ECE 484: Safety Analysis and Verse Tutorial Related to Bonus Problem

Spring 24

Professors: Sayan Mitra

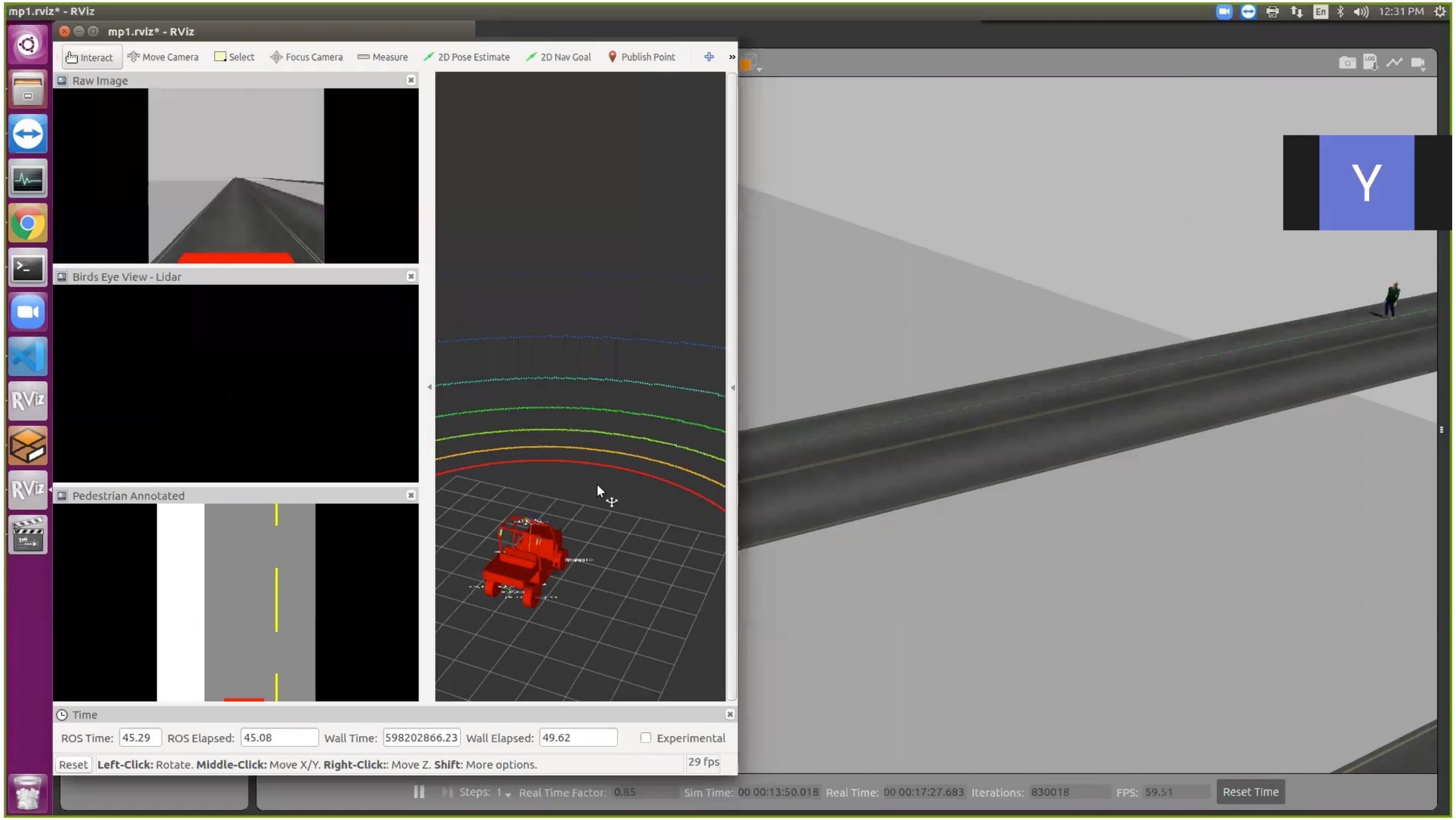
April 2, 2024

Goal: Provide evidence for safety of an autonomous system

- What does safety mean?
- What does it mean for a system to be safe?
- What does it mean to provide evidence for the above?

- Tl:dr: Safety is defined by a set of *bad* states that should never be reached. Evidence of safety = tests or proofs that show that none of the behaviors of the system ever reach the bad *states.*

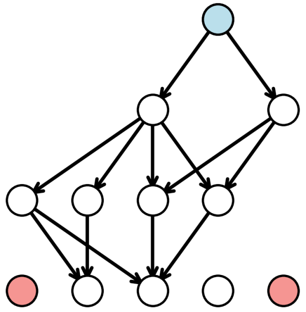
Automatic Emergency Braking



Setup: Automata, executions, and safety

An automaton is a triple $A = \langle Q, Q_0, D \rangle$ where

- Q is a set of states
- Q_0 is a set of initial states
- $D \subseteq Q \times Q$ is a set of transitions



An **execution** of A is a sequence of states $\alpha = q_0, q_1, q_2, \ldots$ such that $q_0 \in Q_0$ and for each i, $(q_i, q_{i+1}) \in D$.

Q: Set of vertices
$Q_0 = \{o\}$
D: Set of edges

Generally, an automaton can have uncountably infinite executions

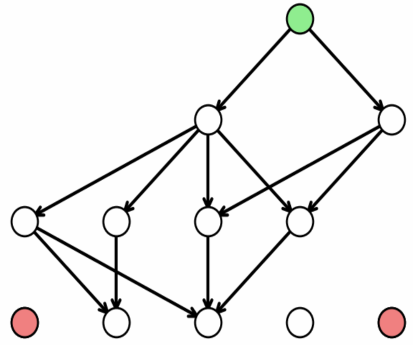
Automata, executions, and safety

Safety of automaton $A = \langle Q, Q_0, D \rangle$ is specified by a set of unsafe states $U \subseteq Q$ that the automaton should *never reach*

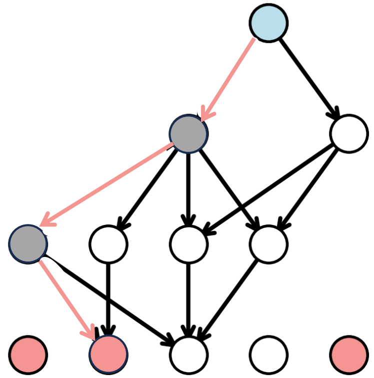
Automaton A is safe with respect to U if for every execution $\alpha = q_0 q_1 \ldots$ of A if for every q_i in α, $q_i \notin U$.

