

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:

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

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 $\Xi = \{1, \dots, m\}$, and the abstraction function $q(x) = \xi$ maps each continuous state to its symbolic representative.

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

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

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

```

1 class RobotAbstraction:
2     def __init__(self,
3         state_intervals,
4             control_values,
5                 perturbation, delta_t):
6                     self.state_intervals =
7                         state_intervals
8                             # [(x_min, x_max), (y_min,
9                                 , y_max), (theta_min,
10                                     , theta_max)]
11                                     self.control_values =
12                                         control_values
13                                             # [(v_min, v_max), (
14                                                 omega_min, omega_max)]
15                                                 self.perturbation =
16                                                     perturbation
17                                                     # [(w1_min, w1_max), (
18                                                       w2_min, w2_max), (w3_min,
19                                                       w3_max)]
20                                                       self.delta_t = delta_t
21                                                       # sampling time (time to
22                                         change a state) = 1s
23                                         self.state_to_index = {"Out
24                                         OfGrid": -1}
25                                         # Map from state tuple to
26                                         symbolic state
27                                         self.index_to_intervals =
28                                         {}
29                                         # Map from symbolic state
30                                         to intervals
31                                         self.state_edges = []
32                                         self.discrete_x_y = 20
33                                         self.discrete_theta = 10
34                                         self.v_vals = 3
35                                         self.omega_vals = 5
36                                         self.
37                                         _create_state_mapping()
38                                         # Precompute the symbolic
39                                         state mapping
40                                         self.
41                                         compute_transitions_dict()
```

Listing 1: Robot Abstraction Class

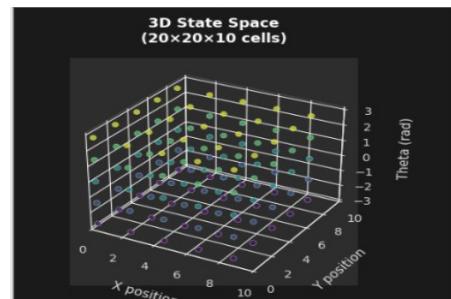


Figure 1: Visualization of the abstraction process showing continuous state space discretization into symbolic states.

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:

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

where:

