Let A Neural Network Be Your Invariant

∗The authors are listed alphabetically

Mirco Giacobbe∗
University of Birmingham, UK

Daniel Kroening∗
Amazon Web Services, USA

Speaker/Presenter

Abhinandan Pal∗
University of Birmingham, UK
Amazon Web Services, USA
NII Tokyo, Japan

Michael Tautschnig∗
Amazon Web Services, USA
Queen Mary University of London, UK

SAFETY

Nothing bad
ever happens.

LIVENESS

Something Good
Eventually Happens

SAFETY

Nothing bad ever happens.

Use an Inductive Invariant.

LIVENESS

Something Good Eventually Happens

Use a Ranking Function.

Shorter Runtime are always Nicer.

Shorter Runtime are always Nicer.

Without it Developers Skip on Formal Verification

Shorter Runtime are always Nicer.

Without it Developers Skip on Formal Verification

Unfortunately, Symbolic Model checkers are Fast on only Pure Safety.

Shorter Runtime are always Nicer.

Without it Developers Skip on Formal Verification

Unfortunately, Symbolic Model checkers are Fast on only Pure Safety.

Prior Neural Model checkers focus on Pure Liveness.

Shorter Runtime are always Nicer.

Without it Developers Skip on Formal Verification

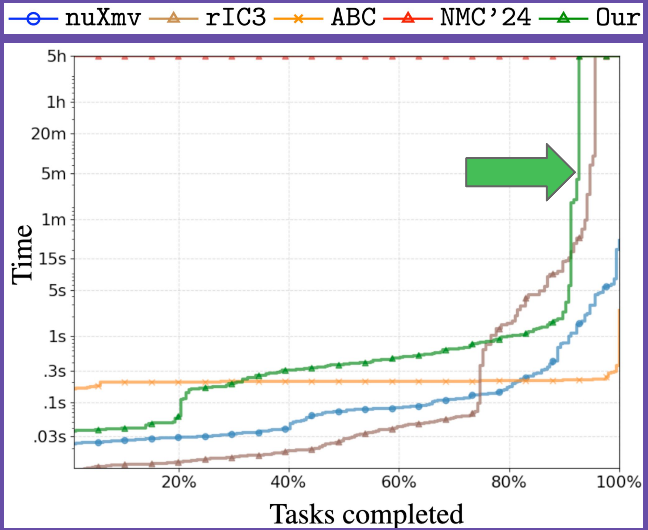
Unfortunately, Symbolic Model checkers are Fast on only Pure Safety.

Prior Neural Model checkers focus on Pure Liveness.

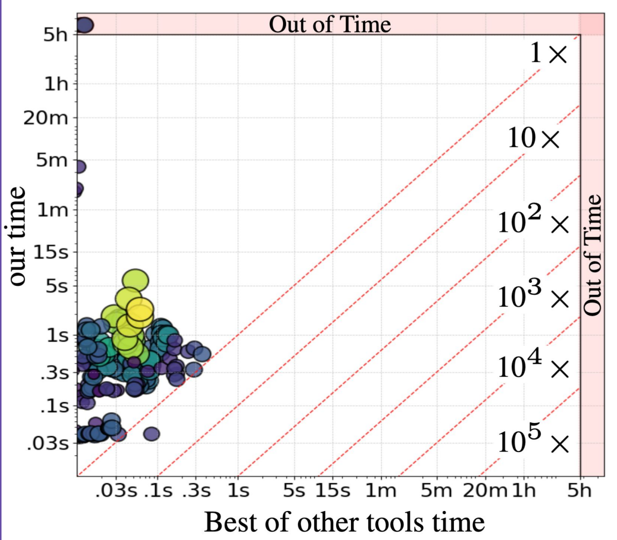
Picture a Fast Neural Model Checking Approach For both Safety and Liveness

Runtime Improvement

Pure Safety

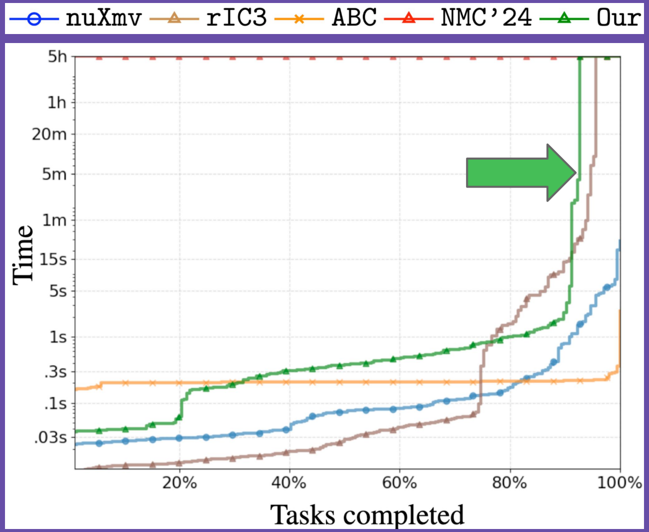


We are certainly not the fastest for pure safety—

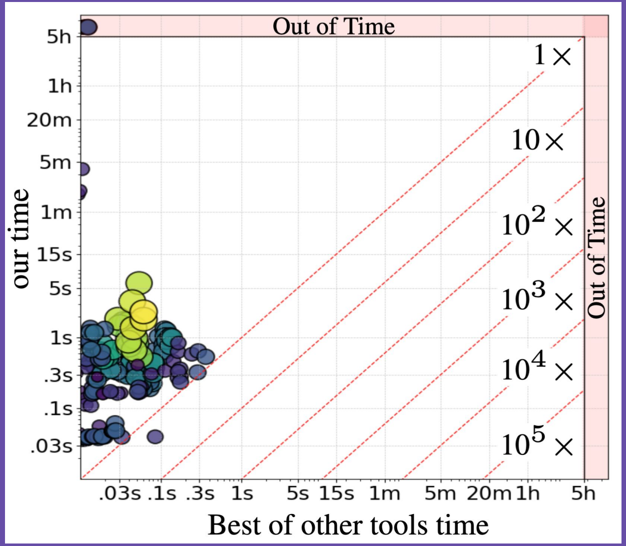


—but 80% tasks complete in 1 sec for all tools.

Pure Safety

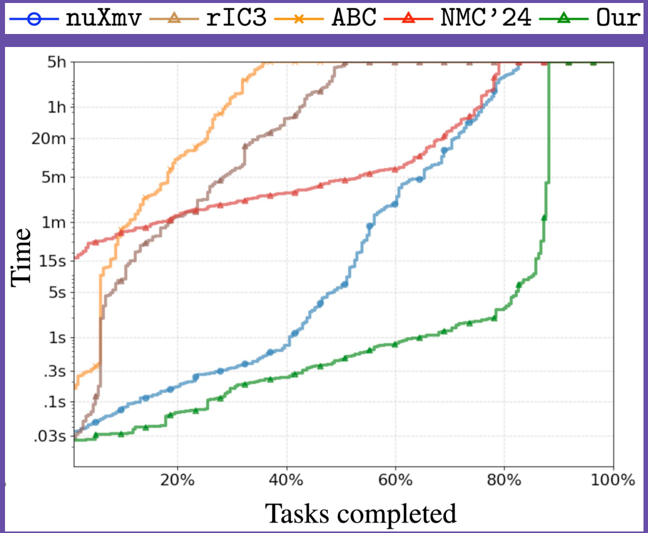


We are certainly not the fastest for pure safety—

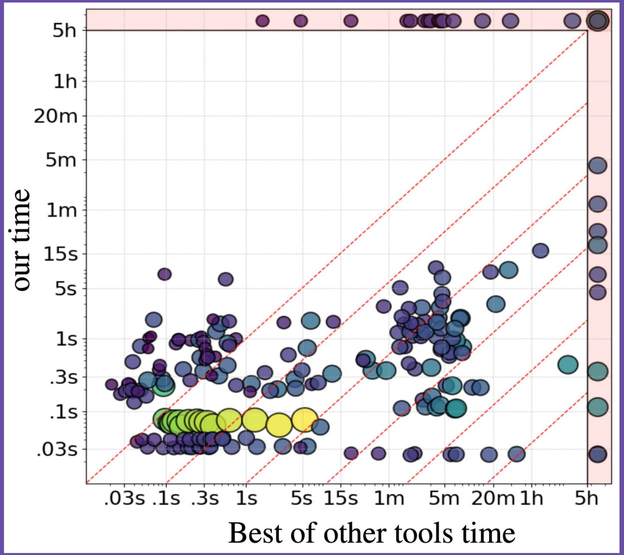


—but 80% tasks complete in 1 sec for all tools.

Pure Liveness

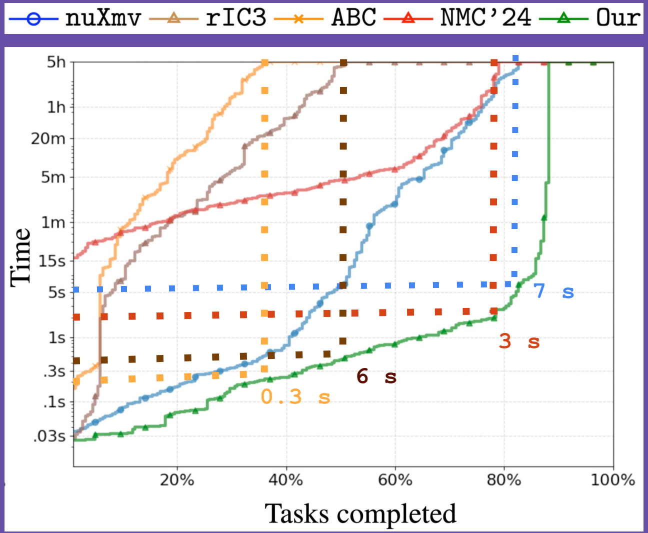


Tasks completed in 5 h per task by nuXmv, NMC'24, rIC3, and ABC are completed in under 7 s, 3 s, 0.6 s, 0.3 s by our method.

