Symbolic Controllers for Robotic Systems

Project Report

Team Members:

Salma AMMARI
Ghita BELLAMINE
Yasser BAOUZILE

Supervisor:

Pr. Adnane SAOUD

December 2025

Project Summary

Overview of Symbolic Control Methods

1. **Symbolic Abstraction**
2. **Safety Controller Synthesis**
3. **Reachability Controller Synthesis**
4. **Combined Safe Reachability**
5. **Automaton-Based Specifications**

Introduction

In the field of automatic control, the use of highly faithful mathematical models — nonlinear dynamics, state and input constraints, and perturbations — generally increases the reliability and confidence in the results obtained from system analysis. However, this complexity often makes the synthesis of controllers extremely challenging, and in some cases, even impossible.

Traditional methods in continuous control theory mainly focus on objectives such as stability. While effective in certain contexts, they are not well-suited to handle richer specifications such as safety, reachability, or temporal logic properties that are increasingly relevant in computer science and robotics.

Symbolic approaches aim to bridge this gap by relying on three successive steps:

1. **Abstraction** of the continuous dynamic system into a symbolic discrete model with a finite number of states and inputs.
2. **Controller synthesis** in the discrete domain.
3. **Concretization** of the symbolic controller for application on the original continuous system.

Unlike predictive control methods, all three steps are performed offline, and the online application of the controller is reduced to a simple correspondence table. This methodology offers several advantages:

- Automatic handling of nonlinear systems with perturbations, uncertainties, and constraints.
- Specification of complex properties using temporal logic and automata.
- Formal guarantees of correctness by construction, thanks to algorithms from formal methods and model checking.
- Robustness through over-approximations of system behaviors.

Nevertheless, symbolic approaches face the challenge of exponential complexity with respect to the dimension of the state space. Recent research has proposed several techniques to mitigate this issue and improve scalability.

Our project builds upon these foundations to design symbolic controllers for robotic systems, enabling them to satisfy diverse specifications such as safety, reachability, and temporal logic constraints. By leveraging abstraction, synthesis, and concretization, we aim to demonstrate the practical applicability of symbolic control in robotics through illustrative examples and case studies.

1. Symbolic Abstraction of a Complex System

1.1 Methodology of Symbolic Abstraction

We consider a robot modeled by the following nonlinear dynamics:

$$x(t+1) = f(x(t), u(t), w(t)) \qquad (1)$$

where $x(t) \in \mathbb{R}^{n_x}$, $u(t) \in \mathbb{R}^{n_u}$, and $w(t) \in \mathbb{R}^{n_w}$ represent the state, control input, and perturbation respectively.

To abstract this system, we discretize the state space $\mathbb{R}^{n_x}$ into a finite set $\mathcal{X} = \bigcup_{i=1}^{m} \mathcal{X}_i$, where each $\mathcal{X}_i$ is a non-overlapping cell. The symbolic state space is then defined as $\Xi = \{1, \ldots, m\}$, and the abstraction function $q(x) = \xi$ maps each continuous state to its symbolic representative.

Similarly, the control space is discretized into $\Sigma = \{1, \ldots, p\}$, representing symbolic control actions. The symbolic dynamics are defined by:

$$\xi(t+1) \in g(\xi(t), \sigma(t)) \qquad (2)$$

1.1.1 Discretization of State and Control

In our implementation, we discretize the robot's state space as follows: - x and y: each divided into 20 intervals $\rightarrow N_x = N_y = 20$ - θ (orientation): divided into 10 intervals $\rightarrow N_\theta = 10$

This results in a total of $20 \times 20 \times 10 = 4000$ symbolic states.

The control space is discretized as: - v (linear velocity): 3 values - ω (angular velocity): 5 values

This yields $3 \times 5 = 15$ symbolic control actions.

Code Snippet: Abstraction Class

```python
class RobotAbstraction:
    def __init__(self, state_intervals,
                 control_values, perturbation, delta_t):
        self.state_intervals = state_intervals
        # [(x_min, x_max), (y_min, y_max), (theta_min, theta_max)]
        self.control_values = control_values
        # [(v_min, v_max), (omega_min, omega_max)]
        self.perturbation = perturbation
        # [(w1_min, w1_max), (w2_min, w2_max), (w3_min, w3_max)]
        self.delta_t = delta_t
        # sampling time (time to change a state) = 1s
        self.state_to_index = {"OutOfGrid": -1}
        # Map from state tuple to symbolic state
        self.index_to_intervals = {}
        # Map from symbolic state to intervals
        self.state_edges = []
        self.discrete_x_y = 20
        self.discrete_tetha = 10
        self.v_vals = 3
        self.omega_vals = 5
        self._create_state_mapping()
        # Precompute the symbolic state mapping
        self.compute_transitions_dict()
```

Listing 1: Robot Abstraction Class

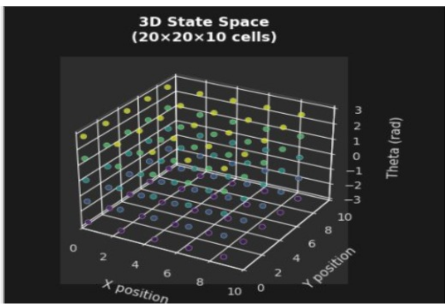

Figure 1: Visualization of the abstraction process showing continuous state space discretization into symbolic states.

