AeroGuard-MAS

Multi-Agent Airspace Conflict Manager

Intelligent Systems Engineering

Alex Testa

July 20, 2026

Abstract

AeroGuard-MAS is a didactic intelligent system for simulated airspace conflict management. The project combines a Kotlin simulation core, Jason/AgentSpeak(L) BDI agents, tuProlog symbolic reasoning, a small STRIPS-style planning layer, JSON scenarios, JSONL event logging, and a Python events visualizer. The goal is not to build an operational aviation system, but to engineer a clear, testable and explainable prototype that demonstrates how multi-agent modelling, declarative reasoning and automated planning can cooperate inside a software system.

Contents

1 Introduction **5**

1.1 Project Overview 5

1.2 Motivation 5

1.3 Educational Purpose 6

1.4 Main Contributions 6

2 Domain Context **8**

2.1 Airspace Conflict Management 8

2.2 Simplified Flight Model 8

2.3 Separation Rules 9

2.4 TCAS Inspiration 9

2.5 Accident Patterns Used as Inspiration 10

2.6 Human Factors and Biases 10

3 Requirements and Scope **12**

3.1 Functional Requirements 12

3.2 Non-Functional Requirements 13

3.3 Scope Delimitation 13

3.4 Main Scenarios 14

4 Architecture **15**

4.1 Overall Architecture 15

4.2 Package Organization 15

4.3 Domain Layer 16

4.4 Simulation Layer 16

4.5 Reasoning Layer 16

4.6 Planning Layer 16

4.7 Event Layer 17

4.8 GUI Architecture 17

4.9 Design Rationale 17

5 Mapping to Intelligent Systems Engineering Topics **19**
5.1 Overview . . . 19
5.2 AgentSpeak and BDI Programming . . . 19
5.2.1 Role in the Project . . . 19
5.2.2 Beliefs, Goals and Intentions . . . 20
5.2.3 Why AgentSpeak Is Useful Here . . . 20
5.3 Prolog and Symbolic Reasoning . . . 20
5.3.1 Role in the Project . . . 20
5.3.2 Facts . . . 21
5.3.3 Rules . . . 21
5.3.4 Why Prolog Is Useful Here . . . 22
5.4 STRIPS and Automated Planning . . . 22
5.4.1 Role in the Project . . . 22
5.4.2 Action Model . . . 23
5.4.3 Planning Problem . . . 23
5.4.4 Secondary Conflict Prevention . . . 23
5.5 Integration of the Three Topics . . . 24

6 Implementation **25**
6.1 Technology Stack . . . 25
6.2 Scenario Loading . . . 26
6.3 Simulation Engine . . . 26
6.4 Managed Simulation Engine . . . 27
6.5 Maneuver Application . . . 27
6.6 Event Logging . . . 28
6.7 Explanation Layer . . . 28
6.8 Graphical User Interface . . . 28

7 Scenarios and Demonstration **30**
7.1 Core Scenarios . . . 30
7.1.1 Simple Conflict . . . 30
7.1.2 Emergency Priority . . . 30
7.1.3 Secondary Conflict . . . 30
7.1.4 Weather Replanning . . . 31
7.1.5 No Conflict Baseline . . . 31
7.2 Stress Scenarios . . . 31
7.3 Demo Flow . . . 31
7.4 Command-Line Execution . . . 32
7.5 GUI Execution . . . 32

8 Testing and CI/CD **33**
8.1 Testing Strategy . . . 33
8.2 Domain and Scenario Tests . . . 34
8.3 Simulation and Conflict Tests . . . 34
8.4 Reasoning Tests . . . 34
8.5 Planning Tests . . . 34
8.6 Event and GUI Validation Tests . . . 34
8.7 Jason Smoke Tests . . . 35
8.8 Continuous Integration . . . 35
8.9 Release and Documentation . . . 35
8.10 Reliability Benefits . . . 35

9 Evaluation **36**
9.1 Functional Evaluation . . . 36
9.2 Engineering Evaluation . . . 36
9.3 Intelligence Evaluation . . . 37
9.4 Limitations . . . 37
9.5 Stress Test Results . . . 38
9.6 Overall Assessment . . . 38

10 Declaration on the Use of Artificial Intelligence **39**
10.1 Purpose of This Declaration . . . 39
10.2 How AI Was Used . . . 39
10.3 How AI Was Not Used . . . 40
10.4 Impact on the Project . . . 40

11 Conclusion **41**
11.1 Summary . . . 41
11.2 Main Achievements . . . 41
11.3 Lessons Learned . . . 42
11.4 Future Work . . . 42

List of Tables

2.1 Mapping between real aviation concepts and AeroGuard-MAS abstractions 9

5.1 Role of course topics inside AeroGuard-MAS 24

Chapter 1

Introduction

1.1 Project Overview

AeroGuard-MAS, short for *Multi-Agent Airspace Conflict Manager*, is a university project developed for the course of Intelligent Systems Engineering. The project implements a simulated airspace management system in which multiple aircraft move through a simplified sector while maintaining horizontal and vertical separation.

