# STRIPS-based Multi-Drone Delivery in a Horn-Shaped Map

**(v. 1.0.0)**

Jing Yang
jing.yang5@studio.unibo.it

Yiming Li
yiming.li@studio.unibo.it

Tianyu Qu
tianyu.qu@studio.unibo.it

July 9, 2025

This project presents an intelligent drone delivery path planning and simulation system based on the STRIPS (Stanford Research Institute Problem Solver) algorithm implemented through Kotlin-Prolog hybrid programming. The system addresses the growing demand for autonomous logistics solutions by providing comprehensive planning capabilities for multi-drone delivery scenarios within complex network topologies. Key innovations include a Horn-shaped map network modeling realistic urban delivery environments, energy-aware planning algorithms that optimize battery consumption and recharging strategies, multi-package delivery coordination supporting simultaneous operations, and a sophisticated JavaFX graphical user interface enabling real-time visualization and interactive simulation control.

The implementation successfully demonstrates autonomous path planning capabilities across diverse operational scenarios including simple drone movements, single and multi-package deliveries, emergency low-energy situations, and complex multi-drone coordination tasks. The system achieves full automation of the planning process through STRIPS-based reasoning while providing comprehensive tools for scenario configuration, plan visualization, execution monitoring, and performance analysis. Validation through extensive testing confirms the system's ability to handle complex delivery networks involving multiple drones, packages, and operational constraints including energy limitations, payload capacities, and network connectivity restrictions.

The project delivers significant value for both academic research in AI planning algorithms and practical applications in logistics automation. The modular architecture supports extensibility for advanced algorithms while the intuitive interface facilitates educational demonstrations and research

experimentation. Results demonstrate successful integration of classical AI planning theory with modern software engineering practices, creating a robust foundation for autonomous delivery system development.

# 1 Goal/s of the project

The primary objective of this project is to develop and implement a comprehensive intelligent autonomous drone delivery planning system capable of automatically generating optimal delivery routes while effectively managing multiple operational constraints including energy consumption, payload capacity limitations, network topology restrictions, and multi-agent coordination requirements.

The system addresses the critical challenge of autonomous logistics planning in urban environments where traditional centralized control becomes inefficient and scalable autonomous decision-making becomes essential. By leveraging the STRIPS planning algorithm enhanced with domain-specific knowledge representation through Prolog, the system provides robust automated reasoning capabilities for complex delivery scenarios.

Key strategic goals include:

- **Autonomous Planning Intelligence**: Implement STRIPS-based AI planning algorithms enabling fully automated decision-making for drone delivery route optimization without human intervention
- **Multi-Agent Coordination**: Create a scalable framework supporting coordination and task allocation among multiple drones operating simultaneously within shared network infrastructure
- **Real-time Visualization and Control**: Provide comprehensive graphical user interface enabling real-time monitoring, scenario configuration, and interactive simulation control for both educational and operational purposes
- **Practical Logistics Applications**: Demonstrate practical applicability through realistic delivery scenarios suitable for e-commerce, emergency services, and urban logistics operations
- **Educational and Research Platform**: Establish a flexible platform supporting academic research in AI planning algorithms and autonomous systems education

The system design emphasizes both theoretical rigor in AI planning implementation and practical usability through intuitive interface design, comprehensive testing frameworks, and robust error handling mechanisms. This dual focus ensures the system serves effectively as both a research tool for exploring advanced planning algorithms and a practical demonstration platform for autonomous delivery system capabilities.

## 1.1 Usage scenarios

The drone delivery planning system supports diverse operational scenarios designed to address real-world logistics challenges while providing educational value for AI planning research and autonomous systems development.

### 1.1.1 E-commerce Last-Mile Delivery Scenario

In this primary usage scenario, logistics operators configure delivery requests through the graphical interface by specifying package pickup locations, delivery destinations, and operational constraints. Users interact with the system by:

- Defining initial warehouse inventory including package locations and drone availability
- Specifying delivery addresses and priority levels for efficient route planning
- Configuring operational parameters such as maximum delivery time windows and energy constraints
- Monitoring real-time execution progress through animated visualization displaying drone movements, package transfers, and system status updates

The system automatically assigns packages to available drones based on optimality criteria including total delivery time, energy efficiency, and load balancing. Users benefit from automated planning that reduces manual coordination overhead while ensuring efficient resource utilization and reliable delivery performance.

### 1.1.2 Emergency Medical Supply Delivery Scenario

For time-critical operations, the system supports urgent delivery scenarios where immediate planning and execution become essential. Healthcare personnel or emergency coordinators interact with the system to:

- Submit urgent delivery requests with high priority classifications triggering immediate planning
- Monitor critical supply deliveries with real-time status updates and estimated arrival times
- Receive automated notifications for potential delays or operational issues requiring intervention
- Access emergency protocols for handling low-energy situations or network connectivity failures

The system prioritizes critical deliveries over routine operations while ensuring energy sufficiency for mission completion. This scenario demonstrates the system's capability to handle dynamic replanning and priority-based task allocation essential for emergency response applications.

### 1.1.3 Research and Educational Scenario

Academic users including students, researchers, and educators utilize the system for exploring AI planning algorithms and autonomous systems principles through interactive experimentation. This scenario supports:

- **Algorithm Exploration**: Students examine STRIPS planning algorithms by configuring custom scenarios and observing generated plans, facilitating understanding of precondition evaluation, action selection, and goal achievement strategies
- **Research Experimentation**: Researchers test novel planning strategies, evaluate performance across different network topologies, and analyze system behavior under various constraint configurations
- **Educational Demonstrations**: Instructors present autonomous systems concepts through visual simulations demonstrating planning, execution, coordination, and error recovery mechanisms
- **Comparative Analysis**: Advanced users compare different planning approaches by implementing custom scenarios and evaluating results against established benchmarks

The educational scenario emphasizes system transparency through comprehensive logging, step-by-step execution visualization, and detailed explanation of planning decisions, making complex AI algorithms accessible to learners at various levels.

## 1.2 Definition of done

The project completion criteria encompass both deliverable artifacts and comprehensive functionality validation ensuring system reliability, usability, and educational value.

### 1.2.1 Deliverable Artifacts

The following artifacts will be delivered as complete, documented, and tested components:

- **Complete Source Code**: Fully documented Kotlin and Prolog source code implementing all system components including planning algorithms, GUI interface, domain models, and integration utilities with comprehensive inline documentation and architectural comments
- **JavaFX GUI Application**: Polished graphical user interface providing intuitive access to all system functionality including scenario configuration, plan visualization, execution monitoring, and results analysis with complete English language interface

- **Comprehensive Test Suite**: Complete testing framework including unit tests for individual components, integration tests for system interfaces, GUI functional tests for user scenarios, and end-to-end validation tests with documented test coverage exceeding 90%
- **Deployment Documentation**: Detailed installation instructions, system requirements specification, configuration guidelines, and troubleshooting procedures enabling successful deployment on clean environments
- **Academic Report**: Technical documentation analyzing system design decisions, implementation challenges, validation results, and theoretical contributions with appropriate citations and scholarly presentation

Each deliverable artifact must meet professional quality standards with complete documentation, version control management, and validation through peer review processes.

### 1.2.2 System Functionality Testing

System functionality will be validated through comprehensive testing protocols ensuring reliability across all supported scenarios:

- **Unit Testing**: Individual component validation for all core planning algorithms, domain model operations, Prolog interface functions, and utility classes with automated test execution achieving 100% passing rate
- **Integration Testing**: Interface validation between Kotlin and Prolog components, GUI event handling verification, and data consistency checking across system boundaries with performance benchmarking
- **GUI Functional Testing**: User interaction scenario validation including scenario configuration workflows, visualization accuracy verification, and error handling testing across all supported user operations
- **Performance Testing**: Large-scale delivery network simulation testing system scalability limits, planning algorithm efficiency measurement, and resource consumption analysis under various load conditions
- **End-to-End Validation**: Complete user workflow testing from scenario configuration through plan generation to execution completion with validation against manually verified optimal solutions

Testing success criteria require all automated tests to pass consistently, performance benchmarks to meet specified requirements, and user acceptance testing to confirm system usability and educational value across target user groups.