1.2 Dynamics and Reachability Analysis

The transitions between symbolic states are computed using interval-based reachability analysis. For a system with bounded derivatives, we can over-approximate the reachable set using:

$$f(\mathrm{cl}(X_\xi), u_\sigma, W) \subseteq [f(x^*, u_\sigma, w^*) - D_x\delta_x - D_w\delta_w, \\ f(x^*, u_\sigma, w^*) + D_x\delta_x + D_w\delta_w]$$

where:

- x^*: Center of state interval $X_\xi = [\underline{x}, \overline{x}]$
- w^*: Center of perturbation interval $W = [\underline{w}, \overline{w}]$
- $\delta_x = \frac{\overline{x}-\underline{x}}{2}$: Half-width of state interval
- $\delta_w = \frac{\overline{w}-\underline{w}}{2}$: Half-width of perturbation interval
- D_x, D_w: Bounds on partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial w}$

Code Snippet: Dynamics and Transition Computation

```
def dynamics(self, x_center, u, w_center, D_x, D_w):
    """Modified system dynamics based on interval reachability."""
    # Nominal dynamics calculation
    nominal_dynamics = np.array([
        x_center[0] + self.delta_t *
            (u[0] * np.cos(x_center[2]) + w_center[0]),
        x_center[1] + self.delta_t *
            (u[0] * np.sin(x_center[2]) + w_center[1]),
        x_center[2] + self.delta_t *
            (u[1] + w_center[2])
    ])

    # Calculate delta_x based on state discretization
    delta_x = np.array([
        (self.state_intervals[0][1] -
         self.state_intervals[0][0]) /
         (self.discrete_x_y - 1) / 2,
        (self.state_intervals[1][1] -
         self.state_intervals[1][0]) /
         (self.discrete_x_y - 1) / 2,
        (self.state_intervals[2][1] -
         self.state_intervals[2][0]) /
         (self.discrete_tetha - 1) / 2
    ])

    # Calculate delta_w based on perturbation intervals
    delta_w = np.array([
        (self.perturbation[0][1] -
         self.perturbation[0][0]) / 2,
        (self.perturbation[1][1] -
         self.perturbation[1][0]) / 2,
        (self.perturbation[2][1] -
         self.perturbation[2][0]) / 2
    ])

    # Apply reachability formula
    lower_bound = nominal_dynamics - D_x @ delta_x - D_w @ delta_w
    upper_bound = nominal_dynamics + D_x @ delta_x + D_w @ delta_w

    return lower_bound, upper_bound
```

Listing 2: Dynamics with Reachability Analysis

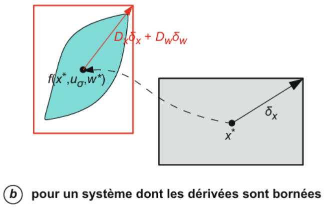

Figure 2: Interval-based reachability analysis for transition computation.

2. Synthesis of Symbolic Controller

2.1 Safety Controller Synthesis

After constructing the symbolic model through abstraction, we now address the controller synthesis problem. Given a safety specification defined by a set of safe states $Q_s \subseteq \Xi$, our goal is to synthesize a controller that guarantees the system remains within Q_s indefinitely.

The core algorithm for safety controller synthesis is based on computing the maximal safe set R^* using the predecessor operator $\mathrm{Pre}(R)$, defined as:

$$\mathrm{Pre}(R) = \{\xi \in \Xi \mid \exists \sigma \in \Sigma, \emptyset \neq g(\xi, \sigma) \subseteq R\}$$

where $g(\xi, \sigma)$ represents the successors of state ξ under control σ.

Algorithm 1: Maximal Safe Set Computation

Fixed-point Computation of R^*

Input: $Q_s \subseteq \Xi$ (safe states)
Output: R^* (maximal safe set)

1. $R_0 \leftarrow Q_s$
2. **repeat**
   - $R_{k+1} \leftarrow Q_s \cap \mathrm{Pre}(R_k)$
3. **until** $R_{k+1} = R_k$
4. **return** $R^* \leftarrow R_k$

Python Implementation

```
class SymbolicControllerSynthesis:
    def __init__(self, transitions, safety_states):
        """
        Initialize the symbolic controller synthesis.

        Parameters:
        - transitions: List of transitions in the form
            [(state, control, successors), ...].
        - safety_states: Set of safe states (Q_s).
        """
        self.transitions = self._process_transitions(transitions)
        # Convert list to dictionary
        self.safety_states = safety_states

    def _process_transitions(self, transitions):
        """
        Convert the list of transitions into a dictionary format.

        Parameters:
        - transitions: List of transitions
```

```
            [(state, control, successors), ...].

        Returns:
        - A dictionary mapping (state, control) -> set(successors).
        """
        transition_dict = defaultdict(set)
        for state, control, successors in transitions:
            transition_dict[(state, control)] = successors
        return dict(transition_dict)

    def pre(self, R):
        """
        Compute the predecessor operator Pre(R), considering only
        states with valid transitions.

        Parameters:
        - R: Set of states (current safe set).

        Returns:
        - Set of states that can transition to R.
        """
        pre_states = set()
        for (state, control), successors in self.transitions.items():
            # Check if all successors are in R
            if successors and successors.issubset(R):
                pre_states.add(state)
        return pre_states

    def compute_safe_controller(self):
        """
        Compute the maximal safe set (R*).

        Returns:
        - R*: Maximal set of safe states.
        """
        R = self.safety_states.copy()
        Q_s = R
        while True:
            R_next = Q_s.intersection(self.pre(R))
            if R_next == R:
                break
            R = R_next
        return R

    def synthesize_controller(self):
        """
        Synthesize a safe controller.

        Returns:
        - R*: Maximal safe set.
        - Controller: Mapping from states to safe controls.
        - Q_0: Set of valid initial states.
        """
        R_star = self.compute_safe_controller()
        controller = defaultdict(set)
        Q_0 = set()

        for (state, control), successors in self.transitions.items():
            # Add controls only if all successors are in R*
            if state in R_star and successors.issubset(R_star):
                controller[state].add(control)

        # Q_0: States in R_star with at least one valid control
        for state in R_star:
            if state in controller:
                Q_0.add(state)

        return R_star, controller, Q_0
```

Listing 3: Symbolic Controller Synthesis Class

Safety Controller Application

Once the maximal safe set R^* is computed and non-empty ($R^* \neq \emptyset$), we can define a safe controller $h : \Xi \to \Sigma$ as:

$$h(\xi) \in H(\xi) = \{\sigma \in \Sigma \mid \emptyset \neq g(\xi, \sigma) \subseteq R^*\}$$

