

# Parallel and Incremental Verification of Hybrid Automata with Ray and Verse



Haoqing Zhu



Yangge Li



Keyi Shen



Sayan Mitra

Presented by: Kristina Miller ([kmmille2@illinois.edu](mailto:kmmille2@illinois.edu))

Corresponding Author: Yangge Li ([li213@illinois.edu](mailto:li213@illinois.edu))

University of Illinois Urbana-Champaign

ATVA 2023, Singapore

<https://github.com/AutoVerse-ai/Verse-library>



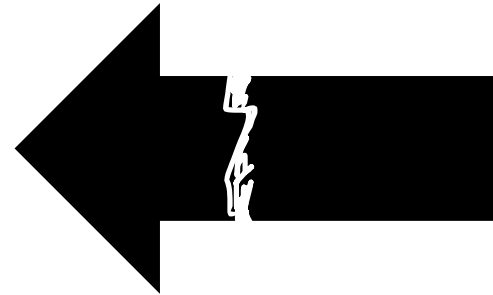
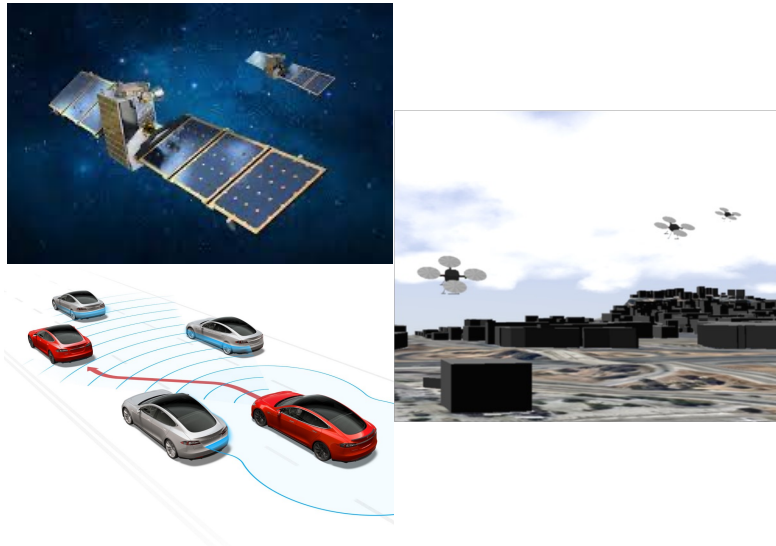
UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# Need for yet another hybrid verification tool



Verifying scenarios with multiple hybrid systems is important for autonomous ground, air, & space vehicle systems

Many advances in hybrid verification, but they are siloed and find limited use in multi-agent scenarios



# Barriers to usability of hybrid verification



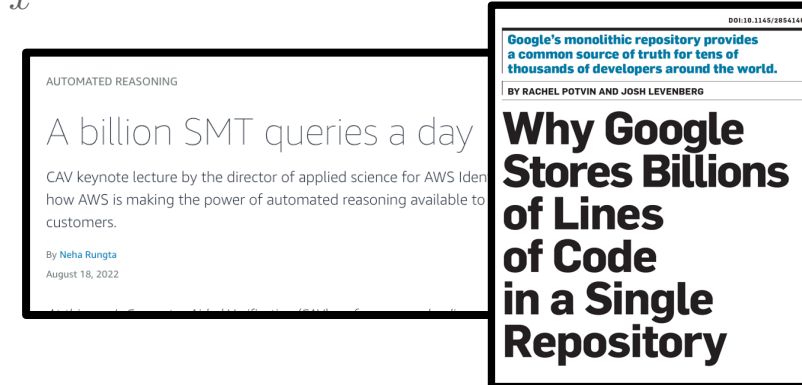
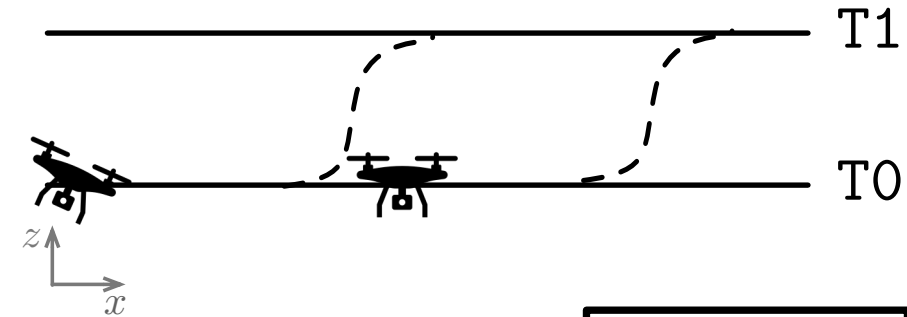
Each tool has its own modeling language

Discrete modes have to be spelled out explicitly

Code-level verification is more likely to be used (compared to code->model->verification->code)

Loss of expressive power (for not using fancy logics) more than made up by benefits of using *same language for code & spec*

**Goal: Make hybrid modeling and verification accessible to undergraduate engineering students**



[1] Sadowski, et al. Lessons from Building Static Analysis Tools at Google. CACM, 18.

[2] Chong, et al. Code-level model checking in the software development workflow. ICSE, 20.

[3] O'Hearn. Continuous reasoning: Scaling the impact of formal methods. LICS '18.

