Flyby-F11 Implementation Syllabus

Ontology-Constrained Reinforcement Learning for Autonomous UAV Navigation

Finley Holt

2025-12-25

Table of contents

1 Course Overview

Objective: Build a fully autonomous quadcopter using ontology-constrained reinforcement learning by completing weekly exercises that incrementally add capability.

Platform: ASUS ProArt P16 (RTX 5090 24GB, 64GB RAM) for development, deploy to Jetson Orin NX 16GB

Deliverable: Flying drone that accepts mission intent, verifies safety via formal reasoning, and executes autonomously with explainable decisions.

Prerequisites: Basic Python, ROS 2 concepts, willingness to learn Prolog and SUO-KIF

---

2 Module 1: Planning Mode Foundation (Weeks 1-3)

Goal: Prove the two-phase architecture works with formal verification of safety properties.

2.1 Week 1: Ontology Toolchain Setup

2.1.1 Exercise 1.1: Build Vampire Theorem Prover

Objective: Compile and verify Vampire works on your development machine.

```
cd flyby-f11/docs/ontology/repos/vampire
mkdir build && cd build
cmake ..
make -j$(nproc)
```

Deliverable: Run `./bin/vampire --version` and confirm output.

Verification: Test on simple FOL problem:

```
echo "fof(axiom1, axiom, uav(drone1))." > test.tptp
echo "fof(axiom2, axiom, ! [X] : (uav(X) => vehicle(X)))." >> test.tptp
```

```
echo "fof(goal, conjecture, vehicle(drone1))." >> test.tptp
./bin/vampire test.tptp
```

Expected: "Refutation found" (proof successful)

2.1.2 Exercise 1.2: Explore SUMO Structure

Objective: Understand SUMO's organization and identify relevant axioms.

Tasks: 1. Read `docs/ontology/repos/sumo/Merge.kif` (lines 1-500) 2. Find axioms for: `Object`, `Process`, `agent`, `distance` 3. Identify spatial relations: `between`, `orientation`

Deliverable: Document 5 SUMO axioms relevant to UAV navigation in `week1_notes.md`

2.1.3 Exercise 1.3: Install SWI-Prolog

Objective: Set up execution mode runtime environment.

```
sudo apt install swi-prolog
pip install pyswip
```

Verification:

```
from pyswip import Prolog
prolog = Prolog()
prolog.assertz("uav(drone1)")
list(prolog.query("uav(X)"))  # Should return [{'X': 'drone1'}]
```

Deliverable: Screenshot of successful query.

2.2 Week 2: UAV Domain Ontology

2.2.1 Exercise 2.1: Define Flight Phases

Objective: Write your first SUMO ontology extension.

File: `flyby-f11/ontology/planning_mode/uav_domain.kif`

Assignment: Define these concepts in SUO-KIF:

```
;; Subclass hierarchy
(subclass UAV Aircraft)
(subclass Quadcopter UAV)
(subclass FlightPhase Process)
(subclass Takeoff FlightPhase)
(subclass Cruise FlightPhase)
(subclass Landing FlightPhase)

;; Add documentation
(documentation UAV EnglishLanguage "Unmanned Aerial Vehicle...")
```

Deliverable: 50-line `.kif` file with UAV taxonomy.

2.2.2 Exercise 2.2: Battery Safety Axiom

Objective: Write a safety property in first-order logic.

Assignment: Encode "Cannot land if battery below 15%":

```
(=>
  (and
    (instance ?LANDING Landing)
    (agent ?LANDING ?UAV)
    (batteryLevel ?UAV ?LEVEL))
  (greaterThan ?LEVEL 0.15))
```

Deliverable: Add 3 more safety axioms (collision avoidance, geofence, altitude limits).

2.2.3 Exercise 2.3: Prove Safety with Vampire

Objective: Use automated theorem proving to verify properties.