This controller guarantees that for any initial state $\xi_0 \in R^*$, all trajectories satisfy the safety specification:

$$\forall t \in \mathbb{N}, \xi(t) \in Q_s$$

Explanation of the Synthesis Code

The provided Python class `SymbolicControllerSynthesis` implements Algorithm 1 for computing the maximal safe set R^*. The key methods are:

- `__init__`: Initializes the class with system transitions and safety states. The transitions are converted from a list format to a dictionary for efficient lookup.
- `pre(R)`: Implements the predecessor operator $\text{Pre}(R)$, returning all states that can transition entirely into the set R in one step. It checks if *all* successors of a given state-control pair are contained in R.
- `compute_safe_controller()`: Executes the fixed-point iteration $R_{k+1} = Q_s \cap \text{Pre}(R_k)$ until convergence ($R_{k+1} = R_k$) to find the maximal safe set R^*.
- `synthesize_controller()`: Constructs the actual controller mapping $h : \Xi \to \Sigma$ from states in R^* to safe control actions, ensuring all successors remain in R^*. It also returns the set Q_0 of initial states from which the controller is defined.

This implementation guarantees that for any initial state in the resulting set $Q_0 \subseteq R^*$, applying the synthesized controller will keep the system within the safe set Q_s indefinitely. The algorithm's complexity is determined by the number of state-control transitions and the number of iterations needed for the fixed-point computation.

Implementation Example: Safety Specification for Avoiding a Certain Region

Consider a robot navigating in a grid with obstacles. The safety specification Q_s includes all states not occupied by obstacles. The synthesis algorithm computes:

1. **Initialization**: $R_0 = Q_s$ (all non-obstacle states)
2. **Iteration**: Remove states that cannot avoid entering unsafe regions within k steps
3. **Fixed-point**: R^* contains states from which safety can be guaranteed indefinitely

The resulting controller provides safe controls for each state in R^*, ensuring obstacle avoidance while allowing progress toward goals.

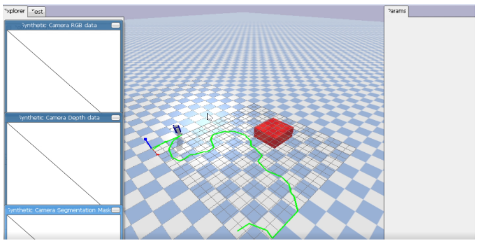

Figure 3: Safety controller application: The robot avoids the unsafe region (red) while navigating to the target (green). The blue arrows represent safe control actions synthesized by the symbolic controller.

2.2 Reachability Controller Synthesis

Reachability Properties

A reachability property consists of driving the closed-loop trajectories of the symbolic model toward a target subset $Q_e \subseteq \Xi$. Formally, a reachability specification is given by the following set of trajectories:

$$S = \{\xi : \mathbb{N} \to \Xi \mid \exists t \in \mathbb{N}, \xi(t) \in Q_e\}.$$

The synthesis of controllers achieving reachability specifications is similar to safety controller synthesis. Indeed, consider the sequence of sets R_k computed by Algorithm 2. The only difference with Algorithm 1 is the replacement of the intersection computation by a union computation. The set R_k represents the set of initial states from which the trajectories of the symbolic model can reach Q_e in at most k time steps.

Algorithm 2: Reachability Domain Computation

Reachability Domain R^* Computation

Input: $Q_e \subseteq \Xi$ (target states)
Output: R^* (reachability domain)

1. $R_0 \leftarrow Q_e$
2. **repeat**
   - $R_{k+1} \leftarrow Q_e \cup \mathrm{Pre}(R_k)$
3. **until** $R_{k+1} = R_k$
4. **return** $R^* \leftarrow R_k$

Once the sequence of sets R_k is obtained by Algorithm 2, a static reachability controller can be obtained as follows:

Theorem 4: Reachability Controller

The reachability problem is solved on the symbolic model by any controller $h : \Xi \to \Sigma$ satisfying:

$$\forall k \in \mathbb{N}, \forall \xi \in R_{k+1}, \emptyset \neq g(\xi, h(\xi)) \subseteq R_k.$$

Using such a controller in closed-loop on the symbolic model guarantees satisfaction of the specification:

$$\forall \xi_0 \in R^*, T_{\text{symb}}(\xi_0) \subseteq \{\xi : \mathbb{N} \to \Xi \mid \exists t \in \mathbb{N}, \xi(t) \in Q_e\}.$$