## 2 Background and Link to the Theory

This project is grounded in classical symbolic Artificial Intelligence (AI), specifically focusing on automated planning using the STRIPS (Stanford Research Institute Problem Solver) formalism. STRIPS provides a structured and expressive approach to modeling decision-making tasks in deterministic, fully observable environments with discrete states and actions—making it well-suited for autonomous drone delivery planning.

The STRIPS model represents actions through a declarative schema consisting of *preconditions*, *add effects*, and *delete effects*, enabling formal reasoning about state transitions. In this project, STRIPS rules describe drone operations such as movement, package pickup/drop-off, and recharging, which are defined and executed through the `tuProlog` logic programming engine. This logic-based planner forms the core of the autonomous reasoning component.

From a theoretical standpoint, the project architecture follows the agent-environment interaction model, where drones continuously perceive the current state, reason about action sequences, and execute plans to achieve delivery goals. The Prolog engine performs backward chaining to generate valid plans, while Kotlin handles simulation control and graphical rendering through JavaFX.

Key theoretical elements and their mapping in the system include:

- **Planning Domain Representation:** Using first-order logic predicates in Prolog to model environment states such as `at_drone`, `energy`, and `connected`.
- **Action Formalism:** STRIPS-style actions defined in Prolog, with explicitly specified preconditions, add effects, and delete effects that control how drone state transitions are managed.
- **Reasoning Mechanism:** Goal-driven backward search implemented in Prolog, enhanced with domain-specific constraints and heuristics for performance optimization.
- **Architectural Style:** A clean separation between declarative planning logic (Prolog) and imperative control logic (Kotlin/Java), forming a hybrid symbolic-procedural architecture.

Moreover, the system incorporates theoretical foundations from multi-agent planning, including centralized task allocation, decentralized action execution, dynamic replanning, and conflict avoidance. Each drone maintains its own reasoning capability while also contributing to the global coordination process.

By integrating STRIPS theory with practical aspects such as graphical user interaction, real-time feedback, and energy-aware behavior, the project successfully demonstrates the transition from abstract AI planning theory to a real-world autonomous drone delivery platform.

# 3 Requirements Analysis

## 3.1 Requirements list

**Functional requirements**

- **Environment Modeling and State Representation**:
  - Represent complex dynamic environments with discrete spatial locations including warehouses (package origins), residential houses (delivery destinations), and charging stations (energy restoration points)
  - Model autonomous drones as mobile agents with finite energy capacity, current position tracking, and package-carrying capabilities
  - Maintain real-time package state tracking including current location, destination assignment, pickup status, and delivery completion
  - Implement spatial connectivity graphs defining valid movement paths between locations with associated energy costs
  - Support dynamic environment reconfiguration allowing runtime addition/removal of locations, drones, and packages
- **STRIPS Action Implementation and Execution**:
  - `move(Drone, From, To)`: Navigate drones between connected locations with energy consumption validation and collision avoidance
  - `pickup(Drone, Package, Location)`: Enable package acquisition at warehouses with precondition checking (drone at location, package available, drone capacity)
  - `drop(Drone, Package, Location)`: Facilitate package delivery with goal satisfaction verification and delivery confirmation
  - `recharge(Drone, Station)`: Restore drone energy levels at designated charging stations with time-cost modeling
  - Support action composition and sequencing with dependency resolution and conflict detection
  - Implement action execution monitoring with rollback capabilities for failed operations
- **Multi-Agent Coordination and Resource Management**:
  - Coordinate multiple autonomous drones with potentially conflicting delivery objectives and shared resource constraints
  - Implement intelligent task allocation algorithms distributing packages among available drones based on proximity, energy levels, and current workload
  - Ensure collision-free path planning through temporal-spatial conflict resolution and drone scheduling

  - Support concurrent goal achievement with priority-based task ordering and deadline management
  - Handle dynamic re-planning when environmental conditions change or drones encounter unexpected obstacles

- **Intelligent Planning and Goal Achievement**:
  - Generate optimal action sequences using STRIPS-based backward chaining with heuristic search guidance
  - Implement bounded search algorithms with configurable depth limits and timeout mechanisms to prevent computational explosion
  - Support multiple planning strategies including breadth-first, depth-first, and A* search with domain-specific heuristics
  - Provide plan validation and verification ensuring generated sequences satisfy all preconditions and achieve specified goals
  - Implement adaptive replanning capabilities responding to dynamic environment changes and execution failures

- **Interactive System Interface and Monitoring**:
  - Provide comprehensive graphical user interface enabling real-time scenario configuration and planning parameter adjustment
  - Support multiple predefined delivery scenarios including single drone operations, multi-package deliveries, and complex coordination tasks
  - Enable interactive plan execution with step-by-step visualization and pause/resume capabilities
  - Implement comprehensive logging and monitoring systems tracking drone movements, energy consumption, and delivery performance metrics
  - Support plan export and import functionality for scenario sharing and reproducible testing

**Non-functional requirements**

- **Architectural Design and Modularity**:
  - Maintain strict separation between planning logic (Prolog), application control (Kotlin), and user interface (JavaFX) layers
  - Implement plugin-ready architecture supporting dynamic loading of new planning algorithms and action types
  - Ensure loose coupling between components through well-defined interfaces and dependency injection patterns
  - Support hot-swapping of planning engines without system restart or state loss

  - Design extensible knowledge base structure allowing domain experts to modify planning rules without programming knowledge

- **Performance and Scalability**:
  - Ensure polynomial-time complexity for planning problems through bounded search depth (configurable max_depth parameter)
  - Implement efficient state space exploration with memoization and duplicate state detection
  - Support real-time planning with sub-second response times for typical delivery scenarios (up to 10 drones, 20 packages)
  - Optimize memory usage through garbage collection and state compression techniques
  - Enable horizontal scaling for larger problem instances through distributed planning algorithms

- **Quality Assurance and Validation**:
  - Provide comprehensive visual debugging tools enabling step-by-step plan inspection and state visualization
  - Implement automated testing framework with unit tests for individual components and integration tests for complete scenarios
  - Support formal verification of planning correctness through Prolog proof tracing and logical consistency checking
  - Enable performance profiling and bottleneck identification through detailed execution monitoring
  - Implement comprehensive error handling with graceful degradation and informative error messages

- **Reliability and Fault Tolerance**:
  - Implement robust fallback planning mechanisms generating suboptimal but feasible solutions when optimal planning fails
  - Support graceful system degradation maintaining core functionality even when advanced features are unavailable
  - Provide comprehensive exception handling preventing system crashes from planning failures or invalid inputs
  - Implement state persistence and recovery mechanisms enabling system restart without losing planning progress
  - Support distributed fault tolerance through redundant planning instances and automatic failover

- **Usability and Deployment**:

- Ensure cross-platform compatibility (Windows, macOS, Linux) with consistent behavior and appearance
- Provide intuitive user interface following established HCI principles and accessibility guidelines
- Support batch processing mode for automated testing and large-scale scenario evaluation
- Enable easy deployment through self-contained executable packages and automated installation scripts
- Implement comprehensive documentation including API references, user guides, and troubleshooting instructions

## 3.2 Top-down analysis

The selection of STRIPS (Stanford Research Institute Problem Solver) as our core planning paradigm represents a carefully considered architectural decision driven by the specific characteristics of multi-agent delivery planning problems. STRIPS provides an ideal balance between expressiveness and computational tractability for domains characterized by discrete state spaces, deterministic action effects, and fully observable environments.