If Q is finite (and small) then DFS on A gives an algorithm for checking safety

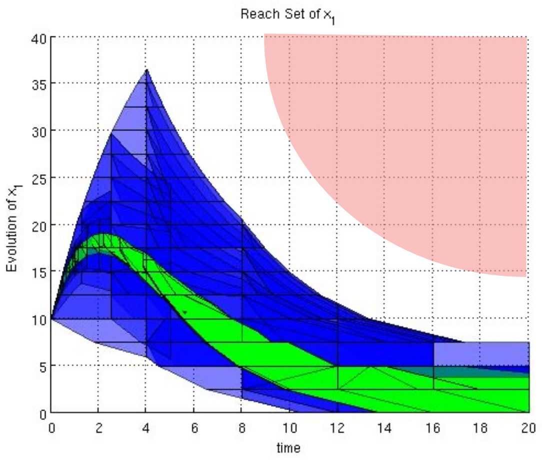
Enumerating individual executions is insufficient for checking safety for automata with uncountably many executions



All executions are safe



Unsafe execution found



Thermostat in Verse

State variables

$x : \mathbb{R} = 70$ // temperature

$mode : \{on, off\} = on$ // heater state

Transitions

Jumps

if $x \geq 70$ then $mode = off$

if $x \leq 62$ then $mode = on$

Flows (every Δ time)

$\frac{dx}{dt} = H - kx$ where $H = 0$ for on and 10 for off

$k = 0.5$

```
class ThermMode(Enum):
   On = auto()
   Off = auto()
```

```
class State:
'''State variables'''
   x: float
   mode: ThermMode
```

```
def decisionLogic(ego: State):
'''Jump Transitions'''
    output = copy.deepcopy(ego)
    if ego.x >= 75.0:
        output.mode = ThermMode.Off
    if ego.x <= 62.0:
        output.mode = ThermMode.On
    return output
```

You will write this

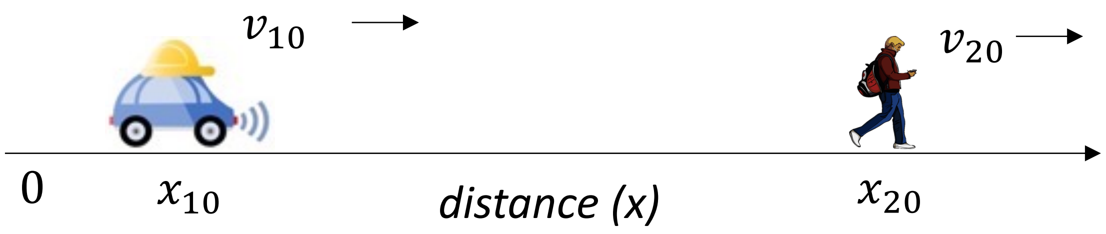
```
class ThermAgent(BaseAgent):
def dynamic(t, state, u):
'''RHS of ODE defining Flows '''
   x = state
   H, k = u
   x_dot = H – k*temp
   return [x_dot]

def TC_simulate(self, mode: List[str], init, timeBound, Delta,..)
…
        r = ode(self.dynamic)
        if mode[0]=="On":
            r.set_initial_value(init).set_f_params([10, 0.5])
        else:
            r.set_initial_value(init).set_f_params([0, 0.5])
        trace = r.integrate(r.t + Delta)
return np.array(trace)

thermostat = ThermAgent('thermostat')
trace = thermostat.simulate(["On"], [70,0], 10, 0.05)
```

You do not need to modify

Automatic Emergency Braking



State variables

$x_1, x_2 : \mathbb{R}$

$v_1, v_2 : \mathbb{R}$

State transitions

```
def decisionLogic(ego:State, others:List[State], track_map):
    output = copy.deepcopy(ego)
    if ego.mode == Normal and in_front(ego, others):
        output.mode = Brake
    if ego.mode == Normal and in_front(ego, others):
        output.mode = SwitchLeft
    if ego.mode == Normal and in_front(ego, others):
        output.mode = SwitchRight
...
```

Nondeterministic transitions

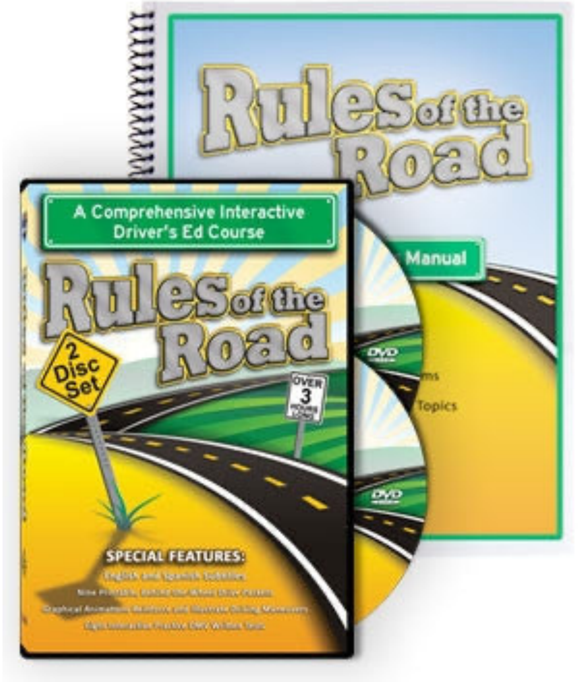
Automaton model for AEB

$Q = \mathbb{R}^4$

$Q_0 = \{\langle x_{10}, x_{20}, v_{10}, v_{20} \rangle\}$

$D = ?$

Safety and requirements



A *requirement* is a statement about a system's executions.