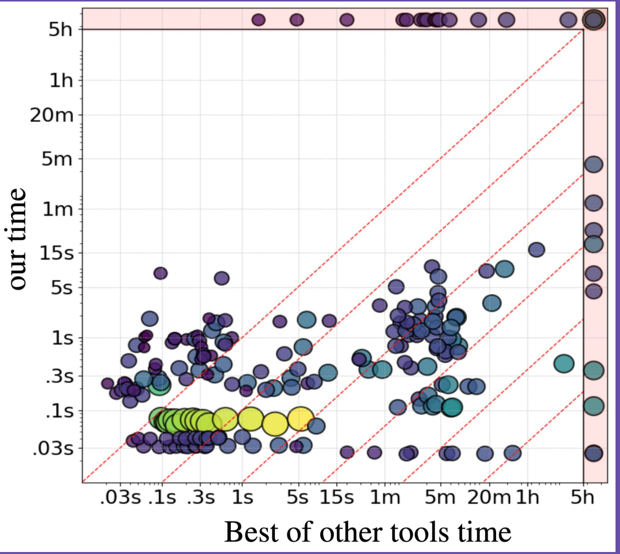


Against the per-task fastest, our method is faster on 66 % of tasks, 10× faster on 46 %, 1000× on 11 %, and 10000× on 4 %.

Pure Liveness

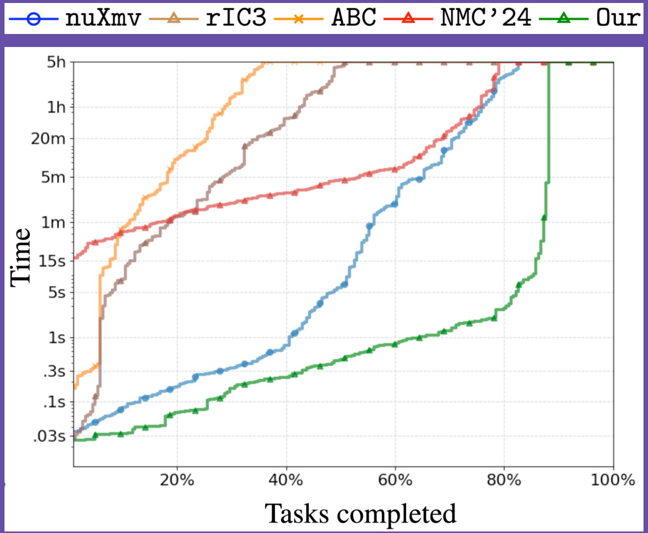


Tasks completed in 5 h per task by nuXmv, NMC'24, rIC3, and ABC are completed in under 7 s, 3 s, 0.6 s, 0.3 s by our method.

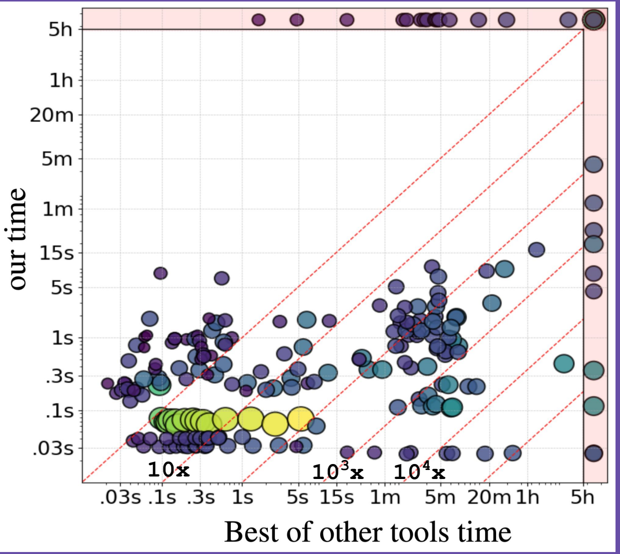


Against the per-task fastest, our method is faster on 66 % of tasks, 10× faster on 46 %, 1000× on 11 %, and 10000× on 4 %.

Pure Liveness

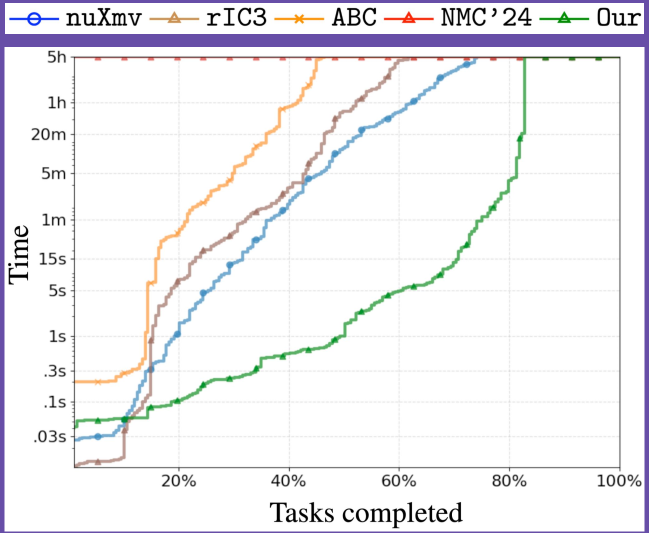


Tasks completed in 5 h per task by nuXmv, NMC'24, rIC3, and ABC are completed in under 7 s, 3 s, 0.6 s, 0.3 s by our method.

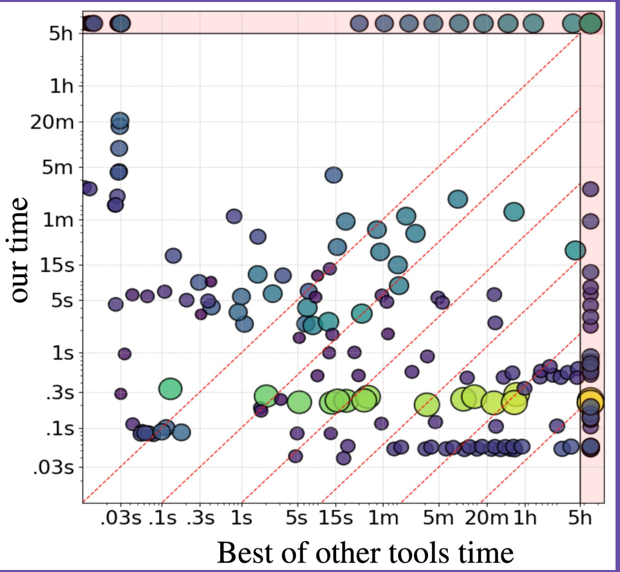


Against the per-task fastest, our method is faster on 66 % of tasks, 10× faster on 46 %, 1000× on 11 %, and 10000× on 4 %.

Safety + Liveness

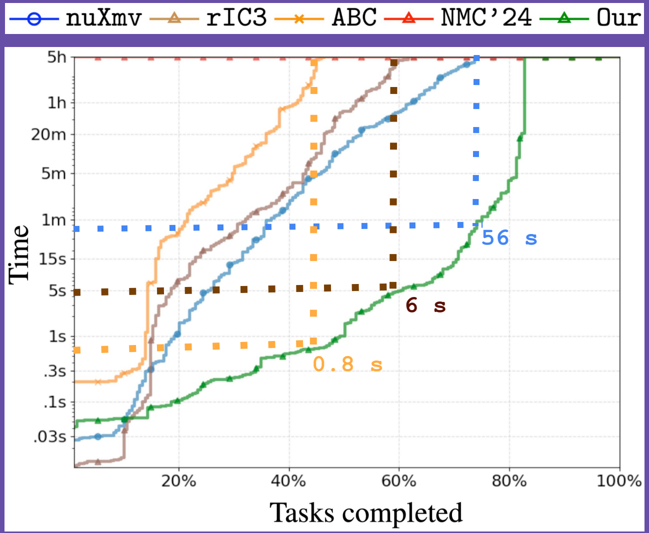


Tasks that take nuXmv, ABC or rIC3 5 h we do in 56 s, 6 s, 0.8 s per task.

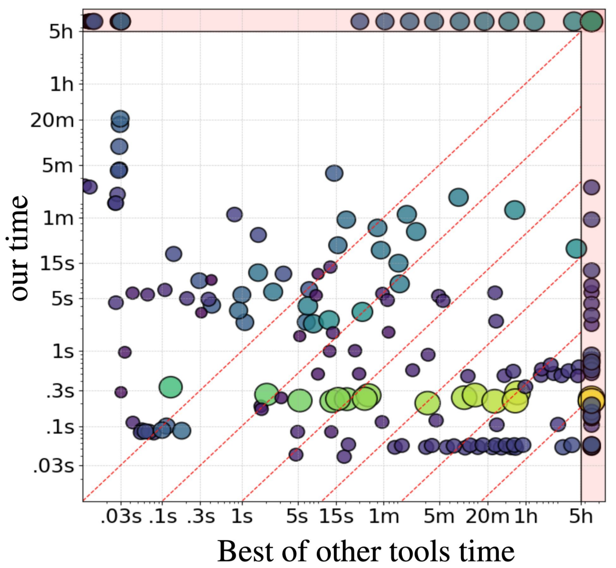


Faster than everyone else on 61 %, 100× faster on 43 %, 10000× faster on 27 %, and 100000× faster on 6 %.

