Compositional Inductive Invariant Based Verification of Neural Network Controlled Systems

Yuhao Zhou and **Stavros Tripakis**
Northeastern University

What is this talk about?

A **compositional** method for verifying the

safety properties of

neural network controlled systems

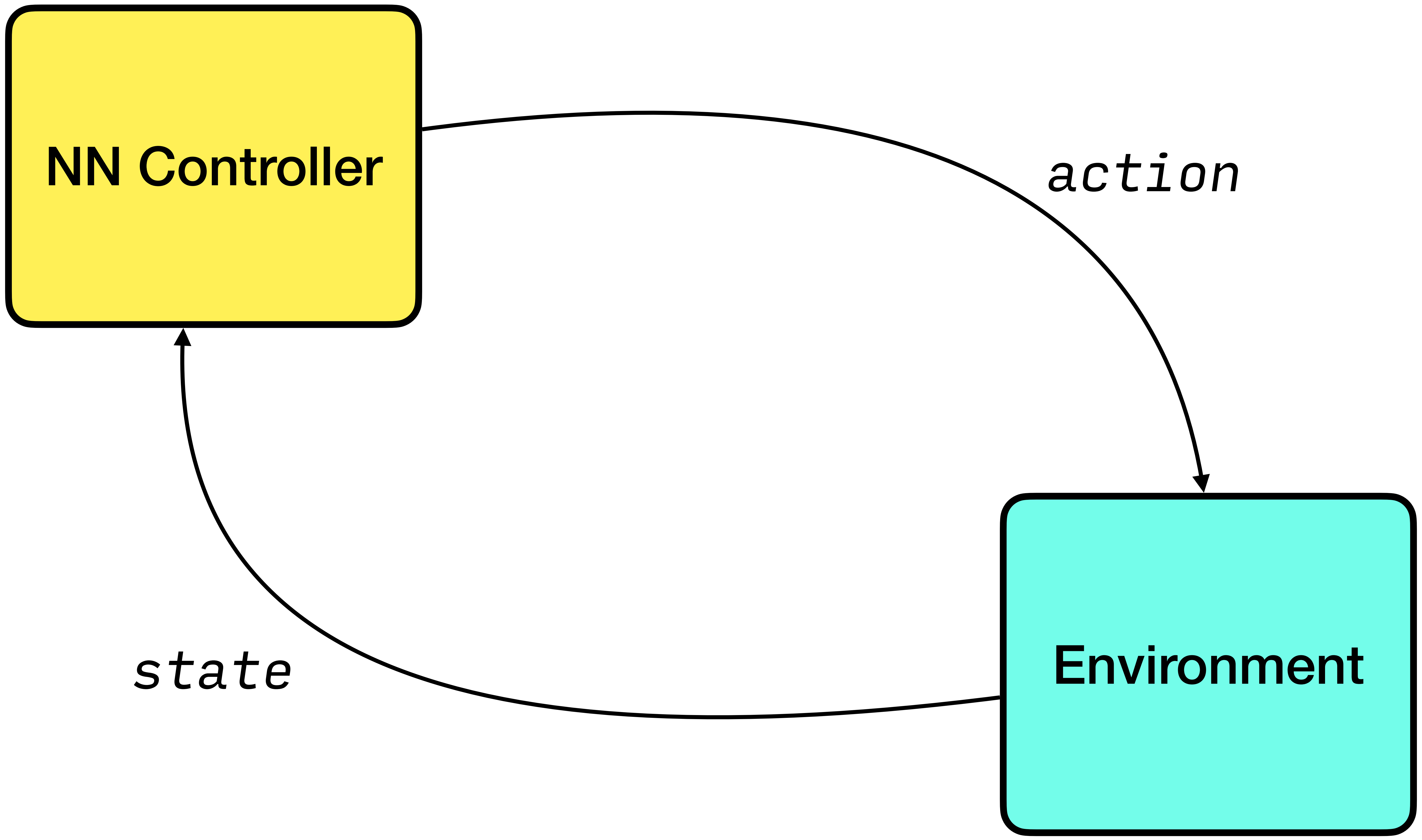
over an **infinite time horizon**

using **inductive invariant method**

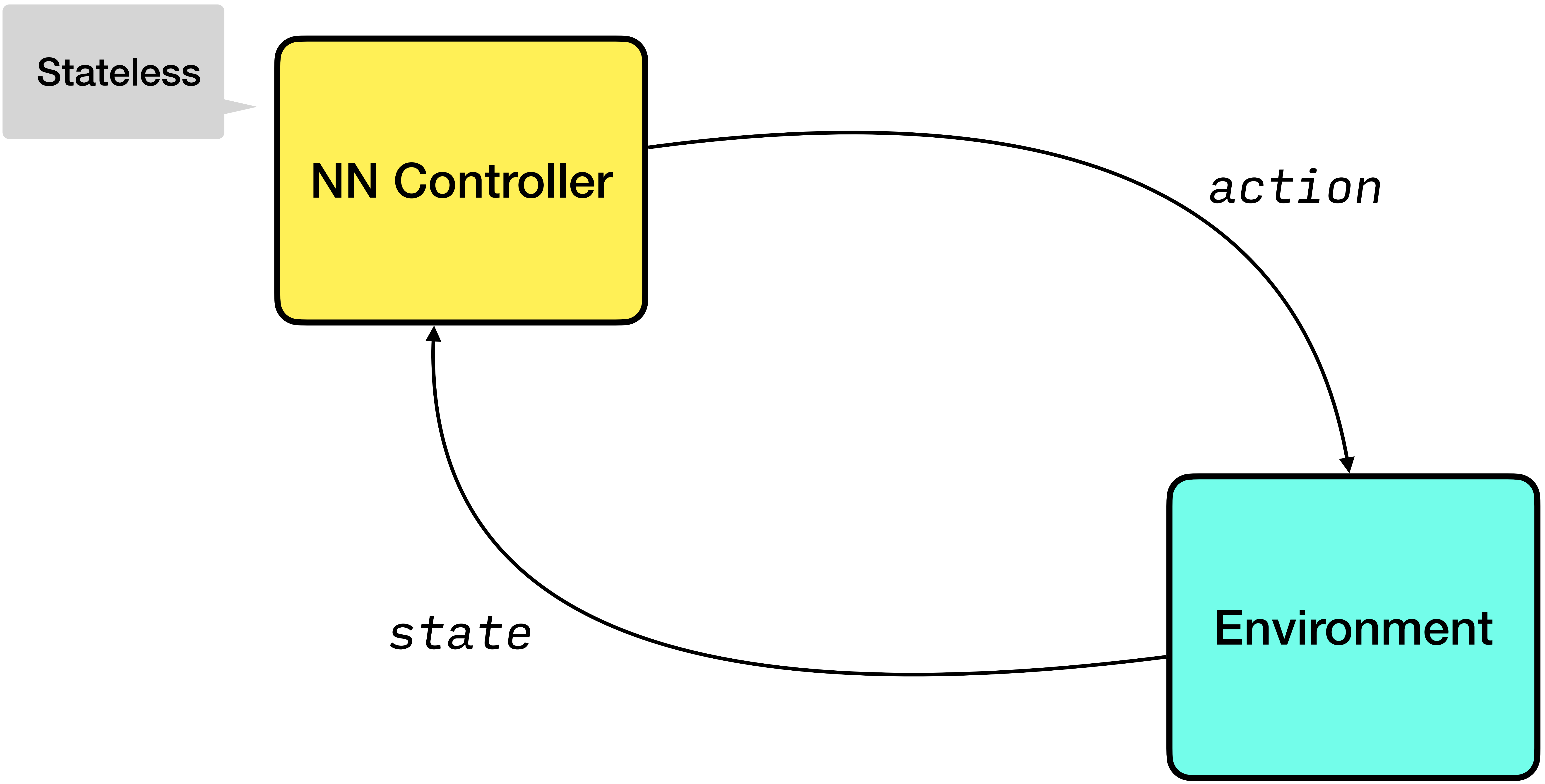
Neural Network Controlled Systems

(NNCS)

