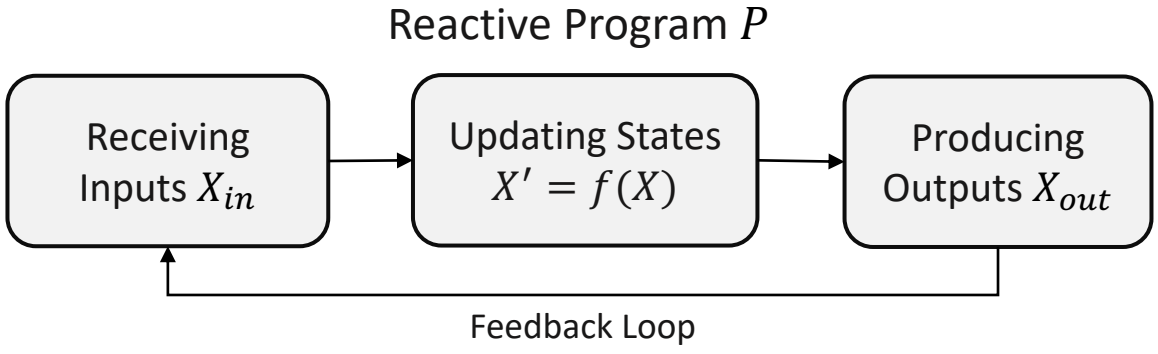
Active Learning of Symbolic Automata for Reactive Programs via Dynamic Symbolic Mapper

Yoel Kim and Yunja Choi

Reactive Programs

- It **continuously** interacts with external **inputs**, updates internal **states**, and produces **outputs** within an **infinite loop**.



- A reactive program P is a tuple $(Q, X_{in}, X_{out}, f, q_0)$:
  - Q is a set of states; X_{in} and X_{out} are the sets of input and output variables;
  - $f: Q \times Val(X_{in}) \rightarrow Q \times Val(X_{out})$ is a transition function;
  - It allows a **feedback loop**, i.e., $X_{in} \cap X_{out} \neq \emptyset$.

Example: Mine-Pump Controller

```
1 : while (true) {
2 :   w := ReadWater();
3 :   m := ReadMethane();

4 :   if (st = Ready) {
5 :     h := 0;
6 :     if (m < 600 ∧ w ≥ 70) {
7 :       st := Pump; }}
```

```
8 :     else if (st = Pump) {
9 :       h := h + 1;
10 :      if (m ≥ 600) {
11 :        st := Locked; a := true; }
12 :      else if (w ≤ 30) {
13 :        st := Ready; }
14 :      else if (h ≥ 3 ∧ w ≤ 50) {
15 :        st := Ready; }}
```

```
16 :    else if (st = Locked) {
17 :      if (h > 0) {
18 :        h := h − 1; }
19 :      if (¬a) {
20 :        st := Pump; }
21 :      if (m ≤ 550) {
22 :        a := false; }}
23 : }
```

- $X_{in} = \{w, m, st, h, a\}$; $X_{out} = \{st', h', a'\}$.
- We write $X_{st} = X_{in} \cap X_{out} = \{st, h, a\}$ as a set of **state variables**.

Example: Mine-Pump Controller

```
1 : while (true) {
2 :   w := ReadWater();
3 :   m := ReadMethane();

4 :   if (st = Ready) {
5 :     h := 0;
6 :     if (m < 600 ∧ w ≥ 70) {
7 :       st := Pump; }}
```

```
8 :     else if (st = Pump) {
9 :       h := h + 1;
10 :      if (m ≥ 600) {
11 :        st := Locked; a := true; }
12 :      else if (w ≤ 30) {
13 :        st := Ready; }
14 :      else if (h ≥ 3 ∧ w ≤ 50) {
15 :        st := Ready; }}
```

```
16 :    else if (st = Locked) {
17 :      if (h > 0) {
18 :        h := h − 1; }
19 :      if (¬a) {
20 :        st := Pump; }
21 :      if (m ≤ 550) {
22 :        a := false; }}
23 : }
```

- $X_{in} = \{w, m, st, h, a\}$; $X_{out} = \{st', h', a'\}$.
- We write $X_{st} = X_{in} \cap X_{out} = \{st, h, a\}$ as a set of **state variables**.

Example: Mine-Pump Controller

```
1 : while (true) {
2 :   w := ReadWater();
3 :   m := ReadMethane();

4 :   if (st = Ready) {
5 :     h := 0;
6 :     if (m < 600 ∧ w ≥ 70) {
7 :       st := Pump; }}
```

```
8 :     else if (st = Pump) {
9 :       h := h + 1;
10 :      if (m ≥ 600) {
11 :        st := Locked; a := true; }
12 :      else if (w ≤ 30) {
13 :        st := Ready; }
14 :      else if (h ≥ 3 ∧ w ≤ 50) {
15 :        st := Ready; }}
```

```
16 :    else if (st = Locked) {
17 :      if (h > 0) {
18 :        h := h − 1; }
19 :      if (¬a) {
20 :        st := Pump; }
21 :      if (m ≤ 550) {
22 :        a := false; }}
23 : }
```

- $X_{in} = \{w, m, st, h, a\}$; $X_{out} = \{st', h', a'\}$.
- We write $X_{st} = X_{in} \cap X_{out} = \{st, h, a\}$ as a set of **state variables**.

Example: Mine-Pump Controller

```
1 : while (true) {
2 :   w := ReadWater();
3 :   m := ReadMethane();

4 :   if (st = Ready) {
5 :     h := 0;
6 :     if (m < 600 ∧ w ≥ 70) {
7 :       st := Pump; }}
```

```
8 :   else if (st = Pump) {
9 :     h := h + 1;
10 :    if (m ≥ 600) {
11 :      st := Locked; a := true; }
12 :    else if (w ≤ 30) {
13 :      st := Ready; }
14 :    else if (h ≥ 3 ∧ w ≤ 50) {
15 :      st := Ready; }}
```

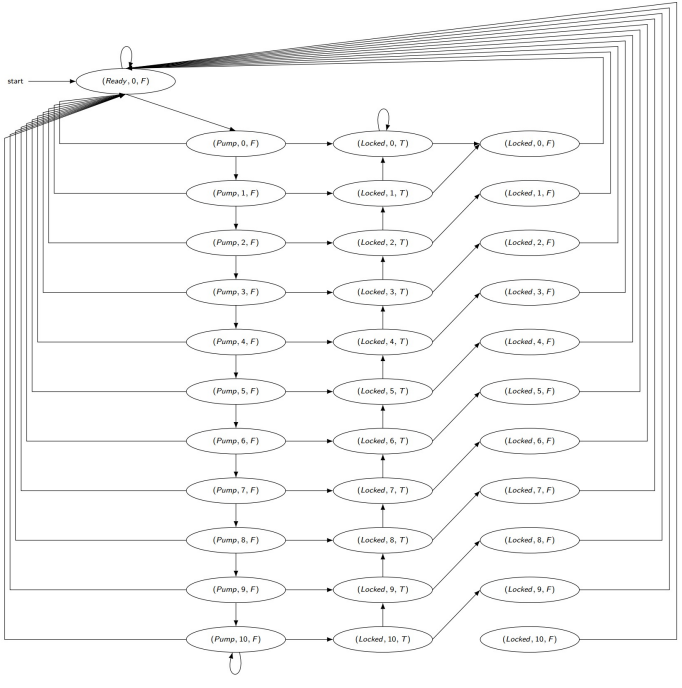
```
16 :  else if (st = Locked) {
17 :    if (h > 0) {
18 :      h := h − 1; }
19 :    if (¬a) {
20 :      st := Pump; }
21 :    if (m ≤ 550) {
22 :      a := false; }}
23 : }
```

- $X_{in} = \{w, m, st, h, a\}$; $X_{out} = \{st', h', a'\}$.
- We write $X_{st} = X_{in} \cap X_{out} = \{st, h, a\}$ as a set of **state variables**.

Goal: To learn automata that capture how reactive programs behave.

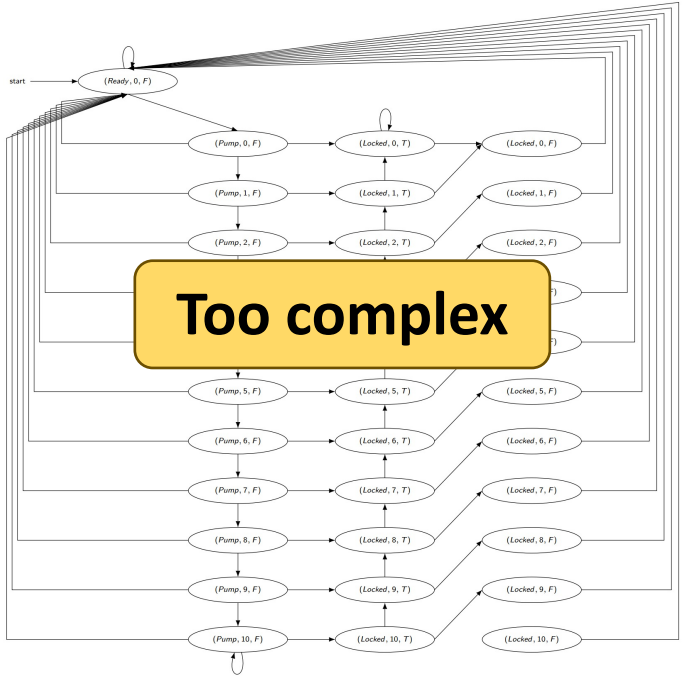
Our Target Automata

Automaton for all state variables (st, h, a)



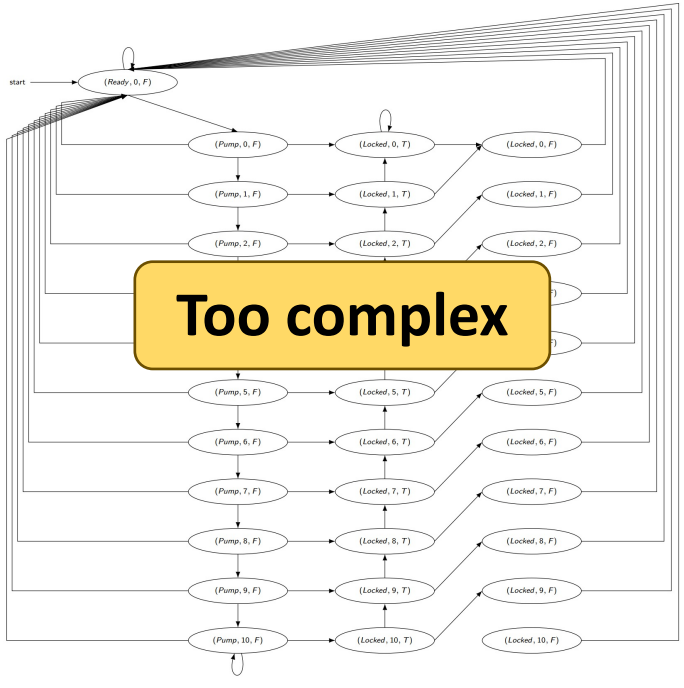
Our Target Automata

Automaton for all state variables (st, h, a)



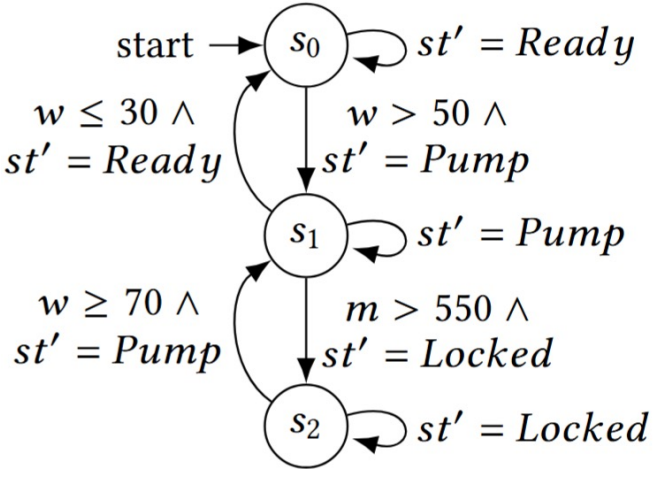
Our Target Automata

Automaton for all state variables (st, h, a)



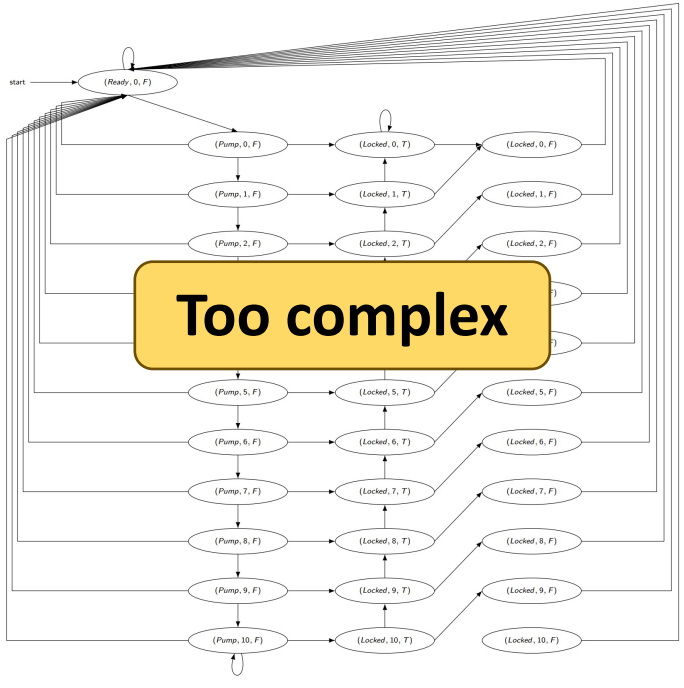
Idea: Focus on only one **primary variable** in the program.