# Maps, moves, and transition logic in Python

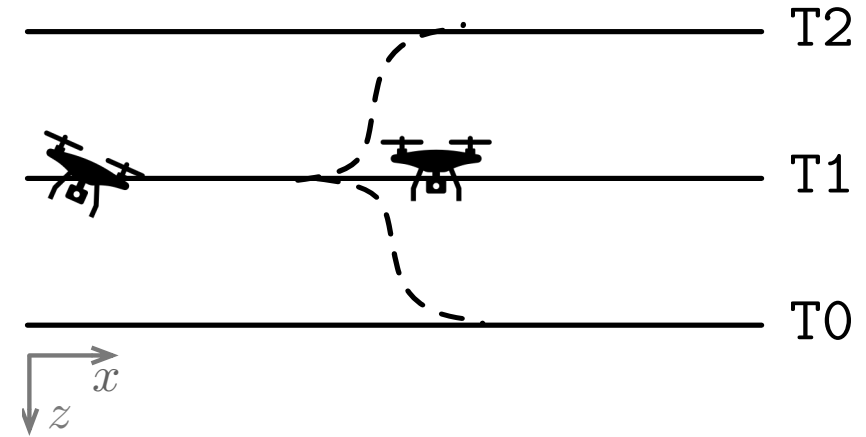


```
class Move(Enum):
    Cruise = auto()
    Up = auto()
    Down = auto()

class State:
    x: float
    ...
    vz: float
    move_mode: Move
    track_mode: Track

class Track(Enum):
    T0 = auto()
    T1 = auto()
    T2 = auto()
    TShift = auto()

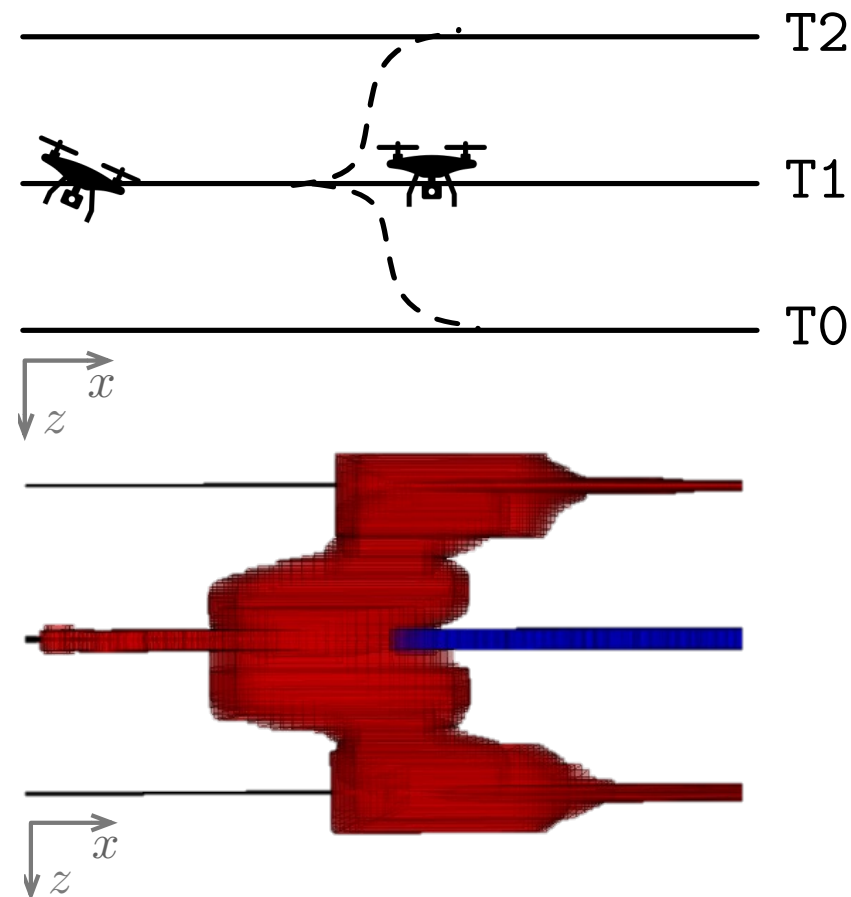
def decisionLogic(ego: State, other: State, track_map):
    next = copy.deepcopy(ego)
    if ego.move_mode == Cruise:
        if ego.x - other.x <= 10:
            next.move_mode = Move.Up
        if ego.x - other.x <= 10:
            next.move_mode = Move.Down
    next.track_mode = track_map.Map2Mode(ego.track_mode,
                                         ego.move_mode, next.move_mode)
    return next
```



# Scenarios and reachability

```
# Initialize agents and map
drone1 = QuadrotorAgent("red", file_name=drone1logic.py, ...)
Init1 = [[1.5, -0.5, -0.5, 0, 0, 0], [2.5, 0.5, 0.5, 0, 0, 0]]
drone1.set_initial(Init1, (Move.Cruise, Track.T1))
drone2 = QuadrotorAgent("blue", file_name=drone1logic2, ...)
...
myMap = M5()
# Populate scenario
scenario.add_agent(drone1)
scenario.add_agent(drone2)
scenario.set_map(myMap)
reachTree = scenario.verify(60, time_step,...)
# Plot results
fig = go.Figure()
fig = plot3dReachtube(reachTree, "test1", 1, 2, 3, ...)
```

Reachability performed by NeuReach [Sun, TACAS'21], DryVR [Fan, CAV'17] and mixed-monotonicity [Abate-Coogan, CDC'20]