Our goal is to *provide evidence* that *all executions* satisfy the given requirements

- Examples. "Ball never reaches a height above h" $\forall\, t, x(t) \leq h$
- "Ball eventually sits on the ground at x = 0" $\exists t, x(t) = 0$
- "Car always maintains safe distance to pedestrian" $\forall t, x_2(t) - x_1(t) > 2\,m$

```
assert not (other.signal == RED and (other.x - 20 < ego.x < other.x -15))
assert not (other.signal == RED and (other.x - 15 < ego.x < other.x) and ego.v<vo)
```

Safety requirements are statements that must always hold (or never be violated) along all executions

Safety requirements as Asserts in Verse

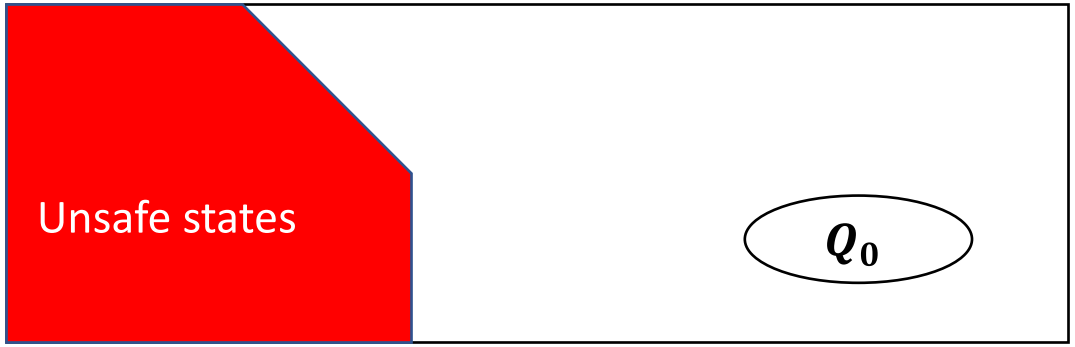
Safety requirements can be seen as a set of **unsafe states** that must be avoided

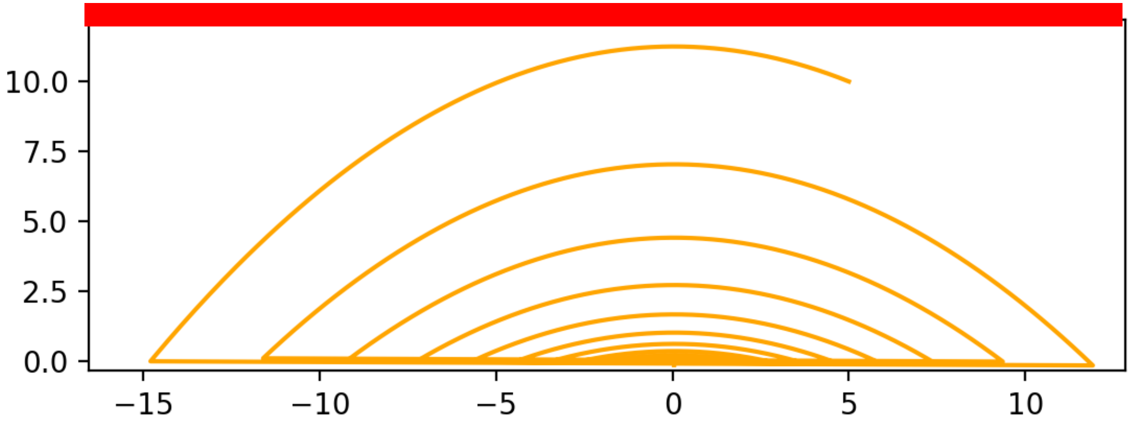
"Cars always remain >= 1 m apart"

"Ball never goes above h" $\forall\, t, x(t) \leq h$

corresponding unsafe set

$U = \{\langle x, v\rangle | x > h\} \subseteq \mathbb{R}^2$





In verse:

```
def vehicle_close(ego, others):
  return any(abs(ego.x - other.x)<1.0 and abs(ego.y-other.y)<1.0 for other in others)

assert not vehicle_close(ego, others), "Seperation"
```

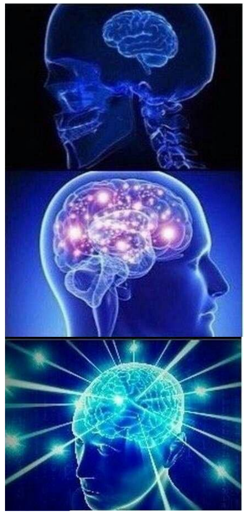
Evidence for safety: Coverage

- An automaton can have many executions
- Sources of nondeterminism
  - Set of initial states Q_0
  - Many transitions from one state

3 vehicles starting in different *sets* of initial states (4d-rectangles)

```
scenario.set_init(
[[[5, -0.5, 0, 1.0], [5.5, 0.5, 0, 1.0]],
[[20, -0.2, 0, 0.5], [20, 0.2, 0, 0.5]],
[[4-2.5, 2.8, 0, 1.0], [4.5-2.5, 3.2, 0, 1.0]],],
[(AgentMode.Normal, TrackMode.T1),
(AgentMode.Normal, TrackMode.T1),
(AgentMode.Normal, TrackMode.T0),]
)
```

Different levels of evidence



scenario.simple_simulate(T, Delta)

computes a single execution from a single initial state up to time T (runs Python code)

scenario.simulate(T, Delta)

computes *all* simulations from a single initial state up to time T (does DFS, Verse function)

scenario.verify(T, Delta)

computes *all* simulations from *all* initial states up to time T (does DFS + reachability analysis, Verse function)

```
def decisionLogic(ego:State, others:List[State], track_map):
   output = copy.deepcopy(ego)
   if ego.mode == Normal and in_front(ego, others):
      output.mode = Brake
   if ego.mode == Normal and in_front(ego, others):
      output.mode = SwitchLeft
   if ego.mode == Normal and in_front(ego, others):
      output.mode = SwitchRight
…
```

