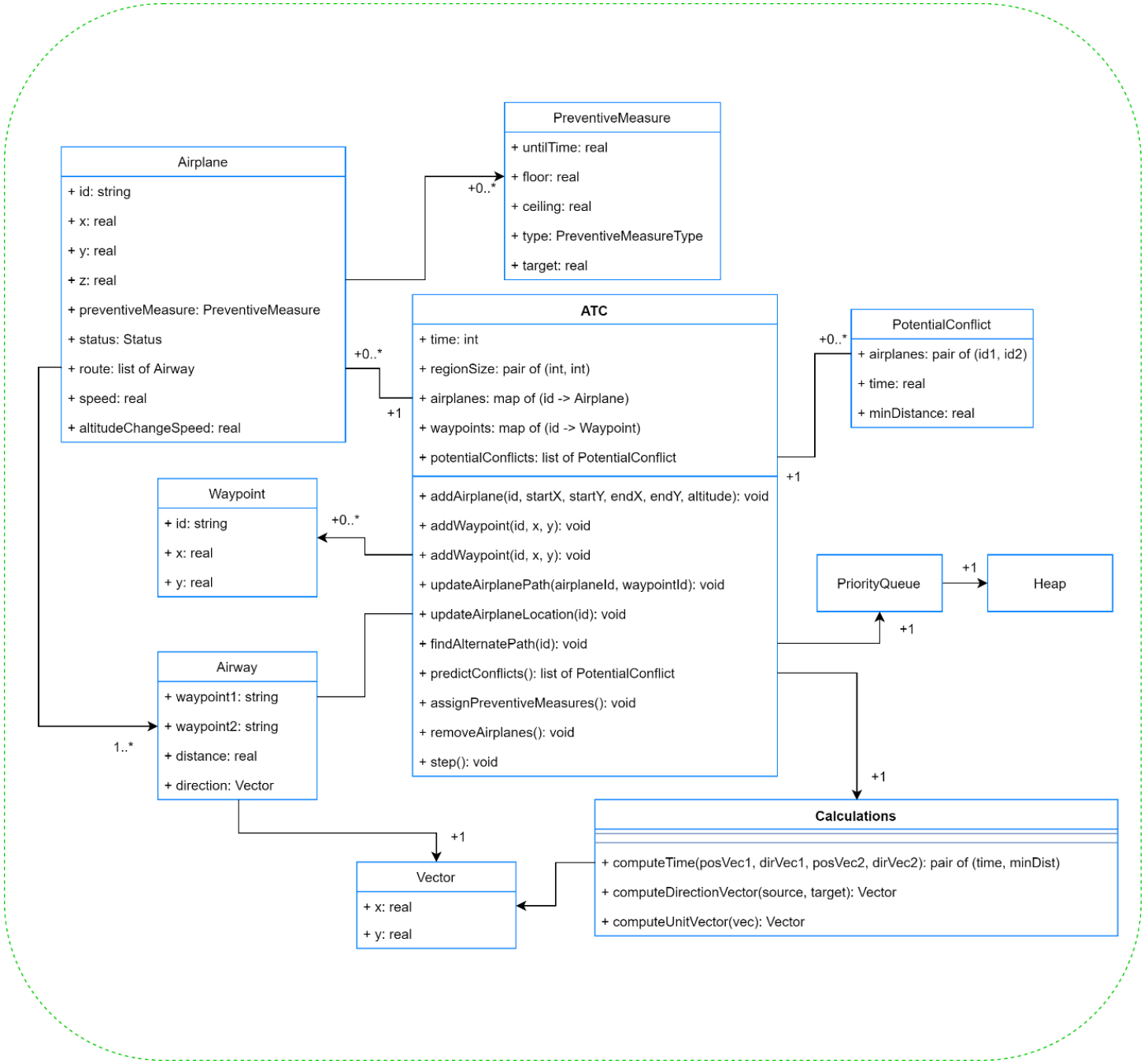


Figure 3.3: The class diagram of the backend; only the most important classes, methods and class variables have been included

ATC

The ATC class forms the foundation of the backend (see Figure 3.3); it carries out all the tasks relating to the air traffic control system. It is also responsible for interacting with the frontend. Therefore, it is necessary for it to provide all the functionalities that are required for the simulation to run.

Understandably, the ATC class is responsible for many things. One of its most important responsibilities is for the class to keep track of all airplanes within the controlled region. This was done by keeping a dictionary/map with the ID of an airplane as the key, and the value being the object representing the airplane. Waypoints in the controlled region are stored in a similar manner.

As the simulation model is that of an automated air traffic control system, the ATC class also detects and keeps track of all potential conflicts between airplanes within the controlled region. Once potential conflicts have been detected, the ATC class moves on the assigning preventive measures to airplanes that could be in danger.

The ATC class also contains a step() method, which represents one simulation timestep. The purpose of this method is to invoke many of the other methods included in the ATC class. This will prove to be very useful when interacting with the frontend. When this method is invoked, the following operations are carried out in the same order as they are presented:

1. Potential conflicts are detected
2. Preventives measures are assigned
3. Locations of all airplanes are updated
4. Airplanes that have left the controlled region or have collided are removed from the simulation

Lastly, the ATC class also contains all accessor and mutator methods required by the front-end to make changes to the simulation and obtain comprehensive information regarding the state of the simulation.

Calculations

The Calculations class forms a vital part of the backend. It is firstly responsible for all vector geometric calculations. It also contains a method

(the computeTime() method) that determines when two airplanes will be closest, and if the two airplanes will be in a conflicted status at that time.

Airplane

An Airplane object represents an average commercial airplane. Although oversimplified, it does contain all the core attributes that an ordinary airplane would logically have. This includes its unique ID, speed, route, and attributes that represent its position in a 3D plane.

Airway

The Airway class represents an airway between a pair of waypoints. A separate class was constructed for this, because the length of the airway and the direction could be computed and stored. This makes it easier to carry out other tasks, like updating the location of an airplane.

PreventiveMeasure

An object of type PreventiveMeasure represents an action (specifically an altitude changing action) carried out by an airplane to avoid potential conflicts. It specifies the type of the preventive measure (whether the airplane must elevate or descend), the target altitude that an airplane should be aiming to reach, the bounds within which the airplane should be flying, and the time until which the airplane must carry out this manoeuvre.

PriorityQueue

The application comes with an option to find alternate routes for airplanes automatically. This has been done using Dijkstra's shortest path algorithm. To implement the algorithm, a priority queue was required. Therefore, a priority queue was implemented from scratch using a heap.

PotentialConflict

This class represents a potential conflict between a pair of airplanes. It contains the IDs of the airplanes that will be involved in the potential conflict. Moreover, it also contains the time at which the airplanes will be closest or

start conflicting. Furthermore, it also keeps track of the distance the two airplanes will have at this time.

3.5 Mathematics underlying the Collision Avoidance Feature

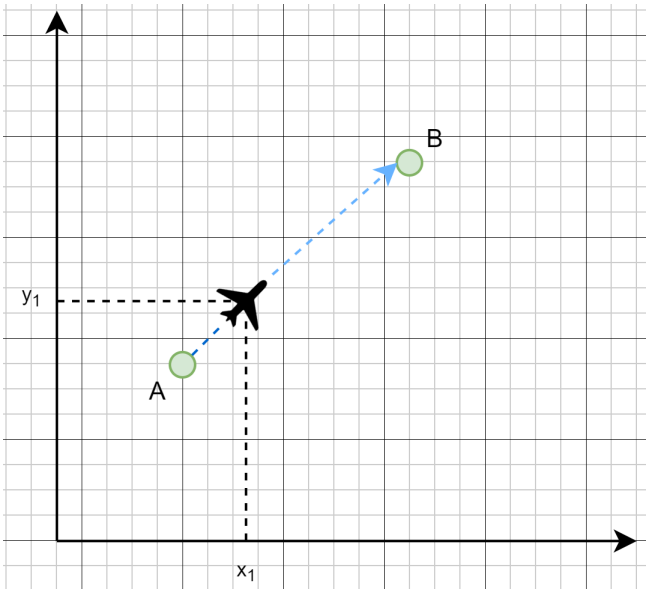


Figure 3.4: An airplane traveling from waypoint A to waypoint B

3.5.1 Modelling the motion of airplanes using vector geometry

The airplane shown in Figure 3.4 is traveling from waypoint A to waypoint B with a speed of exactly 200m/s. We can see from the figure that its current position is:

$$\vec{p} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{3.1}$$

Vectors can be used to represent straight lines. A very useful application of this is to use it to model the position and movement of objects. The

position of an object can be directly related to time using the following equation:

$$\vec{p_t} = \vec{p_c} + \vec{v}t \tag{3.2}$$

Where $\vec{p_t}$ represents the position of the object after t units of time, $\vec{p_c}$ represents the current position of the object, and $\vec{v}$ is the velocity vector of the object. The velocity vector is simply the object's direction vector multiplied by its speed.

We have been given the speed of the airplane; using this we must obtain the velocity vector for the airplane. We know that the airplane is flying from waypoint A to waypoint B. Therefore, the velocity vector can be easily computed using the unit vector of $\vec{AB}$:

$$\vec{v} = 200 \times \hat{\vec{AB}} \tag{3.3}$$

We now have everything we need to model the movement of the airplane using vectors:

$$\vec{r} = \vec{p} + \vec{v}t \tag{3.4}$$

3.5.2 Conflict prediction feature

Based on the separation rules that were mentioned in chapter 2, this application finds two airplanes to be in a conflicted state if both the following conditions are satisfied:

1. Their horizontal separation is less than 10km
2. Their vertical separation is less than 300m

For this project, **potential** conflicts are those that take place when two airplanes will have a horizontal separation of less than 10km, without considering the altitudes of the airplanes. The main reason for doing this was that the application was built to avoid potential conflicts without the user's interference, by instructing airplanes to fly at a different altitude instead of altering their paths. This implies that if the user does nothing, then situations, where the airplane will be horizontally less than 10km away from other airplanes, will be inevitable. However, the altitudes of every airplane will be constantly changing, therefore, it is logical not to make any assumptions regarding the probability of two airplanes conflicting in the future, based on

their present altitudes. Two airplanes currently having a vertical separation of more than 300m apart, must still maintain that separation when they are horizontally less than 10km apart. Moreover, two airplanes that are currently less than 300m apart, must alter the altitude at which they fly so that they do not start conflicting in the future.

