Loop Invariant Synthesis with Machine Learning

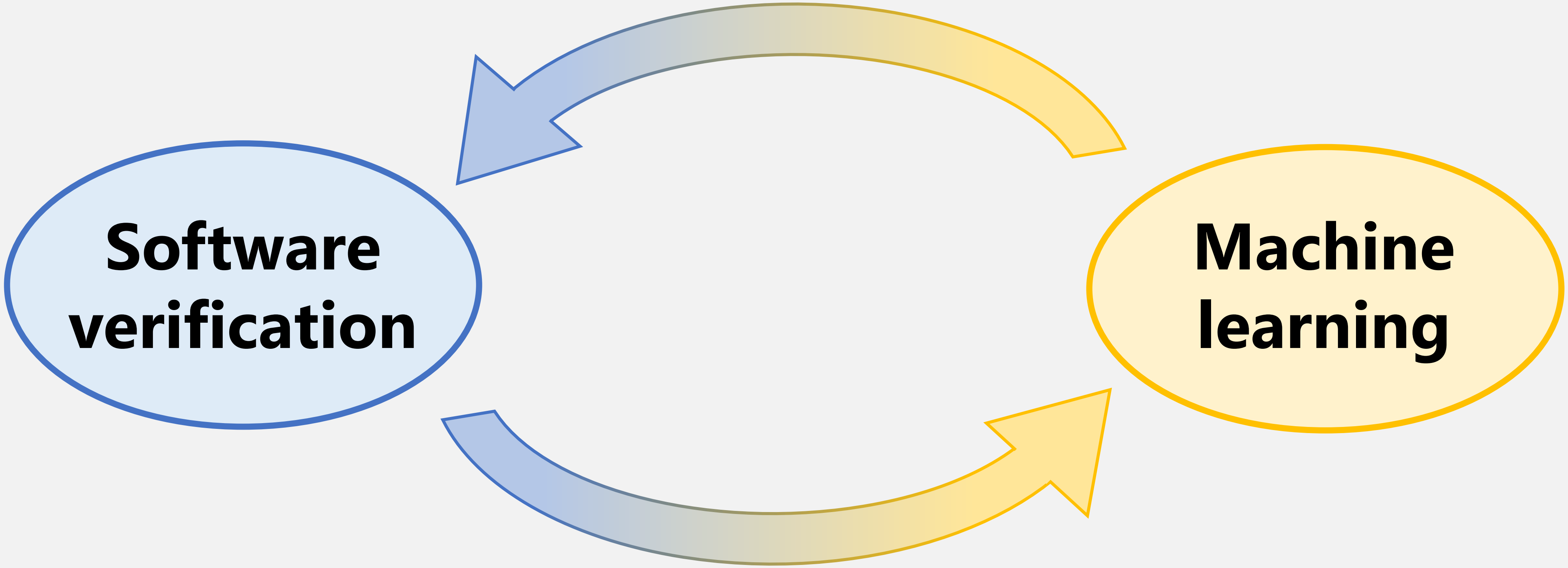
Taro Sekiyama

National Institute of Informatics

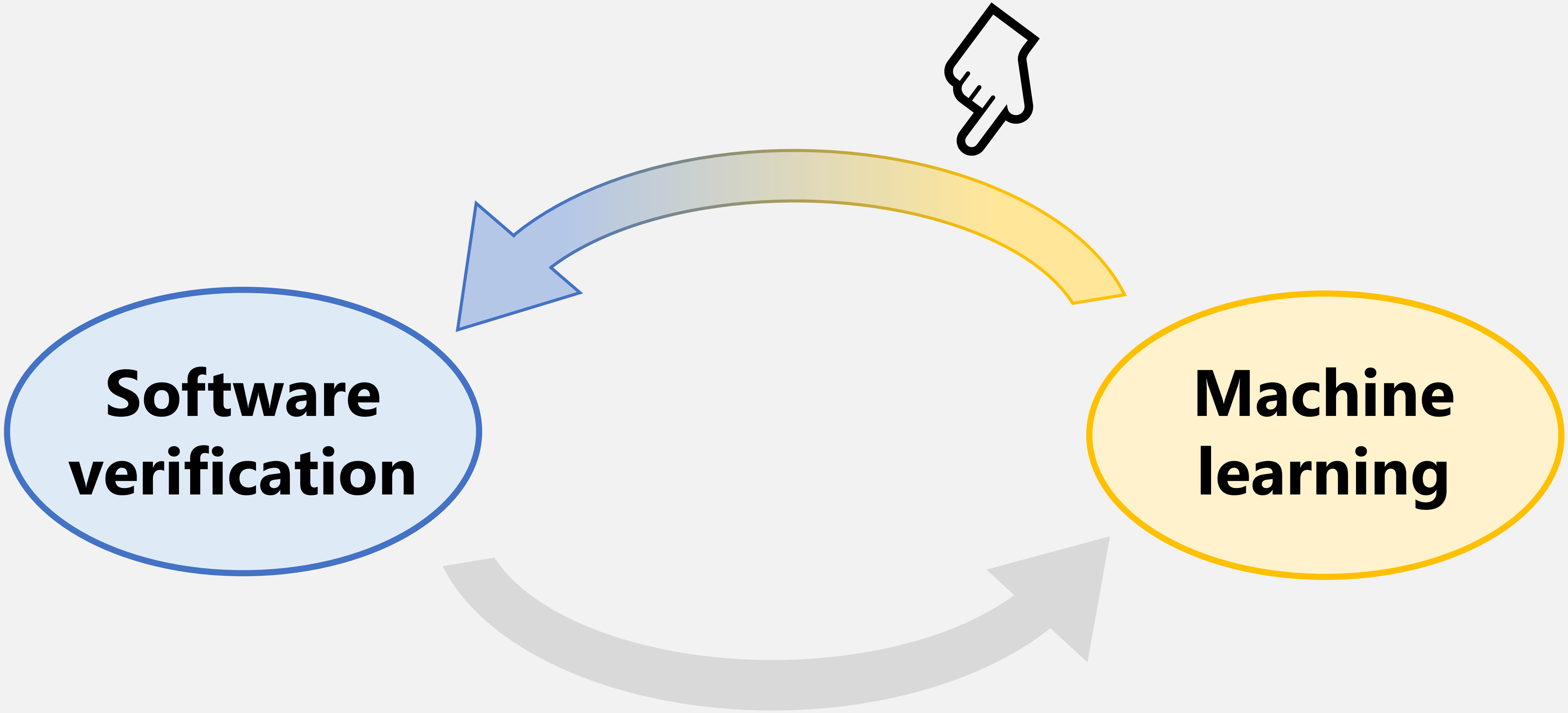
Including joint work with

Unno Hiroshi, Naoki Kobayashi, Issei Sato,
Kohei Suenaga, Takeshi Tsukada, Minchao Wu