Real sources of nondeterminism / uncertainty

- Range of initial conditions $x_1 : \mathbb{R} \in [x_{10} - 0.5, x_{10} + 0.5]$
- Range of braking force
  - $a_{brake} = choose\ [a_1, a_2]$
  - $v_1' = \max(0, v_1 - a_{brake})$
- Noise in sensing distances …
- Unpredictable motion of pedestrians
- Error / drift in timers
- Uncertainty in model parameters, e.g., friction

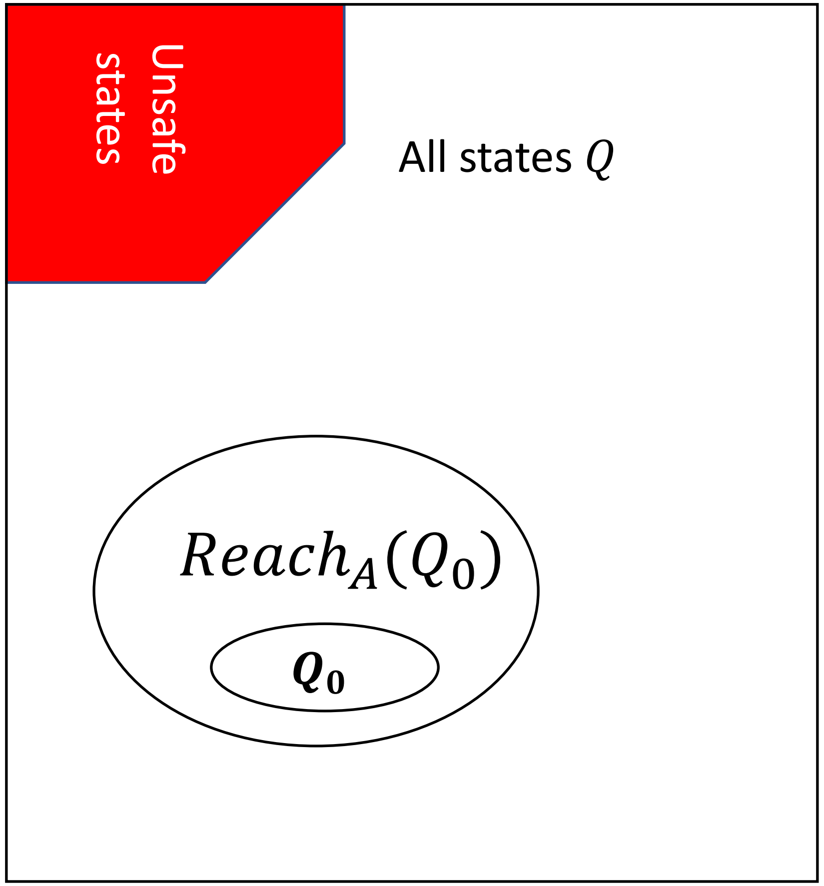
Verify() and Reachable states

Given an automaton $A = \langle Q, Q_0, D\rangle$ the set of **reachable states** of A is defined as

$$\mathrm{Reach}_A = \{q_i \in Q \mid \exists\ \alpha = q_0, \ldots, q_i\}.$$

A state is **reachable** if there is some execution that reaches it.

The **safety verification** problem can be restated as checking $\mathrm{Reach}_A \cap U = \emptyset$?



Computing $Reach_A$ in Verse

$Post_A(S) = \{q' \in Q \mid \exists\, q \in S, (q', q) \in D\}$

States that can be reached from S in a single transition

Fact. if $S_1 \subseteq S_2, Post_A(S_1) \subseteq Post_A(S_2)$ [Monotonicity]

Define. $Post_A^0(S) = S$; $Post_A^k(S) = Post_A(Post_A^{k-1}(S))$

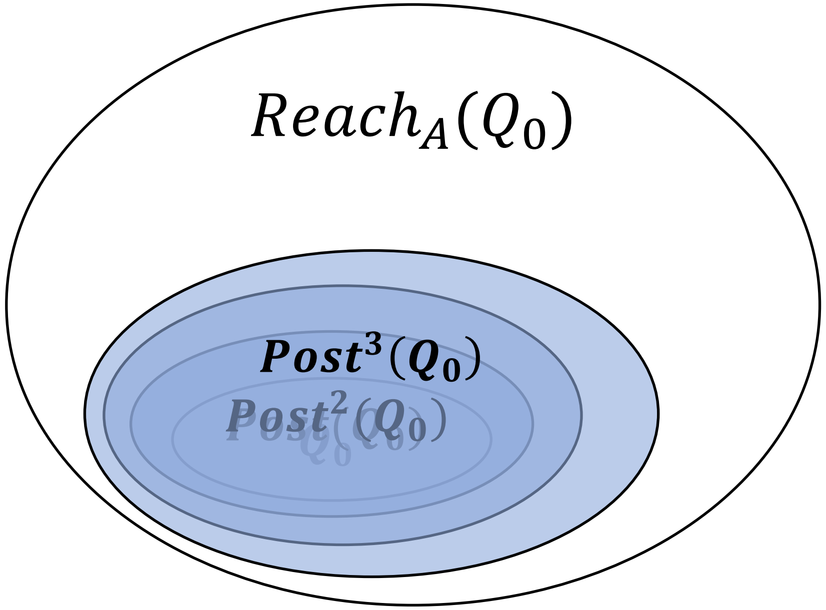
Exercise*. $Post_A^k(Q_0)$ = States reachable after k steps

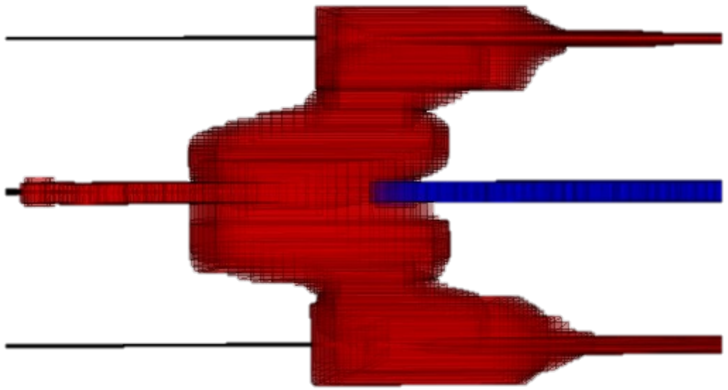
If $Post_A^k$ converges, then we could compute $Reach_A$