Assume we have two airplanes A and B flying towards each other as shown in Figure 3.5:

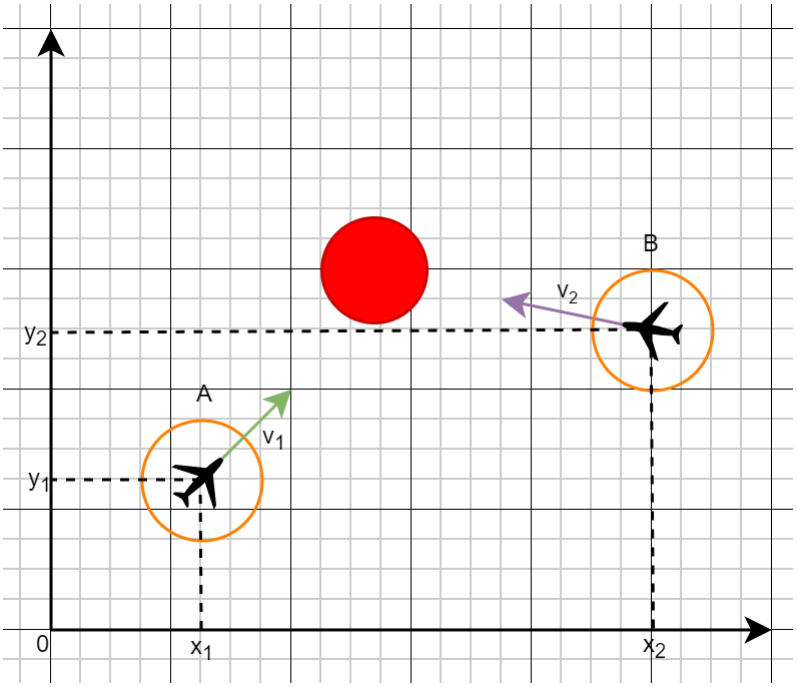


Figure 3.5: Airplanes A and B are flying towards each other. The orange circles around both airplanes have a radius of 5km. The red circle represents the region where they will be closest. Distances are not drawn to scale

The two airplanes are flying towards each other, but not necessarily at each other. The orange circles around them have a radius of 5km and represent their respective "safe-space". When the circles of two airplanes intersect, then the two airplanes must have a vertical separation of at least 300m.

Regardless of whether they crash or not, they will still be conflicting when they reach the red region. Therefore, this situation must be classified as a potential conflict.

To figure out the time at which both airplanes will be closest to each other, we first need to derive an expression that relates the distance between the two airplanes with the time:

$$\begin{aligned} \vec{n}_1 &= \begin{pmatrix} u_1 \\ v_1 \end{pmatrix} \\ \vec{n}_2 &= \begin{pmatrix} u_2 \\ v_2 \end{pmatrix} \end{aligned} \tag{3.5}$$

$$\begin{aligned} \vec{r}_1 &= \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} u_1 \\ v_1 \end{pmatrix} t \\ \vec{r}_2 &= \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} u_2 \\ v_2 \end{pmatrix} t \end{aligned} \tag{3.6}$$

The above equations mathematically represent the motion of A and B.

Now the Euclidean distance between the two airplanes after t units of time can be found using the following expression:

$$f(t) = \sqrt{[(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2} \tag{3.7}$$

t is greater than 0 since we are only concerned with events that will take place in the future.

Now, if at any point in the future t_f, $f(t_f) = 2 \times 5km$, then the two circles will be intersecting. However, we are interested in the time t_c when the two airplanes are closest.

$$\frac{df}{dt} = 0 \tag{3.8}$$

When we find the derivative of f, then equate it to 0 and solve for t, we can get the time at which the two airplanes are closest (the gradient at peaks is 0). Moreover, since the paths are straight lines, it is guaranteed that there will only be one global minimum. This is because straight lines only intersect once, and the greatest distance between two points on the lines is infinite. Therefore, there will only be one moment in time when both airplanes are closest. The only situation where this does not apply is when the velocity vectors are the same. The process of differentiating f is long; therefore, details of how this was done has been provided in appendix A.

The above methodology works for most situations; however, how do we deal with situations where two airplanes will be conflicting within the controlled region, but are closest once both have left the region?

To deal with situations where two airplanes start conflicting within the controlled region but are closest to each other after leaving the controlled region, we compute the time at which the airplanes will start conflicting.

$$\begin{aligned} \sqrt{[(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2} &= 10 \\ [(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2 &= 10^2 \end{aligned} \tag{3.9}$$

When the distance between the two airplanes is exactly 10km, the two airplanes have just started conflicting. Rearranging the above equation and then grouping the expressions, we get a quadratic equation of the form $at^2 + bt + c = 0$:

It makes sense that it is a quadratic equation. If the airplanes do not have the same velocity vector, there will be two distinct moments when the airplanes are exactly 10km away from one another. The most trivial example of this is shown in Figure 3.6, where two airplanes A and B are flying exactly towards each other from opposite directions. Among the two solutions, we obviously require the smaller one since we need to predict the time at which the airplane could start conflicting.

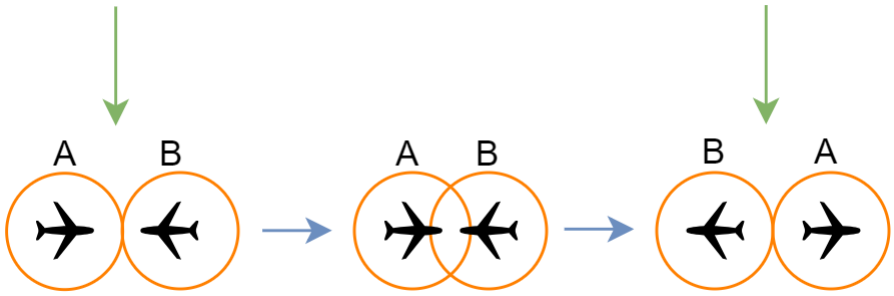


Figure 3.6: Airplanes A and B flying in opposite directions

To solve the quadratic equation, we can use the quadratic formula, and subsequently, select the smaller of the two solutions:

$$t_{1,2} = \frac{-b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a} \tag{3.10}$$

Interestingly, $\frac{-b}{2a}$ is the same as t when solving $\frac{df}{dt} = 0$ for t (a proof of why this is the case has been provided in appendix B). This fact proves to be extremely useful since the expressions do get quite large; therefore, there will be small improvements in computation speed, and the codebase will look more concise and elegant as well.

Chapter 4

Implementation

This chapter provides an overview of how the application was implemented. It first includes details about the main tool used for the backend, the Perfect Developer tool. It also justifies why this tool was used instead of others. Following this, the chapter moves on to discuss how some of the more interesting features in the backend were implemented without going too much into low-level detail. Finally, an overview of what the frontend is capable of, and some of its main features, have been provided towards the end of the chapter.

4.1 Overview of The Perfect Developer Tool

The Perfect Developer tool played a very significant role in the development of this project. Therefore, it stands to reason that an overview of its most important features should be covered in this report. This section does just that, along with also discussing the problems it solves, and the benefits of using this tool instead of other tools currently available. A brief overview of the most important characteristics of the Perfect language (pun unintended) will also be included in this section.

4.1.1 Specification Languages vs Implementation Languages

Specification languages work at a higher level of abstraction than implementation languages [34]. They are used to specify the behaviour of a

program without specifying how this behaviour can be implemented [15]. Specification languages with well-defined syntaxes allow programmers to formally define the behaviour of a program. This formal description of the behaviour can then be used to create verification conditions to prove the correctness of the system [15]. Presently, numerous specification languages exist, for instance, Z Notation and LOTOS [34, 15].

Implementation languages, on the other hand, are used to define how a specific behaviour can be achieved [15]. At present, numerous implementation languages exist and have evolved through multiple programming paradigm. For instance, object-oriented programming is one of the most popular programming paradigms.

4.1.2 Verifying Correctness

Automated testing has been the norm when attempting to make sure that programs behave as they are expected to. Here, specific inputs are strategically selected, and the program is made to run on those inputs. If the program outputs the expected results, then the program is assumed to be correct.

For safety-critical software, assuming the correctness of a program based on just a handful of test cases is not a viable solution, especially when the domain of the input is very large (potentially infinite) [37]. Therefore, specification languages are used to mathematically prove the correctness of the program. This effectively allows us to confirm that the program will run correctly on all inputs.

4.1.3 Introduction to Escher Technologies Perfect Developer

The Perfect Developer (PD) tool consists of two main components. The first is the tool itself, and the second is PD's own language, Perfect. What sets Perfect apart from all other specification languages is that it is also an implementation language. Specifically, it is an object-oriented programming language. Therefore, we have both types of languages succinctly packaged into one language.

The PD tool deals with verifying the code written in Perfect. It is also responsible for generating equivalent Java code so that it can be integrated with codebases written in Java (very useful!) [37].