Automaton for the **primary variable** st



Our Target Automata

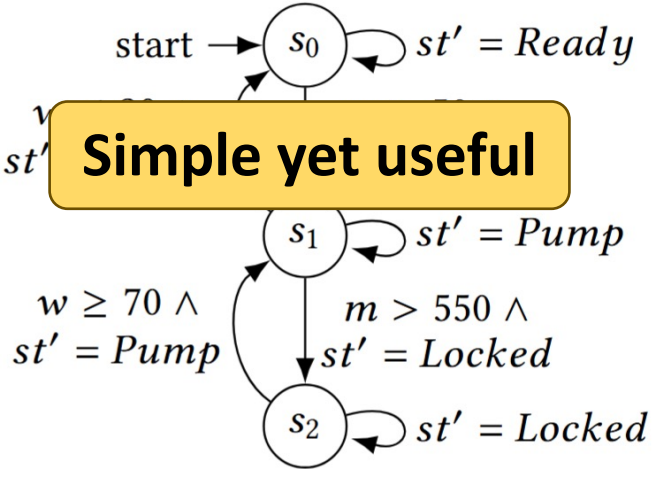
Automaton for all state variables (st, h, a)



Too complex

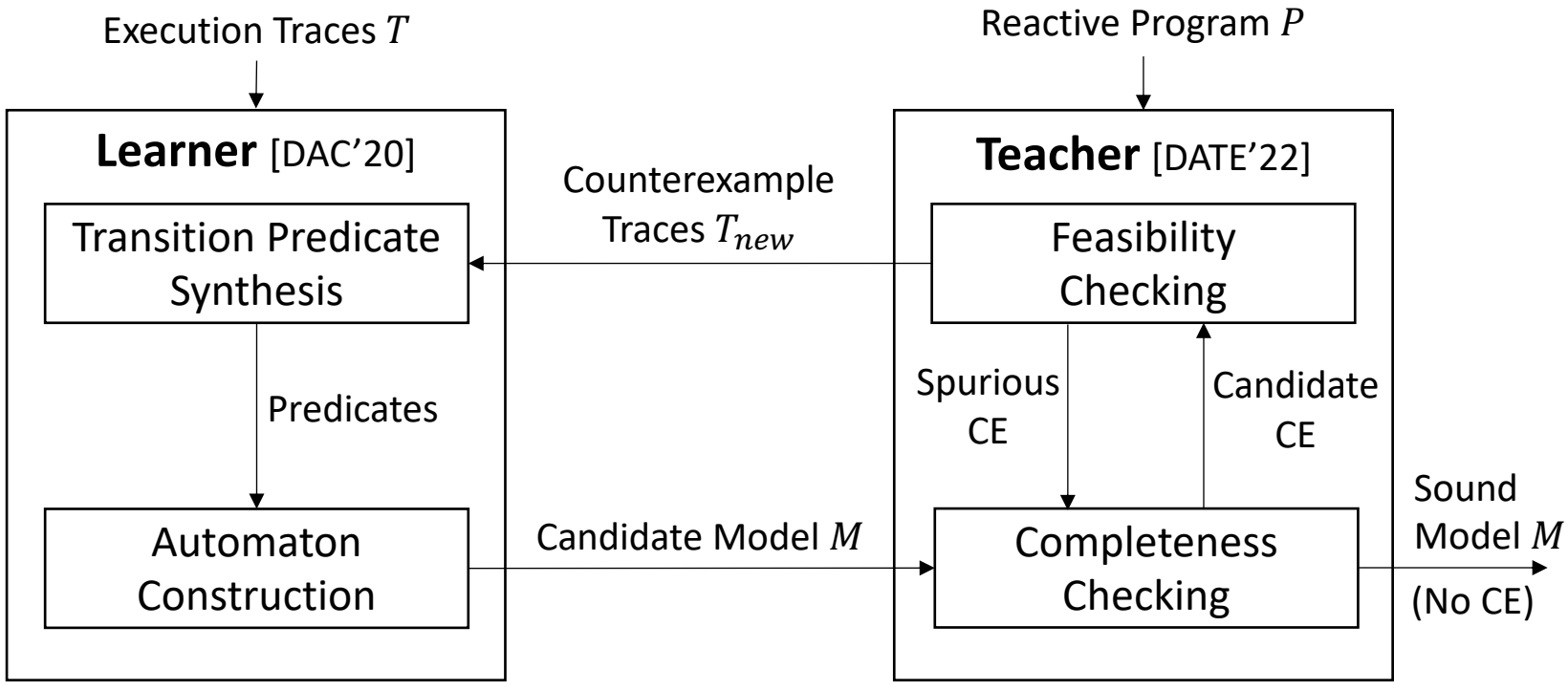
Idea: Focus on only one **primary variable** in the program.

Automaton for the **primary variable** st



Simple yet useful

SOTA: Active Learning with PBE + MC



[DAC'20] Jeppu et al., Learning Concise Models from Long Execution Traces

[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

Learning with Programming-By-Example

Execution Trace T

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	F	0	F	0	*Ready*
2	*Ready*	73	596	F	0	F	0	*Pump*
3	*Pump*	28	548	F	0	F	1	*Ready*
4	*Ready*	75	588	F	1	F	0	*Pump*
5	*Pump*	72	544	F	0	F	1	*Pump*
6	*Pump*	68	610	F	1	T	2	*Locked*
7	*Locked*	67	550	T	2	F	1	*Locked*
8	*Locked*	71	540	F	1	F	0	*Pump*

← Observation

[DAC'20] Jeppu et al., Learning Concise Models from Long Execution Traces

Learning with Programming-By-Example

Execution Trace T

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	F	0	F	0	*Ready*
2	*Ready*	73	596	F	0	F	0	*Pump*
3	*Pump*	28	548	F	0	F	1	*Ready*
4	*Ready*	75	588	F	1	F	0	*Pump*
5	*Pump*	72	544	F	0	F	1	*Pump*
6	*Pump*	68	610	F	1	T	2	*Locked*
7	*Locked*	67	550	T	2	F	1	*Locked*
8	*Locked*	71	540	F	1	F	0	*Pump*

1. **Group** observations by the current value of the primary variable st.

[DAC'20] Jeppu et al., Learning Concise Models from Long Execution Traces

Learning with Programming-By-Example

Execution Trace T

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	F	0	F	0	*Ready*
2	*Ready*	73	596	F	0	F	0	*Pump*
3	*Pump*	28	548	F	0	F	1	*Ready*
4	*Ready*	75	588	F	1	F	0	*Pump*
5	*Pump*	72	544	F	0	F	1	*Pump*
6	*Pump*	68	610	F	1	T	2	*Locked*
7	*Locked*	67	550	T	2	F	1	*Locked*
8	*Locked*	71	540	F	1	F	0	*Pump*

1. **Group** observations by the current value of the primary variable st.

2. **Synthesize** predicates for each observation group.

$G_{Ready} = \{o_1, o_2, o_4\}$
$w = 31 \rightarrow Ready$
$w = 73{,}75 \rightarrow Pump$
$Ready \rightarrow Pump: \boldsymbol{w > 50}$

$G_{Pump} = \{o_3, o_5, o_6\}$
$w = 28, m = 548 \rightarrow Ready$
$w = 68, m = 610 \rightarrow Locked$
$Pump \rightarrow Ready: \boldsymbol{w \leq 30}$
$Pump \rightarrow Locked: \boldsymbol{m > 550}$

$G_{Locked} = \{o_7, o_8\}$
$w = 67 \rightarrow Locked$
$w = 71 \rightarrow Pump$
$Locked \rightarrow Pump: \boldsymbol{w \geq 70}$

[DAC'20] Jeppu et al., Learning Concise Models from Long Execution Traces

Learning with Programming-By-Example

Execution Trace T

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	F	0	F	0	*Ready*
2	*Ready*	73	596	F	0	F	0	*Pump*
3	*Pump*	28	548	F	0	F	1	*Ready*
4	*Ready*	75	588	F	1	F	0	*Pump*
5	*Pump*	72	544	F	0	F	1	*Pump*
6	*Pump*	68	610	F	1	T	2	*Locked*
7	*Locked*	67	550	T	2	F	1	*Locked*
8	*Locked*	71	540	F	1	F	0	*Pump*

1. **Group** observations by the current value of the primary variable st.

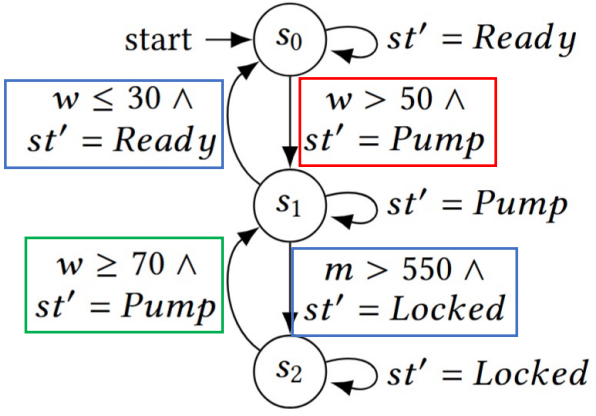
2. **Synthesize** predicates for each observation group.

$G_{Ready} = \{o_1, o_2, o_4\}$
$w = 31 \rightarrow Ready$
$w = 73{,}75 \rightarrow Pump$
$Ready \rightarrow Pump$: $\boldsymbol{w > 50}$

$G_{Pump} = \{o_3, o_5, o_6\}$
$w = 28, m = 548 \rightarrow Ready$
$w = 68, m = 610 \rightarrow Locked$
$Pump \rightarrow Ready$: $\boldsymbol{w \leq 30}$
$Pump \rightarrow Locked$: $\boldsymbol{m > 550}$

$G_{Locked} = \{o_7, o_8\}$
$w = 67 \rightarrow Locked$
$w = 71 \rightarrow Pump$
$Locked \rightarrow Pump$: $\boldsymbol{w \geq 70}$

3. **Learn** an automaton M.



[DAC'20] Jeppu et al., Learning Concise Models from Long Execution Traces

Teaching with Model Checking

1. Completeness Checking

Check: $L(P) \subseteq L(M)$?

Does the automaton M miss any possible program behavior?

Checking code:

```
assume(ψ); // assume prev. state
X' := f(X); // run one iteration
assert(ϕ);  // check next state
```

No violation ⟹ Sound model

Violation ⟹ Candidate CE τ

2. Feasibility Checking

Check: $\exists \pi\ s.t.\ \pi\tau \in L(P)$?

Does the program P have a feasible prefix π that reaches τ?

Checking code:

```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬τ); }
```

Violation ⟹ Feasible CE τ

No violation ⟹ Assume($\neg\tau$)

[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

Teaching with Model Checking

1. Completeness Checking

Check: $L(P) \subseteq L(M)$?

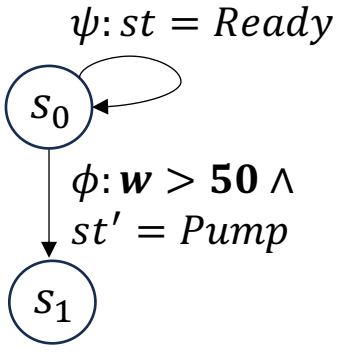
Does the automaton M miss any possible program behavior?

Checking code:

```
assume(ψ); // assume prev. state
X' := f(X); // run one iteration
assert(ϕ);  // check next state
```

No violation ⟹ Sound model

Violation ⟹ Candidate CE τ



Transition Pair

2. Feasibility Checking

Check: $\exists \pi\ s.t.\ \pi\tau \in L(P)$?

Does the program P have a feasible prefix π that reaches τ?

Checking code:

```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬τ); }
```

Violation ⟹ Feasible CE τ

No violation ⟹ Assume($\neg\tau$)

[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

Teaching with Model Checking

1. Completeness Checking

Check: $L(P) \subseteq L(M)$?

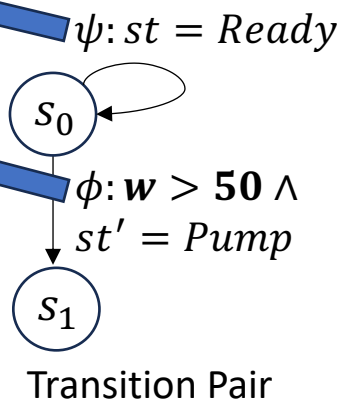
Does the automaton M miss any possible program behavior?

Checking code:

```
assume(ψ); // assume prev. state
X' := f(X); // run one iteration
assert(ϕ);  // check next state
```

No violation ⟹ Sound model

Violation ⟹ Candidate CE τ



2. Feasibility Checking

Check: $\exists \pi\ s.t.\ \pi\tau \in L(P)$?

Does the program P have a feasible prefix π that reaches τ?

Checking code:

```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬τ); }
```

Violation ⟹ Feasible CE τ

No violation ⟹ Assume($\neg\tau$)

[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

Teaching with Model Checking

1. Completeness Checking

Check: $L(P) \subseteq L(M)$?

Does the automaton M miss any possible program behavior?

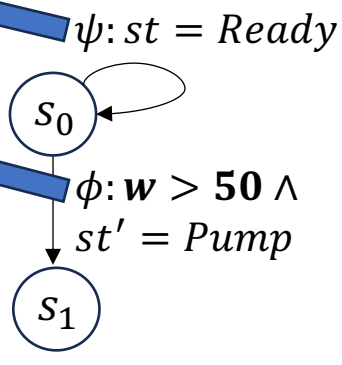
Checking code:

```
assume(ψ); // assume prev. state
X' := f(X); // run one iteration
assert(ϕ);  // check next state
```

No violation ⟹ Sound model

Violation ⟹ Candidate CE τ

Candidate CE $\tau :=$
$(st = Ready \wedge$
$\boldsymbol{w = 49} \wedge$
$m = 500 \wedge$
$a = true \wedge$
$h = 0 \wedge$
$st' = Pump)$



Transition Pair

2. Feasibility Checking

Check: $\exists \pi\ s.t.\ \pi\tau \in L(P)$?