We can compute states that are reachable up to a time bound T and prove bounded safety

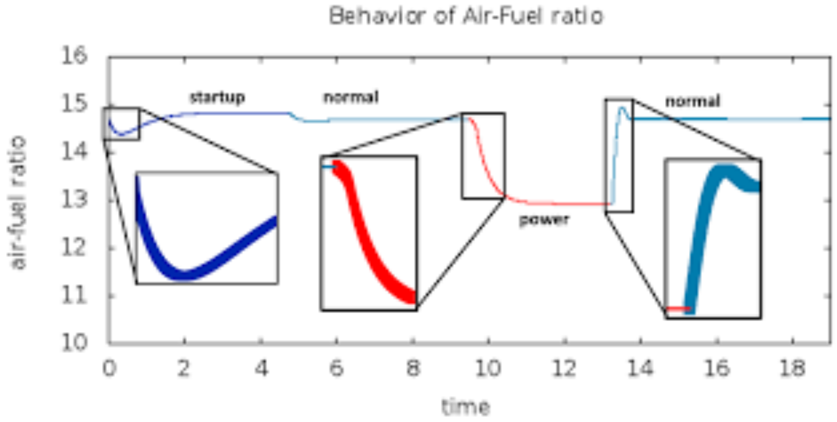
This is the strategy implemented in the Verse tool

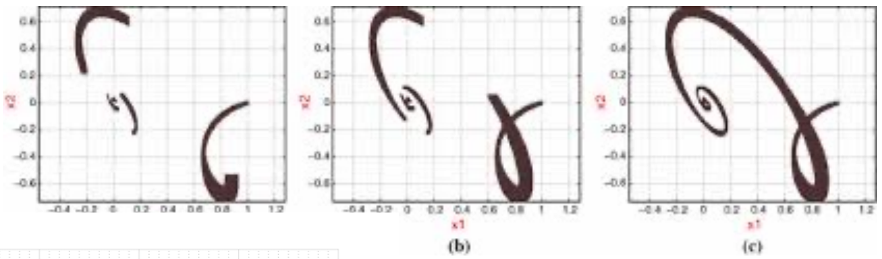
```
traces = scenario.verify(40, 0.1, params={"bloating_method": 'GLOBAL’})
fig = reachtube_tree(traces, tmp_map, fig, 1, 2, [1, 2], 'lines', 'trace')
```

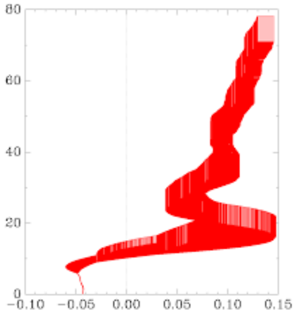


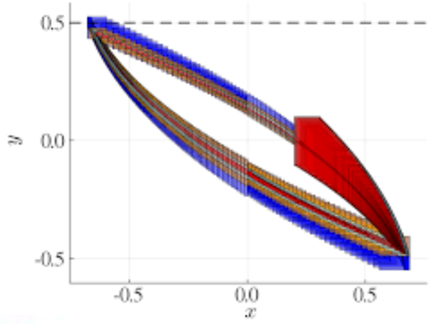


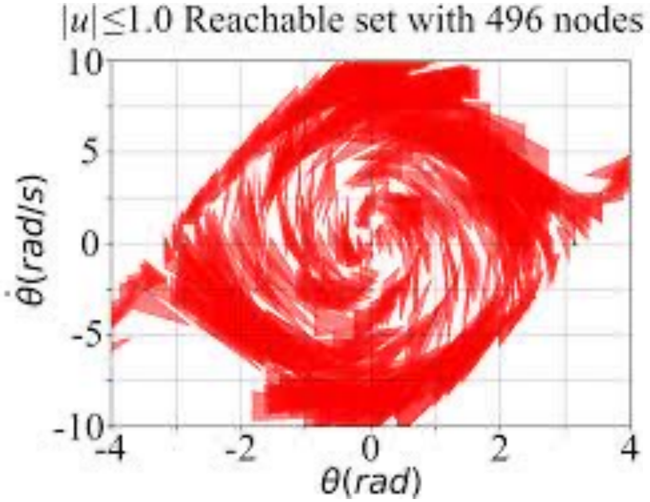
For general automata, computing $Reach_A$ is hard (undecidable)

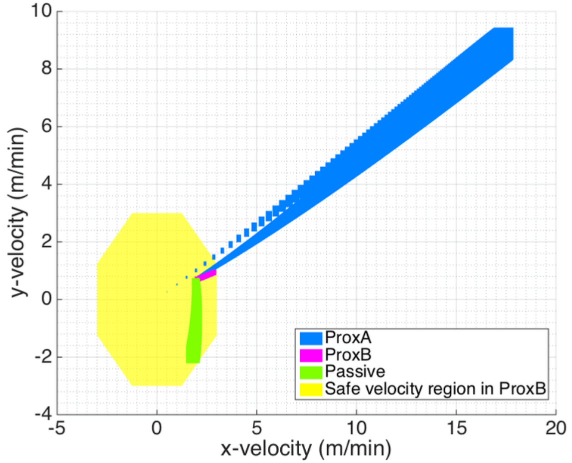


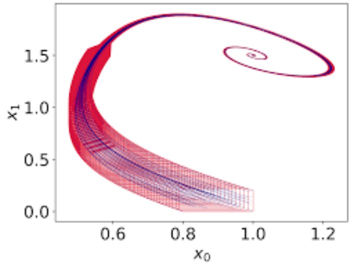












Summary

- Automata models in general have many, many behaviors / executions
- Safety requires us to show that all the possible behaviors stay away from bad states (given as safety requirements)
- For systems with complete models we seek to provide evidence for safety by checking $\mathbf{Reach}_A \cap Bad\ set = \emptyset$
- Verse implements this for models/scenarios described using Python and ODEs