- x^* : Center of state interval $X_\xi = [\underline{x}, \bar{x}]$
- w^* : Center of perturbation interval $W = [\underline{w}, \bar{w}]$
- $\delta_x = \frac{\bar{x} - \underline{x}}{2}$: Half-width of state interval
- $\delta_w = \frac{\bar{w} - \underline{w}}{2}$: 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

```

1 def dynamics(self, x_center, u,
2             w_center, D_x, D_w):
3     """Modified system dynamics
4     based on interval
5     reachability."""
6     # Nominal dynamics
7     calculation
8     nominal_dynamics = np.array([
9         x_center[0] + self.
10        delta_t *
11        (u[0] * np.cos(
12            x_center[2]) + w_center[0]),
13        x_center[1] + self.
14        delta_t *
15        (u[1] * np.sin(
16            x_center[2]) + w_center[1]),
17        x_center[2] + self.
18        delta_t *
19        (u[1] + w_center[2])
20    ])
21
22    # Calculate delta_x based on
23    # state discretization
24    delta_x = np.array([
25        (self.state_intervals
26        [0][1] -
27        self.state_intervals
28        [0][0]) /
29        (self.discrete_x_y - 1)
30        / 2,
31        (self.state_intervals
32        [1][1] -
33        self.state_intervals
34        [1][0]) /
35        (self.discrete_x_y - 1)
36        / 2,
37        (self.state_intervals
38        [2][1] -
39        self.state_intervals
40        [2][0]) /
41        (self.discrete_x_y - 1)
42        / 2,
43        (self.state_intervals
44        [3][1] -
45        self.state_intervals
46        [3][0]) /
47        (self.discrete_x_y - 1)
48        / 2,
49        (self.state_intervals
50        [4][1] -
51        self.state_intervals
52        [4][0]) /
53        (self.discrete_x_y - 1)
54        / 2,
55        (self.state_intervals
56        [5][1] -
57        self.state_intervals
58        [5][0]) /
59        (self.discrete_x_y - 1)
60        / 2,
61        (self.state_intervals
62        [6][1] -
63        self.state_intervals
64        [6][0]) /
65        (self.discrete_x_y - 1)
66        / 2,
67        (self.state_intervals
68        [7][1] -
69        self.state_intervals
70        [7][0]) /
71        (self.discrete_x_y - 1)
72        / 2,
73        (self.state_intervals
74        [8][1] -
75        self.state_intervals
76        [8][0]) /
77        (self.discrete_x_y - 1)
78        / 2,
79        (self.state_intervals
80        [9][1] -
81        self.state_intervals
82        [9][0]) /
83        (self.discrete_x_y - 1)
84        / 2,
85        (self.state_intervals
86        [10][1] -
87        self.state_intervals
88        [10][0]) /
89        (self.discrete_x_y - 1)
90        / 2,
91        (self.state_intervals
92        [11][1] -
93        self.state_intervals
94        [11][0]) /
95        (self.discrete_x_y - 1)
96        / 2,
97        (self.state_intervals
98        [12][1] -
99        self.state_intervals
100       [12][0]) /
101       (self.discrete_x_y - 1)
102       / 2,
103       (self.state_intervals
104       [13][1] -
105       self.state_intervals
106       [13][0]) /
107       (self.discrete_x_y - 1)
108       / 2,
109       (self.state_intervals
110       [14][1] -
111       self.state_intervals
112       [14][0]) /
113       (self.discrete_x_y - 1)
114       / 2,
115       (self.state_intervals
116       [15][1] -
117       self.state_intervals
118       [15][0]) /
119       (self.discrete_x_y - 1)
120       / 2,
121       (self.state_intervals
122       [16][1] -
123       self.state_intervals
124       [16][0]) /
125       (self.discrete_x_y - 1)
126       / 2,
127       (self.state_intervals
128       [17][1] -
129       self.state_intervals
130       [17][0]) /
131       (self.discrete_x_y - 1)
132       / 2,
133       (self.state_intervals
134       [18][1] -
135       self.state_intervals
136       [18][0]) /
137       (self.discrete_x_y - 1)
138       / 2,
139       (self.state_intervals
140       [19][1] -
141       self.state_intervals
142       [19][0]) /
143       (self.discrete_x_y - 1)
144       / 2,
145       (self.state_intervals
146       [20][1] -
147       self.state_intervals
148       [20][0]) /
149       (self.discrete_x_y - 1)
150       / 2,
151       (self.state_intervals
152       [21][1] -
153       self.state_intervals
154       [21][0]) /
155       (self.discrete_x_y - 1)
156       / 2,
157       (self.state_intervals
158       [22][1] -
159       self.state_intervals
160       [22][0]) /
161       (self.discrete_x_y - 1)
162       / 2,
163       (self.state_intervals
164       [23][1] -
165       self.state_intervals
166       [23][0]) /
167       (self.discrete_x_y - 1)
168       / 2,
169       (self.state_intervals
170       [24][1] -
171       self.state_intervals
172       [24][0]) /
173       (self.discrete_x_y - 1)
174       / 2,
175       (self.state_intervals
176       [25][1] -
177       self.state_intervals
178       [25][0]) /
179       (self.discrete_x_y - 1)
180       / 2,
181       (self.state_intervals
182       [26][1] -
183       self.state_intervals
184       [26][0]) /
185       (self.discrete_x_y - 1)
186       / 2,
187       (self.state_intervals
188       [27][1] -
189       self.state_intervals
190       [27][0]) /
191       (self.discrete_x_y - 1)
192       / 2,
193       (self.state_intervals
194       [28][1] -
195       self.state_intervals
196       [28][0]) /
197       (self.discrete_x_y - 1)
198       / 2,
199       (self.state_intervals
200       [29][1] -
201       self.state_intervals
202       [29][0]) /
203       (self.discrete_x_y - 1)
204       / 2,
205       (self.state_intervals
206       [30][1] -
207       self.state_intervals
208       [30][0]) /
209       (self.discrete_x_y - 1)
210       / 2,
211       (self.state_intervals
212       [31][1] -
213       self.state_intervals
214       [31][0]) /
215       (self.discrete_x_y - 1)
216       / 2,
217       (self.state_intervals
218       [32][1] -
219       self.state_intervals
220       [32][0]) /
221       (self.discrete_x_y - 1)
222       / 2,
223       (self.state_intervals
224       [33][1] -
225       self.state_intervals
226       [33][0]) /
227       (self.discrete_x_y - 1)
228       / 2,
229       (self.state_intervals
230       [34][1] -
231       self.state_intervals
232       [34][0]) /
233       (self.discrete_x_y - 1)
234       / 2,
235       (self.state_intervals
236       [35][1] -
237       self.state_intervals
238       [35][0]) /
239       (self.discrete_x_y - 1)
240       / 2,
241       (self.state_intervals
242       [36][1] -
243       self.state_intervals
244       [36][0]) /
245       (self.discrete_x_y - 1)
246       / 2,
247       (self.state_intervals
248       [37][1] -
249       self.state_intervals
250       [37][0]) /
251       (self.discrete_x_y - 1)
252       / 2,
253       (self.state_intervals
254       [38][1] -
255       self.state_intervals
256       [38][0]) /
257       (self.discrete_x_y - 1)
258       / 2,
259       (self.state_intervals
260       [39][1] -
261       self.state_intervals
262       [39][0]) /
263       (self.discrete_x_y - 1)
264       / 2,
265       (self.state_intervals
266       [40][1] -
267       self.state_intervals
268       [40][0]) /
269       (self.discrete_x_y - 1)
270       / 2,
271       (self.state_intervals
272       [41][1] -
273       self.state_intervals
274       [41][0]) /
275       (self.discrete_x_y - 1)
276       / 2,
277       (self.state_intervals
278       [42][1] -
279       self.state_intervals
280       [42][0]) /
281       (self.discrete_x_y - 1)
282       / 2,
283       (self.state_intervals
284       [43][1] -
285       self.state_intervals
286       [43][0]) /
287       (self.discrete_x_y - 1)
288       / 2,
289       (self.state_intervals
290       [44][1] -
291       self.state_intervals
292       [44][0]) /
293       (self.discrete_x_y - 1)
294       / 2,
295       (self.state_intervals
296       [45][1] -
297       self.state_intervals
298       [45][0]) /
299       (self.discrete_x_y - 1)
300       / 2,
301       (self.state_intervals
302       [46][1] -
303       self.state_intervals
304       [46][0]) /
305       (self.discrete_x_y - 1)
306       / 2,
307       (self.state_intervals
308       [47][1] -
309       self.state_intervals
310       [47][0]) /
311       (self.discrete_x_y - 1)
312       / 2,
313       (self.state_intervals
314       [48][1] -
315       self.state_intervals
316       [48][0]) /
317       (self.discrete_x_y - 1)
318       / 2,
319       (self.state_intervals
320       [49][1] -
321       self.state_intervals
322       [49][0]) /
323       (self.discrete_x_y - 1)
324       / 2,
325       (self.state_intervals
326       [50][1] -
327       self.state_intervals
328       [50][0]) /
329       (self.discrete_x_y - 1)
330       / 2,
331       (self.state_intervals
332       [51][1] -
333       self.state_intervals
334       [51][0]) /
335       (self.discrete_x_y - 1)
336       / 2,
337       (self.state_intervals
338       [52][1] -