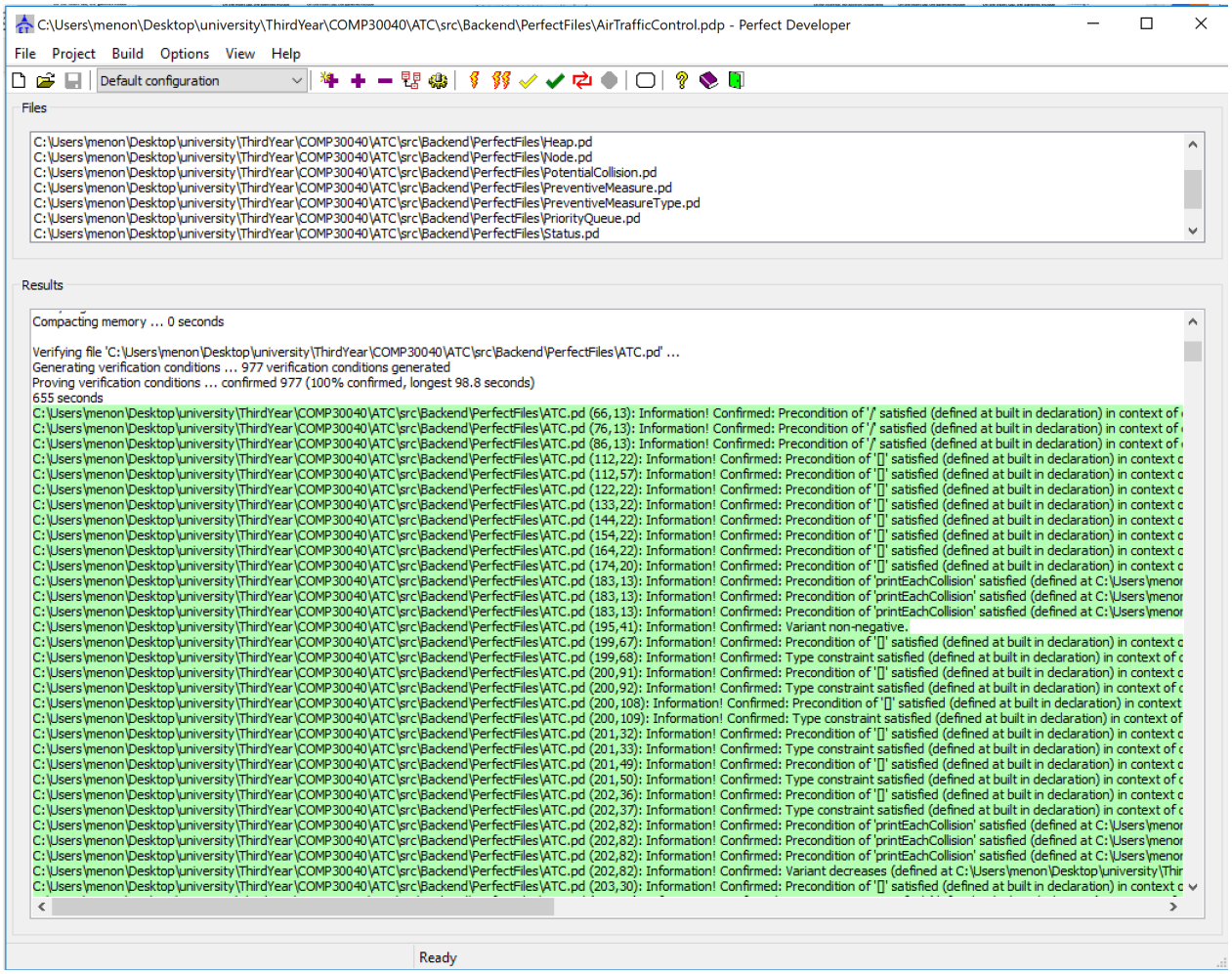


Figure 4.1: The Perfect Developer tool

4.1.4 The Perfect language

Although Perfect does have similarities with other object-oriented programming languages, it still has a very distinctive syntax. Logically, it also contains many other features without which it cannot be classified as a specification language. This section looks at the most noteworthy features that the language has to offer. For a more comprehensive understanding of the language, please refer to the documentation for PD

Preconditions

Perfect allows users to include preconditions to methods. These are conditions that must be true when the method has been called [28].

Postconditions

Mostly used as the body of methods called schemas. They are used to describe the final state of objects that have been modified [28].

Class Invariants

The structure of classes in Perfect is very similar to other languages like Java. However, Perfect also permits users to include class invariants [28]. These are expressions that must always be true for any instance of the class that contains these invariants [28].

They are very useful for the verifier, as after an invariant has been proven to be true, the verifier is able to prove (or disprove) other conditions faster.

Methods

Perfect comes with three types of methods: functions, schemas, and constructors. Constructors play a similar role in Perfect as they do in other languages. Functions and schemas, on the other hand, are quite unique to Perfect.

Functions: Functions, in Perfect, are methods that can change nothing [28]. This means that it cannot change the arguments that are supplied to it, it cannot change class variables, and it cannot even change variables that are declared within its body. The only thing it can do is return a result [28]. Although initially quite cumbersome to use, they become more useful later as the codebase gets larger. This is because it can be guaranteed that functions have not changed anything.

Schemas: These are methods that, unlike functions, can change class variables, arguments, and variables that have been declared within the body of the schema. However, what they cannot do is return results. The body of a schema is a postcondition [28].

Recursion in Perfect

Recursive methods in Perfect must include a well-defined variant. This variant must decrease every time the method is recursively invoked [36]. Furthermore, the recursive method must also contain a base case [36].

Loops in Perfect

One major inconvenience with Perfect is the absence of normal loops. Perfect does include expressions like **for..yield**; **forall**; and **exists**, to do operations with collections. However, every branch of these expressions is executed parallelly [28]; therefore, they cannot be used as loops where every iteration must be executed one after the other, for e.g. appending to a list inside a loop. Perfect also includes a loop statement; these statements, however, are exclusively used for specification, and not for implementation [36]. Therefore, situations where loops would have been the logical choice were implemented using recursion.

4.2 The Backend

4.2.1 Predicting potential conflicts

Potential conflicts for every airplane within the controlled region are determined one after the other. All Airplane objects have a variable named **examined** within them, which is always set to false at the beginning of a timestep; when all potential conflicts for an airplane have been found, this variable is set to true. Therefore, when the algorithm finds that an airplane is conflicting with an airplane that has already been examined, it skips this potential conflict because the system is already aware of it. Furthermore, during the next timestep, since all Airplane objects will have their **examined** instance variable set to false again, the function also makes sure that new potential conflicts are always between a unique pair of airplanes. Lastly, when potential conflicts are found, the algorithm also ensures that the potential conflict happens before both airplanes reach the waypoint that they are headed towards, after that their directions will most probably change, so the conflict might not occur at all.

```
schema predictPotentialConflicts(A)
    precondition
        A in controlled region,
        A's route not empty,
        A not examined
    postcondition
    (
        for all airplanes B in controlled region
        (
            if B not examined and A != B
            (
                if forall p in potentialConflicts :- not(p.involves(id,
                    inputId))
                (
                    find time t at which they are closest
                    set var minDist to be the distance between A and B at
                        time t
                    if minDist <= 10km
                    (
                        if t > time to next waypoints for both airplanes
                        (
                            set t to be the time at which dist(A, B) = 10km
                            update minDist variable
                        )
                        pC = new PotentialConflict([A, B], t, minDist)
                        potentialConflicts.append(pC)
                    )
                )
            )
        )
        set A to be examined
     );
```

The pseudocode above shows how potential conflicts are found for an airplane. The pseudocode inherits some features of Perfect. For instance, since this method appends to the list of potential conflicts, a class variable, it must be a schema. Furthermore, notice the preconditions that must be satisfied when the method is called.

An important thing to note is that the above method provides a high-level description of how the code was actually implemented. As was mentioned earlier, Perfect does not include normal loops in its syntax, where the loop iterations are executed sequentially. Therefore, appending to the potentialConflicts list shown in the pseudocode will not be allowed because the list would be accessed parallelly, which Perfect does not permit).

Potential conflicts were found by implementing the method shown in the previous chapter. The complete implementation is a function in the Calculations class called computeTime(). This method takes the position vectors

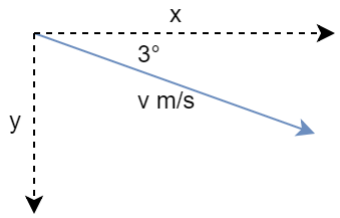
and velocity vectors of two airplanes as input. The function returns the time at which they will be closest(or the time at which they start conflicting), and the distance between them at this time. If two airplanes happened to be already less than 10km away from one another, then the time at which the potential conflict will occur is set to the current simulation time. Doing this is very important because it will ensure that preventive measures that are assigned to these airplanes will ensure that they maintain their vertical separation while they are horizontally close to each other.

Now that we have a complete list of potential conflicts, we can move on to sorting the list by time (in ascending order). As we had mentioned in the previous chapter, the PotentialConflict class also keeps track of the distance that the pair of airplanes will have when they are closest. Therefore, for PotentialConflict objects that have the same value assigned to their time instance variable, the sorting is done on the minDistance instance variable in the object.

Sorting in Perfect is done using the predefined opermndec method, which sorts a list in non-decreasing order, based on ordering specified by an operator that provides similar functionalities to the Comparator interface in Java. Sorting the potential conflicts like this will ensure that earlier and more dangerous potential conflicts are dealt with first.

4.2.2 Assigning preventive measures

In chapter two, we had mentioned that average commercial airplanes usually descended at an angle of three degrees. This was to ensure that passengers within the airplane were disturbed as little as possible. We require the actual speed at which an airplane elevates or descends so that its altitude can be altered at the correct rate. This can be done using basic trigonometry:



If the airplane has a speed v m/s, we can easily find the y component of

the velocity using the sine function:

$$\begin{aligned} sin(3^\circ) &= \frac{y}{vm/s} \\ y &= 0.05223 \times vm/s \end{aligned} \tag{4.1}$$

Clearly, the vertical component is very small compared to the horizontal component. Therefore, in this project, it was assumed that the horizontal component is equal to the speed of the airplane, even when it is elevating or descending. Moreover, for this application, it has been assumed that all airplanes fly with a constant speed of 250m/s (900km/hr), which is the average cruising speed of a commercial airplane.