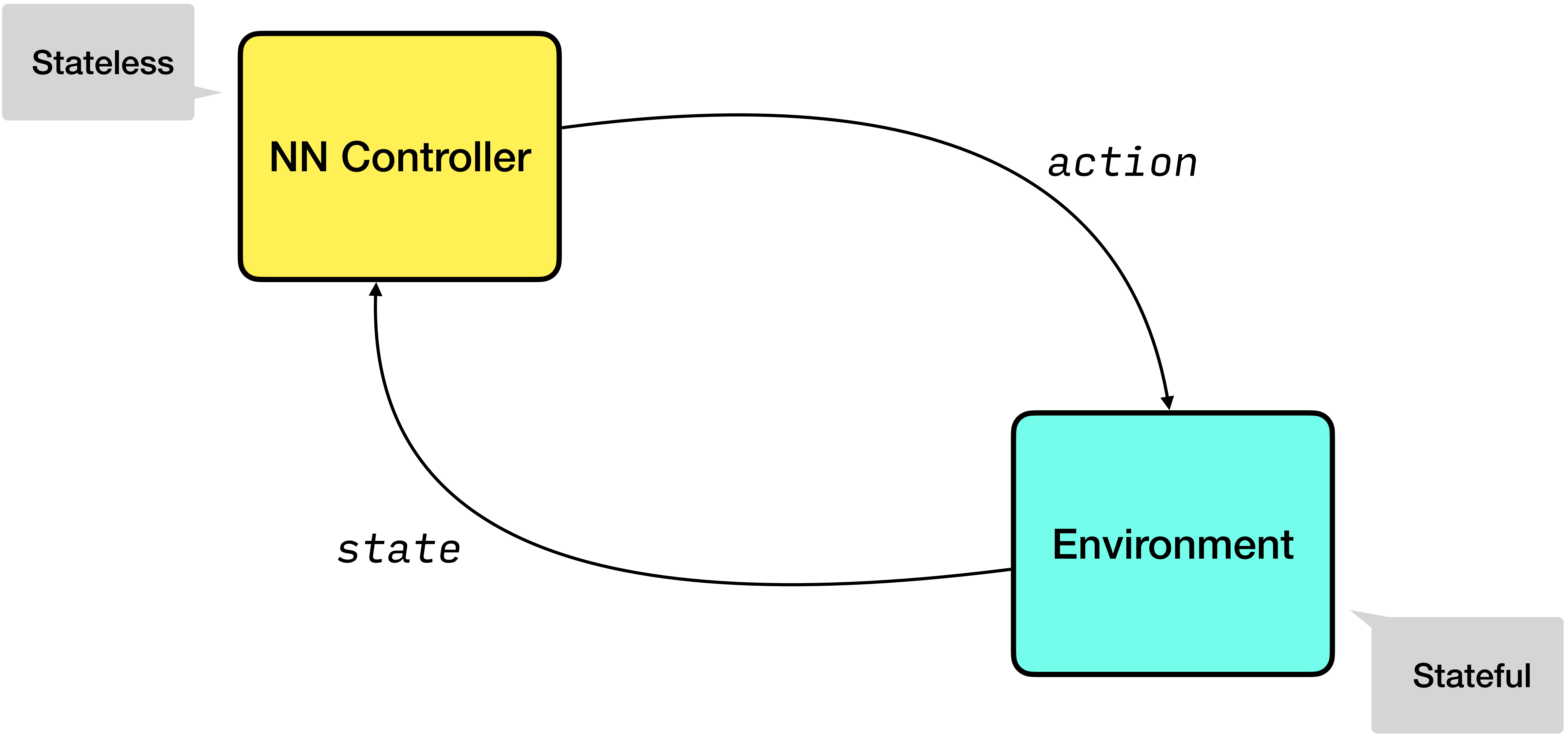
Neural Network Controlled Systems



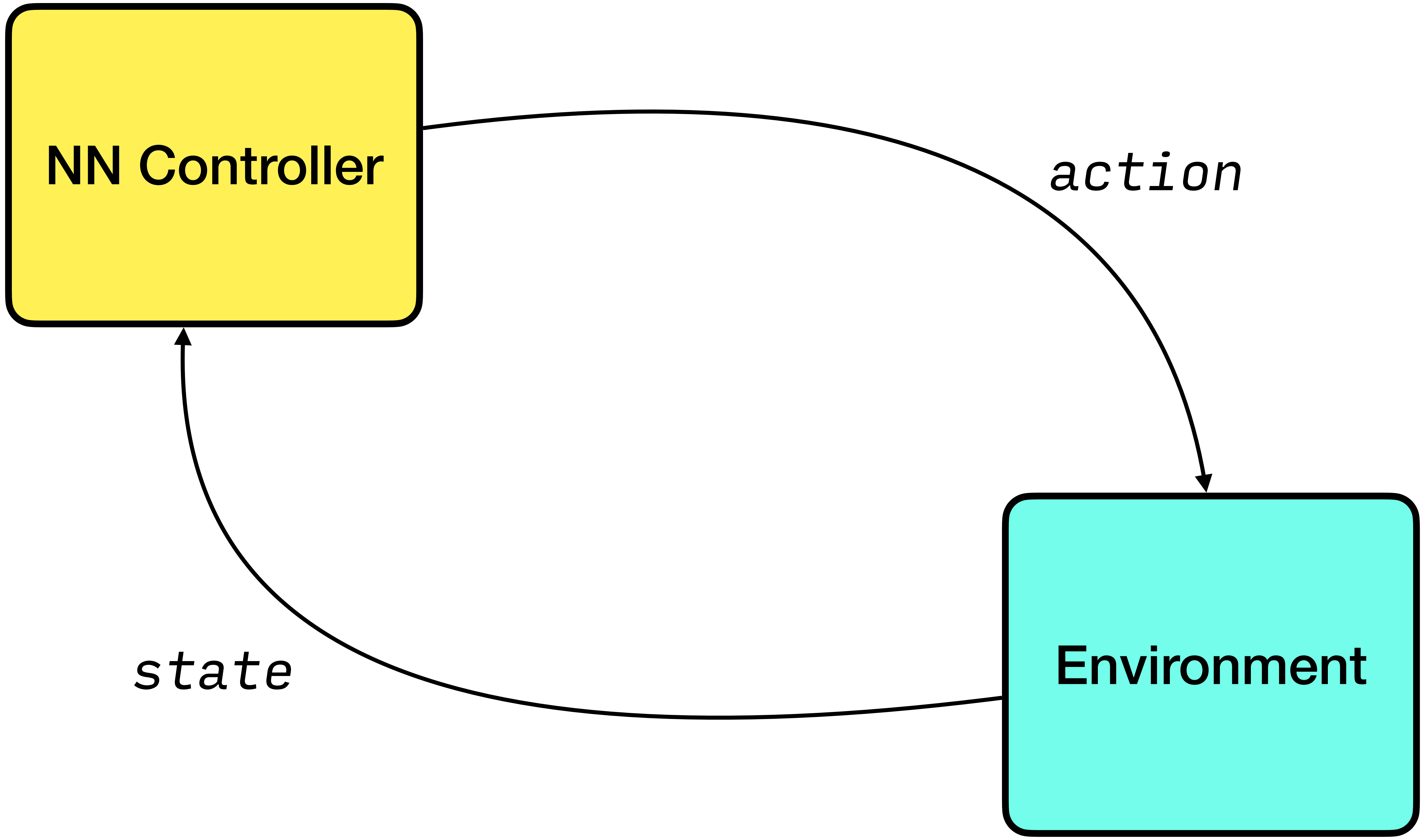
Neural Network Controlled Systems



Neural Network Controlled Systems



NNCS Safety Verification Problem



Check that every reachable `state` satisfies a given state predicate Safe

Why NNCS Verification?

NNCS Used in Safety-Critical Applications

Autonomous Cars

Industrial Control

Healthcare

Our Approach: Key Difference with Prior Work

Majority of Past Work	Bounded-Time Horizon	Reachability Analysis
Ours	Infinite-time Horizon	Inductive Invariant Method

Inductive Invariant Method

Discover a state predicate $IndInv$, such that

(1) $Init \implies IndInv$

(2) $(IndInv \land Next) \implies IndInv'$

(3) $IndInv \implies Safe$

Challenges for Inductive Invariant Method

Discovery of $IndInv$

Check if candidate $IndInv$ is indeed inductive

Challenges for Inductive Invariant Method

Discovery of $IndInv$

Check if candidate $IndInv$ is indeed inductive

This paper

Inductiveness Check in NNCS

Inductiveness

$$(IndInv \wedge Next) \implies IndInv'$$

NNCS

$$Next = Next_{NNC} \wedge Next_{ENV}$$

Inductiveness Check in NNCS

Inductiveness

$$(IndInv \land Next) \implies IndInv'$$

NNCS

$$Next = Next_{NNC} \land Next_{ENV}$$

Inductiveness in NNCS

$$(IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

Inductiveness Check in NNCS

Inductiveness

$$(IndInv \wedge Next) \implies IndInv'$$

NNCS

$$Next = Next_{NNC} \wedge Next_{ENV}$$

Inductiveness in NNCS

Monolithic Method:
Check this formula directly

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

Monolithic Inductiveness Check

Monolithic Inductiveness Check

$$(IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

Monolithic Inductiveness Check

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

SMT Solvers
(e.g. Z3)
Do not scale due to the size of NN controller

Monolithic Inductiveness Check

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

SMT Solvers

(e.g. Z3)

Do not scale due to the size of NN controller

NN Verifier

(e.g. α, β-CROWN)

Limited expressiveness of verifiable specifications

Monolithic Inductiveness Check

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

SMT Solvers

(e.g. Z3)

Do not scale due to the size of NN controller

NN Verifier

(e.g. α, β-CROWN)

Limited expressiveness of verifiable specifications

Monolithic method cannot efficiently check if the candidate $IndInv$ is indeed inductive

Our Contribution

Compositional Method

Replace **monolithic** check

$$(M) \qquad (IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

with 2 **easier** checks

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

Compositional Method

Replace **monolithic** check

$$(M) \qquad (IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

Guaranteed by construction of $Bridge$

with 2 **easier** checks

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

Compositional Method

Replace **monolithic** check

$$(M) \qquad (IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

Guaranteed by construction of $Bridge$

with 2 **easier** checks

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

SMT Solvers can handle this (but we help with parallelization)

Compositional Method

Soundness and Completeness

$$
\begin{array}{ll}
(M) & (IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv' \\
(C1) & (IndInv \land Next_{NNC}) \implies Bridge \\
(C2) & (Bridge \land Next_{ENV}) \implies IndInv'
\end{array}
$$

Compositional Method

Soundness and Completeness

$$(M) \qquad (IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

$$(C1) \qquad (IndInv \wedge Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \wedge Next_{ENV}) \implies IndInv'$$

Theorem: Soundness

If $(C1)$ and $(C2)$ hold, then (M) holds

Compositional Method

Soundness and Completeness

$$(M) \qquad (IndInv \land Next_{NNC} \land Next_{ENV}) \implies IndInv'$$

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

Theorem: Soundness

If $(C1)$ and $(C2)$ hold, then (M) holds

Theorem: Completeness

If (M) holds, then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

Compositional Method

Questions

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

Compositional Method

Questions

$$(C1) \qquad (IndInv \wedge Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \wedge Next_{ENV}) \implies IndInv'$$

How to **discover** $Bridge$

Compositional Method

Questions

$$(C1) \qquad (IndInv \land Next_{NNC}) \implies Bridge$$
$$(C2) \qquad (Bridge \land Next_{ENV}) \implies IndInv'$$

How to **discover** $Bridge$

How to **check** $(C1)$ and $(C2)$ in a scalable way

Compositional Method

Questions

$$(C1) \qquad (IndInv \wedge Next_{NNC}) \implies Bridge$$

$$(C2) \qquad (Bridge \wedge Next_{ENV}) \implies IndInv'$$

How to **discover** $Bridge$

Automatic method

How to **check** $(C1)$ and $(C2)$ in a scalable way

Automatic $Bridge$ Construction

Decompose $IndInv$

$$IndInv \iff (p_1 \lor p_2 \lor \cdots \lor p_n)$$

Automatic *Bridge* Construction

Decompose $IndInv$

$$IndInv \iff (p_1 \vee p_2 \vee \cdots \vee p_n)$$

For each state predicate p_i, compute postcondition ψ_i, such that

$$(p_i \wedge Next_{NNC}) \implies \psi_i$$

Automatic *Bridge* Construction

Decompose $\boldsymbol{IndInv}$

$$IndInv \iff (p_1 \vee p_2 \vee \cdots \vee p_n)$$

For each state predicate $\boldsymbol{p_i}$, compute postcondition $\boldsymbol{\psi_i}$, such that

NN controller input constraint

$$(p_i \wedge Next_{NNC}) \implies \psi_i$$

Automatic *Bridge* Construction

Decompose *IndInv*

$$IndInv \iff (p_1 \vee p_2 \vee \cdots \vee p_n)$$

For each state predicate p_i, compute postcondition ψ_i, such that

NN controller input constraint

$$(p_i \wedge Next_{NNC}) \implies \psi_i$$

Output range verified by NN verifier

Automatic *Bridge* Construction

Decompose $\textbf{\textit{IndInv}}$

$$IndInv \iff (p_1 \vee p_2 \vee \cdots \vee p_n)$$

For each state predicate $\boldsymbol{p_i}$, compute postcondition $\boldsymbol{\psi_i}$, such that

NN controller input constraint

$$(p_i \wedge Next_{NNC}) \implies \psi_i$$

Output range verified by NN verifier

$$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$$

Condition $(C1)$ Holds by Construction!

$$(IndInv \land Next_{NNC}) \implies Bridge$$

Condition $(C1)$ Holds by Construction!

$$(IndInv \land Next_{NNC}) \implies Bridge$$

Replaced by $IndInv$ decomposition

$$IndInv \iff (p_1 \lor p_2 \lor \cdots \lor p_n)$$

Condition $(C1)$ Holds by Construction!

$$(IndInv \land Next_{NNC}) \implies Bridge$$

Replaced by $IndInv$ decomposition
$IndInv \iff (p_1 \lor p_2 \lor \cdots \lor p_n)$

Replaced by $Bridge$ definition
$Bridge := (p_1 \land \psi_1) \lor \cdots \lor (p_n \land \psi_n)$

Condition $(C1)$ Holds by Construction!

$$(IndInv \land Next_{NNC}) \implies Bridge$$

Replaced by $IndInv$ decomposition
$IndInv \iff (p_1 \lor p_2 \lor \cdots \lor p_n)$

Replaced by $Bridge$ definition
$Bridge := (p_1 \land \psi_1) \lor \cdots \lor (p_n \land \psi_n)$

Condition $(C1)$ becomes

$$\bigvee_i (p_i \land Next_{NNC}) \implies \bigvee_i (p_i \land \psi_i)$$

Condition $(C1)$ Holds by Construction!