Python Implementation: Reachability Controller

```
class SymbolicReachabilityController:
    """
    Symbolic reachability controller based on
    Algorithm 2 and Theorem 4
    """

    def __init__(self, robot, target_states):
        self.robot = robot
        self.target_states = target_states  # Q_e - target states
        self.R_star = None  # Fixed point R*
        self.R_sequence = []  # R_k sequence for control
        self.H = {}  # Multivalued controller

    def compute_R_star(self):
        """
        Algorithm 2: Computation of fixed point R* for reachability
        R_{k+1} = Q_e    Pre(R_k)
        """
        print("Computing R* (reachability fixed point)...")

        # R_0 = Q_e (target states)
        R_prev = set(self.target_states)
        self.R_sequence = [R_prev.copy()]
        iteration = 0

        while True:
            iteration += 1
```

```python

            # R_{k+1} = Q_e  Pre(R_k)
            R_new = set(self.target_states)  # Start with Q_e
            R_new.update(self.pre(R_prev))   # Union with Pre(R_k))

            print(f"Iteration {iteration}: |R| = {len(R_new)}")
            self.R_sequence.append(R_new.copy())

            # Fixed point condition
            if R_new == R_prev:
                break

            R_prev = R_new

            # Safety: avoid infinite loops
            if iteration > 100:
                print("Warning: Maximum iterations reached")
                break

        self.R_star = R_new
        print(f"R* computation completed: {len(self.R_star)} reachable states")
        return self.R_star

    def pre(self, R):
        """
        Pre(R) operator - states that can reach R in one step
        """
        pre_states = set()

        # For each possible state, check if it can reach R
        for state in range(1, len(self.robot.index_to_intervals) + 1):
            if self._can_reach_set(state, R):
                pre_states.add(state)

        return pre_states

    def _can_reach_set(self, state, target_set):
        """
        Checks if a state can reach the target set in one step
        """
        possible_actions = self._get_possible_actions(state)

        for action in possible_actions:
            next_states = self._get_next_states(state, action)

            # Check if at least one successor is in target set
            for next_state in next_states:
                if next_state in target_set:
                    return True

        return False

    def compute_reachability_controller(self):
        """
        Computes reachability controller according to Theorem 4
             k       N,            R_{k+1},        g(   , h(   ))
            R_k
        """
        if self.R_star is None:
            self.compute_R_star()

        print("Computing reachability controller...")

        # For each state in R*, find actions leading to target
        for state in self.R_star:
            # Find smallest k such that state     R_k
            k_level = None
            for k, R_k in enumerate(self.R_sequence):
                if state in R_k:
                    k_level = k
                    break

            if k_level is not None and k_level > 0:
                # We want to reach R_{k-1}
```

```python
                    target_set = self.R_sequence[k_level - 1]
                    valid_actions = []

                    possible_actions = self._get_possible_actions(state)
                    for action in possible_actions:
                        next_states = self._get_next_states(state, action)

                        # Check if at least one successor is in target_set
                        reaches_target = any(ns in target_set for ns in next_states)

                        if reaches_target and next_states:
                            # Theorem 4 condition
                            valid_actions.append(action)

                    self.H[state] = valid_actions

        print(f"Reachability controller computed for {len(self.H)} states")
        return self.H

    def get_reachability_action(self, continuous_state):
        """
        Implements the concretized controller for reachability
        """
        # Discretization
        discrete_state = self.robot._find_state(np.array(continuous_state))

        # Check if state is in R*
        if discrete_state in self.H and self.H[discrete_state]:
            # Choose a valid action
            return self.H[discrete_state][0]
        else:
            # Default action (exploration)
            print(f"Warning: No reachability action for state {discrete_state}")
            return (1.0, 0.0)  # Move forward
```

Listing 4: Symbolic Reachability Controller Class

Explanation of the Reachability Controller Code

The `SymbolicReachabilityController` class implements Algorithm 2 for computing the reachability domain R^*:

- `__init__`: Initializes with robot abstraction and target states Q_e
- `compute_R_star()`: Implements Algorithm 2 with fixed-point iteration $R_{k+1} = Q_e \cup \mathrm{Pre}(R_k)$
- `pre(R)`: Computes predecessor states that can reach R in one step (different from safety controller)
- `compute_reachability_controller()`: Implements Theorem 4 to synthesize controller ensuring progression toward target
- `get_reachability_action()`: Concretizes the symbolic controller for continuous state application

Implementation Example: Reaching a Target Region

The reachability controller can be used to navigate a robot to a specific region:

```python
# Define target region
target_region = [[4.0, 6.0], [4.0, 6.0]]  # x: [4,6], y: [4,6]

# Find symbolic states in target region
```

```python
target_states = []
for state_idx, intervals in robot
    .index_to_intervals.items():
     x_center = (intervals[0][0] +
     intervals[0][1]) / 2
     y_center = (intervals[1][0] +
     intervals[1][1]) / 2
     if target_region[0][0] <=
    x_center <= target_region
    [0][1] and \
         target_region[1][0] <=
    y_center <= target_region
    [1][1]:
          target_states.append(
    state_idx)

# Create and compute reachability
     controller
reachability_controller =
    SymbolicReachabilityController
    (
     robot, target_states)
R_star = reachability_controller.
    compute_R_star()
controller =
    reachability_controller.
    compute_reachability_controller
    ()

# Run simulation
initial_state = [1.0, 1.0, 0.0]
    # Start at (1,1)
trajectory, reached =
    run_reachability_simulation(
     robot,
    reachability_controller,
     initial_state, target_region,
     steps=100
)
```

Listing 5: Reachability Controller Application

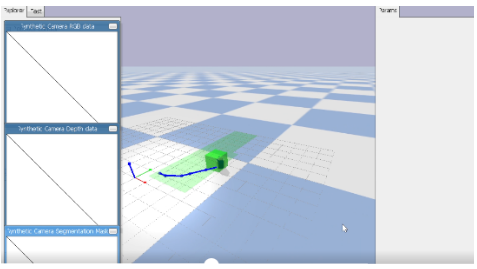

Figure 4: Reachability specification visualization: The robot must reach the target region (green) from various initial positions (blue).

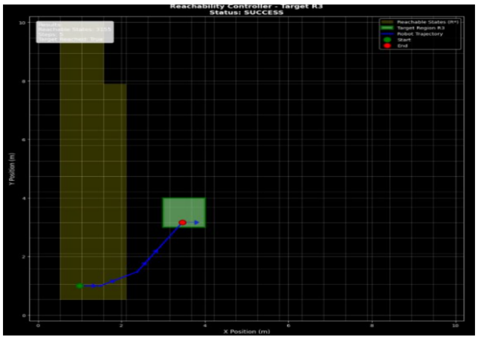

Figure 5: Reachability-controlled trajectory: The robot successfully navigates to the target region using the synthesized symbolic controller.

2.3 Combined Safe Reachability Controller

Problem Formulation: Reach While Avoiding

In many practical scenarios, we need to combine safety and reachability specifications. A common requirement is to reach a target region Q_e while avoiding an unsafe region Q_u. Formally, we want trajectories that satisfy:

$$\exists t \in \mathbb{N}, \xi(t) \in Q_e \quad \text{and} \quad \forall t \in \mathbb{N}, \xi(t) \notin Q_u$$

This combined specification can be solved using a backward BFS algorithm that simultaneously considers both constraints. The key insight is to maintain only states that can reach the target without ever entering the unsafe region.