# Outline

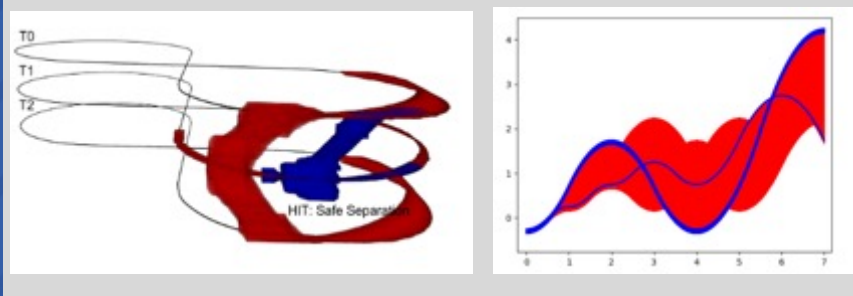
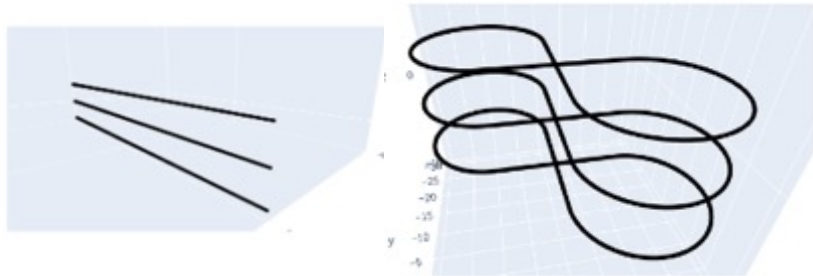
Quick introduction to Verse [Li 23]

Parallel Verification

Incremental Verification

Summary & Ongoing work

```
38 def decisionLogic(ego: State, others: List[State], track_map):
39     next = copy.deepcopy(ego)
40     if ego.tactical_mode == TacticalMode.Normal:
41         if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
42             ↪ others):
43             next.tactical_mode = TacticalMode.MoveDown
44             next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
45             ↪ TacticalMode.MoveDown)
46         if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
47             ↪ others):
48             next.tactical_mode = TacticalMode.MoveUp
49         # ...
50     if ego.tactical_mode == TacticalMode.MoveUp:
51         if in_interval(track_map.altitude(ego.track_mode)-ego.z, -1, 1):
52             next.tactical_mode = TacticalMode.Normal
53             next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
54             ↪ TacticalMode.Normal)
55         # ...
56     return next
```



# Verse Internals and interfaces



Verse internally constructs: hybrid system  $(X, D, \text{Init}, G, R, \text{TL})$

Continuous state space  $X = X_1 \times \dots \times X_k$

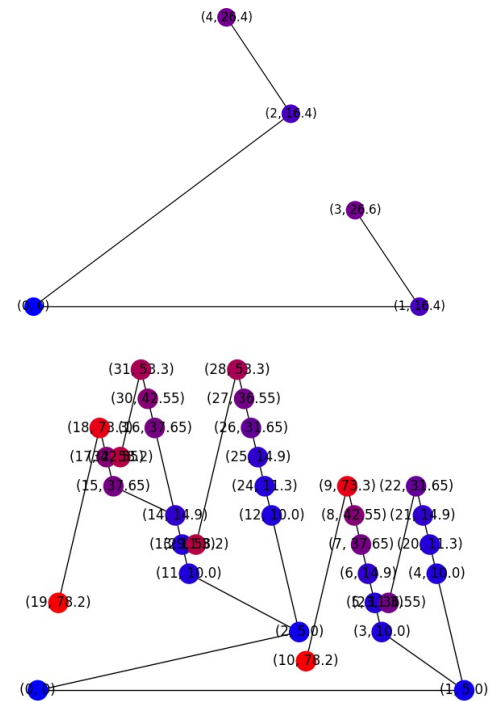
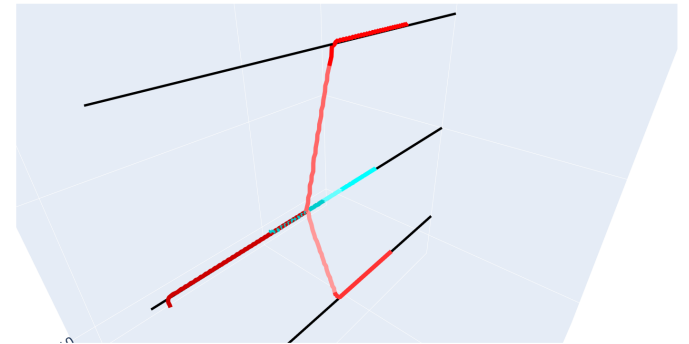
For each mode pair guard  $G(d, d') \subseteq X$  and reset  $R(d, d'): X \rightarrow X$

TL is a set of trajectories  $\langle \xi, d \rangle$  where continuous evolution in mode  $d$  is given by  $\dot{\xi}(t) = \Pi_i F(x_j, d, t)$

$\delta$ -executions  $\alpha := \langle \xi^0, d^0 \rangle, \langle \xi^1, d^1 \rangle$

- $\xi^0(0)$  is in Init
- $\xi^{i-1}(\delta) \in G(d^{i-1}, d^i)$  and  $\xi^i(0) = R(d^{i-1}, d^i)(\xi^{i-1}(\delta))$

$\text{post}_{d, d'}(X)$  and  $\text{post}_{d, \delta}(X)$  discrete and continuous post operators can be used to build verification algorithms



