UM6P
University
Mohammed VI
Polytechnic

Symbolic Control for Mobile Robots

Advanced Control Systems and Robotics Project

Project Team

Ghita Bellamine
Salma Ammari
Yasser Baouzil

Supervision

Prof. Adnane Saoud

Table of Contents

I Introduction

I-A Project Overview

This research presents a formal framework for **provably correct control of autonomous mobile robots** using symbolic control methodologies. The project bridges theoretical formal methods with practical robotic applications by developing and implementing a comprehensive control synthesis pipeline for 2D navigation systems.

The core innovation lies in transforming continuous control problems into discrete synthesis tasks through **finite-state abstractions**, enabling automated generation of controllers with mathematical guarantees for:

- **Safety:** Certified obstacle avoidance and constraint satisfaction
- **Liveness:** Guaranteed mission completion under bounded uncertainties
- **Complex Specifications:** Temporal logic missions with sequencing and reactivity

I-B Project Scope and Division

This collaborative research initiative is systematically partitioned into two complementary implementation domains, each addressing distinct challenges in autonomous robotic control:

TABLE I
PROJECT IMPLEMENTATION SCOPE AND TECHNICAL FOCUS

System Domain	Technical Focus	Primary Challenges
2D Mobile Robot	Planar navigation, dynamic obstacle avoidance, temporal logic mission specifications, reachability analysis	State space discretization, transition relation computation, automata-based synthesis, perturbation robustness

II 2D Mobile Robot System Implementation

II-A Symbolic Abstraction Implementation

II-A1 Core Abstraction Class

tomaton

```
class RobotAbstraction2D:
    def __init__(self, state_intervals,
    control_values, perturbation, delta_t):
        self.state_intervals = state_intervals
        self.control_values = control_values
        self.perturbation = perturbation
        self.delta_t = delta_t
        self.state_to_index = {"OutOfGrid": -1}
        self.index_to_intervals = {}
        self.state_edges = []
        self.discrete_x_y = 50
        self.vx_vals = 5
        self.vy_vals = 5
        self._create_state_mapping()
```

Listing 1. Robot Abstraction Class Definition

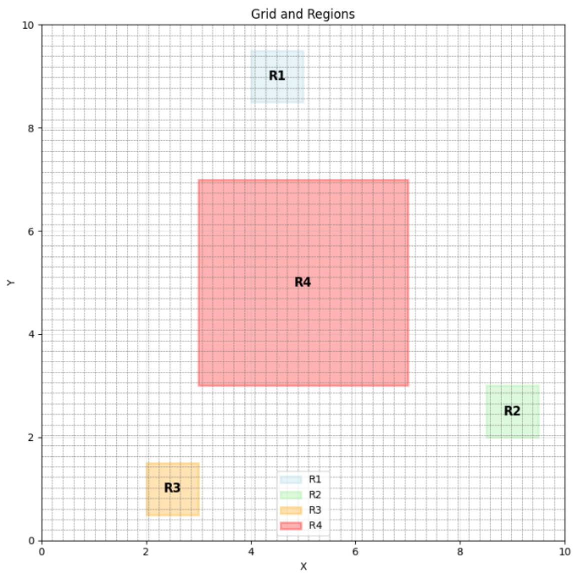
The `RobotAbstraction2D` class serves as the foundation for the symbolic control framework, encapsulating all parameters necessary for the abstraction process. The class takes continuous system parameters including state intervals defining the workspace boundaries, control value ranges specifying available velocities, perturbation bounds representing environmental uncertainties, and the sampling time Δt for discrete-time dynamics.

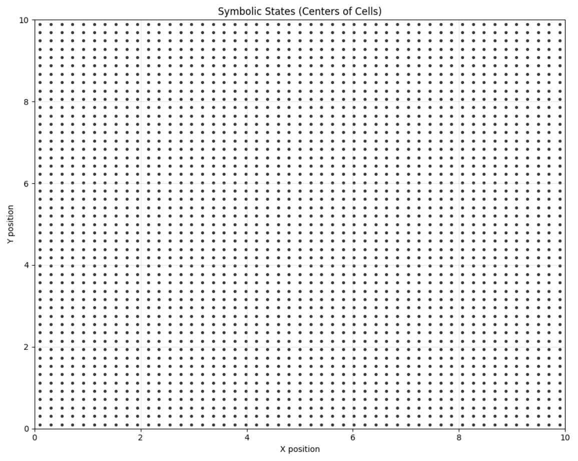
Key attributes include `state_to_index` for mapping discrete coordinates to unique identifiers, `index_to_intervals` for storing continuous regions corresponding to discrete states, and discretization parameters controlling the granularity of both state and control spaces. The initialization automatically calls `_create_state_mapping()` to construct the symbolic representation of the continuous state space.

