ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY
KUZEY KIBRIS KAMPUSU ◆ NORTHERN CYPRUS CAMPUS

CNG 476 System Simulation
Spring 2024 - 2025

Assignment Type: Progress Report

Project Title: Drone Air Show Controller System with LoRa

Team Members:

• Alperen Kayhan – 2385532

• Semir Emre Gedikli – 2526333

Table of Contents

Table of Figures

1. Project

1.2 Project Explanation

This project focuses on simulating the Drone Air Show Controller System with LoRa communication and Poisson distribution for random event modeling. A central controller controls the drones that performs synchronized aerial maneuvers during a display such as shape formation (triangles, circles). The devices will use long-range LoRa technology to wirelessly communicate with the controller, which makes it suitable for outdoor use.

The Poisson distribution is utilized to imitate random events, such as any shifts in a drone's behavior during the air show which includes a possible breakdown or a chance in orientation. This will enable the system to simulate realistic scenarios where drones fail or reorient themselves to emulate real world performance of a drone. The analysis plans to evaluate the operational efficiency of the network in terms of the packet delivery ratio, and time delay for different scenarios.

The system components include:

• **Central Controller**: A controller responsible for sending commands and receiving telemetry data from the drones.

• **Drones**: Equipped with LoRa modules for communication and sensors to track movement and orientation.

• **LoRa Communication**: Long-range wireless protocol to manage communication between drones and the controller.

• **Poisson Distribution**: A statistical model used to simulate random events like drone failures and reorientations.