Once a sorted list of potential conflicts has been made, each of them can be dealt with one after the other. When assigning preventive measures, we would like to be able to deal with as many potential conflicts as we can (as we have seen, commercial airplanes elevate and descend very slowly). However, while dealing with later potential conflicts, we do not want to jeopardize the safety of airplanes by making them start conflicting for earlier potential conflicts. For instance, assume that there are two potential conflicts ((A, B), (A, C)) involving airplanes A, B, and C, in ascending order of the time that the conflicts will occur. Intuitively, preventive measures should be assigned to A and B first, and then the rest. Once A and B have carried out the preventive measures that were assigned to them, the ATC system must move on to assigning preventive measures to A that will not disagree with the preventive measures assigned to it for avoiding B.

Figure 4.2 is a screenshot of the application that was built for this project. In the screenshot, there are three airplanes in the controlled region, initially with the same altitude of 12000m. They have been made to potentially conflict around the centre of the controlled region (the application was able to detect all of three potential conflicts). Notice that airplane A is moving above airplane C to create enough vertical separation, and airplane B is descending. If both airplanes were to move above airplane C, then one of them would have to move even higher up. This could lead to a crash if there is not enough time for the second manoeuvre. Moreover, it forces one of the airplanes to alter their altitudes twice. As we had mentioned earlier, commercial airplanes usually avoid situations where they would have to change the altitude at which they are flying.

One good solution to the previously mentioned requirements is to make all airplanes within the controlled region each have a “ceiling” and a “floor”.

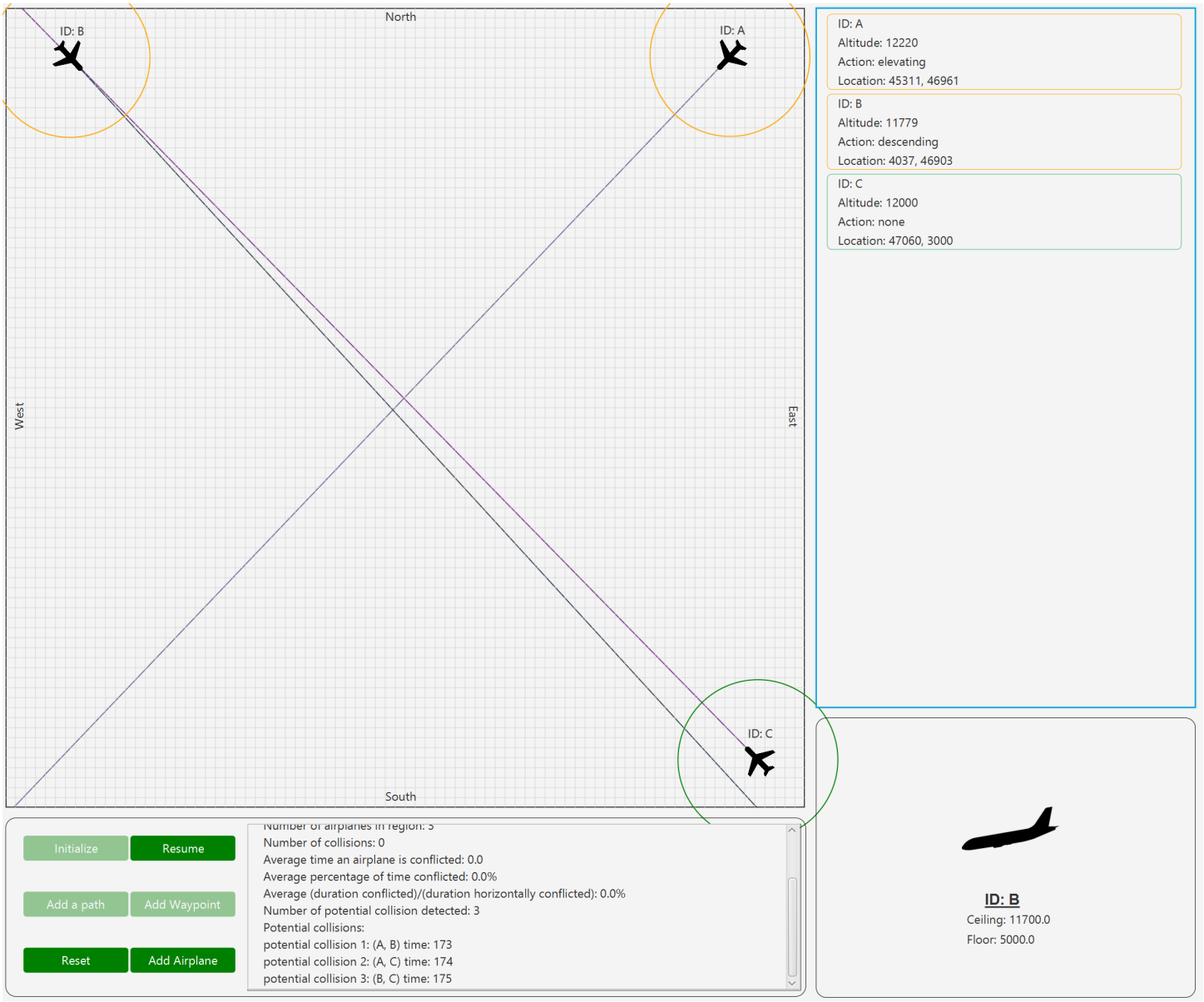


Figure 4.2: Three airplanes attempting to avoid potential conflicts near the centre of the controlled region

Airplanes must always be aiming to fly between their bounds. The basic idea is that for any airplane the ceiling can only be lowered, and the floor can

only be raised. Therefore, for all future potential conflicts that an airplane deals with, it gets progressively more restricted in terms of the altitudes it can fly at. However, this method ensures that earlier potential conflicts get higher priority.

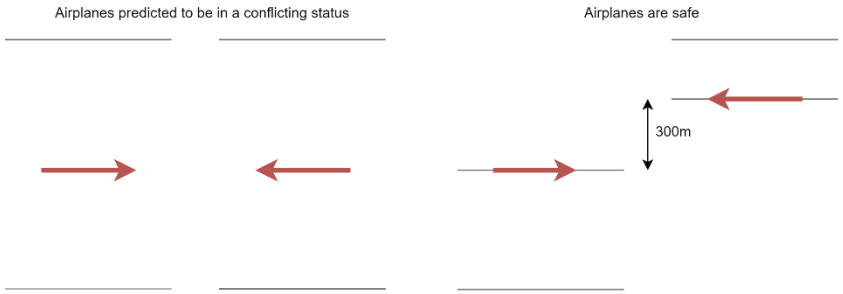


Figure 4.3: A simple depiction of how ceilings and floors are changed

Figure 4.3 shows a simple situation where two airplanes (red arrows) will be potentially conflicting after a certain period of time (both airplanes currently have the same values for their respective ceilings and floors). The way the algorithm works it that it updates the ceilings and the floors of both airplanes. The airplane on the left will now have a ceiling that will prevent it from moving up, and the airplane on the right (once it has reached the desired altitude) will not be able to descend below its new floor. When each airplane is dealing with later potential conflicts, they will be forced to move from the altitude at which they are flying to one that is within their respective altitude bounds. Once a potential conflict has passed, these bounds are reset to 15000m and 5000m for the ceiling and floor, respectively(doing this is okay because conflicts that could still occur will be detected in the next time step, and preventive measures can be assigned for the airplanes involved without the restrictions on the altitude bounds due to potential conflicts in the past). Note that, if there is enough time and enough separation can be achieved, then just one of the airplanes needs to elevate to create enough vertical separation.

For every potential conflict, the algorithm works by first checking if both airplanes are not elevating or descending. Thereafter, it finds out which airplane is more restricted by calculating the distance between the ceilings and the floors of both airplanes. It then takes the more restricted airplane, and it compares the distance between its current altitude with the altitude of its ceiling and its floor. If there is more space above, it assumes that it

will be safer for **both** airplanes to elevate than descend. On the other hand, if there is more space below, then the system will assume that it is safer for preventive measures to involve descent.

To better visualize this, we can use the situation shown in Figure 4.2 again. As we can see from the image, three potential conflicts have been found: (A, B), (A, C), and (B, C). The system deals with each of these in the same order as they appear in the list of potential conflicts. The explanation provided in the previous paragraph is the reason why airplane A elevates above C and B descends below C. The initial preventive measures assigned made airplane B have a ceiling of 12000m, and airplane A have a floor of 12300m (it is elevating to this altitude in the image). Potential conflict (A, C) gets skipped until airplane A stops changing its altitude. The next potential conflict is between B and C, both of which currently have the same altitude. However, B's bounds are more restricted as it has a ceiling of 12000m, while C has a ceiling of 15000m. So, the space that airplane B has above it is 0m, while the space it has below it is 12000m – 5000m = 7000m. Therefore, the system assumes that it would be unsafe for both B and C to elevate. Therefore, in the situation described above, B descends until it is at a safe distance from C. Once A has reached 12300m, potential conflict (A, C) is handled. However, they already have enough spacing between each other. Moreover, they do not even have to update their bounds to make sure that this separation is maintained.

There are three methods by which preventive measures are assigned:

1. The higher airplane elevates, and the lower airplane maintains its original altitude

2. The lower airplane elevates, and the higher airplane maintains its altitude. In this situation, the lower airplane will have to close the gap between it and the higher airplane, and it will have to elevate a further 300m

3. The higher airplane elevates, and the lower airplane descends at the same time

Most of the time using method 1 is enough. However, if enough vertical separation cannot be achieved (because of the restrictions imposed by ceilings and floors), then the system goes on to considering method 2. If that is not a viable solution either, then it makes both airplanes fly in opposite

directions at the same time, until enough vertical separation exists between the airplanes. Moreover, if there is insufficient time for method 1 to achieve enough separation, then method 3 is used. A similar set of methods is used for when there is more space below the more restricted airplane.