Does the program P have a feasible prefix π that reaches τ?

Checking code:

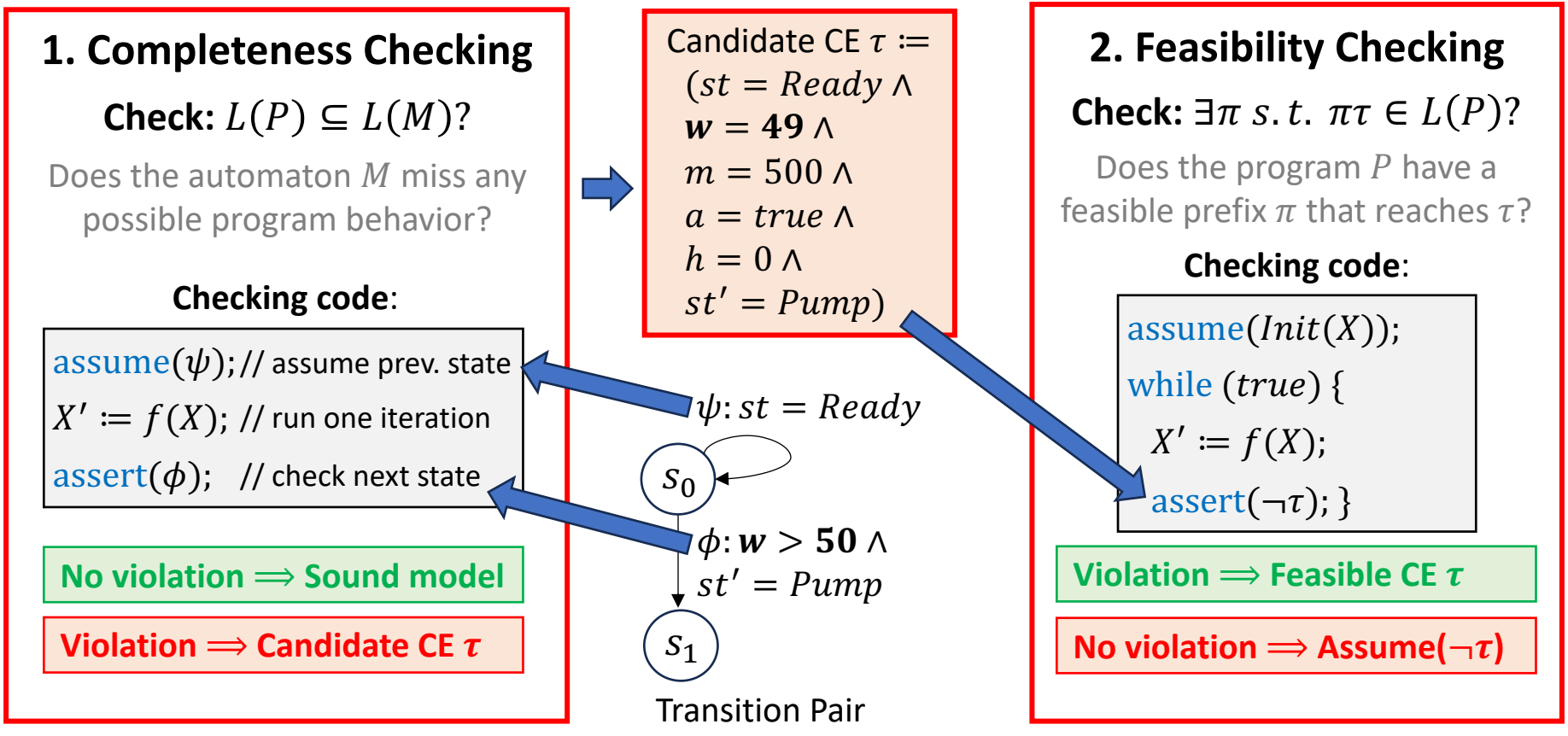
```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬τ); }
```

Violation ⟹ Feasible CE τ

No violation ⟹ Assume($\neg\tau$)

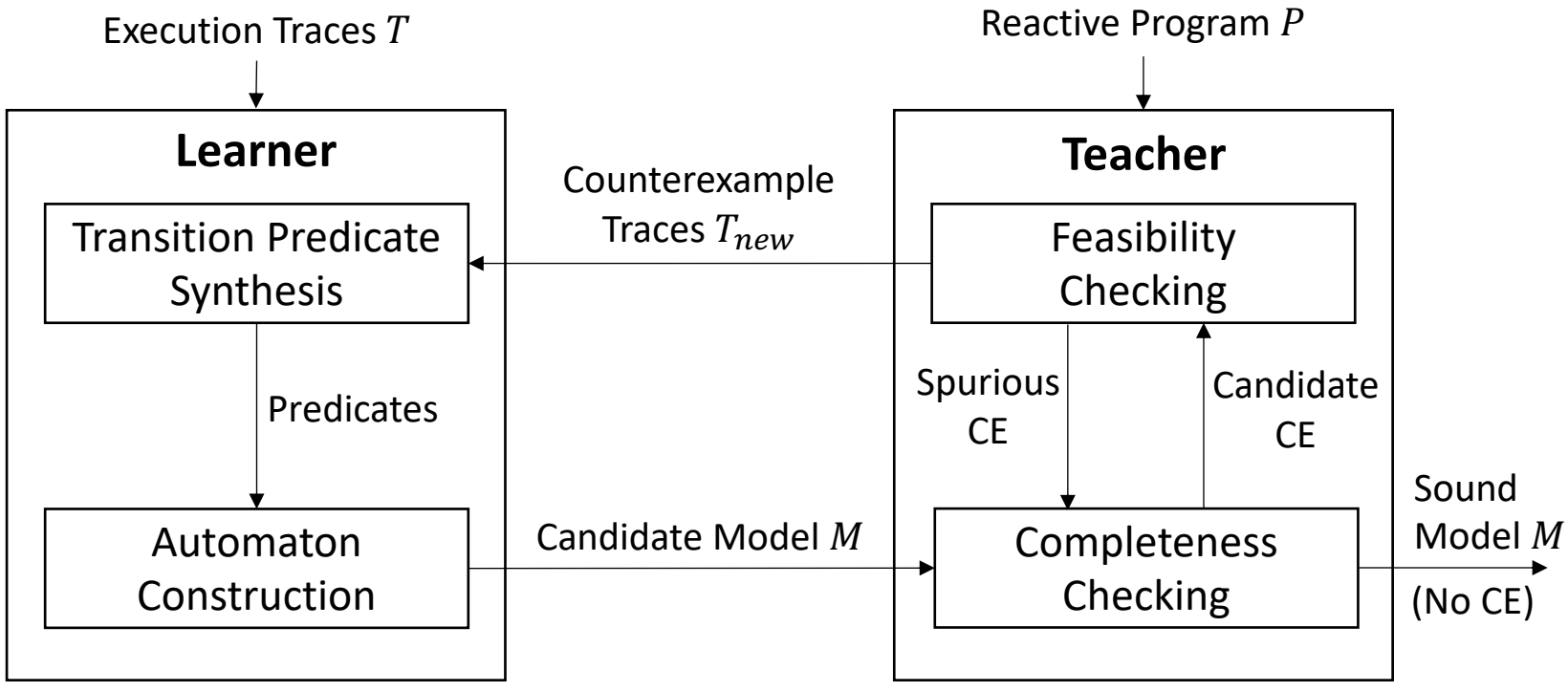
[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

Teaching with Model Checking



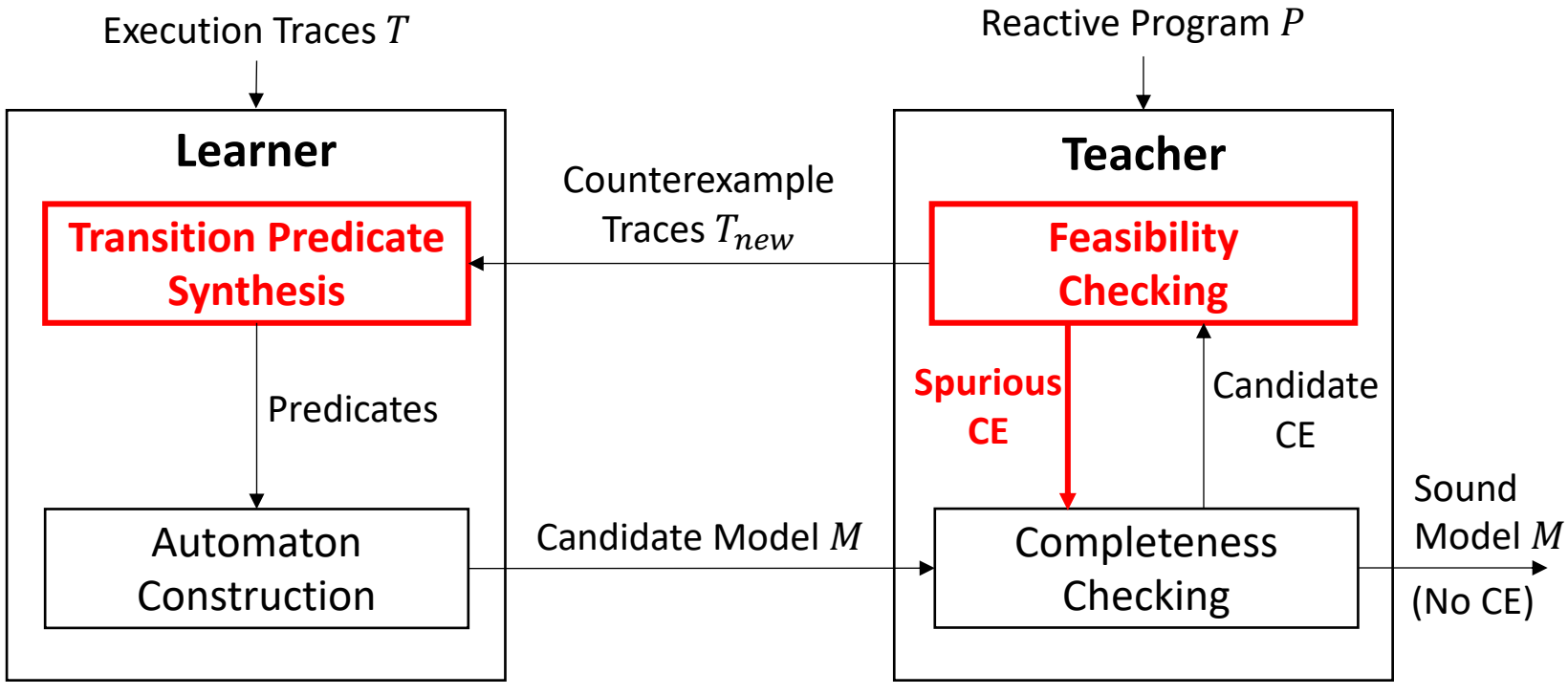
[DATE'22] Jeppu et al., Active Learning of Abstract System Models from Traces using Model Checking

SOTA: Active Learning with PBE + MC



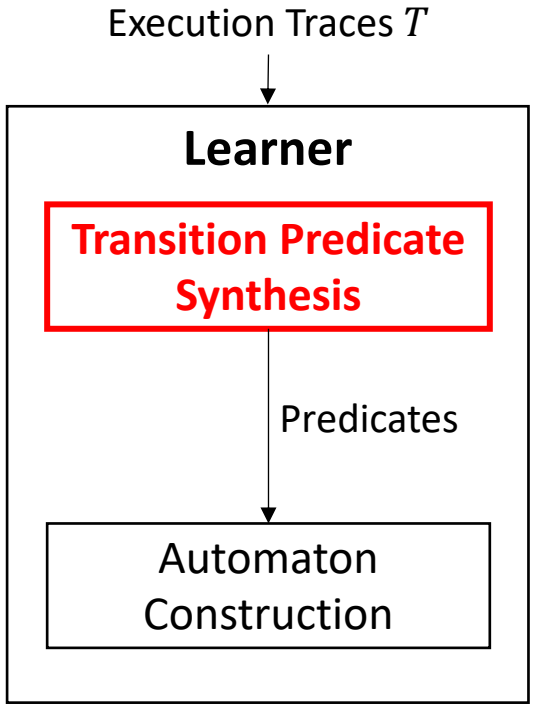
Programming-By-Example + Model Checking:
High model expressiveness and soundness guarantees!