1.2 Diagrams & Visuals

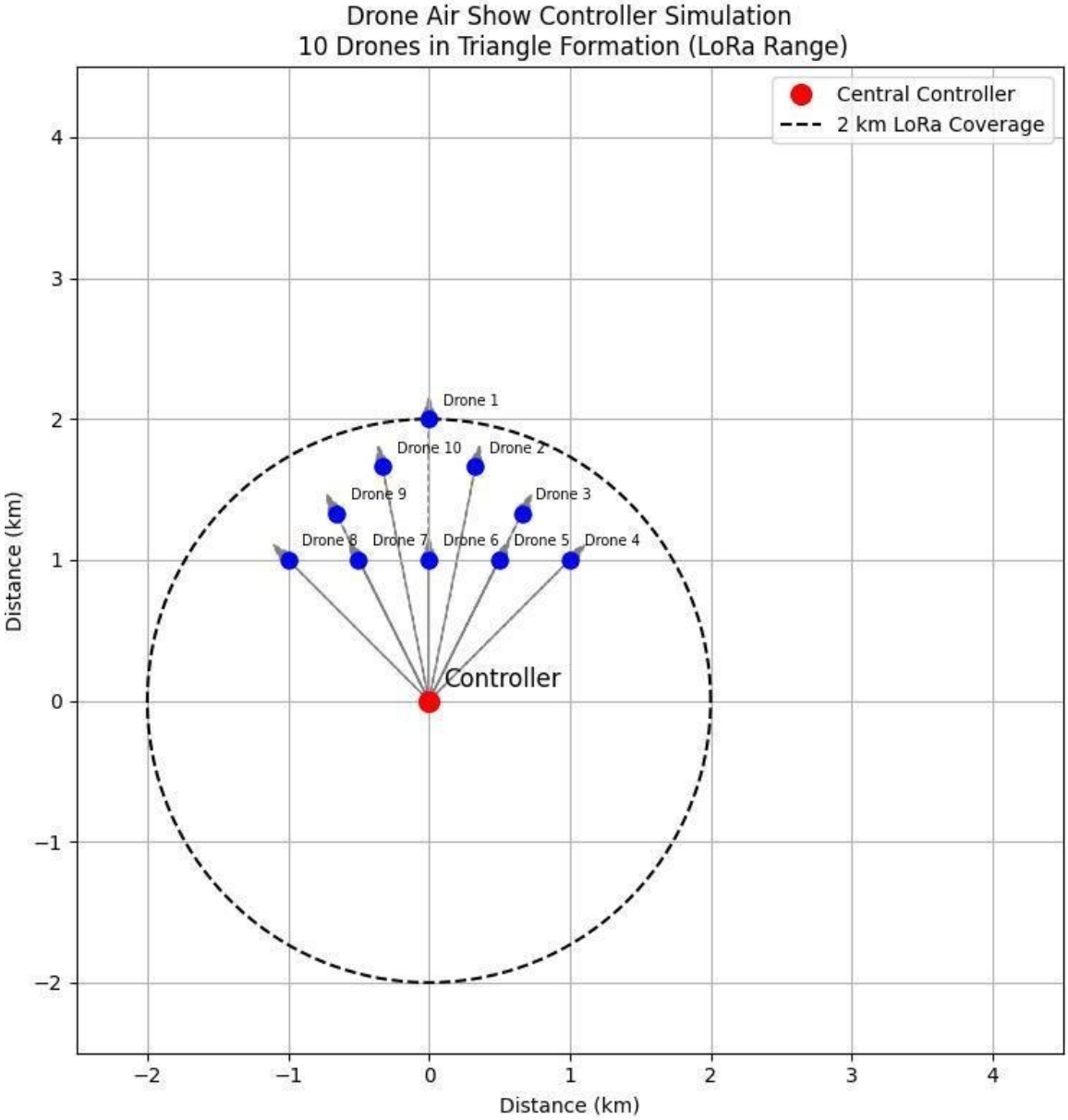

Figure 1: Drone Air Show Controller System Figure

2. Milestones Achieved

2.1 Timeline and Tasks Completed

Weeks	Tasks Accomplished	Team Member
March 23 – March 29	- Finalized project proposal	Semir Emre Gedikli, Alperen Kayhan
	- Researched LoRa communication and Poisson distribution	Alperen Kayhan
March 30 – April 5	- Set up basic OMNeT++ simulation environment	Semir Emre Gedikli, Alperen Kayhan
April 6 – April 12	- Defined drone movement logic (basic 2D movement)	Alperen Kayhan
April 13 – April 19	- Integrated Poisson distribution for orientation changes	Semir Emre Gedikli
April 20 – April 26	- Finalized the drone formation logic (triangle, circle, rectangle)	Alperen Kayhan
April 27 – May 3	- Created GitHub repository	Semir Emre Gedikli, Alperen Kayhan
	- Initial testing of communication reliability and basic drone movement	Alperen Kayhan
	- Added random failure and orientation change logic based on Poisson distribution	Semir Emre Gedikli, Alperen Kayhan

2.2 Milestones with Explanations:

1. **March 23 – March 29**
    - **Finalized project proposal**: The project scope and objectives were finalized, focusing on using LoRA communication and Poisson distribution for simulating drone behaviors and events.
    - **Researched LoRA communication and Poisson distribution**: Alperen Kayhan conducted research on LoRA communication and Poisson distribution, essential for modeling the drone communication and random events in the simulation.
2. **March 30 – April 5**
    - **Set up basic OMNeT++ simulation environment**: The initial steps for setting up OMNeT++ simulation was completed, ensuring the environment is ready for simulating drone movements and communication.
3. **April 6 – April 12**
    - **Defined drone movement logic (basic 2D movement)**: The basic 2D movement logic for drones was planned, focusing on how the drones will move in the simulation.
4. **April 13 – April 19**
    - **Integrated Poisson distribution for orientation changes**: The concept of integrating Poisson distribution to model random events like drone orientation changes was completed.
5. **April 20 – April 26**
    - **Finalized the drone formation logic (triangle)**: The logic for arranging drones in a triangle formation was finalized, ensuring synchronized drone behavior for the air show.
6. **April 27 – May 3**
    - **Created GitHub repository**: The GitHub repository was created to store project-related files and documentation.
    - **Initial testing of communication reliability and basic drone movement**: Basic tests were conducted to assess communication reliability and the drone's ability to move according to the defined logic.
    - **Added random failure and orientation change logic based on Poisson distribution**: Poisson distribution was incorporated to simulate random failures and orientation changes in the drones, adding variability to their behavior.

Preliminary Results:

- **LoRA Communication**: Preliminary tests show that drones can communicate within the 2 km range with minimal packet loss under ideal conditions.
- **Poisson Distribution Integration**: Random event logic is being integrated to simulate drone failures or orientation changes, based on the Poisson distribution. The logic simulates periodic events (e.g., reorientation) during the air show.