Algorithm 3: Safe Reachability Controller

Safe Reachability Controller Synthesis

Input: $Q_e \subseteq \Xi$ (target states), $Q_u \subseteq \Xi$ (unsafe states)
Output: R^* (safe reachable states), H (controller)

1. Initialize $R_0 \leftarrow Q_e$, level $\ell(s) \leftarrow 0$ for all $s \in Q_e$
2. **for** $k = 1, 2, \ldots$ **do**:
   - $R_k \leftarrow R_{k-1}$
   - **for each** state $s \notin R_{k-1} \cup Q_u$:
     - Find action a such that:
       (a) $\exists s' \in g(s,a)$ with $s' \in R_{k-1}$
       (b) $\forall s' \in g(s,a), s' \notin Q_u$
     - If such a exists, add s to R_k, set $\ell(s) \leftarrow k$
3. **until** no new states added
4. Synthesize controller $H(s)$: actions that reduce $\ell(s)$ while avoiding Q_u

Python Implementation: Safe Reachability Controller

```python
class SafeReachabilityController:
    """
    Reach target while avoiding unsafe region.
    Combines safety and reachability constraints.
    """

    def __init__(self, robot, target_states, unsafe_states):
        self.robot = robot
        self.target_states = set(target_states)
        self.unsafe_states = set(unsafe_states)
        self.R_star = set()  # Safe reachable states
        self.level = {}      # BFS level (0 for target states)
        self.H = {}          # Controller: state -> list of actions

        # Precompute transitions for efficiency
        self.actions = self.robot.sample_actions()
        self.transitions = {}
        for state in self.robot.index_to_intervals.keys():
            for a in self.actions:
                self.transitions[(state, a)] = \
                    self.robot.successors_overapprox(state, a, samples=3)

    def compute_R_star(self):
        """
        Backward BFS from target states while avoiding unsafe region
        """
        print("Computing safe R* (fast BFS)...")
        frontier = set(self.target_states)
        self.R_star = set(frontier)
        for s in frontier:
            self.level[s] = 0

        max_iters = 40
        it = 0

        while frontier and it < max_iters:
            it += 1
            new_frontier = set()

            for state in self.robot.index_to_intervals.keys():
                if state in self.R_star:  # Already in
                    continue
                if state in self.unsafe_states:  # Avoid unsafe
```

```python
                    continue

                # Check if state can reach frontier safely
                for a in self.actions:
                    succ = self.transitions[(state, a)]
                    if not succ:
                        continue

                    reaches_frontier = any(ns in frontier for ns in succ)
                    avoids_unsafe = all(ns not in self.unsafe_states
                                        for ns in succ)

                    if reaches_frontier and avoids_unsafe:
                        self.R_star.add(state)
                        self.level[state] = self._min_level_in(succ) + 1
                        new_frontier.add(state)
                        break

            frontier = new_frontier
            print(f"Iteration {it}: |R*| = {len(self.R_star)}")

        print(f"Safe R* done. Total states: {len(self.R_star)}")
        return self.R_star

    def compute_safe_reachability_controller(self):
        """
        Synthesize controller that reduces level
        while avoiding unsafe region
        """
        if not self.R_star:
            self.compute_R_star()

        print("Computing safe controller policy...")

        for state in self.R_star:
            valid_actions = []
            best_level = float('inf')

            for a in self.actions:
                succ = self.transitions[(state, a)]
                if not succ:
                    continue

                avoids_unsafe = all(ns not in self.unsafe_states
                                    for ns in succ)
                succ_levels = [self.level.get(ns, float('inf'))
                               for ns in succ]
                min_succ_level = min(succ_levels) if succ_levels else float('inf')

                # Choose actions that reduce level while avoiding unsafe region
                if avoids_unsafe and min_succ_level < self.level.get(state,
                                                                     float('inf')):
                    if min_succ_level < best_level:
                        valid_actions = [a]
                        best_level = min_succ_level
                    elif min_succ_level == best_level:
                        valid_actions.append(a)

            self.H[state] = valid_actions

        print(f"Policy computed for {len(self.H)} states.")
        return self.H
```

Listing 6: Safe Reachability Controller Class

Key Features of the Combined Controller

- **Efficiency**: Precomputes transitions to avoid repeated computations
- **Safety guarantee**: Ensures trajectories never enter unsafe region
- **Progress guarantee**: Always reduces BFS level toward target
- **Optimality**: Prefers actions with maximum progress toward target

Figure 6: Safe reachability specification: Reach green target while avoiding red unsafe region.

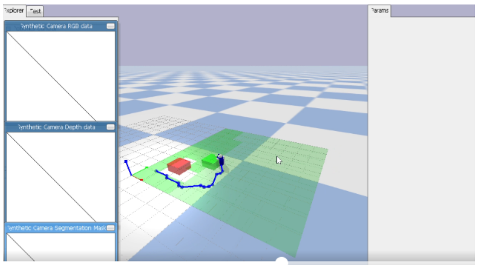

Figure 7: Safe reachability trajectory: The robot successfully reaches target while avoiding obstacles.

3. Automaton-Based Specifications

3.1 Methodology for Automaton-Based Specifications

Automaton-based specifications allow us to express complex temporal logic properties using finite automata. The approach follows these key steps:

Step 1: Define Labeling Function and DFA

Consider a labeling function $\ell : \Xi \to L$ where L is a finite set of labels. We then define a Deterministic Finite Automaton (DFA) $\mathcal{A} = (W, L, \delta, w_{\text{init}}, W_f)$ where:

- W: Set of automaton states
- L: Set of labels
- $\delta : W \times L \to W$: Transition function
- $w_{\text{init}} \in W$: Initial state
- $W_f \subseteq W$: Accepting (final) states

Given a trajectory $\xi : \mathbb{N} \to \Xi$, we can associate an automaton trace $w = \tau_{\mathcal{A}}(\xi)$ defined by:

$$w(t) = \delta(w(t-1), \ell(\xi(t))) \quad \text{with} \quad w(-1) = w_{\text{init}}$$

The specification defined by automaton $\mathcal{A}$ corresponds to trajectories whose trace reaches a final state:

$$S_{\mathcal{A}} = \{\xi : \mathbb{N} \to \Xi \mid \exists t \in \mathbb{N}, w(t) \in W_f, \text{ with } w = \tau_{\mathcal{A}}(\xi)\}$$