339       self.state_intervals
340       [52][0]) /
341       (self.discrete_x_y - 1)
342       / 2,
343       (self.state_intervals
344       [53][1] -
345       self.state_intervals
346       [53][0]) /
347       (self.discrete_x_y - 1)
348       / 2,
349       (self.state_intervals
350       [54][1] -
351       self.state_intervals
352       [54][0]) /
353       (self.discrete_x_y - 1)
354       / 2,
355       (self.state_intervals
356       [55][1] -
357       self.state_intervals
358       [55][0]) /
359       (self.discrete_x_y - 1)
360       / 2,
361       (self.state_intervals
362       [56][1] -
363       self.state_intervals
364       [56][0]) /
365       (self.discrete_x_y - 1)
366       / 2,
367       (self.state_intervals
368       [57][1] -
369       self.state_intervals
370       [57][0]) /
371       (self.discrete_x_y - 1)
372       / 2,
373       (self.state_intervals
374       [58][1] -
375       self.state_intervals
376       [58][0]) /
377       (self.discrete_x_y - 1)
378       / 2,
379       (self.state_intervals
380       [59][1] -
381       self.state_intervals
382       [59][0]) /
383       (self.discrete_x_y - 1)
384       / 2,
385       (self.state_intervals
386       [60][1] -
387       self.state_intervals
388       [60][0]) /
389       (self.discrete_x_y - 1)
390       / 2,
391       (self.state_intervals
392       [61][1] -
393       self.state_intervals
394       [61][0]) /
395       (self.discrete_x_y - 1)
396       / 2,
397       (self.state_intervals
398       [62][1] -
399       self.state_intervals
400       [62][0]) /
401       (self.discrete_x_y - 1)
402       / 2,
403       (self.state_intervals
404       [63][1] -
405       self.state_intervals
406       [63][0]) /
407       (self.discrete_x_y - 1)
408       / 2,
409       (self.state_intervals
410       [64][1] -
411       self.state_intervals
412       [64][0]) /
413       (self.discrete_x_y - 1)
414       / 2,
415       (self.state_intervals
416       [65][1] -
417       self.state_intervals
418       [65][0]) /
419       (self.discrete_x_y - 1)
420       / 2,
421       (self.state_intervals
422       [66][1] -
423       self.state_intervals
424       [66][0]) /
425       (self.discrete_x_y - 1)
426       / 2,
427       (self.state_intervals
428       [67][1] -
429       self.state_intervals
430       [67][0]) /
431       (self.discrete_x_y - 1)
432       / 2,
433       (self.state_intervals
434       [68][1] -
435       self.state_intervals
436       [68][0]) /
437       (self.discrete_x_y - 1)
438       / 2,
439       (self.state_intervals
440       [69][1] -
441       self.state_intervals
442       [69][0]) /
443       (self.discrete_x_y - 1)
444       / 2,
445       (self.state_intervals
446       [70][1] -
447       self.state_intervals
448       [70][0]) /
449       (self.discrete_x_y - 1)
450       / 2,
451       (self.state_intervals
452       [71][1] -
453       self.state_intervals
454       [71][0]) /
455       (self.discrete_x_y - 1)
456       / 2,
457       (self.state_intervals
458       [72][1] -
459       self.state_intervals
460       [72][0]) /
461       (self.discrete_x_y - 1)
462       / 2,
463       (self.state_intervals
464       [73][1] -
465       self.state_intervals
466       [73][0]) /
467       (self.discrete_x_y - 1)
468       / 2,
469       (self.state_intervals
470       [74][1] -
471       self.state_intervals
472       [74][0]) /
473       (self.discrete_x_y - 1)
474       / 2,
475       (self.state_intervals
476       [75][1] -
477       self.state_intervals
478       [75][0]) /
479       (self.discrete_x_y - 1)
480       / 2,
481       (self.state_intervals
482       [76][1] -
483       self.state_intervals
484       [76][0]) /
485       (self.discrete_x_y - 1)
486       / 2,
487       (self.state_intervals
488       [77][1] -
489       self.state_intervals
490       [77][0]) /
491       (self.discrete_x_y - 1)
492       / 2,
493       (self.state_intervals
494       [78][1] -
495       self.state_intervals
496       [78][0]) /
497       (self.discrete_x_y - 1)
498       / 2,
499       (self.state_intervals
500       [79][1] -
501       self.state_intervals
502       [79][0]) /
503       (self.discrete_x_y - 1)
504       / 2,
505       (self.state_intervals
506       [80][1] -
507       self.state_intervals
508       [80][0]) /
509       (self.discrete_x_y - 1)
510       / 2,
511       (self.state_intervals
512       [81][1] -
513       self.state_intervals
514       [81][0]) /
515       (self.discrete_x_y - 1)
516       / 2,
517       (self.state_intervals
518       [82][1] -
519       self.state_intervals
520       [82][0]) /
521       (self.discrete_x_y - 1)
522       / 2,
523       (self.state_intervals
524       [83][1] -
525       self.state_intervals
526       [83][0]) /
527       (self.discrete_x_y - 1)
528       / 2,
529       (self.state_intervals
530       [84][1] -
531       self.state_intervals
532       [84][0]) /
533       (self.discrete_x_y - 1)
534       / 2,
535       (self.state_intervals
536       [85][1] -
537       self.state_intervals
538       [85][0]) /
539       (self.discrete_x_y - 1)
540       / 2,
541       (self.state_intervals
542       [86][1] -
543       self.state_intervals
544       [86][0]) /
545       (self.discrete_x_y - 1)
546       / 2,
547       (self.state_intervals
548       [87][1] -
549       self.state_intervals
550       [87][0]) /
551       (self.discrete_x_y - 1)
552       / 2,
553       (self.state_intervals
554       [88][1] -
555       self.state_intervals
556       [88][0]) /
557       (self.discrete_x_y - 1)
558       / 2,
559       (self.state_intervals
560       [89][1] -
561       self.state_intervals
562       [89][0]) /
563       (self.discrete_x_y - 1)
564       / 2,
565       (self.state_intervals
566       [90][1] -
567       self.state_intervals
568       [90][0]) /
569       (self.discrete_x_y - 1)
570       / 2,
571       (self.state_intervals
572       [91][1] -
573       self.state_intervals
574       [91][0]) /
575       (self.discrete_x_y - 1)
576       / 2,
577       (self.state_intervals
578       [92][1] -
579       self.state_intervals
580       [92][0]) /
581       (self.discrete_x_y - 1)
582       / 2,
583       (self.state_intervals
584       [93][1] -
585       self.state_intervals
586       [93][0]) /
587       (self.discrete_x_y - 1)
588       / 2,
589       (self.state_intervals
590       [94][1] -
591       self.state_intervals
592       [94][0]) /
593       (self.discrete_x_y - 1)
594       / 2,
595       (self.state_intervals
596       [95][1] -
597       self.state_intervals
598       [95][0]) /
599       (self.discrete_x_y - 1)
600       / 2,
601       (self.state_intervals
602       [96][1] -
603       self.state_intervals
604       [96][0]) /
605       (self.discrete_x_y - 1)
606       / 2,
607       (self.state_intervals
608       [97][1] -
609       self.state_intervals
610       [97][0]) /
611       (self.discrete_x_y - 1)
612       / 2,
613       (self.state_intervals
614       [98][1] -
615       self.state_intervals
616       [98][0]) /
617       (self.discrete_x_y - 1)
618       / 2,
619       (self.state_intervals
620       [99][1] -
621       self.state_intervals
622       [99][0]) /
623       (self.discrete_x_y - 1)
624       / 2,
625       (self.state_intervals
626       [100][1] -
627       self.state_intervals
628       [100][0]) /
629       (self.discrete_x_y - 1)
630       / 2,
631       (self.state_intervals
632       [101][1] -
633       self.state_intervals
634       [101][0]) /
635       (self.discrete_x_y - 1)
636       / 2,
637       (self.state_intervals
638       [102][1] -
639       self.state_intervals
640       [102][0]) /
641       (self.discrete_x_y - 1)
642       / 2,
643       (self.state_intervals
644       [103][1] -
645       self.state_intervals
646       [103][0]) /
647       (self.discrete_x_y - 1)
648       / 2,
649       (self.state_intervals
650       [104][1] -
651       self.state_intervals
652       [104][0]) /
653       (self.discrete_x_y - 1)
654       / 2,
655       (self.state_intervals
656       [105][1] -
657       self.state_intervals
658       [105][0]) /
659       (self.discrete_x_y - 1)
660       / 2,
661       (self.state_intervals
662       [106][1] -
663       self.state_intervals
664       [106][0]) /
665       (self.discrete_x_y - 1)
666       / 2,
667       (self.state_intervals
668       [107][1] -
669       self.state_intervals
670       [107][0]) /
671       (self.discrete_x_y - 1)
672       / 2,
673       (self.state_intervals
674       [108][1] -
675       self.state_intervals
676       [108][0]) /
677       (self.discrete_x_y - 1)
678       / 2,
679       (self.state_intervals
680       [109][1] -
681       self.state_intervals
682       [109][0]) /
683       (self.discrete_x_y - 1)
684       / 2,
685       (self.state_intervals
686       [110][1] -
687       self.state_intervals
688       [110][0]) /
689       (self.discrete_x_y - 1)