$$\bigvee_i (p_i \wedge Next_{NNC}) \implies \bigvee_i (p_i \wedge \psi_i)$$

Condition $(C1)$ Holds by Construction!

$$\bigvee_i (p_i \wedge Next_{NNC}) \implies \bigvee_i (p_i \wedge \psi_i)$$

Since $p_i \wedge Next_{NNC} \implies p_i$, we can simplify the formula to

$$\bigvee_i (p_i \wedge Next_{NNC}) \implies \bigvee_i \psi_i$$

Condition $(C1)$ Holds by Construction!

$$\bigvee_i (p_i \land Next_{NNC}) \implies \bigvee_i (p_i \land \psi_i)$$

Since $p_i \land Next_{NNC} \implies p_i$, we can simplify the formula to

$$\bigvee_i (p_i \land Next_{NNC}) \implies \bigvee_i \psi_i$$

This holds because the NN verifier guarantees

$$(p_i \land Next_{NNC}) \implies \psi_i$$

How to Check Condition $(C2)$ Efficiently?

$$(Bridge \land Next_{ENV}) \implies IndInv'$$

How to Check Condition $(C2)$ Efficiently?

$$(Bridge \land Next_{ENV}) \implies IndInv'$$

Replaced by the definition

$Bridge := (p_1 \land \psi_1) \lor \cdots \lor (p_n \land \psi_n)$

How to Check Condition $(C2)$ Efficiently?

$$(Bridge \land Next_{ENV}) \implies IndInv'$$

Replaced by the definition
$Bridge := (p_1 \land \psi_1) \lor \cdots \lor (p_n \land \psi_n)$

Condition $(C2)$ becomes

$$(\bigvee_i (p_i \land \psi_i \land Next_{ENV})) \implies IndInv'$$

How to Check Condition $(C2)$ Efficiently?

$$(Bridge \wedge Next_{ENV}) \implies IndInv'$$

Replaced by the definition
$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$

Condition $(C2)$ becomes

$$(\bigvee_i (p_i \wedge \psi_i \wedge Next_{ENV})) \implies IndInv'$$

which is equivalent to

$$\bigwedge_i ((p_i \wedge \psi_i \wedge Next_{ENV}) \implies IndInv')$$

How to Check Condition $(C2)$ Efficiently?

$$(Bridge \land Next_{ENV}) \implies IndInv'$$

Replaced by the definition
$Bridge := (p_1 \land \psi_1) \lor \cdots \lor (p_n \land \psi_n)$

Condition $(C2)$ becomes

$$(\bigvee_i (p_i \land \psi_i \land Next_{ENV})) \implies IndInv'$$

which is equivalent to

$$\bigwedge_i ((p_i \land \psi_i \land Next_{ENV}) \implies IndInv')$$

This conjunction of smaller formulas can be checked in parallel!

Algorithm

Algorithm

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$

Algorithm

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$

Construct
$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$
by leveraging NN verifiers

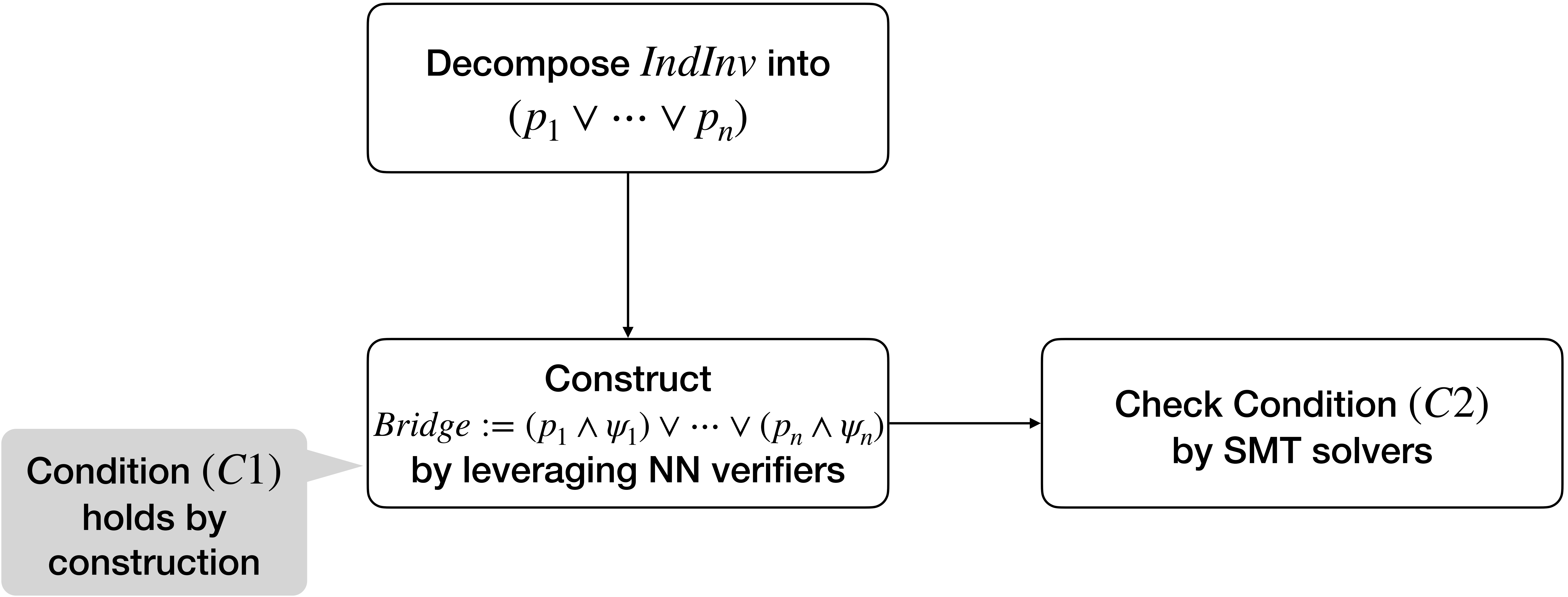
Algorithm

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$

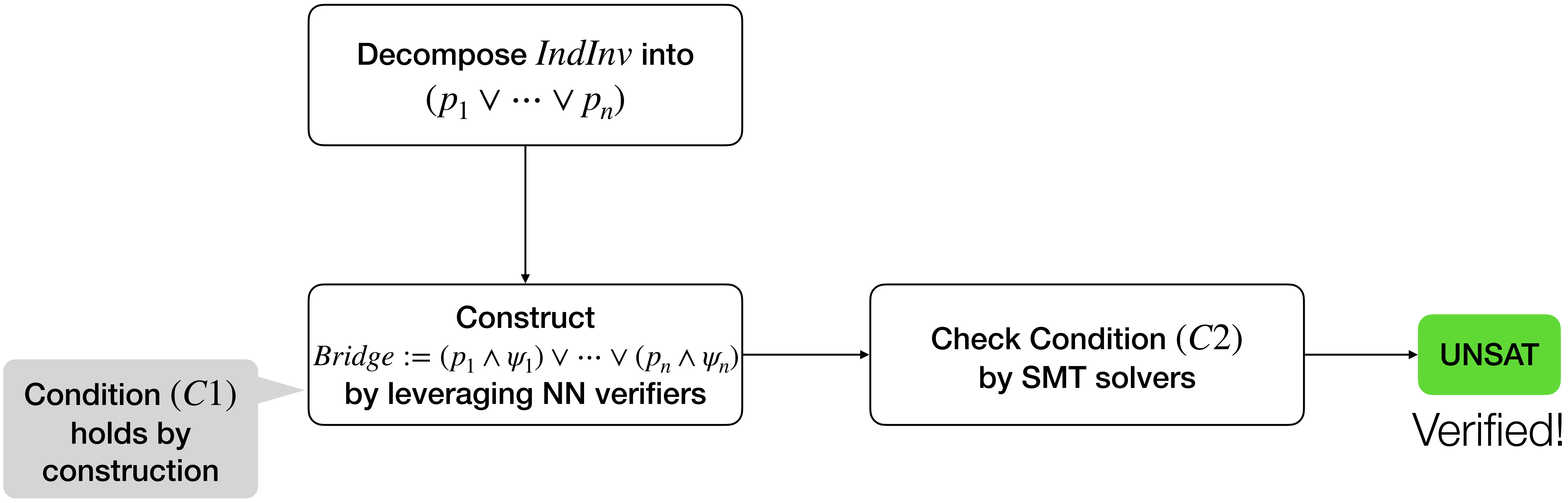
Construct
$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$
by leveraging NN verifiers

Condition $(C1)$ holds by construction

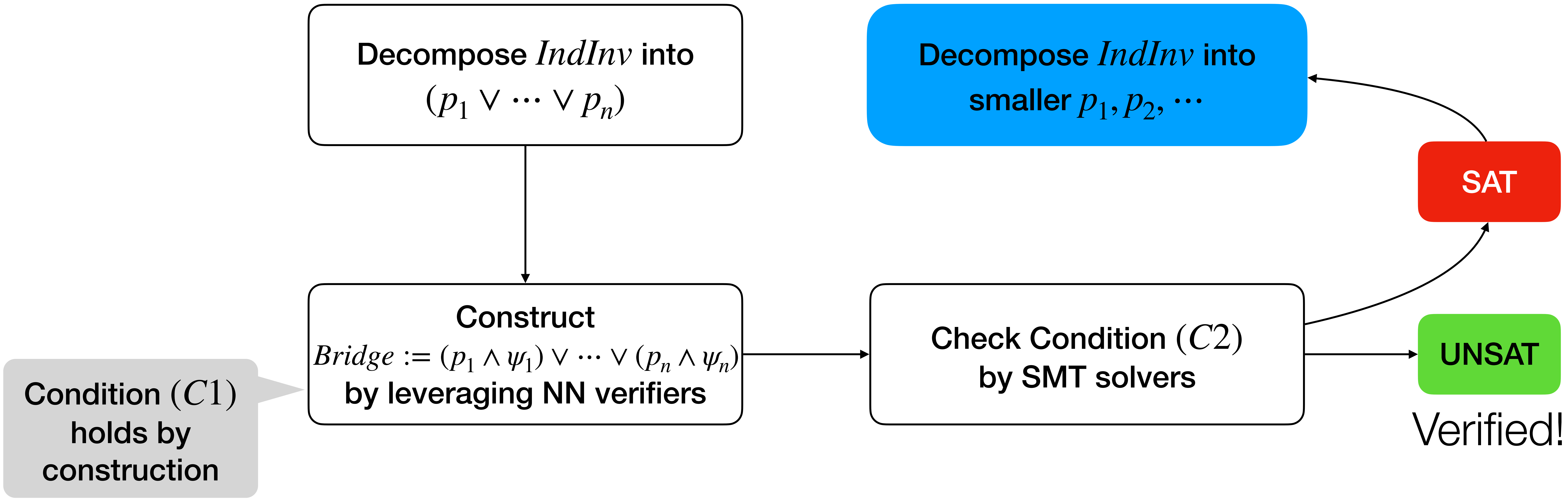
Algorithm


Decompose $IndInv$ into $(p_1 \vee \cdots \vee p_n)$
Construct $Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$ by leveraging NN verifiers
Check Condition $(C2)$ by SMT solvers
Condition $(C1)$ holds by construction