SOTA: Active Learning with PBE + MC

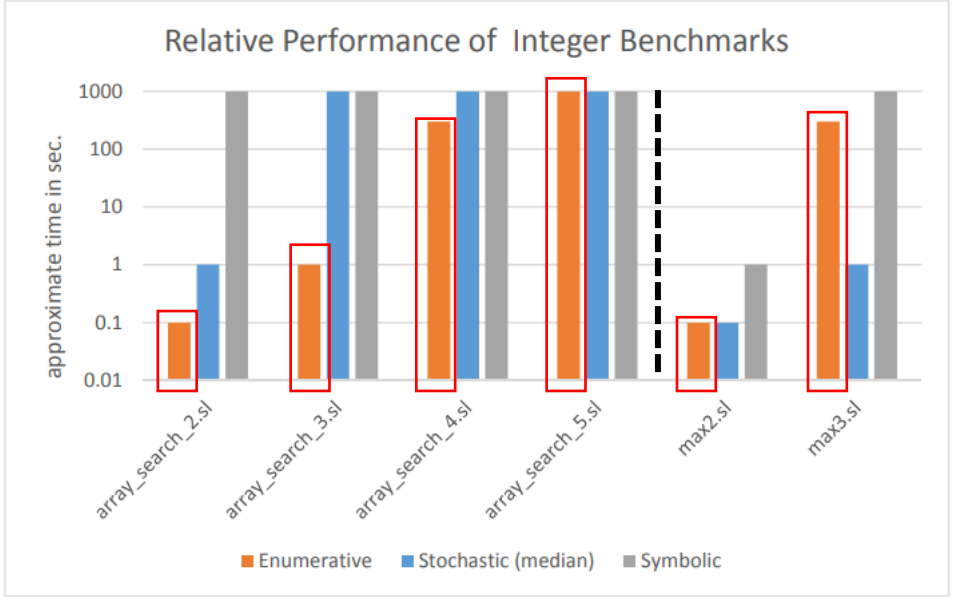


Programming-By-Example + Model Checking:
Performance bottlenecks…

Bottleneck 1: Predicate Synthesis

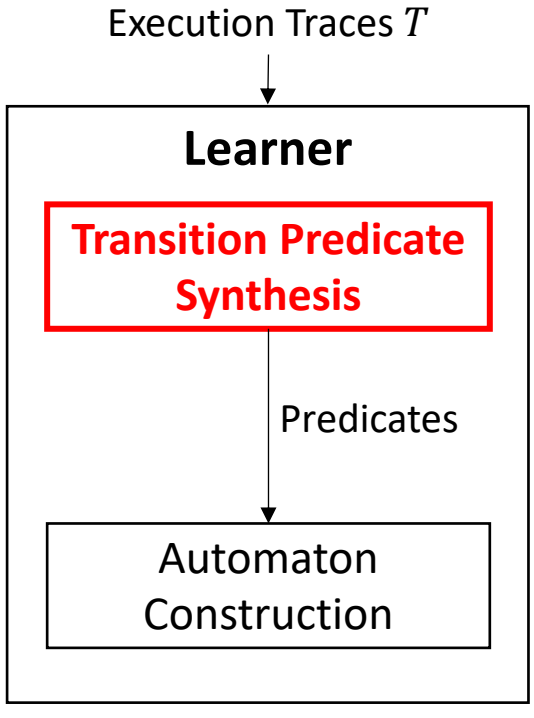


[FMCAD'13] Synthesis performance vs. # of variables.

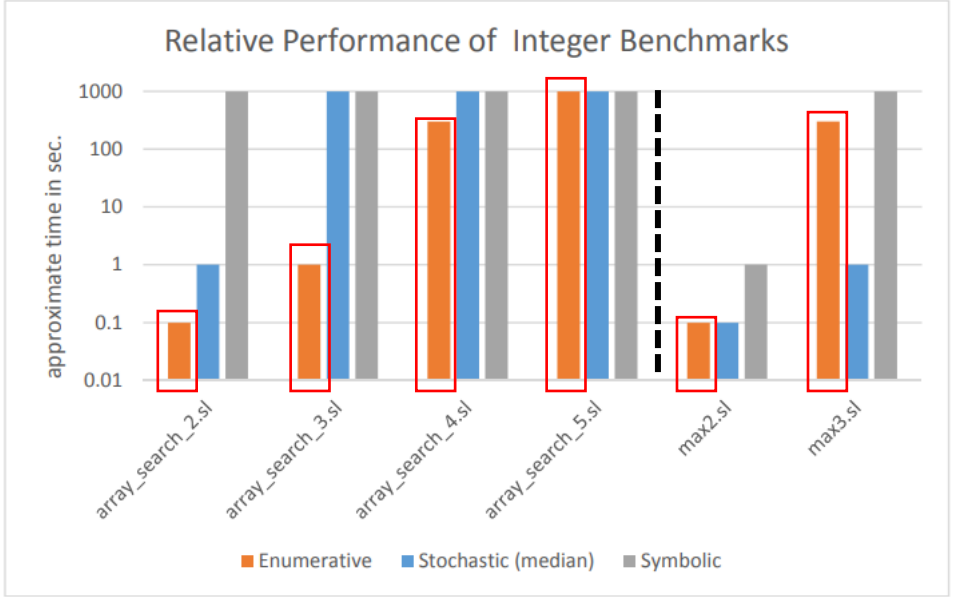


[FMCAD'13] Alur et al., Syntax-Guided Synthesis

Bottleneck 1: Predicate Synthesis



[FMCAD'13] Synthesis performance vs. # of variables.



Performance degrades as the number of variables increases, especially with large domains.

[FMCAD'13] Alur et al., Syntax-Guided Synthe

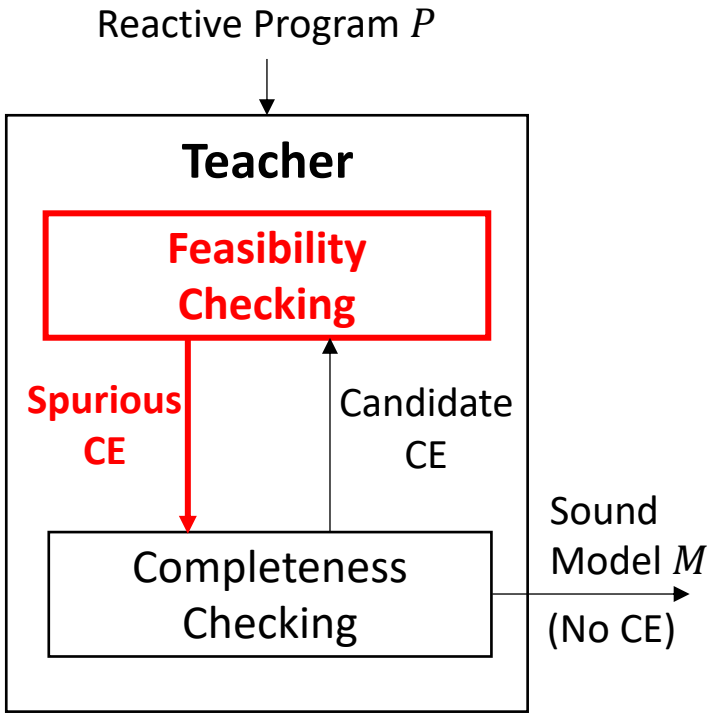
Bottleneck 2: Feasibility Checking

Feasibility checking code

```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬(st = Ready ∧ w = 49 ∧
  m = 500 ∧ a ∧ h = 0 ∧ st' = Pump));
}
```

The same candidate CE τ in the previous slide.

This candidate CE is spurious because the variable a cannot be $true$ when $st = Ready$.



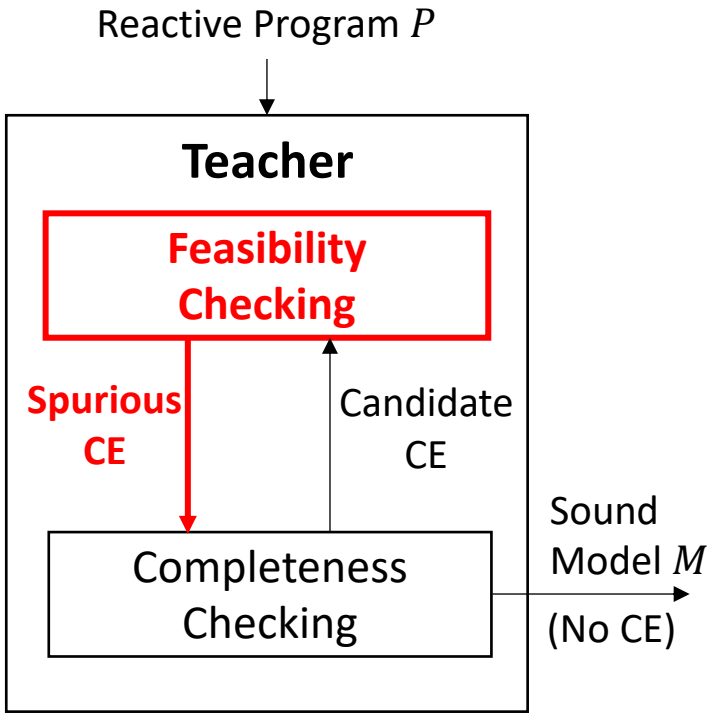
Bottleneck 2: Feasibility Checking

Feasibility checking code

```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬(st = Ready ∧ w = 49 ∧
  m = 500 ∧ a ∧ h = 0 ∧ st' = Pump));
}
```

The same candidate CE τ in the previous slide.

This candidate CE is spurious because the variable a cannot be $true$ when $st = Ready$.



Many variables with large domains generate too many spurious counterexamples satisfying ($st = Ready \wedge a = true$).

Key Idea: Boolean Abstraction

Problem:

- Many variables with large domains.

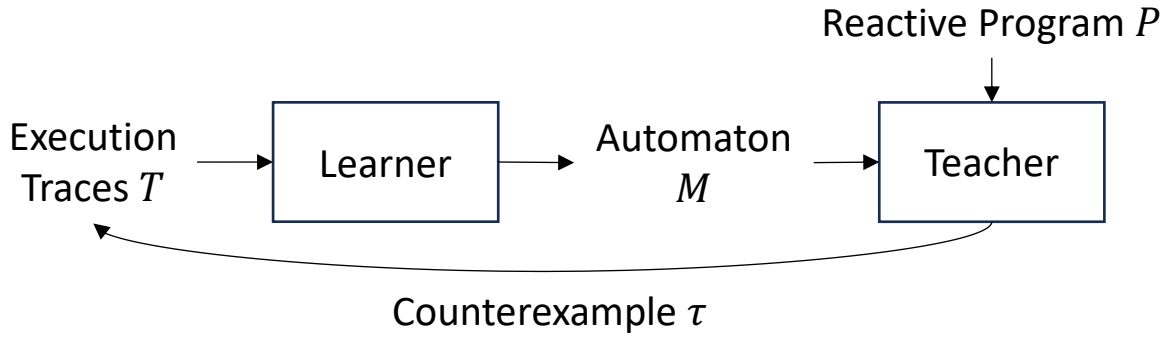
Goal:

- Reduce the number of variables and their value domains.

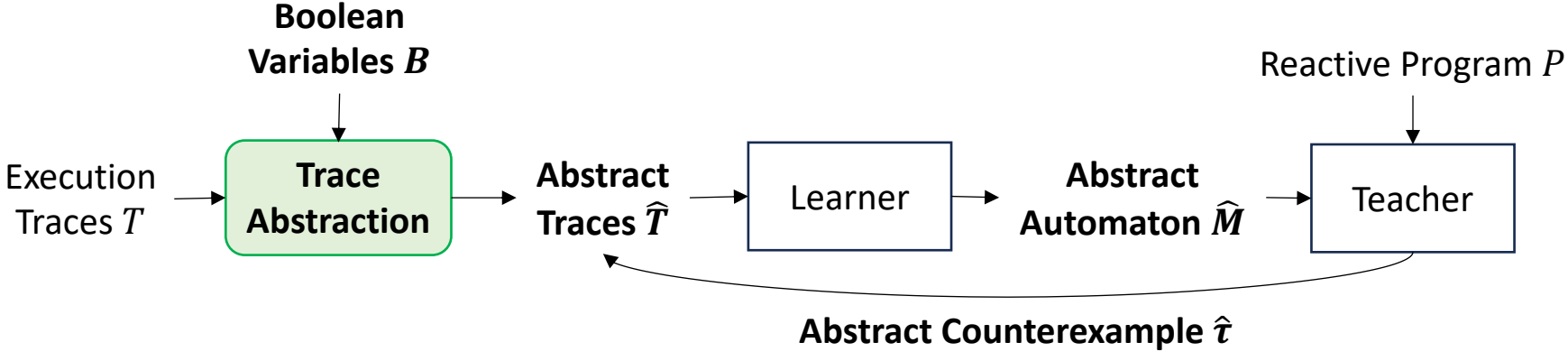
Idea:

- **Boolean abstraction** in active automata learning.

High-Level Process



Baseline



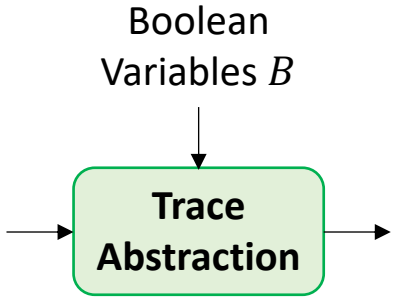
Ours

Trace Abstraction

- We are given a **Boolean variable** set $B = \{b_1, b_2, b_3, b_4\}$ where $b_1 : (w \leq 30), b_2 : (w < 50), b_3 : (w \geq 70), b_4 : (m \leq 550)$.
- Each observation is mapped into a corresponding **Boolean vector**.