690       / 2,
691       (self.state_intervals
692       [111][1] -
693       self.state_intervals
694       [111][0]) /
695       (self.discrete_x_y - 1)
696       / 2,
697       (self.state_intervals
698       [112][1] -
699       self.state_intervals
700       [112][0]) /
701       (self.discrete_x_y - 1)
702       / 2,
703       (self.state_intervals
704       [113][1] -
705       self.state_intervals
706       [113][0]) /
707       (self.discrete_x_y - 1)
708       / 2,
709       (self.state_intervals
710       [114][1] -
711       self.state_intervals
712       [114][0]) /
713       (self.discrete_x_y - 1)
714       / 2,
715       (self.state_intervals
716       [115][1] -
717       self.state_intervals
718       [115][0]) /
719       (self.discrete_x_y - 1)
720       / 2,
721       (self.state_intervals
722       [116][1] -
723       self.state_intervals
724       [116][0]) /
725       (self.discrete_x_y - 1)
726       / 2,
727       (self.state_intervals
728       [117][1] -
729       self.state_intervals
730       [117][0]) /
731       (self.discrete_x_y - 1)
732       / 2,
733       (self.state_intervals
734       [118][1] -
735       self.state_intervals
736       [118][0]) /
737       (self.discrete_x_y - 1)
738       / 2,
739       (self.state_intervals
740       [119][1] -
741       self.state_intervals
742       [119][0]) /
743       (self.discrete_x_y - 1)
744       / 2,
745       (self.state_intervals
746       [120][1] -
747       self.state_intervals
748       [120][0]) /
749       (self.discrete_x_y - 1)
750       / 2,
751       (self.state_intervals
752       [121][1] -
753       self.state_intervals
754       [121][0]) /
755       (self.discrete_x_y - 1)
756       / 2,
757       (self.state_intervals
758       [122][1] -
759       self.state_intervals
760       [122][0]) /
761       (self.discrete_x_y - 1)
762       / 2,
763       (self.state_intervals
764       [123][1] -
765       self.state_intervals
766       [123][0]) /
767       (self.discrete_x_y - 1)
768       / 2,
769       (self.state_intervals
770       [124][1] -
771       self.state_intervals
772       [124][0]) /
773       (self.discrete_x_y - 1)
774       / 2,
775       (self.state_intervals
776       [125][1] -
777       self.state_intervals
778       [125][0]) /
779       (self.discrete_x_y - 1)
780       / 2,
781       (self.state_intervals
782       [126][1] -
783       self.state_intervals
784       [126][0]) /
785       (self.discrete_x_y - 1)
786       / 2,
787       (self.state_intervals
788       [127][1] -
789       self.state_intervals
790       [127][0]) /
791       (self.discrete_x_y - 1)
792       / 2,
793       (self.state_intervals
794       [128][1] -
795       self.state_intervals
796       [128][0]) /
797       (self.discrete_x_y - 1)
798       / 2,
799       (self.state_intervals
800       [129][1] -
801       self.state_intervals
802       [129][0]) /
803       (self.discrete_x_y - 1)
804       / 2,
805       (self.state_intervals
806       [130][1] -
807       self.state_intervals
808       [130][0]) /
809       (self.discrete_x_y - 1)
810       / 2,
811       (self.state_intervals
812       [131][1] -
813       self.state_intervals
814       [131][0]) /
815       (self.discrete_x_y - 1)
816       / 2,
817       (self.state_intervals
818       [132][1] -
819       self.state_intervals
820       [132][0]) /
821       (self.discrete_x_y - 1)
822       / 2,
823       (self.state_intervals
824       [133][1] -
825       self.state_intervals
826       [133][0]) /
827       (self.discrete_x_y - 1)
828       / 2,
829       (self.state_intervals
830       [134][1] -
831       self.state_intervals
832       [134][0]) /
833       (self.discrete_x_y - 1)
834       / 2,
835       (self.state_intervals
836       [135][1] -
837       self.state_intervals
838       [135][0]) /
839       (self.discrete_x_y - 1)
840       / 2,
841       (self.state_intervals
842       [136][1] -
843       self.state_intervals
844       [136][0]) /
845       (self.discrete_x_y - 1)
846       / 2,
847       (self.state_intervals
848       [137][1] -
849       self.state_intervals
850       [137][0]) /
851       (self.discrete_x_y - 1)
852       / 2,
853       (self.state_intervals
854       [138][1] -
855       self.state_intervals
856       [138][0]) /
857       (self.discrete_x_y - 1)
858       / 2,
859       (self.state_intervals
860       [139][1] -
861       self.state_intervals
862       [139][0]) /
863       (self.discrete_x_y - 1)
864       / 2,
865       (self.state_intervals
866       [140][1] -
867       self.state_intervals
868       [140][0]) /
869       (self.discrete_x_y - 1)
870       / 2,
871       (self.state_intervals
872       [141][1] -
873       self.state_intervals
874       [141][0]) /
875       (self.discrete_x_y - 1)
876       / 2,
877       (self.state_intervals
878       [142][1] -
879       self.state_intervals
880       [142][0]) /
881       (self.discrete_x_y - 1)
882       / 2,
883       (self.state_intervals
884       [143][1] -
885       self.state_intervals
886       [143][0]) /
887       (self.discrete_x_y - 1)
888       / 2,
889       (self.state_intervals
890       [144][1] -
891       self.state_intervals
892       [144][0]) /
893       (self.discrete_x_y - 1)
894       / 2,
895       (self.state_intervals
896       [145][1] -
897       self.state_intervals
898       [145][0]) /
899       (self.discrete_x_y - 1)
900       / 2,
901       (self.state_intervals
902       [146][1] -
903       self.state_intervals
904       [146][0]) /
905       (self.discrete_x_y - 1)
906       / 2,
907       (self.state_intervals
908       [147][1] -
909       self.state_intervals
910       [147][0]) /
911       (self.discrete_x_y - 1)
912       / 2,
913       (self.state_intervals
914       [148][1] -
915       self.state_intervals
916       [148][0]) /
917       (self.discrete_x_y - 1)
918       / 2,
919       (self.state_intervals
920       [149][1] -
921       self.state_intervals
922       [149][0]) /
923       (self.discrete_x_y - 1)
924       / 2,
925       (self.state_intervals
926       [150][1] -
927       self.state_intervals
928       [150][0]) /
929       (self.discrete_x_y - 1)
930       / 2,
931       (self.state_intervals
932       [151][1] -
933       self.state_intervals
934       [151][0]) /
935       (self.discrete_x_y - 1)
936       / 2,
937       (self.state_intervals
938       [152][1] -
939       self.state_intervals
940       [152][0]) /
941       (self.discrete_x_y - 1)
942       / 2,
943       (self.state_intervals
944       [153][1] -
945       self.state_intervals
946       [153][0]) /
947       (self.discrete_x_y - 1)
948       / 2,
949       (self.state_intervals
950       [154][1] -
951       self.state_intervals
952       [154][0]) /
953       (self.discrete_x_y - 1)
954       / 2,
955       (self.state_intervals
956       [155][1] -
957       self.state_intervals
958       [155][0]) /
959       (self.discrete_x_y - 1)
960       / 2,
961       (self.state_intervals
962       [156][1] -
963       self.state_intervals
964       [156][0]) /
965       (self.discrete_x_y - 1)
966       / 2,
967       (self.state_intervals
968       [157][1] -
969       self.state_intervals
970       [157][0]) /
971       (self.discrete_x_y - 1)
972       / 2,
973       (self.state_intervals
974       [158][1] -
975       self.state_intervals
976       [158][0]) /
977       (self.discrete_x_y - 1)
978       / 2,
979       (self.state_intervals
980       [159][1] -
981       self.state_intervals
982       [159][0]) /
983       (self.discrete_x_y - 1)
984       / 2,
985       (self.state_intervals
986       [160][1] -
987       self.state_intervals
988       [160][0]) /
989       (self.discrete_x_y - 1)
990       / 2,
991       (self.state_intervals
992       [161][1] -
993       self.state_intervals
994       [161][0]) /
995       (self.discrete_x_y - 1)
996       / 2,
997       (self.state_intervals
998       [162][1] -
999       self.state_intervals
1000      [162][0]) /
1001      (self.discrete_x_y - 1)
1002      / 2,
1003      (self.state_intervals
1004      [163][1] -
1005      self.state_intervals
1006      [163][0]) /
1007      (self.discrete_x_y - 1)
1008      / 2,
1009      (self.state_intervals
1010      [164][1] -
1011      self.state_intervals
1012      [164][0]) /
1013      (self.discrete_x_y - 1)
1014      / 2,
1015      (self.state_intervals
1016      [165][1] -
1017      self.state_intervals
1018      [165][0]) /
1019      (self.discrete_x_y - 1)
1020      / 2,
1021      (self.state_intervals
1022      [166][1] -
1023      self.state_intervals
1024      [166][0]) /
1025      (self.discrete_x_y - 1)
1026      / 2,
1027      (self.state_intervals
1028      [167][1] -
1029      self.state_intervals
1030      [167][0]) /
1031      (self.discrete_x_y - 1)
1032      / 2,
1033      (self.state_intervals
1034      [168][1] -
1035      self.state_intervals
1036      [168][0]) /
1037      (self.dis
```