Safety + Liveness

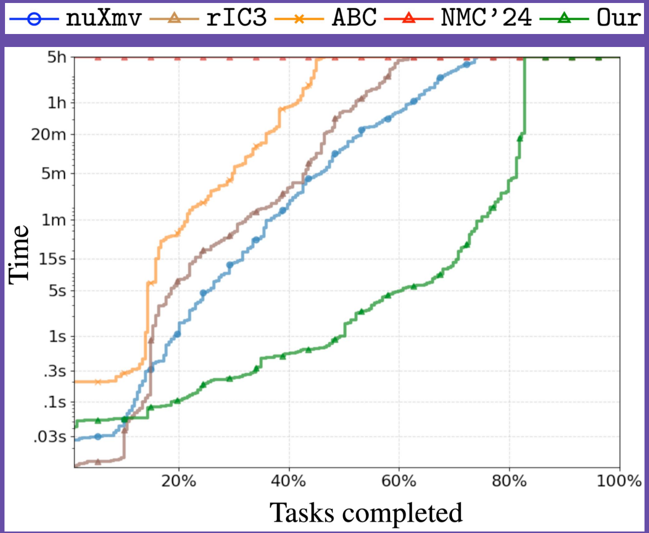


Tasks that take nuXmv, ABC or rIC3 5 h we do in 56 s, 6 s, 0.8 s per task.

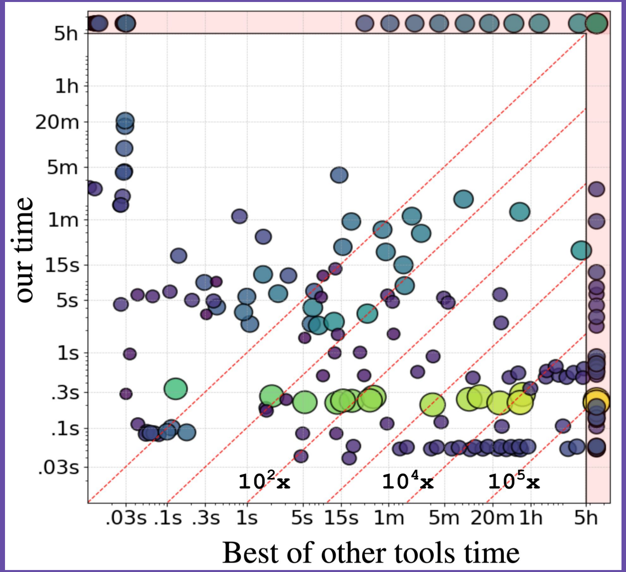


Faster than everyone else on 61 %, 100× faster on 43 %, 10000× faster on 27 %, and 100000× faster on 6 %.

Safety + Liveness



Tasks that take nuXmv, ABC or rIC3 5 h we do in 56 s, 6 s, 0.8 s per task.

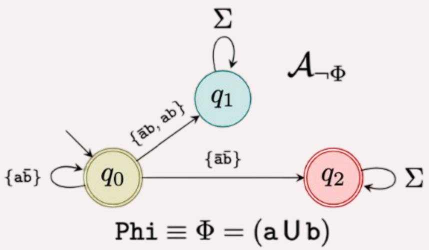


Faster than everyone else on 61 %, 100× faster on 43 %, 10000× faster on 27 %, and 100000× faster on 6 %.

Our Approach

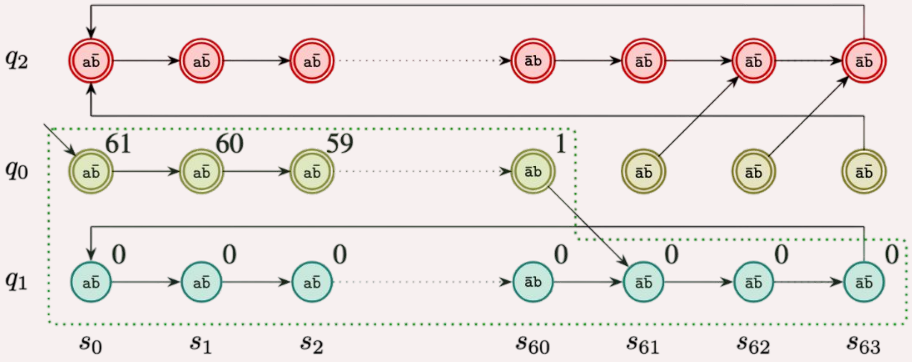
```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
       c <= c + 1;
  end
  Phi: assert property
     (@(posedge clk) a s_until b);
endmodule
```

(a)



(b)

Decide whether all trances of a **System Verilog** Design satisfies an **LTL specification**.



(c)

$$\boldsymbol{s} \in S_0 \implies (\mathrm{reg}\,\boldsymbol{s}, q_0) \in I, \tag{1}$$

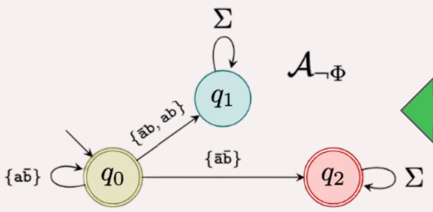
$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \wedge (\mathrm{reg}\,\boldsymbol{s}, q) \in I \implies (\mathrm{reg}\,\boldsymbol{s}', q') \in I, \tag{2}$$

$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \wedge (\mathrm{reg}\,\boldsymbol{s}, q) \in I \implies V(\mathrm{reg}\,\boldsymbol{s}, q) \succeq V(\mathrm{reg}\,\boldsymbol{s}', q'), \tag{3}$$

$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \wedge (\mathrm{reg}\,\boldsymbol{s}, q) \in I \wedge q \in F \implies V(\mathrm{reg}\,\boldsymbol{s}, q) \succ V(\mathrm{reg}\,\boldsymbol{s}', q'). \tag{4}$$

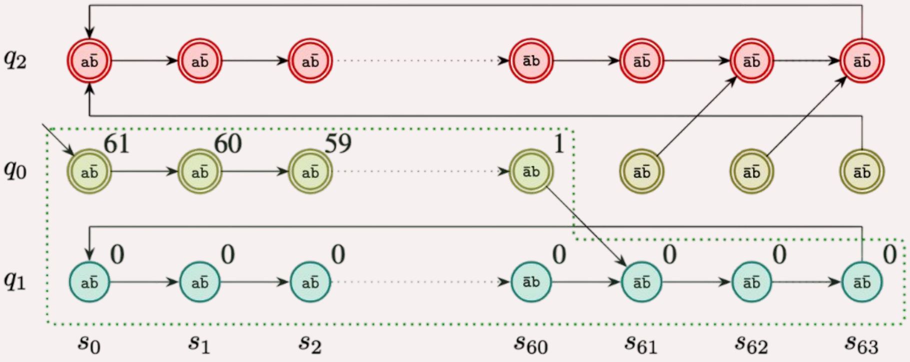
```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
      c <= c + 1;
  end
  Phi: assert property
    (@(posedge clk) a s_until b);
endmodule
```

(a)



$\texttt{Phi} \equiv \Phi = (\mathsf{a}\,\mathsf{U}\,\mathsf{b})$

(b)



(c)

We encode the **negated specification** as an **ω**-regular automaton.

An **ω**-regular automaton's accepting traces **visit fair states infinitely often**.

$$\boldsymbol{s} \in S_0 \implies (\mathrm{reg}\,\boldsymbol{s}, q_0) \in I, \tag{1}$$

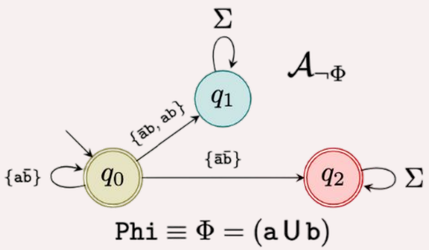
$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I \implies (\mathrm{reg}\,\boldsymbol{s}', q') \in I, \tag{2}$$

$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I \implies V(\mathrm{reg}\,\boldsymbol{s}, q) \succeq V(\mathrm{reg}\,\boldsymbol{s}', q'), \tag{3}$$

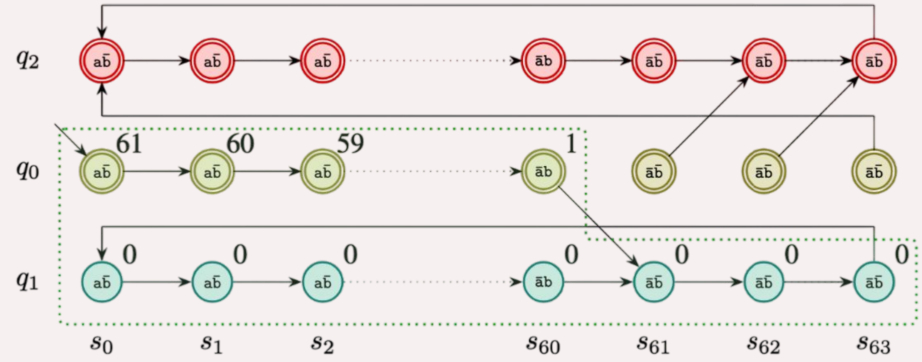
$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I \;\wedge\; q \in F \implies V(\mathrm{reg}\,\boldsymbol{s}, q) \succ V(\mathrm{reg}\,\boldsymbol{s}', q'). \tag{4}$$

```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
      c <= c + 1;
  end
  Phi: assert property
    (@(posedge clk) a s_until b);
endmodule
```