Example Specification: Visit R1 XOR R2, Avoid R4, Reach R3

Consider four disjoint subsets Q_1, Q_2, Q_3, Q_4 of symbolic states Ξ. The specification: "Go to Q_1 OR Q_2 (exclusively), then to Q_3, while avoiding Q_4 throughout the path" can be formalized as:

- Labels: $L = \{0, 1, 2, 3, 4\}$ with labeling $\ell(\xi) = i$ if $\xi \in Q_i$, or $\ell(\xi) = 0$ if $\xi \notin \bigcup_{i=1}^{4} Q_i$
- Automaton states: $W = \{a, b, c, d, e\}$, initial state $w_{\text{init}} = a$, final states $W_f = \{d\}$

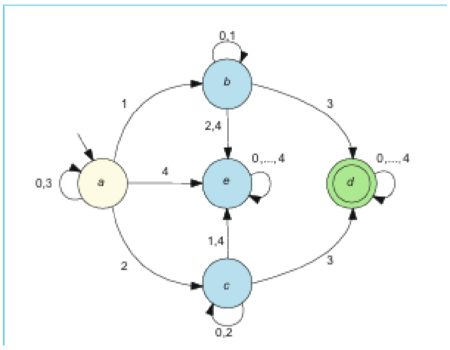

Figure 8: DFA structure for the specification: Visit R1 XOR R2, Avoid R4, Reach R3.

Step 2: Build Augmented System

To synthesize a controller for specification $S_{\mathcal{A}}$, we construct an **augmented system** that combines the symbolic model with the DFA:

$$\begin{cases} w(t+1) = h_1(w(t), \xi(t+1)) \\ \xi(t+1) = g(\xi(t), \sigma(t)) \end{cases}$$

with $w(0) = h_1(w_{\text{init}}, \xi(0))$, where $h_1(w, \xi) = \delta(w, \ell(\xi))$.

Letting $\zeta(t) = \begin{pmatrix} w(t) \\ \xi(t) \end{pmatrix}$, the augmented system can be written as:

$$\zeta(t+1) = \tilde{g}(\zeta(t), \sigma(t)), \quad \zeta(t) \in W \times \Xi, \quad \sigma(t) \in \Sigma$$

Step 3: Convert to Reachability Problem

The key insight is that a trajectory ξ of the original system satisfies $S_{\mathcal{A}}$ if and only if the corresponding trajectory ζ of the augmented system satisfies a **reachability specification**:

$$S_{\text{reach}} = \{\zeta : \mathbb{N} \to W \times \Xi \mid \exists t \in \mathbb{N}, \zeta(t) \in W_f \times \Xi$$

Thus, we have transformed the complex automaton specification into a standard reachability problem on the augmented state space.

Step 4: Synthesize Controller on Augmented System

Using the reachability synthesis approach from Section 2.2, we can synthesize a controller $\tilde{h} : W \times \Xi \to \Sigma$ and an initial set $\tilde{Q}_0 \subseteq W \times \Xi$ for specification S_{reach}.

Step 5: Project Back to Original System

Finally, we define the controller for the original system as:

$$h(\xi) = \tilde{h}(h_1(w_{\text{init}}, \xi), \xi)$$

with initial states:

$$Q_0 = \{\xi \in \Xi \mid (h_1(w_{\text{init}}, \xi), \xi) \in \tilde{Q}_0\}$$

Theorem 5: Automaton Controller Synthesis

The synthesis problem for specification $S_{\mathcal{A}}$ is solved on the symbolic model by the controller defined above. Using this controller in closed-loop on the symbolic model guarantees satisfaction of the automaton specification:

$$\forall \xi_0 \in Q_0, \quad T_{\text{symb}}(\xi_0) \subseteq S_{\mathcal{A}}$$

Python Implementation: DFA Controller

```
class DFA:
    """Deterministic Finite Automaton for complex specifications"""

    def __init__(self, states, init_state,
                 accepting_states, transitions):
        self.states = states
        self.init_state = init_state
        self.accepting_states = accepting_states
        self.transitions = transitions
        # dict: (state, label) -> next_state