Listing 2: Dynamics with Reachability Analysis

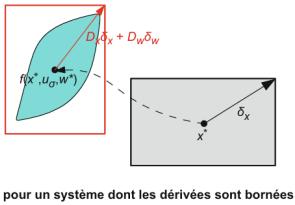


Figure 2: Interval-based reachability analysis for transition computation.

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 $\text{Pre}(R)$, defined as:

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

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

Python Implementation

```

1  class SymbolicControllerSynthesis
2      :
3          def __init__(self,
4              transitions, safety_states):
5                  """
6                      Initialize the symbolic
7                      controller synthesis.
8
9                  Parameters:
10                 - transitions: List of
11                     transitions in the form
12                     [(state, control,
13                     successors), ...].
13                 - safety_states: Set of
14                     safe states ( $Q_s$ ).
15
16                 """
17
18             self.transitions = self.
19             _process_transitions(
20                 transitions)
21
22                 # Convert list to
23                 dictionary
24                 self.safety_states =
25                 safety_states
26
27             def _process_transitions(self
28                 , transitions):
29
30                 """
31                     Convert the list of
32                     transitions into a dictionary
33                     format.
34
35                 Parameters:
36                 - transitions: List of
37                     transitions

```

```

21         [(state, control,
22          successors), ...].
23
24         Returns:
25         - A dictionary mapping (state, control) -> set(successors).
26         """
27
28         transition_dict =
29         defaultdict(set)
30         for state, control,
31         successors in transitions:
32             transition_dict[(state, control)] = successors
33         return dict(
34         transition_dict)
35
36     def pre(self, R):
37         """
38             Compute the predecessor operator Pre(R), considering
39             only
40             states with valid
41             transitions.
42
43             Parameters:
44             - R: Set of states (current safe set).
45
46             Returns:
47             - Set of states that can
48             transition to R.
49             """
50
51             pre_states = set()
52             for (state, control),
53             successors in self.
54             transitions.items():
55                 # Check if all
56                 successors are in R
57                 if successors and
58                 successors.issubset(R):
59                     pre_states.add(
60                     state)
61             return pre_states
62
63     def compute_safe_controller(
64         self):
65         """
66             Compute the maximal safe
67             set ( $R^*$ ).
68
69             Returns:
70             -  $R^*$ : Maximal set of safe
71             states.
72             """
73
74             R = self.safety_states.
75             copy()
76             Q_s = R
77
78             while True:
79                 R_next = Q_s.
80                 intersection(self.pre(R))
81                 if R_next == R:
82                     break
83                 R = R_next
84             return R
85
86     def synthesize_controller(
87         self):
88         """
89             Synthesize a safe
90             controller.
91
92             Returns:
93             -  $R^*$ : Maximal safe set.
94             - Controller: Mapping
95             from states to safe controls.
96             -  $Q_0$ : Set of valid
97             initial states.
98             """
99
100            R_star = self.
101            compute_safe_controller()
102            controller = defaultdict(
103                set)
104            Q_0 = set()
105
106            for (state, control),
107            successors in self.
108            transitions.items():
109                # Add controls only
110                if all successors are in  $R^*$ 
111                    if state in R_star
112                    and successors.issubset(
113                        R_star):
114                        controller[state].
115                        add(control)
116
117                        #  $Q_0$ : States in  $R^*$ 
118                        with at least one valid
119                        control
120                            for state in R_star:
121                                if state in
122                                controller:
123                                    Q_0.add(state)
124
125
126            return R_star, controller
127            , 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 $h : \Xi \rightarrow \Sigma$ 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 $h : \Xi \rightarrow \Sigma$ 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.

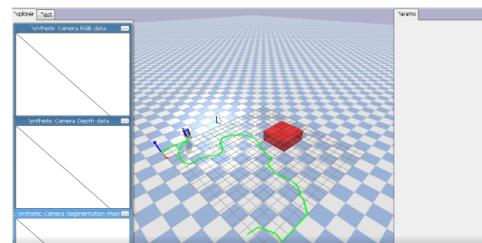


Figure 3: Safety controller application: The robot avoids the unsafe region (red) while navigating to the target (green). The blue arrows represent safe control actions synthesized by the symbolic controller.

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:

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

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**
 - $R_{k+1} \leftarrow Q_e \cup \text{Pre}(R_k)$
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 $h : \Xi \rightarrow$

Σ 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:

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

Python Implementation: Reachability Controller

```

1  class SymbolicReachabilityController:
2      """
3          Symbolic reachability
4          controller based on
5          Algorithm 2 and Theorem 4
6      """
7
8      def __init__(self, robot,
9                  target_states):
10         self.robot = robot
11         self.target_states =
12             target_states # Q_e - target
13             states
14         self.R_star = None # Fixed point R*
15         self.R_sequence = [] # R_k sequence for control
16         self.H = {} # Multivalued controller
17
18     def compute_R_star(self):
19         """
20             Algorithm 2: Computation
21             of fixed point R* for
22             reachability
23             R_{k+1} = Q_e      Pre(R_k)
24         """
25         print("Computing R* ("
26             "reachability fixed point)... "
27         )
28
29         # R_0 = Q_e (target
30         states)
31         R_prev = set(self.
32             target_states)
33         self.R_sequence = [R_prev
34             .copy()]
35         iteration = 0
36
37         while True:
38             iteration += 1

```