II-A2 State Space Discretization

```
def _create_state_mapping(self):
        """Symbolic discretization"""
        self.state_edges = [
            np.linspace(interval[0], interval[1],
     self.discrete_x_y) for interval in self.
    state_intervals
        ]

        state_grid = np.array(np.meshgrid(*[edges
    [:-1] for edges in self.state_edges])).T.
    reshape(-1, 2)

        for idx, state in enumerate(state_grid,
    start=1):
            discrete_state = tuple(
                np.digitize(state[i], self.
    state_edges[i]) for i in range(len(state))
            )
            self.state_to_index[discrete_state] =
     idx

            intervals = []
            for i, edge in enumerate(self.
    state_edges):
                low = edge[discrete_state[i] - 1]
                high = edge[discrete_state[i]]
                intervals.append((low, high))
            self.index_to_intervals[idx] =
    intervals
```

Listing 2. State Space Discretization Method





The discretization process creates a uniform grid over the continuous state space $[0, 10] \times [0, 10]$, generating 50×50 discrete cells. Each cell is mapped to a unique symbolic state identifier and stores the corresponding continuous intervals for backward mapping during controller execution.

II-B Controller Synthesis Implementation

II-B1 Core Controller Class

```
class SymbolicControllerSynthesis:
    def __init__(self, transitions, state_space):
        self.transitions = self.
_process_transitions(transitions)
        self.state_space = state_space

    def _process_transitions(self, transitions):
        transition_dict = defaultdict(set)
        for state, control, successors in
transitions:
            transition_dict[(state, control)] =
successors
        return dict(transition_dict)

    def pre(self, R):
        pre_states = set()
        for (state, control), successors in self.
transitions.items():
            if successors.issubset(R):
                pre_states.add(state)
        return pre_states
```

Listing 3. Controller Class Definition

The `SymbolicControllerSynthesis` class provides the algorithmic foundation for computing provably correct controllers from the symbolic abstraction. The class takes the computed transition relation and the complete state space as inputs, then processes the transitions into a dictionary structure for efficient access during fixed-point computations.

II-B2 Safety Synthesis Algorithm

```
def compute_safety_domain(self, Q_s):
    R_k = Q_s.copy()
    iteration = 0
    while True:
        iteration += 1
        R_k_plus_1 = Q_s.intersection(self.pre(
R_k))
        if R_k_plus_1 == R_k:
            break
        R_k = R_k_plus_1
    return R_k
```

Listing 4. Safety Domain Computation Algorithm

This algorithm implements the safety domain computation, which calculates the maximal set of states R^* from which the system can remain indefinitely within the safe set Q_s despite adversarial uncertainties. The algorithm operates through fixed-point iteration, starting with the entire safe set Q_s and progressively removing states that cannot guarantee safety.

II-B3 Reachability Synthesis Algorithm

```
def compute_reachability_domain(self, Q_a):
    R_k = Q_a.copy()
    iteration = 0
    while True:
        iteration += 1
        R_k_plus_1 = Q_a.union(self.pre(R_k))
        if R_k_plus_1 == R_k:
            break
        R_k = R_k_plus_1
    return R_k
```

Listing 5. Reachability Domain Computation Algorithm

This Algorithm implements the reachability domain computation, which calculates the set of all states from which the system can be guaranteed to eventually reach the target set Q_a. Unlike the safety algorithm which uses intersection to restrict the domain, the reachability algorithm uses union to expand the domain progressively.

II-B4 Controller Extraction

```
def synthesize_controller(self, R_star):
    controller = defaultdict(set)
    Q_0 = set()
    for (state, control), successors in self.
transitions.items():
        if state in R_star and successors and
successors.issubset(R_star):
            controller[state].add(control)
    for state in R_star:
        if controller[state]:
            Q_0.add(state)
    return dict(controller), Q_0
```

Listing 6. Controller Synthesis from Computed Domains

The controller synthesis method implements the formal extraction of a provably correct control law from the computed domain R^*. The algorithm constructs a state-dependent control policy where for each state $\xi \in R^*$, it collects all control inputs $\sigma \in \Sigma$ that guarantee the system remains within the safe/reachable domain. The method returns both the synthesized controller mapping and the set of valid initial states Q_0 from which the control strategy is applicable.

III Objectives Implementation

III-A Objective 1: Safety Navigation and Reachability

```
print("\n=== OBJECTIVE 1: Safety (Avoid R4) and
    Reachability (Reach R3) ===")
Q_s = all_states - R4_indices  # Avoid R4
synth1 = SymbolicControllerSynthesis(transitions,
     Q_s)
R_star_1 = synth1.compute_safety_domain(Q_s)
ctrl_1, Q0_1 = synth1.synthesize_controller(
    R_star_1)
if Q0_1:
    x0, _ = select_initial_state_from_Q0_safe(
    Q0_1, robot)
    concretized_1 = {}
    for state, controls in ctrl_1.items():
        intervals = robot.index_to_intervals.get(
    state, "No intervals")
        concretized_1[state] = (intervals,
    controls)
        traj2 =
    generate_safe_trajectory_direct_to_goal(
        x0, np.zeros(2), concretized_1, robot,
        max_steps=200, avoid_regions=[R4],
    goal_region=R3,
        goal_center=np.array([(2+3)/2, (0.5+1.5)
    /2])  # Center of R3
    )
    if traj2:
        plot_trajectory(traj2, regions_dict, "Obj
     1: Safety (Avoid R4) + Reach R3 ")
```