Execution Trace T

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	F	0	F	0	*Ready*
2	*Ready*	73	596	F	0	F	0	*Pump*
3	*Pump*	28	548	F	0	F	1	*Ready*
4	*Ready*	75	588	F	1	F	0	*Pump*
5	*Pump*	72	544	F	0	F	1	*Pump*
6	*Pump*	68	610	F	1	T	2	*Locked*
7	*Locked*	67	550	T	2	F	1	*Locked*
8	*Locked*	71	540	F	1	F	0	*Pump*



Abstract Trace $\hat{T}$

i	st	b_1	b_2	b_3	b_4	st'
1	*Ready*	F	T	F	T	*Ready*
2	*Ready*	F	F	T	F	*Pump*
3	*Pump*	T	T	F	T	*Ready*
4	*Ready*	F	F	T	F	*Pump*
5	*Pump*	F	F	T	T	*Pump*
6	*Pump*	F	F	F	F	*Locked*
7	*Locked*	F	F	F	T	*Locked*
8	*Locked*	F	F	T	T	*Pump*

Boolean vectors

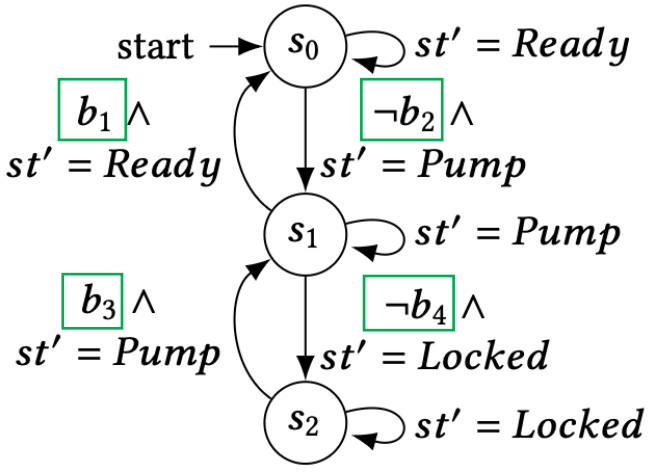
Learning from Abstract Traces

Abstract Trace $\hat{T}$

i	st	b_1	b_2	b_3	b_4	st'
1	*Ready*	F	T	F	T	*Ready*
2	*Ready*	F	F	T	F	*Pump*
3	*Pump*	T	T	F	T	*Ready*
4	*Ready*	F	F	T	F	*Pump*
5	*Pump*	F	F	T	T	*Pump*
6	*Pump*	F	F	F	F	*Locked*
7	*Locked*	F	F	F	T	*Locked*
8	*Locked*	F	F	T	T	*Pump*

Learner

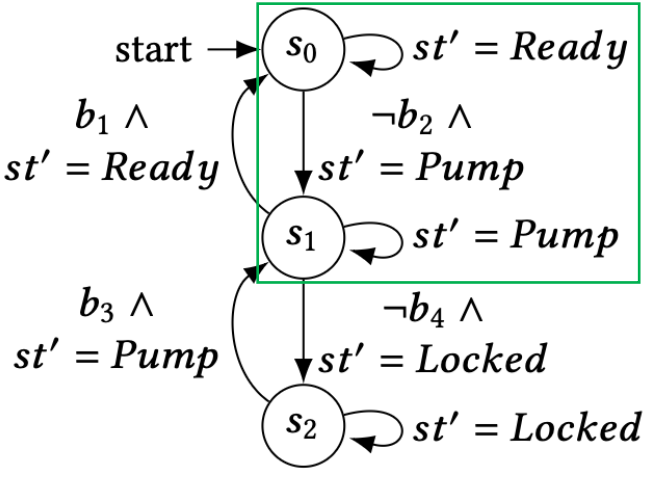
Abstract Automaton $\widehat{M}$



Learner can synthesize predicates considering only logical operations.

Teaching Abstract Automaton

Abstract Automaton $\widehat{M}$



Teacher

Completeness Checking Code for $\widehat{M}$

```
assume(st = Ready);
X' := f(X);
assert(¬b2 ∧ st' = Pump);
```

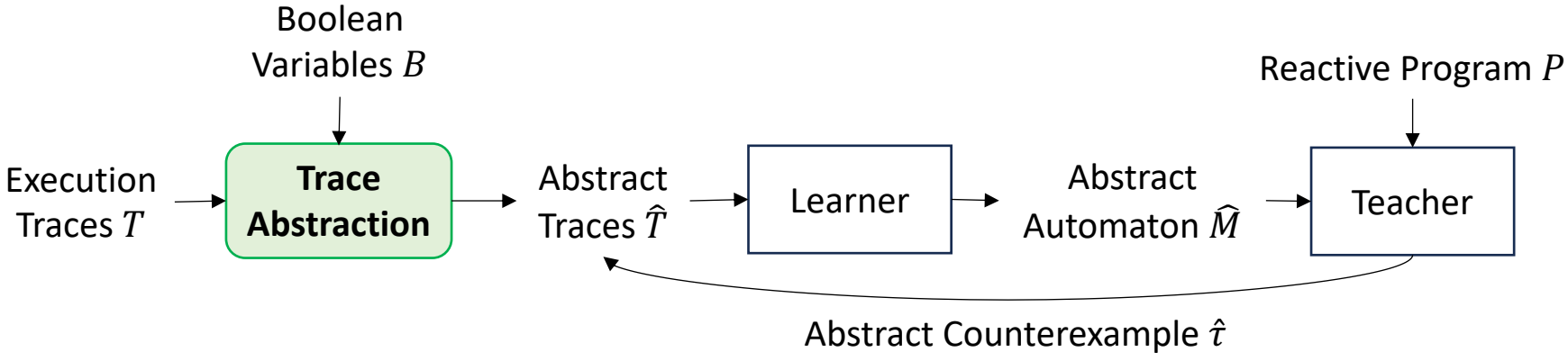
Abstract Candidate CE $\hat{\tau} := (st = Ready \wedge \neg b_1 \wedge \boldsymbol{b_2} \wedge \neg b_3 \wedge b_4 \wedge st' = Pump)$;

Feasibility Checking Code for $\widehat{M}$

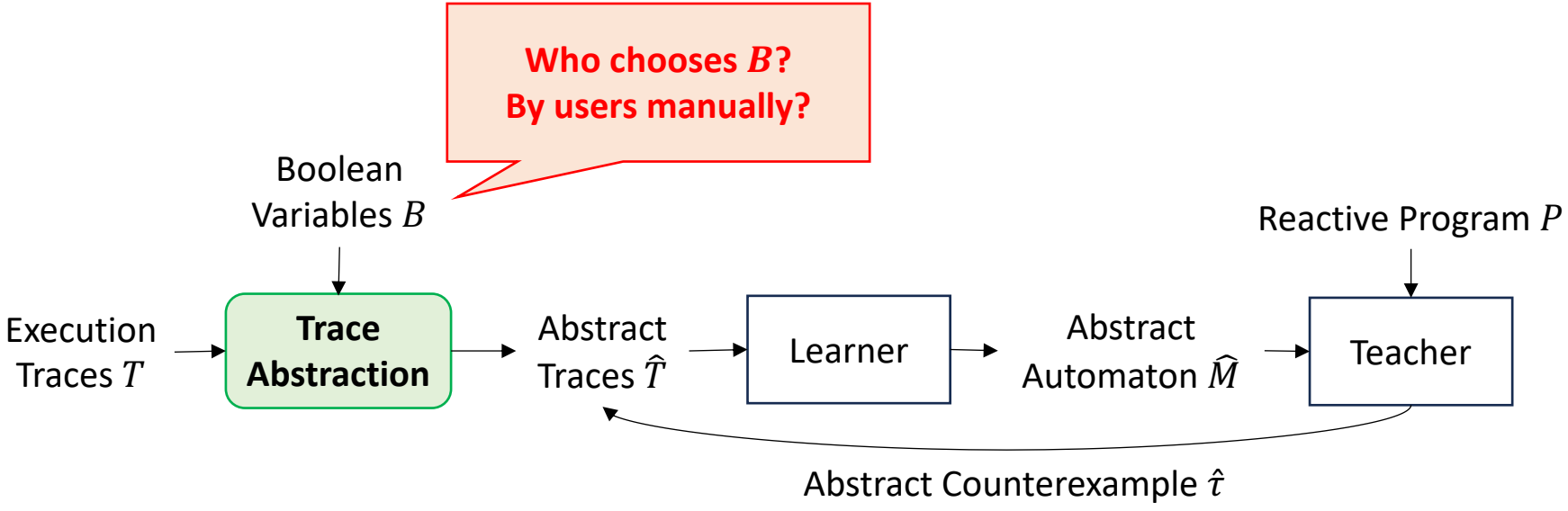
```
assume(Init(X));
while (true) {
  X' := f(X);
  assert(¬(st = Ready ∧ ¬b1 ∧ b2 ∧ ¬b3 ∧ b4 ∧ st' = Pump);}
```

Teacher can find a feasible counterexample within $2^{|B|}$ feasibility checks.