Algorithm

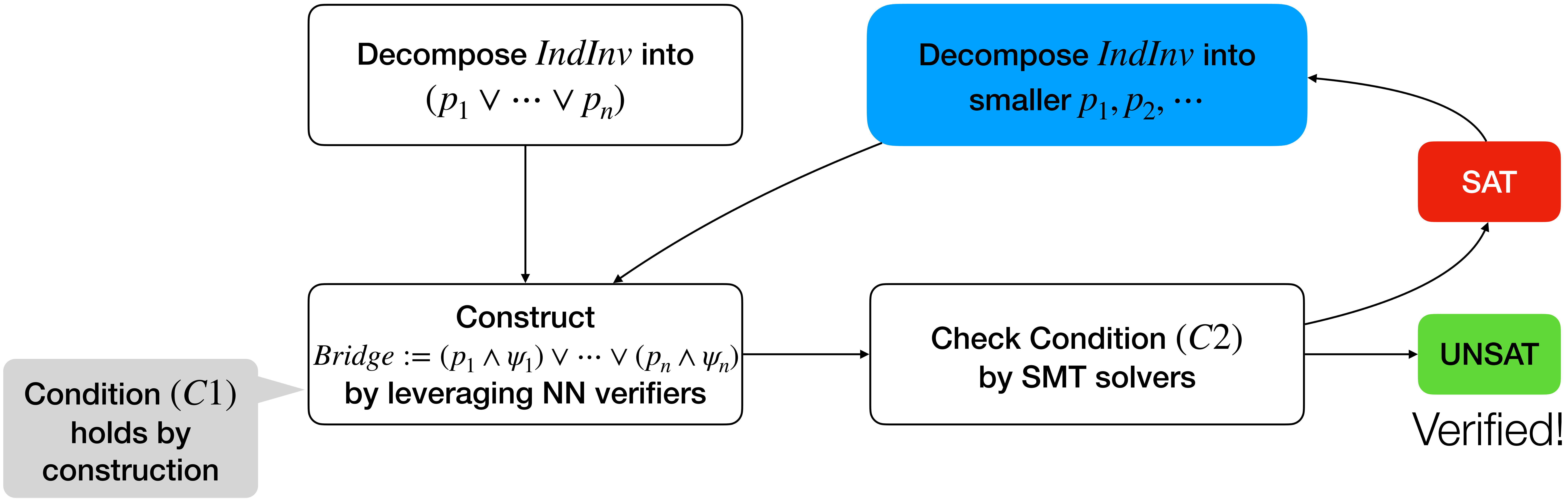

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$
Construct
$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$
by leveraging NN verifiers
Condition $(C1)$ holds by construction
Check Condition $(C2)$
by SMT solvers
UNSAT
Verified!

Algorithm



Decompose $IndInv$ into $(p_1 \vee \cdots \vee p_n)$
Decompose $IndInv$ into smaller $p_1, p_2, \cdots$
SAT
Construct $Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$ by leveraging NN verifiers
Check Condition $(C2)$ by SMT solvers
UNSAT
Condition $(C1)$ holds by construction
Verified!

Algorithm

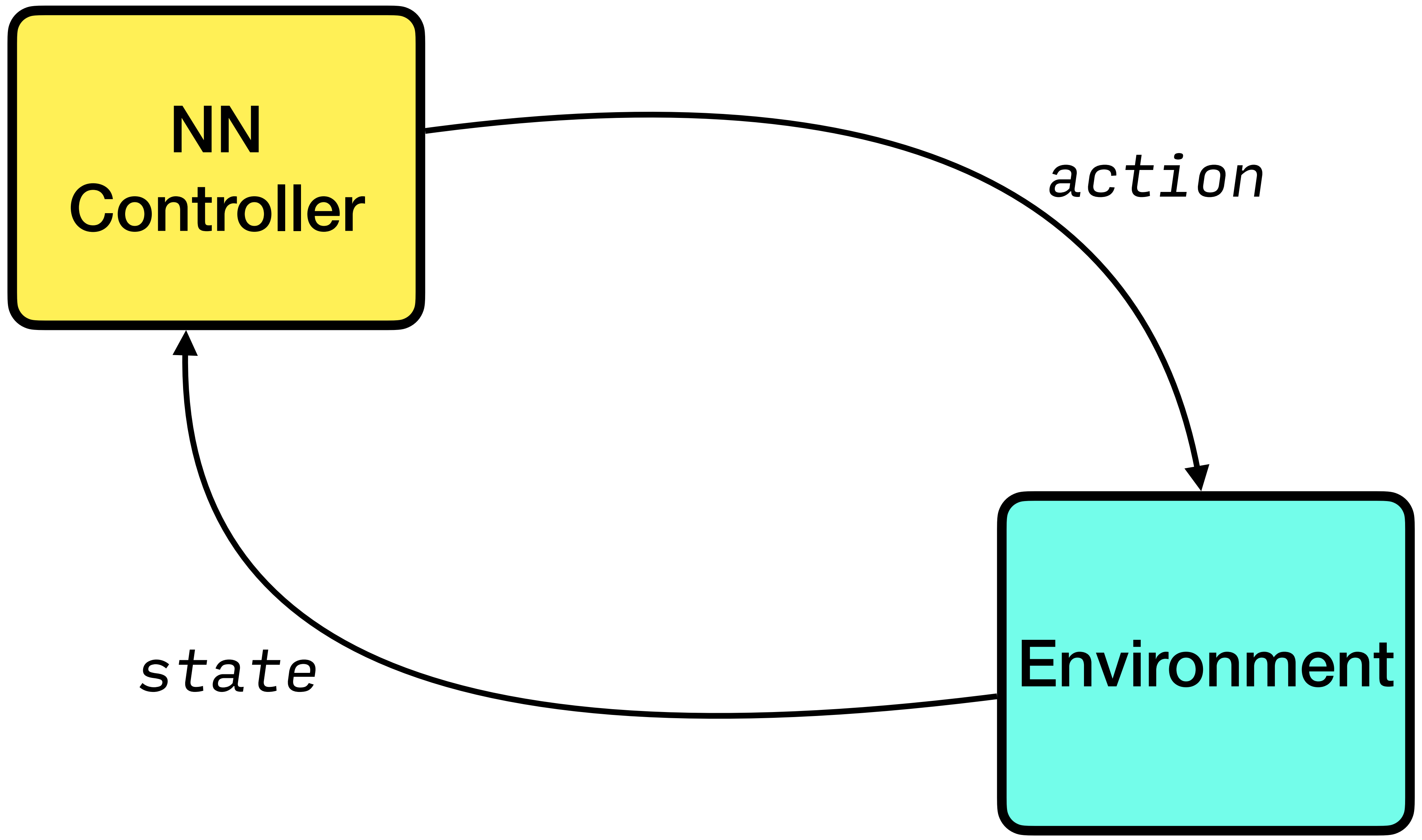


Decompose $IndInv$ into $(p_1 \vee \cdots \vee p_n)$
Decompose $IndInv$ into smaller $p_1, p_2, \cdots$
Construct $Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$ by leveraging NN verifiers
Check Condition $(C2)$ by SMT solvers
SAT
UNSAT
Verified!
Condition $(C1)$ holds by construction

Evaluation

Evaluation

2D Navigation Case Study



State variables

$$x, y \in \mathbb{R}$$

NN controller

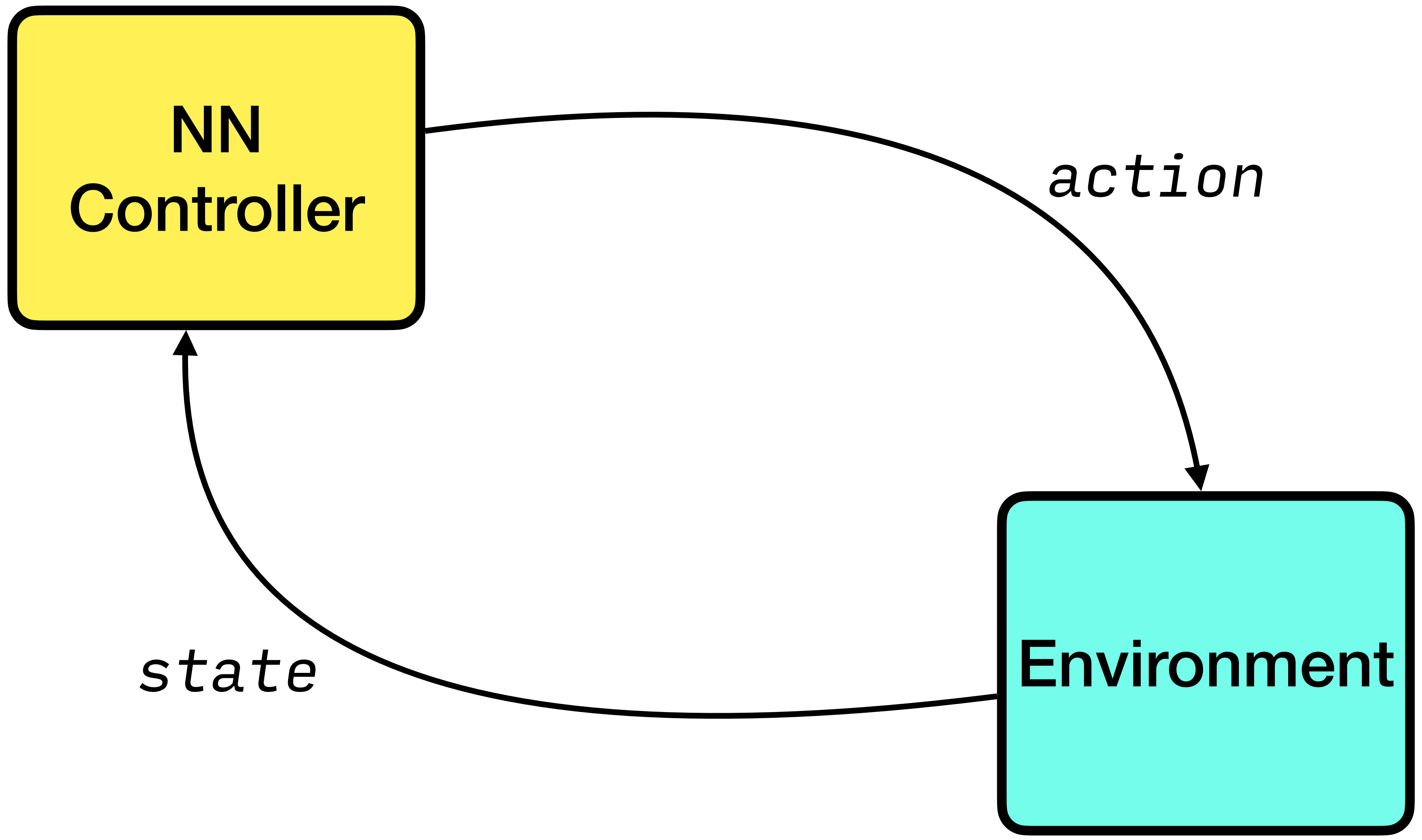
$$(a, b) = NN(x, y)$$

Environment

$$0.5 \leq c \leq 1.0$$
$$x' = x + 0.1 \cdot c \cdot a,$$
$$y' = y + 0.1 \cdot c \cdot b$$