(a)



(b)



(c)

The **synchronous product** of model and negation automaton **accepts violation of the spec.**

$$\boldsymbol{s} \in S_0 \implies (\mathrm{reg}\ \boldsymbol{s}, q_0) \in I, \tag{1}$$

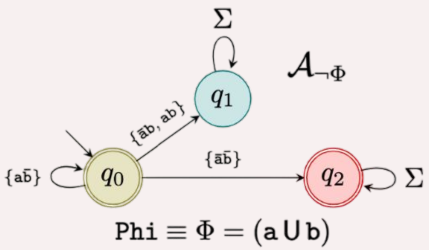
$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\ \boldsymbol{s}, q) \in I \implies (\mathrm{reg}\ \boldsymbol{s}', q') \in I, \tag{2}$$

$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\ \boldsymbol{s}, q) \in I \implies V(\mathrm{reg}\ \boldsymbol{s}, q) \succeq V(\mathrm{reg}\ \boldsymbol{s}', q'), \tag{3}$$

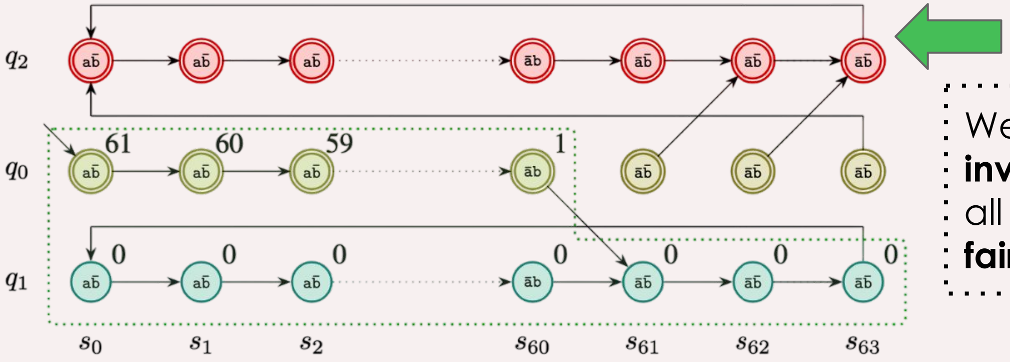
$$(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\ \boldsymbol{s}, q) \in I \ \wedge\ q \in F \implies V(\mathrm{reg}\ \boldsymbol{s}, q) \succ V(\mathrm{reg}\ \boldsymbol{s}', q'). \tag{4}$$

```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
      c <= c + 1;
  end
  Phi: assert property
    (@(posedge clk) a s_until b);
endmodule
```

(a)



(b)



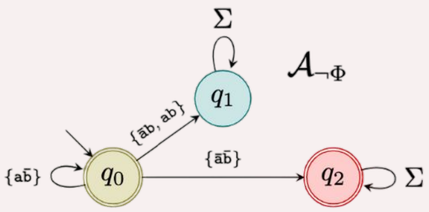
(c)

We first constructing an **invariant** that **excludes** all **cycles** containing **fair states**.

$$\begin{aligned}
\boldsymbol{s} \in S_0 \quad &\Longrightarrow (\mathrm{reg}\,\boldsymbol{s}, q_0) \in I, && (1)\\
(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\,\boldsymbol{s}, q) \in I \quad &\Longrightarrow (\mathrm{reg}\,\boldsymbol{s}', q') \in I, && (2)\\
(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\,\boldsymbol{s}, q) \in I \quad &\Longrightarrow V(\mathrm{reg}\,\boldsymbol{s}, q) \succeq V(\mathrm{reg}\,\boldsymbol{s}', q'), && (3)\\
(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \ \wedge\ (\mathrm{reg}\,\boldsymbol{s}, q) \in I \ \wedge\ q \in F \quad &\Longrightarrow V(\mathrm{reg}\,\boldsymbol{s}, q) \succ V(\mathrm{reg}\,\boldsymbol{s}', q'). && (4)
\end{aligned}$$

```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
       c <= c + 1;
  end
  Phi: assert property
    (@(posedge clk) a s_until b);
endmodule
```

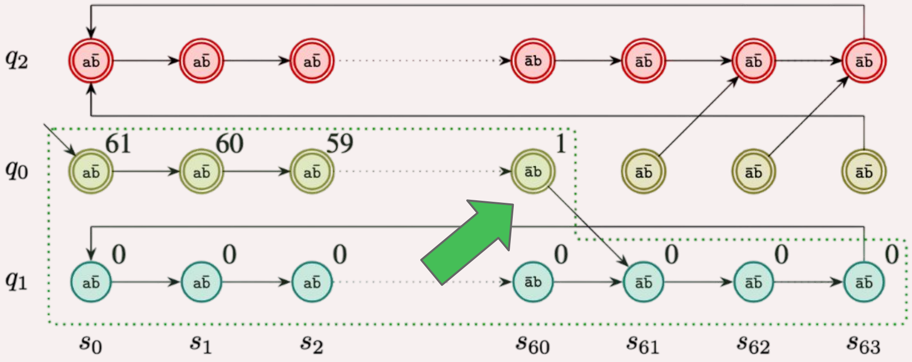
(a)



$\text{Phi} \equiv \Phi = (\mathsf{a}\,\mathsf{U}\,\mathsf{b})$

(b)

Inside this invariant we assign each state a **ranking bounded from below**, ensuring along any trace the rank **never increases** and **strictly decreases** whenever a **fair state** occurs.



(c)

Since the rank can only **decrease finitely many times**, every trace visits **fair states finitely many times** and thus cannot be accepting—which means the **property holds**.

$$s \in S_0 \implies (\mathrm{reg}\, s, q_0) \in I, \tag{1}$$

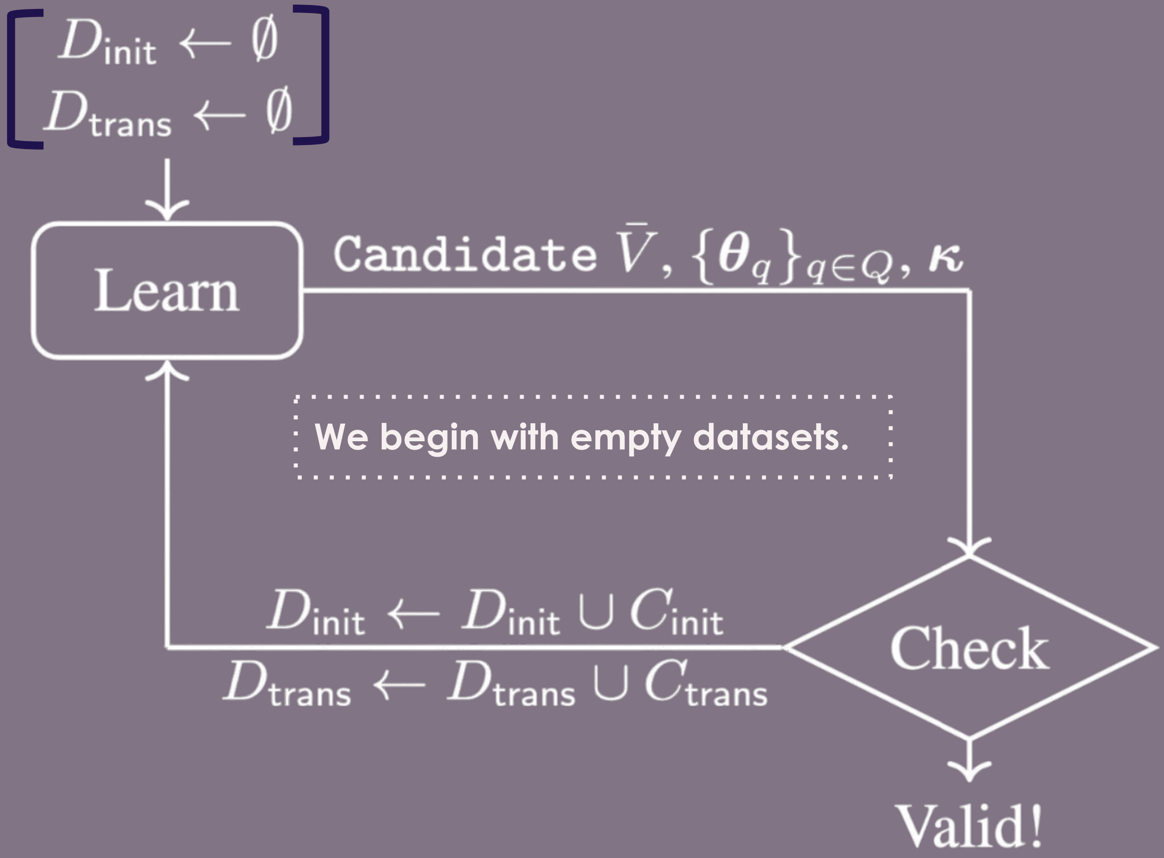
$$(s, q) \to_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (s', q') \land (\mathrm{reg}\, s, q) \in I \implies (\mathrm{reg}\, s', q') \in I, \tag{2}$$

$$(s, q) \to_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (s', q') \land (\mathrm{reg}\, s, q) \in I \implies V(\mathrm{reg}\, s, q) \succeq V(\mathrm{reg}\, s', q'), \tag{3}$$

