Aircraft Controller Collision Avoidance Synchronous System

Mary Stirling Brown
CS 6376-01 Hybrid/Embedded Systems
Vanderbilt University
Nashville, Tennessee
mary.s.brown@vanderbilt.edu

***Abstract*—Cyber-Physical Systems (CPS) prevail in nearly every aspect of today's technological world. An example of a CPS is an autonomous aircraft controller that safely guides aircraft to its target destination. The purpose of this paper is to design a synchronous model that achieves a simplified collision avoidance system with N amount of planes. The model was developed and implemented to allow for N number of controllers, one for each aircraft in the simulation. The model successfully sends and receives the (x, y, z) locations of other aircraft in the vicinity to each aircraft controller in the model. The collision avoidance and flight algorithms for the aircraft to takeoff, fly, and land at its target destination are implemented in Python. The system proves successful for up to 2,500 aircraft reaching their target destination in a short amount of time while always avoiding collisions with other aircraft. Future work on this project can be adapting this system to an asynchronous model or integrate timing, while also preserving the safety of the system.**

***Index Terms*—component, modelling, task**

I. Introduction

A cyber-physical system (CPS) is a system in which various devices are sending messages to one another, while also receiving inputs from the physical world through sensors and actuators in a feedback loop. These systems have become so prevalent in today's society that it is nearly impossible to think of a technology that is not a CPS. A synchronous model is system in which all tasks within a clock cycle round execute instantaneously. One key feature of cyber-physical systems is that they are constantly reacting to different inputs from the environment or received from other devices. They are also constantly outputting some value. Two types of CPS are synchronous and asynchronous models.

In synchronous models, all tasks within a round execute instantaneously, meaning that all components executed in a sequence of logical rounds. All enabled tasks of that component are likewise executed. Synchronous systems are deterministic in this way in that all enabled tasks are executed. The order of the execution of tasks in a component can be modeled by task precedence and implemented in the system. Because this is computationally more expensive, asynchronous machines are more widely implemented in CPS. In asynchronous systems, only one enabled task is executed within a clock-step. This means that there is an element of nondeterminism in asynchronous systems. Without fairness, some tasks may never execute in an asynchronous system [1].

In this paper, a synchronous system for a CPS autonomous aircraft collision avoidance system is modeled and developed. Section II lays out the purpose and goals of the project to achieve all the required assumptions and aircraft functions. Section III details the design model of the synchronous system and the function of the controller component. Section IV details the implemented model and algorithms of the system in Python. Section V argues that the safety and liveness properties hold in the proposed model and implementation. Section VI details how this system would be implemented in an asynchronous manner with related works on turning synchronous systems to asynchronous systems. Finally, section VII concludes the paper and discusses the results of the model.

II. The Problem

The purpose of this project was to design a model that directs autonomous aircraft to each of their target destinations while avoiding collision with other aircraft that may come in their path. Assume the aircraft fly in the 3-D plane. Their source, current position, and destination are integer-valued points in the plane with an x-, y-, and z-value. Assume that all aircraft fly with a constant velocity v = 1 km/min. The aircraft has a direction of either 0°, 90°, 180°, or 270°. The flight of direction of 0° is the positive x-direction, 90° is the positive y-direction, 180° is the negative x-direction, and 270° is the negative y-direction. It can only rotate by 90° at a time. It has three possible forms of movement:

- An aircraft can fly along the X-axis.
- An aircraft can fly along the Y-axis.
- An aircraft can fly at an angle to increase or decrease its altitude. Depending on the direction of flight, it does this by moving one coordinate in the z-direction and either moving one coordinate space in the x- or y-coordinate.

To explain further the last form of movement, suppose an aircraft has the position (1, 1, 1) in the 3-D plane with a direction of 180°. In order to increase its elevation, the x-value is decremented and the z-value is incremented for the new current position of (0, 1, 2). Now, from this position, suppose the aircraft turns to the 90° direction. If the aircraft descends,

its y-value is incremented, and the z-value is decremented for the new current position of (0, 2, 1). For this project, the aircraft controller can either update its direction of flight or move forward every k = 1 minutes. In this project, one minute is equal to one clock cycle. Thus, in one clock cycle, the aircraft can either rotate by 90° or move forward by one coordinate space in its current direction of flight.

In order to perform collision avoidance, each aircraft has a cubic region of side length 2q = 2 km as a "warning zone." Assume that each coordinate space has a length of 1 km. Thus, the warning zone spans out 1 coordinate space from the aircraft, and the current aircraft is at its center. No aircraft should enter this warning boundary at any time during the flight as it would cause a collision. If an aircraft is notified of another aircraft in its warning zone, it will perform a necessary collision avoidance maneuver.