Evaluation

2D Navigation Case Study



State variables

$$x, y \in \mathbb{R}$$

Represent a 2D plane

NN controller

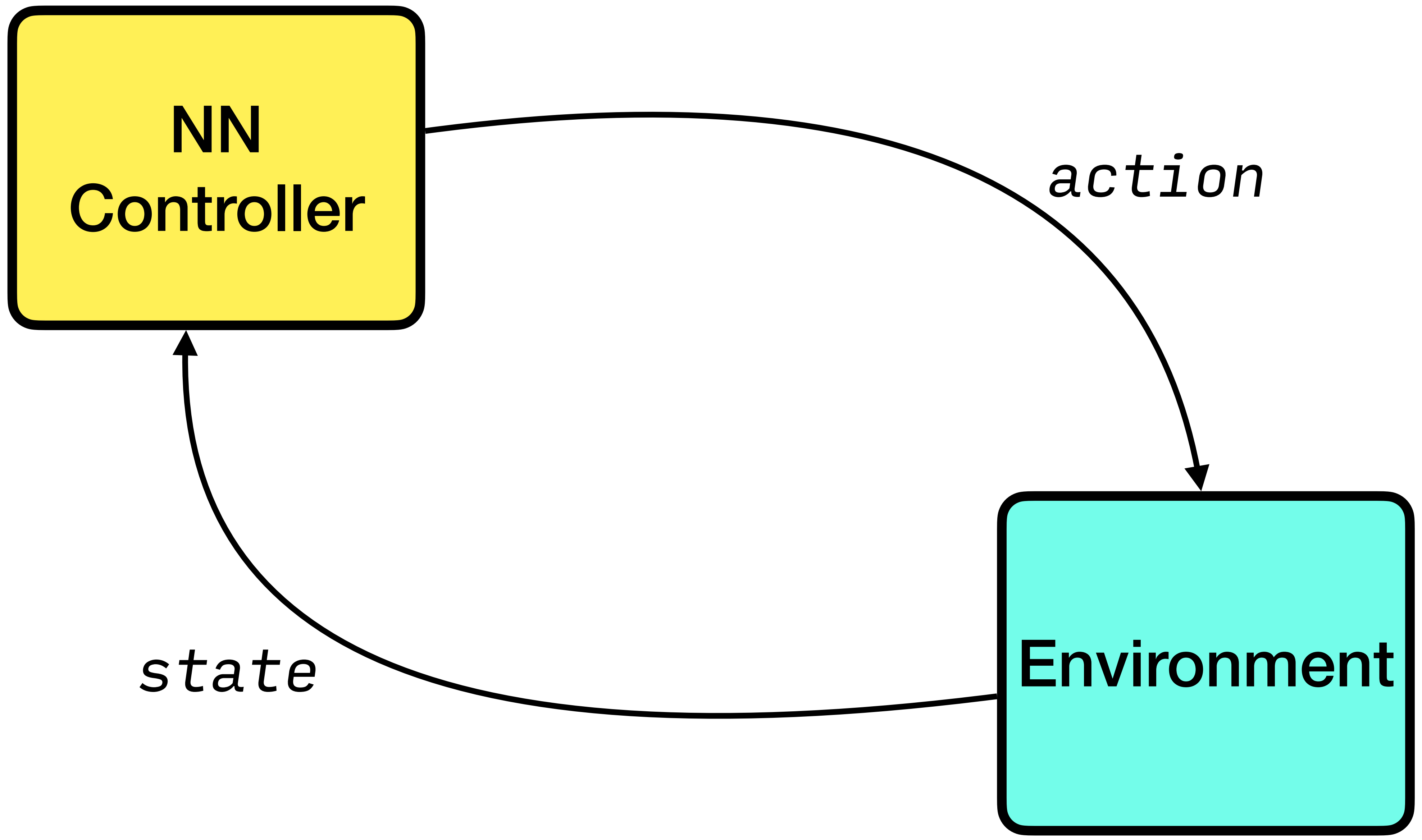
$$(a, b) = NN(x, y)$$

Environment

$$0.5 \leq c \leq 1.0$$
$$x' = x + 0.1 \cdot c \cdot a,$$
$$y' = y + 0.1 \cdot c \cdot b$$

Evaluation

2D Navigation Case Study



State variables

$$x, y \in \mathbb{R}$$

Represent a 2D plane

NN controller

$$(a, b) = NN(x, y)$$

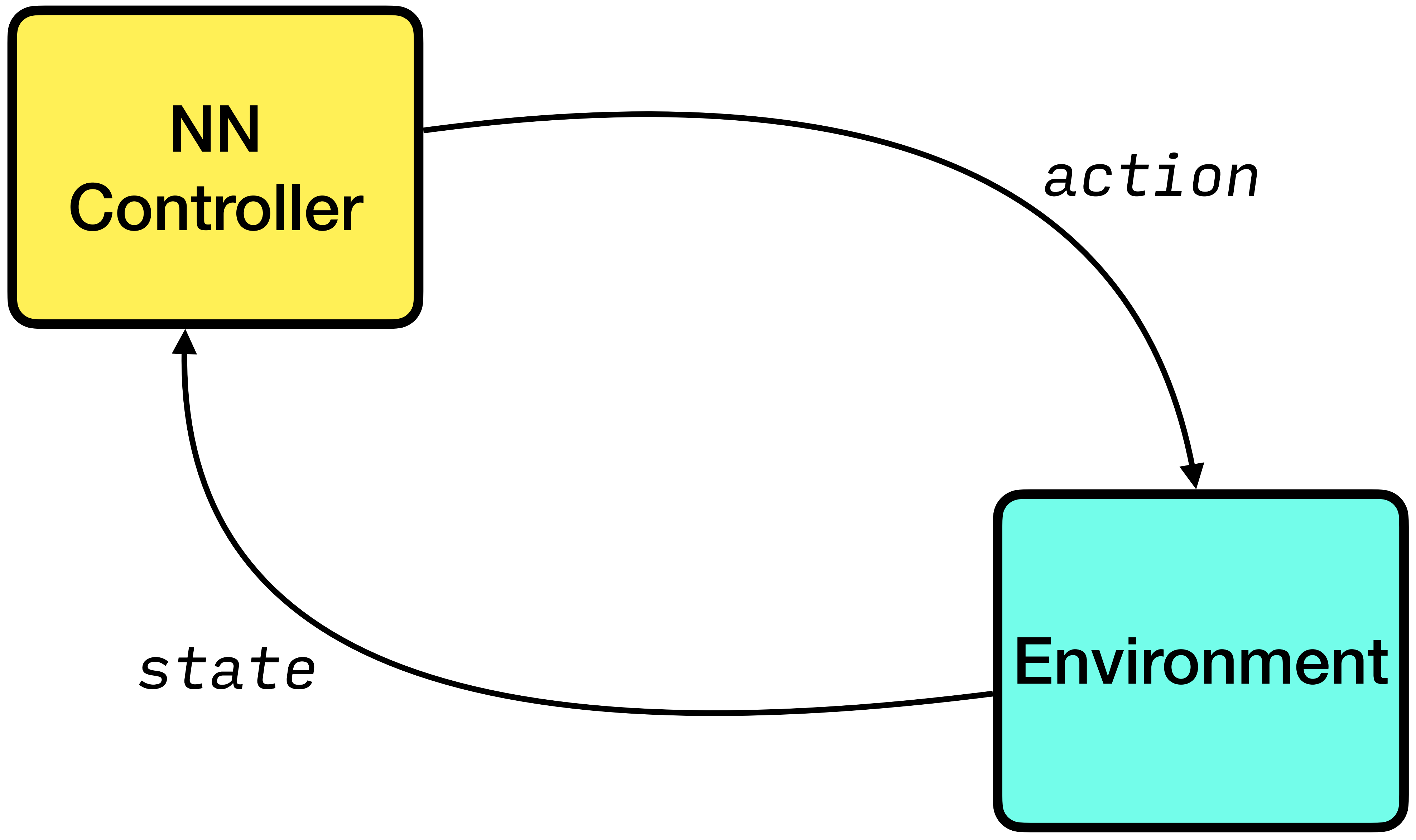
NNC takes states, outputs 2 reals a, b

Environment

$$0.5 \leq c \leq 1.0$$
$$x' = x + 0.1 \cdot c \cdot a,$$
$$y' = y + 0.1 \cdot c \cdot b$$

Evaluation

2D Navigation Case Study



State variables

$$x, y \in \mathbb{R}$$

Represent a 2D plane

NN controller

$$(a, b) = NN(x, y)$$

NNC takes states, outputs 2 reals a, b

Environment

$$0.5 \leq c \leq 1.0$$
$$x' = x + 0.1 \cdot c \cdot a,$$
$$y' = y + 0.1 \cdot c \cdot b$$

Non-deterministic, discrete-time

Evaluation

2D Navigation Case Study

Evaluation

2D Navigation Case Study

- Trained 9 NN controllers, from 2×32 neurons to 2×1024 neurons

Evaluation

2D Navigation Case Study

- Trained 9 NN controllers, from 2×32 neurons to 2×1024 neurons
- Use same environment with different NN controllers

Evaluation

2D Navigation Case Study

- Trained 9 NN controllers, from 2×32 neurons to 2×1024 neurons
- Use same environment with different NN controllers
- Compare monolithic vs compositional method

Evaluation

2D Navigation Case Study

- Trained 9 NN controllers, from 2×32 neurons to 2×1024 neurons
- Use same environment with different NN controllers
- Compare monolithic vs compositional method
- Z3 for the monolithic method

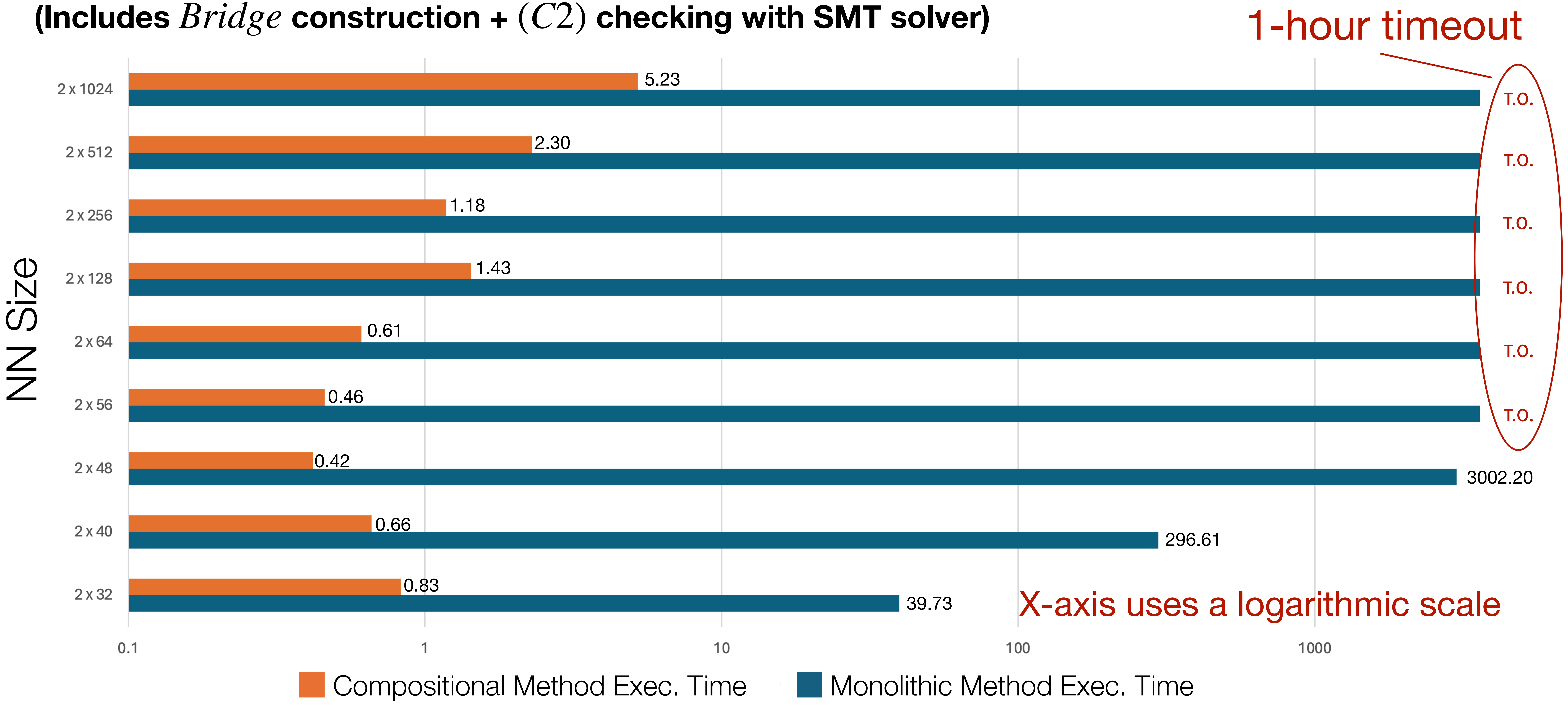
Evaluation

2D Navigation Case Study

- Trained 9 NN controllers, from 2×32 neurons to 2×1024 neurons
- Use same environment with different NN controllers
- Compare monolithic vs compositional method
- Z3 for the monolithic method
- Z3 and AutoLIRPA for the compositional method