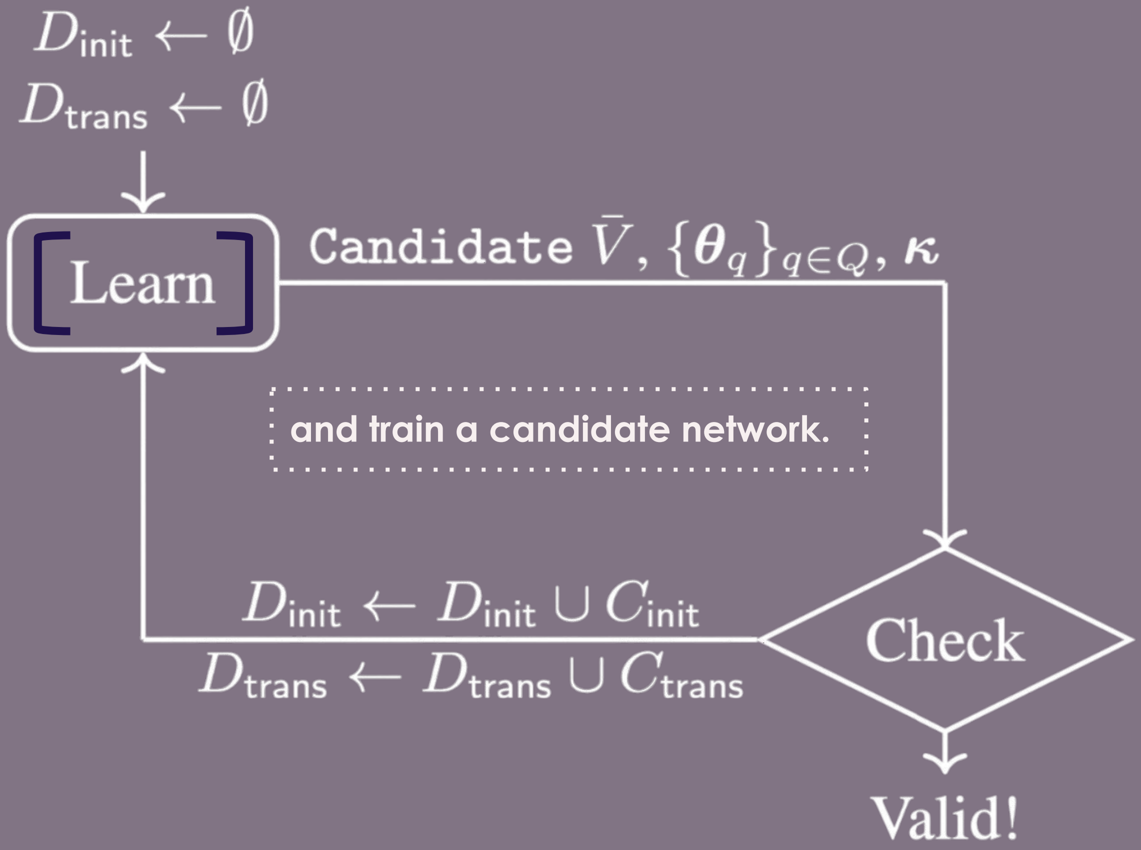
$$(s, q) \to_{\mathcal{M}\|\mathcal{A}_{\neg\Phi}} (s', q') \land (\mathrm{reg}\, s, q) \in I \land q \in F \implies V(\mathrm{reg}\, s, q) \succ V(\mathrm{reg}\, s', q'). \tag{4}$$

We learn a single function—represented by a neural network—that is both an invariant and a ranking function.