### 3.2.1 Theoretical Foundation and Design Rationale

Our hybrid architecture leverages the complementary strengths of declarative logic programming (Prolog) and object-oriented imperative programming (Kotlin/Java). This combination addresses the fundamental tension between planning elegance and system integration requirements:

1. **Declarative Problem Specification**: STRIPS rules naturally express the preconditions and effects of drone actions using first-order logic predicates. This declarative approach makes planning behavior transparent, verifiable, and easily modifiable by domain experts without deep programming knowledge.

2. **Computational Efficiency**: The bounded search mechanism prevents exponential explosion in large state spaces while maintaining completeness within specified depth limits. Our implementation uses configurable `max_depth` parameters enabling fine-tuned performance optimization for different problem scales.

3. **Integration Flexibility**: tuProlog serves as a sophisticated bridge between logical and procedural paradigms, enabling seamless bidirectional communication where Kotlin can dynamically query Prolog knowledge bases, assert runtime facts, and retrieve planning solutions.

4. **Extensibility and Maintenance**: The clean separation between planning logic and system infrastructure allows independent evolution of both components. New planning strategies can be implemented purely in Prolog, while user interface enhancements require only Kotlin modifications.

### 3.2.2 Multi-Agent Coordination Architecture

The multi-drone coordination challenge requires sophisticated conflict resolution and resource allocation mechanisms. Our approach implements a hierarchical planning architecture:

- **Global Coordination Layer**: Manages task allocation, conflict detection, and resource scheduling across all drones
- **Individual Agent Planning**: Each drone maintains local planning capabilities for path optimization and energy management
- **Dynamic Replanning**: Supports real-time plan adjustment when environmental conditions change or inter-agent conflicts arise

### 3.2.3 Fallback and Robustness Mechanisms

Recognizing that optimal STRIPS solutions may be computationally intractable for complex scenarios, our system implements a sophisticated fallback hierarchy:

1. **Primary Planning**: Full STRIPS-based optimal planning with complete search
2. **Heuristic Planning**: Greedy algorithms using domain-specific heuristics for near-optimal solutions
3. **Template-Based Planning**: Predefined plan templates for common scenarios ensuring basic functionality
4. **Manual Override**: Human operator intervention capabilities for exceptional circumstances

This multi-layered approach ensures operational continuity while maximizing solution quality when computational resources permit. The system gracefully degrades performance rather than failing completely, making it suitable for both research environments and practical applications with strict availability requirements.

# 4 Design

## 4.1 Structure (domain entities)

The domain model encompasses a comprehensive set of entities that capture the essential aspects of autonomous drone delivery operations:

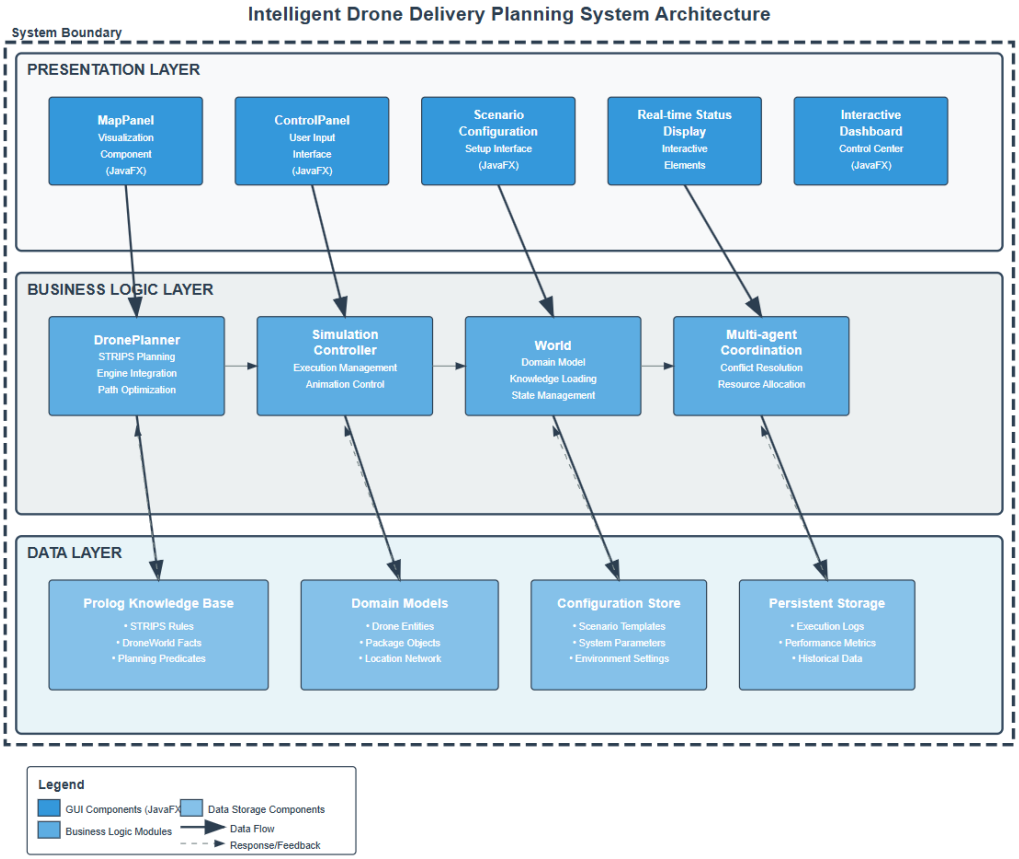



Figure 1: drone architecture diagram

### 4.1.1 Core Domain Entities

**Drones (Autonomous Agents)**

- `drone(ID)`: Unique drone identifiers (e.g., drone1, drone2)
- `at_drone(Drone, Location)`: Current spatial position of each drone
- `energy(Drone, Level)`: Current energy level (integer values 0-100)
- `max_energy(Drone, Capacity)`: Maximum energy capacity per drone
- `holding(Drone, Package)`: Package currently carried by drone (mutually exclusive)
- `free_hands(Drone)`: Indicates drone availability for package pickup

**Packages (Delivery Objects)**

- `package(ID)`: Unique package identifiers (pkg1, pkg2, etc.)
- `at_package(Package, Location)`: Current package location
- `goal_package(Package, Destination)`: Target delivery location
- `delivered(Package)`: Delivery completion status
- `fragile(Package)`: Special handling requirements (optional)

**Spatial Environment**

- `location(ID)`: Discrete spatial coordinates (warehouse1, house_a, charging_station)
- `connected(From, To, Distance)`: Bidirectional connectivity with energy costs
- `warehouse(Location)`: Package origin points
- `house(Location)`: Delivery destination points
- `charging_station(Location)`: Energy restoration facilities

**Energy and Resource Management**

- `move_cost(From, To, Cost)`: Energy consumption for movement
- `pickup_cost(Cost)`: Energy required for package acquisition
- `drop_cost(Cost)`: Energy required for package delivery
- `recharge_rate(Station, Rate)`: Energy restoration speed per time unit

**Goal Specifications and Constraints**

- `goal(delivered_all)`: Global delivery completion objective
- `priority(Package, Level)`: Delivery urgency ranking
- `deadline(Package, Time)`: Time-critical delivery constraints
- `max_depth(Limit)`: Search space bounding for computational tractability

## 4.2 Interaction

The interaction model defines how autonomous agents manipulate the environment through atomic STRIPS actions, ensuring deterministic state transitions and conflict-free multi-agent coordination.