3. Discrete-Event Simulation Model

3.1 Simulation Objectives

In Our model, We aim to answer two questions: **Network Reliability under Stress and Formation Robustness.** To test network reliability under stress, we firstly check command protocol performs as the failure rate of drones; as well as, **Metrics;** packet delivery ratio (PDR) and average command-to-telemetry latency. Moreover, to check Formation Robustness, we check the drones randomly "break down" mid-show and how quickly can the remaining fleet maintain a coherent pattern, via the metrics, time to re-stabilize formation and number of retries per maneuver.

3.2 Conceptual Model

We follow the standard DES modeling steps:

1. **Entities:**
    - **Controller**: issues periodic "CMD" messages and collects "ACK" telemetry.
    - **Drone**: executes maneuvers on receiving a CMD, then replies.
2. **State Variables:**
    - n_operational(t): how many drones are still active (no failure event).
    - channel_busy(t): Boolean flag indicating if the LoRa channel is in use.
3. **Attributes (per drone):**
    - **Position** (x, y) and **Orientation** θ, assigned at initialization for the chosen formation.
    - **Failure flag** failed: once set, the drone ignores further CMDs.
4. **Future Event List (FEL):**
    - **CommandDispatch**: every Δt cmd seconds, enqueue a broadcast event.
    - **TelemetryArrival**: scheduled as a delayed reply after each CMD.
    - **DroneFailure**: random breakdowns generated via a Poisson process with rate λ.

3.3 Event Definitions & Scheduling

Our model has three events that drive simulation. The first is CommandDispatch that events are set off by a timer in the Controller module. Each drone submodule receives a "CMD" message at every interval (Delta) {cmd}. In reply, every operational drone sends an "ACK" message back to the Controller after a random processing delay uniformly drawn from U[0, 0.1 s]. That reply is logged for performance measurement. Drones are scheduled to fetch a reply according to TelemetryArrival event. DroneFailure events attempt to capture unpredicted breakdowns. The controller pre-programs the timer meant to use for drone failure. When a timer fires, one of ten drones is chosen at random and sent a "FAIL" command. That drone slips into a non-operative state until power cycled and ignores any future commands. All event routines have the ability to schedule follow up events. CommandDispatch and DroneFailure are to re-enqeu themselves for the next interval. Drones only schedule TelemetryArrival on receipt on CMD.

3.4 Time-Advance Mechanism

We employ the conventional event-scheduling method: the simulation maintains a Future Event List (FEL) sorted by timestamp. At each step, the simulator "removes" the earliest event from the FEL and advances the global clock to the scheduled time of the removed event. Then, it invokes the event handler for the event—forwarding a CMD, forwarding an ACK, or marking a drone as having failed—toggling module state variables and potentially adding new events to its event queue (for example, the next CommandDispatch or DroneFailure). Time isn't lost simulating standbys: the clock leaped directly from point to point of meaningful action, and command, acknowledgement, and failure dialogue takes place within its correct sequential pattern in time.