Assignment: 1. Convert your KIF axioms to TPTP format 2. Create a test scenario that violates battery safety 3. Prove Vampire detects the contradiction

Deliverable: `safety_proofs.txt` showing Vampire output for 3 scenarios.

2.3 Week 3: SUMO → Prolog Translation

2.3.1 Exercise 3.1: Manual Translation

Objective: Translate battery axiom to executable Prolog.

Assignment: Convert this SUMO axiom:

```
(=> (and (instance ?L Landing) (batteryLevel ?UAV ?B))
    (greaterThan ?B 0.15))
```

To this Prolog rule:

```
canExecute(landing) :-
    batteryLevel(Battery),
    Battery > 0.15.
```

Deliverable: Translate all 4 safety axioms from Week 2.

2.3.2 Exercise 3.2: Build Translator Tool

Objective: Automate KIF → Prolog conversion for simple patterns.

File: `flyby-f11/scripts/kif_to_prolog.py`

Assignment:

```
def translate_safety_axiom(kif_str):
    """Parse simple => patterns and output Prolog."""
    # Hint: Use regex to extract predicates
```

```
    # Pattern: (=> (and ...) (greaterThan ...))
    pass
```

Deliverable: Script that translates 3+ axiom patterns correctly.

2.3.3 Exercise 3.3: Mission Verification Demo

Objective: Create end-to-end demo of planning mode → execution mode.

File: `flyby-f11/demo/verify_mission.py`

Assignment: Build a demo that: 1. Loads compiled Prolog rules 2. Asserts mission scenario (battery=10%, obstacle at 2m) 3. Queries safety constraints 4. Reports violations with explanations

Deliverable: Working demo that passes/fails 5 test scenarios.

Grading: Pass if demo correctly identifies unsafe conditions.

---

3 Module 2: Execution Mode Integration (Weeks 4-6)

Goal: Connect Prolog reasoning to a simulated quadcopter in real-time.

3.1 Week 4: ArduPilot SITL Setup

3.1.1 Exercise 4.1: Install ArduPilot SITL

Objective: Set up software-in-the-loop simulation.

```
git clone https://github.com/ArduPilot/ardupilot.git
cd ardupilot
git submodule update --init --recursive
./Tools/environment_install/install-prereqs-ubuntu.sh -y
. ~/.profile
```

Verification:

```
cd ArduCopter
sim_vehicle.py -w  # Wipe params and start
```

Deliverable: Screenshot of MAVProxy console showing "APM: EKF2 IMU0 is using GPS"

3.1.2 Exercise 4.2: MAVSDK Connection

Objective: Control SITL from Python using MAVSDK.

Assignment:

```
from mavsdk import System

async def run():
    drone = System()
    await drone.connect(system_address="udp://:14540")
```

```
    # Verify connection
    async for state in drone.core.connection_state():
        if state.is_connected:
            print("Connected!")
            break

    # Arm and takeoff
    await drone.action.arm()
    await drone.action.takeoff()
    await asyncio.sleep(10)
    await drone.action.land()

asyncio.run(run())
```

Deliverable: Script that arms, takes off to 5m, hovers 10s, lands.

3.1.3 Exercise 4.3: Ontology-Constrained Takeoff

Objective: Block unsafe actions using Prolog queries.

Assignment: Modify takeoff script to query Prolog before arm:

```
from pyswip import Prolog

prolog = Prolog()
prolog.consult("ontology/execution_mode/compiled_rules.pl")
prolog.assertz("batteryLevel(0.50)")

# Check if safe to arm
if list(prolog.query("canExecute(takeoff)")):
    await drone.action.arm()
else:
    print("  Takeoff blocked by safety constraint")
```

Deliverable: Demo video showing takeoff blocked when battery=10%.