Listing 7. Objective 1: Safety + Reachability

The implementation begins by computing the safe state set Q_s by excluding obstacle region R_4 from all possible states. A symbolic controller is then synthesized that guarantees both safety (permanent avoidance of R_4) and reachability (eventual attainment of R_3). If valid initial states exist in Q_0, the system selects one and executes the concrete trajectory generation. This process bridges the symbolic controller back to continuous execution by mapping discrete states to their continuous intervals and employing a heuristic guidance system that prioritizes goal-directed controls while maintaining safety constraints through continuous collision checking. The `generate_safe_trajectory_direct_to_goal` method executes the synthesized symbolic controller in continuous space while maintaining safety guarantees. It operates through a closed-loop process that continuously:

- **Monitors safety**: Checks for forbidden region violations at each step
- **Tracks progress**: Verifies goal achievement in continuous coordinates
- **Selects controls**: Applies heuristic guidance preferring goal-directed actions
- **Validates transitions**: Ensures each control maintains safety before execution

The method bridges the gap between discrete symbolic guarantees and continuous robotic execution, providing runtime assurance that the trajectory satisfies both safety and reachability objectives.

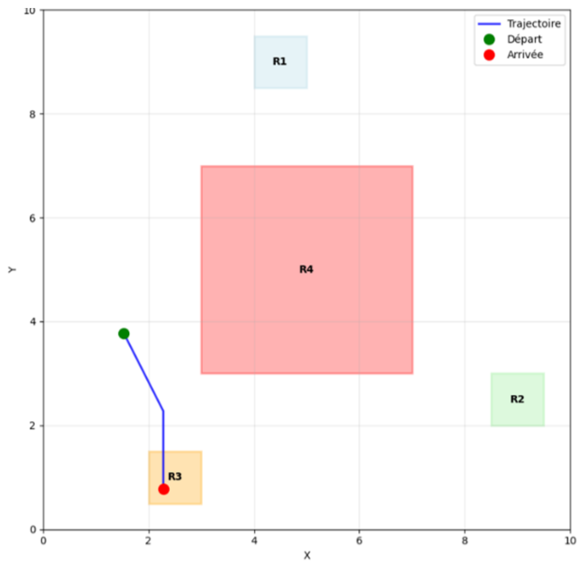


Fig. 1. Objective 1: Safety Navigation and Reachability Results

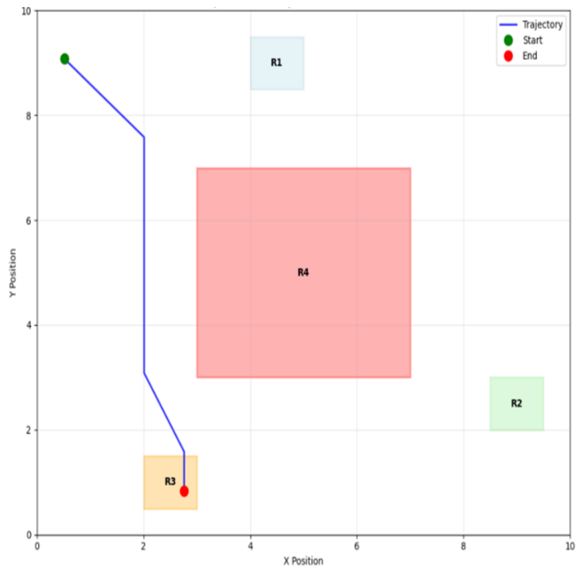


Fig. 2. Objective 1: Trajectory Execution

III-B Objective 2: Temporal Mission with Disjunction

Objective 2 implements the complex temporal specification $\Diamond(R_1 \vee R_2) \wedge \Diamond R_3 \wedge \Box\neg R_4$, which requires the robot to:

- **Visit either R1 or R2 first**: The disjunction $(R_1 \vee R_2)$ allows flexibility in which region is visited initially

- **Then reach R3**: After visiting R1 or R2, the robot must eventually reach R3
- **Always avoid R4**: Safety constraint maintained throughout the mission

This specification requires tracking the *temporal ordering* of region visits, which cannot be expressed using simple safety/reachability controllers. The finite-state automaton serves as a *mission monitor* that tracks progression through the required sequence while enforcing the safety constraints.