```

```python
    def step(self, state, label):
        """Execute one transition in the DFA"""
        return self.transitions.get((state, label), None)

class AugmentedSystem:
    """
    Product system: Symbolic Model  DFA
    Combines robot dynamics with automaton state
    """

    def __init__(self, robot, dfa, region_labels):
        self.robot = robot
        self.dfa = dfa
        self.region_labels = region_labels
        # maps symbolic states to labels

        # Build augmented state space: (dfa_state, robot_state)
        self.augmented_states = []
        self.state_to_idx = {}
        self.idx_to_state = {}
        self.targets = set()
        self._build_augmented_space()

    def _build_augmented_space(self):
        idx = 0
        for robot_state in self.robot.index_to_intervals.keys():
            for dfa_state in self.dfa.states:
                aug_state = (dfa_state, robot_state)
                self.augmented_states.append(aug_state)
                self.state_to_idx[aug_state] = idx
                self.idx_to_state[idx] = aug_state

                # Mark accepting states as targets
                if dfa_state in self.dfa.accepting_states:
                    self.targets.add(idx)
                idx += 1

    def label_state(self, robot_state):
        """Get DFA label for a robot symbolic state"""
        return self.region_labels.get(robot_state, 'other')

    def successors(self, aug_idx, action):
        """
        Compute successors in augmented system
        Returns list of augmented state indices
        """
        dfa_state, robot_state = self.idx_to_state[aug_idx]
        successors = []

        # Get robot successors for this action
        robot_successors = self.robot.get_successors(robot_state, action)

        for next_robot_state in robot_successors:
            # Get label for next robot state
            label = self.label_state(next_robot_state)

            # Compute next DFA state
            next_dfa_state = self.dfa.step(dfa_state, label)
            if next_dfa_state is None:
                continue  # Invalid transition in DFA

            # Create augmented successor state
            next_aug_state = (next_dfa_state, next_robot_state)
            next_idx = self.state_to_idx[next_aug_state]
            successors.append(next_idx)

        return successors

class DFAReachabilityController:
    """
```

```python
    Controller synthesis for DFA specifications via
    reachability on augmented system
    """

    def __init__(self, augmented_system):
        self.augmented = augmented_system
        self.R_star = None  # Winning region in augmented space
        self.layers = []    # Backward reachability layers
        self.H = {}         # Controller: aug_state_idx -> list of actions

    def compute_R_star(self):
        """Backward reachability on augmented system"""
        print("Computing R* for DFA specification...")
        R_prev = set(self.augmented.targets)
        self.layers = [R_prev.copy()]
        it = 0

        while True:
            it += 1
            R_new = set(self.augmented.targets)

            # Pre operator on augmented system
            pre_states = set()
            for idx in range(len(self.augmented.augmented_states)):
                if idx in R_prev:
                    continue

                # Check if state can reach R_prev
                for action in self.augmented.robot.ACTIONS:
                    succs = self.augmented.successors(idx, action)
                    if any(s in R_prev for s in succs):
                        pre_states.add(idx)
                        break

            R_new.update(pre_states)
            self.layers.append(R_new.copy())

            if R_new == R_prev:
                break

            R_prev = R_new

            if it > 100:
                print("Warning: Max iterations reached")
                break

        self.R_star = R_prev
        print(f"R* computed: {len(self.R_star)} states")
        return self.R_star

    def compute_controller(self):
        """Synthesize controller on winning region"""
        if self.R_star is None:
            self.compute_R_star()

        print("Synthesizing DFA controller...")

        for idx in self.R_star:
            # Find earliest layer containing this state
            k = None
            for i, layer in enumerate(self.layers):
                if idx in layer:
                    k = i
                    break

            if k is None or k == 0:
                continue

            # Target: states in layer k-1
            target_layer = self.layers[k-1]
            valid_actions = []

            for action in self.augmented.robot.ACTIONS:
                succs = self.augmented.successors(idx, action)
                if succs and any(s in target_layer for s in succs):
```

```
                    valid_actions.append(action)

            if valid_actions:
                self.H[idx] = valid_actions

        print(f"Controller synthesized for {len(self.H)} states")
        return self.H

    def get_action(self, continuous_state, dfa_state=None):
        """
        Get control action for continuous state
        If dfa_state not provided, starts from initial DFA state
        """
        # Discretize robot state
        robot_state = self.augmented.robot._find_state(continuous_state)

        # Use initial DFA state if not provided
        if dfa_state is None:
            dfa_state = self.augmented.dfa.init_state

        # Look up augmented state
        aug_state = (dfa_state, robot_state)
        aug_idx = self.augmented.state_to_idx.get(aug_state)

        if aug_idx is None or aug_idx not in self.H:
            print(f"Warning: No action for state {aug_state}")
            return None

        # Return first valid action
        return self.H[aug_idx][0]
```

Listing 7: DFA Controller Implementation

Implementation Example

```
# Define regions in the environment
R1_states = robot.find_indices_for_interval([[1, 3], [7, 9]])  # Top-left
R2_states = robot.find_indices_for_interval([[7, 9], [7, 9]])  # Top-right
R3_states = robot.find_indices_for_interval([[4, 6], [1, 3]])  # Bottom-center
R4_states = robot.find_indices_for_interval([[2, 8], [4, 6]])  # Middle obstacle

# Create labeling function
region_labels = {}
for state in R1_states:
    region_labels[state] = 'R1'
for state in R2_states:
    region_labels[state] = 'R2'
for state in R3_states:
    region_labels[state] = 'R3'
for state in R4_states:
    region_labels[state] = 'R4'

# Define DFA for specification: (R1 XOR R2) then R3, avoid R4
dfa = DFA(
    states=['q0', 'q1', 'q2', 'q3', 'q4', 'qfail'],
    init_state='q0',
    accepting_states=['q3', 'q4'],
    transitions={
        ('q0', 'R1'): 'q1', ('q0', 'R2'): 'q2',
        ('q0', 'R3'): 'q0', ('q0', 'R4'): 'qfail',
        ('q0', 'other'): 'q0',
        ('q1', 'R1'): 'q1', ('q1', 'R2'): 'qfail',
        ('q1', 'R3'): 'q3', ('q1', 'R4'): 'qfail',
        ('q1', 'other'): 'q1',
        ('q2', 'R1'): 'qfail', ('q2', 'R2'): 'q2',
        ('q2', 'R3'): 'q4', ('q2', 'R4'): 'qfail',
        ('q2', 'other'): 'q2',
        # Absorbing states
        ('q3', 'R1'): 'q3', ('q3', 'R2'): 'q3',
        ('q3', 'R3'): 'q3', ('q3', 'R4'): 'q3',
        ('q3', 'other'): 'q3',
```

```
        ('q4', 'R1'): 'q4', ('q4', 'R2'): 'q4',
        ('q4', 'R3'): 'q4', ('q4', 'R4'): 'q4',
        ('q4', 'other'): 'q4',
        ('qfail', 'R1'): 'qfail', ('qfail', 'R2'): 'qfail',
        ('qfail', 'R3'): 'qfail', ('qfail', 'R4'): 'qfail',
        ('qfail', 'other'): 'qfail',
    }
)

# Build augmented system
augmented = AugmentedSystem(robot, dfa, region_labels)

# Synthesize controller
controller = DFAReachabilityController(augmented)
R_star = controller.compute_R_star()
H = controller.compute_controller()

# Run simulation
initial_state = [0.5, 0.5, 0.0]  # Start at bottom-left
trajectory = simulate_with_dfa(robot, controller, initial_state, steps=50)
```

Listing 8: DFA Controller Application

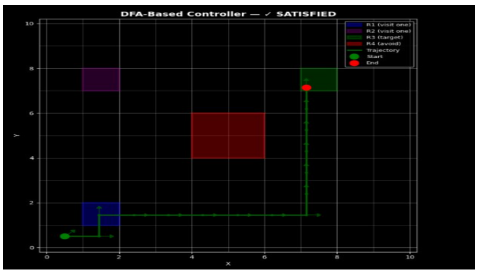

Figure 9: First Specification: Visit (R1 XOR R2) , Reach R3 and avoid R4

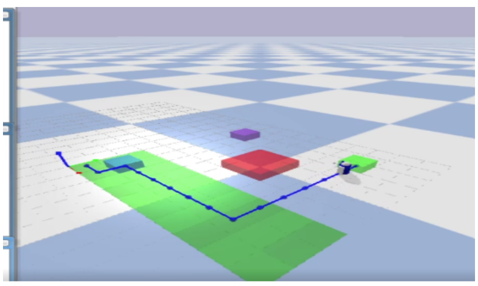

Figure 10: Visualization of the trajectory .