3.2 Week 5: ROS 2 Grounding Nodes

3.2.1 Exercise 5.1: ROS 2 Development Environment

Objective: Set up ROS 2 Humble workspace.

```
mkdir -p flyby-f11/ros2_ws/src
cd flyby-f11/ros2_ws
colcon build
source install/setup.bash
```

Deliverable: Create simple talker/listener to verify ROS 2 works.

3.2.2 Exercise 5.2: Mock Vision Publisher

Objective: Simulate YOLO detections for testing.

Assignment: Create `mock_vision_node.py`:

```
import rclpy
from vision_msgs.msg import Detection2DArray, Detection2D

class MockVisionNode(Node):
    def __init__(self):
        super().__init__('mock_vision')
        self.pub = self.create_publisher(Detection2DArray, '/detections', 10)
        self.timer = self.create_timer(0.1, self.publish_detection)

    def publish_detection(self):
        msg = Detection2DArray()
        det = Detection2D()
        det.results[0].id = "person"
        det.bbox.center.x = 320
        det.bbox.center.y = 240
        msg.detections.append(det)
        self.pub.publish(msg)
```

Deliverable: Node that publishes fake person detections at 10Hz.

3.2.3 Exercise 5.3: Grounding Node - Vision to Prolog

Objective: Convert bounding boxes to symbolic facts.

File: `ros2_ws/src/perception_grounding/object_grounding_node.py`

Assignment:

```
class ObjectGroundingNode(Node):
    def __init__(self):
        super().__init__('object_grounding')
        self.sub = self.create_subscription(
            Detection2DArray, '/detections', self.callback, 10)
        self.prolog = Prolog()

    def callback(self, msg):
        for det in msg.detections:
            obj_id = f"obj_{det.id}"
            obj_type = det.results[0].id
            # Assert to Prolog
            self.prolog.assertz(f"objectType({obj_id}, {obj_type})")
            self.prolog.assertz(f"inView({obj_id})")
```

Deliverable: Node that asserts 10+ Prolog facts per second from vision.

3.3 Week 6: Closed-Loop Ontology Constraints

3.3.1 Exercise 6.1: Safety Monitor Node

Objective: Continuously check constraints and publish alerts.

Assignment: Create `safety_monitor_node.py`:

```
class SafetyMonitorNode(Node):
    def __init__(self):
        self.prolog = Prolog()
        self.prolog.consult("compiled_rules.pl")
        self.pub = self.create_publisher(SafetyAlert, '/safety_alerts', 10)
        self.timer = self.create_timer(0.1, self.check_safety)  # 10Hz

    def check_safety(self):
        violations = list(self.prolog.query("violatesSafetyConstraint(X)"))
        for v in violations:
            alert = SafetyAlert()
            alert.constraint = v['X']
            alert.severity = "CRITICAL"
            self.pub.publish(alert)
```

Deliverable: Monitor that triggers alerts when object <3m away.

3.3.2 Exercise 6.2: Action Filter

Objective: Block unsafe velocity commands using Prolog.

Assignment: Create `action_filter_node.py` that: 1. Subscribes to `/cmd_vel` (desired velocity) 2. Queries Prolog: `canExecute(moveToward([vx, vy, vz]))` 3. Publishes filtered velocity to `/cmd_vel_safe`

Deliverable: Demo showing forward motion blocked when obstacle ahead.

3.3.3 Exercise 6.3: Integration Test

Objective: Full pipeline: Mock vision → Grounding → Safety → Filter → SITL

Assignment: Launch all nodes together:

```
ros2 launch perception_grounding integration_test.launch.py
```

Test Scenario: 1. Mock vision publishes "person" at 2m distance 2. Grounding node asserts `distance(drone, person, 2.0)` 3. Safety monitor triggers `proximity_alert` 4. Action filter blocks forward motion 5. SITL receives zero velocity

Deliverable: Video showing complete pipeline working.

---

4 Module 3: Real Perception (Weeks 7-9)

Goal: Replace mock data with actual YOLO and depth sensors.