The central idea of the project is to show how an intelligent system can be engineered by combining different paradigms: multi-agent modelling, symbolic reasoning, automated planning, simulation, structured logging, explainability, testing and visualization. The system is not intended to reproduce the full complexity of aviation operations. But still expressive enough to demonstrate non-trivial intelligent behaviour.

The system loads scenarios from JSON files, simulates aircraft movement over discrete ticks, detects current and predicted conflicts, evaluates safety constraints through a symbolic Prolog reasoner, generates corrective plans using a STRIPS-style planner, applies selected maneuvers to the simulation state, and exports all relevant events as JSONL. A separate Python GUI reads the JSONL stream and visualizes the simulation as a replay.

1.2 Motivation

Airspace conflict management is a meaningful domain for an Intelligent Systems Engineering project because it naturally involves perception, reasoning, decision-making, coordination and explanation. Even in a simplified simulation, the system must answer questions such as:

- Which aircraft are currently unsafe?
- Which aircraft may become unsafe in the near future?

- Which aircraft should maneuver?
- Is a proposed maneuver allowed?
- Does a maneuver solve the original conflict?
- Could a maneuver create another conflict?
- Why was a specific maneuver selected?

These questions are not only numerical. They also involve symbolic rules, priorities, goals, constraints and explanations. This makes the domain appropriate for combining Kotlin engineering, Jason BDI agents, Prolog reasoning and STRIPS planning.

1.3 Educational Purpose

AeroGuard-MAS is a didactic prototype. It is not a safety-critical aviation system and must not be interpreted as one. The simulation uses a simplified two-dimensional space, discrete ticks, discrete flight levels and simplified aircraft dynamics.

The educational value of the project lies in its architecture and in the integration of intelligent components. The project demonstrates how different techniques studied in the course can be connected inside an executable, testable and observable software system.

1.4 Main Contributions

The main contributions of the project are:

- a typed Kotlin domain model for aircraft, routes, conflicts, maneuvers and simulation states;
- a JSON scenario loader with validation;
- a tick-based simulation engine;
- current and predicted conflict detection;
- a tuProlog-based symbolic reasoning layer;
- a STRIPS-style planning layer for conflict resolution;
- real Jason/AgentSpeak(L) source files representing BDI agents;
- secondary conflict prevention through forward simulation;

- weather-zone replanning;
- structured JSONL event logging;
- a Python Streamlit GUI for replay visualization;
- automated tests and CI/CD through GitHub Actions;
- project documentation through MkDocs and Kotlin API documentation through Dokka.

Chapter 2

Domain Context

2.1 Airspace Conflict Management

In real aviation, aircraft separation is a fundamental safety requirement. Aircraft must be kept apart horizontally and vertically, and controllers and onboard systems must anticipate future loss of separation before it becomes critical. A conflict is not only a matter of two aircraft being close in space. It also depends on altitude, trajectory, time and operational constraints.

AeroGuard-MAS abstracts this problem into a simplified airspace sector. Aircraft move in a two-dimensional plane and have a discrete altitude expressed in feet. Time advances in discrete ticks. Aircraft follow routes composed of waypoints, and each tick moves them closer to their active waypoint.

2.2 Simplified Flight Model

The project does not simulate aerodynamics. In real flight, an aircraft is affected by lift, weight, thrust and drag. Pilots control heading, altitude, speed and configuration. AeroGuard-MAS reduces these concepts to a computational model suitable for reasoning and planning.

The mapping between the real domain and the project model is summarized in Table 2.1.

Table 2.1: Mapping between real aviation concepts and AeroGuard-MAS abstractions

Real-world concept	**AeroGuard-MAS abstraction**
Geographical position	2D coordinates
Altitude or flight level	`FlightLevel` in feet
Aircraft speed	`Velocity` in units per tick
Navigation fix	`Waypoint`
Flight plan	`Route`
Time	Discrete simulation tick
Conflict avoidance logic	Agents, Prolog rules and planner
Operational event	Dynamic scenario event

2.3 Separation Rules

The system uses configurable separation thresholds. A pair of aircraft is considered unsafe when both horizontal and vertical separation are below their thresholds.

A typical configuration is:

```
{
  "separation": {
    "horizontal": 5.0,
    "vertical": 1000
  }
}
```

This means that two aircraft are in conflict only if their horizontal distance is less than 5.0 simulation units and their vertical distance is less than 1000 feet. This distinction is important. Two aircraft may appear visually aligned on the 2D map while still being safe because they are separated vertically.

2.4 TCAS Inspiration

AeroGuard-MAS is conceptually inspired by the idea of collision avoidance systems such as TCAS/ACAS. TCAS observes nearby aircraft, evaluates collision risk and may issue traffic or resolution advisories to the flight crew. In particular, TCAS II is associated with vertical resolution advisories.

AeroGuard-MAS is not an implementation of TCAS. It does not reproduce the certified algorithms, timing requirements or operational rules of TCAS. However, it follows a similar conceptual pattern:

observe traffic → predict conflict → evaluate safety → select maneuver → explain decision

2.5 Accident Patterns Used as Inspiration

Some scenarios in the project were inspired by well-known aviation accidents and incidents involving loss of separation or mid-air collision patterns. The goal was not to reconstruct those events precisely. Instead, those cases were used to identify recurring patterns that are meaningful in a didactic simulation:

- converging routes;
- aircraft at the same or incompatible flight levels;
- delayed or incorrect conflict resolution;
- conflicts between instructions or expectations;
- insufficient situational awareness;
- interaction between automation and human decision-making.