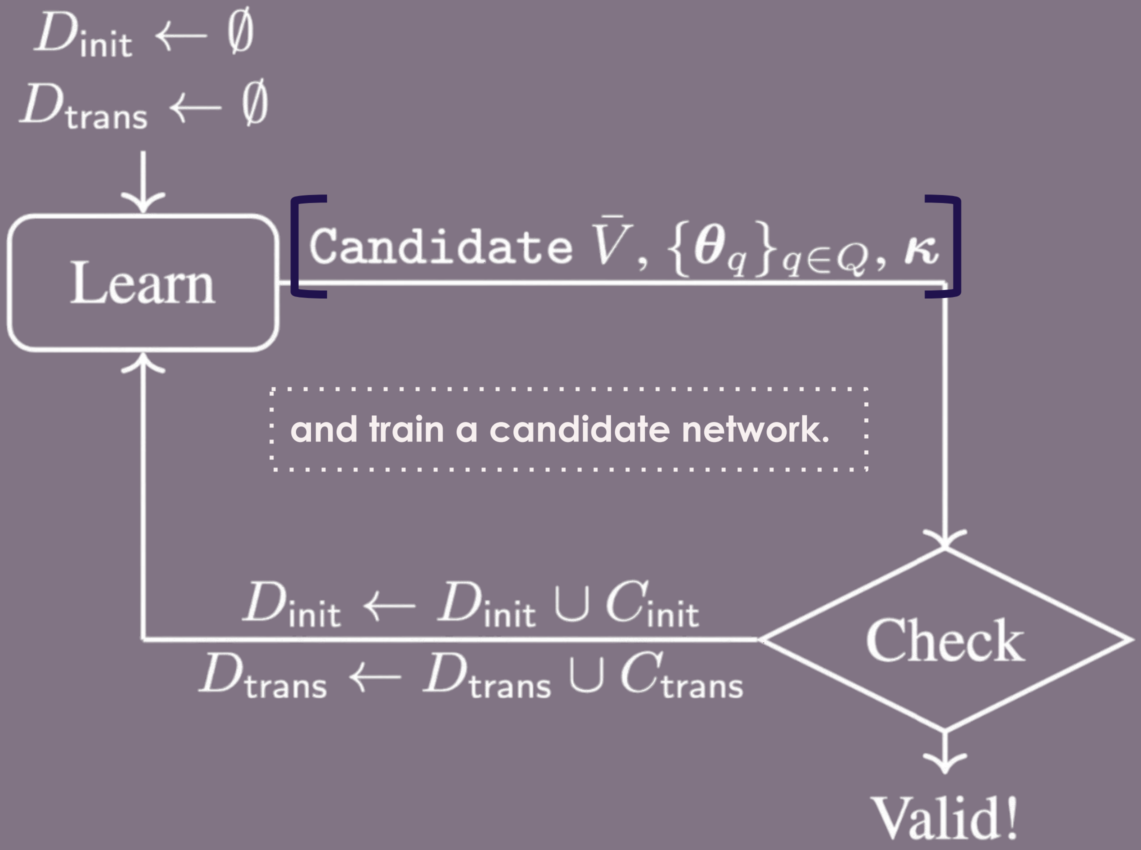
Learn Check Loop



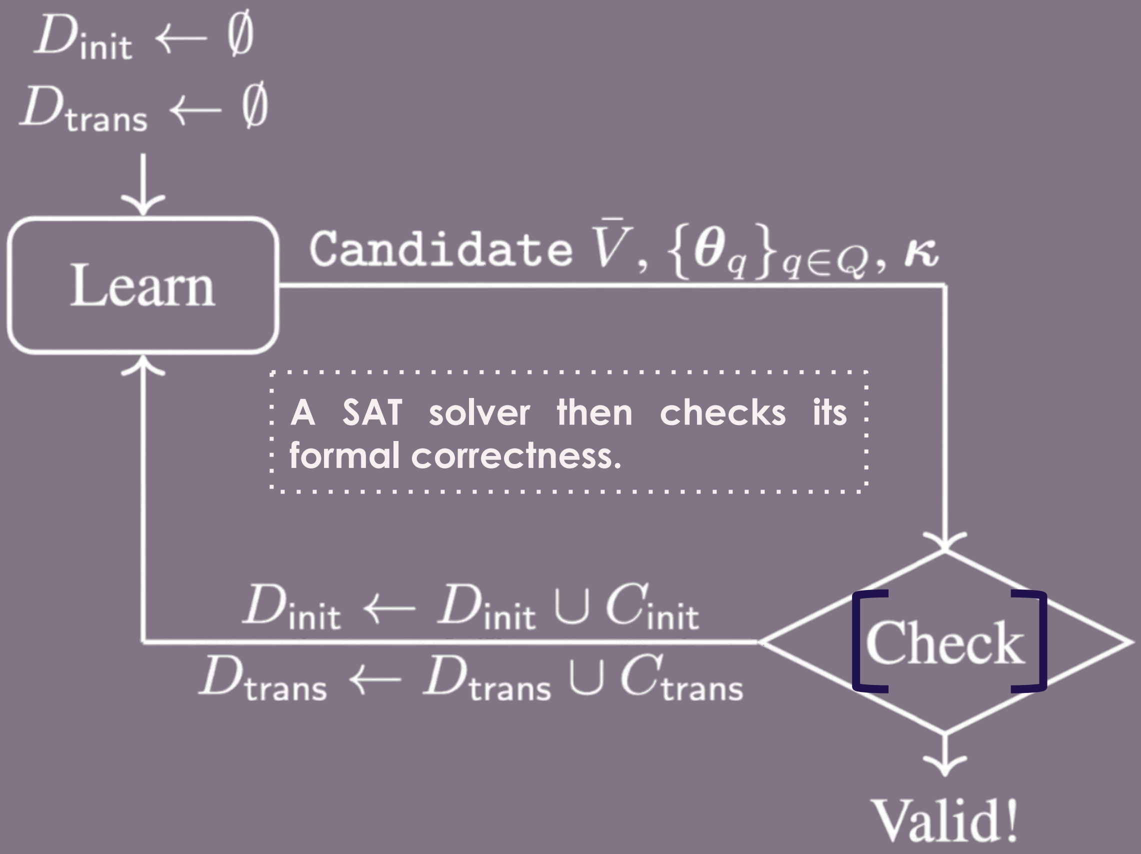
Learn Check Loop



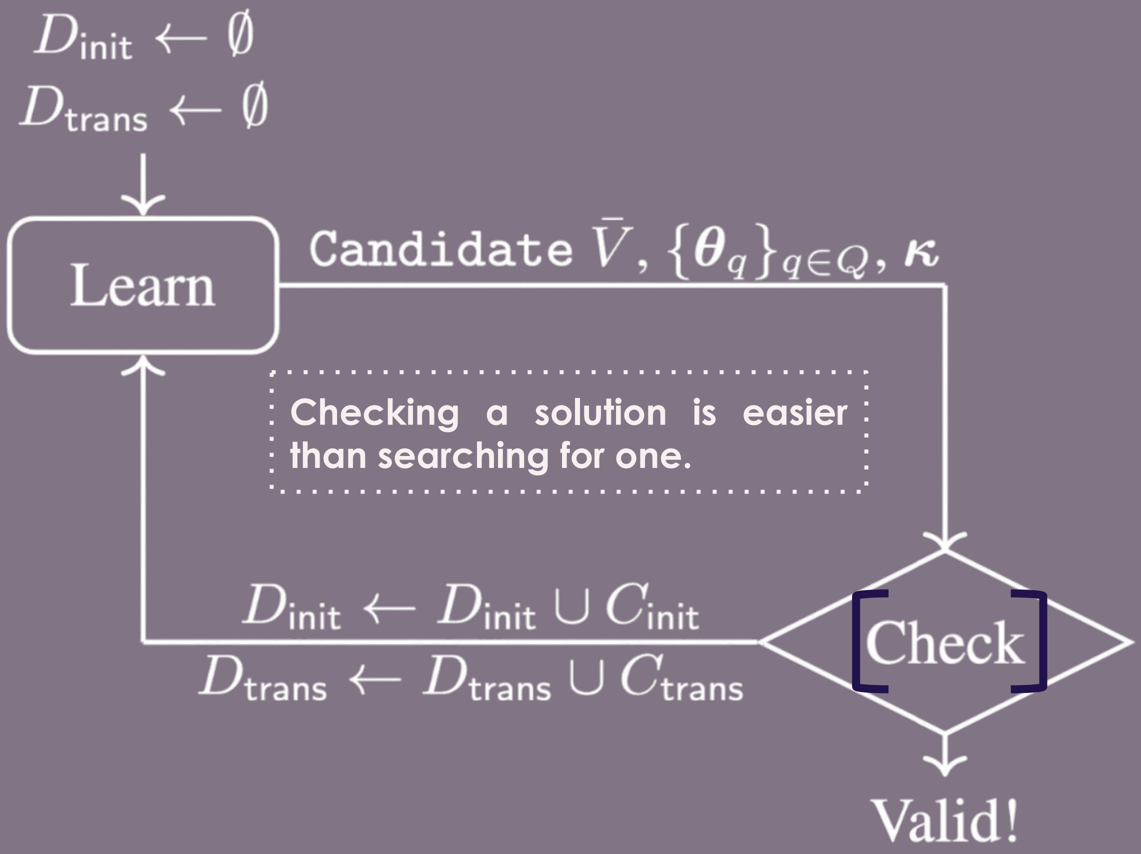
Learn Check Loop



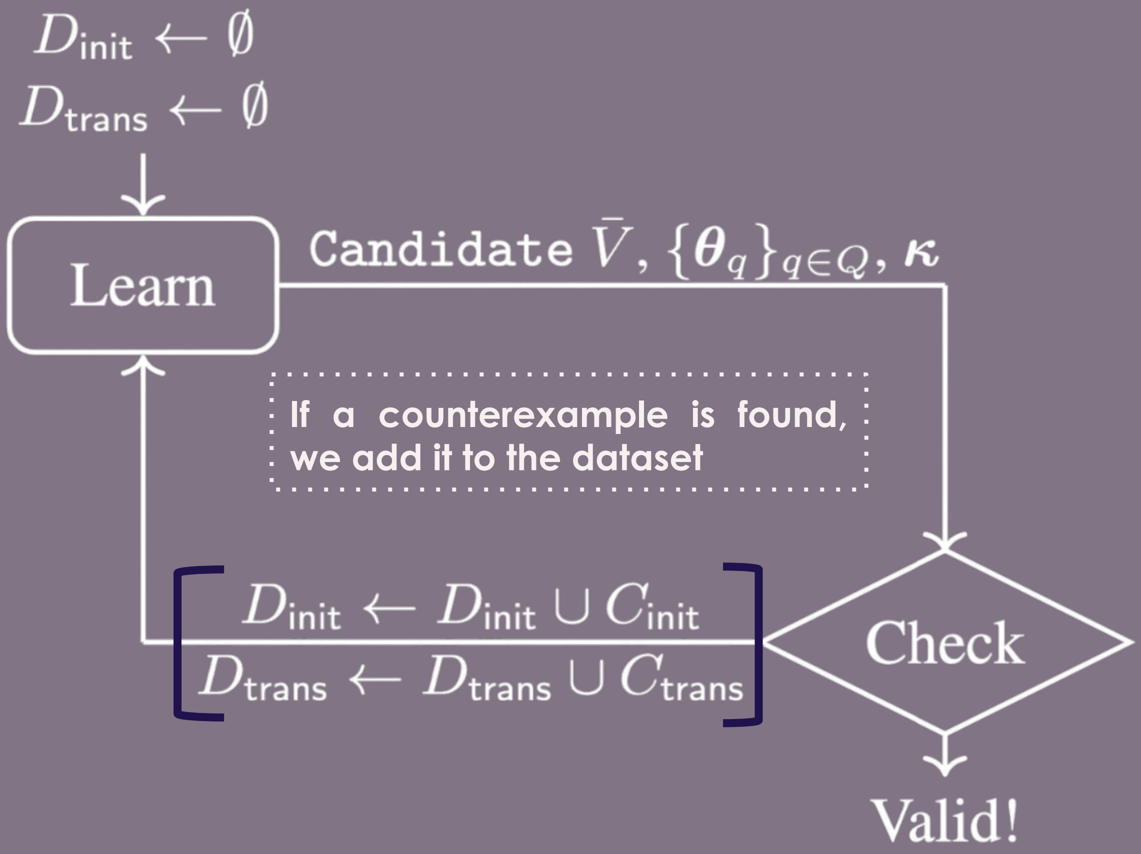
Learn Check Loop



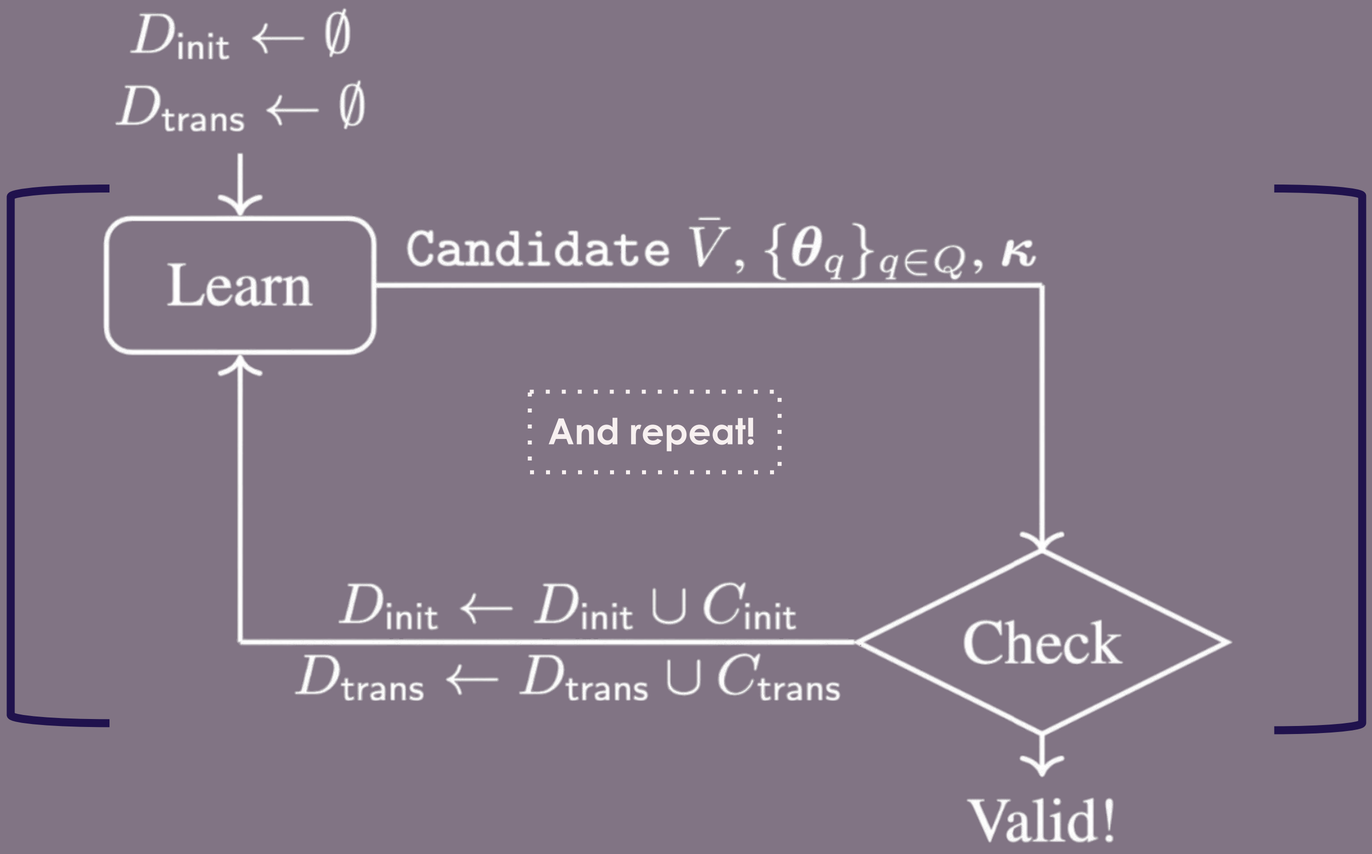
Learn Check Loop



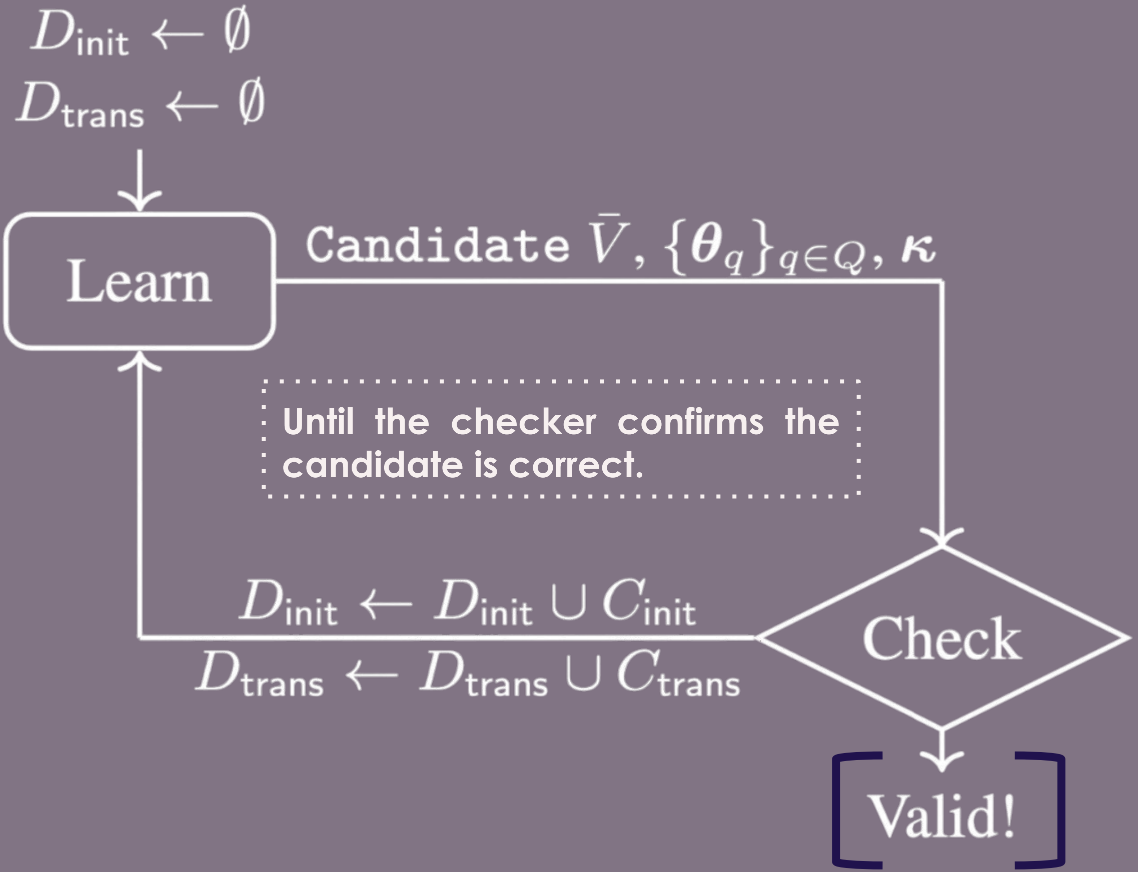
Learn Check Loop



Learn Check Loop

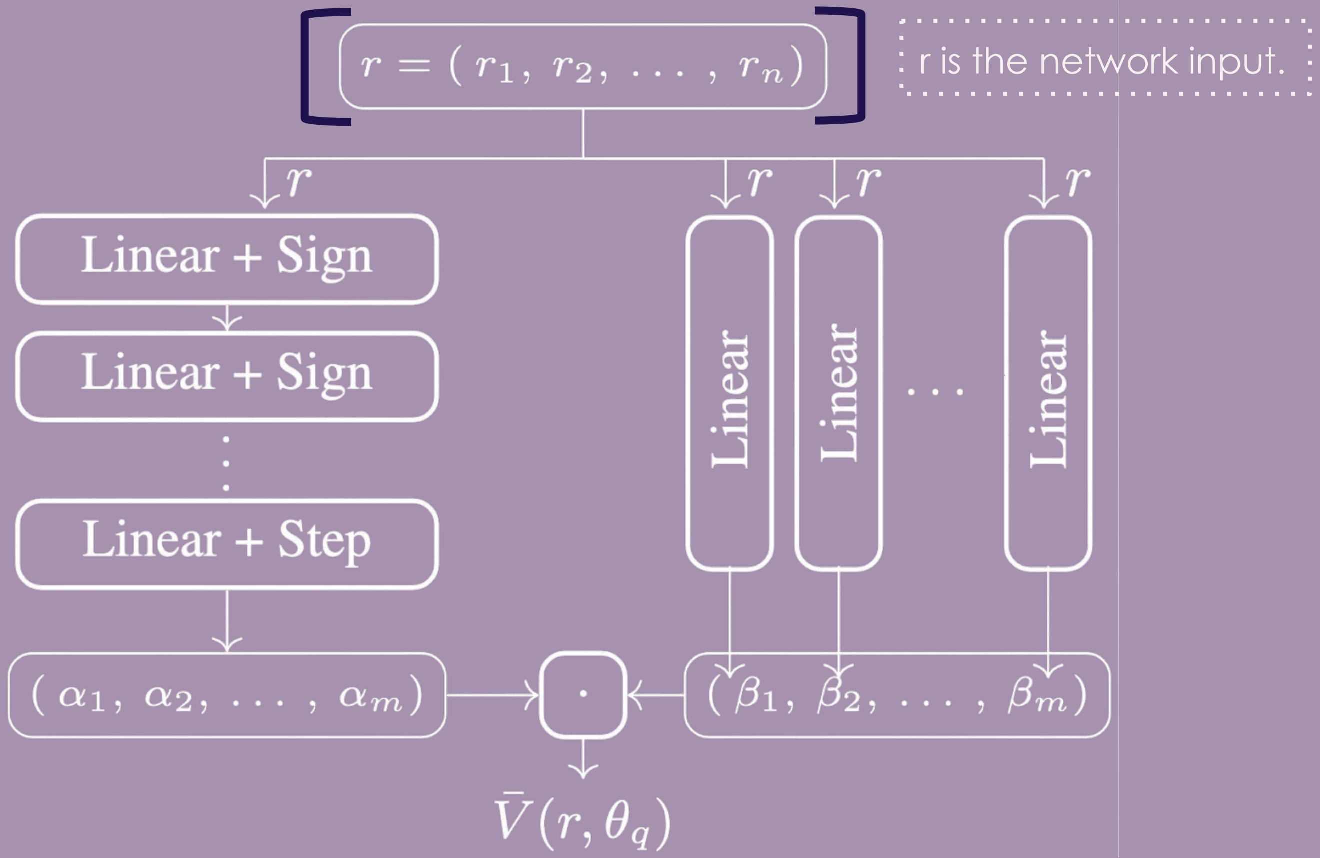