```

28             #  $R_{\{k+1\}} = Q_e$ 
29     Pre( $R_k$ )
30          $R_{\text{new}} = \text{set}(\text{self}.$ 
31         target_states) # Start with
32          $Q_e$ 
33          $R_{\text{new}}.\text{update}(\text{self}.\text{pre}$ 
34         ( $R_{\text{prev}}$ )) # Union with Pre(
35          $R_k$ ))
36
37             print(f"Iteration {iteration}: |R| = {len(R_new)}")
38             self.R_sequence.
39             append( $R_{\text{new}}.\text{copy}()$ )
40
41             # Fixed point
42             condition
43             if  $R_{\text{new}} == R_{\text{prev}}$ :
44                 break
45
46              $R_{\text{prev}} = R_{\text{new}}$ 
47
48             # Safety: avoid
49             infinite loops
50             if iteration > 100:
51                 print("Warning: Maximum iterations reached")
52                 break
53
54             self.R_star =  $R_{\text{new}}$ 
55             print(f" $R^*$  computation completed: {len(self.R_star)}
56             reachable states")
57             return self.R_star
58
59     def pre(self, R):
60         """
61             Pre( $R$ ) operator - states
62             that can reach  $R$  in one step
63         """
64         pre_states = set()
65
66             # For each possible state
67             , check if it can reach  $R$ 
68             for state in range(1, len
69             (self.robot.
70             index_to_intervals) + 1):
71                 if self.
72                 _can_reach_set(state, R):
73                     pre_states.add(
74                     state)
75
76             return pre_states
77
78     def _can_reach_set(self,
79     state, target_set):
80         """
81             Checks if a state can
82             reach the target set in one
83             step
84             """
85             possible_actions = self.
86             _get_possible_actions(state)
87
88             for action in
89             possible_actions:
90                 next_states = self.
91                 _get_next_states(state,
92                 action)
93
94                 # Check if at least
95                 one successor is in target
96                 set
97                 for next_state in
98                 next_states:
99                     if next_state in
100                     target_set:
101                         return True
102
103             return False
104
105     def
106     compute_reachability_controller
107     (self):
108         """
109             Computes reachability
110             controller according to
111             Theorem 4
112              $k \in N, R_{\{k+1\}}, g(\cdot, h(\cdot))$ 
113              $R_k$ 
114         """
115         if self.R_star is None:
116             self.compute_R_star()
117
118             print("Computing
119             reachability controller...")
120
121             # For each state in  $R^*$ ,
122             find actions leading to
123             target
124             for state in self.R_star:
125                 # Find smallest  $k$ 
126                 such that state in  $R_k$ 
127                 k_level = None
128                 for k,  $R_k$  in
129                 enumerate(self.R_sequence):
130                     if state in  $R_k$ :
131                         k_level = k
132                         break
133
134                     if k_level is not
135                     None and k_level > 0:
136                         # We want to
137                         reach  $R_{\{k-1\}}$ 

```

```

101         target_set = self
102             .R_sequence[k_level - 1]
103                 valid_actions =
104                     []
105
106             possible_actions
107                 = self._get_possible_actions(
108                     state)
109                     for action in
110                         possible_actions:
111                             next_states =
112                                 self._get_next_states(state,
113                                     action)
114
115                             # Check if at
116                             least one successor is in
117                             target_set
118
119             reaches_target = any(ns in
120                 target_set for ns in
121                 next_states)
122
123             if
124                 reaches_target and
125                 next_states:
126                     # Theorem
127                     4 condition
128
129             valid_actions.append(action)
130
131         self.H[state] =
132             valid_actions
133
134             print(f"Reachability
135                 controller computed for {len(
136                     self.H)} states")
137             return self.H
138
139     def get_reachability_action(
140         self, continuous_state):
141         """
142             Implements the
143             concretized controller for
144             reachability
145             """
146
147             # Discretization
148             discrete_state = self.
149             robot._find_state(np.array(
150                 continuous_state))
151
152             # Check if state is in R*
153             if discrete_state in self
154                 .H and self.H[discrete_state
155                 ]:
156                 # Choose a valid
157                 action
158                 return self.H[
159                     discrete_state][0]
160
161         else:
162             # Default action (
163                 exploration)
164             print(f"Warning: No
165                 reachability action for state
166                 {discrete_state}")
167             return (1.0, 0.0) #
168             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 $R_{k+1} = Q_e \cup \text{Pre}(R_k)$
- `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:

```

1 # Define target region
2 target_region = [[4.0, 6.0],
3                  [4.0, 6.0]] # x: [4,6], y:
4                  [4,6]
4
# Find symbolic states in target
region

```

```

5 target_states = []
6 for state_idx, intervals in robot
7     .index_to_intervals.items():
8         x_center = (intervals[0][0] +
9             intervals[0][1]) / 2
10        y_center = (intervals[1][0] +
11            intervals[1][1]) / 2
12        if target_region[0][0] <=
13            target_region[0][1] and \
14                target_region[1][0] <=
15                target_region[1][1]:
16                    target_states.append(
17                        state_idx)
18
19 # Create and compute reachability
20 # controller
21 reachability_controller =
22     SymbolicReachabilityController(
23
24     robot, target_states)
25 R_star = reachability_controller.
26     compute_R_star()
27 controller =
28     reachability_controller.
29     compute_reachability_controller(
30
31
32 # Run simulation
33 initial_state = [1.0, 1.0, 0.0]
34     # Start at (1,1)
35 trajectory, reached =
36     run_reachability_simulation(
37
38         robot,
39         reachability_controller,
40         initial_state, target_region,
41         steps=100
42 )

```

Listing 5: Reachability Controller Application

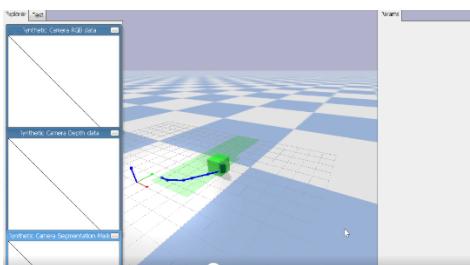


Figure 4: Reachability specification visualization: The robot must reach the target region (green) from various initial positions (blue).



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** $k = 1, 2, \dots$ **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

```
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)
```

```

10         self.unsafe_states = set(
11             unsafe_states)
12             self.R_star = set() # Safe reachable states
13             self.level = {} # BFS level (0 for target states)
14             self.H = {} # Controller: state -> list of actions
15
16                 # Precompute transitions for efficiency
17                 self.actions = self.robot.sample_actions()
18                 self.transitions = {}
19                 for state in self.robot.index_to_intervals.keys():
20                     for a in self.actions:
21                         self.transitions[(state, a)] = \
22                             self.robot.successors_overapprox(state,
23 a, samples=3)
24
25         def compute_R_star(self):
26             """
27                 Backward BFS from target states while avoiding unsafe region
28             """
29                 print("Computing safe R*(fast BFS)...")
30                 frontier = set(self.target_states)
31                 self.R_star = set(frontier)
32
33                 for s in frontier:
34                     self.level[s] = 0
35
36                 max_iters = 40
37                 it = 0
38
39                 while frontier and it < max_iters:
40                     it += 1
41                     new_frontier = set()
42
43                     for state in self.robot.index_to_intervals.keys():
44                         if state in self.R_star: # Already in
45                             continue
46                         if state in self.unsafe_states: # Avoid unsafe