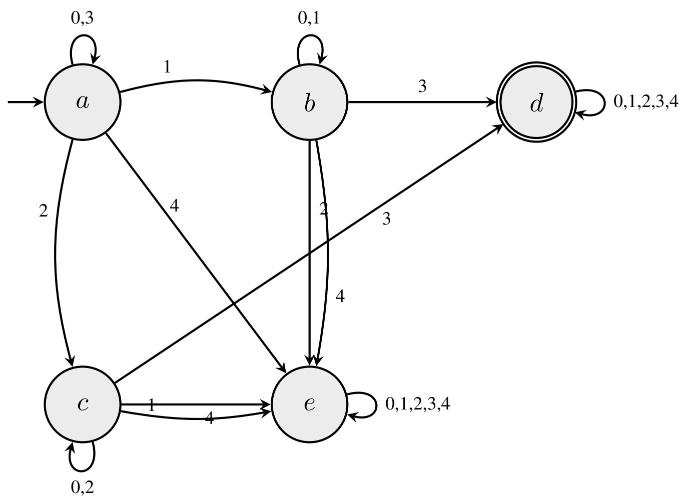


Fig. 3. **Finite-State Automaton for Objective 2**

Automaton State Semantics:

- a **(Initial)**: Waiting to visit either R1 or R2
- b: R1 visited, waiting for R3
- c: R2 visited, waiting for R3
- d **(Accepting)**: Mission completed (R3 reached after R1 or R2)
- e: Mission failed (safety violation or incorrect sequence)

Input Labels:

- **0**: Neutral region (outside R1-R4)
- **1**: Region R1
- **2**: Region R2
- **3**: Region R3
- **4**: Forbidden region R4 (safety violation)

The automaton ensures correct temporal ordering by only allowing transition to the accepting state d after either R1 or R2 has been visited (states b or c), while immediately trapping to state e if R4 is encountered at any point.

Implementation for Objective 2

```
class AutomatonObjective2:
    def __init__(self):
        self.states = {'a', 'b', 'c', 'd', 'e'}
        self.initial = 'a'
        self.accepting = {'d'}
        self.transitions = {
            'a': {'0': 'a', '1': 'b', '2': 'c', '
    3': 'a', '4': 'e'},
            'b': {'0': 'b', '1': 'b', '2': 'e', '
    3': 'd', '4': 'e'},
            'c': {'0': 'c', '1': 'e', '2': 'c', '
    3': 'd', '4': 'e'},
            'e': {'0': 'e', '1': 'e', '2': 'e', '
    3': 'e', '4': 'e'},
            'd': {'0': 'd', '1': 'd', '2': 'd', '
    3': 'd', '4': 'd'}
        }

    def next_state(self, current_state, label):
        return self.transitions[current_state].
    get(label, None)

    def is_accepting(self, state):
        return state in self.accepting
```

III-C Objective 3: Strict Sequential Mission

Objective 3 implements the strict temporal specification $R_1 \to R_2 \to R_3 \to R_4$, which requires the robot to visit all four regions in exact sequential order:

- **Visit R1 first**: Must visit region R1 before any other target region
- **Then R2**: After R1, must visit R2 (cannot skip or reorder)
- **Then R3**: After R2, must visit R3 in sequence
- **Finally R4**: Complete the mission by reaching R4 after R3
- **No deviations**: Any incorrect region visit immediately fails the mission

This specification represents a *strict sequence* constraint where the temporal ordering is fixed and any deviation results in mission failure. The automaton enforces this strict progression through a linear chain of states.

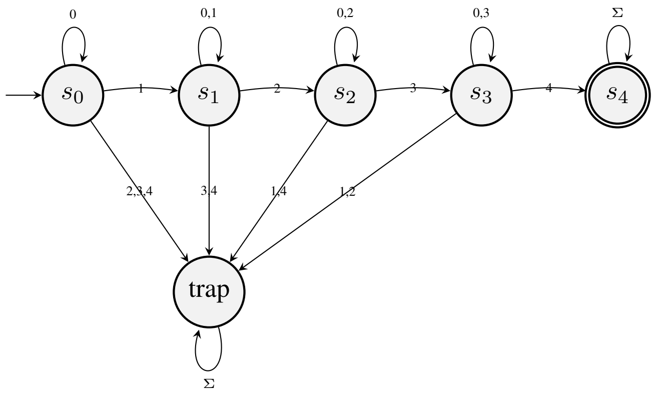


Fig. 4. **Finite-State Automaton for Objective 3**

Automaton State Semantics:

- s_0 **(Initial)**: Waiting to visit R1
- s_1: R1 visited, waiting for R2
- s_2: R2 visited, waiting for R3
- s_3: R3 visited, waiting for R4
- s_4 **(Accepting)**: Mission completed (all regions visited in sequence)
- **trap**: Mission failed (incorrect sequence or premature completion)

Input Labels:

- **0**: Neutral region (outside R1-R4)
- **1**: Region R1
- **2**: Region R2
- **3**: Region R3
- **4**: Region R4

The automaton enforces strict sequential progression by only allowing transitions along the linear chain $s_0 \to s_1 \to s_2 \to s_3 \to s_4$. Any deviation from this exact sequence (e.g., visiting R2 before R1, or R4 before R3) immediately transitions to the trap state, representing mission failure. This provides strong guarantees for applications requiring precise visitation sequences, such as delivery missions or inspection routines.