Thus, the goal of this project is to design a controller with a collision avoidance algorithm that is the same for each aircraft. The project is deemed successful if each aircraft takes off from its source destination, navigates to its target destination, and successfully lands with no collision to other aircraft.

III. Proposed Model

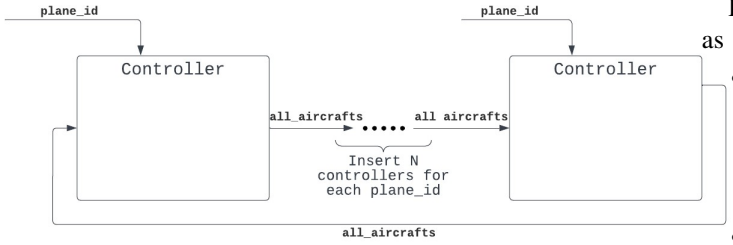

Fig. 1. Interface of proposed synchronous model

A synchronous model was developed to solve the problem described in the previous section as seen in Figure 1. As mentioned in the introduction, all enabled tasks in a component execute within that clock step, and this is assumed to be infinitely fast within a synchronous system. The component in this model is the Controller component(s). There can be 1 to N controllers, where N is the number of planes run in the synchronous system simulation. This completed system will require the 1st plane to pass its output, which is the input for the 2nd plane. The output of this 2nd plane is the input for the 3rd plane, and so on for N planes. The Nth plane's output is the input of the 1st plane. The process repeats itself until all aircraft's have successfully reached their target destination.

The controller component is a different instance for each aircraft, so there can be at most N controller components in the system. The controller has different, input, output, and state variables. The controllers' tasks implement the algorithm for an aircraft to reach its target destination without collision.

A. Input and Output Variables

The input variable for one aircraft is the current locations of all other aircraft in its vicinity. This means that the input is the current x, current y, and current z positions of type int of all other aircraft in the simulation. The output of that controller is the current location of the current aircraft, so its x, y, and z position of type int. In the proposed model, this all is represented as a variable called *all_aircrafts*. This holds and updates all the positions of the current locations of all aircrafts in the system. The output of a controller is thus the updated *all_aircrafts* data structure with the updated (x, y, z) coordinates for the aircraft in that clock cycle. Therefore, this data structure is used as input since the aircraft in question knows where all other aircraft are in its vicinity with their most updated coordinates. The controller of the current aircraft then outputs this data structure where it updates its (x, y, z) coordinates of the movement it just made. Each controller repeats this process for the aircraft it is controlling.

The way to know what current location in *all_aircrafts* belongs to which aircraft is by the other input into the controller called *plane_id*. This is unique for each aircraft and is of type string. In a system, there are N different *plane_id*, and it is what distinguishes each aircraft controller from another. This will be later expanded upon in Section IV.

B. State Variables

For each controller component, there are 13 state variables, as detailed below:

- **direction**: The state variable direction of type int indicates the direction of flight in the xy-plane with the possible values of 0, 90, 180, and 270. All aircraft controllers initialize this to 0, as all aircraft in this system are assumed to take off in the 0°direction.
- **current_x**, **current_y**, and **current_z**: These three state variables of type int indicate the current x, y, and z values of the (x, y, z) coordinate of the current aircraft that is running. *current_x* and *current_y* are initialized to the null value *None* since the initial position on the xy-plane is unknown for takeoff until *all_aircrafts* is first inputted into the controller. *current_z* is initialized to be 0 since all aircraft takeoff from the ground, which is assumed to be z = 0.
- **target_x** and **target_y**: These two state variables of type int indicate the target x and y values of the target destination for the aircraft. There is no z variable since all aircraft are assumed to land on the ground where z = 0. These are both initialized by the parameters of the target values passed to the controller upon instantiation.
- **landing** This state variable of type string tells whether the x or y value will be changed when the aircraft is descending towards z = 0. A value of "x" indicates that landing will either occur in the 0°or 180°direction. A value of "y" indicates that landing will occur in the 90°or 270°direction. It is initialized to the null value *None* since it is determined in the controller based on the distance of the aircraft's current position to its target destination.