Advantages of the Automaton-Based Approach

- **Expressiveness**: Can encode complex temporal logic formulas
- **Modularity**: Separates robot dynamics from high-level specifications
- **Formal guarantees**: Correctness by construction
- **Reusability**: Same DFA can be used with different robot models
- **Scalability**: Complex specifications decomposed into simple automata

3.2 Other Automaton-Based Specifications

The same methodology can be applied to various complex specifications:

Sequential Visit Specification: R1 → R2 → R3 → R4 → R3

A more complex specification requiring sequential visits to multiple regions: "Visit R1, then R2, then R3, then R4, and finally return to R3."

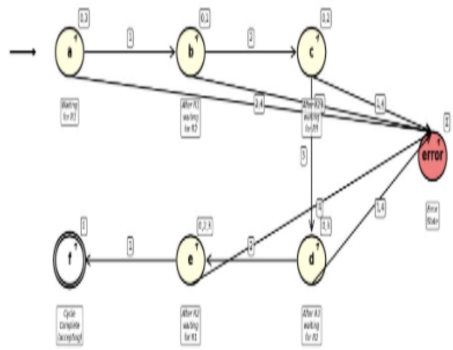

Figure 11: Sequential visit specification: R1 → R2 → R3 → R4 → R3.

Oscillatory Specification: R1 → R2 → R3 → R2 → R1

A specification requiring oscillation between regions: "Visit R1, then R2, then R3, then return to R2, and finally return to R1."

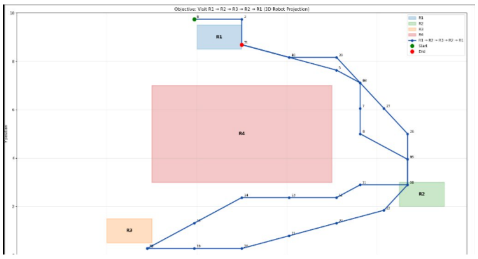

Figure 12: Oscillatory specification: R1 → R2 → R3 → R2 → R1.

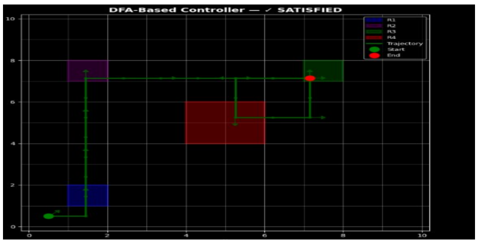

Figure 13: Visualization of trajectory for oscillatory specification.

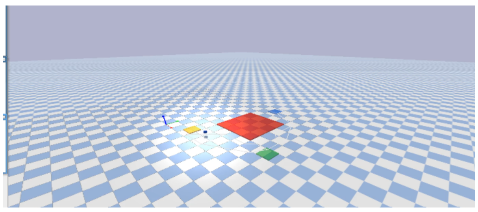

Figure 14: Another visualization of oscillatory trajectory.

4. Conclusion

Summary of Contributions

This project has demonstrated the practical application of symbolic control methods for robotic systems through several key contributions:

- **Symbolic Abstraction**: Developed a comprehensive framework for abstracting continuous robotic systems into discrete symbolic models using interval-based reachability analysis, handling nonlinear dynamics, perturbations, and constraints.
- **Safety Controller Synthesis**: Implemented Algorithm 1 for synthesizing controllers that guarantee the system remains within safe regions indefinitely, with formal correctness guarantees.
- **Reachability Controller Synthesis**: Implemented Algorithm 2 for synthesizing controllers that ensure the system eventually reaches target regions, demonstrating the duality with safety synthesis.
- **Combined Safe Reachability**: Developed Algorithm 3 for synthesizing controllers that simultaneously guarantee reaching target regions while avoiding unsafe areas, addressing practical navigation scenarios.
- **Automaton-Based Specifications**: Extended the framework to handle complex temporal logic specifications using Deterministic Finite Automata (DFA), enabling the expression of sophisticated behavioral requirements.

Key Insights and Results

1. **Scalability**: While symbolic approaches face exponential complexity with state dimension, our implementation demonstrates practical applicability for moderate-dimensional robotic systems through efficient algorithms and data structures.

2. **Formal Guarantees**: All synthesized controllers provide formal guarantees of correctness by construction, a crucial advantage over traditional control methods for safety-critical applications.

3. **Flexibility**: The automaton-based approach provides exceptional flexibility in specifying complex behaviors, from simple reachability to intricate temporal patterns and conditional sequences.

4. **Offline Computation**: The three-step methodology (abstraction, synthesis, concretization) enables all complex computations to be performed offline, resulting in simple online controllers requiring only table lookups.

5. **Practical Applicability**: Through extensive simulations and case studies, we have shown that symbolic controllers can handle realistic robotic scenarios including obstacle avoidance, target reaching, and complex temporal specifications.

6. **Modular Design**: The separation of concerns between abstraction, synthesis, and concretization allows for modular development and testing of each component independently.

7. **Extensibility**: The framework can be extended to handle more complex specifications, larger state spaces, and different robotic platforms through the well-defined interfaces between components.

8. **Educational Value**: This project serves as a comprehensive educational resource for understanding symbolic control methods, providing both theoretical foundations and practical implementations.