Software verification & Machine learning



Software verification & Machine learning



Accidents caused by software bugs

- Therac-25 radiation therapy
  - Involved six accidents of radiation overdoses
- Ariane 5 rocket
  - Resulted in the launch failure and a loss > $370 million

- Heartbleed (an OpenSSL vulnerability)
  - Major servers (Apache, nginx, etc.) on the internet were vulnerable
- Others: [List of software bugs (Wikipedia)]

Program verification

Methodology to assure correctness of programs
by **mathematical reasoning**

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1; }
 return y; }
```

Program P

Specification ψ

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Verification tool

Program P has been ***proven*** to work as specified by ψ

OK

Program verification

Methodology to assure correctness of programs
by **mathematical reasoning**

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1; }
 return y; }
```

Program $\boldsymbol{P}$

Specification $\boldsymbol{\psi}$

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Verification tool

NG

A counterexample that witnesses
$\boldsymbol{P}$ doesn't work as specified by $\boldsymbol{\psi}$

Difference from software testing

Program verification

- **Logical** Specification

$$\forall n \in \mathbb{N}.\, P(n) = \sum_{i=0}^{n} i$$

- Assuring correctness for **any** input

Input	Correctness Assured
1	✓
2	✓
3	✓

Software testing

- **Executable** Specification

```
assert (P(1) == 1);
assert (P(2) == 3);
assert (P(5) == 15);
```

- Assuring correctness for **given** inputs

Input	Correctness Assured
1	✓
2	✓
3	×

Verification of real-world software

- **SLAM:** a research project to verify Windows device drivers
- **Infer:** a verification tool for Java, C, and C++ code
  - Used to verify Facebook's Android / iOS apps
- **CPAcheker:** a verification tool for C
  - Used to verify control software of airplanes and a space station
- **Astree:** a verification tool for C
  - Used to verify Linux device drivers

in Hoare Logic

Let’s try verification!

Example: sum from 1 to n

Specification

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
  int x = n, y = 0;
  while (x != 0) {
    y = y + x;
    x = x - 1;
  }
★ return y;
}
```

}?

Goal

Proving "$y = \sum_{i=0}^{n} i$" holds after exiting from the loops (★)

Question

What holds during the loops?

Example: sum from 1 to n

Specification

$$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

Before the loop

x	y
n	0

Example: sum from 1 to n

Specification

$$\forall n \in \mathbb{N}.\, P(n) = \sum_{i=0}^{n} i$$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n

Example: sum from 1 to n

Specification

$$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n
After the 2nd loop	n-2	n + n-1

Example: sum from 1 to n

Specification

$$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n
After the 2nd loop	n-2	n + n-1
After the 3rd loop	n-3	n + n-1 + n-2

Example: sum from 1 to n

Specification

$$\forall n \in \mathbb{N}.\, P(n) = \sum_{i=0}^{n} i$$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n
After the 2nd loop	n-2	n + n-1
After the 3rd loop	n-3	n + n-1 + n-2
	…	…
After the nth loop	0	n + n–1 + n-2 + … + 1

Example: sum from 1 to n

$= \sum_{i=x+1}^{n} i$

Specification

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n
After the 2nd loop	n-2	n + n-1
After the 3rd loop	n-3	n + n-1 + n-2
	…	…
After the nth loop	0	n + n–1 + n-2 + … + 1

$(x \geq 0)$

is a property that holds before / after every loop

Example: sum from 1 to n

$= \sum_{i=x+1}^{n} i$

Specification

$\forall n \in \mathbb{N}.\, P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
 return y;
}
```

	x	y
Before the loop	n	0
After the 1st loop	n-1	n
After the 2nd loop	n-2	n + n-1
After the 3rd loop	n-3	n + n-1 + n-2
	…	…
After the nth loop	0	n + n–1 + n-2 + … + 1

$$\left(\sum_{i=0}^{x} i + y = \sum_{i=0}^{n} i\right) \wedge (x \geq 0)$$

is a property that holds before / after every loop

Loop invariant

Example: sum from 1 to n

Specification

$\forall n \in \mathbb{N}.\, P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
  int x = n, y = 0;
  while (x != 0) {
    y = y + x;
    x = x - 1;
  }
★ return y;
}
```

✓

Goal

Proving "$y = \sum_{i=0}^{n} i$" holds after exiting from the loops (★)

Answer

Loop invariant
$\phi(x, y) \equiv (\sum_{i=0}^{x} i + y = \sum_{i=0}^{n} i\,) \land (x \geq 0)$

Proof of the goal

The final loop exits with $x = 0$, so
$\phi(0, y) \equiv \left(\sum_{i=0}^{0} i + y = \sum_{i=0}^{n} i\right) \land (0 \geq 0)$
holds

Example: sum from 1 to n

Specification

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
  int x = n, y = 0;
  while (x != 0) {
    y = y + x;
    x = x - 1;
  }
★ return y;
}
```

Goal

Proving "$y = \sum_{i=0}^{n} i$" holds after exiting from the loops (★)

Answer

Loop invariant

$\phi(x, y) \equiv (\sum_{i=0}^{x} i + y = \sum_{i=0}^{n} i) \land (x \geq 0)$

Proof of the goal

The final loop exits with $x = 0$, so

$\phi(0, y) \equiv (\sum_{i=0}^{0} i + y = \sum_{i=0}^{n} i) \land (0 \geq 0)$

holds

Example: sum from 1 to n

Specification

$\forall n \in \mathbb{N}. \mathrm{P}(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
  int x = n, y = 0;
  while (x != 0) {
    y = y + x;
    x = x - 1;
  }
★ return y;
}
```

Goal

Proving “$y = \sum_{i=0}^{n} i$” holds after exiting from the loops (★)

Answer

Loop invariant

$\phi(x, y) \equiv (\sum_{i=0}^{x} i + y = \sum_{i=0}^{n} i) \wedge (x \geq 0)$

Proof of the goal

The final loop exits with $x = 0$, so

$\phi(0, y) \equiv (\quad y = \sum_{i=0}^{n} i) \wedge (0 \geq 0)$

holds

Example: sum from 1 to n

Specification

$\forall n \in \mathbb{N}. \mathrm{P}(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1;
 }
★ return y;
}
```

✓

Goal

Proving "$y = \sum_{i=0}^{n} i$" holds after exiting from the loops (★)

Answer

Loop invariant
$\phi(x, y) \equiv (\sum_{i=0}^{x} i + y = \sum_{i=0}^{n} i) \wedge (x \geq 0)$

Proof of the goal ✓

The final loop exits with $x = 0$, so
$\phi(0, y) \equiv (\boldsymbol{y = \sum_{i=0}^{n} i}) \wedge (0 \geq 0)$
holds