- **warning**: This state variable of type Boolean indicates whether there is another aircraft on the warning boundary of the current aircraft. If it is false, then there is no warning. If it is true, then there is an aircraft(s) on the warning boundary.
- **warning_cube** and **warning_coordinates**: The state variable *warning_cube* is a list of (x, y, z) coordinates that indicate potential warning coordinates for the current aircraft. Because this is in 3-D, it resembles a cube around the aircraft, and is recalculated for each new current position of the aircraft. It is initialized as the empty list. The state variable *warning_coordinates* is also a list of (x, y, z) coordinates that is initialized the empty set. Another aircraft's coordinates is appended to this list if its coordinate matches one of the coordinates in *warning_cube*. These state variables help with collision avoidance maneuvers in the controller.
- **collision**: This state variable is of type Boolean and initialized to false. Its purpose is to indicate if there has been a collision in the system between aircraft. The algorithm developed and its liveness properties ensure that this will never be changed to true.
- **k** and **start_k**: These two state variables are both of type int and initialized to 0. The purpose of them is to ensure that only one action, whether rotation or moving forward in current direction, is taken at each clock cycle for that controller. These will be expanded upon in Section IV.

C. Tasks

For each controller instance, there are 11 tasks associated with it for the aircraft to successfully takeoff, direct itself to target position, land, and avoid collision with other aircraft in its vicinity. These tasks are detailed in Figure 2 with their read and write sets and the tasks precedence as shown in the arrows.

Tasks A is collecting the inputs and writing them to the current locations state variables. Task B keeps track of the clock variables to make sure only one action is executed by the aircraft in a single clock cycle. Task D is a safety monitor for collision that any aircraft are within the warning zone of the aircraft (the algorithm ensures that this will never be executed). Tasks C and E locates if any aircraft is on the warning boundary of the aircraft. Task G is only executed as an aircraft is taking off. Tasks H, I, and J are purposes of changing direction or the movement of the current aircraft if no previous action had been completed, such as a collision avoidance maneuver. Task K writes to the output variable *all_aircraft* to update the current position of that controller's aircraft. The actual algorithms for these tasks in movement and collision avoidance is expanded upon in Section IV.

IV. Implemented Model

The model was implemented in the Python programming language with version 3.9.9, and was tested on Mac, Windows, and Linux machines. The only required import is the random module. There are two python files: controller.py and system.py [2]. The system.py file is responsible for creating the synchronous system for the desired number of aircraft, as seen in Figure 1. The controller.py file implements a Python class called Controller that is utilized by each aircraft in the system in order to keep track of the different state variable values for each aircraft. It has two functions to the Controller class:

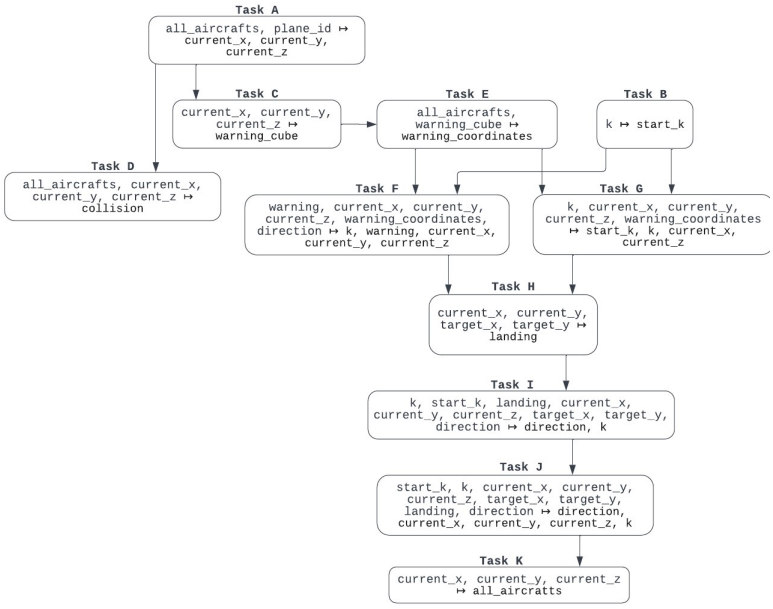

Fig. 2. Task Precedence

- **__init__**: This function initializes the state variables for each aircraft listed in Section III. Two parameters are passed into this function in order for the initialization of the target x and y locations to be different for each aircraft.
- **ClockCycle**: This function holds all the tasks for the controller that were detailed in Section III. It passes in the two input variables: *plane_id* and *all_aircraft*. It is called by each aircraft in the clock cycle that is executing.