Execution Times (seconds)

(Includes $Bridge$ construction + $(C2)$ checking with SMT solver)



Summary

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon
- Novel compositional method to overcome the scalability issues of the monolithic method

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon
- Novel compositional method to overcome the scalability issues of the monolithic method
- Results: reduce the execution time from hours to seconds

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon
- Novel compositional method to overcome the scalability issues of the monolithic method
- Results: reduce the execution time from hours to seconds
- Artifact available online: https://github.com/YUH-Z/comp-indinv-verification-nncs

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon
- Novel compositional method to overcome the scalability issues of the monolithic method
- Results: reduce the execution time from hours to seconds
- Artifact available online: https://github.com/YUH-Z/comp-indinv-verification-nncs
- Future work: automated discovery of $IndInv$

Summary

- Use inductive invariant method to verify NNCS over infinite time horizon
- Novel compositional method to overcome the scalability issues of the monolithic method
- Results: reduce the execution time from hours to seconds
- Artifact available online: https://github.com/YUH-Z/comp-indinv-verification-nncs
- Future work: automated discovery of $IndInv$

Thank you!

Thank you!

Evaluation

Details

Evaluation

Details

- 3.3GHz 8-core AMD CPU, 16 GB memory, RTX3060 GPU

Evaluation

Details

- 3.3GHz 8-core AMD CPU, 16 GB memory, RTX3060 GPU
- Z3 on CPU

Evaluation

Details

- 3.3GHz 8-core AMD CPU, 16 GB memory, RTX3060 GPU
- Z3 on CPU
- AutoLIRPA on GPU with CUDA enabled

Evaluation

2D Maze Case Study

NN size	Verified?		Monolithic execution time		Compositional execution time		#Splits		#SMT queries		#NNV queries	
	Det	NDet	Det	NDet	Det	NDet	Det	NDet	Det	NDet	Det	NDet
2×32	T	T	51.59	39.73	0.73	0.83	12	14	61	71	49	57
2×40	T	T	113.69	296.61	0.70	0.66	13	13	66	66	53	53
2×48	T	T	410.14	3002.20	0.52	0.42	10	8	51	41	41	33
2×56	T	T	1203.76	T.O.	0.42	0.46	8	9	41	46	33	37
2×64	T	T	T.O.	T.O.	0.76	0.61	15	12	76	61	61	49
2×128	F	T	T.O.	T.O.	2.21	1.43	64	28	225	141	160	113
2×256	T	T	T.O.	T.O.	1.68	1.18	27	23	136	116	109	93
2×512	T	T	T.O.	T.O.	3.04	2.30	60	45	301	226	241	181
2×1024	T	T	T.O.	T.O.	1.94	5.23	38	102	191	511	153	409

Falsification

Falsification

Theorem: Completeness

If (M) holds, then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

Falsification

Theorem: Completeness

If (M) holds, then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

Falsification

Theorem: Completeness

$\boxed{\text{If } (M) \text{ holds,}}$ then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

If (M) does not hold, we want to avoid hopeless search for $Bridge$

Falsification

Theorem: Completeness

If (M) holds, then there exists Bridge such that $(C1)$ and $(C2)$ hold

If (M) does not hold, we want to avoid hopeless search for Bridge

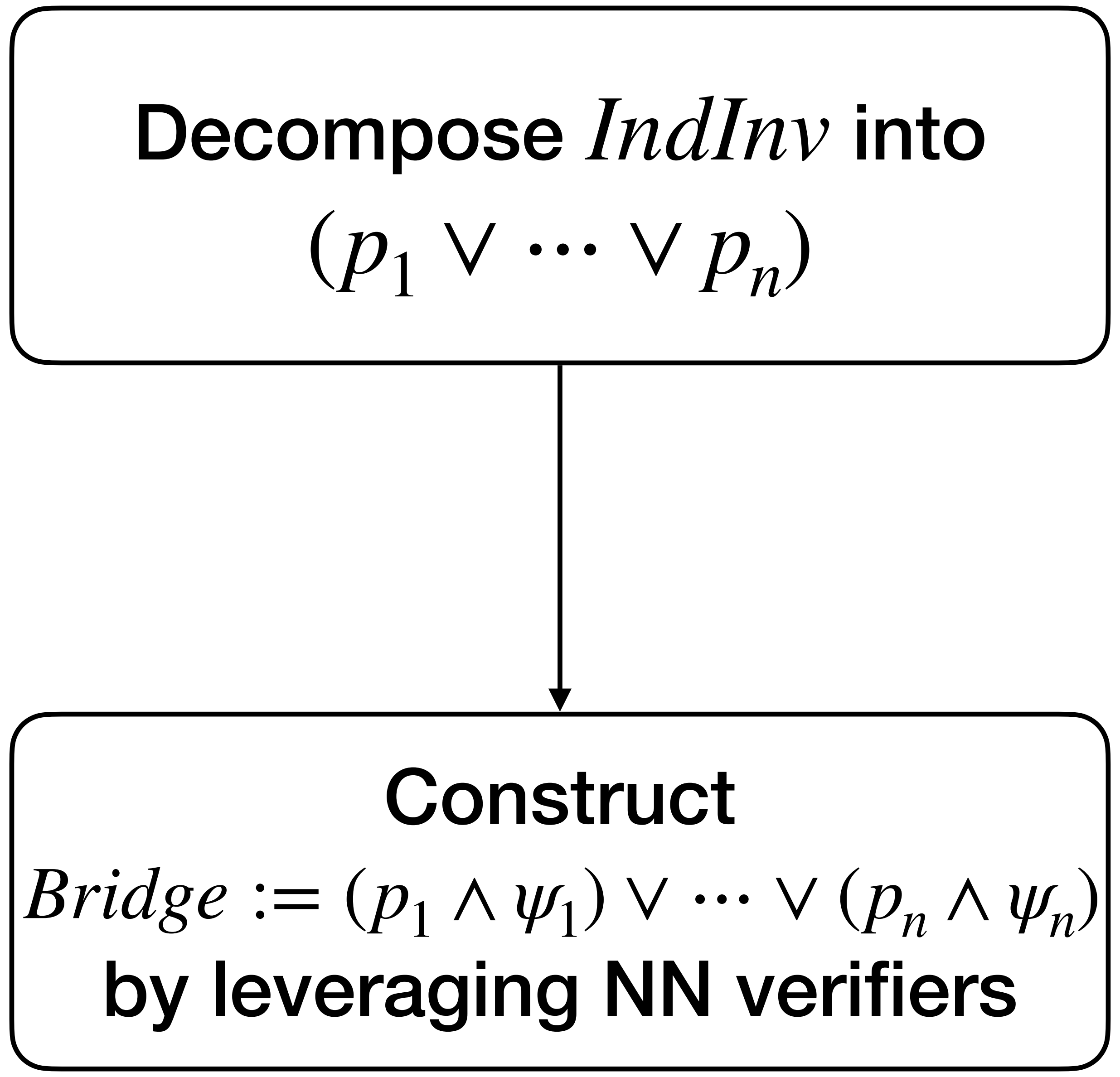
A practical **falsification** heuristic inherits the decomposition ideas