```

```

44         continue
45
46             # Check if state
47             can reach frontier safely
48             for a in self.
49             actions:
50                 succ = self.
51                 transitions[(state, a)]
52                 if not succ:
53                     continue
54
55             reaches_frontier = any(ns in
56             frontier for ns in succ)
57             avoids_unsafe =
58             all(ns not in self.
59             unsafe_states
60
61             for ns in succ)
62
63             if
64             reaches_frontier and
65             avoids_unsafe:
66                 self.
67                 R_star.add(state)
68                 self.
69                 level[state] = self.
70                 _min_level_in(succ) + 1
71
72                 new_frontier.add(state)
73
74                 break
75
76                 frontier =
77                 new_frontier
78
79                 print(f"Iteration {it}
80                 ): |R*| = {len(self.R_star)}")
81
82
83             print(f"Safe R* done.
84             Total states: {len(self.
85             R_star)}")
86             return self.R_star
87
88     def
89     compute_safe_reachability_controller
90     (self):
91
92         """
93             Synthesize controller
94             that reduces level
95             while avoiding unsafe
96             region
97             """
98
99             if not self.R_star:
100                 self.compute_R_star()
101
102                 print("Computing safe
103                 controller policy...")
104
105
106             for state in self.R_star:
107                 valid_actions = []
108                 best_level = float('
109                 inf')
110
111                 for a in self.actions
112                 :
113                     succ = self.
114                     transitions[(state, a)]
115                     if not succ:
116                         continue
117
118                     avoids_unsafe =
119                     all(ns not in self.
120                     unsafe_states
121
122                     for ns in succ)
123                     succ_levels = [
124                     self.level.get(ns, float('inf
125                     '))
126
127                     for ns in succ]
128                     min_succ_level =
129                     min(succ_levels) if
130                     succ_levels else float('inf
131
132                     # Choose actions
133                     that reduce level while
134                     avoiding unsafe region
135                     if avoids_unsafe
136                     and min_succ_level < self.
137                     level.get(state,
138
139                     float('inf')):
140                         if
141                         min_succ_level < best_level:
142
143                             valid_actions = [a]
144
145                             best_level = min_succ_level
146                             elif
147                             min_succ_level == best_level:
148
149                             valid_actions.append(a)
150
151                             self.H[state] =
152                             valid_actions
153
154                             print(f"Policy computed
155                             for {len(self.H)} states.")
156
157                             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.

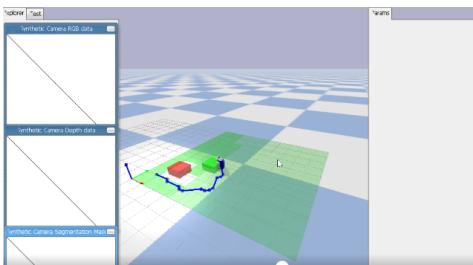


Figure 7: Safe reachability trajectory: The robot successfully reaches target while avoiding obstacles.

3. Automaton-Based Specifications

3.1 Methodology for Automaton-Based Specifications

Automaton-based specifications allow us to express complex temporal logic prop-

erties using finite automata. The approach follows these key steps:

Step 1: Define Labeling Function and DFA

Consider a labeling function $\ell : \Xi \rightarrow L$ 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
- $\delta : W \times L \rightarrow W$: Transition function
- $w_{\text{init}} \in W$: Initial state
- $W_f \subseteq W$: Accepting (final) states

Given a trajectory $\xi : \mathbb{N} \rightarrow \Xi$, 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:

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

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 $W_f = \{d\}$

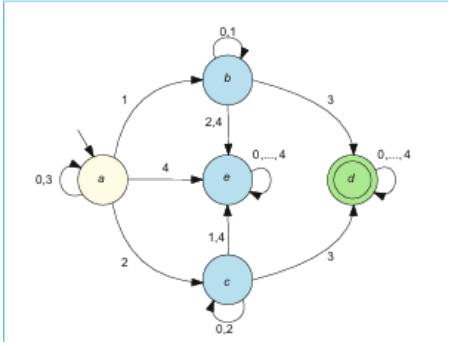


Figure 8: DFA structure for the specification: Visit R1 XOR R2, Avoid R4, Reach R3.

Step 2: Build Augmented System

To synthesize a controller for specification S_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:

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

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**:

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

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 $\tilde{h} : W \times \Xi \rightarrow \Sigma$ and an initial set $Q_0 \subseteq W \times \Xi$ 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

```
1 class DFA:
2     """Deterministic Finite
3         Automaton for complex
4         specifications"""
5
6     def __init__(self, states,
7                  init_state,
8                      accepting_states
9                      , transitions):
10            self.states = states
11            self.init_state =
12                init_state
13                self.accepting_states =
14                    accepting_states
15                    self.transitions =
16                        transitions
17                        # dict: (state, label) ->
18                        next_state
```

```

12     def step(self, state, label):
13         """Execute one transition
14         in the DFA"""
15         return self.transitions.
16     get((state, label), None)
17
18 class AugmentedSystem:
19     """
20     Product system: Symbolic
21     Model    DFA
22     Combines robot dynamics with
23     automaton state
24     """
25
26     def __init__(self, robot, dfa,
27      , region_labels):
28         self.robot = robot
29         self.dfa = dfa
30         self.region_labels =
31     region_labels
32         # maps symbolic states to
33     labels
34
35         # Build augmented state
36     space: (dfa_state,
37     robot_state)
38         self.augmented_states =
39     []
40         self.state_to_idx = {}
41         self.idx_to_state = {}
42         self.targets = set()
43         self.
44     _build_augmented_space()
45
46     def _build_augmented_space(
47     self):
48         idx = 0
49         for robot_state in self.
50     robot.index_to_intervals.keys
51     ():
52             for dfa_state in self.
53     .dfa.states:
54                 aug_state = (
55                 dfa_state, robot_state)
56                 self.
57             augmented_states.append(
58             aug_state)
59                 self.state_to_idx
60             [aug_state] = idx
61                 self.idx_to_state
62             [idx] = aug_state
63
64                 # Mark accepting
65     states as targets
66                 if dfa_state in
67             self.dfa.accepting_states:
68                     self.targets.
69             add(idx)
70
71         idx += 1
72
73     def label_state(self,
74     robot_state):
75         """Get DFA label for a
76         robot symbolic state"""
77         return self.region_labels
78     .get(robot_state, 'other')
79
80     def successors(self, aug_idx,
81     action):
82         """
83             Compute successors in
84             augmented system
85             Returns list of augmented
86             state indices
87             """
88         dfa_state, robot_state =
89     self.idx_to_state[aug_idx]
90         successors = []
91
92         # Get robot successors
93     for this action
94             robot_successors = self.
95     robot.get_successors(
96             robot_state, action)
97
98             for next_robot_state in
99             robot_successors:
100                 # Get label for next
101                 robot state
102                 label = self.
103             label_state(next_robot_state)
104
105                 # Compute next DFA
106                 state
107                 next_dfa_state = self.
108             .dfa.step(dfa_state, label)
109                 if next_dfa_state is
110             None:
111                     continue # Invalid transition in DFA
112
113                     # Create augmented
114                     successor state
115                     next_aug_state = (
116                     next_dfa_state,
117                     next_robot_state)
118                     next_idx = self.
119             state_to_idx[next_aug_state]
120                     successors.append(
121                     next_idx)
122
123             return successors
124
125 class DFAReachabilityController:
126     """
127

```