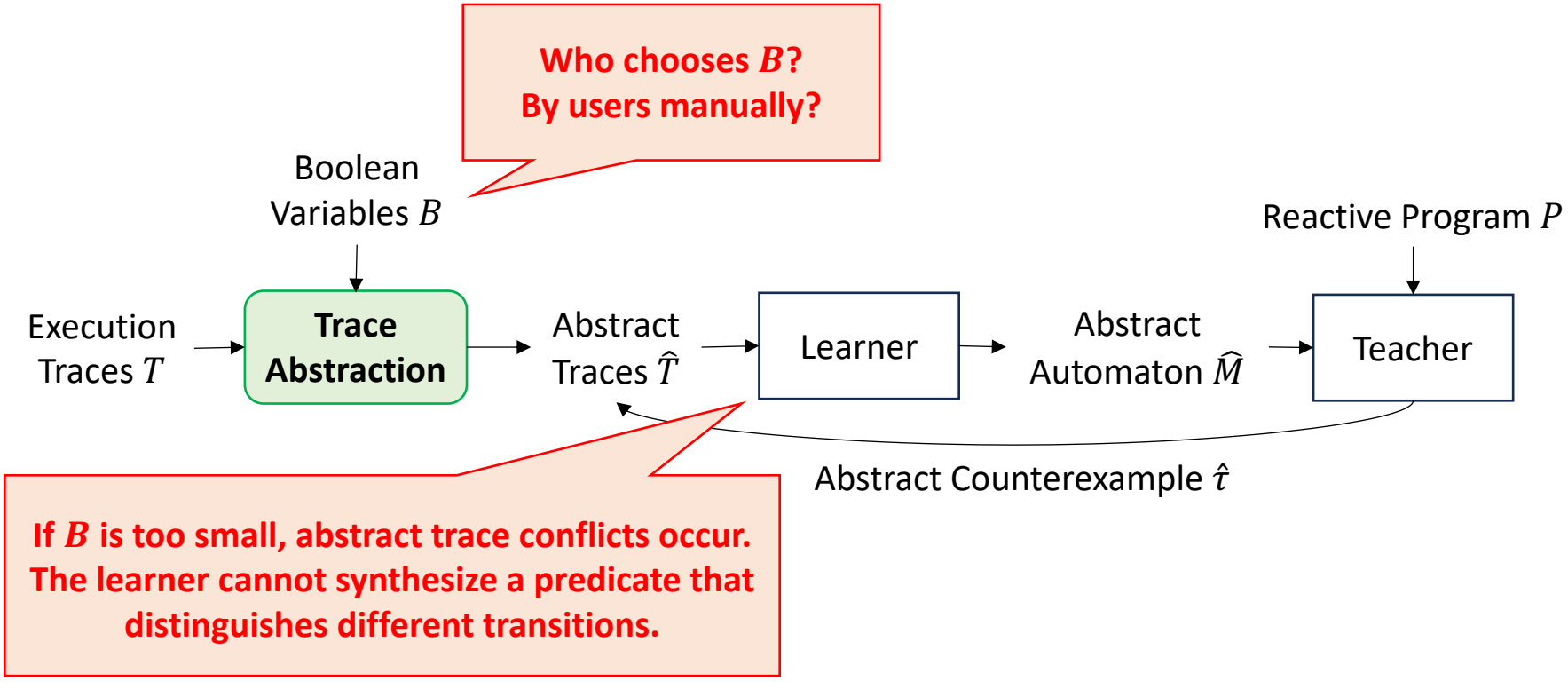
Problems of Boolean Abstraction



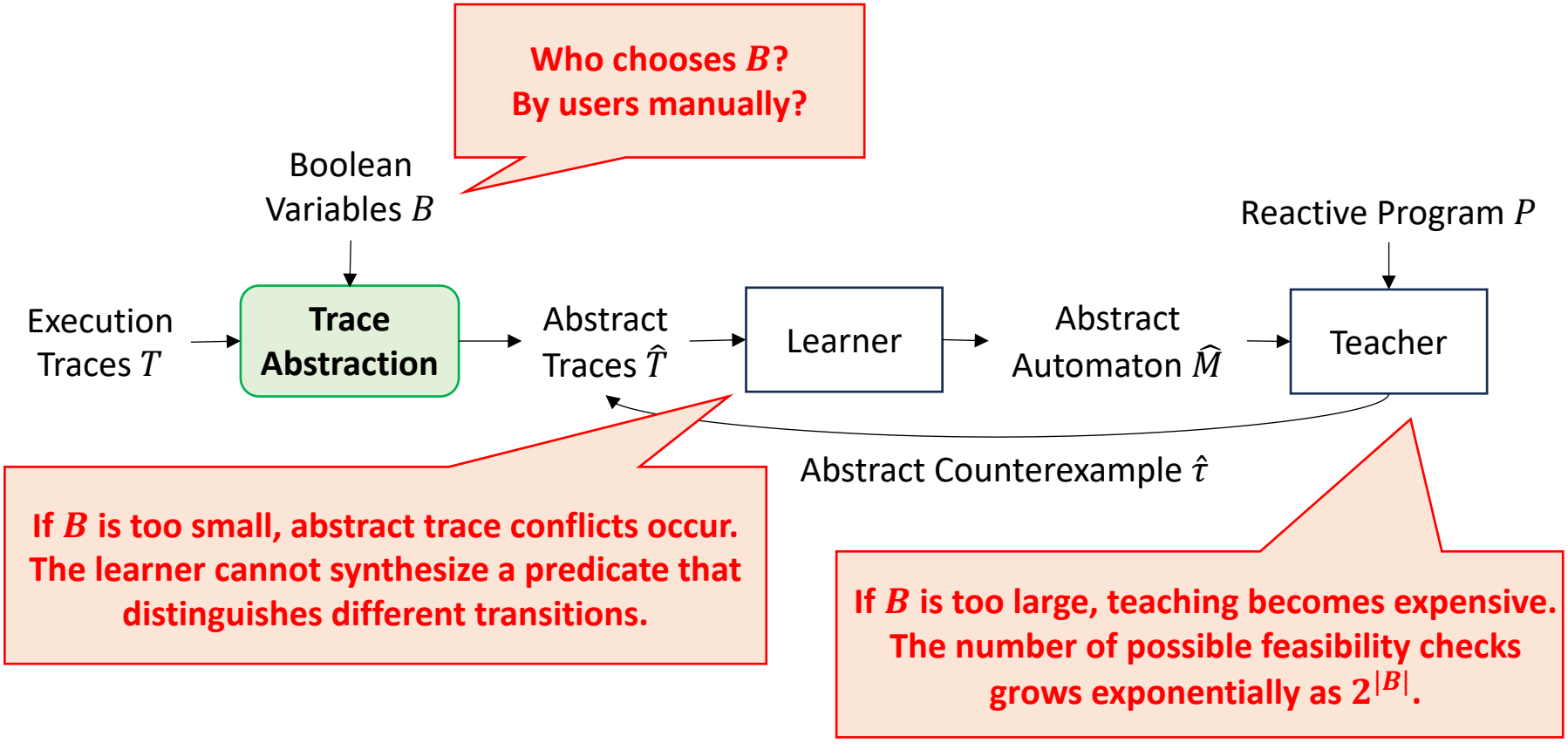
Problems of Boolean Abstraction



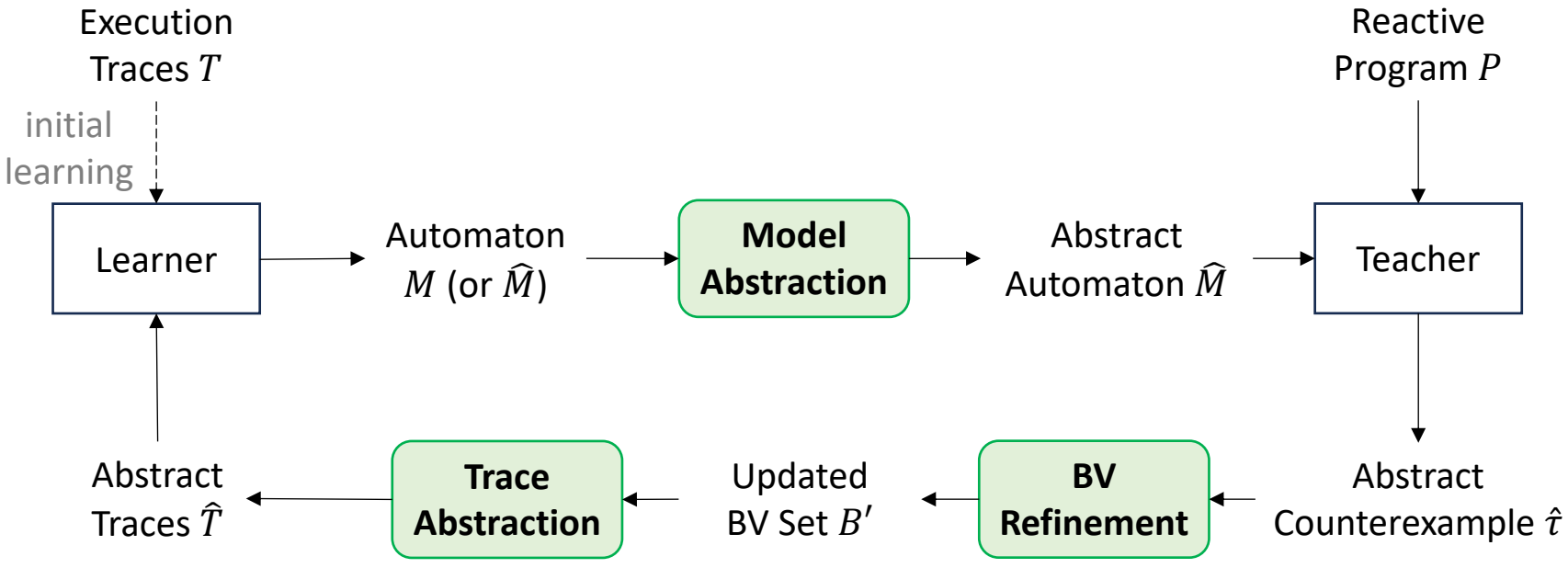
Problems of Boolean Abstraction



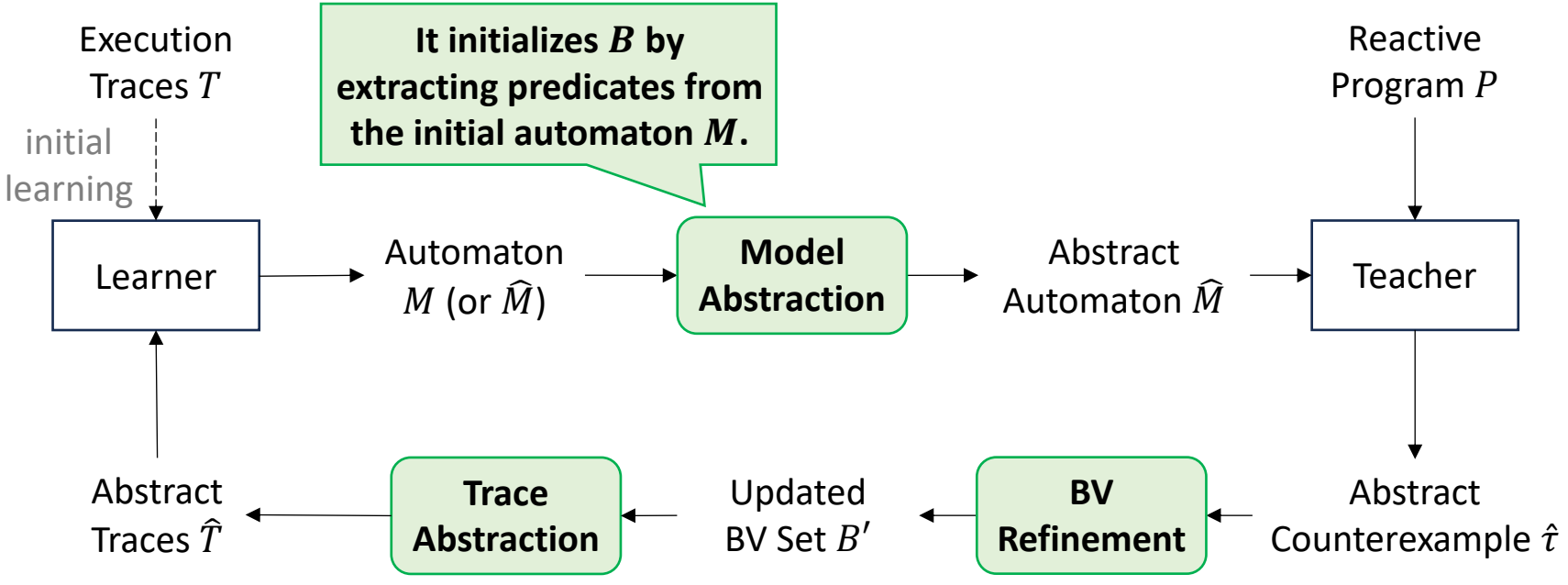
Problems of Boolean Abstraction



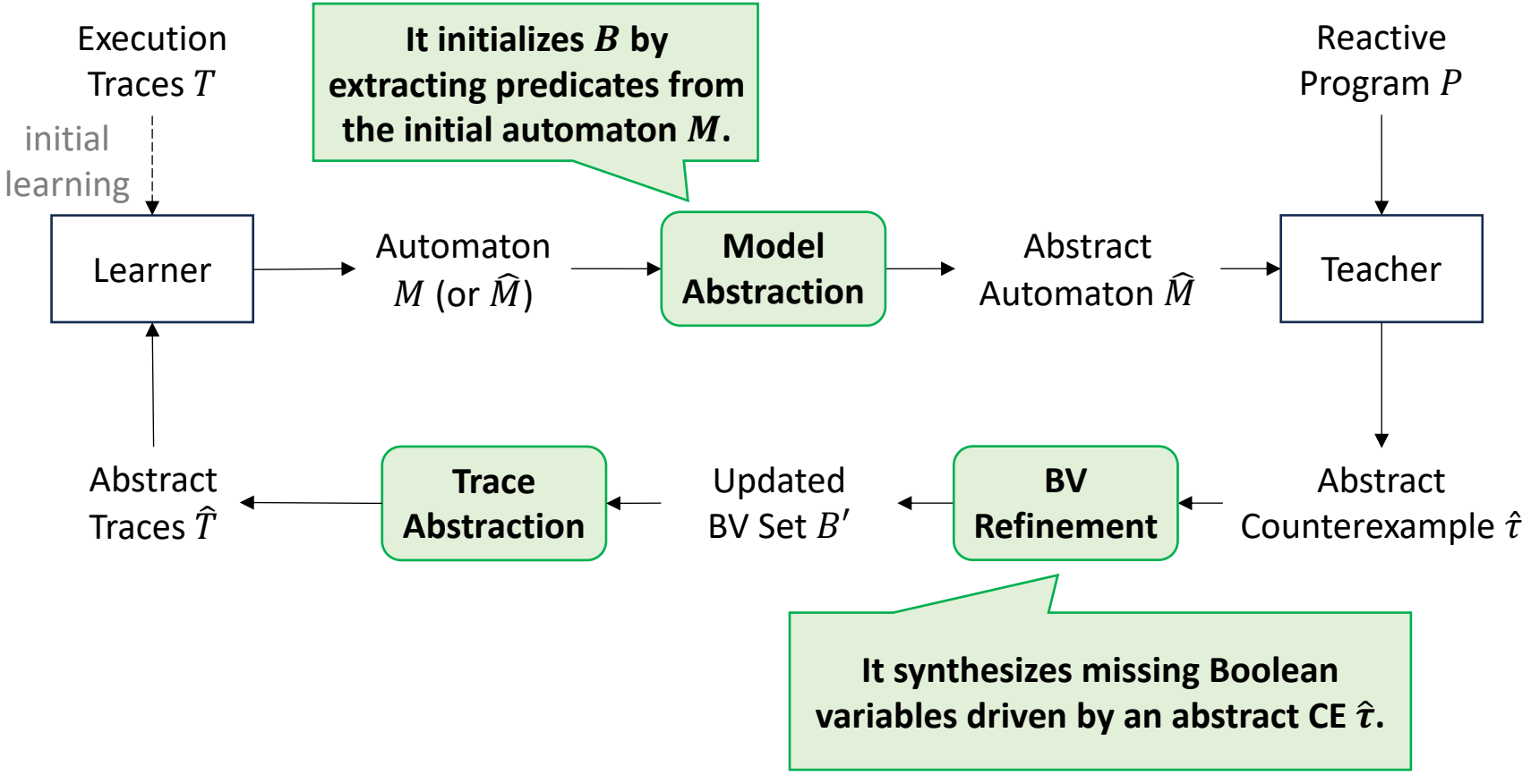
Solution: Dynamic Symbolic Mapper



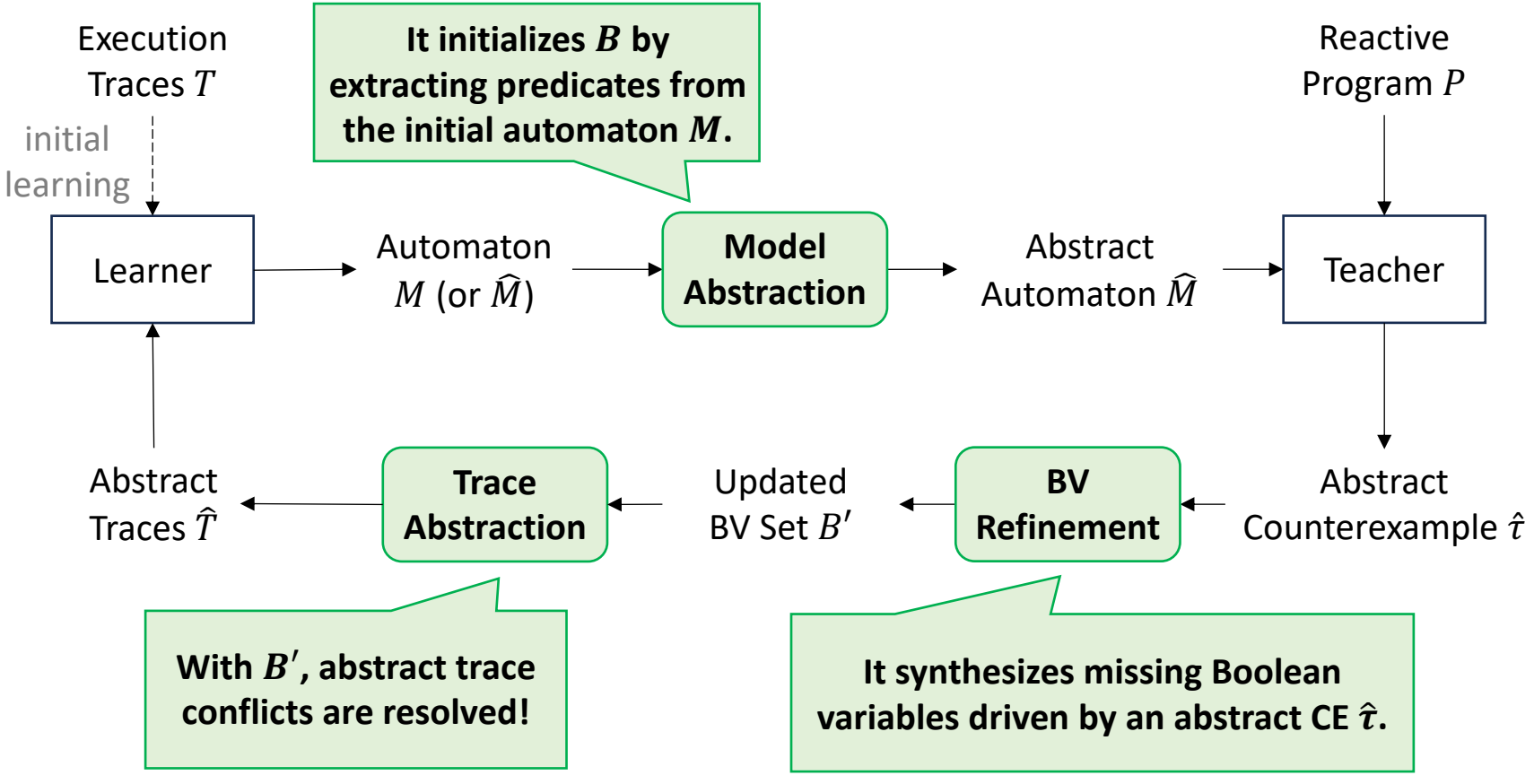
Solution: Dynamic Symbolic Mapper



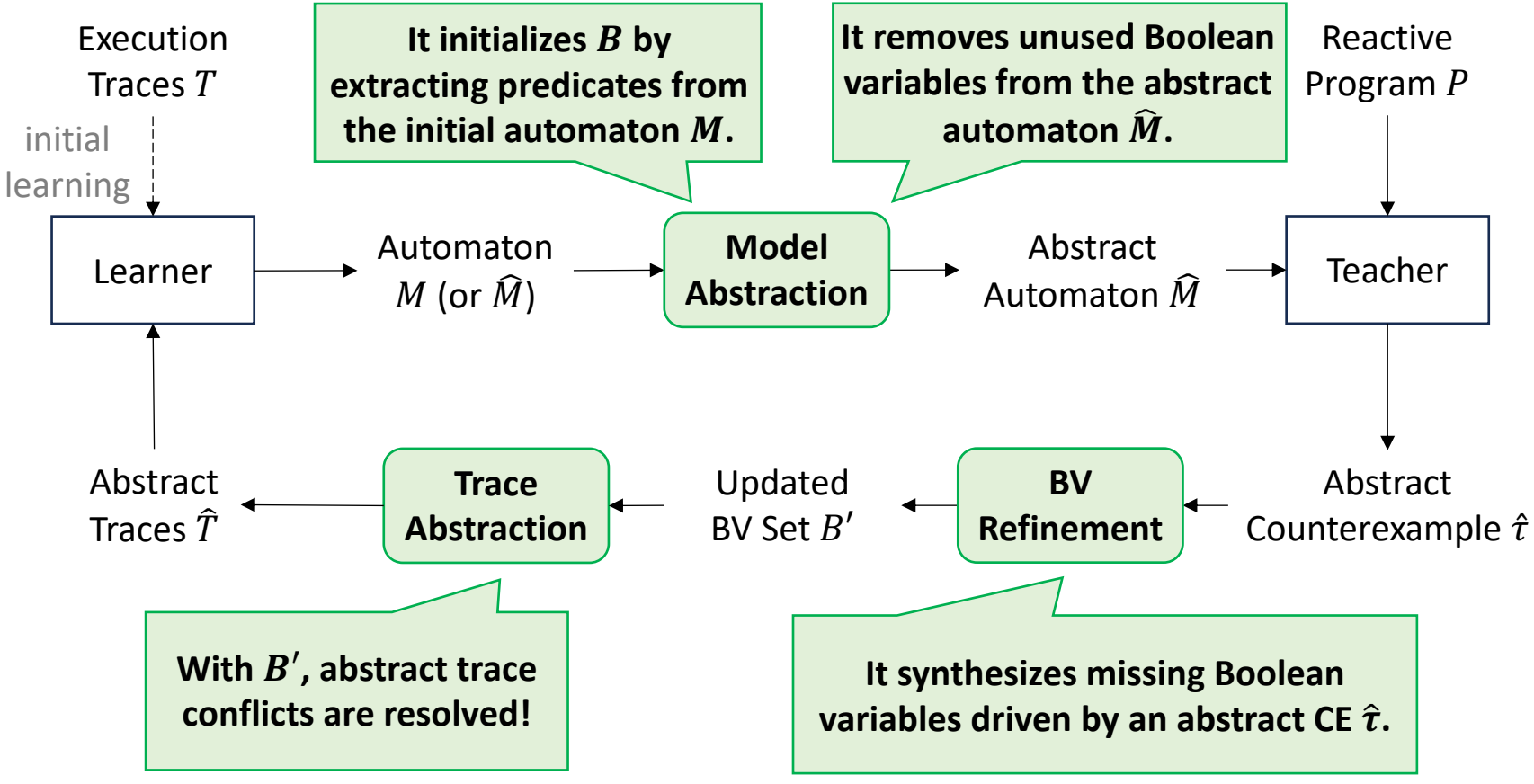
Solution: Dynamic Symbolic Mapper



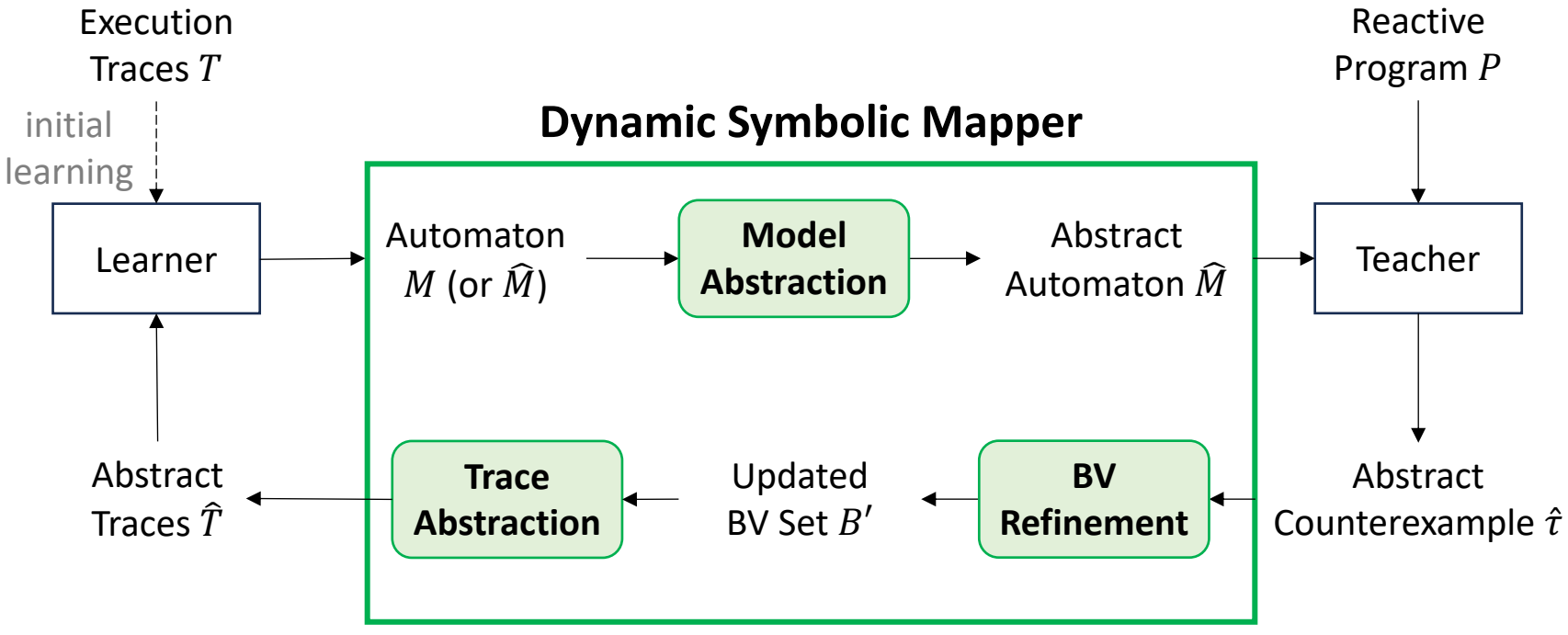
Solution: Dynamic Symbolic Mapper



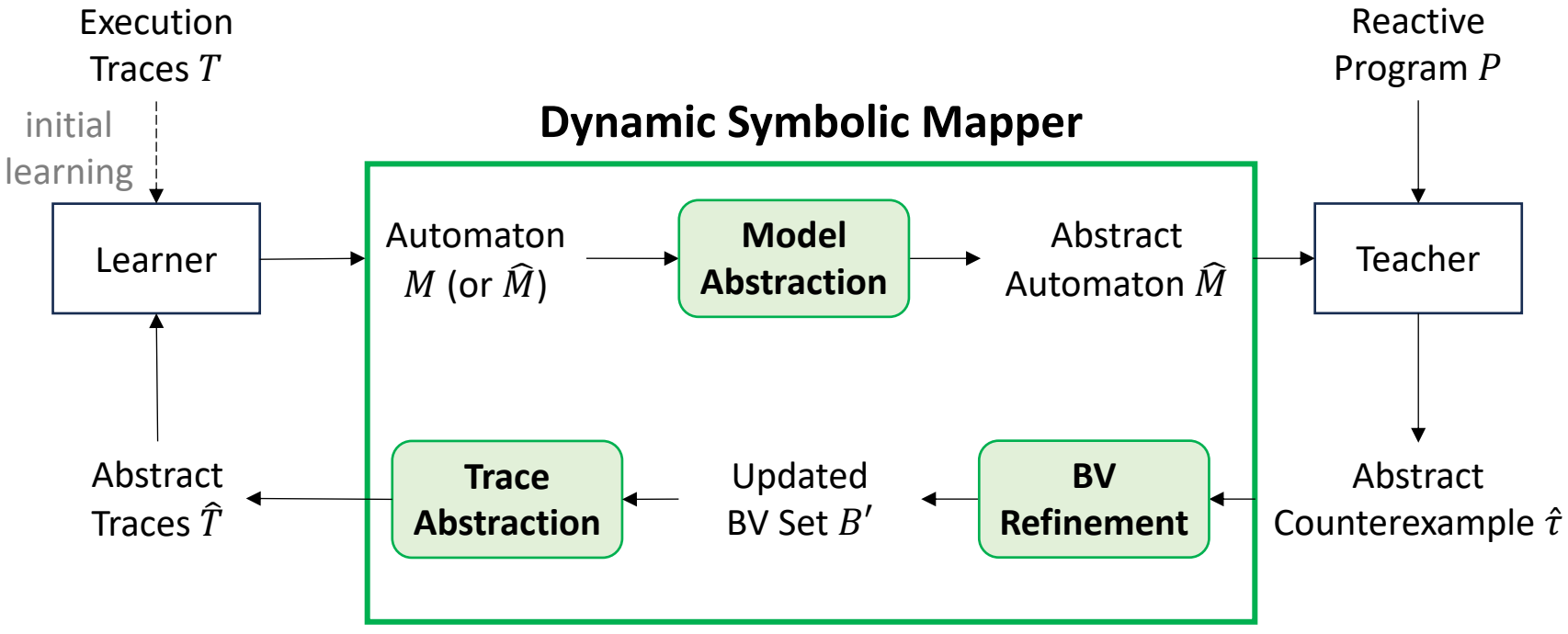
Solution: Dynamic Symbolic Mapper



Solution: Dynamic Symbolic Mapper



Solution: Dynamic Symbolic Mapper



Our mapper dynamically identifies necessary and sufficient Boolean variable sets at each active learning iteration.