Algorithm with Falsification

Algorithm with Falsification

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$

Algorithm with Falsification



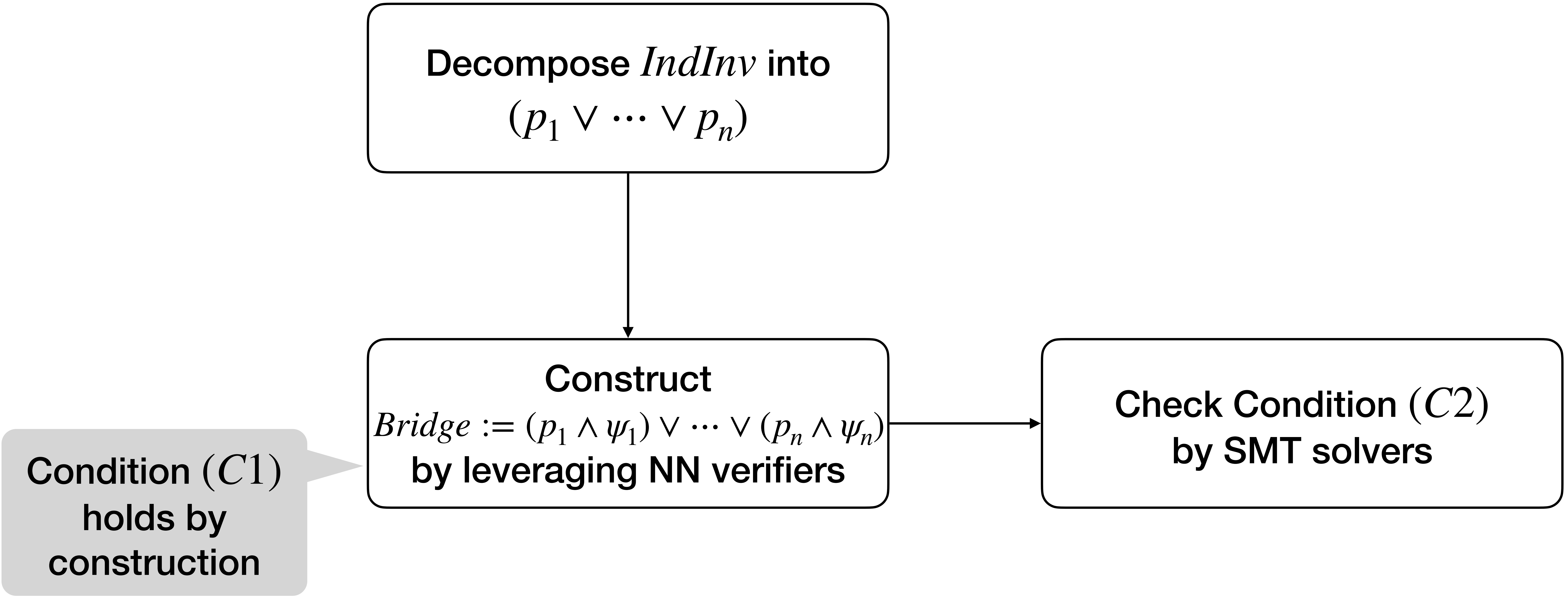
Algorithm with Falsification

Decompose $IndInv$ into
$(p_1 \vee \cdots \vee p_n)$

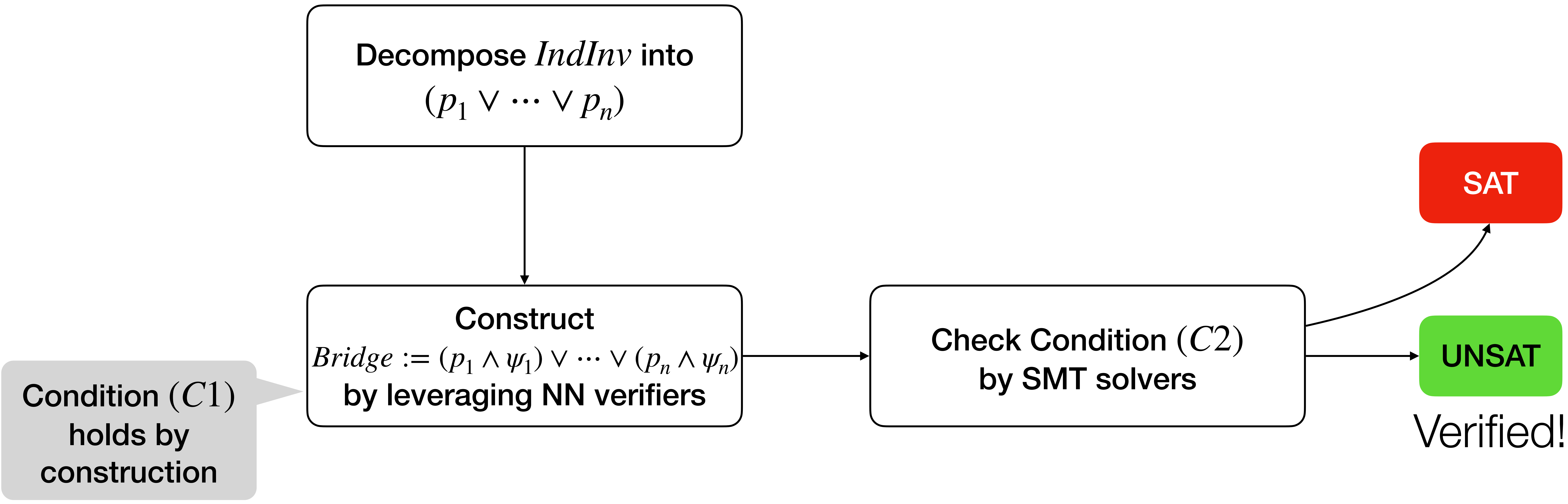
Construct
$Bridge := (p_1 \wedge \psi_1) \vee \cdots \vee (p_n \wedge \psi_n)$
by leveraging NN verifiers