A. System Assumptions

The system.py file implements a few important concepts in order for the synchronous system to run as intended. First, it uses the Python import random in order to randomly assign each aircraft a unique initial and target destination. No two aircraft can have these same coordinates. It also ensures that each aircraft's initial position is at least 1 km, or 1 coordinate space, away from its target destination. Thus, the system is running with the assumption that an aircraft cannot takeoff and land in the same destination. The initial and target locations of an aircraft are also only assigned with the bounds of $0 \leq x \leq 50$ and $0 \leq y \leq 50$. Thus, there are only 2,500 possible locations for planes. When starting the program and running system.py, it will ask for an input from 0 to 2,500 planes.

Another assumption that the system.py is implementing is that only messages about an aircraft's current location is sent to other aircraft if it is still in route to its target destination. Once an aircraft has reached its target destination, the aircraft controller is no longer called to execute and that aircraft's current location is removed from the *all_aircrafts* input and output variable for the system. Thus, system.py is keeping track of which controller in the system still need to be run through. It calls upon the specific controller by using that aircraft's unique plane ID. Once all aircraft have reached their respective target destination, the system stops running.

The controller.py file holds the assumption that an aircraft can only perform one action, either rotating or moving forward in its current direction, during each clock cycle. This is done by keeping track of two state variables: *k* and *start_k*. In Task A, *start_k* is assigned the value of *k*. When an action is performed by the aircraft's controller, *k* is incremented by one. For each task that would execute some motion of the aircraft, the value of *start_k* and *k* have to be equal in order for that task to be enabled. Thus, *start_k* is reset to the value of *k* at the beginning of each clock cycle in Task A in order to keep track that this assumption holds in our Python system.

B. Warning Zone

As mentioned in Section II, each aircraft has a 3-D cubic warning zone with side length of 2 km. Since each coordinate space has a length of 1 km to another coordinate space, the warning boundary is 1 coordinate space in all directions from the aircraft. In Task C of the controller, the coordinate spaces on the warning zone boundary are recalculated every clock cycle so it is update with the aircraft's current position.

Task D reads all the current positions of the other aircraft. If one of those aircraft is on the warning zone boundary, that location is appended to the list *warning_coordinates*. This list is reset every clock cycle to remove and add any aircraft that have arrived or left from the warning zone. After calculation, if one of the aircrafts is on the warning zone boundary, the Boolean state variable will be switched to True. This warning state variable will be set back to False in the next clock cycle before these two tasks are run. These two tasks are crucial for the safety of the system in order for the collision maneuvers to be correctly executed.

C. Collision Avoidance Algorithm

If the warning state variable is True, then Task F is executed for collision avoidance purposes. Furthermore, a collision avoidance maneuver is only run if any aircraft in the warning zone is located on the same z-plane as the current aircraft. This is because if two aircraft are on different z-planes, then there is no reason for the aircraft to alter its route since they will not collide in the same plane. In further tasks, if an aircraft needs to descend, then the coordinate space to descend to is verified available by the same *warning_coordinates* list.

If an aircraft on the warning zone boundary is on the same z-plane, the direction of the current plane is checked in order to see what maneuver is possible. The potential maneuvers is the same for each direction and is given in the priority with 1 being the first option and 4 being the last option of collision avoidance.

1) If there is a plane in the same z-plane, check to see if the current plane's next normal move would be the final descent to its target destination in its current direction. If it is, then descend in the direction to get the current aircraft's current z-value equal to 0.
2) Check to see if the coordinate space for the aircraft to ascend by one coordinate space in its current direction is available by checking if that location is occupied by another aircraft on its warning zone boundary. If it is available, then ascend by changing the x or y variable depending on direction and by incrementing the z-value.
3) If the next space forward in the aircraft's current direction is free on the same z-plane, then move forward by one coordinate space in the x or y direction.
4) Finally, if none of the above three options were possible or that location was occupied by another aircraft, then change its current direction by rotating a factor of 90. Once a collision avoidance maneuver has been executed, increment k by 1 to indicate that an action has already been taken by the aircraft in this clock cycle.

D. Takeoff, Flight, and Landing Algorithms

The aircraft takes off only if it is the first clock cycle in the system, such that *k* has the value of 0. This takes place in Task G in controller.py. Since all aircraft takeoff in the 0°direction, the current x and current z values of the aircraft are incremented by 1. This ascending takeoff task will only take place in the first clock cycle, so Task G is only enabled when *k* is equal to 0. This state variable is also incremented by 1 to ensure that the takeoff action is the only movement that happens by the aircraft in this clock cycle.