# Reachability Tree

Verse uses the post operators to construct the reachtree

Tree node  $N = \langle X, d, t, \text{stride}, \text{children} \rangle$

- stride computed by  $\text{post}_{d,\delta}$

Root  $\langle X_0, d_0, 0, \text{stride}, \text{children} \rangle$

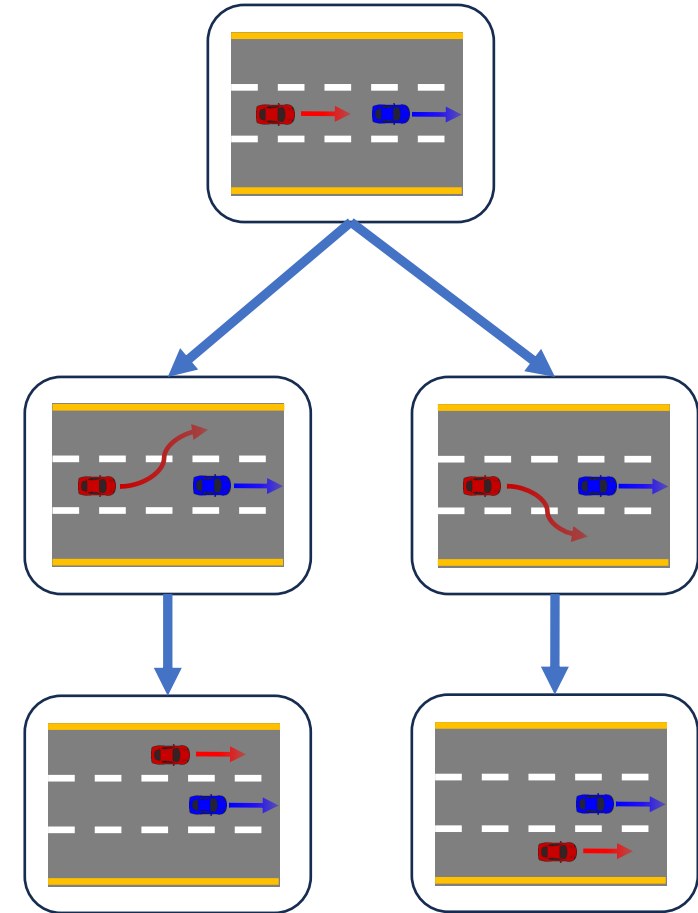
Edge from  $\langle X, d, t, \text{stride}, \text{children} \rangle$  to  $\langle X', d', t', \text{stride}', \text{children}' \rangle$

- $X' = \text{post}_{d,\delta}(\text{post}_{d,d'}(X)).lstate$  // Step
- $t' = t + \delta$

Each branch corresponds to the unique choices made by all the relevant agents

Tree traversed using BFS with a queue

**Takeaway:** Different branches in reachtree can be computed independently



# Ray Framework Enables Parallelization



Ray framework developed by RISELab from Berkeley [Moritz 2018]

**Remote function** calls as an abstraction for parallelization with fork-join style parallelization

Parallelization does not involve the explicit use of synchronization primitives like locks

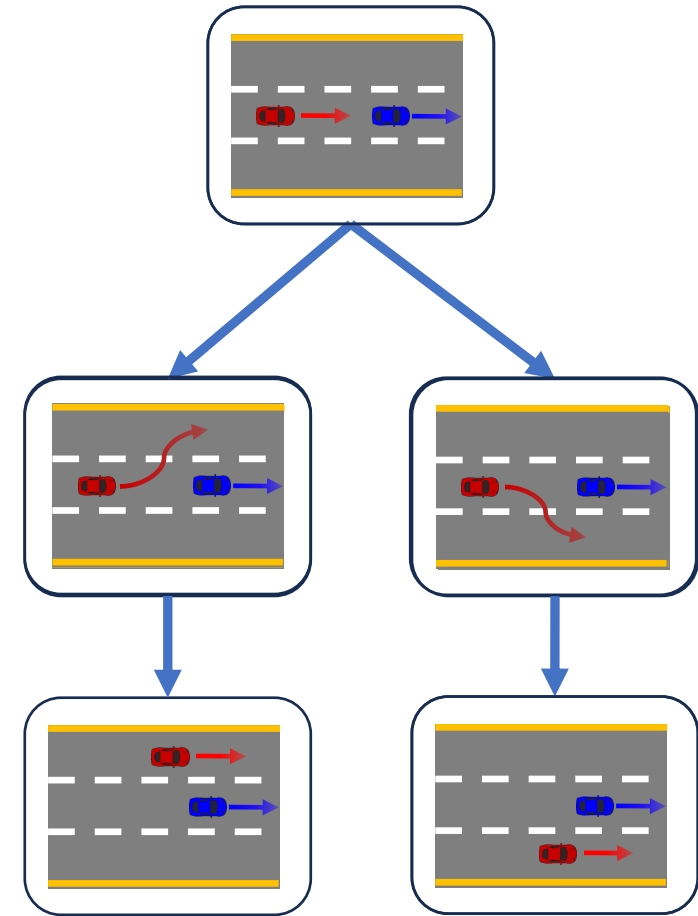
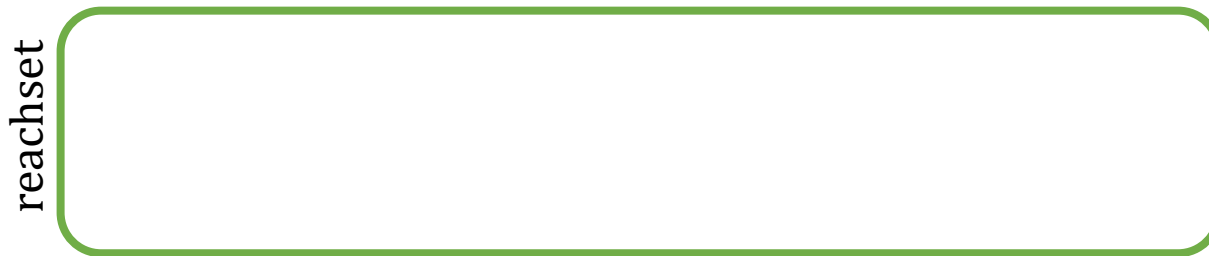
Functions can be turned into remote function using `ray.remote` decorator