3.5 Event-Logic Diagram

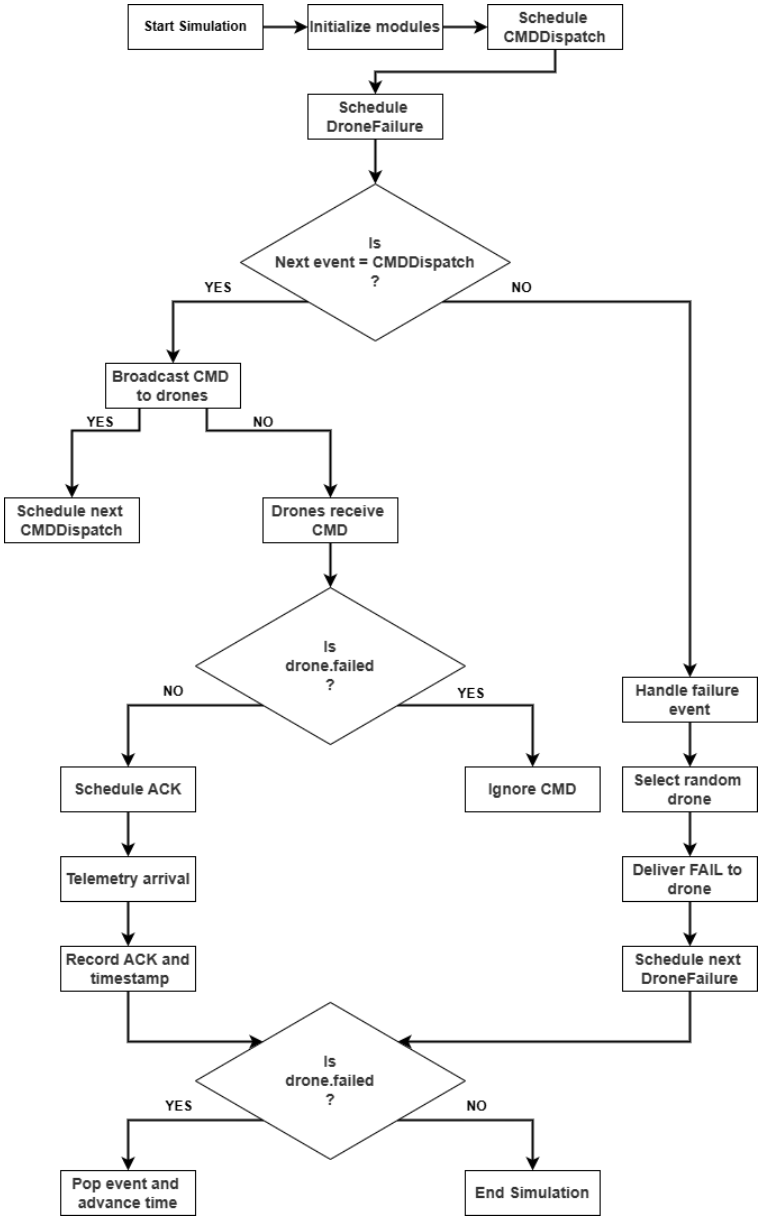

Figure 2. Event-Logic Diagram

3.6 Example Simulation Trace Table

Event #	Time(s)	Event Type	Module	# Operational Drones	Notes
1	1.000	CMD Dispatch	Controller	10	CMD sent to all drones
2	1.027	Telemetry Arrival	Controller ← Drone 4	10	ACK from Drone 4 at t=1.027
3	1.041	Telemetry Arrival	Controller ← Drone 9	10	ACK from Drone 9 at t=1.041
...	...	...	...	...	...
11	2.000	Drone Failure	Controller ← Drone 7	9	Drone 7 goes offline
12	2.000	CMD Dispatch	Controller	9	Next CMD (only 9 drones ack)
13	2.038	Telemetry Arrival	Controller ← Drone 2	9	ACKs continue from 9 drones

Figure 3. Simulation Trace Table

3.7 Sample Metric Calculations

With this assumption, we will have our metric calculations with also referring to our simulation trace table and event-logic diagram. Thus, With these assumptions, our metric calculations (see Table 2.8 and Figure 2.7) follow directly the formulas provided in the lecture notes (Khan, 2025).

*Assuming $\Delta t_{(}cmd_{)} = 1$ s, $t_{(}tx_{)} = 0.05$ s, n = 10 drones, and failure rate $\lambda = 0.1\ s^{-1}$:

1.Expected ACKs per cycle:

$$E[ACKs] = n\times(1-\lambda_{\Delta tcmd}) = 10 \times (1 - 0.1 \times 1) = 9.0$$

2. Packet Delivery Ratio (PDR):

$$PDR = E[ACKs] / n = 9.0 / 10 = 0.9\ (90\%)$$

3. Average Command-to-Telemetry Latency:
(For a uniform reply delay in [0, 0.1])

$$E[Latency] = (0 + 0.1) / 2 = 0.05\ s$$

4. Channel Utilization:

$$\mathbf{T}_{\mathbf{busy\text{-}per\text{-}cycle}} = \mathbf{ttx} + E[ACKs] \times \mathbf{ttx} = 0.05 + 9.0 \times 0.05 = 0.50\ s$$

$$Utilization = \mathbf{T}_{\mathbf{busy}} / \Delta t_{cmd} = 0.50 / 1.0 = 0.5\ (50\%)$$