Once the aircraft is in flight, if no collision avoidance maneuvers were executed, then *start_k* should still be equal to *k*. There are several variables that keep track of what "normal" routing move the aircraft should make next. Task H first details what direction of descent the aircraft should take to its target destination. Descent is ideal if the length from either the x or y distance from its current location to its target location equals the altitude z-value. This is because both the z value and either the x or y value is changed during descent depending on its direction of flight. The algorithm sets the state variable *landing* equal to "y" if the distance of the current y to the target y positions of the aircraft is greater than the distance of the x's. Otherwise, if the distance of the current x to the target x positions of the aircraft is greater than the distance of the y's, then the landing is set to "x." This task thus helps us ensure that the aircraft has a greater chance of having the aircraft descend without additional maneuvers of direction changes.

Next, Task I aids Task H in ensuring that there is enough space to land in the x or y directions. This task first checks to see if no previous action has been made by the aircraft controller in this task. If not, then it checks if the aircraft still has enough room to descend and land depending on

the landing direction specified by the previous task. This is because landing can only take place if the distance of either the x or y positions of the current location to the target location is equal to the z value. For example, if the landing direction is ”x” and the distance from the current x value to the target x value is less than the z value, then the aircraft needs to give itself more room in the x-plane to land. If the aircraft's direction is 0 or 180, then the current aircraft is either incremented or decremented, respectively. If the direction is 90 or 270, then it will rotate to 0°to get in the x-direction. A similar process is done for if the landing direction is equal to ”y”. If this task executes, then k will be incremented.

Finally, if no other action has been made by the aircraft in this clock cycle, then Task J will execute for the aircraft. It first checks to see which x or y value from its target destination is less. For example, imagine the x distance is less than the y distance or the y value already equals its target y. If that is true, it then checks to be sure the current x does not equal its target x value and that the current z is not 0. If this is the case, then it will be given a prioritization of three maneuvers with the 1st given the top priority and the 3rd as a last option.

1) First, if the landing direction specified in Task H is ”x” and the distance of the x-value from its target x-value is equal to the current-z and the current y is equal to its target y, then this decision will execute. This decision aids in landing the aircraft. It will then check to see whether the current x value is less or greater than the target x. If less than, then direction should be 0°. If greater than, then direction should be 180°. If that respective direction is already satisfied, then move the x-value in that direction and decrement z. If not, then rotate aircraft to desired direction. Note that it will not descend if another aircraft is in its desired location. If this is the case, the aircraft will turn in hopes to land in the next clock cycle.
2) If the current x is less than the target x, then the aircraft should be in the 0°direction. If it already is, then the current x will be incremented by one, resembling the aircraft moving forward. If not, then it will rotate its direction by 90°in order to be at 0°or closer to it.
3) If the current x is greater than the target x, then the aircraft should be in the 180°direction. If it already is, then the current x will be decremented by one. If not, then it will rotate its direction by 90°in order to be at 180°or closer to it.

A similar process and reasoning occurs if the y distance is less than the x distance or the x value already equals its target x. It will then adjust the following prioritization to be for the y directions and values. Therefore, the controller will direct and move the aircraft from taking off, to moving and changing directions, to landing. In Task K, the current (x, y, z) coordinate values of the current aircraft is updated in *all_aircrafts* and that is outputted to be the input to the next aircraft's controller in the system.

V. Safety and Liveness Properties

The synchronous system developed ensures safety requirements will hold and that its liveness properties will occur. For safety, this means that an aircraft never collides with another aircraft. This is done by creating a safety monitor for monitoring the warning zone in the aircraft as discussed in Section IV. This safety monitor alerts the aircraft if another aircraft is on its warning boundary. This forces the aircraft in that clock cycle to prioritize a collision avoidance maneuver before its ”normal” routing algorithm as a response requirement.

This safety requirement is fulfilled by the tasks in the controller. This is because if a there is another aircraft on the warning zone boundary, then the current aircraft will be required to perform a collision avoidance maneuver due to its task precedence shown in Figure 2. Because this collision avoidance Task F detailed in the previous section only performs a maneuver if a plane is on the same z-plane, it does not descend in subsequent tasks without first checking to be sure that that position is not occupied by another aircraft. The aircraft also only ascends during take off and checks that that position is unoccupied as well. The only other time the aircraft ascends is if it has to avoid an aircraft directly in front of it in its direction in the xy-plane in Task F. Before it ascends, it checks to see if that position is available. This safety requirement was proven to hold in all cases because the Task D never executed. The purpose of this task was to see if two (x, y, z) locations of different planes were ever exactly the same. If so, the code would exit with the message there had been a collision. In the final prototype, there were no collisions made for all possible cases of aircraft maneuvers. The code was tested over 100 times to ensure that it never executed. Even with the maximum number of planes of 2,500 allowed in the bounded space with collision avoidance maneuvers in nearly every clock cycle, there was still no collision. Thus, it can be reasonable argued that the safety requirement that there will never be a collision is satisfied.