Examples include accidents such as Überlingen 2002, Charkhi Dadri 1996, Zagreb 1976, Grand Canyon 1956 and Cerritos 1986. The corresponding project scenarios are simplified educational scenarios and not historical reconstructions.

2.6 Human Factors and Biases

Human factors are relevant in airspace conflicts because pilots and controllers operate under time pressure, incomplete information and high workload. Several cognitive biases may contribute to incorrect decisions:

- **Confirmation bias**: interpreting information in a way that confirms an existing belief.
- **Expectation bias**: perceiving what one expects to happen rather than what is actually happening.
- **Automation bias**: relying too strongly on automated systems or ignoring conflicting evidence.
- **Plan continuation bias**: continuing with an existing plan even after the situation has changed.

- **Startle effect**: degraded performance after an unexpected alert.
- **Loss of situational awareness**: losing an accurate mental model of the operational context.

AeroGuard-MAS does not model pilot psychology directly. However, it addresses some related engineering issues by making rules explicit, decisions traceable and explanations visible.

Chapter 3

Requirements and Scope

3.1 Functional Requirements

The project was designed to satisfy the following functional requirements:

- load airspace scenarios from JSON files;
- represent aircraft, positions, flight levels, velocities, routes and waypoints;
- simulate aircraft movement over discrete ticks;
- detect current conflicts;
- predict future conflicts within a configurable horizon;
- evaluate symbolic safety rules through Prolog;
- compute aircraft priority;
- check maneuver feasibility;
- generate resolution plans;
- apply maneuvers to the simulation state;
- prevent secondary conflicts caused by a maneuver;
- support dynamic weather-zone replanning;
- emit structured JSONL events;
- visualize generated events through a separate GUI;
- provide explanations for important decisions.

3.2 Non-Functional Requirements

The most important non-functional requirements were:

- **Testability**: the system must be covered by automated tests.
- **Modularity**: reasoning, planning, simulation and visualization must be separated.
- **Observability**: decisions and state changes must be visible through JSONL logs.
- **Reproducibility**: scenarios must be deterministic and executable from the command line.
- **Explainability**: the system must produce human-readable explanations.
- **Maintainability**: the code must be organized in packages with clear responsibilities.
- **Didactic clarity**: the implementation must make the intelligent-system concepts recognizable.

3.3 Scope Delimitation

The project deliberately excludes several aspects of real aviation systems:

- continuous physics;
- real aerodynamics;
- certified separation standards;
- real-time aircraft data;
- professional ATC procedures;
- real TCAS implementation;
- safety-critical guarantees;
- distributed runtime execution of all agents.

These limitations are intentional. The project focuses on demonstrating intelligent-system engineering, not on building an operational aviation product.

3.4 Main Scenarios

The project includes five core demonstration scenarios:

1. **Simple Conflict**: two aircraft converge toward a loss of separation.
2. **Emergency Priority**: one aircraft has higher priority and the other should be maneuvered.
3. **Secondary Conflict**: a naive solution would create another conflict.
4. **Weather Replanning**: a weather zone becomes active and blocks a route.
5. **No Conflict Baseline**: aircraft remain safely separated and no false positive should be generated.

Additional stress scenarios are used to evaluate the limits of the system. For example, a many-aircraft scenario intentionally exceeds the current global planner capabilities and is useful for discussing future work.

Chapter 4

Architecture

4.1 Overall Architecture

AeroGuard-MAS follows a layered architecture. Each layer has a clear responsibility and communicates through explicit domain objects or structured events.

The main execution flow is:

`JSON scenario` → `Scenario loader` → `Simulation state` → `Conflict detection` → `Reasoning` → `Planning` → `Maneuver application` → `JSONL events` → `GUI replay`

4.2 Package Organization

The Kotlin core is organized into the following main packages:

- `domain`: immutable domain objects;
- `simulation`: movement, conflict detection and managed simulation;
- `reasoning`: Prolog reasoning interface and implementation;
- `planning`: STRIPS planning and resolution planning;
- `replanning`: weather-zone replanning;
- `events`: JSONL event model and event sinks;
- `explanation`: deterministic explanation generation;
- `integration`: JSON scenario loading and Jason source validation;
- `cli`: command-line entry point.

The Python GUI is located in a separate `gui` directory and does not contain decision logic.

4.3 Domain Layer

The domain layer defines the vocabulary of the system. It includes aircraft, positions, velocities, routes, waypoints, conflicts, maneuvers, resolution plans, weather zones and simulation states.

Most domain objects are immutable. This design choice makes the simulation easier to test and reason about. Each tick produces a new state rather than mutating the previous one.