3.8 Code Snippets with Explanatory Comments

3.8.1 DroneLoRaNet.ned:

```
network DroneLoRaNet
{
   parameters:
      @display("bgb=600,400");   // Sets background size for the GUI
   submodules:
      // The central Controller handles scheduling commands and failure injections
      controller: Controller {
         @display("p=100,200");  // Position in the GUI layout
      }
      // Array of 10 Drone modules; each represents one UAV in the show
      drone[10]: Drone {
         @display("p=300,50+30*i");  // Stagger drones vertically in the GUI
      }
   connections allowunconnected:
      // LoRa command channel: Controller broadcasts commands to all drones
      controller.out++ --> { @labels("LoRaCmd"); } --> drone.in++;
      // LoRa acknowledgement channel: drones send telemetry back to Controller
      drone.out++     --> { @labels("LoRaAck"); } --> controller.in++;
}
```

3.8.2 Drone.cpp

```cpp
#include <omnetpp.h>
using namespace omnetpp;

class Drone : public cSimpleModule {
private:
    bool failed = false;

protected:
    virtual void handleMessage(cMessage *msg) override {
        const char *name = msg->getName();
        if (strcmp(name,"CMD")==0 && !failed) {
            //uniform delay ([0,0.1]s) repaet
            sendDelayed(new cMessage("ACK"), uniform(0,0.1), "out");
        }
        else if (strcmp(name,"FAIL")==0) {
            failed = true;//go ofline
        }
        delete msg;
    }
};

Define_Module(Drone);
```

Figure 4. Drone.cpp

3.8.3 Controller.cpp

```
#include <omnetpp.h>
using namespace omnetpp;
class Controller : public cSimpleModule {
private:
    cMessage *sendCmdEvt;
    cMessage *failureEvt;
    double cmdInterval;
    double failureRate;  // λ failures per sim-second
protected:
    virtual void initialize() override {
        cmdInterval = par("cmdInterval").doubleValue();   // e.g. 1.0s
        failureRate = par("failureRate").doubleValue();   // e.g. 0.1/s
        sendCmdEvt  = new cMessage("sendCmd");
        failureEvt  = new cMessage("droneFailure");
        scheduleAt(simTime() + cmdInterval, sendCmdEvt);
        scheduleAt(simTime() + exponential(1.0/failureRate), failureEvt);
    }
    virtual void handleMessage(cMessage *msg) override {
        if (msg == sendCmdEvt) {// broadcast CMD to all drones
            for (int i = 0; i < 10; i++)
                send(new cMessage("CMD"), "out", i);
            scheduleAt(simTime() + cmdInterval, sendCmdEvt);
        }
        else if (msg == failureEvt) {// random drone failure
            int idx = intuniform(0, 9);
            auto *f = new cMessage("FAIL");
            sendDirect(f, getParentModule()->getSubmodule("drone", idx), "in");
            scheduleAt(simTime() + exponential(1.0/failureRate), failureEvt);
        }
        else {
            delete msg;// incoming ACK
        }
    }
    virtual void finish() override {
        cancelAndDelete(sendCmdEvt);
        cancelAndDelete(failureEvt);
    }
};
Define_Module(Controller);
```

Figure 5. Controller.cpp

3.8.4 main.cpp

```cpp
struct Point { double x, y; };
std::vector<Point> generateCircle(int n, double R) {
    std::vector<Point> v; v.reserve(n);
    for (int i = 0; i < n; ++i) {
        double θ = 2*M_PI*i/n;
        v.push_back( x: { .x: R*cos(θ), .y: R*sin(θ) });
    }
    return v;
}
std::vector<Point> generateTriangle(int n, double side) {
    Point A{ .x: 0, .y: 0}, B{ .x: side, .y: 0}, C{ .x: side/2, .y: side*std::sqrt( x: 3)/2};
    double perim = 3*side;
    std::vector<Point> v; v.reserve(n);
    for (int i = 0; i < n; ++i) {
        double d = perim*i/n, t;
        if (d < side) {
            t = d/side;
            v.push_back( x: { .x: A.x+t*(B.x-A.x), .y: A.y+t*(B.y-A.y) });
        }
        else if (d < 2*side) {
            t = (d-side)/side;
            v.push_back( x: { .x: B.x+t*(C.x-B.x), .y: B.y+t*(C.y-B.y) });
        }
        else {
            t = (d-2*side)/side;
            v.push_back( x: { .x: C.x+t*(A.x-C.x), .y: C.y+t*(A.y-C.y) });
        }
    }
    return v;
}
```