Processes in Ray communicate through distributed database.

- Store arguments and return value
- Caller can poll and fetch return value from database using **`ray.wait()`**

# Parallel Verification: Algorithm

```
function verify_parallel( $H, X^0, d^0, \delta, T_{max}$ )  
  queue  $\leftarrow [(X^0, d^0, \emptyset)]$   
  refs, reachset  $\leftarrow \emptyset$   
  while queue  $\neq \emptyset$  & refs  $\neq \emptyset$  do  
    if queue  $\neq \emptyset$  then  
       $N \leftarrow$  queue.dequeue()  
      refs.add(remote Step( $N, \delta$ ))  
    else  
       $(N, refs) \leftarrow$  ray.wait(refs)  
      reachset  $\leftarrow$  reachset  $\cup$   $N.stride$   
      for  $N' \in N.children$  do  
        if  $N'.t < T_{max}$  then  
          queue.add( $N'$ )  
  return reachset
```



# Performance Gain with Parallel Verification

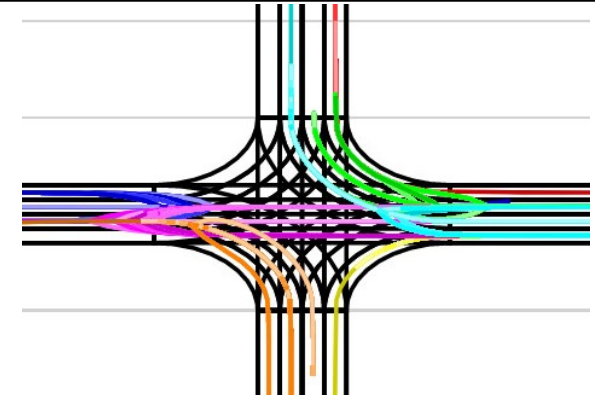
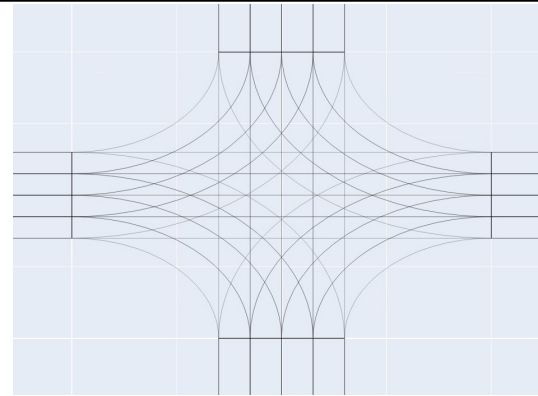


Parallel Verse speeds up running time from 3.5 hours ~30 mins for a 4-lane 20 agent scenario

Number of leaves in the reachtree correlates to speedup

Increasing number of cores generally increases speedup

Sometimes, rarely, parallel overhead can make it slower than baseline



Name	# leaves	Serial	2 cores	4 cores	8 cores
curve	2	5056 (0.89)	55 (0.91)	56 (0.89)	
drone	2	2937 (0.78)	30 (0.97)	29 (1)	
race	2	220162 (1.36)	156 (1.41)	157 (1.4)	
drone8	4	3136 (0.86)	36 (0.86)	37 (0.84)	
wide (7,2)	7	188135 (1.39)	136 (1.38)	137 (1.37)	
isect (4,9)	11	342349 (0.98)	190 (1.8)	130 (2.63)	
isect (4,10)	15	587606 (0.97)	318 (1.85)	197 (2.98)	
wide (8,3)	20	311313 (0.99)	172 (1.81)	104 (2.99)	
isect (4,15)	37	21152085 (1.01)	1081 (1.96)	653 (3.24)	
isect (4,20)	140	131008416 (1.56)	4477 (2.93)	2085 (6.28)	
isect (4,12)	225	71364445 (1.65)	2214 (3.3)	1302 (5.62)	