Condition $(C1)$
holds by
construction

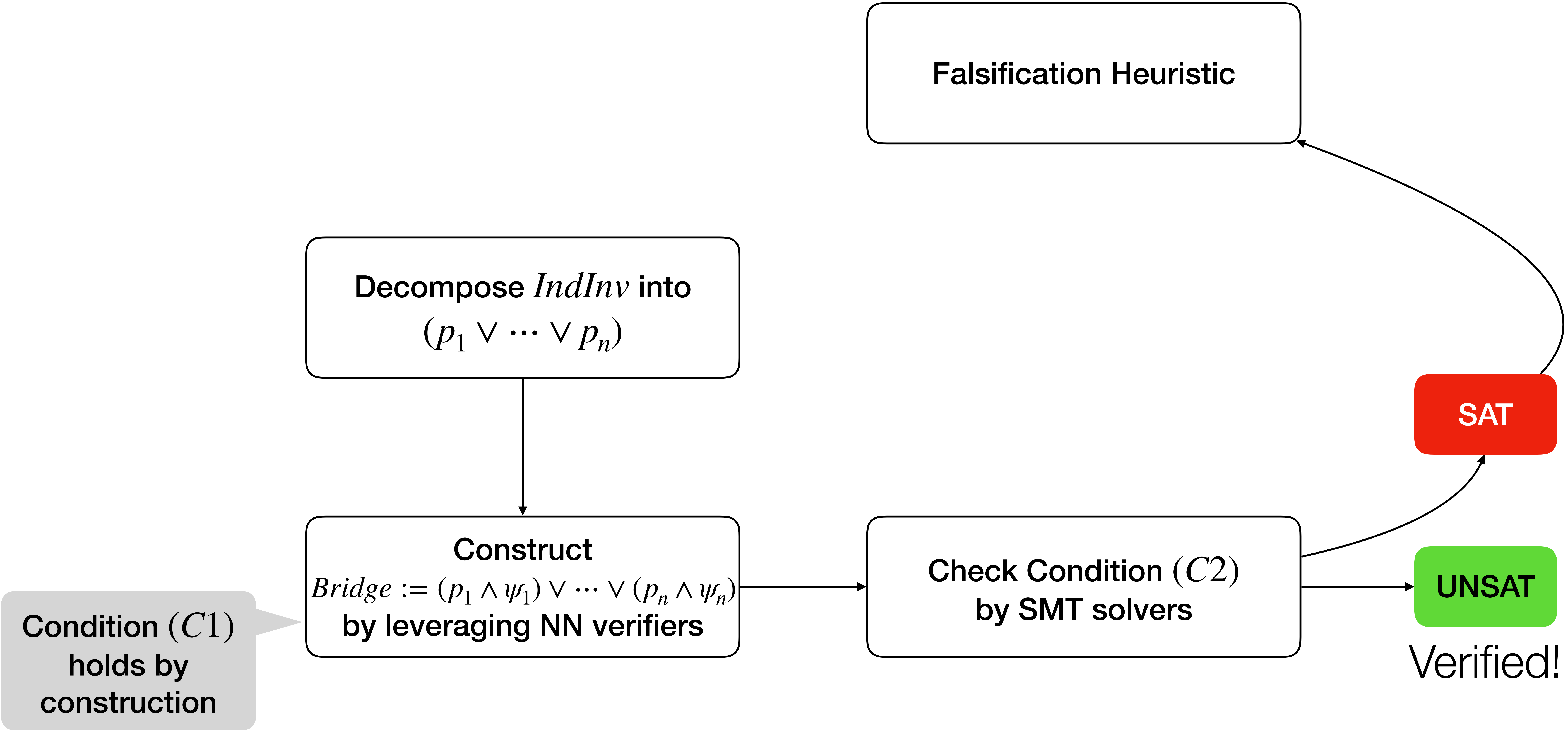
Algorithm with Falsification



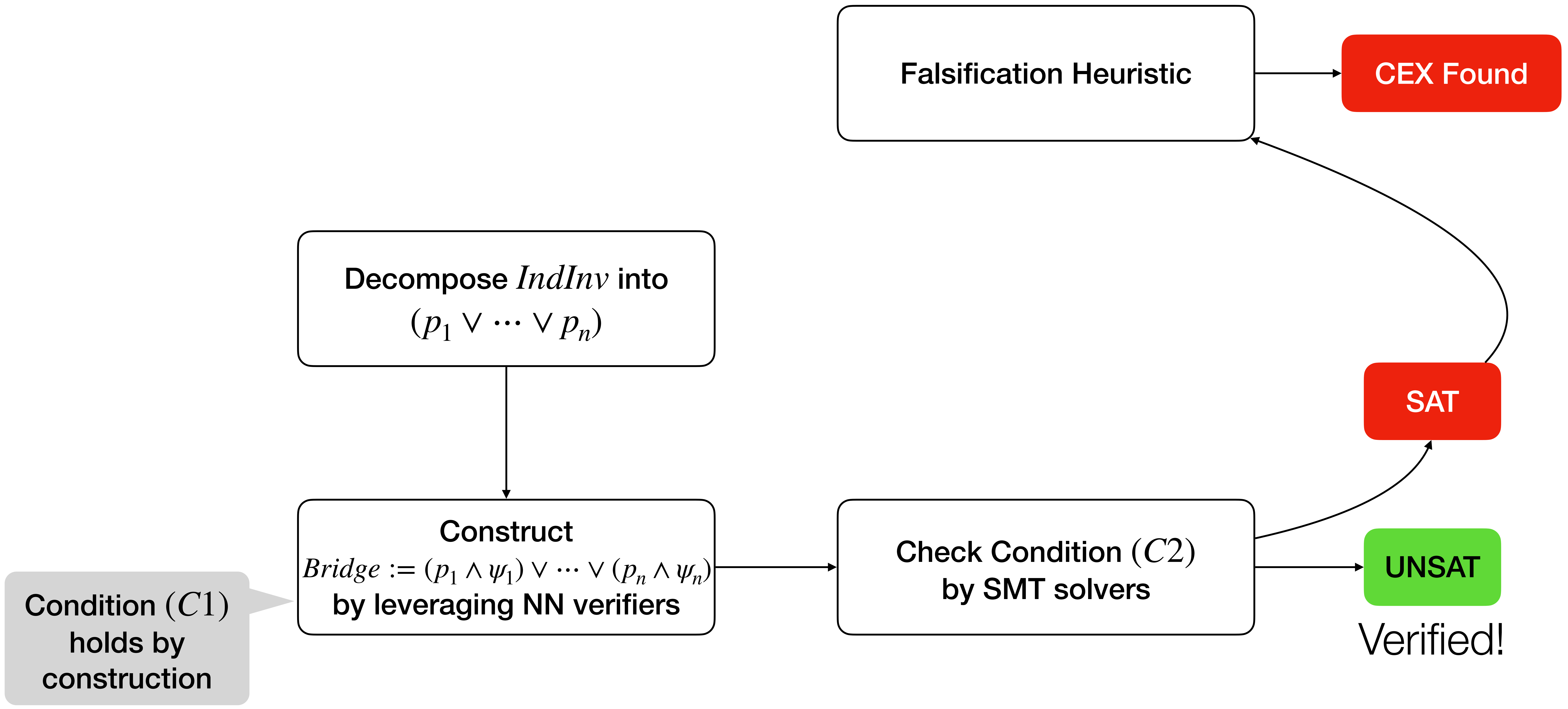
Algorithm with Falsification



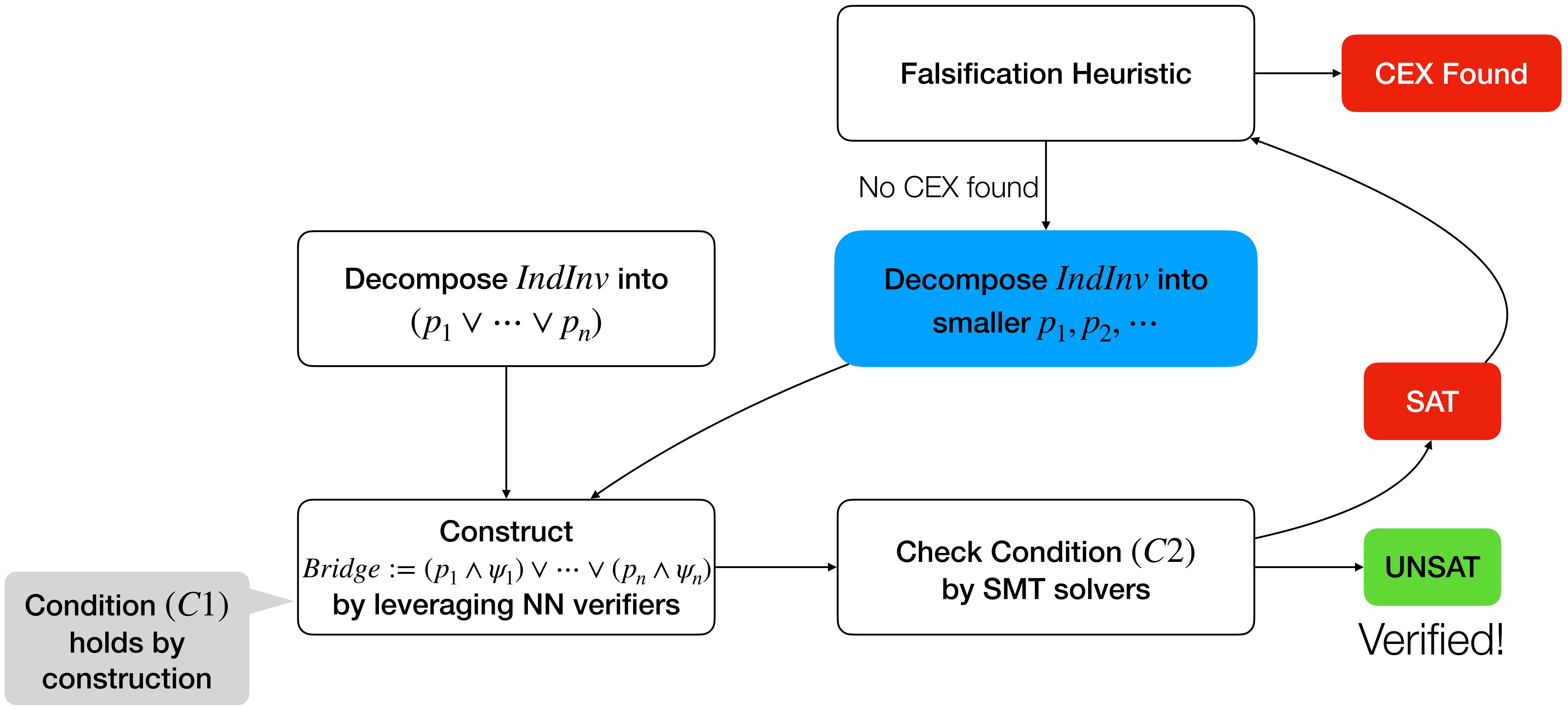
Algorithm with Falsification



Algorithm with Falsification



Algorithm with Falsification



Termination of the Algorithm

Termination of the Algorithm

Theorem: Completeness

If (M) holds, then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

Termination of the Algorithm

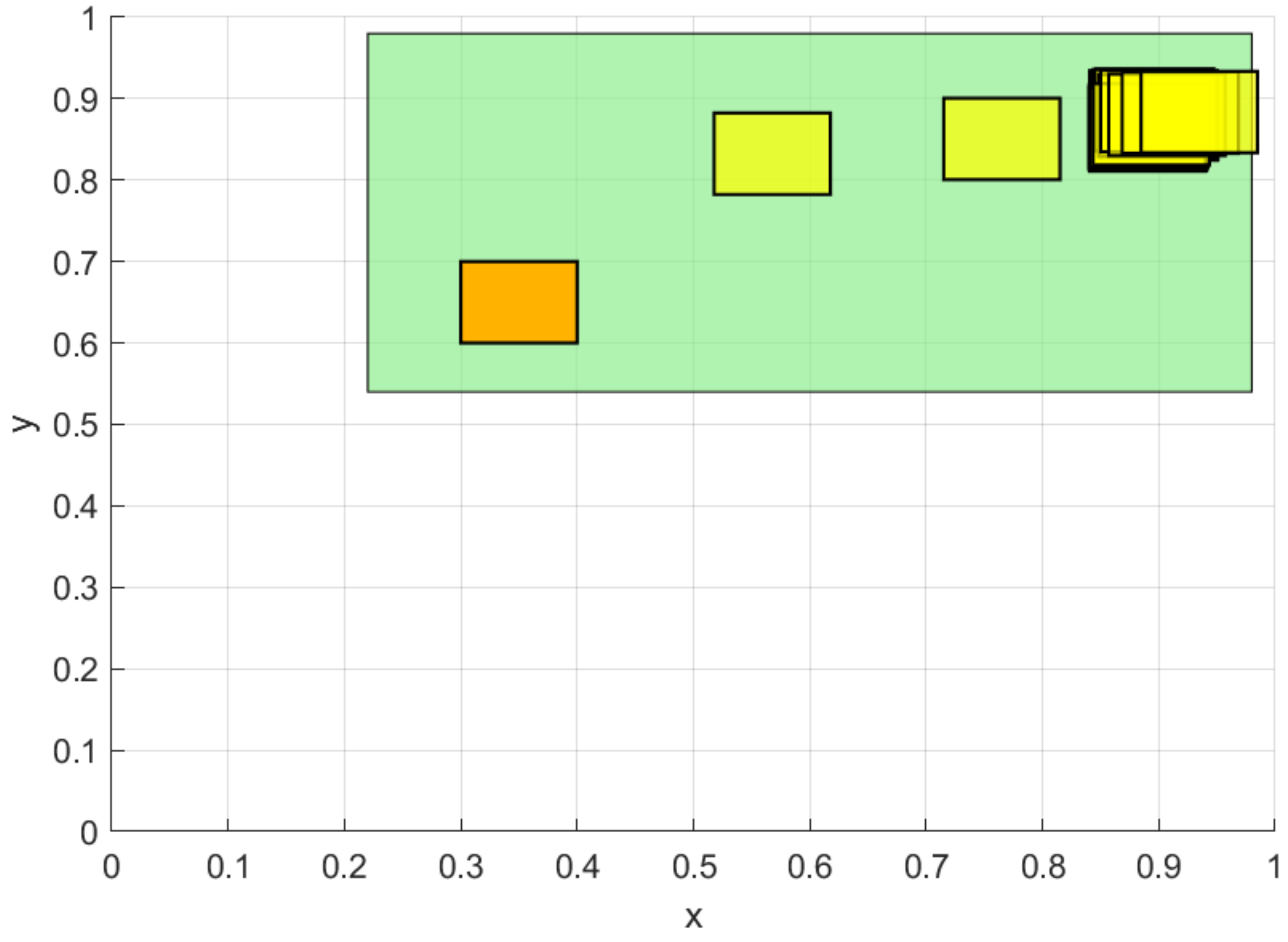
Theorem: Completeness

If (M) holds, then there exists $Bridge$ such that $(C1)$ and $(C2)$ hold

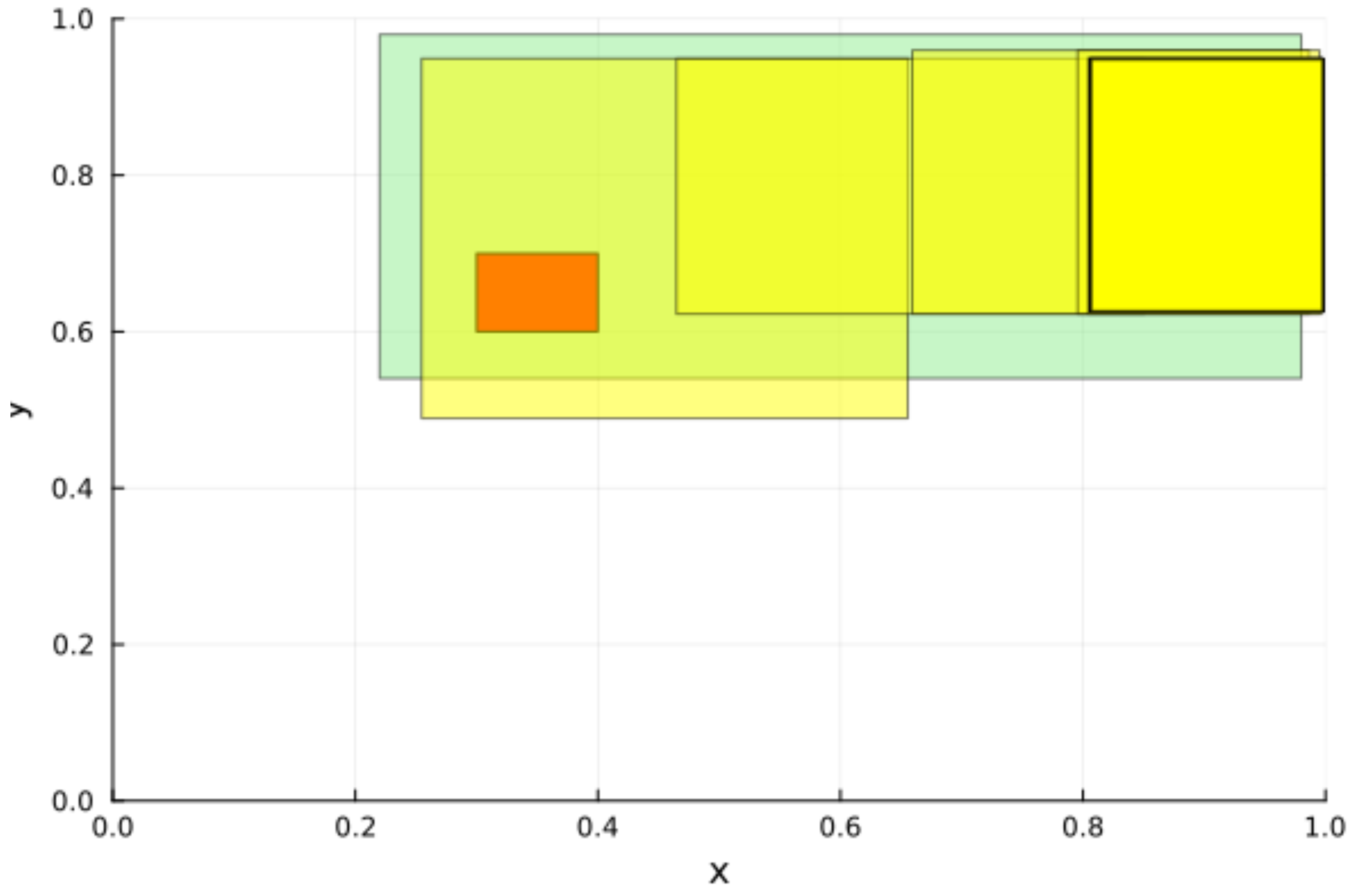
Algorithm

Termination not guaranteed

Compare with Reachability Analysis



NNV (2 x 512)



JuliaReach (2 x 512)

Why NN Verifiers Not Working for Monolithic Method

I. Encode $Next_{ENV}$ into NNV Specification

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

Use NNV as black-box

NN takes x, outputs y

NNV Specification

Given $P(x)$, guarantee $Q(y)$

Why NN Verifiers Not Working for Monolithic Method

I. Encode $Next_{ENV}$ into NNV Specification

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

not straightforward to encode,
e.g. our case study

Use NNV as black-box

NNV Specification

NN takes x, outputs y

Given $P(x)$, guarantee $Q(y)$

Why NN Verifiers Not Working for Monolithic Method

II. Encode $Next_{ENV}$ as a Part of Neural Network

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

Why NN Verifiers Not Working for Monolithic Method

II. Encode $Next_{ENV}$ as a Part of Neural Network

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

$Next_{ENV}$ may contain operators that NNVs cannot handle

Why NN Verifiers Not Working for Monolithic Method

II. Encode $Next_{ENV}$ as a Part of Neural Network

$$(IndInv \wedge Next_{NNC} \wedge Next_{ENV}) \implies IndInv'$$

$Next_{ENV}$ may contain operators that NNVs cannot handle

$Next_{ENV}$ may be non-deterministic, while NNVs typically assume NN to be deterministic