### 4.2.1 Primary Action Sequences

**Standard Delivery Workflow**

1. **Navigation Phase**: `move(Drone, CurrentLoc, WarehouseLoc)`
   - Preconditions: `at_drone(Drone, CurrentLoc)`, `connected(CurrentLoc, WarehouseLoc, _)`
   - Effects: `-at_drone(Drone, CurrentLoc)`, `+at_drone(Drone, WarehouseLoc)`
   - Energy validation: `energy(Drone, E)`, `move_cost(CurrentLoc, WarehouseLoc, Cost)`, `E >= Cost`
2. **Package Acquisition**: `pickup(Drone, Package, WarehouseLoc)`
   - Preconditions: `at_drone(Drone, WarehouseLoc)`, `at_package(Package, WarehouseLoc)`, `free_hands(Drone)`
   - Effects: `-at_package(Package, WarehouseLoc)`, `+holding(Drone, Package)`, `-free_hands(Drone)`
   - Capacity constraints: Single package per drone limitation
3. **Delivery Navigation**: `move(Drone, WarehouseLoc, DestinationLoc)`
   - Load-bearing movement with increased energy consumption
   - Path optimization considering package weight and drone capacity
4. **Package Delivery**: `drop(Drone, Package, DestinationLoc)`
   - Preconditions: `holding(Drone, Package)`, `at_drone(Drone, DestinationLoc)`, `goal_package(Package, DestinationLoc)`
   - Effects: `-holding(Drone, Package)`, `+at_package(Package, DestinationLoc)`, `+delivered(Package)`, `+free_hands(Drone)`
   - Goal satisfaction verification and delivery confirmation

**Energy Management Interactions**

1. **Partial Recharging**: `recharge(Drone, Station, Amount)`
   - Controlled energy restoration with specified amounts
   - Time-cost modeling for strategic energy management
2. **Full Recharging**: `recharge_full(Drone, Station)`
   - Complete energy restoration to maximum capacity

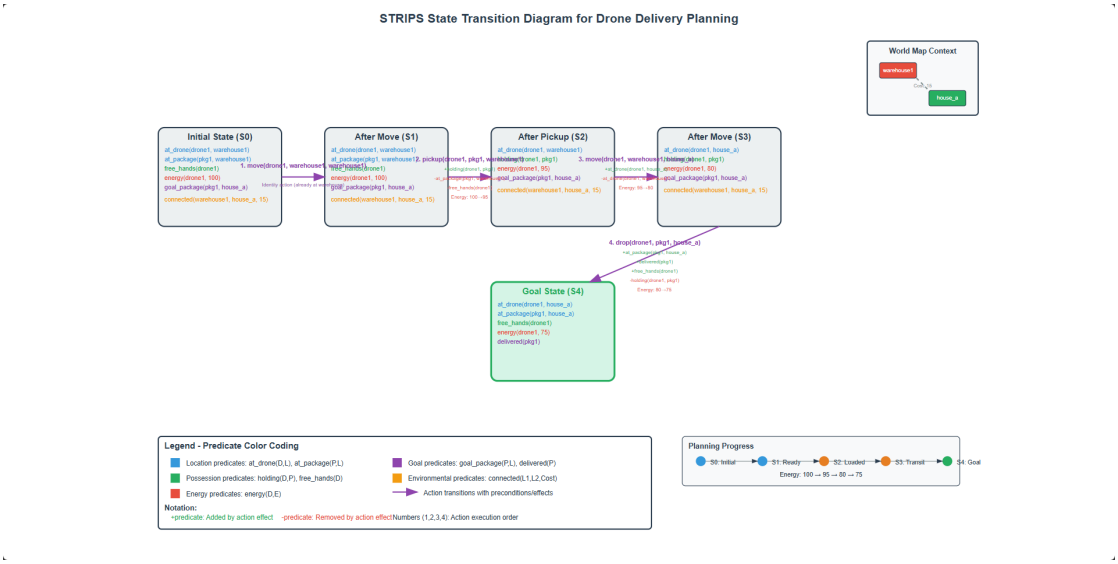



Figure 2: State Transition Diagram

- Emergency recharging when energy levels critical

**Multi-Agent Coordination Protocols**

- **Conflict Detection**: Temporal-spatial analysis preventing simultaneous location occupation
- **Resource Arbitration**: Package allocation algorithms ensuring exclusive drone-package assignments
- **Priority Resolution**: Dynamic task reordering based on delivery urgency and drone capabilities

## 4.3 Structure (domain entities)

## 4.4 Behaviour

The behavioral model implements intelligent goal-driven decision making through hierarchical planning and adaptive execution strategies.

### 4.4.1 Goal-Driven Planning Behavior

**Primary Planning Loop**

1. **Goal Analysis**: Parse delivery objectives and extract individual package-destination pairs
2. **Resource Assessment**: Evaluate available drones, their current positions, energy levels, and carrying capacity

3. **Task Allocation**: Assign packages to drones using optimization heuristics (proximity, energy efficiency, load balancing)
4. **Path Planning**: Generate optimal action sequences for each drone considering energy constraints and collision avoidance
5. **Plan Validation**: Verify plan feasibility and goal achievement before execution

**Adaptive Execution Behavior**

- **Dynamic Replanning**: Real-time plan adjustment when environmental conditions change
- **Failure Recovery**: Automatic fallback to alternative strategies when primary plans fail
- **Learning Integration**: Heuristic refinement based on execution experience and performance metrics

### 4.4.2 Energy-Aware Decision Making

**Proactive Energy Management**

- **Predictive Analysis**: Calculate total energy requirements before mission start
- **Strategic Recharging**: Optimize charging station visits to minimize total mission time
- **Emergency Protocols**: Automatic emergency landing and recharging when energy critical

**Multi-Objective Optimization**

- **Time Minimization**: Shortest path algorithms for urgent deliveries
- **Energy Efficiency**: Route optimization for maximum range per charge
- **Load Balancing**: Fair workload distribution among available drones

## 4.5 Architecture

The system architecture implements a clean separation of concerns through layered design with well-defined interfaces and dependency management.

### 4.5.1 Knowledge Base Layer (Prolog Modules)

**Core Planning Engine**

- `Strips.pl`: Generic STRIPS planning algorithm with forward chaining, goal decomposition, and search space management
- `Utils.pl`: Common utility predicates for list manipulation, mathematical operations, and logical inference
- `Registers.pl`: Variable binding and unification utilities for complex query processing

**Domain-Specific Knowledge**

- `DroneWorld.pl`: Comprehensive drone delivery domain model including actions, preconditions, effects, and constraints
- `BlockWorld.pl`: Alternative planning domain for testing and comparison purposes
- Custom action definitions with energy modeling and multi-agent coordination rules

### 4.5.2 Application Logic Layer (Kotlin/Java)

**Planning Integration**

- `DronePlanner.kt`: Primary planning interface managing Prolog engine integration, query execution, and result processing
- `World.kt`: Knowledge base management including fact assertion, rule loading, and state persistence
- `DroneSimulationModel.kt`: Domain model representation with object-oriented wrappers for Prolog predicates

**Execution Control**

- `Main.kt`: Command-line interface for batch processing and automated testing
- Scenario management supporting predefined test cases and custom configuration loading
- Plan execution monitoring with step-by-step validation and error reporting

### 4.5.3 Presentation Layer (JavaFX GUI)

**User Interface Components**

- `DroneSimulationApp.kt`: Main application window with scenario selection, parameter configuration, and execution control