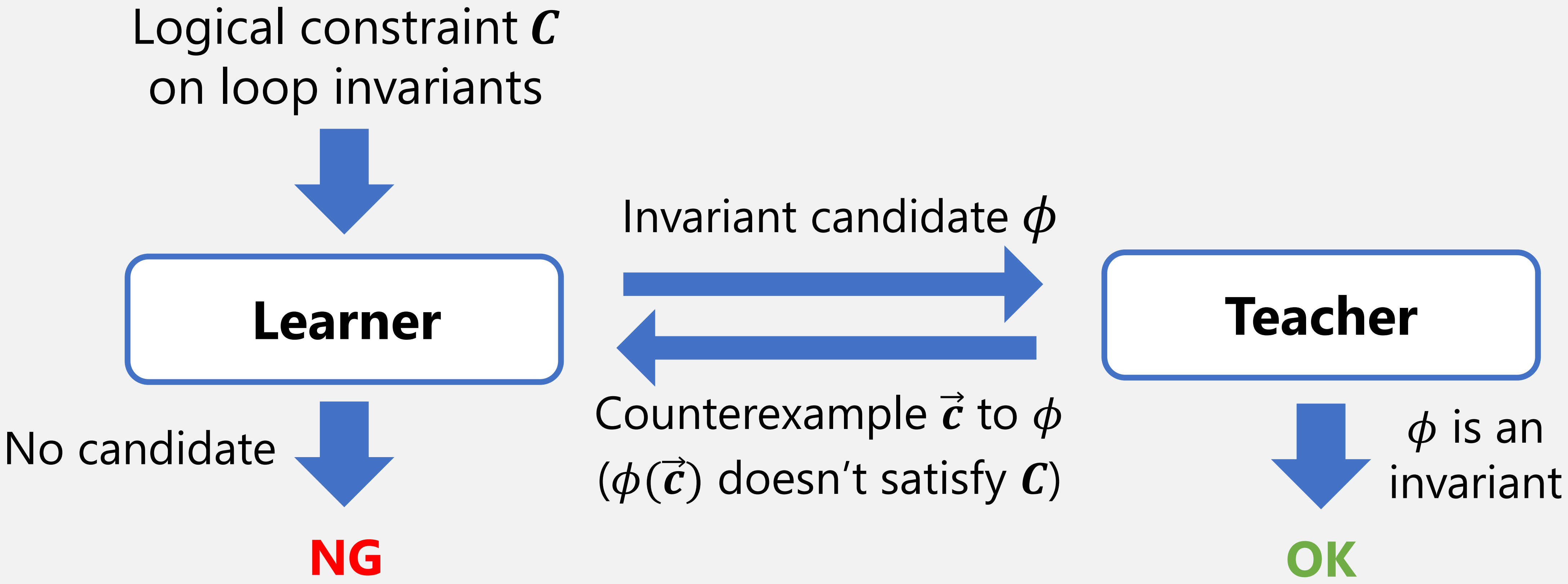
Challenge in automating verification

Finding loop invariants

- It is an undecidable problem in general
- (Incomplete) approaches to invariant synthesis
  - **Learning-based approaches**
  - Template-based approaches
    - Fixing the shape of invariants and searching for parameters that satisfy constraints on invariants

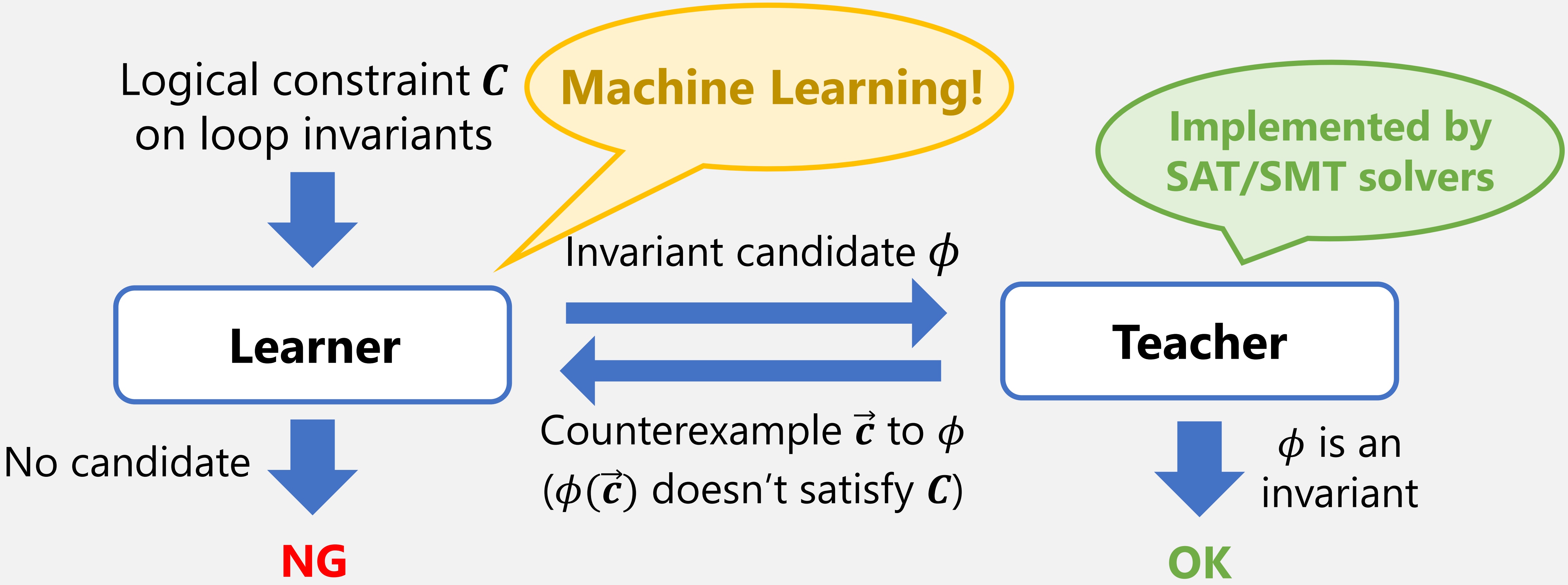
Learning framework for invariant synthesis

Interleaving **learning** and **checking** of invariant candidates



Learning framework for invariant synthesis

Interleaving **learning** and **checking** of invariant candidates



Invariant learning

Specification

$\forall n \in \mathbb{N}. P(n) = \sum_{i=0}^{n} i$

Program

```
int P(int n) {
 int x = n, y = 0;
 while (x != 0) {
  y = y + x;
  x = x - 1; }
 return y; }
```

Goal: To find a loop invariant ϕ

Given:

A set $\mathcal{E}$ of counterexamples to candidates, categorized into:

- Positive examples: $\vec{c}$ s.t. $\phi(\vec{c})$ must be true

  In P, $\forall n \geq 0.\ \phi(x \coloneqq n, y \coloneqq 0, n)$ must be true as invariants must hold before entering the loops

- Negative examples: $\vec{c}$ s.t. $\phi(\vec{c})$ must be false

  In P, $\phi(x \coloneqq 0, y \coloneqq 10, n \coloneqq 2)$ must be false as $y = \Sigma_{i=0}^{n} i$ must hold after exiting the loops

- Implication constraints: $(\vec{c}, \vec{d})$ s.t. $\phi(\vec{c}) \Longrightarrow \phi(\vec{d})$

ML-based approaches to invariant learning

- ML to lean invariants
  - **Learning invariants as decision trees**
    [Krishna, Puhrsch & Wies, arXiv'15; Garg, Neider, Madhusudan & Roth, POPL'16]
  - Learning by deep reinforcement learning
    [Si, Dai, Raghothaman, Naik & Song, NeurIPS'18]
  - Encoding constraints into neural networks
    [Ryan, Wong, Yao, Gu & Jana, ICLR'20 & PLDI'20]
- ML to aid symbolic reasoning
  - **Speeding up symbolic approaches with reinforcement learning**
    [Tsukada, Unno, Sekiyama & Suenaga, arXiv'21]

ML-based approaches to invariant learning

- ML to lean invariants
  - **Learning invariants as decision trees**

    [Krishna, Puhrsch & Wies, arXiv'15; Garg, Neider, Madhusudan & Roth, POPL'16]
  - Learning by deep reinforcement learning

    [Si, Dai, Raghothaman, Naik & Song, NeurIPS'18]
  - Encoding constraints into neural networks

    [Ryan, Wong, Yao, Gu & Jana, ICLR'20 & PLDI'20]
- ML to aid symbolic reasoning
  - **Speeding up symbolic approaches with reinforcement learning**

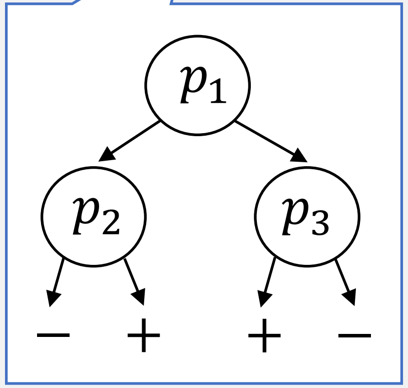
    [Tsukada, Unno, Sekiyama & Suenaga, arXiv'21]

Learning invariants as decision trees

Decision tree learning

Converting to logical formula

Example set $\mathcal{E}$ → Decision tree → Invariant candidate



$$(p_1 \Longrightarrow \neg p_2) \land (\neg p_1 \Longrightarrow p_3)$$

Nodes are predicates over program variables

E.g., for integer-manipulating programs, every node $\boldsymbol{p_i}$ is an inequation $\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{x}} \geq \boldsymbol{c}$

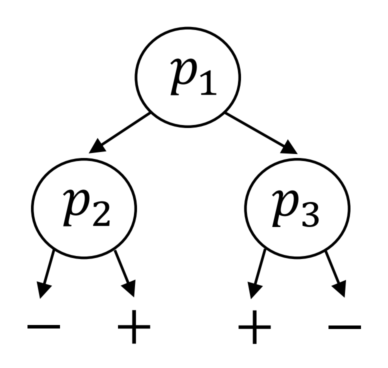
- $\vec{\boldsymbol{x}}$ are program variables of integers
- $\vec{\boldsymbol{a}}, \boldsymbol{c}$ are **parameters to be optimized**

de

Challenge: poor scalability of decision tree learning in the number of parameters $\vec{a}, c$

Optimizing $\vec{\boldsymbol{a}}, \boldsymbol{c}$ in each node

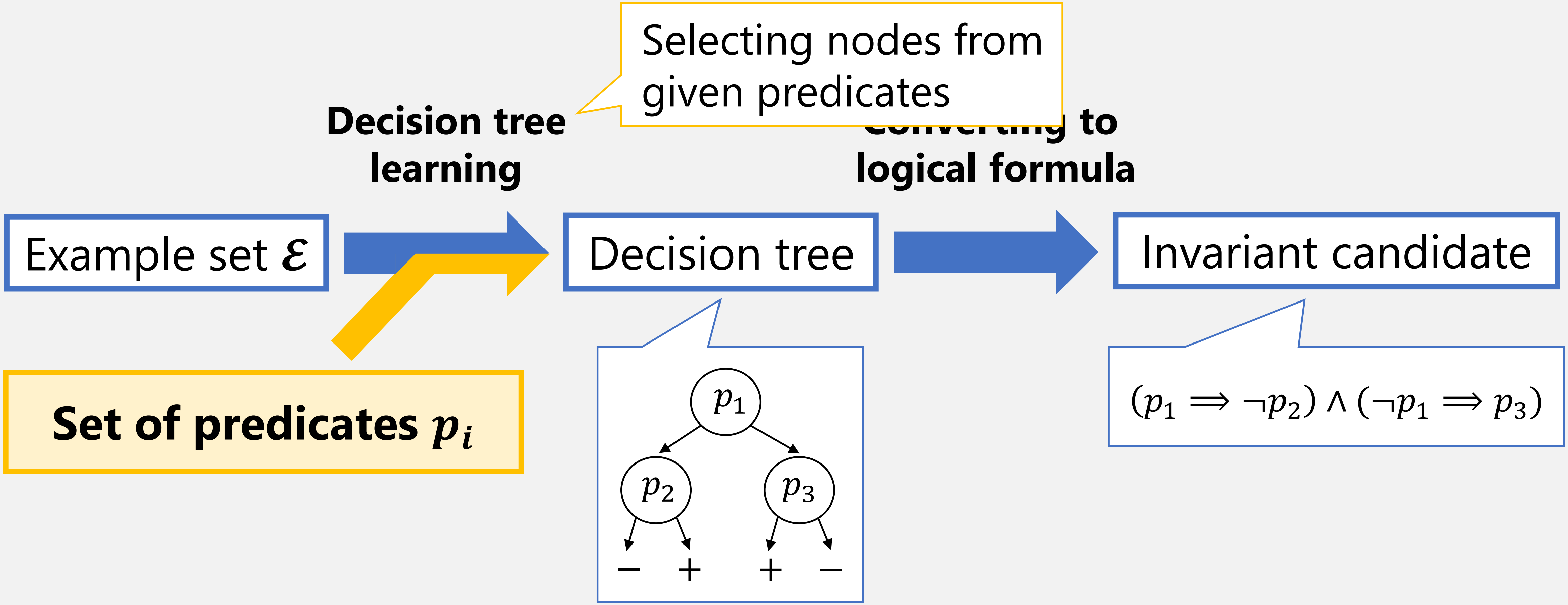
Example set $\mathcal{E}$ → **Decision tree learning** → Decision tree → **Converting to logical formula** → Invariant candidate