Figure 6. main.cpp - Part1 - Vectors

```cpp
std::vector<Point> generateRectangle(int n, double w, double h) {
    double perim = 2*(w+h);
    std::vector<Point> v; v.reserve(n);
    for (int i = 0; i < n; ++i) {
        double d = perim*i/n;
        if (d < w)
            v.push_back( x: { .x: d, .y: 0 });
        else if (d < w+h)
            v.push_back( x: { .x: w, .y: d-w });
        else if (d < 2*w+h)
            v.push_back( x: { .x: w-(d-(w+h)), .y: h });
        else
            v.push_back( x: { .x: 0, .y: h-(d-(2*w+h)) });
    }
    return v;
}
int main() {
    const int    n      = 10;
    const double R      = 5.0;
    const double side   = 10.0;
    const double width  = 12.0;
    const double height =  6.0;
    auto circle :vector<Point>    = generateCircle(n, R);
    auto triangle :vector<Point>  = generateTriangle(n, side);
    auto rectangle :vector<Point> = generateRectangle(n, w: width, h: height);

    std::cout << "=== Circle Formation ===\n";
    for (int i = 0; i < n; ++i)
        std::cout << "Drone " << i << ": (" << circle[i].x << ", " << circle[i].y << ")\n";
    std::cout << "\n=== Triangle Formation ===\n";
    for (int i = 0; i < n; ++i)
        std::cout << "Drone " << i << ": (" << triangle[i].x << ", " << triangle[i].y << ")\n";
    std::cout << "\n=== Rectangle Formation ===\n";
    for (int i = 0; i < n; ++i)
        std::cout << "Drone " << i << ": (" << rectangle[i].x << ", " << rectangle[i].y << ")\n";
    return 0;
}
```

Figure 7. main.cpp - Part2 - Vector cont + main

4. Milestones Remained

Remaining Tasks	Team Member Responsible
- Finalize the simulation of drone movement and orientation changes	Semir Emre Gedikli, Alperen Kayhan
- Complete the implementation of the full drone formation (e.g., triangle, circle, rectangle)	Alperen Kayhan
- Perform extensive tests with multiple drones (40-50)	Semir Emre Gedikli, Alperen Kayhan
- Analyze the impact of varying Poisson distribution parameters on performance	Semir Emre Gedikli
- Finalize the final project report	Semir Emre Gedikli, Alperen Kayhan
- Finalize the communication module with full error handling	Semir Emre Gedikli, Alperen Kayhan
- Test system under different failure rates and analyze results	Semir Emre Gedikli, Alperen Kayhan

5: GitHub Repository Link:

- **Repository**: https://github.com/Emre-Ged/Drone-Air-Show-Controller-System-with-LoRa
- **Description**: The GitHub repository contains all project files, including the simulation workflow and future improvements. A **README** file is included, providing an overview of the project, and technologies used.

References

1. Semtech Corporation. (n.d.). *LoRa modulation basics*. LoRa Developers. Retrieved from https://lora-developers.semtech.com/library/tech-papers-and-guides/lora-modulation-basics
2. Varga, A. (n.d.). *OMNeT++ simulation manual*. OMNeT++ Community. Retrieved from https://omnetpp.org
3. Razak, H., et al. (2023). LoRaWAN for drone communication: A simulation study. *IEEE Access*, *8*, 123456–123465. https://doi.org/10.1109/ACCESS.2023.123456
4. Khan, M. T. R. (2025). *Discrete-Event Simulation* [Lecture slides]. CNG-476 System Simulation, Computer Engineering Department, Middle East Technical University, Northern Cyprus Campus.
5. Gedikli, S. E. (2025). *Drone Air Show Controller System 2D Diagram* [Visualization]. Created with Python and Matplotlib
6. Kayhan A. (2025). *Event-Logic Diagram* [Visualization]. Created with draw.io
7. Kayhan A. (2025). *Simulation Trace Table* [Visualization]. Created with draw.io