4.4 Simulation Layer

The simulation layer is responsible for advancing the world. It contains components such as:

- `AircraftMover`, which moves aircraft toward waypoints;
- `ConflictDetector`, which detects current and predicted conflicts;
- `SimulationEngine`, which performs baseline simulation;
- `ManagedSimulationEngine`, which integrates planning and maneuver application;
- `ManeuverApplier`, which physically applies decisions to the state.

The managed engine is the main orchestration component. It detects conflicts, asks the planner for a resolution, applies scheduled maneuvers and handles weather replanning.

4.5 Reasoning Layer

The reasoning layer is represented by the `SafetyReasoner` interface. This interface hides the details of tuProlog from the simulation and planning layers.

This is an important architectural decision. Kotlin components do not need to know how Prolog theories are loaded, how facts are generated or how queries are solved. They simply ask domain-level questions such as whether a maneuver is allowed or what priority an aircraft has.

4.6 Planning Layer

The planning layer implements a small STRIPS-style model with propositions, actions, preconditions and effects. It is intentionally simple but real: it searches for a plan that transforms an initial symbolic state into a goal state.

The system includes a secondary-conflict-aware planner. This planner does not accept a maneuver only because it solves the original conflict. It also simulates the consequences of the maneuver and rejects it if it creates a new conflict.

4.7 Event Layer

The event layer is responsible for observability. Instead of only printing text to the console, the system emits structured JSONL events.

Examples of event types include:

- `aircraft_state`;
- `route_snapshot`;
- `conflict_detected`;
- `plan_generated`;
- `maneuver_selected`;
- `belief_update`;
- `explanation`;
- `weather_zone_activated`;
- `replanning_triggered`.

This event stream is the interface between the intelligent core and the GUI.

4.8 GUI Architecture

The GUI is implemented with Python and Streamlit. It reads JSONL event files, validates them and provides a replay visualization.

The GUI is intentionally passive. It does not resolve conflicts, select maneuvers or run Prolog queries. This separation keeps the intelligent logic in the core and makes the GUI a transparent observer of the system behaviour.

4.9 Design Rationale

The main design choices were:

- use Kotlin for a typed and maintainable core;

- use Prolog for declarative rules;
- use Jason files to explicitly represent BDI agent concepts;
- use STRIPS to demonstrate automated planning;
- use JSON and JSONL for reproducible input and observable output;
- use Streamlit for an easy-to-run demonstration GUI;
- use automated tests and CI/CD to keep the project reliable.

The resulting architecture is not overly complex, but it is sufficiently modular to show the role of each intelligent-system technique.

Chapter 5

Mapping to Intelligent Systems Engineering Topics

5.1 Overview

This chapter explicitly summarizes how the main topics of the Intelligent Systems Engineering course are represented in AeroGuard-MAS. The three most important course-related technologies used in the project are AgentSpeak, Prolog and STRIPS. Each of them plays a distinct role in the system.

5.2 AgentSpeak and BDI Programming

5.2.1 Role in the Project

AgentSpeak is used to model the multi-agent structure of the system. The project includes real Jason/AgentSpeak(L) source files for the following conceptual agents:

- `aircraft_agent`;
- `sector_controller`;
- `conflict_detector`;
- `resolution_planner`;
- `explanation_agent`.

These agents represent responsibilities rather than physical deployment units. The goal was to make the BDI organization explicit: aircraft report state, the sector controller coordinates, the conflict detector identifies dangerous situations, the resolution planner handles corrective plans and the explanation agent is responsible for justifying decisions.

5.2.2 Beliefs, Goals and Intentions

The BDI model is visible through:

- **beliefs**: facts the agent assumes, such as its role or sector;
- **goals**: desired states or tasks, such as scanning the sector or resolving a conflict;
- **intentions**: activated plans in response to events.

A simplified AgentSpeak example is:

```
agent_role(sector_controller).
sector(sector_alpha).

!coordinate_sector.
!request_conflict_scan.
```

Another conceptual example of delegation is:

```
+conflict_detected(A1, A2, Tick)
  <- .send(resolution_planner,
           achieve,
           resolve_conflict(A1, A2, Tick)).
```

This expresses the idea that the sector controller does not solve every task directly. It delegates the resolution goal to the planner agent.

5.2.3 Why AgentSpeak Is Useful Here

AgentSpeak is useful because it provides a natural language for modelling autonomous entities in terms of beliefs, goals and plans. Even if the current integration is intentionally lightweight, the AgentSpeak files make the multi-agent structure concrete and inspectable.

The project also includes smoke tests that verify that the required agent files exist and contain BDI-oriented concepts and message passing. This choice avoids a fragile runtime integration while still making the agent layer real.

5.3 Prolog and Symbolic Reasoning

5.3.1 Role in the Project

Prolog is used as the symbolic reasoning layer. The Kotlin core delegates safety and priority questions to the `SafetyReasoner`, implemented with tuProlog.

The reasoner answers questions such as:

- Is this aircraft pair unsafe?
- Is this maneuver allowed?
- What is the priority score of this aircraft?
- Which explanation facts support a decision?