4.2.3 Updating an airplane's path

Users have the option to manually alter the route of an airplane. The new route is simply just a different sequence of waypoints. This new sequence of waypoints must be sent in the correct order to the backend from the frontend.

Methods in java code that have been generated from Perfect code can only accept primitive types like int; double; and boolean, without any alterations being required. User-defined classes and arrays, on the other hand, are very difficult to convert to types that the PD generated java code can work with [23]. Thankfully, PD does provide a function to deal with strings, by providing a class called _eSystem [23]. This class comes with a method called l_String(), which takes a string as input and converts it to Perfect's "sequence of char" representation of a string [23].

The l_String() method was extremely useful for multiple features. One of the features that heavily uses it is the feature that this subsection is describing.

Due to the reasons that were mentioned before, it is not an option to pass in an actual array of waypoint IDs to the backend. Moreover, Perfect is not suitable for string manipulation either. Therefore, I decided to pass in the sequence of waypoints one after the other. Figure 4.4 depicts how the feature works at a very high level of abstraction.

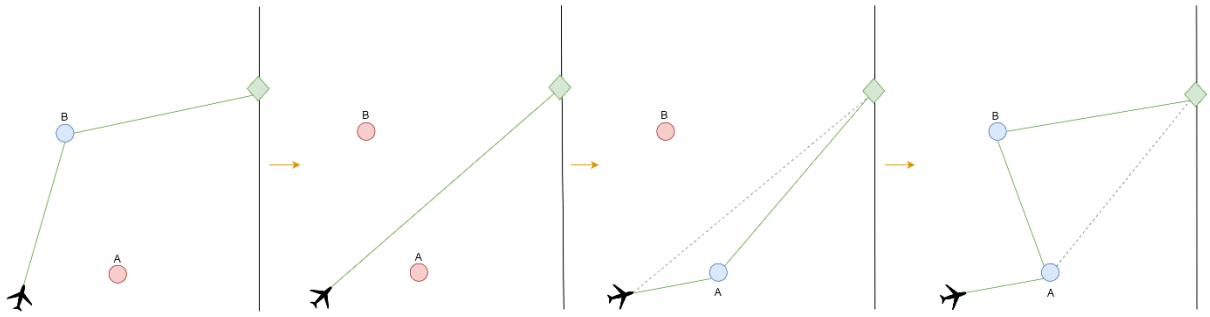


Figure 4.4: Figure showing how an airplane's path is changed

Assume we have an airplane that currently has a path from waypoint B to its exit point in the controlled region (the green diamonds in Figure 4.4). We would like to change the path to A→B→exit point. Initially, all airplanes have a straight path; therefore, their routes are simply comprised of an airway consisting of the airplane's starting coordinates and its ending coordinates (the waypoint class has two constructors: one for waypoints that do have IDs, and the other for waypoints that do not). Moreover, the constructor of the Airway class accepts two Waypoint objects. In the body of the constructor itself, its direction and length are computed. Therefore, we can treat the starting and ending coordinates as waypoints that are no different from two actual waypoints when altering the route of an airplane.

Once the user has set a new route for an airplane, the new route is stored in the frontend. Following this, the airplane that must change its route is "reset" to a straight path consisting of a single airway that has the airplane's current location as its starting coordinates, and the airplane's exit point as the ending coordinates. Then, each waypoint in the new route is passed to the backend in the same order that the user had set. Every time a new waypoint is passed to the backend, the last airway in the airplane's route is split into two sections (see Figure 4.4). For instance, if the final airway in the route of an airplane is one that goes from arbitrary waypoints X and Z, and we would like it to go from X→Y→Z, then we remove the (X, Z) airway, and replace it with airways (X, Y) and (Y, Z). Furthermore, if the first airway of an airplane has been changed, then its velocity vector will be updated since there is a high chance that its direction will be different now as well.

The pseudocode shown below presents how the route of an airplane is updated once its route has already been reset.

```
schema updateRoute(airplane, waypoint)
precondition
    airplane in controlled region,
    waypoint in controlled region,
    airplane route not empty
postcondition
(
    firstSection = Airway(airplane.route.last.starting, waypoint)
    secondSection = Airway(waypoint, airplane.route.last.ending)
    remove last element from airplane's route
    append firstSection and secondSection to airplane's route
    if(airplane.route.size() == 2)
    (
        update the direction of the airplane
    )
);
```

Once an airplane's path has been updated, the list of potential conflicts is emptied. Moreover, preventive measures that are being carried out by all airplanes are stopped, and the ceiling and floors of all airplanes are reset. It is okay to do this since conflicts that can still occur can just be found in the next timestep.

4.2.4 Finding alternate paths

In air traffic, waypoints act as nodes and airways can be edges. Therefore, Dykstra's shortest path algorithm can be used to find an alternate route for airplanes. Alternate routes will be not found by the application unless the user instructs the application to find a new route for an airplane. Therefore, this feature cannot be a way to automatically avoid potential conflicts.

Dijkstra's algorithm uses a priority queue. As part of the project, a priority queue using a heap was built from scratch in Perfect because no implementation of a priority queue existed in Perfect. As we will see in the Evaluation chapter, the implementation of the priority queue was absolutely correct. Unlike usual path-finding problems, there are a few additional factors that must be considered when implementing a path-finding algorithm for air traffic. Firstly, because the user defines the airways between the waypoints, there may be waypoints that are completely "isolated". However, as this is air traffic, there is nothing stopping an airplane from flying towards these waypoints. It does not need a road to get there. Therefore, the application must implement this feature so that these waypoints are used. Secondly, during any timestep, it is seldom the case that coordinates of an airplane will exactly match the coordinates of a waypoint. Therefore, the application must be capable of finding a new route for an airplane regardless of its current 2D coordinates.

The first factor was accounted for by temporarily making all waypoints in the controlled region neighbours of the isolated waypoints, i.e. there is a direct edge from these waypoints to all other waypoints in the controlled region. Instead of having an instance variable holding a list of airways in the controlled region, each Waypoint object contains a set of IDs of the waypoints that it is adjacent to.

The second factor is accounted for by temporarily treating the current location of an airplane(source) and its exit point(target) as waypoints. Once this has been done, three of the closest "real" waypoints to the source are found, and these waypoints are then connected by temporary airways to the

source. The same procedure is carried out for the target waypoint.

We are now able to implement Dijkstra's algorithm in Perfect. Implementing the algorithm was challenging. Due to the absence of loops, the algorithm ended up becoming much longer when implemented in Perfect. The implementation of the algorithm comprised three recursive methods that handled each of the following tasks:

1. Initializing the priority queue with the nodes representing the waypoints.

2. Recursing through the priority queue, and extracting the minimum node from it during each recursive step

3. Recursing through the neighbours of the waypoint associated with the minimum node, and updating the distances for these neighbours if required

To better understand how the above methods were implemented, the pseudocode for all three methods have been included below. Notice how each of the methods have an explicit variant that decreases every time the recursive function is called. Once a new path has been found, it is sent to the frontend as a string.

```
schema initializePriorityQueue(waypointIdList, index)
    precondition
        waypointIdList[index] in waypoints.keys(),
        index >= 0,
        index < waypointIdList.size(),
    decrease (waypointIdList.size() - 1) - index
    postcondition
    (
        if(waypointIdList[index] == "source")
        (
            priorityQueue.insert(waypointIdList[index], 0.0)
        )
        else
        (
            priorityQueue.insert(waypointIdList[index], 200000.0)
        )

        if((index + 1) < waypointIdList.size())
            initializePriorityQueue(waypointIdList, index + 1)
    );


schema recurseThroughPriorityQueue(size)
    precondition
        size = priorityQueue.size(),
```

```
        size >= 0
    decrease size
    postcondition
    (
        pop min node u

        if waypoint u.id has neighbours
            recurseThroughNeighbours(u.id, 0, u.distance)

        if((size - 1) > 0 & priorityQueue.size() != 0)
            recurseThroughPriorityQueue(size - 1)
    );

schema recurseThroughNeighbours(waypointId, index, distance)
    precondition
        waypointId in waypoints.keys(),
        index >= 0,
        index < waypoints[waypointId].neighbours.size(),
        waypoints[waypointId].neighbours[index] in waypoints.keys(),
        distance >= 0.0
    decrease (waypoints[waypointId].neighbours.size() - 1) - index
    postcondition
    (
        neighbour = waypoints[waypointId].neighbours[index]
        alt = distance + Airway(waypointId, neighbour).distance
        if(alt < dist[neighbour])
        (
            newRoute[neighbour] = waypointId
            priorityQueue.updateNode(neighbour, alt)
        )

        if((index + 1) < waypoints[waypointId].neighbours.size())
            recurseThroughNeighbours(waypointId, index + 1, distance)
    );
```

4.3 Frontend

4.3.1 Introduction to JavaFX

The frontend was implemented using Java. The library used for the frontend was JavaFX. JavaFX was used because it is currently the newest graphics library that is available. Moreover, it is constantly being updated to include new functionalities. Compared to Java Swing, it is more adept at event handling, and it is also a better library for animations [1, 25]. Furthermore, CSS can also be used to easily style the GUI [24]. Therefore, I believe it provides all features necessary to implement the graphical user interface that I had envisioned during the design stage of the project.