Implementation for Objective 3

```
class AutomatonObjective3:
    def __init__(self):
        self.states = {'s0', 's1', 's2', 's3', '
    s4', 'trap'}
        self.initial = 's0'
        self.accepting = {'s4'}
        self.transitions = {
            's0': {'0': 's0', '1': 's1', '2': '
    trap', '3': 'trap', '4': 'trap'},
            's1': {'0': 's1', '1': 's1', '2': 's2
    ', '3': 'trap', '4': 'trap'},
            's2': {'0': 's2', '2': 's2', '3': 's3
    ', '1': 'trap', '4': 'trap'},
            's3': {'0': 's3', '3': 's3', '4': 's4
    ', '1': 'trap', '2': 'trap'},
            's4': {'0': 's4', '1': 's4', '2': 's4
    ', '3': 's4', '4': 's4'},
            'trap': {'0': 'trap', '1': 'trap', '2
    ': 'trap', '3': 'trap', '4': 'trap'}
        }

    def next_state(self, current_state, label):
        return self.transitions[current_state].
    get(label, 'trap')

    def is_accepting(self, state):
        return state in self.accepting


def get_region_label(state_idx, R1_indices,
    R2_indices, R3_indices, R4_indices):
    if state_idx in R4_indices:
        return '4'
    elif state_idx in R3_indices:
        return '3'
    elif state_idx in R2_indices:
        return '2'
    elif state_idx in R1_indices:
        return '1'
    else:
        return '0'
```

IV Finite-State Automata Specifications

The `AutomatonBasedControllerSynthesis` class implements the product system synthesis that combines the robot's symbolic abstraction with the temporal logic automaton. This enables the synthesis of controllers for complex temporal specifications.

```
class AutomatonBasedControllerSynthesis:
    def __init__(self, transitions, automaton,
    R1_indices, R2_indices,
                R3_indices, R4_indices, robot):
        self.transitions = transitions
        self.automaton = automaton
        self.R1_indices = R1_indices
        self.R2_indices = R2_indices
        self.R3_indices = R3_indices
        self.R4_indices = R4_indices
        self.robot = robot
        self.transition_dict = self.
    _build_transition_dict(transitions)
        self.state_space = set(state_id for (
    state_id, control, successors)
                                in transitions if
    state_id != -1)
```

Listing 8. Automaton-Based Controller Synthesis Class

Key Components:

- **Product System**: Combines robot states Ξ with automaton states Ψ to form $\Xi \times \Psi$
- **Labeling Function**: Maps symbolic states to region labels (0-4) using `_get_label()`
- **Fixed-Point Computation**: Computes the winning set W^* through iterative predecessor operations
- **Controller Extraction**: Synthesizes safe control policies from the winning set

Synthesis Algorithm:

```
def synthesize_dynamic_controller(self):
        L = self._get_label
        product_states = [(x, q) for x in self.
    state_space for q in self.automaton.states]
        print(f"[Synth] Produit |      | = {len(
    self.state_space)}    {len(self.automaton.
    states)} = {len(product_states)}")

        target = {(x, q) for (x, q) in
    product_states if q in self.automaton.
    accepting}
        W = target.copy()
        iteration = 0
        changed = True

        while changed:
            iteration += 1
            new_pre = set()

            for (x, q) in product_states:
                if (x, q) in W:
                    continue

                for u in {u for (s, u) in self.
    transition_dict.keys() if s == x}:
                    successors = self.
    transition_dict.get((x, u), set())
                    if not successors:
                        continue

                    all_covered = True
                    for x_next in successors:
                        label = L(x_next)
                        q_next = self.automaton.
    next_state(q, label)
                        if q_next is None or (
    x_next, q_next) not in W:
                            all_covered = False
                            break

                    if all_covered:
                        new_pre.add((x, q))
                        break

            if new_pre:
                W |= new_pre
            if iteration % 10 == 0:
                print(f"  Iter {iteration:2d
}: |W| = {len(W)}")
        else:
            changed = False

    print(f"[Synth] Converg  en {iteration}
it rations. |W*| = {len(W)}")

    controller = defaultdict(set)
    Q0 = set()
    for (x, q) in W:
        if q == self.automaton.initial:
            Q0.add(x)
        for u in {u for (s, u) in self.
transition_dict.keys() if s == x}:
            successors = self.transition_dict
.get((x, u), set())
            if not successors:
                continue
            all_covered = True
            for x_next in successors:
                label = L(x_next)
                q_next = self.automaton.
next_state(q, label)
                if q_next is None or (x_next,
 q_next) not in W:
                    all_covered = False
                    break
            if all_covered:
                controller[(x, q)].add(u)

    print(f"[Synth] Contr leur: {len(
controller)} paires, | Q  | = {len(Q0)}")
    return dict(controller), Q0
```

The `synthesize_dynamic_controller()` method implements a fixed-point algorithm that:

1) Initializes the target set with accepting automaton states
2) Iteratively computes predecessors that can maintain progression toward acceptance
3) Extracts a controller guaranteeing the temporal specification
4) Returns valid initial states Q_0 for mission execution