5.3.2 Facts

At runtime, Kotlin translates the simulation state into Prolog facts. A conceptual example is:

```
aircraft(aza123).
altitude(aza123, 30000).
priority(aza123, normal).
emergency(aza123, false).

horizontal_distance(aza123, dlh456, 3.2).
vertical_distance(aza123, dlh456, 0).

min_horizontal_separation(5.0).
min_vertical_separation(1000).
```

These facts represent the current world from the reasoner's point of view.

5.3.3 Rules

Rules encode declarative knowledge. A simplified unsafe-pair rule is:

```
unsafe_pair(A1, A2) :-
    horizontal_distance(A1, A2, H),
    vertical_distance(A1, A2, V),
    min_horizontal_separation(MH),
    min_vertical_separation(MV),
    H < MH,
    V < MV.
```

A simplified priority rule is:

```
priority_score(Aircraft, 100) :-
    emergency(Aircraft, true).

priority_score(Aircraft, 80) :-
    low_fuel(Aircraft, true).

```

```
priority_score(Aircraft, 10) :-
    priority(Aircraft, normal).
```

A simplified maneuver rule is:

```
maneuver_allowed(Aircraft, climb(TargetAltitude)) :-
    altitude(Aircraft, CurrentAltitude),
    valid_altitude_change(CurrentAltitude, TargetAltitude),
    TargetAltitude =< 40000.
```

5.3.4 Why Prolog Is Useful Here

Prolog makes safety knowledge explicit. Instead of hiding all decisions inside procedural Kotlin code, the project stores part of the reasoning in a declarative form. This improves inspectability and supports explanation. It also reflects a central idea of Intelligent Systems Engineering: intelligent behaviour can emerge from the combination of state representation, declarative knowledge and query solving.

5.4 STRIPS and Automated Planning

5.4.1 Role in the Project

STRIPS is used to model automated planning for conflict resolution. Once a conflict is detected, the system must not only say that a problem exists. It must generate an action that can move the system toward a safer state.

The planning layer uses:

- propositions;
- actions;
- preconditions;
- add effects;
- delete effects;
- initial state;
- goal state.

5.4.2 Action Model

A simplified Kotlin representation of an action is:

```
data class StripsAction(
    val name: String,
    val preconditions: Set<Proposition>,
    val addEffects: Set<Proposition>,
    val deleteEffects: Set<Proposition>,
    val maneuver: Maneuver?
)
```

An action is applicable only if its preconditions are satisfied. Once applied, it adds and removes propositions from the symbolic planning state.

5.4.3 Planning Problem

A simplified planning problem can be represented as:

```
StripsProblem(
    initialState = setOf(
        conflictActive(conflictId)
    ),
    goal = setOf(
        conflictResolved(conflictId)
    ),
    actions = candidateActions
)
```

The planner searches for a sequence of actions that reaches the goal.

5.4.4 Secondary Conflict Prevention

A key improvement in the project is that planning is not accepted blindly. A maneuver may solve the original conflict but create another conflict with a third aircraft. For example:

`climb(SAS111, 32000)`

may solve a conflict between `SAS111` and `IBE222`, but create a new conflict with another aircraft already flying at 32000 feet.

To address this, the secondary-conflict-aware planner:

1. generates candidate maneuvers;

2. checks each maneuver with the Prolog reasoner;

3. applies the maneuver to a simulated future state;

4. advances the simulation over a prediction horizon;

5. rejects the maneuver if a new conflict appears;

6. selects the first safe alternative.

This is one of the most important intelligent behaviours in the system because it combines reasoning, planning and simulation.

5.5 Integration of the Three Topics

AgentSpeak, Prolog and STRIPS are not isolated demonstrations. They play complementary roles:

Table 5.1: Role of course topics inside AeroGuard-MAS

Topic	**Project role**	**Main contribution**
AgentSpeak	BDI agent modelling	Responsibilities, goals, delegation
Prolog	Symbolic reasoning	Safety, priority, feasibility
STRIPS	Automated planning	Corrective action generation

The overall intelligent loop is:

agents structure the responsibilities, Prolog evaluates the rules, STRIPS proposes actions, and simulation validates the consequences.

Chapter 6

Implementation

6.1 Technology Stack

The project uses a heterogeneous but coherent technology stack:

- **Kotlin**: main implementation language;
- **Gradle**: build automation;
- **Jason**: AgentSpeak(L) BDI agent source files;
- **tuProlog**: symbolic reasoning;
- **JSON**: scenario input;
- **JSONL**: event logging;
- **JUnit 5**: automated tests;
- **Python**: GUI implementation;
- **Streamlit**: web-based replay viewer;
- **pandas and Plotly**: data handling and visualization;
- **GitHub Actions**: CI/CD;
- **MkDocs and Mermaid**: project documentation;
- **Dokka**: Kotlin API documentation.

6.2 Scenario Loading

Scenarios are stored as JSON files. Each scenario defines aircraft, routes, initial positions, altitudes, speeds, priorities, separation thresholds and optional dynamic events.