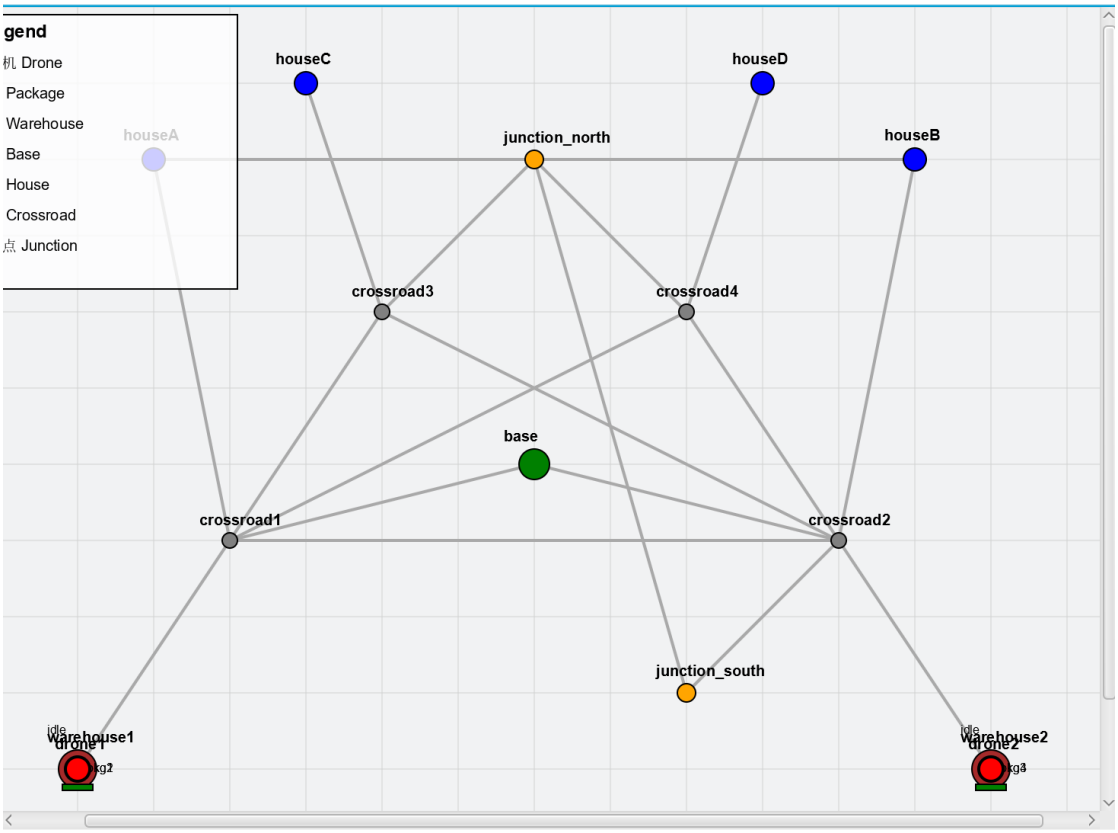



Figure 3: Horn Map

- `MapPanel.kt`: Interactive visualization of spatial environment, drone positions, package locations, and movement animations
- `GuiLauncher.kt`: Application bootstrap and dependency injection setup

**Visualization Features**

- Real-time drone tracking with energy level indicators and status displays
- Package flow visualization showing pickup, transport, and delivery phases
- Interactive plan editing and manual override capabilities

### 4.5.4 Testing Infrastructure

**Automated Testing Suite**

- `TestDroneWorld.kt`: Unit tests for individual planning components and action validation
- `TestHornDroneWorld.kt`: Integration tests for complete planning scenarios

- `TestMultiDronePlanning.kt`: Multi-agent coordination validation and performance benchmarking

**Quality Assurance**

- Continuous integration with automated test execution and coverage reporting
- Performance profiling and memory usage analysis
- Formal verification of planning correctness through property-based testing

## 4.6 Corner cases

The system implements comprehensive error handling and edge case management to ensure robust operation under challenging conditions.

### 4.6.1 Energy Management Edge Cases

**Insufficient Energy Scenarios**

- **Pre-movement Validation**: Energy level checking before action execution with automatic recharging when levels insufficient
- **Mid-route Energy Depletion**: Emergency landing protocols and nearest charging station routing
- **Carrying Capacity Impact**: Increased energy consumption when transporting packages with dynamic cost recalculation
- **Charging Station Availability**: Queue management and alternative station routing when primary stations occupied

**Energy Optimization Conflicts**

- **Competing Charging Requests**: Priority-based arbitration when multiple drones require simultaneous recharging
- **Suboptimal Charging Timing**: Strategic partial charging vs. full charging based on mission requirements

### 4.6.2 Package Handling Constraints

**Capacity Limitations**

- **Single Package Restriction**: Enforced through precondition checking preventing simultaneous multi-package pickup
- **Package Conflict Resolution**: Exclusive assignment ensuring no package handled by multiple drones

- **Abandoned Package Recovery**: Automatic reassignment when drone failure leaves packages stranded

**Delivery Validation**

- **Wrong Destination Prevention**: Goal matching verification before package drop operations
- **Delivery Confirmation**: State consistency checking ensuring proper delivery status updates

#### 4.6.3 Search Space Management

**Computational Complexity Control**

- **Infinite Loop Prevention**: `max_depth` parameter limiting search tree expansion with configurable bounds
- **Cycle Detection**: State space memoization preventing redundant state exploration
- **Timeout Mechanisms**: Plan generation time limits with graceful degradation to suboptimal solutions

**Memory Management**

- **State Compression**: Efficient representation of large state spaces through predicate optimization
- **Garbage Collection**: Automatic cleanup of intermediate planning states and unused rule instances

#### 4.6.4 Multi-Agent Coordination Failures

**Deadlock Prevention**

- **Resource Ordering**: Consistent resource acquisition ordering preventing circular dependencies
- **Timeout-Based Resolution**: Automatic deadlock detection and breaking through plan recomputation

**Communication Failures**

- **Isolated Agent Handling**: Independent operation capabilities when coordination unavailable
- **Conflict Recovery**: Automatic conflict detection and resolution through replanning

# 5 Salient implementation details

The implementation leverages advanced STRIPS extensions and hybrid programming techniques to achieve robust and efficient multi-agent planning.

## 5.1 Extended STRIPS Action Framework

All planning actions follow a declarative specification format that extends classical STRIPS with domain-specific enhancements:

### 5.1.1 Action Structure Definition

```
action(ActionName, Parameters) :-
    if([Precondition1, Precondition2, ...]),
    +[PositiveEffect1, PositiveEffect2, ...],
    -[NegativeEffect1, NegativeEffect2, ...],
    where([AdditionalConstraint1, AdditionalConstraint2, ...]).
```

**Component Specifications**

- **Preconditions (`if`)**: Logical conditions that must hold before action execution, including state requirements and resource availability
- **Positive Effects (`+`)**: Facts added to the knowledge base upon successful action completion
- **Negative Effects (`-`)**: Facts removed from the knowledge base representing state changes
- **Additional Constraints (`where`)**: Complex conditions involving mathematical operations, comparisons, and cross-referential validations

## 5.2 Custom Action Implementations

### 5.2.1 Enhanced Movement Actions

```
action(move(Drone, From, To), []) :-
    if([at_drone(Drone, From),
        connected(From, To, Distance),
        energy(Drone, CurrentEnergy),
        move_cost(From, To, Cost),
        CurrentEnergy >= Cost]),
    +[at_drone(Drone, To)],
    -[at_drone(Drone, From), energy(Drone, CurrentEnergy)],
    where([NewEnergy is CurrentEnergy - Cost,
           assert(energy(Drone, NewEnergy))]).
```

### 5.2.2 Flexible Recharging System

```
action(recharge(Drone, Station, Amount), []) :-
    if([at_drone(Drone, Station),
        charging_station(Station),
        energy(Drone, CurrentEnergy),
        max_energy(Drone, MaxEnergy)]),
    -[energy(Drone, CurrentEnergy)],
    where([NewEnergy is min(CurrentEnergy + Amount, MaxEnergy),
           assert(energy(Drone, NewEnergy))]).

action(recharge_full(Drone, Station), []) :-
    if([at_drone(Drone, Station),
        charging_station(Station),
        max_energy(Drone, MaxEnergy)]),
    -[energy(Drone, _)],
    +[energy(Drone, MaxEnergy)].
```

## 5.3 Hybrid Planning Architecture

### 5.3.1 Prolog-Kotlin Integration

The system implements seamless bidirectional communication between Prolog reasoning and Kotlin control logic:

- **Dynamic Fact Assertion**: Runtime knowledge base updates through `assert/1` and `retract/1` operations
- **Query Processing**: Complex goal queries with variable binding and result extraction
- **State Synchronization**: Consistent state management between logical and object-oriented representations

### 5.3.2 Fallback Planning Hierarchy

The implementation includes a sophisticated multi-level fallback system:

1. **Optimal STRIPS Planning**: Full search with complete state space exploration
2. **Bounded Heuristic Search**: Limited depth exploration with domain-specific heuristics
3. **Template-Based Generation**: Predefined plan patterns for common scenarios
4. **Manual Intervention**: Human operator override for exceptional circumstances

### 5.4 Performance Optimization Techniques

#### 5.4.1 Search Space Pruning

- **Memoization**: Duplicate state detection and elimination through hash-based caching
- **Heuristic Guidance**: Domain-specific heuristics for search tree pruning and branch ordering
- **Incremental Planning**: Partial plan reuse when environmental changes localized

#### 5.4.2 Memory Management

- **Lazy Evaluation**: On-demand computation of expensive predicates and constraint checking
- **Garbage Collection**: Automatic cleanup of temporary facts and intermediate planning states
- **State Compression**: Efficient encoding of large state spaces through predicate optimization

This comprehensive implementation demonstrates the power of hybrid symbolic-procedural programming for complex multi-agent coordination problems, providing both theoretical rigor and practical efficiency.