IV-A Objective 2 execution strategy

```
def execute_controller_objective2(self,
   start_sym_state, controller, R3, max_steps
   =300):
    """Execution for Objective 2 (   ( R 1 R2
)      R3          R4   )."""
    if start_sym_state not in {x for (x, q)
in controller.keys() if q == self.automaton.
initial}:
        print(f"  tat  initial {
start_sym_state}      Q  ")
        return None, None

    intervals_start = self.robot.
index_to_intervals[start_sym_state]
    x_cont = np.array([
        (intervals_start[0][0] +
intervals_start[0][1]) / 2,
        (intervals_start[1][0] +
intervals_start[1][1]) / 2
    ])
    x = start_sym_state
    q = self.automaton.initial
    steps = 0

    path = []
    trajectory = [x_cont.copy()]

    r1_center = np.array([(4+5)/2, (8.5+9.5)
/2])
    r2_center = np.array([(8.5+9.5)/2, (2+3)
/2])
    r3_center = np.array([(2+3)/2, (0.5+1.5)
/2])

    while steps < max_steps:
        if (R3[0][0] <= x_cont[0] <= R3[0][1]
 and R3[1][0] <= x_cont[1] <= R3[1][1]):
            if q == 'b' or q == 'c':
                q = 'd'
                print(f" R3 atteint (continu
 en {steps}  tapes ")
                path.append((x_cont.copy(), x
, q, None))
                trajectory.append(x_cont.copy
())
                return path, trajectory

        if q == 'd':
            path.append((x_cont.copy(), x, q,
 None))
            trajectory.append(x_cont.copy())
            return path, trajectory

        available = controller.get((x, q),
set())
        if not available:
            return None, None

        if q == 'a':
            def score(u):
                s = next(iter(self.
transition_dict.get((x, u), {x})))
                c = np.mean(self.robot.
index_to_intervals[s], axis=1)
                l = self._get_label(s)
                if l == '1': return np.linalg
.norm(c - r1_center) - 10
                if l == '2': return np.linalg
.norm(c - r2_center) - 10
                if l == '3': return float('
inf')
                return min(np.linalg.norm(c -
 r1_center), np.linalg.norm(c - r2_center))
            u = min(available, key=score,
default=next(iter(available)))
        else:
            def score(u):
                s = next(iter(self.
transition_dict.get((x, u), {x})))
                c = np.mean(self.robot.
index_to_intervals[s], axis=1)
                return np.linalg.norm(c -
r3_center)
            u = min(available, key=score,
default=next(iter(available)))

        D_x, D_w = self.robot.
compute_Dx_and_Dw(u)
        lower, upper = self.robot.dynamics(
x_cont, u, np.zeros(2), D_x, D_w)
        x_next_cont = (lower + upper) / 2
        x_next_cont = np.clip(x_next_cont,
[0, 0], [10, 10])
        x_next_sym = self.robot._find_state(
x_next_cont)

        if x_next_sym == -1:
            return None, None

        label = self._get_label(x_next_sym)
        q_next = self.automaton.next_state(q,
 label)
```

```
        if q_next is None:
            return None, None

        path.append((x_cont.copy(), x, q, u))
        trajectory.append(x_next_cont.copy())

        x_cont, x, q = x_next_cont,
x_next_sym, q_next
        steps += 1

    return path, trajectory
```

The `execute_controller_objective2()` method implements a heuristic guidance system that:

- Uses region centroids for goal-directed navigation
- Applies different scoring functions based on current automaton state
- Ensures continuous validation of region membership
- Maintains synchronization between symbolic and continuous execution

IV-B Objective 3 execution strategy

```
def execute_controller_objective3(self,
start_sym_state, controller, target_region,
max_steps=500):
    if start_sym_state not in {x for (x, q)
in controller.keys() if q == self.automaton.
initial}:
        print(f"  tat  initial {
start_sym_state}       Q  ")
        return None, None

    intervals_start = self.robot.
index_to_intervals[start_sym_state]
    x_cont = np.array([
        (intervals_start[0][0] +
intervals_start[0][1]) / 2,
        (intervals_start[1][0] +
intervals_start[1][1]) / 2
    ])
    x = start_sym_state
    q = self.automaton.initial
    steps = 0

    path = []
    trajectory = [x_cont.copy()]

    centers = {
        ’1’: np.array([(4+5)/2, (8.5+9.5)/2])
,
        ’2’: np.array([(8.5+9.5)/2, (2+3)/2])
,
        ’3’: np.array([(2+3)/2, (0.5+1.5)/2])
,
        ’4’: np.array([(3+7)/2, (3+7)/2])
    }

    while steps < max_steps:
        if (target_region[0][0] <= x_cont[0]
<= target_region[0][1] and
            target_region[1][0] <= x_cont[1]
<= target_region[1][1]):
            if q == ’s4’:
                print(f" Objectif 3 accompli
en {steps}  tapes  (R4 atteint)")
                path.append((x_cont.copy(), x
, q, None))
                trajectory.append(x_cont.copy
())
                return path, trajectory

        if q == ’s4’:
            print(f" Automaton reached s4 at
step {steps} (symbolic success)")
            path.append((x_cont.copy(), x, q,
 None))
            trajectory.append(x_cont.copy())
            return path, trajectory

        available = controller.get((x, q),
set())
        if not available:
            print(f" No control at (x={x}, q
={q})     dead end")
            return None, None

        next_target_map = {’s0’: ’1’, ’s1’: ’
2’, ’s2’: ’3’, ’s3’: ’4’}
        next_target = next_target_map.get(q,
’4’)

        def score(u):
            succ_set = self.transition_dict.
get((x, u), {x})
            s = next(iter(succ_set))
            c = np.mean(self.robot.
index_to_intervals[s], axis=1)
            label = self._get_label(s)
            dist = np.linalg.norm(c - centers
.get(next_target, centers[’4’]))
            if label == next_target:
                return dist - 10.0
            if label in [’0’, next_target]:
                return dist
            return float(’inf’)

        try:
            u = min(available, key=score)
        except:
            u = next(iter(available))

        D_x, D_w = self.robot.
compute_Dx_and_Dw(u)
        lower, upper = self.robot.dynamics(
x_cont, u, np.zeros(2), D_x, D_w)
        x_next_cont = (lower + upper) / 2
        x_next_cont = np.clip(x_next_cont,
[0, 0], [10, 10])
        x_next_sym = self.robot._find_state(
x_next_cont)

        if x_next_sym == -1:
            print(" Out of grid")
            return None, None

        label = self._get_label(x_next_sym)
        q_next = self.automaton.next_state(q,
 label)
        if q_next == ’trap’:
            print(f" Trapped (label={label}
in state {q}) at step {steps}")
            return None, None

        path.append((x_cont.copy(), x, q, u))
        trajectory.append(x_next_cont.copy())

        x_cont, x, q = x_next_cont,
x_next_sym, q_next
        steps += 1

    print(" Max steps reached without
completion")
    return path, trajectory
```