A simplified example is:

```
{
  "name": "simple_conflict",
  "maxTicks": 12,
  "separation": {
    "horizontal": 5.0,
    "vertical": 1000
  },
  "aircraft": [
    {
      "id": "AZA123",
      "x": 0.0,
      "y": 0.0,
      "altitude": 30000,
      "speed": 1.0,
      "priority": "normal",
      "emergency": false,
      "route": [
        { "name": "W1", "x": 5.0, "y": 5.0 }
      ]
    }
  ]
}
```

The scenario loader converts JSON data into validated domain objects. This keeps input files human-readable while preserving type safety inside the Kotlin core.

6.3 Simulation Engine

The simulation engine advances all aircraft one tick at a time. Aircraft move toward their active waypoint according to their velocity. Once an aircraft reaches a waypoint, it proceeds to the next one.

The conflict detector computes horizontal and vertical distances between aircraft. It detects both current conflicts and predicted conflicts by simulating future ticks over a configurable horizon.

6.4 Managed Simulation Engine

The managed simulation engine connects detection, planning and execution. Its loop can be summarized as:

1. create the initial simulation state;
2. detect current and predicted conflicts;
3. select a conflict for planning;
4. generate a resolution plan;
5. schedule the selected maneuver;
6. advance the simulation;
7. apply scheduled maneuvers;
8. handle weather replanning;
9. record states and conflicts.

This component is important because it ensures that decisions physically affect the future simulation state.

6.5 Maneuver Application

The `ManeuverApplier` is responsible for applying decisions. This separation avoids mixing decision-making with state mutation.

Examples of physical effects are:

- climb changes the aircraft flight level;
- descend changes the aircraft flight level;
- slow down reduces speed;
- reroute replaces the active route with a target waypoint;
- weather avoidance is represented as a symbolic maneuver combined with rerouting.

6.6 Event Logging

The event recorder translates simulation results into JSONL events. A typical event is:

```
{
  "tick": 4,
  "type": "maneuver_selected",
  "aircraft": "DLH456",
  "maneuver": "climb",
  "targetAltitude": 32000,
  "reason": "Increase vertical separation"
}
```

JSONL was chosen because each event is independent and easy to parse. This makes the event stream suitable for tests, debugging and GUI replay.

6.7 Explanation Layer

The explanation layer produces deterministic textual explanations. It does not use a large language model. It uses structured information from the planner, reasoner and simulation to create readable messages.

For example, the system can explain that an aircraft climbed because it was the lower-priority aircraft and the target flight level was considered feasible.

6.8 Graphical User Interface

The GUI is implemented in Python with Streamlit. It loads JSONL files, validates their structure and displays a tick-by-tick replay.

The GUI includes:

- a 2D aircraft map;
- aircraft symbols;
- route trails;
- waypoint visualization;
- weather zones;
- conflict markers;
- altitude profile;

- vertical separation chart;
- BDI trace panels;
- explanation panel;
- raw event table.

The GUI is intentionally separated from the core. It is a viewer and does not contain the intelligence of the system.

Chapter 7

Scenarios and Demonstration

7.1 Core Scenarios

The project includes five main scenarios.

7.1.1 Simple Conflict

The simple conflict scenario contains two aircraft converging toward the same area at the same altitude. The system should detect a predicted loss of separation and generate a corrective maneuver, usually a vertical maneuver such as climb or descend.

This scenario demonstrates the basic pipeline: scenario loading, simulation, prediction, planning, maneuver application and event logging.

7.1.2 Emergency Priority

The emergency priority scenario introduces an aircraft with higher priority. The expected behaviour is that the system preserves the higher-priority aircraft and selects the lower-priority one for maneuvering when possible.

This scenario demonstrates the role of Prolog priority reasoning.

7.1.3 Secondary Conflict

The secondary conflict scenario tests a more subtle situation. A naive maneuver may solve the primary conflict but create another conflict with a third aircraft.

This scenario demonstrates the secondary-conflict-aware planner. The system must reject unsafe alternatives and select a safer maneuver.

7.1.4 Weather Replanning

The weather replanning scenario includes a weather zone that becomes active during the simulation. If the current aircraft route intersects the active zone, the system generates a replanning decision and reroutes the aircraft toward a safe waypoint.

This scenario demonstrates adaptation to dynamic events.

7.1.5 No Conflict Baseline

The no-conflict scenario verifies that the system does not generate false positives when aircraft remain safely separated.

7.2 Stress Scenarios

Additional stress scenarios were created to evaluate system limitations.

One scenario contains a route with many waypoints and a thunderstorm cell to avoid. It stresses route intersection logic and GUI route visualization.

Another scenario contains many aircraft converging toward the same central area. This intentionally exceeds the current global planning capability. The system is able to detect many conflicts, but it does not yet solve a global multi-aircraft optimization problem. This is a useful result because it exposes a clear boundary of the current design.

7.3 Demo Flow

A recommended demonstration flow is:

1. run the simple conflict scenario;
2. show the predicted conflict;
3. show the generated plan;
4. show the maneuver applied in the GUI;
5. run the emergency priority scenario;
6. explain the role of symbolic priority;
7. run the secondary conflict scenario;
8. show that an unsafe maneuver is rejected;
9. run the weather replanning scenario;

10. show the active weather zone and reroute;
11. open the explanation panel;
12. show raw JSONL events to demonstrate observability.