Verse Tutorial and Extra Slides

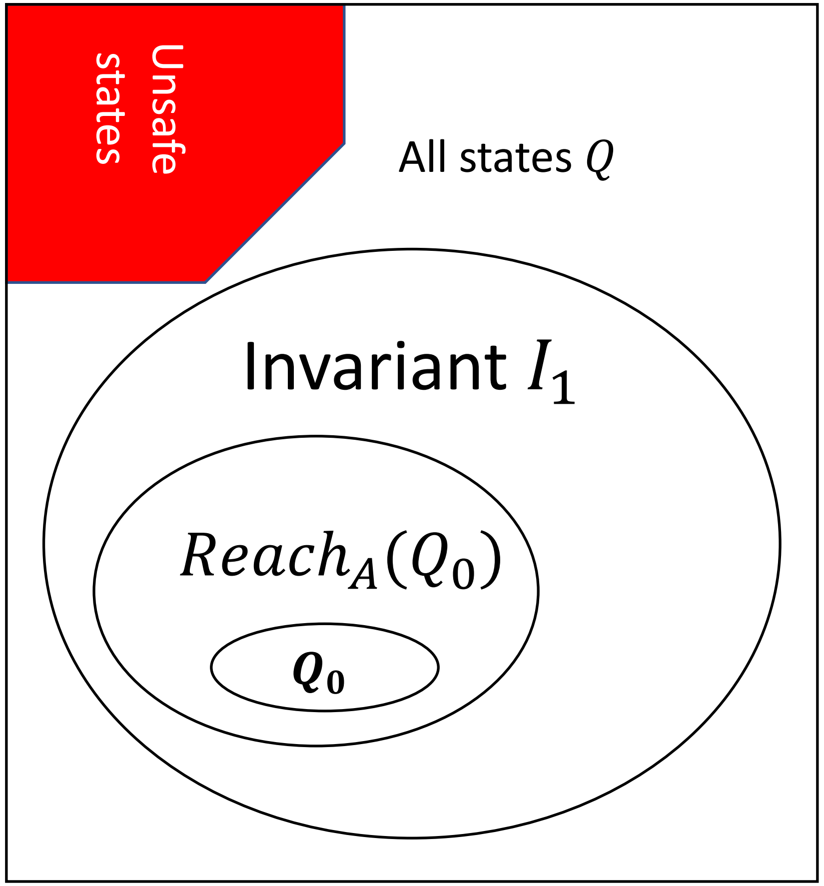
Approximating reachable states is enough for safety

For general automata, computing $Reach_A$ is hard (undecidable)

Notice, even if we can over-approximate $\mathbf{Reach}_A$ that can be adequate.

Definition. An **invariant** for A is any set of states that over-approximates $\mathbf{Reach}_A$. That is, $\mathbf{Reach}_A \subseteq I$.

Q is an invariant, but it is not particularly useful.



Our strategy for safety verification

- Find an invariant set of states $I \subseteq Q$ of A such that $I \cap U = \emptyset$
- How to check that a $I \subseteq Q$ is an invariant of A?

Theorem 1. Given automaton $A = \langle Q, Q_0, \mathcal{D} \rangle$ and a set of states $I \subseteq Q$ if:

- (Start condition) $\mathrm{Q}_0 \subseteq I$, and
- (Transition closure) $\mathrm{Post(I)} \subseteq I$

then I is an invariant of A. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

Theorem 1. Given automaton $A = \langle Q, Q_0, \mathcal{D} \rangle$ and a set of states $I \subseteq Q$ if:

- (Start condition) $\mathrm{Q}_0 \subseteq I$, and
- (Transition closure) $\mathrm{Post(I)} \subseteq I$

then I is an invariant of A. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

Proof. Consider any reachable state $q \in Reach_A$. We will have to show that q is also in I. By the definition of a reachable state, there exists an execution α of $\mathcal{A}$ such that $\alpha(k) \ = q$.

We proceed by induction on the length α

For the base case, α consists of a single starting state $\alpha = q \in \mathrm{Q}_0$, because executions always start at Q_0. And by the Start condition, $q \in I$.

For the inductive step, $\alpha = \alpha' q$ where α' is the prefix or a shorter execution. By the induction hypothesis, we know that the last state of $\alpha' say\ \ q' \in I$.

Invoking Transition condition on $q' \rightarrow q$ we obtain $q \in I$. QED

Back to the bouncing ball

$I_1 = \{\langle x, v\rangle | x \leq h\}$

Can we show that I_1 is an invariant using the Theorem 1?

We have to check

(Start condition) $\mathrm{Q}_0 \subseteq I_1$. Initially $x = h \leq h$ and $v = 0$ but does not matter \checks out

(Transition closure) $\mathrm{Post}(\mathrm{I}_1) \subseteq I_1$

- For any state with $x \leq h$, can we show that $x' \leq h$?
- NO! If the velocity is positive then $x' > x$, and we cannot show the invariant

Theorem 1 is a sufficient condition for proving invariance (not necessary)

State variables
$x: \mathbb{R}$
$v: \mathbb{R}$

State transitions
$v' = v - g$
$x' = x + v - \frac{1}{2}g$
if $x = 0 \;\&\&\; v \leq 0$
$v' = -v$
else

Back to the bouncing

State variables
$x: \mathbb{R}$
$v: \mathbb{R}$

State transitions
$v' = v - g$
$x' = x + v - \frac{1}{2}g$
if $x = 0$ && $v \leq 0$
$\quad v' = -v$
else

$I_2 = \{\langle x, v\rangle | \, v^2 - 2g(h - x) = 0\}$

Can we show that I_2 is an invariant using the Theorem 1?