4.1 Week 7: YOLO11 TensorRT Deployment

4.1.1 Exercise 7.1: Export YOLO to TensorRT

Objective: Optimize YOLO11 for RTX 5090.

```
pip install ultralytics
```

```
from ultralytics import YOLO

model = YOLO("yolo11n.pt")
model.export(format="engine", device=0, half=True)  # FP16 TensorRT
```

Deliverable: `yolo11n.engine` file, benchmark FPS on RTX 5090.

4.1.2 Exercise 7.2: YOLO ROS 2 Node

Objective: Real-time object detection in ROS 2.

Assignment: Create `yolo_detection_node.py`:

```
from ultralytics import YOLO
import cv2

class YOLONode(Node):
    def __init__(self):
        super().__init__('yolo_detector')
        self.model = YOLO("yolo11n.engine", task="detect")
        self.sub = self.create_subscription(Image, '/camera', self.callback, 10)
        self.pub = self.create_publisher(Detection2DArray, '/detections', 10)

    def callback(self, msg):
        img = self.bridge.imgmsg_to_cv2(msg)
        results = self.model(img)
        # Convert to Detection2DArray
```

Deliverable: Node achieving >30 FPS on webcam.

4.1.3 Exercise 7.3: Replace Mock with Real YOLO

Objective: Swap mock vision node with YOLO in integration test.

Deliverable: Same integration test from Week 6, now with real person detection.

4.2 Week 8: Depth Sensing (Simulated)

4.2.1 Exercise 8.1: Gazebo Depth Camera

Objective: Add depth sensor to SITL simulation.

Assignment: 1. Create Gazebo world with obstacles 2. Add depth camera plugin to quadcopter model 3. Publish depth images to `/depth/image_raw`

Deliverable: Gazebo simulation showing depth visualization.

4.2.2 Exercise 8.2: Spatial Relation Node

Objective: Compute distances from depth map.

Assignment: Create `spatial_relation_node.py`:

```
def callback(self, depth_msg, detection_msg):
    for det in detection_msg.detections:
        u, v = det.bbox.center.x, det.bbox.center.y
        depth = depth_image[v, u]

        # Convert to 3D position
        x, y, z = deproject_pixel(u, v, depth, camera_intrinsics)
        distance = np.linalg.norm([x, y, z])

        # Assert to Prolog
        self.prolog.assertz(f"distance(drone, {det.id}, {distance})")
```

Deliverable: Node that computes correct distances in simulation.

4.2.3 Exercise 8.3: Collision Avoidance Test

Objective: Drone stops when approaching wall.

Test: Fly toward wall in Gazebo, verify action filter stops motion at 3m.

Deliverable: Video showing autonomous collision avoidance.

4.3 Week 9: Multi-Sensor Fusion

4.3.1 Exercise 9.1: Temporal Reasoning

Objective: Add time-based constraints to Prolog.

Assignment: Implement "loitering" detection:

```
% Track object positions over time
:- dynamic(position_history/3).

updatePosition(Obj, Pos) :-
    get_time(T),
    assertz(position_history(Obj, Pos, T)),
    retractOldPositions(T).

isLoitering(Obj) :-
    findall(Pos, position_history(Obj, Pos, _), Positions),
    length(Positions, N),
    N > 30,  % 30 observations
    variance(Positions) < 1.0.  % Low movement
```

Deliverable: Detect person loitering near drone for >5 seconds.

4.3.2 Exercise 9.2: Ternary Relations

Objective: Implement `between(Drone, Obj1, Obj2)` predicate.

Assignment:

```
between(A, B, C) :-
    position(A, [X1, Y1, Z1]),
    position(B, [X2, Y2, Z2]),
    position(C, [X3, Y3, Z3]),
    % Check if A on line segment BC
    ...
```

Deliverable: Demo showing drone avoiding path between two objects.

4.3.3 Exercise 9.3: Integration: YOLO + Depth + Temporal

Objective: Full perception pipeline with all constraint types.