Evaluation

- **RQ1: Our Approach (w/ Mapper) vs. Baseline (w/o Mapper)**
- RQ2: Dynamic vs. Static Symbolic Mappers
- RQ3: Fine-Grained vs. Coarse-Grained Variants

- Benchmarks: 120 reactive C programs
  - Simulink (45), SV-COMP (25), LeetCode (25), and Embedded Software (25)

Benchmark	$\|P\|$	# Nest	LoC	$\|Dom(st)\|$	$\|X\|$	# Bool	# Enum	# Num
Simulink	45	5	236	3.7	7.0	1.2	4.3	1.4
SV-COMP	25	5	1,526	4.6	34.8	1.6	11.0	22.2
LeetCode	25	17	187	5.2	25.8	4.4	1.5	19.9
Embedded	25	16	1,242	5.0	19.1	1.3	3.5	14.3
Avg.			704	4.5	19.2	2.0	5.0	12.3

RQ1-1: Active Learning Time

Table 1: Average Time per Benchmark.

Metric	Baseline	Ours	Reduction
Total time	4,915s	**2,184s**	**55.6%**
Teaching time	4,587s	**1,896s**	58.7%
Learning time	328s	**288s**	12.2%

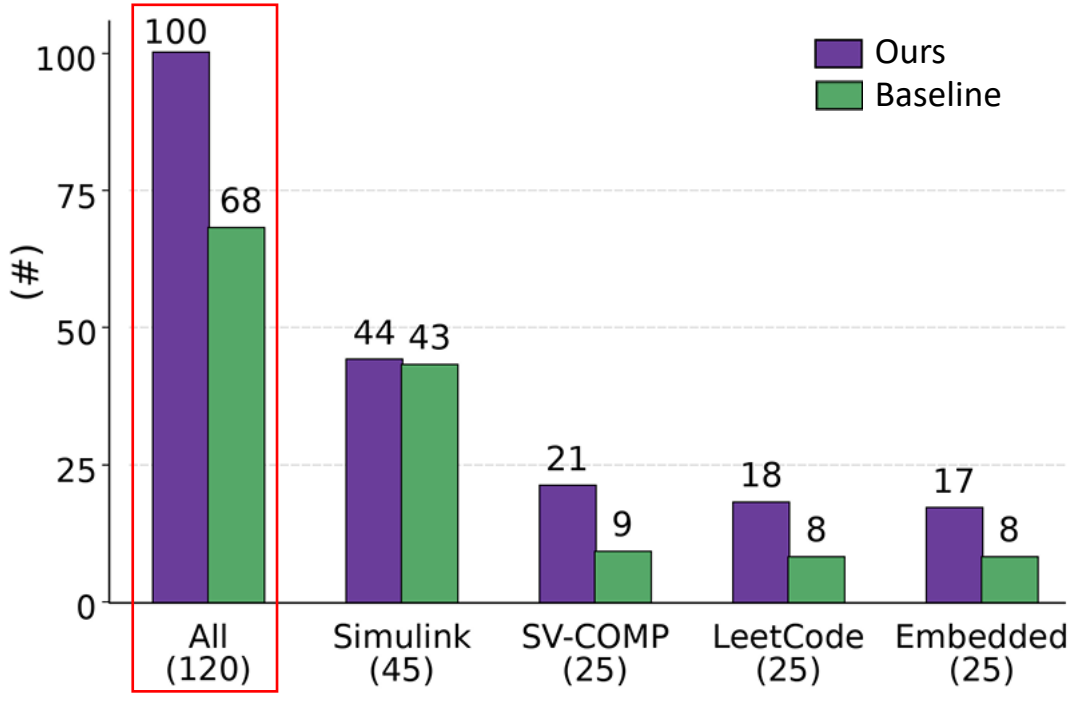
* T/O: 3 hours

Table 2: Cost per Active Learning Iteration.