The liveness property is that $\Diamond[(current\_x == target\_x) \wedge (current\_y == target\_y)] \wedge \Box\neg collision$. This states that the aircraft will eventually reach its target destination and will always have no collisions. The always no collision was proven in the previous paragraph. It can be argued that the aircraft will always eventually reach its target destination. This is because the aircraft is continually moving and turning towards its target destination as expanded upon in Section IV. Another argument that can be made is that when a collision avoidance maneuver has to be made, it prioritizes first landing if applicable and then moving forward in intended direction if available. Thus, its movements, while prioritizing safety, are also always intended to keep the aircraft toward its target destination. With all the testing even with 2,500 planes, all planes eventually reached their target destination always without collision. Thus, this liveness property is satisfied through the the algorithms developed and expansive testing of all edge cases.

VI. Future and Related Work

As mentioned in the Introduction, asynchronous systems exhibit nondeterminism in choosing which task to execute in one lock-step round. This requires asynchronous systems to have fairness in order to avoid some tasks never executing. Because an asynchronous system is less computationally expensive than a synchronous model, future work for this project could include adapting this to an asynchronous model. In order for this to occur, all tasks within the controller should be implemented with strong fairness assumptions. Strong fairness ensures that a repeatedly enabled task should eventually be executed[1]. Strong fairness should be given to each aircraft's controller because autonomous aircraft require many safety task precautions in order to ensure that aircraft do not collide with one another.

In his work, Lee discusses how deterministic models can be preserved without computationally extensive methods. Two projects that are detailed in this project is PRET. PRET machines enable programs with repeatable timing to assure that a system behaves in a more deterministic manner than asynchronous nondeterministic machines. Another model that is mentioned is the Ptides programming model that gives interactions between the cyber and physical world a tolerance on timing based on synchronization clock errors. Thus, this paper demonstrates how these two model scan make systems in the real-world more practical through more precise methods of timing[3]. The proposed aircraft collision avoidance system in this paper could benefit from one of these deterministic models since it will work in the intended way with predicable behavior.

Thus, future work should include either transforming the aircraft model into an asynchronous model or by integrating aspects of the determinstic models in Lee's research with real-time. With the addition of timing, the aircraft can have better understanding of where other aircraft are due to the timing of the messages sent and received between the two. Implementing time in this system is more realistic since the aircraft are constantly moving and changing positions. These two models proposed can be used to make a more realistic aircraft collision avoidance system with less computationally expensive models such as synchronous systems.

VII. Conclusion

The aircraft collision avoidance system proposed in this paper was modeled as a synchronous system. Thus each enabled task was executed in each clock round. The feedback loop detailed in the model allowed each aircraft's controller to be able to send and receive messages about its current location to all other aircraft in the vicinity. The task dependence of the controller allowed for the system to prioritize safety collision avoidance maneuvers before its normal routing. The routing algorithm of takeoff, flying, direction, and landing ensured that the aircraft would always reach its target destination. Edge tests were manually coded for testing purposes to prove that the algorithm passed all collision avoidance maneuvers. The code was executed over 100 times to ensure that the final prototype had no collision.

A deeper understanding of CPS modeling was also acheived through this process. This is because safety and liveness requirements were modeled in synchronous sytems. A safety monitor to alert the system that a collision avoidance maneuver was also needed. Research was done on how this synchronous system could be adapted to an asynchronous model using fairness. Other future work could be adapting different modeling approaches to include time to be able to achieve more realistic timing of the aircraft based on their sending and receiving of messages from other aircraft. Thus, this report proved a succesful aircraft collision avoidance sychronous system, while also thinking through how it could incorporate nondeterminsim and timing modifications.

References

[1] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[2] Mary Stirling Brown. *Marystirling/aircraft-collision-avoidance-controller*. URL: https://github.com/marystirling/Aircraft-Collision-Avoidance-Controller.

[3] Edward A. Lee. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models". In: *Sensors* 15.3 (2015), pp. 4837–4869. ISSN: 1424-8220. DOI: 10.3390/s150304837. URL: https://www.mdpi.com/1424-8220/15/3/4837.