$(p_1 \Rightarrow \neg p_2) \land (\neg p_1 \Rightarrow p_3)$

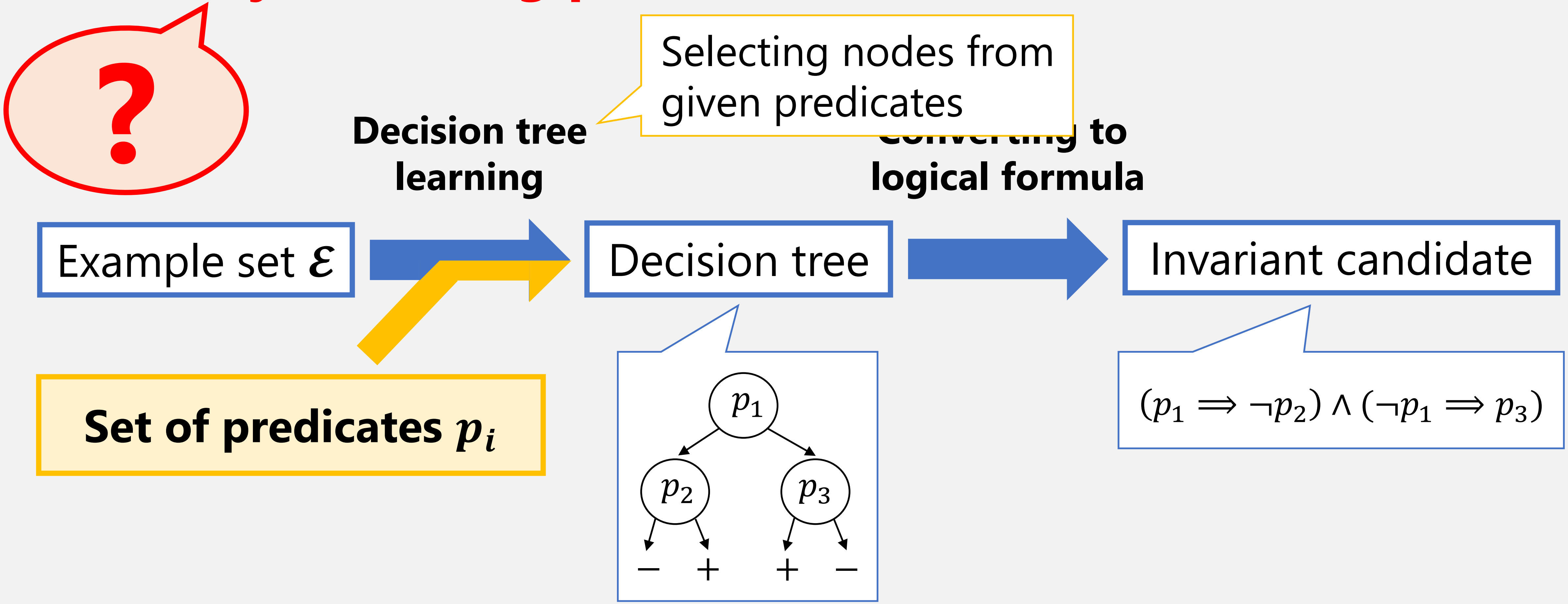
Solution to scalability

- Pre-synthesizing predicates used as nodes



Solution to scalability

- **Pre-synthesizing predicates used as nodes**



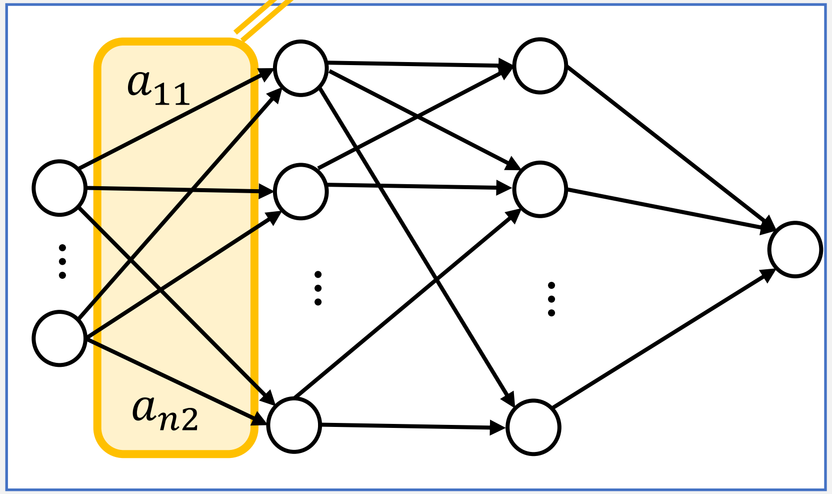
Neural synthesis of predicates over integers

[Kobayashi, Sekiyama, Sato & Unno, SAS'21]

x_1	x_2	label
4	3	-
5	3	+
6	3	-
…	…	…

Example set $\mathcal{E}$

Training



Synthesis from the weights $\overrightarrow{a_i}$ in the NN

$$\left\{ \begin{array}{l} x_1 + x_2 + 9 > 0, \\ 4x_1 - x_2 > 0, \\ \dots \end{array} \right\}$$

Predicates used in DT learning

Predicate synthesis from NNs

Idea: Designing a NN that encodes invariants and represents predicate parameters $\vec{a}, c$ as weights

Assumption: Formulas are logical combinations of $\vec{a} \cdot \vec{x} + c > 0$

Decision tree learning with synthesized predicates to generate invariant candidates