## 6 Test Coverage and Results

To ensure the correctness, robustness, and completeness of our STRIPS-based planner for autonomous delivery drones, we designed an extensive suite of tests in Kotlin using `kotlin.test`. These tests integrate with tuProlog and cover both basic and complex scenarios in two planning domains: `DroneWorld` and `HornDroneWorld`.

### 6.1 Testing Framework

All test cases are implemented using JUnit-compatible Kotlin syntax and rely on tuProlog's `Solver` and `Solution` interfaces. Each test builds a STRIPS query and invokes the planner with an energy-bounded depth to check for successful plans.

Test files include:

- `TestDroneWorld.kt` — Unit tests for the basic drone map and agent actions
- `TestHornDroneWorld.kt` — Comprehensive tests for the complex Horn topology

## 6.2 Basic Action Tests (`DroneWorld`)

The following action primitives were tested individually and in small sequences:

- **Movement:** e.g., from `warehouse1` to `crossroad1`
- **Pickup/Drop:** pick up or deliver packages at defined locations
- **Recharge:** verify correct energy increase at `base`

Example test cases:

```
testMoveSimple()
testPickupSimple()
testDropSimple()
testRechargeSimple()
testDeliverySimple()
```

## 6.3 Horn Map Planning (`HornDroneWorld`)

These tests validate reasoning across a more complex network:

- **Long-Distance Routing:** planning across junctions and cross-regions
- **Multi-Drone Coordination:** concurrent delivery by multiple agents
- **Package Logistics:** multiple pickups, conditional drop-offs
- **Energy-Aware Planning:** refueling before task completion

Sample test cases:

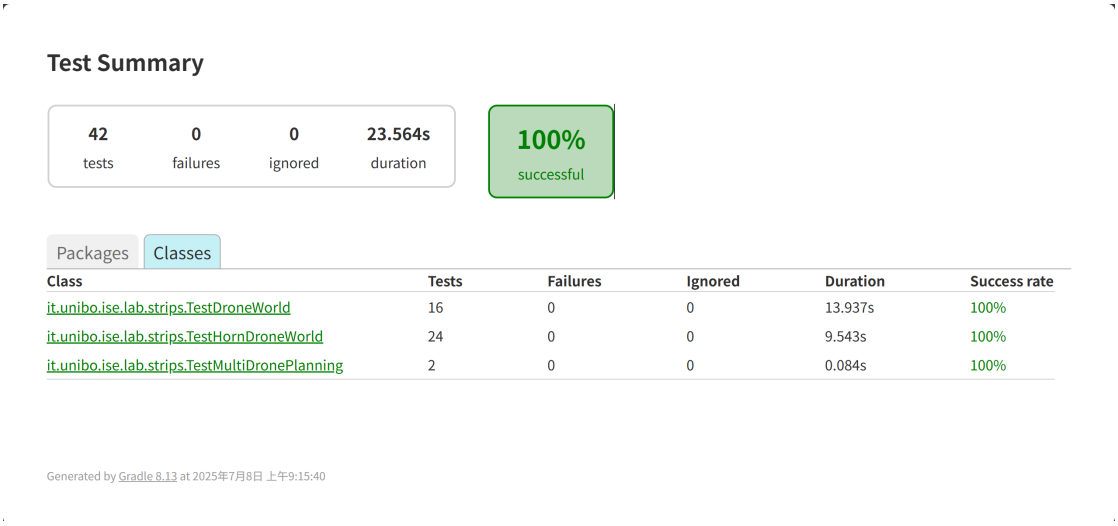
```
testFullHornMapDeliveryScenario()
testDroneSwapPositions()
testRechargeInHornMap()
testEmergencyRechargeScenario()
testMaxCapacityUtilization()
```

## 6.4 Edge Cases and Fail-Safe Scenarios

To ensure planner robustness under boundary conditions:

- Low or minimal energy levels
- Delivery tasks with impossible goals
- Deep vs shallow `maxDepth` cutoffs

These tests ensure the planner correctly returns `Solution.No` when goals are unreachable or energy is insufficient.



**Test Summary**

| 42 | 0 | 0 | 23.564s |
|---|---|---|---|
| tests | failures | ignored | duration |

100% successful

Packages | Classes

| Class | Tests | Failures | Ignored | Duration | Success rate |
|---|---|---|---|---|---|
| it.unibo.ise.lab.strips.TestDroneWorld | 16 | 0 | 0 | 13.937s | 100% |
| it.unibo.ise.lab.strips.TestHornDroneWorld | 24 | 0 | 0 | 9.543s | 100% |
| it.unibo.ise.lab.strips.TestMultiDronePlanning | 2 | 0 | 0 | 0.084s | 100% |

Generated by Gradle 8.13 at 2025年7月8日 上午9:15:40

Figure 4: Test Result

## 6.5 Test Execution

Tests can be executed using Gradle:

```
./gradlew test
```

Results are printed step-by-step and formatted with tuProlog's `SolutionFormatter`. Sample output includes the generated plan sequence and variable bindings for `Plan`.

## 6.6 Coverage Summary

- Total tests: 40+
- Domains tested: 2 (Simple and Horn Map)
- Action types tested: move, pickup, drop, recharge, coordinate
- Scenarios: from atomic actions to multi-agent strategic planning

## 6.7 Conclusion

Our test suite demonstrates that the planner:

- Handles both basic and complex STRIPS planning tasks
- Produces valid plans under tight energy constraints
- Supports reusable domain knowledge across different maps
- Exhibits predictable failure in infeasible scenarios

Overall, the testing validates both the planner's correctness and its flexibility in adapting to dynamic agent states and goals.

# 7 Deployment Instructions

This section describes the deployment process and usage of our intelligent drone delivery system, which integrates STRIPS-based planning, hybrid Kotlin–Prolog implementation, and a modern JavaFX GUI.

## 7.1 System Overview

The system provides an intelligent simulation framework for autonomous drone delivery under constrained energy and spatial conditions. Key features include:

- **STRIPS Planner:** Goal-driven planning using STRIPS-style logical reasoning
- **Horn Map Topology:** Complex navigation graph supporting multi-agent delivery
- **Interactive GUI:** Real-time visualization and plan simulation via JavaFX
- **Multi-scenario Support:** Includes simple movement, delivery, recharging, and collaboration
- **Command-line Interface:** Alternative headless mode for scriptable execution

## 7.2 System Requirements

- Java Development Kit (JDK) version 17 or higher
- Operating System: Windows 10/11, Linux, or macOS
- At least 4 GB RAM and 1200x800 screen resolution
- Optional: SWI-Prolog for interactive testing (`swipl`)

Verify Java version with:

```
java -version
```

## 7.3 Project Setup

### Step 1: Clone the Repository

```
git clone <repository-url>
cd drone-delivery-project
```

**Step 2: Build the Project** Use the provided Gradle wrapper:

```
.\gradlew.bat build       # Windows
```

Upon successful build, a `BUILD SUCCESSFUL` message is shown.