# Outline

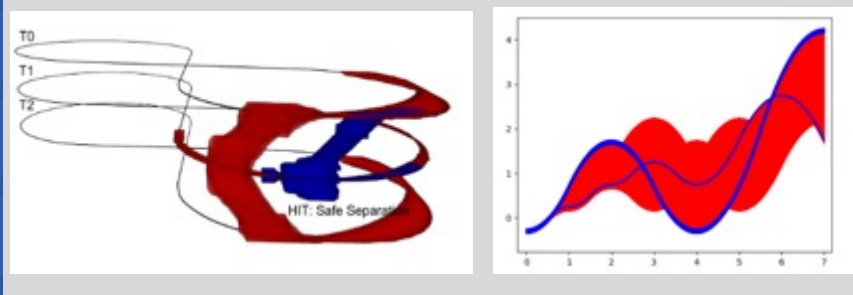
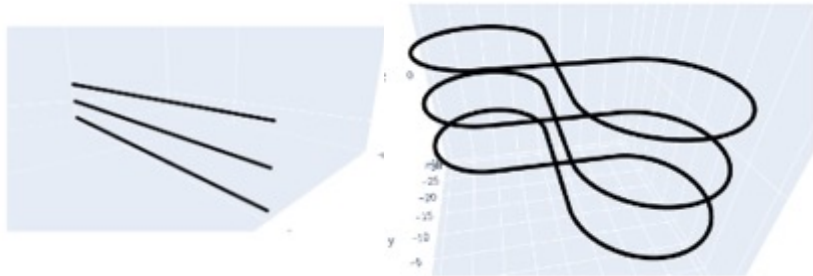
Quick introduction to Verse [Li 23]

Parallel Verification

Incremental Verification

Summary & Ongoing work

```
38 def decisionLogic(ego: State, others: List[State], track_map):
39     next = copy.deepcopy(ego)
40     if ego.tactical_mode == TacticalMode.Normal:
41         if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
42             ↪ others):
43             next.tactical_mode = TacticalMode.MoveDown
44             next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
45             ↪ TacticalMode.MoveDown)
46         if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
47             ↪ others):
48             next.tactical_mode = TacticalMode.MoveUp
49         # ...
50     if ego.tactical_mode == TacticalMode.MoveUp:
51         if in_interval(track_map.altitude(ego.track_mode)-ego.z, -1, 1):
52             next.tactical_mode = TacticalMode.Normal
53             next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
54             ↪ TacticalMode.Normal)
55         # ...
56     return next
```



# Incremental Verification: Intuition



Consider two hybrid automata  $H_1$  and  $H_2$  that only differ in the discrete transitions and we want to run  $\text{Verify}(H_1)$  followed by  $\text{Verify}(H_2)$

**Intuition:** reuse computations in constructing reachtree for  $H_1$  in computing reachtree for  $H_2$

$\text{Step\_batch}(N^0, \delta, T_{max})$ : Batch together all the adjacent **Step** that have the same discrete modes. Output reachset and

Uses a cache (C) to stores the result of **Step\_batch**

- Key:  $\langle X^0, d^0 \rangle$ ; Value: Output from **Step\_batch**

Algorithm checks C before every call to **Step\_batch**

```
H1 = Scenario(decisionLogic)
H1.Verify()
Modify decisionLogic
H2 = Scenario(decisionLogic)
H2.Verify()
```

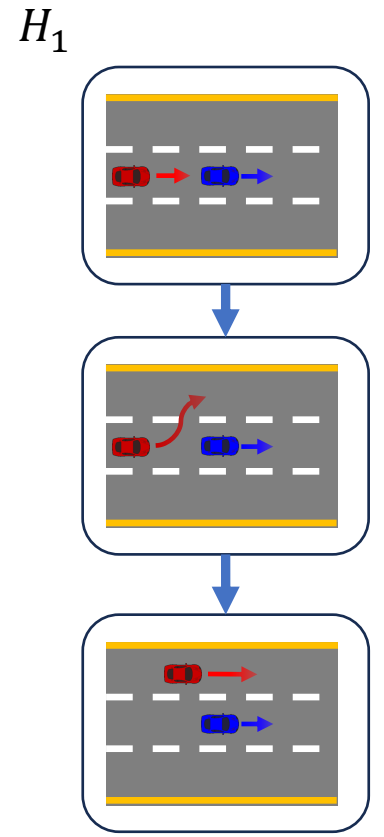
```
class Move(Enum):
    Cruise = auto()
    Up = auto()
    Down = auto()

class State:
    x: float
    ...
    vz: float
    move_mode: Move
    track_mode: Track

def decisionLogic(ego: State, other: State, track_map):
    next = copy.deepcopy(ego)
    if ego.move_mode == Move.Cruise:
        if ego.x - other.x <= 20:
            next.move_mode = Move.Up
        if ego.x - other.x <= 20:
            next.move_mode = Move.Down
    next.track_mode = track_map.Map2Mode(ego.track_mode,
                                         ego.move_mode, next.move_mode)
    return next
```