4.3.2 The Graphical User Interface(GUI)

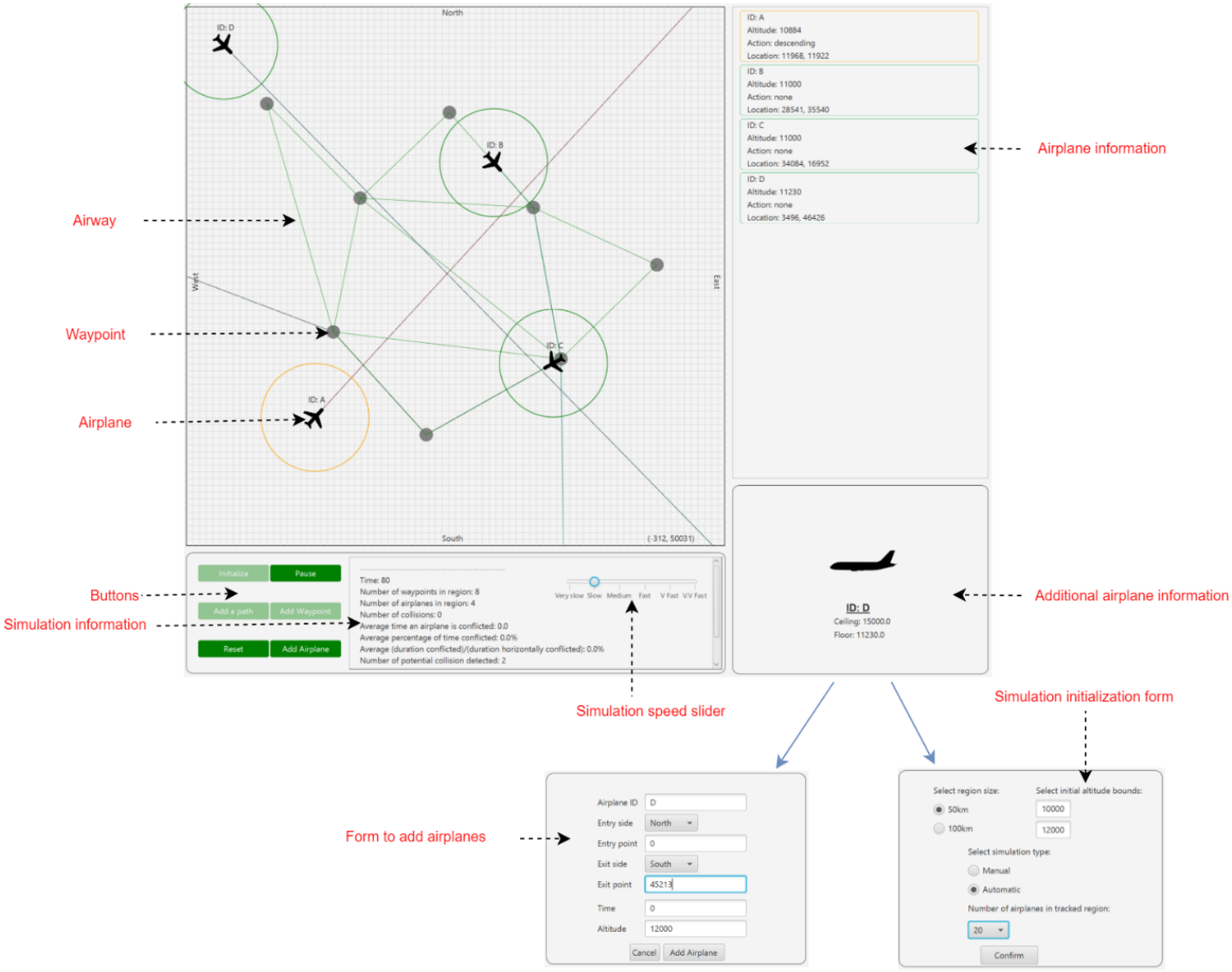


Figure 4.5: Application GUI

Panes are elements that allow one to lay out their UI. They were heavily used in the creation of the GUI for this application. Most panes serve

multiple roles in the GUI. For instance, as can we can see from Figure 4.5, the additional airplane information pane also serves as the pane for displaying forms for initializing the simulation and adding airplanes to the region. Both forms use form validation, so incorrect or invalid information cannot be passed to the backend from the frontend. Moreover, the user can directly interact with the controlled region in the GUI (see Figure 4.5).

The frontend contains an AirplaneFrontEnd class. This class keeps track of all UI elements pertaining to an airplane within the controlled region. Similar to the Airplane object in the backend, the frontend contains a map of IDs as keys and AirplaneFrontEnd objects as values. For a particular airplane in the controlled region, its AirplaneFrontEnd object stores the UI elements that visually represent the actual airplane in the controlled region, the lines that represent the path of the airplane, and its own information pane(the rectangles showing information about the airplane in the airplane information pane). Storing the UI elements like this makes it much easier to retrieve, update and delete them from panes. The AirplaneFrontEnd class also performs one other very important task: in case of airplanes that have been manually added in by the user, but the time for them to enter the controlled region has not come, all initial information about the airplane such as its route and its altitude are stored in its AirplaneFrontEnd object, until it is time for it to enter the controlled region. When this time comes, the addAirplane() method in the PD generated java code is invoked to add the airplane to the controlled region. Therefore, the AirplaneFrontEnd class acts somewhat like a buffer as well.

4.3.3 Animation

JavaFX offers the Timeline class, which is perfect for the animation that this project requires.

```
1 Timeline animation = new Timeline(new KeyFrame(Duration.seconds(2), event ->
2 {
3     updateFrontEnd();
4 }));
5
6 animation.setCycleCount(Timeline.INDEFINITE);
7 animation.play();
```

As we can see from the code above, the class essentially allows us to invoke a method multiple times (possibly infinite), with there being a pause of a

fixed duration each time the method has been called. The updateFrontEnd() method plays a core part in the application as a whole. It firstly updates the whole GUI every timestep, including all UI elements for each of the airplanes. It also calls the step() method in the PD generated Java code, which is responsible for updating the whole simulation model.

The animation runs on a different thread. Therefore, to allow the user to interact with the simulation while the simulation is running, the AtomicReference class is used. Like its name suggests, it only allows data stored in it to be read or written atomically (only one process can write to it and read from it, at a time). Java comes with a separate AtomicBoolean class that is specifically used for Boolean variables. AtomicReferences and AtomicBoolean objects are heavily used in the frontend to make sure that changes are not made to the state of the simulation while executing the code within the step() method of the ATC class. This would lead to concurrent access to resources, which may lead to the application not working as expected.

Chapter 5

Testing and Evaluation

This chapter deals with evaluating the final product that has been built for this project. First, Perfect Developer's verification results for the application are presented. This will reveal the correctness of the application. Then, the chapter will end with an evaluation of the conflict detection, and conflict avoidance features to show the effectiveness of the automated air traffic control system.

5.1 Verification Results

Perfect Developer's automated verifier can be somewhat of a black-box; once a program has been written, including all necessary preconditions, post-conditions,and class invariants, the tool generates proof obligations, and automatically verifies them. The verification process is "hidden", for the most part, from users. However, if an unprovable condition is generated, then the tool does its best to help programmers identify how they can proceed towards making it provable. A far more difficult situation to deal with is when it is unable to prove or disprove a verification condition, within a user-defined time limit. Situations such as these did arise many times during the development process; however, they were eventually dealt with by altering class invariants, preconditions, or how a feature was implemented.

Class invariants and preconditions were thoroughly used where ever possible in the backend. This allowed the verifier to have all it needed to prove the correctness of the backend. Numerous proof obligations were generated for some of the larger classes (see Table 5.1). As we can see from the table,

Class	Number of verification conditions	Time taken/s	%verified
ATC	977	655	100%
Airplane	142	622	100%
Heap	81	2.0	100%
PotentialConflict	19	1.0	100%
Airway	9	0.0	100%
Waypoint	8	0.0	100%
PriorityQueue	5	0.0	100%
Calculations	33	–	–

Table 5.1: Verification results for the backend

it took the verifier more than 10 minutes to verify the ATC class completely; the reason for this cannot be deduced easily since verification is an automated process. Table 1 shows just a subset of the classes in the backend, specifically the largest classes in the backend. As we can see, all the main classes in the backend achieved 100% correctness (based on the specifications that were provided).

The only exception to this is the Calculations class. The verification of the Calculations class ended abruptly with the error message: **Memory allocation failure detected!** Perfect Developer gave no reason for why, and what was causing the error. After increasing the memory limit significantly (to 1200mb), changing the structure of the code, and altering the Calculations class several times, I was unfortunately still unable to fix the error. Therefore, I carried out thorough manual testing to make sure that the class worked as it should. For the specific inputs that I had used, the class seemed to be working as expected.

During the development process, using Perfect proved to be very useful. This was especially true when the development of the frontend had commenced. Any time an error was found, or the application was not functioning as expected, I was much more confident about where the error was originating from.

5.2 Evaluating the Automated Collision Avoidance feature

As mentioned in earlier chapters, the application comes with two distinct modes. The first requires users to manually generate airplanes, and the second creates airplanes randomly so that there are a fixed number of airplanes that are always in the controlled region. The application presents three values that are used to judge how well the application performed at avoiding conflicts:

1. The number of collisions that have occurred in the controlled region
2. The average percentage of time that an airplane is in a conflicted status during its journey within the controlled region
3. The average percentage of time that an airplane is conflicting while it is less than 10km away from another airplane

The second value is most useful when the application is running on mode 1, where the user will constantly have to interact with the simulation by manually adding airplanes to the controlled region. The third value, on the other hand, is very useful, because when the second mode is being used, all airplanes that are randomly created have a straight path, so assuming that the user makes no changes to the simulation, scenarios where airplanes will be less than 10km away from other airplanes are unavoidable. Therefore, only by altering the altitudes of these airplanes can the system avoid conflicts. This makes it perfect for evaluating the automated collision avoidance feature.