The `execute_controller_objective3()` method implements a strict sequential guidance system for the mission $R_1 \to R_2 \to R_3 \to R_4$ that:

- **Dynamically targets next region**: Uses a mapping from automaton states to target regions ($s_0 \to R_1$, $s_1 \to R_2$, etc.)

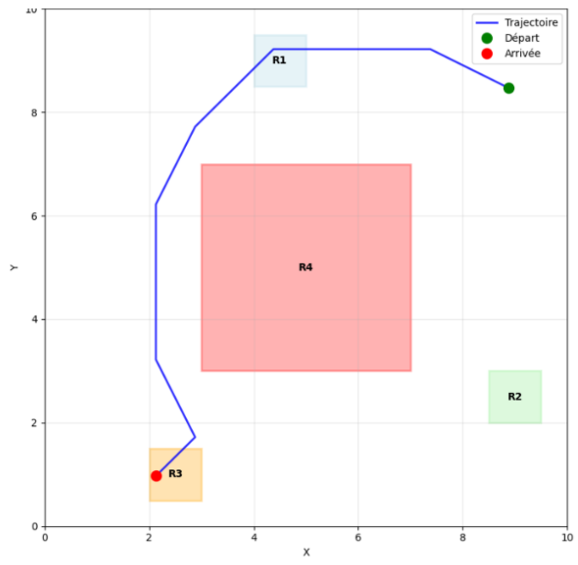
- **Prioritizes correct sequence**: Strongly rewards controls leading to the next required region in the sequence
- **Penalizes deviations**: Assigns infinite cost to controls that would visit incorrect regions
- **Maintains strict progression**: Ensures the robot follows the exact sequence without shortcuts or reversals
- **Continuous validation**: Monitors both symbolic automaton state and continuous region membership

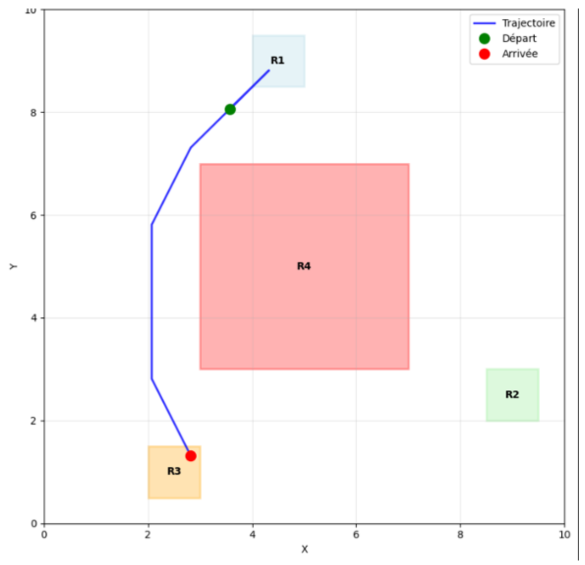
IV-B1 Execution result

Code Execution for Objective 2:

```
# === OBJECTIVE 2 :===
print("\n=== OBJECTIVE 2: ===")
automaton2 = AutomatonObjective2()
synth2 = AutomatonBasedControllerSynthesis(
    transitions, automaton2, R1_indices,
    R2_indices, R3_indices, R4_indices, robot
)
ctrl_2, Q0_2 = synth2.
    synthesize_dynamic_controller()
if Q0_2 and not success:
    print("\Objective 2 :")
    for trial in range(3):
        if not Q0_2:
            break
        s0 = random.choice(list(Q0_2))
        path, traj = synth2.
    execute_controller_objective2(s0, ctrl_2, R3)
        if traj and len(traj) > 1:
            plot_trajectory(traj, regions_dict, "
    Obj 2")
            success = True
            break
```