We have to check

- (Start condition) $Q_0 \subseteq I_2$. Initially $v^2 - 2g(h - x) = 0 - 2g(h - h) = 0$
- (Transition closure) $\mathrm{Post(I)} \subseteq I_1$
  - Consider any state (x', v') after a transition:Two cases:
  - No bounce: $v'^2 - 2g(h - x')$

$= (v - g)^2 - 2g\left(h - x - v + \frac{1}{2}g\right)$
$= v^2 + \cancel{g^2} - \cancel{2vg} - 2g(h - x) + \cancel{2vg} - \cancel{g^2} = v^2 - 2g(h - x)$ = 0

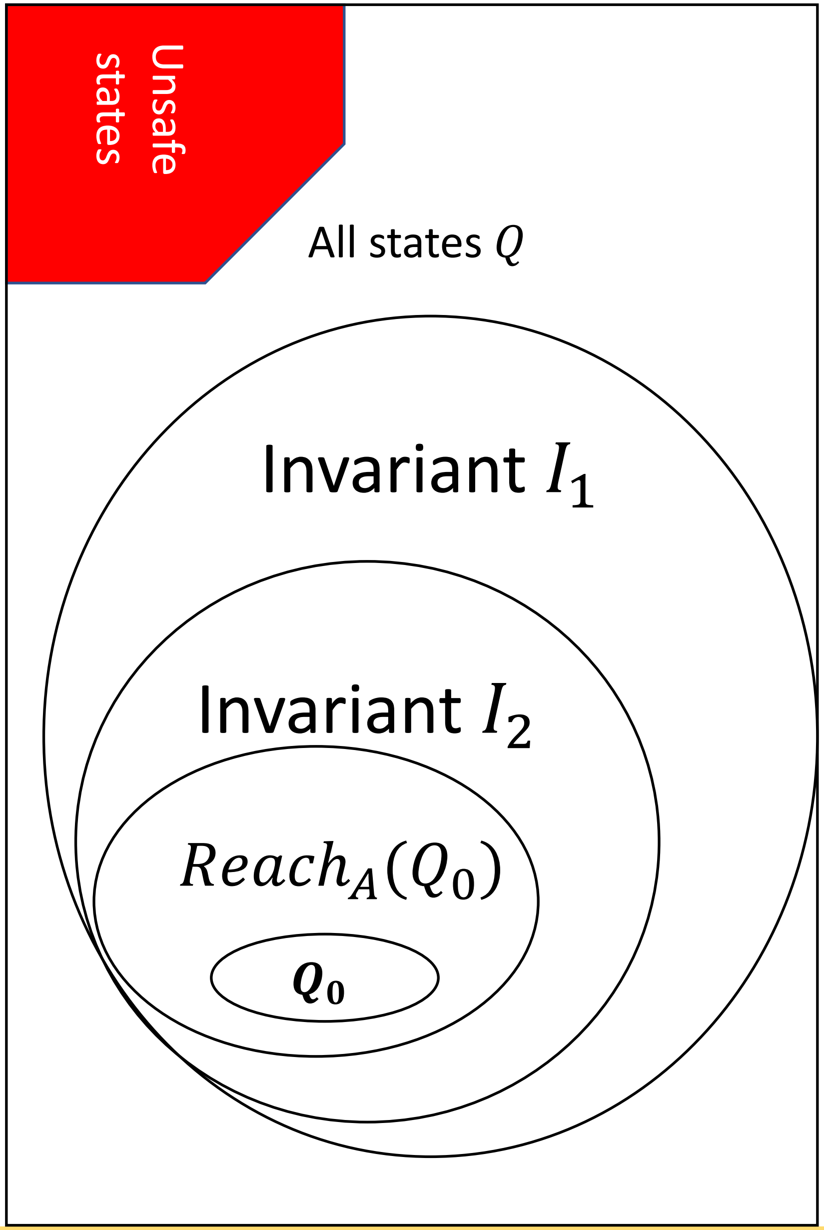
  - Bounce: If condition implies $x = 0$ *that is* $v^2 = 2gh$;

therefore $v'^2 = 2gh$

- Theorem 1 is a sufficient condition for proving invariance (not a necessary condition)

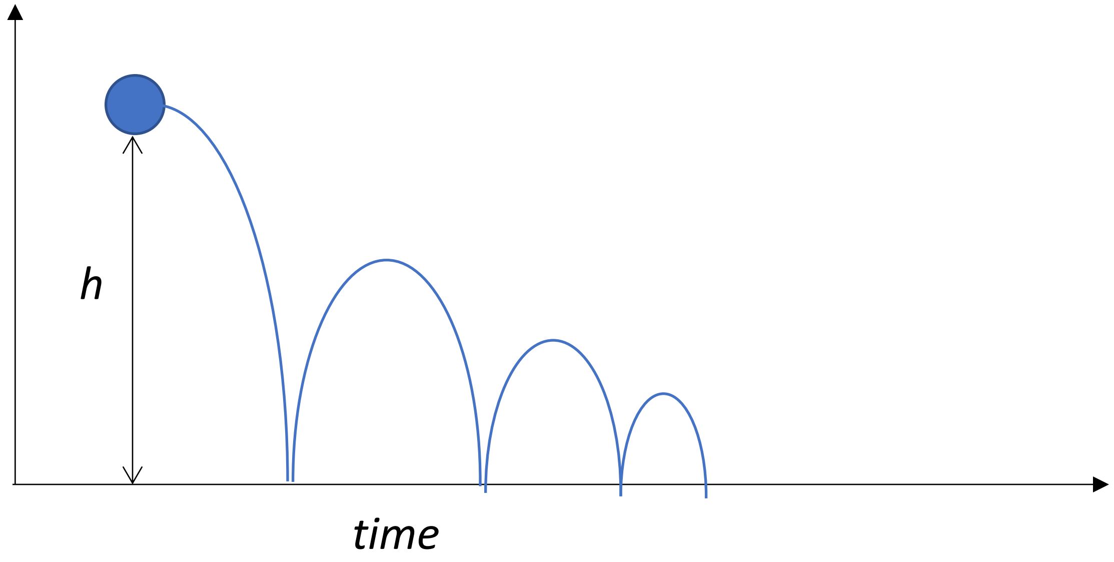
Discussion and takeaways

- I_2 has more information than I_1
  - Which is a bigger set?
- Both are adequate for proving safety ($x < h + 0.5$)
- Only I_2 could be proved with Theorem 1 (Induction), but not I_1
- Finding invariants (that can be proved by induction) still remains for us a challenging problem
  - Hot research topic: learning invariants, barrier certificates
- Still, having created a model and found an invariant now we can give an absolute safety guarantee (about all possible behaviors of the model), just by computing $Post(.)$