$y_i = \sigma(\vec{a_i} \cdot \vec{x} + c_i)$, so

$y_i \simeq 1 \Leftrightarrow \vec{a_i} \cdot \vec{x} + c_i > 0$

$y_i \simeq 0 \Leftrightarrow \vec{a_i} \cdot \vec{x} + c_i < 0$

if $|\vec{a_i} \cdot \vec{x} + c_i| \gg 0$

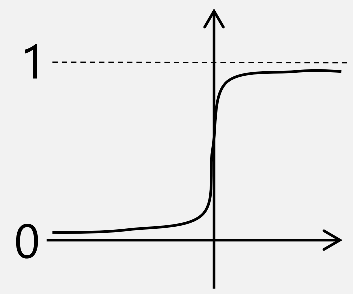
$y_i \approx p_i$

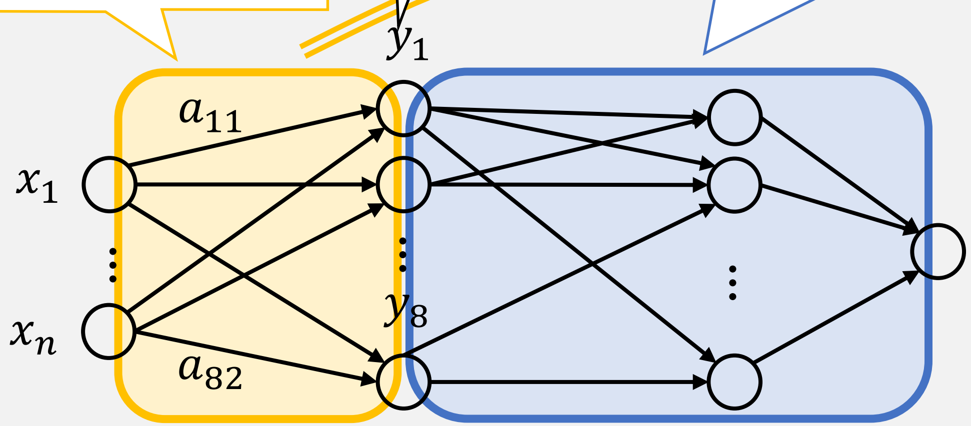
Intended to recognize logical combinations (e.g., $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$)

$$\begin{cases} 2x_1 + 3x_2 + 9 > 0, \\ 4x_1 + 1x_2 + 0 > 0 \end{cases}$$

Grouping the ratios $a_{i1} : a_{i2} : c_i$

σ : sigmoid function





Feedforward NN with 2 hidden layers

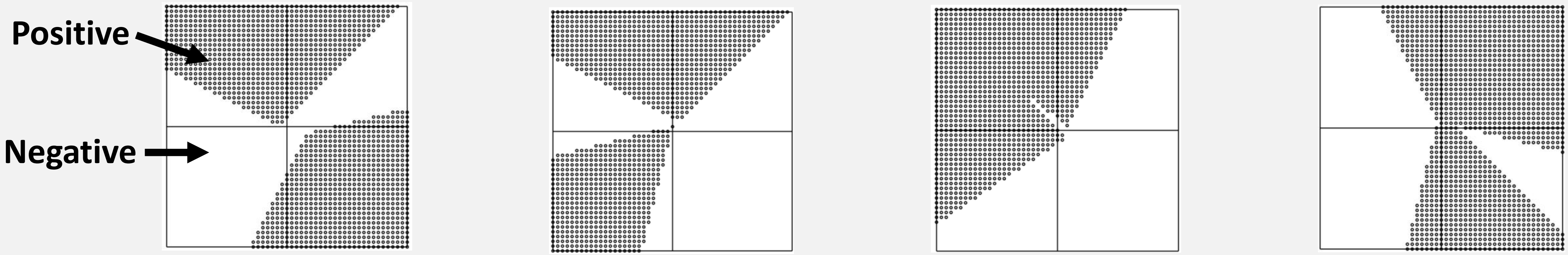
	a_{i1}	a_{i2}	c_i	o_i
y_1	4.037725448	6.05603551[illegible]	[illegible]	−11.76355457
y_2	4.185569763	6.27788019[illegible]	[illegible]	−11.36994552
	3.775603055	5.662680149	16.86475944	−10.83486366
	3.928676843	5.892404079	17.63601112	−10.78136634
⋮	−15.02299022	−3.75841569[illegible]	[illegible]	−9.199707984
	−13.6469354	−3.41494297[illegible]	[illegible]	−8.159229278
	−11.69845199	−2.927870512	0.8412334322	−7.779587745
y_8	−12.65479946	−3.168056249	0.9739738106	−6.938682556

2 : 3 : 9

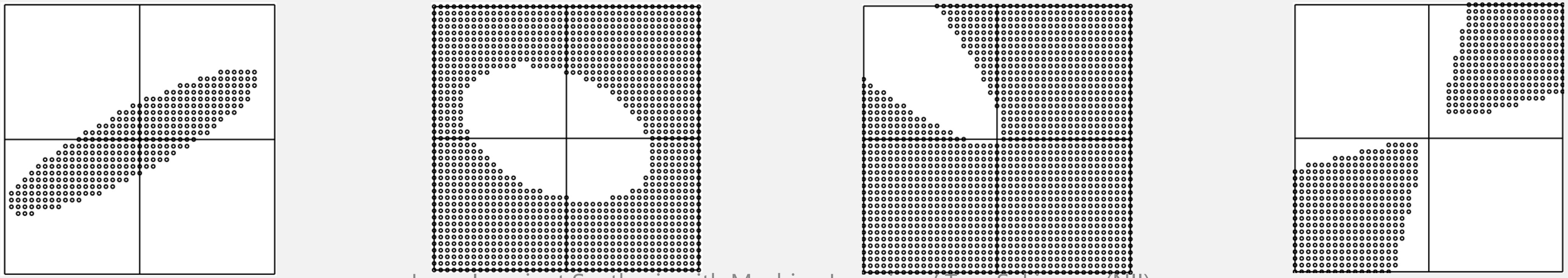
4 : 1 : 0

Experiments

- Predicate synthesis works well on linear invariants



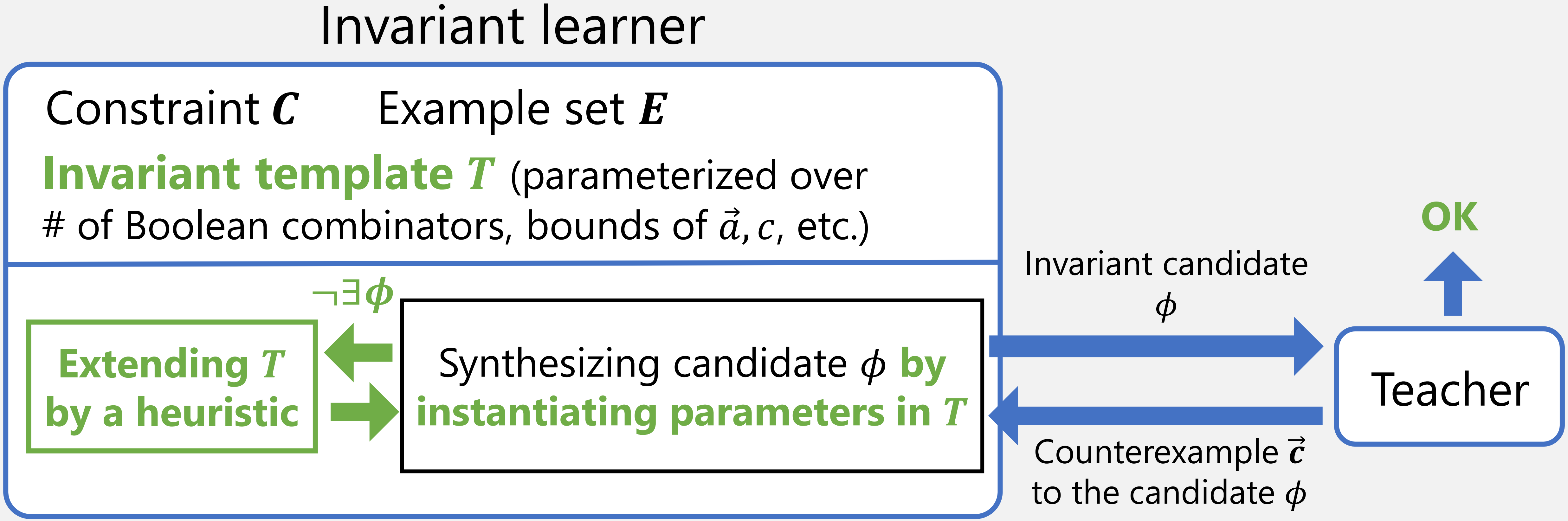
- Quadratic invariants can be supported by preprocessing inputs to the NN