Figure 5.1 presents the average percentage of time that an airplane is conflicted while it is less than 10km away from another airplane. The region used for the experiments was a square region with sides of length 50km. For each experiment two constraints were set:

1. The number of airplanes that are always present within the controlled region
2. The bounds between which random altitudes would be assigned to these airplanes when they enter the controlled region

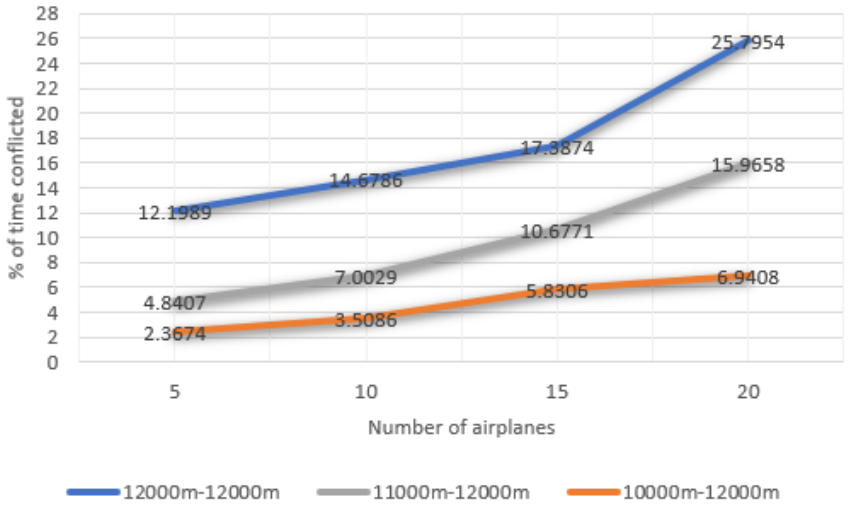


Figure 5.1: Average percentage of time that airplanes are conflicted

During each experiment, the simulation was run for 6000 seconds in the simulation environment. Moreover, each experiment was repeated five times (the results did not vary too much, therefore, more repetitions were not required), and the results were then averaged.

Based on the dimensions of an ordinary commercial airplane, it was decided that, when treating airplanes as single points, a collision would take place when the horizontal distance between a pair of airplanes is less than 80m, and when their vertical separation is less than 20m.

The automated collision avoidance system was not perfect; however, it did perform better than I had expected. The system managed to prevent all collisions during every simulation run. Furthermore, as we can see from the graph, even with 10 airplanes, and an altitude bound of 11000m to 12000m(range = 1000m), the system managed to make sure that on average an airplane spent only 7% of the time being conflicted while having a horizontal separation of less than 10km with another airplane.

The trends in the results were mostly as expected; when the number of airplanes was increased, or the bounds for the random altitudes were made stricter, then the percentage of time that airplanes were conflicting increased. However, I expected the percentage to increase much more when the number of airplanes was increased. The relatively slow increase might have been

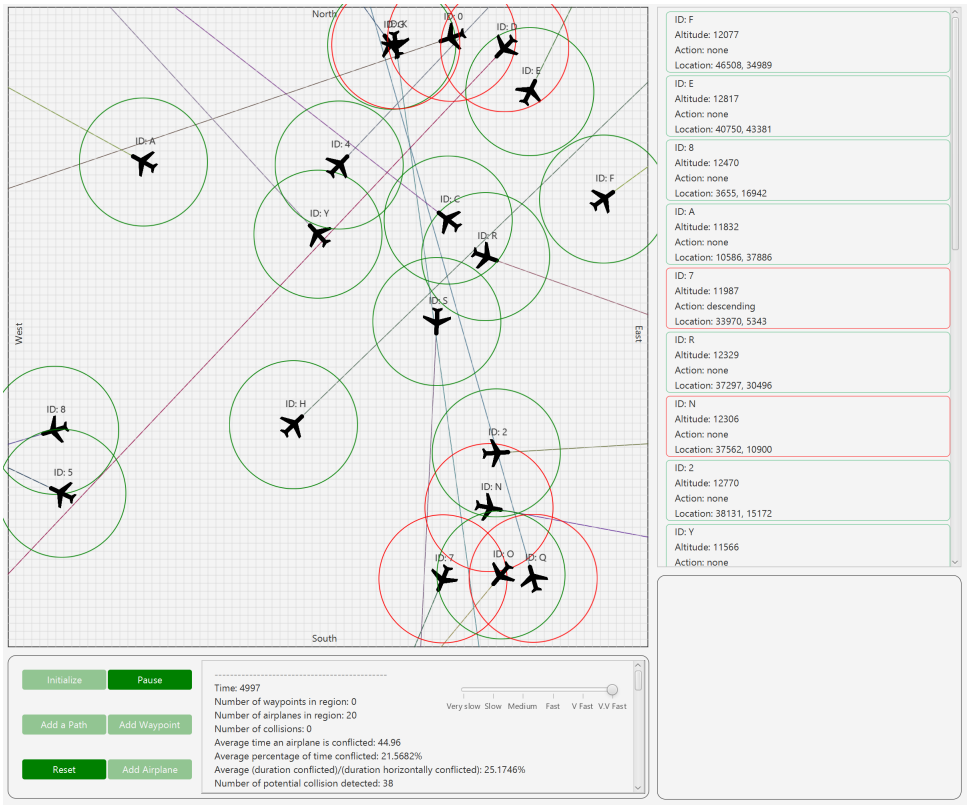


Figure 5.2: imulation with 20 airplanes always being in the controlled region with an altitude of exactly 12000m when entering the controlled region

because more airplanes in the controlled region implied that the amount of time that an airplane spent not being conflicted to other airplanes, but still having a horizontal separation of less than 10km, was more. Lastly, from the graph, we can also see that the application is much more adept at handling situations where the altitudes of airplanes entering the region are more varied (the orange line is increasing more slowly than the other two lines).

Chapter 6

Conclusion

This chapter summarizes the whole development process. It first provides an overview of what was achieved. Following this, the chapter moves on to proposing possible improvements that can be made to the application in the future. After this, the chapter summarized the knowledge and skills I have gained or strengthened after undertaking this project. Finally, the chapter ends with a brief reflection on the undertaking of this project.

6.1 Achievements

The main aim of this project was to build an application that would simulate an en route air traffic control system. To properly define how this application must work, a set of requirements that the application must meet were included. Now that the development process is over, the features that the application includes can be compared with the requirements that it had to satisfy.

The final product, overall, was definitely a success, because it was a well-built and robust implementation of an application that would be able to simulate an automated air traffic control system. Furthermore, the application was able to satisfy almost all the requirements, both functional and non-functional.

A lot of planning and development went into including as many features as possible, given the time that was available. The result was that the initial requirements that were set at the beginning of the development process almost doubled in length. Therefore, the application turned out to be richer

in features and more user-friendly than I had initially planned for it to be.

6.2 Further Improvements

There were a few requirements that the application was only able to partially satisfy. Among these was the requirement that users must have the ability to select the size of the controlled region. The final application unfortunately only came with a square region, with only two sizes to select from 50km and 100km. Furthermore, another requirement that was only partially met was that the backend had to be completely correct. As we saw in the Evaluation chapter, I was unable to formally verify the correctness of the Calculations class. Therefore, I was also not able to mathematically prove that all potential conflicts are found. Moreover, one of the non-functional requirements was to make the application include functionalities that are commonly present in simulation applications. The application that was built did include the option to pause and change the speed of the simulation, but it did not include other features such the ability to roll back/fast forward to a specific timestep or undoing changes that have been made. Therefore, major improvements that can be made are to satisfy the remaining requirements.

The progress that was made with this application, makes it a very good base to make many more improvements. One thing that must be noted is that the application provides an oversimplified representation of air traffic. All airplanes are assumed to have the same speed, even when changing directions. Moreover, weather conditions, land altitude, and wind velocity are important factors that have not at all been considered. Therefore, further improvements to the application could be to incorporate these factors into the simulation.

Furthermore, perhaps AI can be incorporated into the project as well by making the application learn how to assign effective preventive measures to airplanes that could be in danger. This will be a fairly difficult task due to the lack of libraries that Perfect has to offer. But perhaps a simpler improvement would be for the application to assign preventive measures that involve not only changing altitudes, but also changing an airplane's route. Currently, the user must tell the application to find an alternate route for an airplane. The new route is simply the shortest route to the destination and does not look into how safe the route is at all. Therefore, one major improvement that could be made would be to improve the collision avoidance feature of the application.

6.3 Skills and Knowledge Gained

This project offered me a wonderful opportunity to improve my competence in Computer Science. It also allowed me to improve numerous "soft" skills. During the development stage, I had to familiarize myself with two new technologies: Perfect Developer, and JavaFX. Due to the requirements for the application, I had to reach quite a high level of proficiency at using both technologies.

In the case of the Perfect Developer tool, as we have already seen, the tool provides many benefits that are quite unique to it. However, becoming proficient at using the tool was a difficult task. Perfect Developer is a niche product that targets a very specialized group of programmers. Therefore, the resources were very scarce. This made clearing doubts and learning about its more complicated features, time-consuming and a difficult process. In the end, after much research, I was able to understand and effectively use the functionalities that I needed for my project. JavaFX was comparatively easier to learn. The documentation was comprehensive and well-organized. Moreover, there were many resources online to clear doubts, for e.g. StackOverflow. Building the GUI for the application provided me with the opportunity to gain valuable practical experience in using the popular graphics library.

I was also able to become more proficient at planning, which is a skill that I value a lot. Moreover, I was also able to improve my ability to carry out independent research and deal with unforeseen circumstances. I also believe that I have become much better at designing algorithms and solving problems. One of the biggest features in the application, the collision avoidance feature, was designed from scratch, and throughout the development process, the algorithm went through many alterations until it evolved into what it is now.

6.4 Reflection

On the whole, the project was extremely challenging and fascinating at the same time. Although, many unforeseen circumstances had arisen during the development of the application, the final product, more than compensated for it. Thus, this project provided me with a very good opportunity to learn a multitude of new things and strengthen many of my existing skills.

Bibliography

[1] *10 DIFFERENCES BETWEEN JAVAFX AND SWING*. https://www.dummies.com/programming/java/10-differences-between-javafx-and-swing/. Accessed on 25/04/2019.