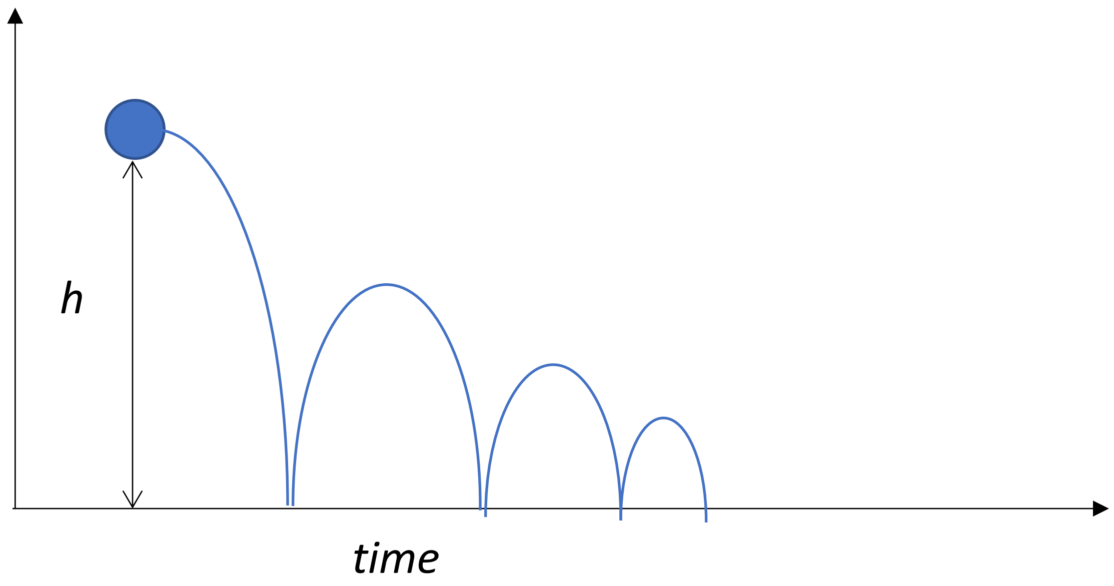
Example model of a bouncing ball

- Write the model of a ball dropped from height *h*



Example model of a bouncing ball

1. Define **states**---the *attributes* of the ball that completely define its motion: height x and velocity v
2. Define **state transitions**---how the state changes



Example model of a bouncing ball

State variables

x: $\mathbb{R}$

v: $\mathbb{R}$

State transitions

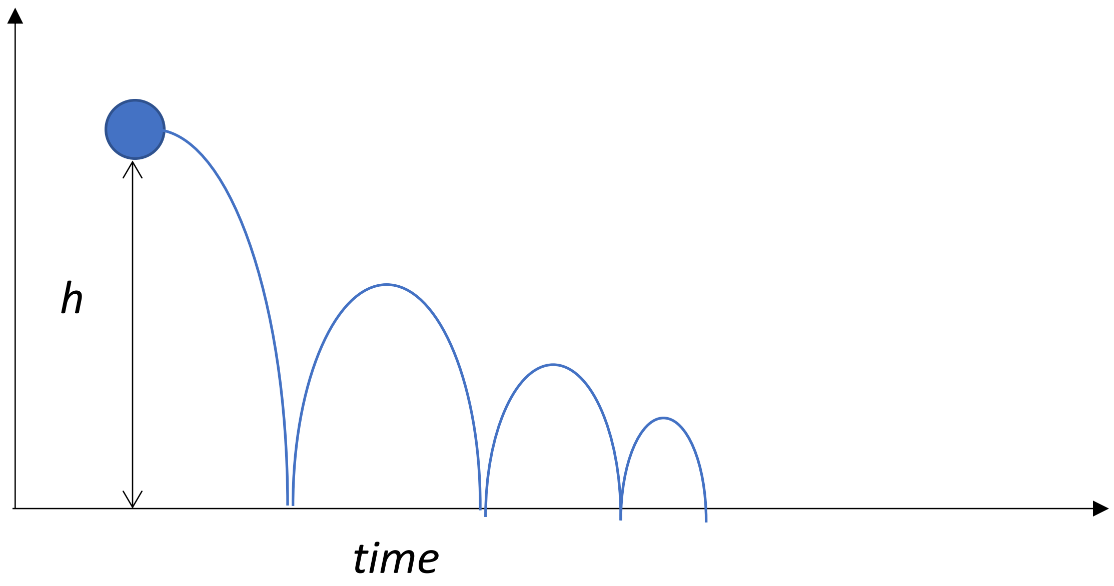
if $x \leq 0$ && $v \leq 0$

$$v' = -c \ * v$$

else $v' = v$

$v' \ = \ v - \ g \ * \ delta$

$$x' \ = \ x \ + \ delta \ * \ v - \frac{1}{2} g.delta^2$$



Parameters

h, g, c, delta

Jupyter notebook https://github.com/PoPGRI/CodeACar22/blob/main/jupyter/control_notebook/main.ipynb

Summary

- Absolute safety checking boils down to showing that none of the executions of the automaton reaches an unsafe set U
- To reason about all executions of we have to work with infinite sets of states
- One way to compute infinite sets is using the Post operator
- But, computing all executions for unbounded time can be hard
- If we can guess an invariant satisfying conditions of Proposition 1.1, that can give a shortcut for proving safety
- The inavariant may contain important information about conserved quantities, and thus, may tell us why the system is safe, and not just that it is so
- Mind the gap between model and reality
- Next. Application of invariants in braking example