Test Scenarios: 1. Person approaching → proximity alert 2. Person loitering → loitering alert 3. Drone path blocked by two close objects → reroute

Deliverable: All 3 scenarios working in Gazebo.

---

5 Module 4: Reinforcement Learning (Weeks 10-14)

Goal: Train RL policies that respect ontology constraints.

5.1 Week 10: Gymnasium Environment

5.1.1 Exercise 10.1: UAV Gym Environment

Objective: Create custom Gymnasium environment for quadcopter.

File: `flyby-f11/training/envs/uav_nav_env.py`

Assignment:

```
class UAVNavigationEnv(gym.Env):
    def __init__(self):
        self.observation_space = gym.spaces.Box(
            low=-np.inf, high=np.inf, shape=(12,))  # pos, vel, goal, obstacles
        self.action_space = gym.spaces.Box(
            low=-1, high=1, shape=(4,))  # vx, vy, vz, yaw_rate

    def step(self, action):
        # Query Prolog for valid actions
        if not self.prolog_query(f"canExecute({action})"):
            return obs, -100, True, {}  # Invalid action penalty
```

```
        # Simulate physics
        obs = self.physics_step(action)
        reward = self.compute_reward(obs)
        done = self.check_done(obs)
        return obs, reward, done, {}
```

Deliverable: Environment that blocks unsafe actions via Prolog.

5.1.2 Exercise 10.2: Reward Shaping

Objective: Design reward function encouraging safe behavior.

Assignment:

```
def compute_reward(self, obs):
    reward = 0

    # Progress toward goal
    reward += -distance_to_goal(obs) * 0.1

    # Ontology compliance bonus
    if all_constraints_satisfied(obs):
        reward += 10

    # Penalty for violations
    violations = self.prolog.query("violatesSafetyConstraint(X)")
    reward -= len(list(violations)) * 50

    # Collision penalty
    if collision_detected(obs):
        reward -= 1000

    return reward
```

Deliverable: Reward function definition in docstring.

5.1.3 Exercise 10.3: Test Environment

Objective: Verify environment works with random policy.

```
env = UAVNavigationEnv()
obs = env.reset()
for _ in range(1000):
    action = env.action_space.sample()
    obs, reward, done, _ = env.step(action)
    if done:
        break
```

Deliverable: Plot showing episode rewards vs. constraint violations.

5.2 Week 11: PPO Training

5.2.1 Exercise 11.1: Stable-Baselines3 Setup

Objective: Train first RL policy.

```
from stable_baselines3 import PPO

env = UAVNavigationEnv()
model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=100_000)
model.save("ppo_uav_nav")
```

Deliverable: Trained model achieving >50% success rate.

5.2.2 Exercise 11.2: Evaluate Policy

Objective: Test trained policy in 100 random scenarios.

Assignment:

```
model = PPO.load("ppo_uav_nav")
success_count = 0
for i in range(100):
    obs = env.reset()
    done = False
    while not done:
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)
    if info['reached_goal']:
        success_count += 1

print(f"Success rate: {success_count}%")
```

Deliverable: Report with success rate, collision rate, average reward.

5.2.3 Exercise 11.3: Compare with/without Ontology

Objective: Prove ontology constraints improve safety.

Assignment: Train two models: 1. With Prolog constraints (current) 2. Without constraints (remove canExecute check)

Deliverable: Graph comparing collision rates over 100 episodes.

5.3 Week 12: Hierarchical RL

5.3.1 Exercise 12.1: Mission Planner Agent

Objective: High-level waypoint selection.

Assignment: Create 3-level hierarchy: - Level 1: Mission planner (selects waypoints) - Level 2: Behavior selector (navigate/loiter/land) - Level 3: Trajectory optimizer (velocity commands)

Deliverable: Three separate Gymnasium environments.