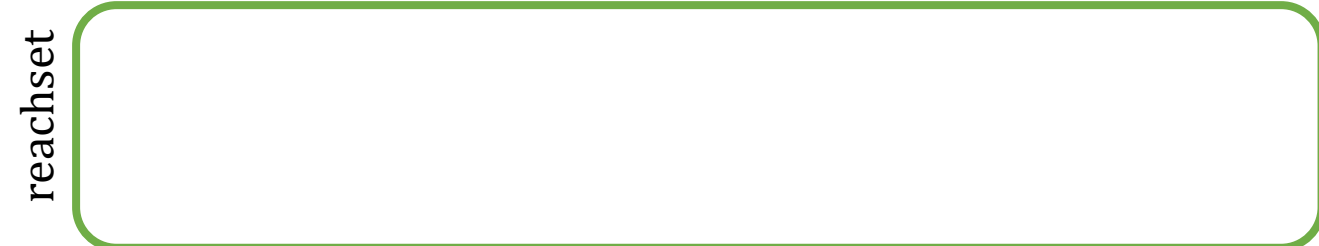
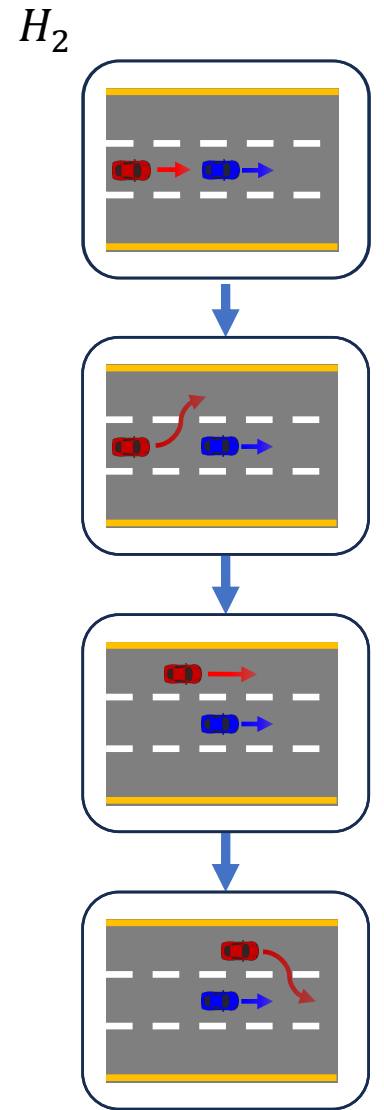
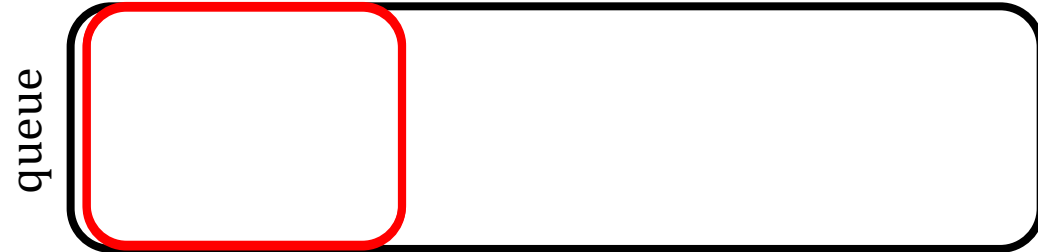
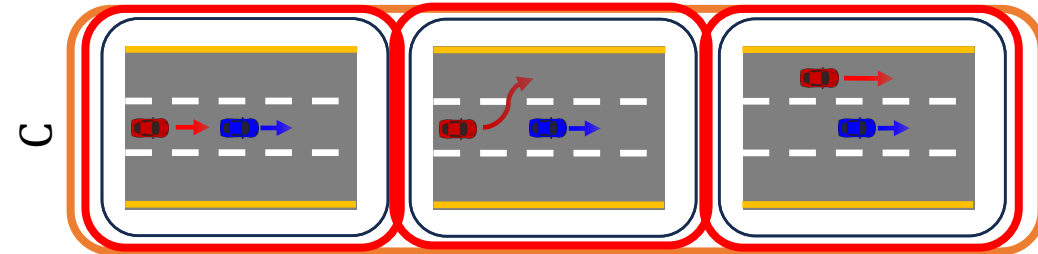
# Incremental Verification: Algorithm

```
function verify_incremental(H, X0, d0, δ, Tmax, C)
  queue ← [(X0, d0, 0)]
  reachset ← ∅
  while queue ≠ ∅ & refs ≠ ∅ do
    N ← queue.dequeue()
    if C(N.X, N.d) ≠ ∅ then
      (subreachset, br) ← C(N.X, N.d)
      reachset ← reachset ∪ subreachset
      for N' ∈ br.children s.t. N'.t < Tmax do
        queue.add(N')
    else
      (subreachset, br) ← Step_batch(N, δ, Tmax)
      C(N.X, N.d) ← (subreachset, br)
      reachset ← reachset ∪ N.stride
      for N' ∈ br.children s.t. N'.t < Tmax do
        queue.add(N')
  return reachset
```



# Incremental Verification: Algorithm

```
function verify_incremental(H, X0, d0, δ, Tmax, C)
  queue ← [(X0, d0, 0)]
  reachset ← ∅
  while queue ≠ ∅ & refs ≠ ∅ do
    N ← queue.dequeue()
    if C(N.X, N.d) ≠ ∅ then
      (subreachset, br) ← C(N.X, N.d)
      reachset ← reachset ∪ subreachset
      for N' ∈ br.children s.t. N'.t < Tmax do
        queue.add(N')
    else
      (subreachset, br) ← Step_batch(N, δ, Tmax)
      C(N.X, N.d) ← (subreachset, br)
      reachset ← reachset ∪ N.stride
      for N' ∈ br.children s.t. N'.t < Tmax do
        queue.add(N')
  return reachset
```



# Performance Gain with Incremental Verification



$T_{\text{change}}$ : Earliest time when the automaton's behavior changes

Provides more speedup when  $T_{\text{change}}$  is large

Combination of parallel and incremental gives more speedup

Sometimes, serial verify\_incremental can be much faster than parallel verify\_incremental