Metric	Baseline	Ours	Speedup
# Learning iterations	979	**2,998**	-
Teaching time / iter	562.2s	**75.9s**	**7.4x faster**
Learning time / iter	40.2s	**11.5s**	**3.5x faster**

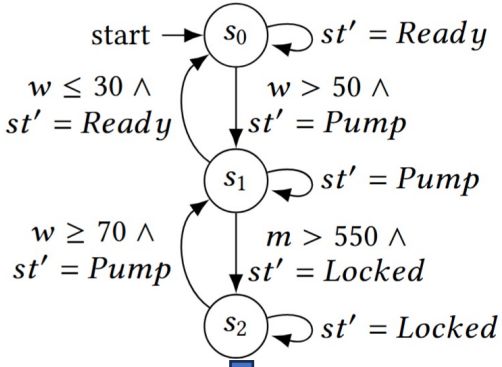
Our approach reduced the average active learning time by 55.6%!

RQ1-2: Number of Completions



Our approach learned 32 more sound automata than the baseline!

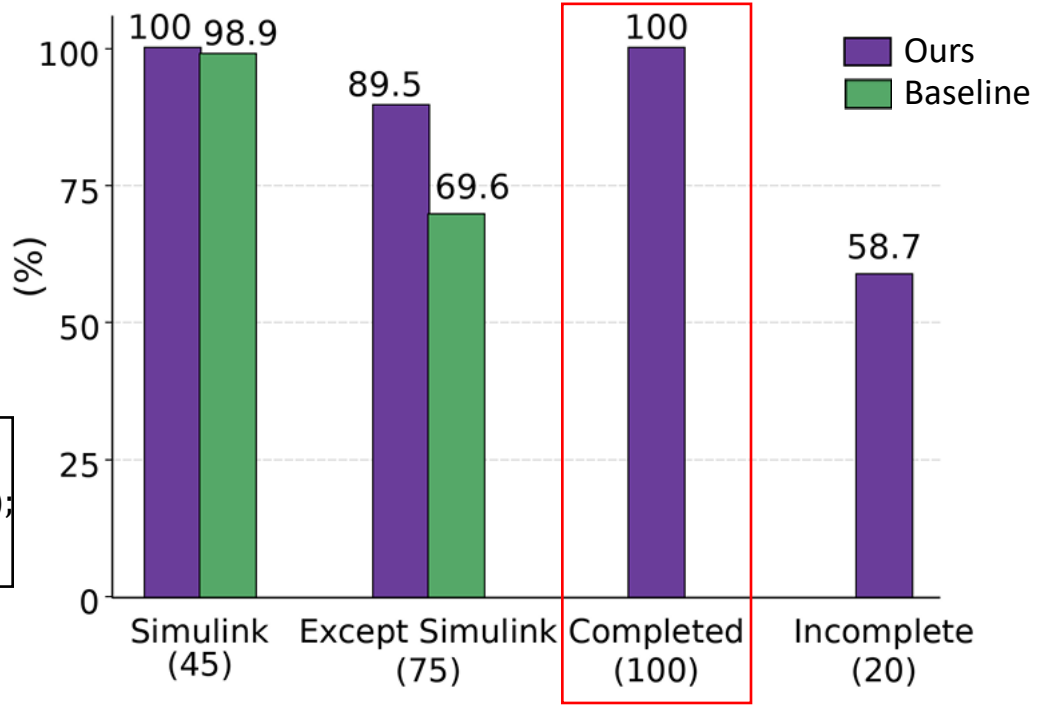
RQ1-3: Correctness



Formal specifications

$$\text{assert}(\neg(st = Ready \wedge w > 50) \vee (st' = Pump));$$
$$\text{assert}(\neg(st = Pump \wedge m > 550) \vee (st' = Locked));$$
$$\text{assert}(\neg(st = Locked \wedge w \geq 70) \vee (st' = Pump));$$

Model Checker (CBMC) → **Pass** or **Fail**



Specifications extracted from completed automata achieved a 100% pass rate!

Conclusion

Contributions:

- We introduced a novel **dynamic symbolic mapper**, which enables active automata learning to operate within a Boolean vector space.
- We implemented our approach and evaluated it on 120 reactive C programs, demonstrating improved active learning performance.

Ongoing and Future Work:

- We are developing a web interface for our approach, named AutoCL.
- We plan to support multi-task programs by learning each task separately and composing them at the system level.

Backup Slides

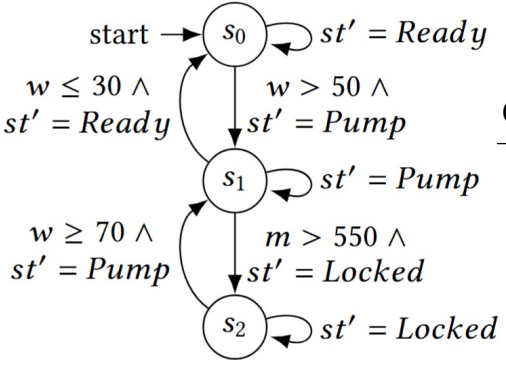
Why Learn Automata?

- Raw execution logs are too detailed:
  - **Hard to understand** the overall behavior of reactive programs.
- Manual code inspection does not scale:
  - State transitions are **scattered across code**.
- Learned automata are compact and formal:
  - They naturally capture the **stateful behavior of reactive programs**.
  - Can support model checking, model-based testing, and formal specification mining.

Model Abstraction (Initializing B)

1. **Extract** each predicate in M_1 and define a Boolean variable set B, containing $b_1: (w \leq 30)$, $b_2: (w \leq 50)$, $b_3: (w \geq 70)$, $b_4: (m \leq 550)$.
2. **Rewrite** each predicate in M_1 into a Boolean expression over B via syntactic substitution.

Initial Automaton M_1



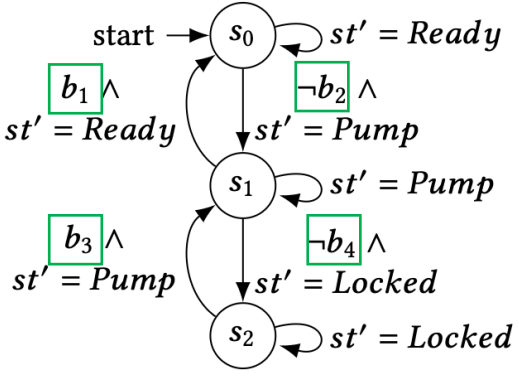
extracts

Dynamic Symbolic Mapper

$B = \{b_1, b_2, b_3, b_4\}$

rewrites

Initial Abstract Automaton $\widehat{M_1}$



Abstract Trace Conflicts

Abstract the initial trace set T_1

Abstract Observation Group for $Pump$ in $\widehat{T_1}$

st	b_1	b_4	st'
$Pump$	T	T	$Ready$
$Pump$	F	T	$Pump$
$Pump$	F	F	$Locked$

Add a new counterexample

New CE after teaching the abstract automaton $\widehat{M_1}$

New CE: $(\boldsymbol{st = Pump} \wedge \neg \boldsymbol{b_1} \wedge \neg \boldsymbol{b_4} \wedge st' = Pump)$

Conflict in the update group

Abstract Observation Group for $Pump$ in $\widehat{T_2}$

st	b_1	b_4	st'
$Pump$	T	T	$Ready$
$Pump$	F	T	$Pump$
$Pump$	F	F	$Locked$
$Pump$	F	F	$Pump$

Abstract Trace Conflict

Same Boolean vector, but two different next values:
$(Pump, F, F) \rightarrow Locked$ and $(Pump, F, F) \rightarrow Pump$

Boolean Variable Refinement

Abstract Trace Conflict

st	b_1	b_4	st'
$Pump$	T	T	$Ready$
$Pump$	F	T	$Pump$
$Pump$	F	F	$Locked$
$Pump$	F	F	$Pump$

Concretizes

Construct a conflict group

st	w	m	a	h'	st'
$Pump$	68	610	F	2	$Locked$
$Pump$	68	599	F	2	$Pump$

Synthesize a new predicate

Program Synthesizer

Synthesis result: $b_5 : (m < 600)$
Refined BV set $= \{b_1, b_4, b_5\}$

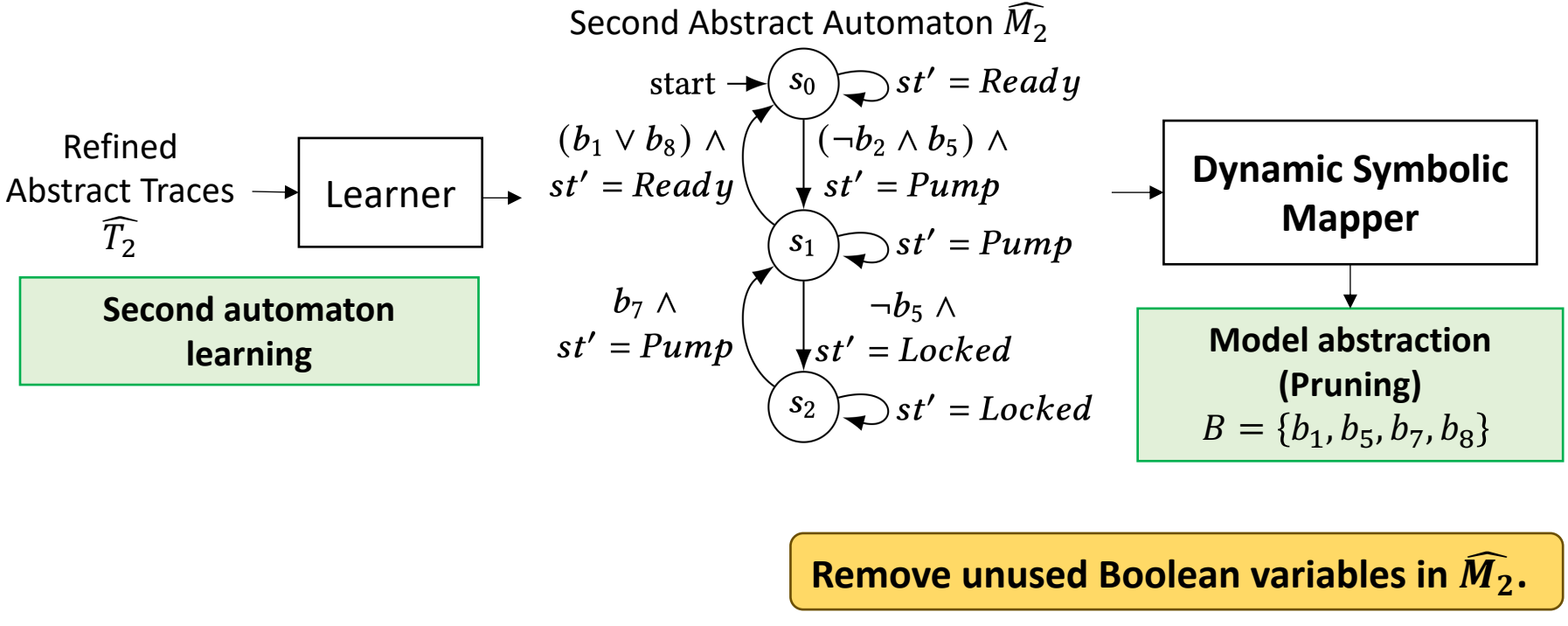
Re-apply trace abstraction

Conflict Resolved

st	b_1	b_4	b_5	st'
$Pump$	T	T	T	$Ready$
$Pump$	F	T	T	$Pump$
$Pump$	F	F	F	$Locked$
$Pump$	F	F	T	$Pump$

Model Abstraction (Pruning B)

- Assume that $B = \{b_1, b_2, b_3, b_4, b_5, b_7, b_8\}$ was extended after Boolean variable refinement.



Remove unused Boolean variables in $\widehat{M_2}$.

RQ2: Dynamic vs Static Mappers

Mapper	# Conflicts	Completed	Avg. $\lvert B \rvert$	Time taken	Mapper overhead
Dynamic	**0**	**87 / 99**	**6.4**	**1,658 s**	**31 s**
Static	21	63 / 99	34.4	3,993 s	0.2 s

* Excluded 21 trace-conflicting cases from the static mapper (statistics on 99 benchmarks)
* T/O: 3 hours

- **Static mapper led to 21 abstract trace conflicts (due to insufficient B).**
- **Dynamic mapper learned 24 more automata than the static mapper!**

Related Work

- Mapper Components:
  - Existing automata learning frameworks such as LearnLib [CAV'15] **require users to implement mappers** for each target system.
  - Aarts et al. [FM'12] automate mapper refinement, but it applies only to **classical automata** rather than symbolic automata.
- Symbolic Automata Learning:
  - Cassel et al. [FAC'16] use simple decision-tree-based partitioning, which **limits the expressiveness** of transition predicates.
  - Drews et al. [TACAS'17] rely on either simple interval predicates or **user-provided** Boolean variables.
- Specification Mining:
  - Template (Lemieux et al. [ASE'15]), deep learning (Le et al. [ISSTA'18]), and LLM-based methods (Ma et al. [ICSE'25]) infer specs but **lack formal soundness guarantees**.

Limitations

1. **Program Coverage**. Currently, AutoCL cannot handle programs with dynamic memory allocation, function pointers, or multi-threading.

2. **Bounded Soundness**. Since AutoCL uses CBMC (C Bounded Model Checker) as a teacher, it may produce unsound automata.

3. **Inefficient Boolean Variable Refinement**. It still relies on program variables, inheriting the learner's limitations: many variables with large domains.