Listing 9. Objective 2 Execution Code

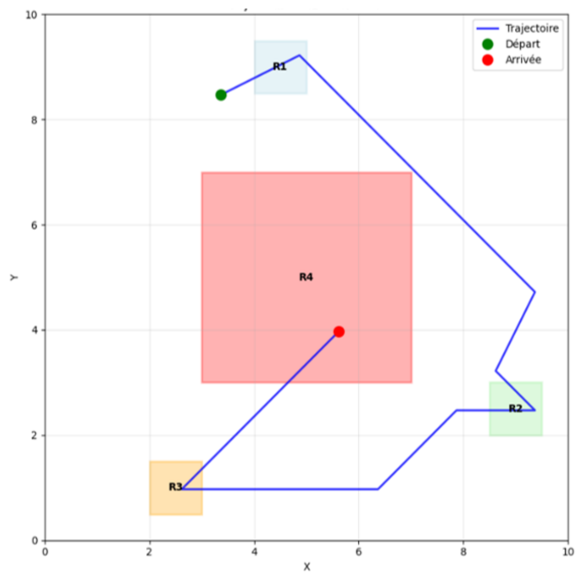


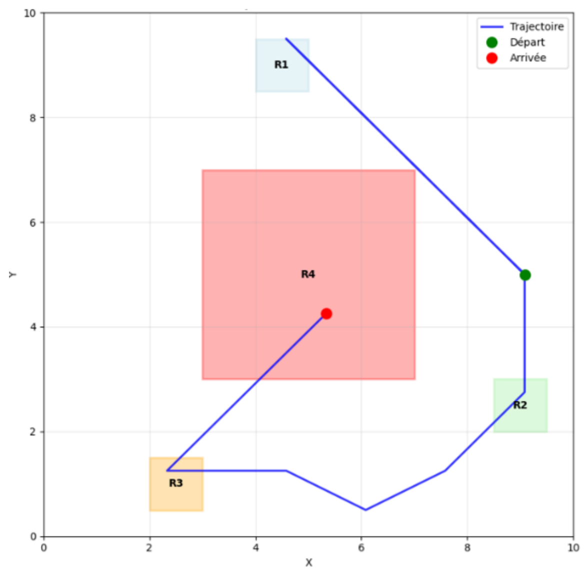


Code Execution for Objective 3:

```
# === OBJECTIVE3 : R1     R2     R3     R4 (
    strict sequence) ===
print("\n=== OBJECTIF 3 : Encha nement strict R1
         R2     R3     R4 ===")
automaton3 = AutomatonObjective3()
synth3 = AutomatonBasedControllerSynthesis(
    transitions, automaton3, R1_indices,
    R2_indices, R3_indices, R4_indices, robot
)
ctrl_3, Q0_3 = synth3.
    synthesize_dynamic_controller()
if Q0_3:
    print("\ n   Test Objective 3 (R1     R2
    R3     R4):")
    for trial in range(5):
        s0 = random.choice(list(Q0_3))
        print(f"  Trial {trial+1}: start   ={s0}"
    )
        path, traj = synth3.
    execute_controller_objective3(s0, ctrl_3, R4)
        if traj and len(traj) > 1:
            plot_trajectory(traj, regions_dict, "
    Obj 3: R1     R2     R3     R4")
            if path:
                phases = [q for (_, _, q, _) in
    path]
                if phases:
                    print(f"    Phases: {phases
    [0]}     ...     {phases[-1]}")
            break
```

Listing 10. Objective 3 Execution Code





V Conclusion

This project presented a complete symbolic control framework for guaranteeing the correctness of autonomous navigation in 2D mobile robot systems. By constructing a finite-state abstraction of the continuous robot dynamics, we transformed complex control problems into algorithmically tractable discrete synthesis tasks. The implementation covered every stage of the pipeline: state-space discretization, transition relation construction under bounded perturbations, safety and reachability synthesis via fixed-point computations, and controller extraction ensuring provable guarantees.

The experimental objectives demonstrated the ability of symbolic controllers to enforce obstacle avoidance, ensure mission completion, and handle sophisticated temporal-ordering constraints. Through the use of automata-based mission encoding, the framework successfully addressed both disjunctive temporal goals and strictly sequential missions. Each controller was derived with mathematical guarantees of correctness, robustness with respect to uncertainties, and completeness with respect to the symbolic abstraction.

Overall, this work illustrates how formal methods can be concretely deployed in robotics to provide verified autonomy. Beyond the specific 2D system studied here, the methodology can be extended to higher-dimensional systems, multi-robot coordination, and more expressive temporal logic specifications. Future work may focus on improving abstraction efficiency, reducing computational complexity, and integrating learning-based components while preserving formal guarantees.