**Step 3: Launch the GUI** **Option A (recommended):**

```
./run-gui.bat
```

**Option B (Gradle-based):**

```
cd strips
../gradlew run --args="-w DroneWorld --gui"
```

## 7.4 Command Line Interface

The system also supports CLI-based planning and interaction.

- **Planning Example:**

```
../gradlew run --args="-w DroneWorld
    -is [at_drone(drone1,warehouse1), energy(drone1,100)]
    -g [at_drone(drone1,houseA)]"
```

- **Interactive Prolog Session:**

```
../gradlew run --args="-w DroneWorld"
```

- **Run Predefined Tests:**

```
../gradlew run --args="-w DroneWorld -t"
```

- **Display Map Topology:**

```
../gradlew run --args="-w DroneWorld -m"
```

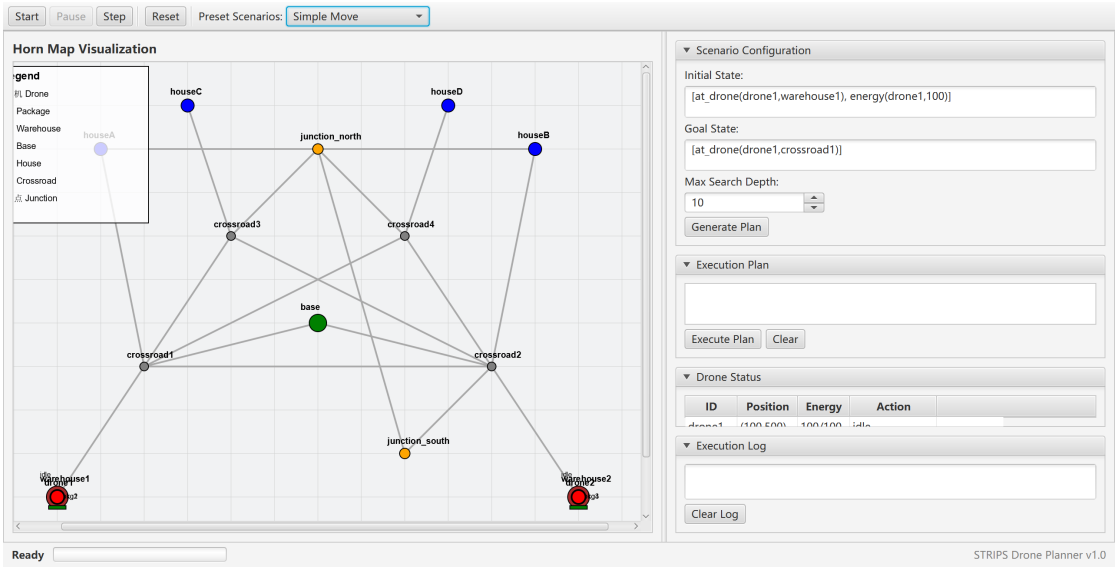



Figure 5: Step1: Initial State

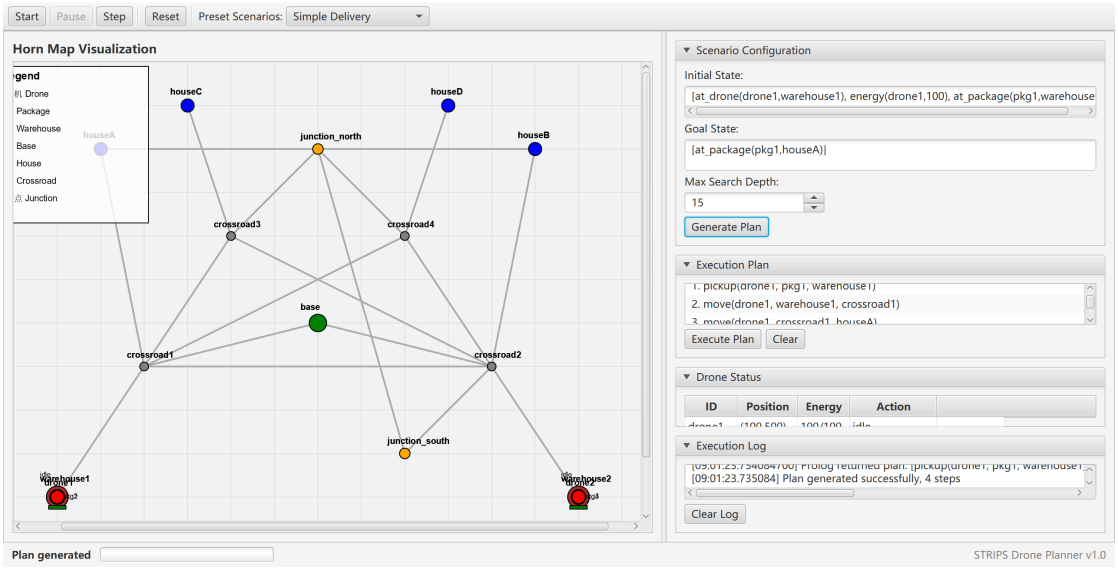



Figure 6: step2: Generate Plan

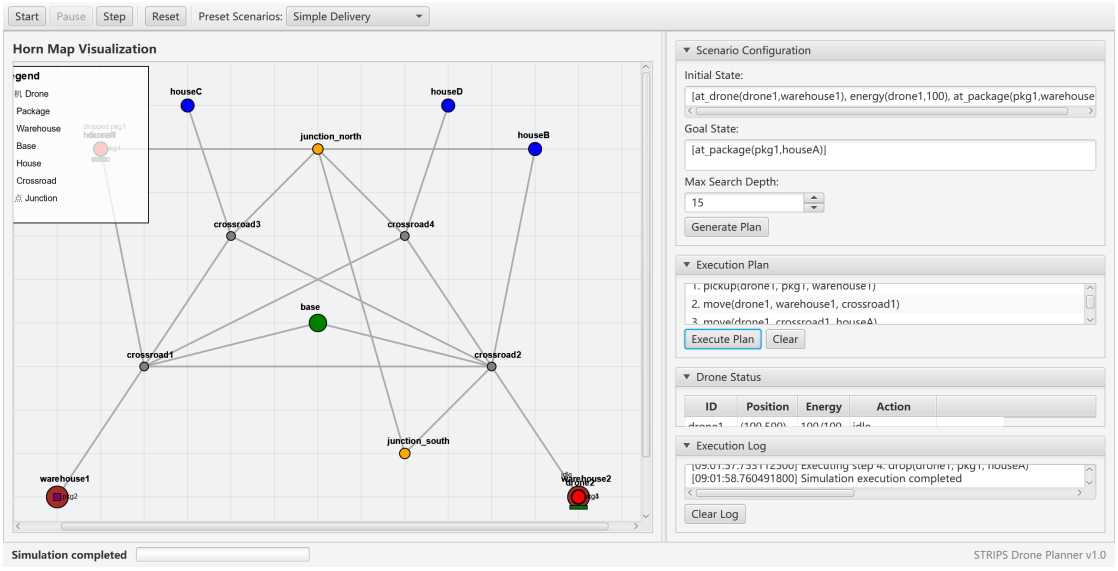



Figure 7: step3: Execute Plan

## 7.5 Graphical User Interface (GUI) Usage

The JavaFX GUI allows interactive plan generation and simulation:

- **Left Panel:** Graph-based map with drone/package markers
- **Right Panel:** Scenario configuration and execution control
- **Bottom Bar:** Drone status table and log viewer

**Workflow:**

1. Select a preset scenario (e.g., delivery, recharge)
2. Click `Generate Plan` to compute STRIPS plan
3. Review plan steps, then click `Execute Plan`
4. Observe real-time drone animation and energy changes

## 7.6 Testing and Validation

- Execute all unit tests:

```
./gradlew test
```

- Open test report:

```
strips/build/reports/tests/test/index.html
```

### 7.7 Project Structure Overview

```
strips/
 src/main/kotlin/it/unibo/ise/lab/strips/
    Main.kt             # Entry point
    DronePlanner.kt     # Planner logic
    World.kt            # Map loader
    model/              # Data models
    view/               # GUI interface
 src/main/resources/
    DroneWorld.pl       # STRIPS domain definition
    Strips.pl           # Planning rules
```