7.4 Command-Line Execution

A typical command is:

```
./gradlew run --args="--scenario scenarios/simple_conflict.json
    --events build/aeroguard/events/simple_conflict_events.jsonl
    --explain"
```

The packaged CLI JAR can be executed with:

```
java -jar build/libs/aeroguard-mas-cli-1.0-SNAPSHOT-all.jar --
   scenario scenarios/simple_conflict.json --explain
```

7.5 GUI Execution

The GUI can be launched with:

```
cd gui
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
streamlit run app.py
```

On Windows:

```
cd gui
python -m venv .venv
.venv\Scripts\activate
pip install -r requirements.txt
streamlit run app.py
```

Chapter 8

Testing and CI/CD

8.1 Testing Strategy

The project was developed with a strong focus on automated testing. The goal was not only to test isolated functions, but also to validate the behaviour of the intelligent pipeline.

The test suite covers:

- domain invariants;
- JSON scenario parsing;
- aircraft movement;
- distance computation;
- current conflict detection;
- future conflict prediction;
- tuProlog reasoning;
- maneuver feasibility;
- priority reasoning;
- STRIPS planning;
- secondary conflict prevention;
- weather replanning;
- event serialization;
- CLI smoke behaviour;

- Jason source validation;
- GUI event validation.

8.2 Domain and Scenario Tests

Domain tests ensure that invalid objects cannot be created silently. For example, aircraft identifiers must not be blank, routes must contain waypoints and separation thresholds must be positive.

Scenario loader tests verify that JSON files are correctly parsed into domain objects and that invalid input is rejected.

8.3 Simulation and Conflict Tests

Simulation tests validate aircraft movement and tick advancement. Conflict detection tests validate both current and predicted conflicts. The prediction tests are especially important because the system must anticipate a conflict before it becomes current.

8.4 Reasoning Tests

Reasoning tests validate that the Prolog layer is actually used and returns meaningful answers. These tests cover priority reasoning, unsafe-pair detection and maneuver feasibility.

8.5 Planning Tests

Planning tests validate STRIPS action application, goal satisfaction and plan generation. Secondary-conflict prevention tests verify that a maneuver is rejected if it creates a new conflict with a third aircraft.

8.6 Event and GUI Validation Tests

Event serialization tests ensure that JSONL output has the expected structure. The GUI validation logic checks that uploaded event files contain required fields and valid values.

This is important because JSONL is the contract between the Kotlin core and the Python GUI.

8.7 Jason Smoke Tests

The project includes smoke tests for Jason AgentSpeak source files. These tests verify that the required files exist and contain BDI concepts and message passing patterns.

This approach is pragmatic. It avoids making the CI pipeline depend on a complex Jason runtime integration while still proving that the project contains real AgentSpeak artifacts.

8.8 Continuous Integration

GitHub Actions is used for CI/CD. The pipeline includes:

- a test job;
- a build job;
- a release job;
- a documentation deployment job.

The test and build jobs run on multiple operating systems: Ubuntu, Windows and macOS. This improves portability and reduces the risk that the project works only on one local machine.

8.9 Release and Documentation

The release job builds the project and can publish the packaged CLI JAR as an artifact. Documentation generation includes:

- MkDocs site generation;
- Mermaid diagrams;
- Dokka Kotlin API documentation;
- deployment to GitHub Pages, when configured.

8.10 Reliability Benefits

The CI/CD setup improves reliability because every push or pull request can validate compilation, tests and build configuration. This is especially useful for a project that integrates many technologies: Kotlin, Prolog, Jason, Python, Gradle, Node-based release tooling and documentation generators.

Chapter 9

Evaluation

9.1 Functional Evaluation

The final system satisfies the main functional goals of the project. It can load scenarios, simulate aircraft, detect conflicts, reason about priorities, generate plans, apply maneuvers and visualize the result.

The simple conflict scenario demonstrates the basic pipeline. The emergency priority scenario demonstrates symbolic priority reasoning. The secondary conflict scenario demonstrates consequence checking. The weather replanning scenario demonstrates dynamic adaptation. The no-conflict scenario validates that safe situations do not automatically produce false positives.

9.2 Engineering Evaluation

From a software engineering point of view, the project has several strengths:

- clear package organization;
- separation between core and GUI;
- typed domain model;
- explicit interfaces around complex integrations;
- structured event logging;
- automated tests;
- CI/CD pipeline;
- documentation with MkDocs and KDocs.

The architecture is intentionally not over-engineered. It is modular enough to support extension, but small enough to be understood and presented in an academic context.

9.3 Intelligence Evaluation

The intelligent behaviour of the system is not based on a single algorithm. It emerges from the cooperation of several components:

- AgentSpeak files model agent responsibilities and BDI concepts;
- Prolog rules evaluate symbolic safety constraints;
- the STRIPS planner generates corrective actions;
- forward simulation checks the consequences of those actions;
- the explanation layer makes decisions understandable.

This combination is appropriate for an Intelligent Systems Engineering project because it shows both representation and action.