Learn Check Loop



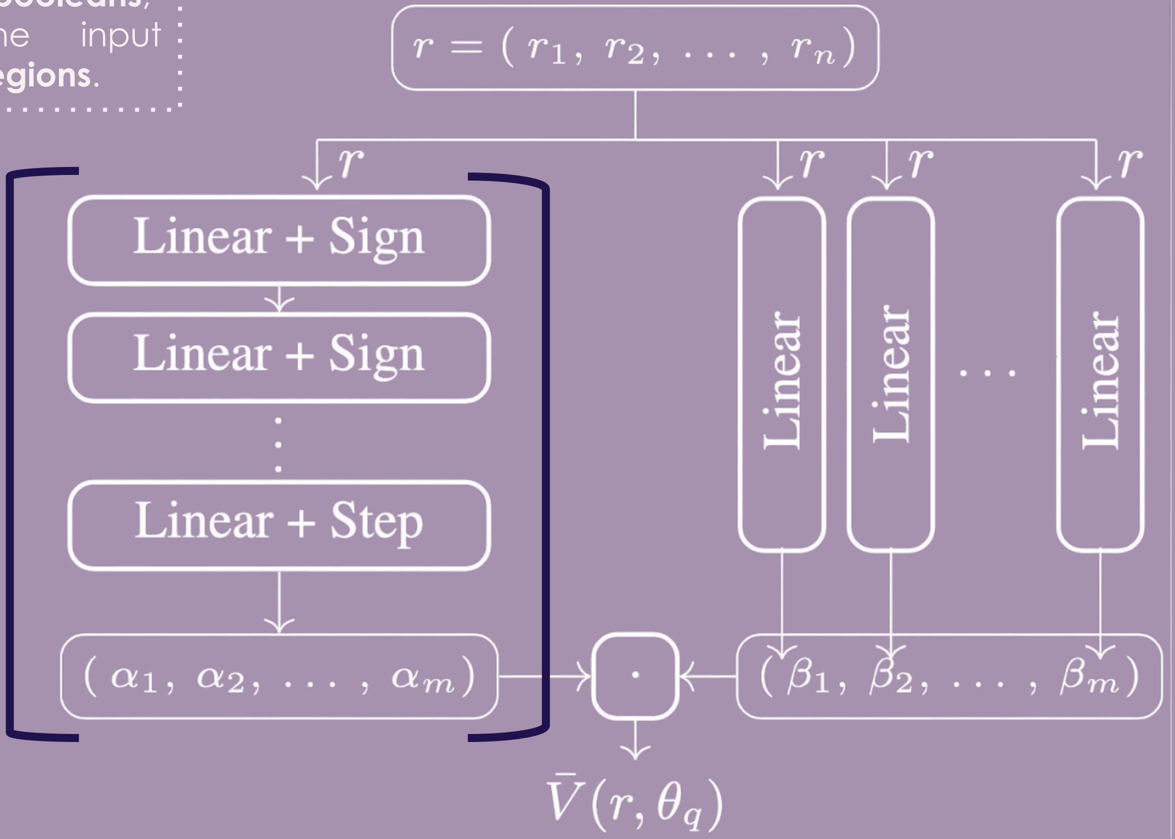
This allows for a task aware sampling approach keeping the dataset small allowing MILP learning.

Network Architecture



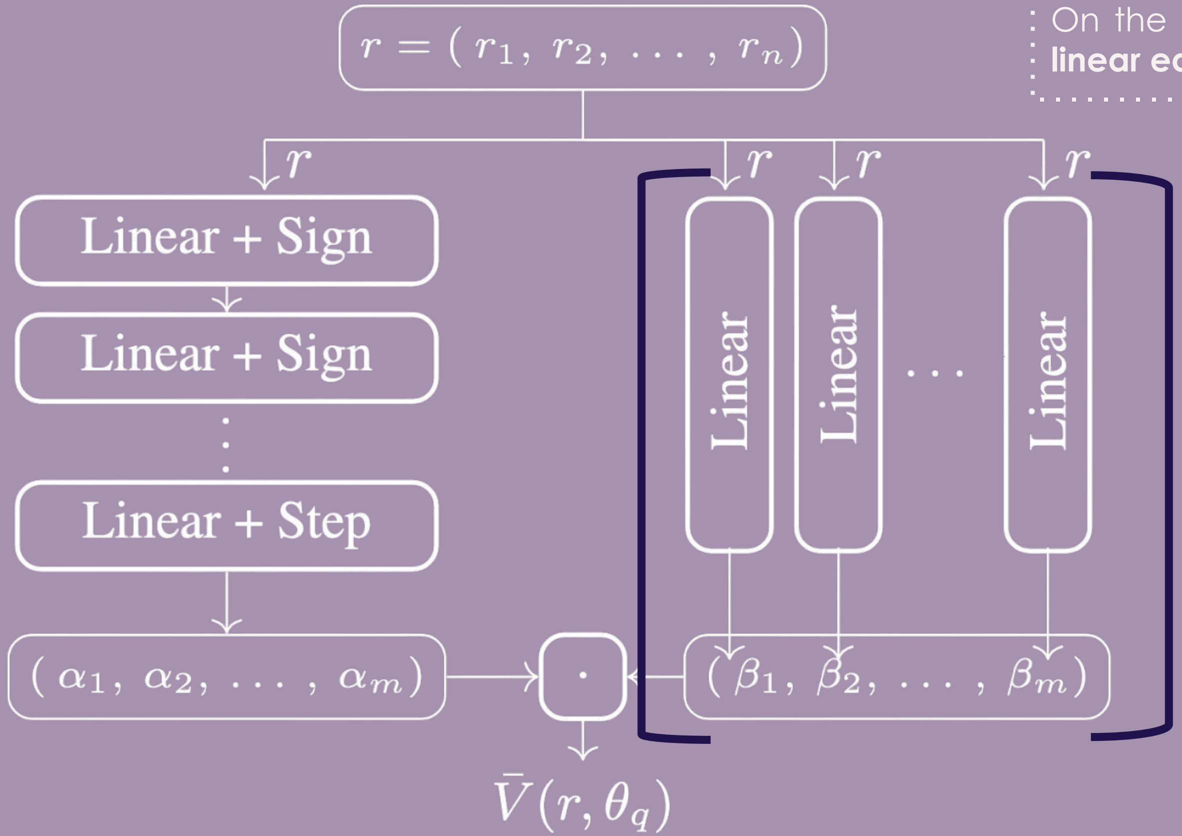
Network Architecture

On the **left**, a **fully connected** network with **sign and step** activations produces **m booleans**, partitioning the input space into **2^m regions**.