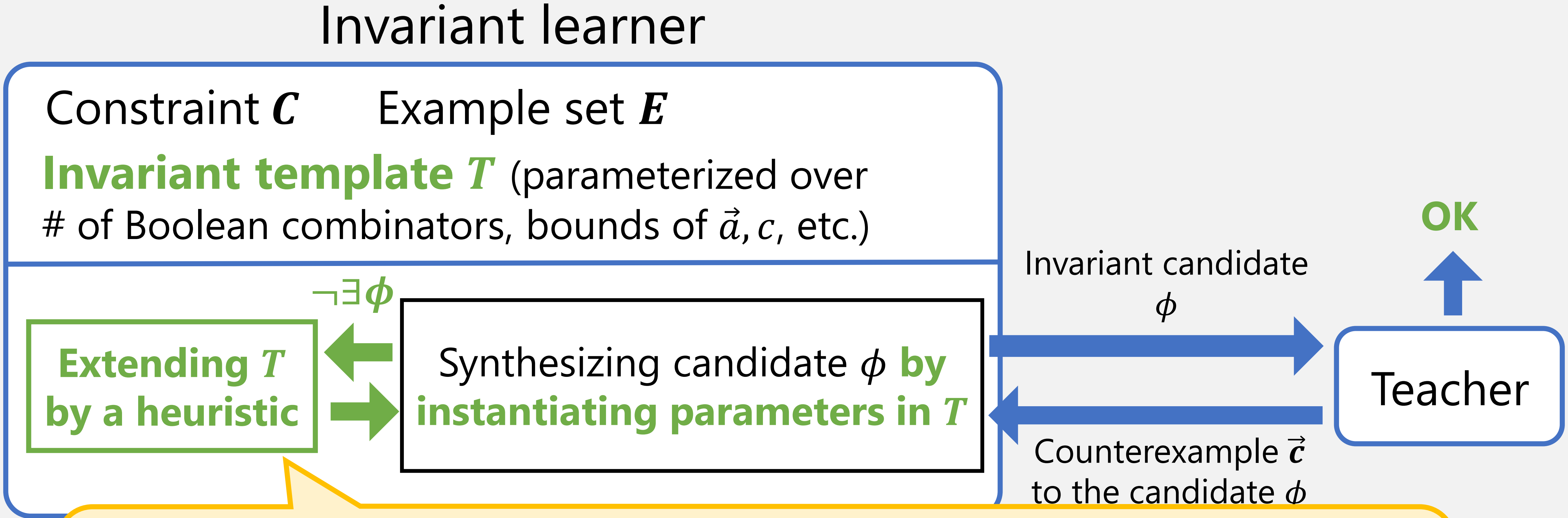
ML-based approaches to invariant learning

- ML to lean invariants
  - **Learning invariants as decision trees**
    [Krishna, Puhrsch & Wies, arXiv'15; Garg, Neider, Madhusudan & Roth, POPL'16]
  - Learning by deep reinforcement learning
    [Si, Dai, Raghothaman, Naik & Song, NeurIPS'18]
  - Encoding constraints into neural networks
    [Ryan, Wong, Yao, Gu & Jana, ICLR'20 & PLDI'20]
- ML to aid symbolic reasoning
  - **Speeding up symbolic approaches with reinforcement learning**
    [Tsukada, Unno, Sekiyama & Suenaga, arXiv'21]

Template-based symbolic invariant synthesis



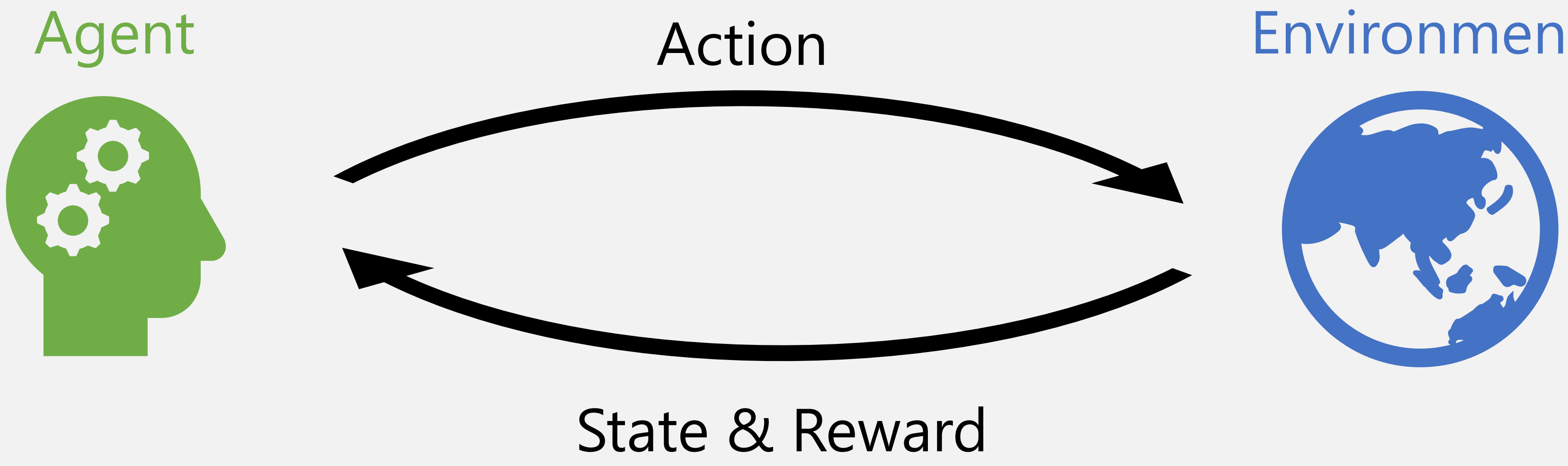
Template-based symbolic invariant synthesis