9.4 Limitations

The main limitations are:

- the airspace model is two-dimensional plus discrete altitude;
- climb and descend are applied instantaneously;
- aircraft performance constraints are simplified;
- Jason integration is primarily source-level and smoke-test based;
- the GUI is replay-based and not real-time;
- the planner does not yet solve global multi-conflict optimization;
- weather rerouting is simplified;
- the system is not safety-critical.

These limitations do not invalidate the project. They define its didactic scope.

9.5 Stress Test Results

The stress weather scenario demonstrates that the system can handle routes with many waypoints and dynamic weather replanning.

The many-aircraft stress scenario is intentionally more difficult. It creates several simultaneous conflicts. The current planner can detect the conflicts, but it is not yet a global optimizer capable of resolving all aircraft at once. This is an important and honest result: it shows a clear direction for future development.

9.6 Overall Assessment

Overall, AeroGuard-MAS is a complete educational prototype. It integrates multiple intelligent-system techniques in a coherent project and remains executable, testable and explainable. The system is not realistic enough for real aviation use, but it is strong as a demonstration of intelligent system design.

Chapter 10

Declaration on the Use of Artificial Intelligence

10.1 Purpose of This Declaration

This chapter provides a transparent declaration about the use of artificial intelligence tools during the development of AeroGuard-MAS. The use of AI is clearly recognizable in the project workflow and in some supporting material, and it is therefore documented explicitly.

10.2 How AI Was Used

Artificial intelligence tools were used as support during the development process, mainly for documentation, explanation, learning support and auxiliary generation tasks.

Specifically, AI support was used for:

- generating and refining parts of the project documentation;
- drafting and improving comments and KDoc/Python docstrings;
- understanding BDI programming concepts and AgentSpeak patterns more clearly;
- discussing the structure and role of Jason agents;
- helping design and refine JSON and JSONL demo scenarios;
- supporting the construction and improvement of the Python Streamlit GUI;
- organizing the report, presentation and documentation structure;
- reviewing architectural explanations and making them clearer.

10.3 How AI Was Not Used

AI was not used as an autonomous replacement for the engineering work. The project required manual decisions about architecture, implementation, testing, debugging, integration and final validation.

In particular, the following activities remained under my responsibility:

- deciding the project scope;
- selecting the architecture;
- integrating Kotlin, tuProlog, Jason, Gradle and Python;
- adapting generated suggestions to the actual codebase;
- running and interpreting tests;
- debugging integration issues;
- checking the behaviour of scenarios;
- deciding which limitations to accept;
- preparing the final repository for delivery.
- write tuProlog code.
- write Agents code.

10.4 Impact on the Project

The use of AI mainly improved productivity and clarity. It helped transform technical choices into readable documentation and supported the learning process around AgentSpeak and BDI programming. It was also useful for generating alternative scenario ideas and for improving the user interface of the GUI.

However, AI output was not accepted blindly. Generated suggestions were reviewed, modified and tested before being integrated. Several parts of the system required manual correction and adaptation, especially where the actual project structure or library behaviour differed from generic suggestions.

Chapter 11

Conclusion

11.1 Summary

AeroGuard-MAS is a didactic intelligent system for simulated airspace conflict management. It combines a Kotlin simulation core, Jason/AgentSpeak(L) agent models, tuProlog symbolic reasoning, STRIPS-style planning, JSON scenarios, JSONL event logging and a Python Streamlit GUI.

The project demonstrates how intelligent behaviour can be engineered by connecting multiple components with clear responsibilities. Aircraft movement and conflict detection provide the simulated environment. Prolog evaluates symbolic constraints. STRIPS planning generates resolution actions. Jason files model BDI responsibilities. The event layer makes the system observable. The GUI makes the behaviour understandable during a demonstration.

11.2 Main Achievements

The project achieved the following results:

- a working scenario-driven simulation;
- conflict prediction over future ticks;
- symbolic reasoning with Prolog;
- real AgentSpeak source files;
- STRIPS-style automated planning;
- maneuver application to future simulation states;
- secondary conflict prevention;

- weather-zone replanning;
- structured JSONL logging;
- replay visualization through a GUI;
- automated testing;
- CI/CD and documentation generation.

11.3 Lessons Learned

The project highlighted several engineering lessons.

First, planning is only useful if plans are actually applied to the system state. A generated maneuver must produce visible consequences in the simulation.

Second, solving a local conflict may create a secondary conflict. For this reason, consequence checking through forward simulation is necessary even in a simplified model.

Third, explainability should not be added only at the end. The system must produce structured events and explanations while it runs.

Fourth, a GUI should not become the place where intelligent decisions are made. Keeping the GUI as a viewer preserves a clean architecture.

Finally, integrating different paradigms requires interfaces and boundaries. Kotlin, Prolog, Jason and Python have different roles, and the project works because these roles are separated.

11.4 Future Work

Possible future improvements include:

- deeper runtime integration with Jason agents;
- a global multi-conflict planner;
- cost-based planning or A* search;
- smoother climb and descent modelling;
- better route rejoin after weather avoidance;
- richer Prolog explanation predicates;
- larger scenario suites;

- property-based testing;
- hosted GUI deployment;
- Docker packaging;
- more realistic aircraft performance constraints.