5.3.2 Exercise 12.2: Train Hierarchy

Objective: Train all three agents with experience sharing.

Assignment:

```
# Level 3 (low-level) trains first
traj_model = PPO("MlpPolicy", TrajOptEnv(), ...)
traj_model.learn(1_000_000)

# Level 2 uses trained Level 3
behavior_model = PPO("MlpPolicy", BehaviorEnv(traj_model), ...)
behavior_model.learn(500_000)

# Level 1 uses trained Level 2
mission_model = PPO("MlpPolicy", MissionEnv(behavior_model), ...)
mission_model.learn(100_000)
```

Deliverable: Three trained models with shared replay buffer.

5.3.3 Exercise 12.3: Full Mission Test

Objective: Complete multi-waypoint mission autonomously.

Test: 5 waypoints with obstacles between each.

Deliverable: Video showing drone navigating all waypoints safely.

5.4 Week 13: Sim-to-Real Transfer

5.4.1 Exercise 13.1: Domain Randomization

Objective: Generalize to varied conditions.

Assignment: Randomize in training: - Wind speed/direction - Obstacle positions - Lighting conditions - Sensor noise

Deliverable: Retrain with randomization, test in 50 novel scenarios.

5.4.2 Exercise 13.2: Deploy to SITL

Objective: Run trained policy in ArduPilot SITL.

Assignment: Create `rl_controller_node.py`:

```
class RLControllerNode(Node):
    def __init__(self):
        self.model = PPO.load("hierarchical_policy")
```

```
        self.timer = self.create_timer(0.1, self.control_loop)

    async def control_loop(self):
        obs = self.get_observation()  # From telemetry + vision
        action, _ = self.model.predict(obs)

        # Check with Prolog
        if self.prolog.query(f"canExecute({action})"):
            await self.drone.offboard.set_velocity_body(action)
```

Deliverable: RL policy flying in SITL with safety constraints.

5.4.3 Exercise 13.3: Hardware Preparation

Objective: Prepare for real flight.

Assignment: 1. Calibrate sensors (IMU, compass, ESCs) 2. Set up RC failsafe 3. Define geofence in parameters 4. Practice manual flight

Deliverable: Pre-flight checklist document.

5.5 Week 14: Flight Testing

5.5.1 Exercise 14.1: Tethered Hover Test

Objective: First powered test with safety tether.

Test: Hover at 1m for 60 seconds using RL policy.

Deliverable: Flight log showing stable hover with constraint monitoring.

5.5.2 Exercise 14.2: Autonomous Waypoint Mission

Objective: Full autonomous flight with ontology constraints.

Mission: 3 waypoints in safe outdoor area, avoid simulated obstacle.

Deliverable: Flight video with overlay showing: - Active constraints - Safety alerts - RL action selection - Prolog query results

5.5.3 Exercise 14.3: Final Demonstration

Objective: Complete mission with explainability.

Scenario: Inspection mission with no-fly zone and person detection.

Requirements: 1. Accept mission via natural language 2. Verify plan with Vampire 3. Execute with RL + Prolog constraints 4. Generate audit trail of all decisions 5. Emergency stop if constraint violated

Deliverable: 10-minute video presentation showing full system.

---

6 Grading Rubric

Each exercise graded on: - **Functionality** (50%): Does it work as specified? - **Safety** (30%): Does it respect ontology constraints? - **Documentation** (20%): Clear code comments and deliverables?

Final Grade: Average of all exercises (14 weeks × 3 exercises = 42 total)

Passing: 70% or higher **Honors**: 90% or higher + innovation beyond requirements

---

7 Resources

- **Documentation**: `flyby-f11/docs/` (all offline)
- **Office Hours**: GitHub Discussions
- **Reference**: `APPROACH.qmd`, `ONTOLOGY_FOUNDATION.qmd`

Document Version: 1.0 **Last Updated**: 2024-12-25 **Status**: Ready for Week 1