- **Challenge:** finding **effective heuristics** for template extension
- **Approach:** applying **reinforcement learning** to optimize heuristic strategies

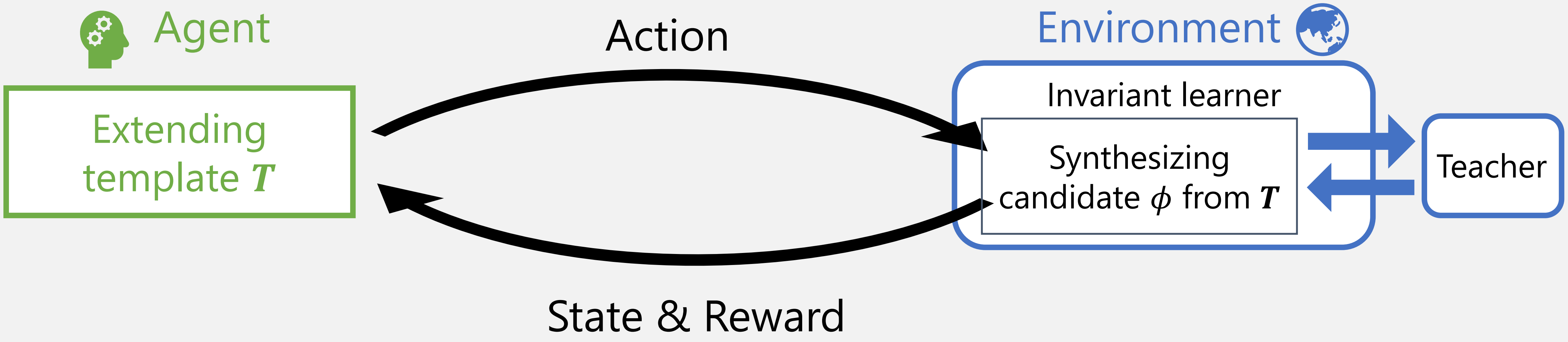
Reinforcement learning

Learning strategies of agent's actions to maximize total rewards obtained from environments



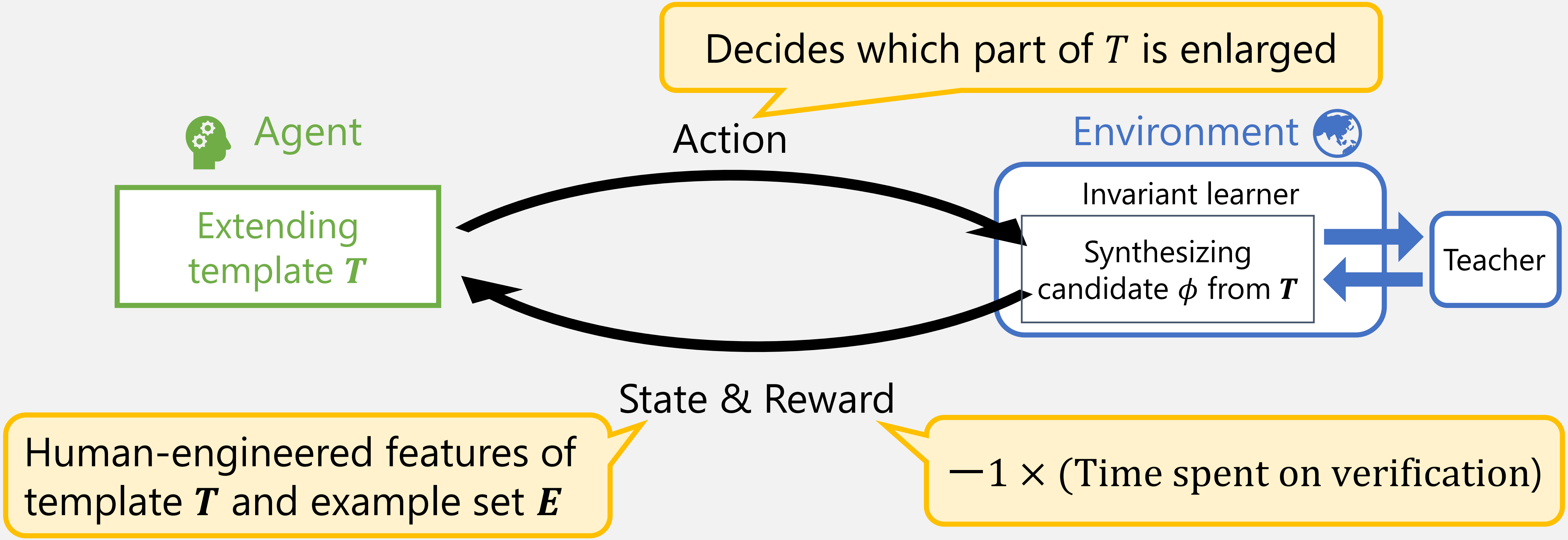
Applying to heuristic learning

Goal: learning template extension strategies to minimize the total time spent by invariant synthesis



Applying to heuristic learning

Goal: learning template extension strategies to minimize the total time spent by invariant synthesis



Experiments

- Implemented on a verifier **PCSat** [Unno+, AAAI'20&CAV'21]
- Effective heuristics can be learned!

Tool	# of solved test problems (total # = 171)
PCSat w/ Advantage Actor-Critic	**154 (90.05%)**
PCSat w/ Monte Carlo	**155 (90.06%)**
LoopInvGen	92 (53.80%)
CVC4	111 (64.91%)
Eldarica	131 (76.61%)
PCSat w/ the hand-tuned heuristic	**144 (84.21%)**
HoIce	149 (87.13%)
Spacer	165 (96.49%)

Ours: PCSat with learned heuristics (PCSat w/ Advantage Actor-Critic, PCSat w/ Monte Carlo)

Baseline (LoopInvGen, CVC4, Eldarica, PCSat w/ the hand-tuned heuristic, HoIce, Spacer)

Outline

1. Introduction to program verification
2. Learning-based invariant synthesis
3. **Conclusion**

Findings

Applying ML to verification is possible but hard

- Program verification is **deductive**, while ML is **inductive**

"Softer" program verification is more suitable to use ML?

- Program verification addresses **hard constraints**, while some of ML techs target only **soft constraints**
- Needing a means to **interpret / explain ML models logically**
  - E.g., converting decision trees to logical formulas, extracting predicates from weights in a neural net
- Available are only small datasets (of the sizes from 10 to 1000)

Conclusion

- A main bottleneck of automating verification is invariant synthesis
- Data-driven invariant synthesis is emerging!
- **Collaboration b/w ML and verification is promising and challenging**