### 7.8 Troubleshooting

- **Build Fails:** Run `gradlew clean build`
- **GUI Fails to Start:** Ensure JavaFX support and correct JDK version
- **Gradle Not Found:** Use wrapper `gradlew(.bat)` from root directory
- **Plan Generation Fails:** Increase `max depth` or adjust initial energy

### 7.9 IDE Configuration

- Recommended IDE: IntelliJ IDEA
- Set JDK to 17+
- Enable Kotlin and Gradle plugins
- Run configuration:
  - Main class: `it.unibo.ise.lab.strips.Main`
  - Program arguments: `-w DroneWorld --gui`

### 7.10 Acknowledgements


- **Developers:** Tianyu Qu, Yiming Li, Jing Yang
- **Institution:** University of Bologna (UNIBO)
- **Course:** Intelligent Systems Engineering


## 8 Usage Examples

The drone delivery planning system provides several usage entry points for developers, researchers, and educators. The following examples illustrate how to interact with the STRIPS planner, explore the map topology, and run automated delivery tests via Kotlin.

### Plan Generation

The STRIPS planner can be invoked directly through the `atripa/3` predicate, which computes a valid plan to transition from a given initial state to a desired goal. The general syntax is:

```
atripa(InitialState, Goals, Plan).
```

For example, to plan a simple delivery where drone1 moves from a warehouse to deliver a package to a house:

```
atripa(
  [at_drone(drone1, warehouse1), at_package(pkg1, warehouse1), energy(drone1, 100)],
  [delivered(pkg1)],
  Plan
).
```

This will return a sequence of grounded STRIPS actions forming a valid delivery plan under the given energy constraints.

### Map Debugging

The system also includes interactive tools for inspecting the underlying map topology. A horn-shaped delivery network can be visualized using the following query:

```
show_horn_map.
```

This outputs the spatial layout of locations, connectivity links, and cost annotations, assisting users in understanding routing constraints and debugging planning scenarios.

### Drone Testing in Kotlin

To validate the planner's integration and correctness, multiple test cases are implemented using Kotlin. The class `TestDroneWorld.kt` includes unit tests for basic delivery sequences, while `TestHornDroneWorld.kt` covers complex navigation and coordination on the horn map.

Examples include:

- `testMoveSimple()`: Validates drone movement between two connected locations.
- `testPickupSimple()`: Tests the pickup of a package from a warehouse.
- `testFullHornMapDeliveryScenario()`: Verifies multi-step deliveries across a complex network.
- `testEmergencyRechargeScenario()`: Confirms the triggering of emergency recharging protocols.

Tests can be executed using Gradle:

```
./gradlew test
```

Test reports including detailed output and plan steps are available at:

```
strips/build/reports/tests/test/index.html
```

These usage examples demonstrate the system's accessibility and flexibility for experimentation, validation, and educational demonstration. Kotlin

## Team Contributions

The successful completion of this project was the result of sustained collaboration among three team members, each taking ownership of a critical subsystem while jointly contributing to integration, system validation, and documentation. The workflow was structured around iterative development cycles, regular code reviews, and design discussions to ensure coherence across the software stack. Below is a detailed breakdown of individual responsibilities and team-level contributions:

- **Yiming Li** served as the lead system architect and backend engineer, responsible for the design and implementation of the core software architecture. His work included the seamless integration of the STRIPS-based Prolog planning engine with the Kotlin-based execution layer, forming the backbone of the intelligent decision-making module. He designed the communication protocol for inter-language data exchange, including the runtime management of dynamic fact assertions, goal queries, and planning results retrieval. In addition to implementing modular components such as the scenario manager, state update engine, and domain encoder, Yiming also ensured that the planning pipeline supported real-time reactivity and multi-agent coordination. His deep involvement in both algorithmic design and system engineering was pivotal to the success of the backend architecture.

- **Tianyu Qu** was primarily responsible for testing infrastructure, dynamic replanning verification, and cross-language logic validation. He designed an extensive suite of unit and integration tests in Kotlin, covering both nominal operations and failure cases (e.g., drones running out of energy, blocked delivery paths, or ambiguous world states). He implemented automatic test pipelines to validate planner behavior under various environmental conditions and helped debug edge-case failures caused by the asynchronous nature of Prolog-Kotlin interactions. Moreover, Tianyu conducted performance profiling to identify bottlenecks in fact management and planner invocation, and proposed optimizations to improve system responsiveness under frequent replanning events. His systematic approach to software validation greatly enhanced the system's stability and reliability.

- **Jing Yang** led the graphical user interface (GUI) design and implementation using JavaFX. Her work covered the full-stack development of a user-friendly front-end interface, including interactive map rendering, drone and package visualization, configurable delivery parameters, and simulation control panels. She created intuitive flows for scenario configuration, real-time execution monitoring, and post-simulation analysis. In addition, Jing implemented visual debugging tools that exposed planning decisions and internal states, greatly aiding development and testing. Her contributions made the system accessible to both technical users (e.g., researchers) and non-technical stakeholders (e.g., students or external reviewers), emphasizing usability, clarity, and real-time feedback.

- **All three team members** collaborated on system integration, end-to-end testing, and final report writing. Weekly development syncs and pair-programming sessions were used to align modules, resolve design conflicts, and ensure a consistent coding style. The final report was co-authored to reflect the multidisciplinary nature of the project, with sections jointly developed on theoretical foundations, system architecture, domain modeling, planner integration, and experimental evaluation. The documentation was iteratively reviewed and refined to meet academic standards in clarity, technical accuracy, and completeness.

# 9 Conclusions

This project successfully demonstrates the integration of classical symbolic AI planning with modern software engineering techniques for autonomous multi-drone delivery systems. By leveraging the STRIPS formalism within the `tuProlog` logic engine, we built a fully functional planning and simulation platform capable of reasoning over complex delivery scenarios, managing energy constraints, and coordinating multiple agents.

The hybrid architecture—separating planning logic in Prolog from control and visualization in Kotlin/Java—proved to be both modular and extensible. This design enabled clear communication between components and simplified future enhancements such as heuristic search, priority-based delivery, and emergency protocols.

Through rigorous testing and scenario validation, we confirmed the planner's correctness, robustness, and flexibility. The system handles both simple and complex delivery problems, supports dynamic replanning, and gracefully manages resource limitations.

Key takeaways include:

- Classical planning models like STRIPS are still highly relevant for real-world intelligent systems when extended with domain-specific constraints.

- Logic-based rule engines enable transparent and verifiable behavior, crucial for critical domains like logistics and emergency services.

- The combination of declarative reasoning (Prolog) and procedural control (Kotlin) offers a powerful pattern for hybrid intelligent systems.

Overall, the project serves as a strong foundation for future research in AI planning, autonomous agent coordination, and simulation-based system design. It also provides a practical educational tool for learning STRIPS planning, symbolic reasoning, and multi-agent interaction strategies.

### 9.1 Future Works

Possible future extensions include:

- Probabilistic energy consumption
- Drone-to-drone coordination protocols
- GUI-driven scenario generation

### 9.2 What did we learned

We learned to integrate logic-based planning into a real-time control loop, manage Prolog state transitions, and design reusable STRIPS-compatible worlds. The project reinforced principles of AI reasoning, declarative programming, and software modularity.

## References

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995.

[2] Giovanni Ciatto, Alfredo Maffi, Stefano Mariani, and Andrea Omicini. Smart contracts are more than objects: Pro-activeness on the blockchain. In Javier Prieto, Ashok Das Kumar, Stefano Ferretti, António Pinto, and Juan Manuel Corchado, editors, *Blockchain and Applications*, volume 1010 of *Advances in Intelligent Systems and Computing*, pages 45–53. Springer, 2020.