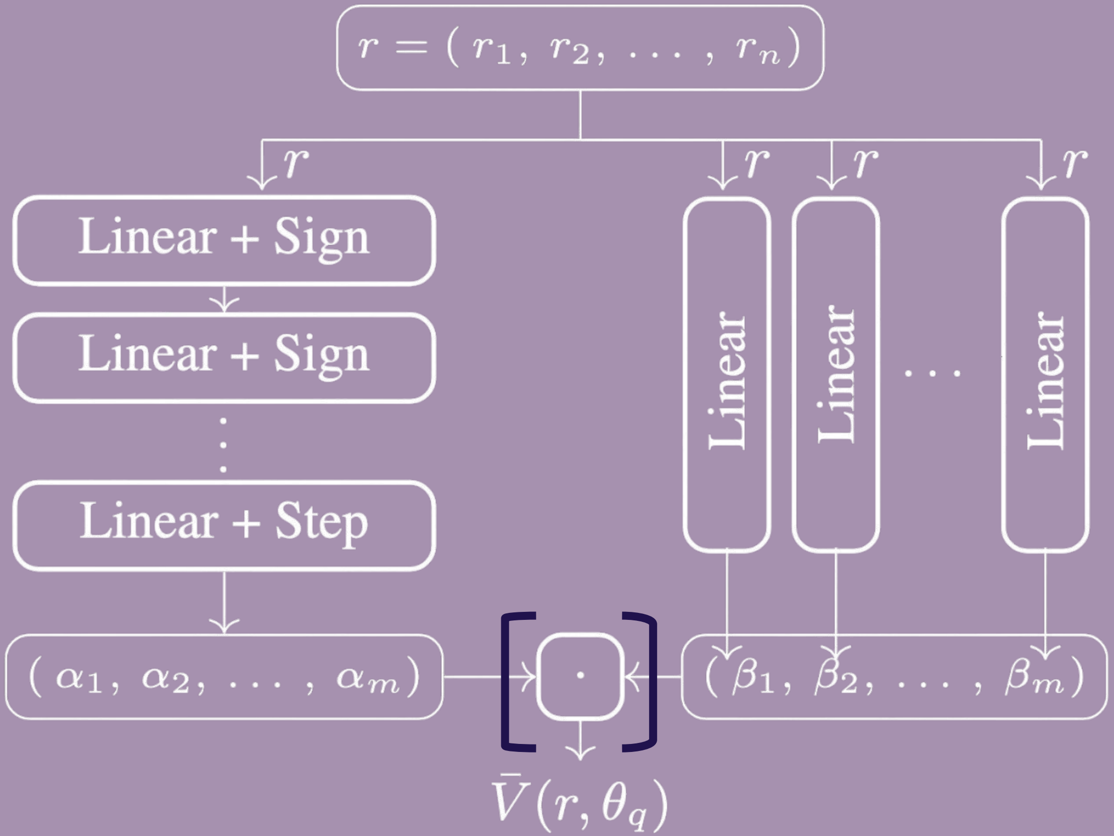


Network Architecture

On the **right**, we have **m linear equations** on input r.



Network Architecture



These linear **equations** on the **right** are **masked** by the **booleans on the left**, and their **dot product** assigns **one affine equation for each of the 2^m region**.

The architecture thus realises piecewise linear function, while allowing solver friendly MILP Encoding.

Thanks!