- Number of leaves small

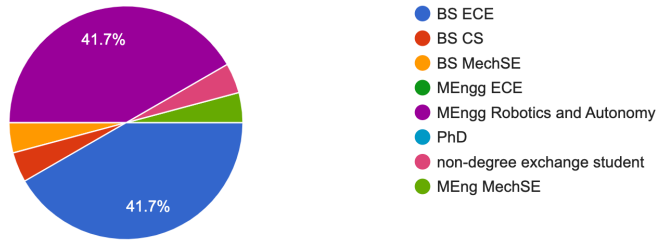
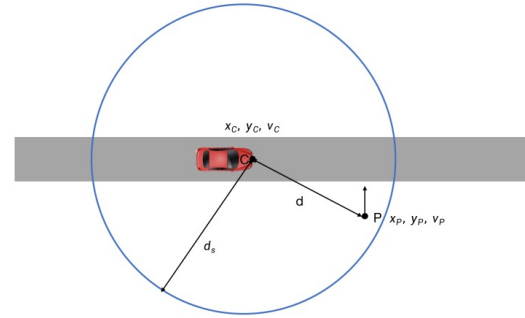
Name	#leaves	$T_{\text{change}}$	ser	inc ser	par	inc par
drone	3	75%	53	29	51	30
wide (8,3)	10	15.67%	173	20	80	51
wide (8,3)	20	$\infty$	335	17	104	16
wide (8,3)	8	0%	182	129	72	53
isect (4,15)	2	0%	116	98	100	83
isect (4, 15)	12	$\infty$	575	36	189	20
isect (4,15)	9	82.25%	404	35	159	82

# Verse deployed for checking designs in autonomy class with 70+ students



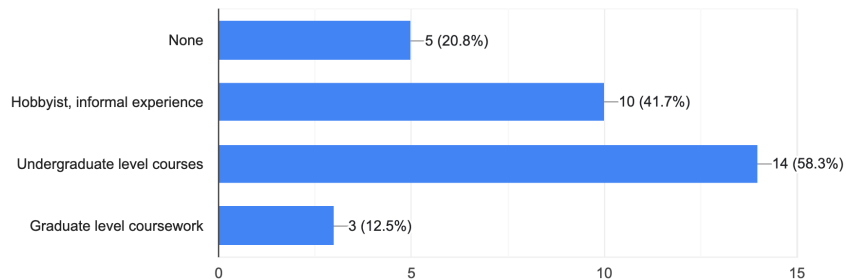
## 3 Testing Automatic Emergency Braking with Simulations

You are in charge of designing the *automatic emergency braking (AEB) decision logic (DL)* for a car. You will write the DL code and test it. You will have to provide evidence that will convince auditors that your design is safe. The design should not be too conservative, i.e., hard-brake all the time, and try to achieve high average speed.



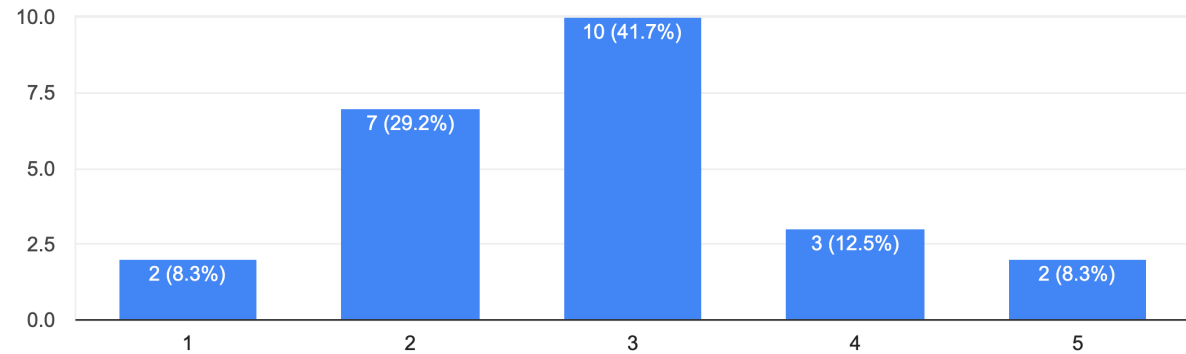
Prior Experience in Robotics

24 responses



Based on your experience, do you think reachability is effective tool for design?

24 responses



Q. What was your overall experience with the reachability tool?

A. It was nice for R1. For R3, it took too long to be used in the design planning.

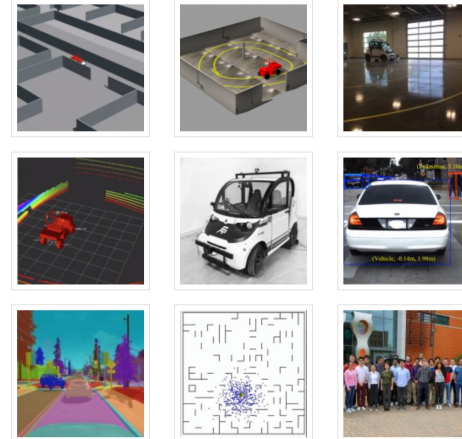
Q. What were some of the **ineffective steps** of your design approach? (Wasted time)

A. math **proofment**

# Summary



- Verse: Accessible Python library for hybrid modeling and verification
  - Tool available from: <https://github.com/AutoVerse-ai/Verse-library>
- Parallelization using Ray, Incremental verification
  - Parallelization and incremental verification techniques saves verification time
  - Making the tool more usable
- Ongoing
  - Verification of  $L_1$  adaptive control [Song 23]
  - Closed-Loop verification of ACAS Xu
  - Fall 23: user study with undergraduates in Safe Autonomy Course (ECE484)
- We will only succeed with your help and participation
  - Try it, help improve it!



<https://publish.illinois.edu/safe-autonomy/>