```

82     Controller synthesis for DFA
83     specifications via
84     reachability on augmented
85     system
86     """
87
88     def __init__(self,
89      augmented_system):
90         self.augmented =
91         augmented_system
92         self.R_star = None # Winning region in augmented
93         space
94         self.layers = [] # Backward reachability layers
95         self.H = {} # Controller: aug_state_idx ->
96         list of actions
97
98         def compute_R_star(self):
99             """Backward reachability
100            on augmented system"""
101             print("Computing R* for
102             DFA specification...")
103             R_prev = set(self.
104             augmented.targets)
105             self.layers = [R_prev.
106             copy()]
107             it = 0
108
109             while True:
110                 it += 1
111                 R_new = set(self.
112                 augmented.targets)
113
114                 # Pre operator on
115                 augmented system
116                 pre_states = set()
117                 for idx in range(len(
118                 self.augmented.
119                 augmented_states)):
120                     if idx in R_prev:
121                         continue
122
123                     # Check if state
124                     can reach R_prev
125                     for action in
126                     self.augmented.robot.ACTIONS:
127                         succs = self.
128                         augmented.successors(idx,
129                         action)
130                         if any(s in
131                         R_prev for s in succs):
132                             pre_states.add(idx)
133                             break
134
135             R_new.update(
136             pre_states)
137             self.layers.append(
138             R_new.copy())
139
140             if R_new == R_prev:
141                 break
142
143             R_prev = R_new
144
145             if it > 100:
146                 print("Warning:
147                 Max iterations reached")
148                 break
149
150             self.R_star = R_prev
151             print(f"R* computed: {len(
152             self.R_star)} states")
153             return self.R_star
154
155             def compute_controller(self):
156                 """Synthesize controller
157                 on winning region"""
158                 if self.R_star is None:
159                     self.compute_R_star()
160
161                     print("Synthesizing DFA
162                     controller...")
163
164                     for idx in self.R_star:
165                         # Find earliest layer
166                         containing this state
167                         k = None
168                         for i, layer in
169                         enumerate(self.layers):
170                             if idx in layer:
171                                 k = i
172                                 break
173
174                             if k is None or k ==
175                             0:
176                                 continue
177
178                                 # Target: states in
179                                 layer k-1
180                                 target_layer = self.
181                                 layers[k-1]
182                                 valid_actions = []
183
184                                 for action in self.
185                                 augmented.robot.ACTIONS:
186                                     succs = self.
187                                     augmented.successors(idx,
188                                     action)
189                                     if succs and any(
190                                     s in target_layer for s in
191                                     succs):

```

```

2 R1_states = robot.
3     find_indices_for_interval
4         ([[1, 3], [7, 9]]) # Top-
5             left
6 R2_states = robot.
7     find_indices_for_interval
8         ([[7, 9], [7, 9]]) # Top-
9             right
10 R3_states = robot.
11     find_indices_for_interval
12         ([[4, 6], [1, 3]]) # Bottom-
13             center
14 R4_states = robot.
15     find_indices_for_interval
16         ([[2, 8], [4, 6]]) # Middle
17             obstacle
18
19 # Create labeling function
20 region_labels = {}
21 for state in R1_states:
22     region_labels[state] = 'R1'
23 for state in R2_states:
24     region_labels[state] = 'R2'
25 for state in R3_states:
26     region_labels[state] = 'R3'
27 for state in R4_states:
28     region_labels[state] = 'R4'
29
30 # Define DFA for specification: (R1 XOR R2) then R3, avoid R4
31 dfa = DFA(
32     states=[ 'q0' , 'q1' , 'q2' , 'q3'
33 , 'q4' , 'qfail' ],
34     init_state='q0',
35     accepting_states=[ 'q3' , 'q4'
36 ], 
37     transitions={
38         ('q0', 'R1'): 'q1', ('q0',
39 , 'R2'): 'q2',
40         ('q0', 'R3'): 'q0', ('q0',
41 , 'R4'): 'qfail',
42         ('q0', 'other'): 'q0',
43         ('q1', 'R1'): 'q1', ('q1',
44 , 'R2'): 'qfail',
45         ('q1', 'R3'): 'q3', ('q1',
46 , 'R4'): 'qfail',
47         ('q1', 'other'): 'q1',
48         ('q2', 'R1'): 'qfail', ('q2',
49 , 'R2'): 'q2',
50         ('q2', 'R3'): 'q4', ('q2',
51 , 'R4'): 'qfail',
52         ('q2', 'other'): 'q2',
53         # Absorbing states
54         ('q3', 'R1'): 'q3', ('q3',
55 , 'R2'): 'q3',
56         ('q3', 'R3'): 'q3', ('q3',
57 , 'R4'): 'q3',
58         ('q3', 'other'): 'q3',
59     }
60 )

```

Listing 7: DFA Controller Implementation

Implementation Example

```
1 # Define regions in the  
environment
```

```

33     ('q4', 'R1'): 'q4', ('q4',
34     , 'R2'): 'q4',
35     ('q4', 'R3'): 'q4', ('q4',
36     , 'R4'): 'q4',
37     ('q4', 'other'): 'q4',
38     ('qfail', 'R1'): 'qfail',
39     ('qfail', 'R2'): 'qfail',
40     ('qfail', 'R3'): 'qfail',
41     ('qfail', 'R4'): 'qfail',
42     ('qfail', 'other'): 'qfail',
43   }
44
45 # Build augmented system
46 augmented = AugmentedSystem(robot,
47   , dfa, region_labels)
48
49 # Synthesize controller
50 controller =
51   DFAResponsibilityController(
52     augmented)
53 R_star = controller.
54   compute_R_star()
55 H = controller.compute_controller
56   ()
57
58 # Run simulation
59 initial_state = [0.5, 0.5, 0.0]
60   # Start at bottom-left
61 trajectory = simulate_with_dfa(
62   robot, controller,
63   initial_state, steps=50)

```

Listing 8: DFA Controller Application

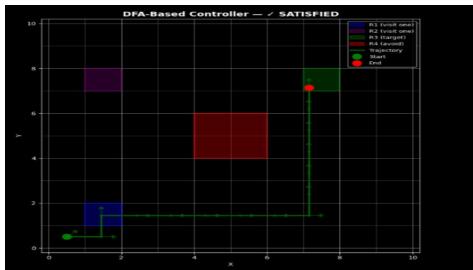


Figure 9: First Specification: Visit (R1 XOR R2), Reach R3 and avoid R4

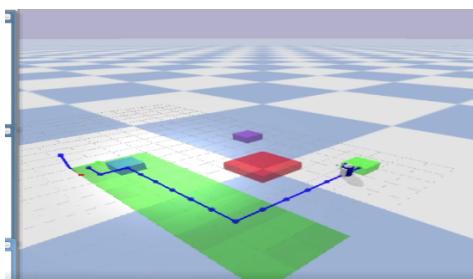


Figure 10: Visualization of the trajectory

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."

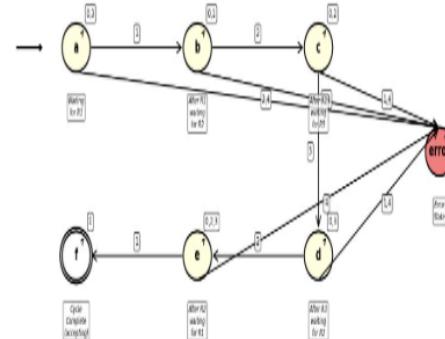


Figure 11: Sequential visit specification: R1 → R2 → R3 → R4 → R3.

Oscillatory Specification: $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$

A specification requiring oscillation between regions: "Visit R_1 , then R_2 , then R_3 , then return to R_2 , and finally return to R_1 ."

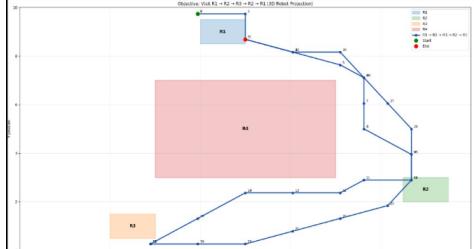


Figure 12: Oscillatory specification: $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$.



Figure 13: Visualization of trajectory for oscillatory specification.

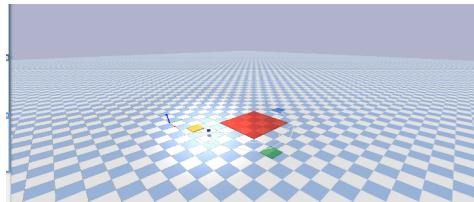


Figure 14: Another visualization of oscillatory trajectory.

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.