[2] *10,000 and counting: the most successful jet aircraft of all time*. https://www.telegraph.co.uk/travel/comment/the-most-popular-plane-ever-made/. Accessed on 22/04/2019.

[3] *1976 Zagreb mid-air collision*. https://en.wikipedia.org/wiki/1976_Zagreb_mid-air_collision. Accessed on 18/04/2019.

[4] *24hr strike for French air traffic controllers*. https://www.bbc.com/news/av/world-europe-10711036/24hr-strike-for-french-air-traffic-controllers. Accessed on 18/04/2019.

[5] *A Pilot Explains Waypoints, the Hidden Geography of the Sky*. https://www.cntraveler.com/stories/2015-06-02/a-pilot-explains-waypoints-the-hidden-geography-of-the-sky. Accessed on 21/04/2019.

[6] *Acknowledging the human factor in air traffic control*. https://www.airport-technology.com/features/human-factor-air-traffic-control/. Accessed on 18/04/2019.

[7] *Air traffic control*. https://en.wikipedia.org/wiki/Air_traffic_control. Accessed on 17/04/2019.

[8] *Air Traffic Control: Here's How it Works*. https://aviationoiloutlet.com/blog/air-traffic-control/. Accessed on 20/04/2019.

[9] *Air Traffic Control System Command Center (ATCSCC)*. https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/systemops/nas_ops/atcscc/. Accessed on 20/04/2019.

[10] *Air Traffic Controller Trust in Automation in NextGen*. https://www.sciencedirect.com/science/article/pii/S2351978915005107. Accessed on 28/04/2019.

[11] *Airborne Collision Avoidance System(ACAS) Manual*. https://www.icao.int/Meetings/anconf12/Document%20Archive/9863_cons_en.pdf. Accessed on 21/04/2019.

[12] Airbus. *Global Networks, Global Citizens*. https://www.airbus.com/content/dam/corporate-topics/publications/media-day/GMF-2018-2037.pdf. Accessed on 28/04/2019.

[13] *Airway*. https://www.skybrary.aero/index.php/Airway. Accessed on 21/04/2019.

[14] *Automated Collision Avoidance Systems*. http://www.ieeecss.org/sites/ieeecss.org/files/documents/IoCT-Part2-08AutomatedCollisionAvoidance-LR.pdf. Accessed on 21/04/2019.

[15] Gareth Carter. *AUTOMATING FORMAL SOFTWARE CONSTRUCTION*. http://www.cs.nuim.ie/research/pop/papers/GarethCarterMSc.pdf. Accessed on 24/04/2019.

[16] *Could Computers Replace Human ATC?* https://thepointsguy.com/2016/02/could-computers-replace-human-atc/. Accessed on 18/04/2019.

[17] *Countries With The Highest Number Of Airline Passengers*. https://www.worldatlas.com/articles/countries-with-the-highest-number-of-airline-passengers.html. Accessed on 19/04/2019.

[18] *Cruising speeds of the most common types of commercial airliners (in knots)*. https://www.statista.com/statistics/614178/cruising-speed-of-most-common-airliners/. Accessed on 21/04/2019.

[19] *Dornier 328*. https://www.globalair.com/aircraft-for-sale/Specifications?specid=716. Accessed on 22/04/2019.

[20] *Emirates A380 Specifications*. https://www.emirates.com/english/flying/our_fleet/emirates_a380/emirates_a380_specifications.aspx. Accessed on 22/04/2019.

[21] CRAIG FREUDENRICH. *How air traffic control works*. https://www.caa.co.uk/Consumers/Guide-to-aviation/How-air-traffic-control-works. Accessed on 17/04/2019.

[22] Nick Harding. *42 Years ago today: British Airways flight in mid-air collision*. https://ukaviation.news/42-years-ago-today-british-airways-flight-in-mid-air-collision/. Accessed on 18/04/2019.

[23] *Interfacing a Java graphical front-end to a Perfect Developer application*. https://www.eschertech.com/tutorial/tutorials.php. Accessed on 25/04/2019.

[24] *JavaFX CSS Reference*. https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. Accessed on 25/04/2019.

[25] *JavaFX Documentation*. https://docs.oracle.com/javase/8/javafx/api/toc.htm. Accessed on 25/04/2019.

[26] *Loss of Separation*. https://www.skybrary.aero/index.php/Loss_of_Separation. Accessed on 20/04/2019.

[27] *Next Generation Air Transportation System*. https://en.wikipedia.org/wiki/Next_Generation_Air_Transportation_System. Accessed on 18/04/2019.

[28] *PD Basic Tutorial*. https://www.eschertech.com/tutorial/tutorials.php. Accessed on 24/04/2019.

[29] *Safety Management and Automation in Air Traffic Control: the future*. https://nats.aero/blog/2013/11/safety-management-automation-air-traffic-control-future/. Accessed on 18/04/2019.

[30] *Section 3.En Route Procedures*. https://web.archive.org/web/20120309151642/http://www.faa.gov/air_traffic/publications/atpubs/aim/aim0503.html. Accessed on 21/04/2019.

[31] *Section 5. Radar Separation*. http://tfmlearning.faa.gov/publications/atpubs/ATC/atc0505.html. Accessed on 20/04/2019.

[32] *Separation Standards*. https://www.skybrary.aero/index.php/Separation_Standards#Vertical_Separation. Accessed on 20/04/2019.

[33] *Separation standards*. http://www.airservicesaustralia.com/services/how-air-traffic-control-works/separation-standards/. Accessed on 20/04/2019.

[34] *Specification Language*. https://en.wikipedia.org/wiki/Specification_language. Accessed on 24/04/2019.

[35] *The beginning of Air Traffic Control*. https://airandspace.si.edu/exhibitions/america-by-air/online/innovation/innovation12.cfm. Accessed on 17/04/2019.

[36] *The PD Language Reference Manual*. http://www.eschertech.com/product_documentation/Language%20Reference/LanguageReferenceManual.html. Accessed on 24/04/2019.

[37] *The Process Tutorials*. https://www.eschertech.com/tutorial/processtutorials.php. Accessed on 24/04/2019.

[38] *What is TRACON?* https://aviation.stackexchange.com/questions/25169/what-is-a-tracon. Accessed on 20/04/2019.

[39] *Why does the descent take so long?* https://eu.usatoday.com/story/travel/columnist/cox/2015/01/25/airplane-descent-landing/22220751/. Accessed on 21/04/2019.

Appendix A

Derivative of equation 3.7

$$f(t) = \sqrt{[(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2} \qquad \text{(A.1)}$$

Steps to find $\frac{df}{dt}$: First use chain rule:

$$f = \sqrt{s}, s = [(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2$$
$$\frac{df}{dt} = \frac{d}{ds}(\sqrt{s})\frac{d}{dt}([(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2)$$

Solve $\frac{d}{ds}(\sqrt{s})$:

$$\frac{d}{ds}(\sqrt{s}) = \frac{1}{2}s^{\frac{1}{2}-1} = \frac{1}{2\sqrt{s}}$$

Solve $\frac{d}{dt}([(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + [(y_1 + v_1 t) - (y_2 + v_2 t)]^2)$:

$$= \frac{d}{dt}[(x_1 + u_1 t) - (x_2 + u_2 t)]^2 + \frac{d}{dt}[(y_1 + v_1 t) - (y_2 + v_2 t)]^2$$

Solve $\frac{d}{dt}[(x_1 + u_1 t) - (x_2 + u_2 t)]^2$ using chain rule:

$$= \frac{d}{dr}(r^2)\frac{d}{dt}[(x_1 + u_1 t) - (x_2 + u_2 t)]$$
$$= 2(x_1 + u_1 t - x_2 - u_2 t)(u_1 - u_2)$$

Identical steps can be carried out for $\frac{d}{dt}[(y_1 + v_1 t) - (y_2 + v_2 t)]^2$:

$$= 2(y_1 + v_1 t - y_2 - v_2 t)(v_1 - v_2)$$

Combine terms:

$$= 2(x_1 + u_1t - x_2 - u_2t)(u_1 - u_2) + 2(y_1 + v_1t - y_2 - v_2t)(v_1 - v_2)$$

Therefore,

$$\frac{df}{dt} = \frac{1}{2\sqrt{s}}(2(x_1 + u_1t - x_2 - u_2t)(u_1 - u_2) + 2(y_1 + v_1t - y_2 - v_2t)(v_1 - v_2))$$

$$= \frac{m}{2\sqrt{[(x_1 + u_1t) - (x_2 + u_2t)]^2 + [(y_1 + v_1t) - (y_2 + v_2t)]^2}} \tag{A.2}$$

where m $= 2(x_1 + u_1t - x_2 - u_2t)(u_1 - u_2) + 2(y_1 + v_1t - y_2 - v_2t)(v_1 - v_2)$

Now, after simplifying equation A.2, we have:

$$\frac{df}{dt} = \frac{(u_1t - u_2t + x_1 - x_2)(u_1 - u_2) + (v_1t - v_2t + y_1 - y_2)(v_1 - v_2)}{\sqrt{(u_1t - u_2t + x_1 - x_2)^2 + (v_1t - v_2t + y_1 - y_2)^2}}$$

Appendix B

Proving that $\frac{-b}{2a}$ is the vertex of a parabola

For any parabola, the vertical line of symmetry passes through the vertex of the parabola. This implies that shifting the curve up or down will not make a difference to its vertex. Therefore, if we assumed to have a quadratic equation $ax^2 + bx + c$, then we can first shift the curve by $-c$ to obtain $ax^2 + bx$. Now factoring x, we get $x(ax + b)$. This means that the solutions of the quadratic equation will be $x = 0$ and $x = \frac{-b}{a}$. Determining the x-intercept of the vertical line of symmetry is then simply the point that is half way between $x = 0$ and $\frac{-b}{a}$, which is $\frac{-b}{2a}$