

# Efficient CTL Verification via Horn Constraints Solving

**Tewodros A. Beyene**<sup>1</sup>

**fortiss GmbH, Munich, Germany**

joint work with **C. Popeea**<sup>2</sup> and **A. Rybalchenko**<sup>3</sup>

**fortiss**

**CQSE**

Microsoft  
**Research**

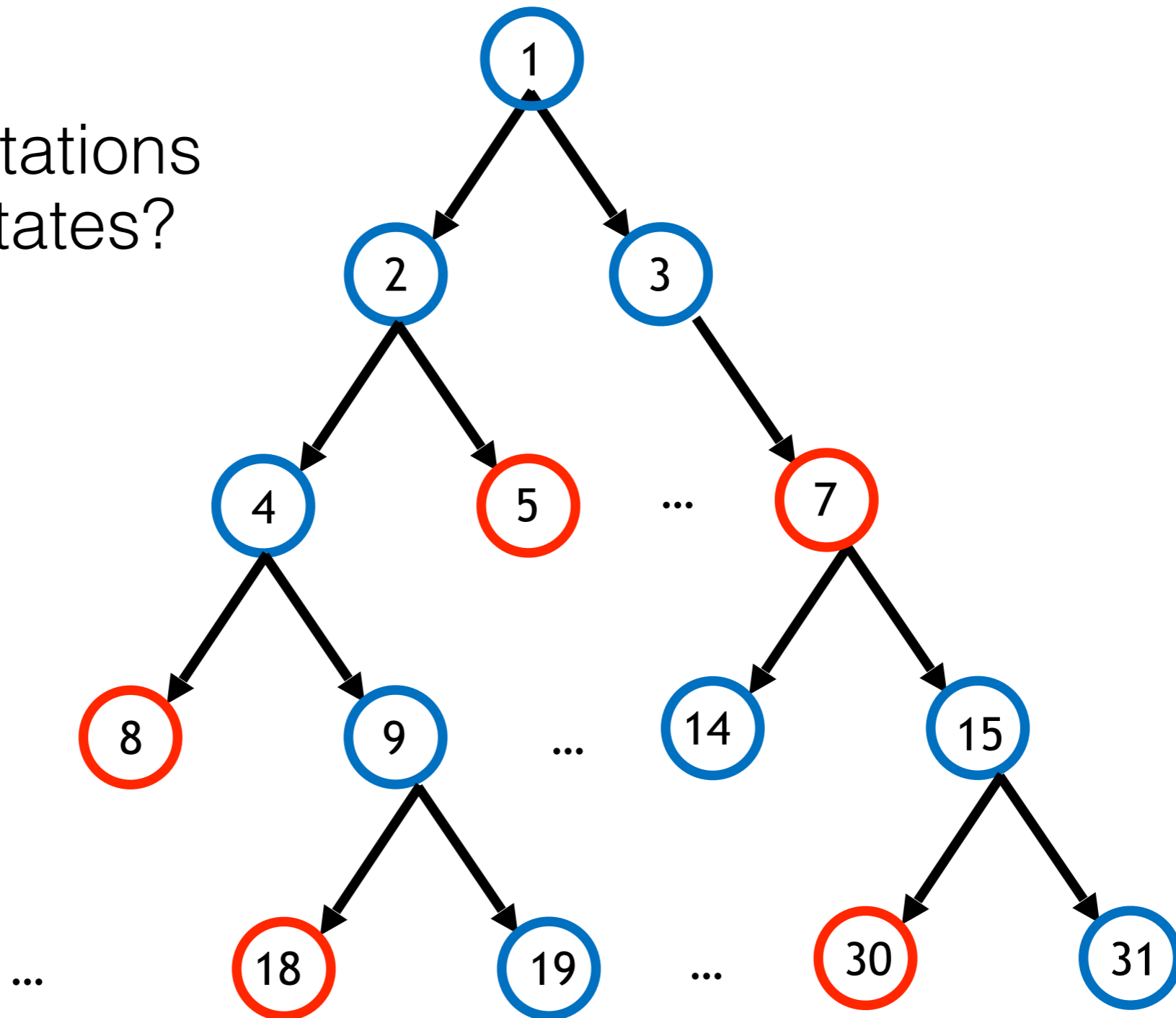
# Temporal logics

- Important class of **specification languages**
  - applications: **program verification, synthesis, security analysis**, etc.
  - when **assertions** do not suffice
- Deductive temporal reasoning:
  - (1) **Proof rules** to generate proof sub-goals
  - (2) **Solvers** for **auxiliary predicates**
- **Universal** and **existential** fragments

# Temporal universal properties

Do all computations avoid **error** states?

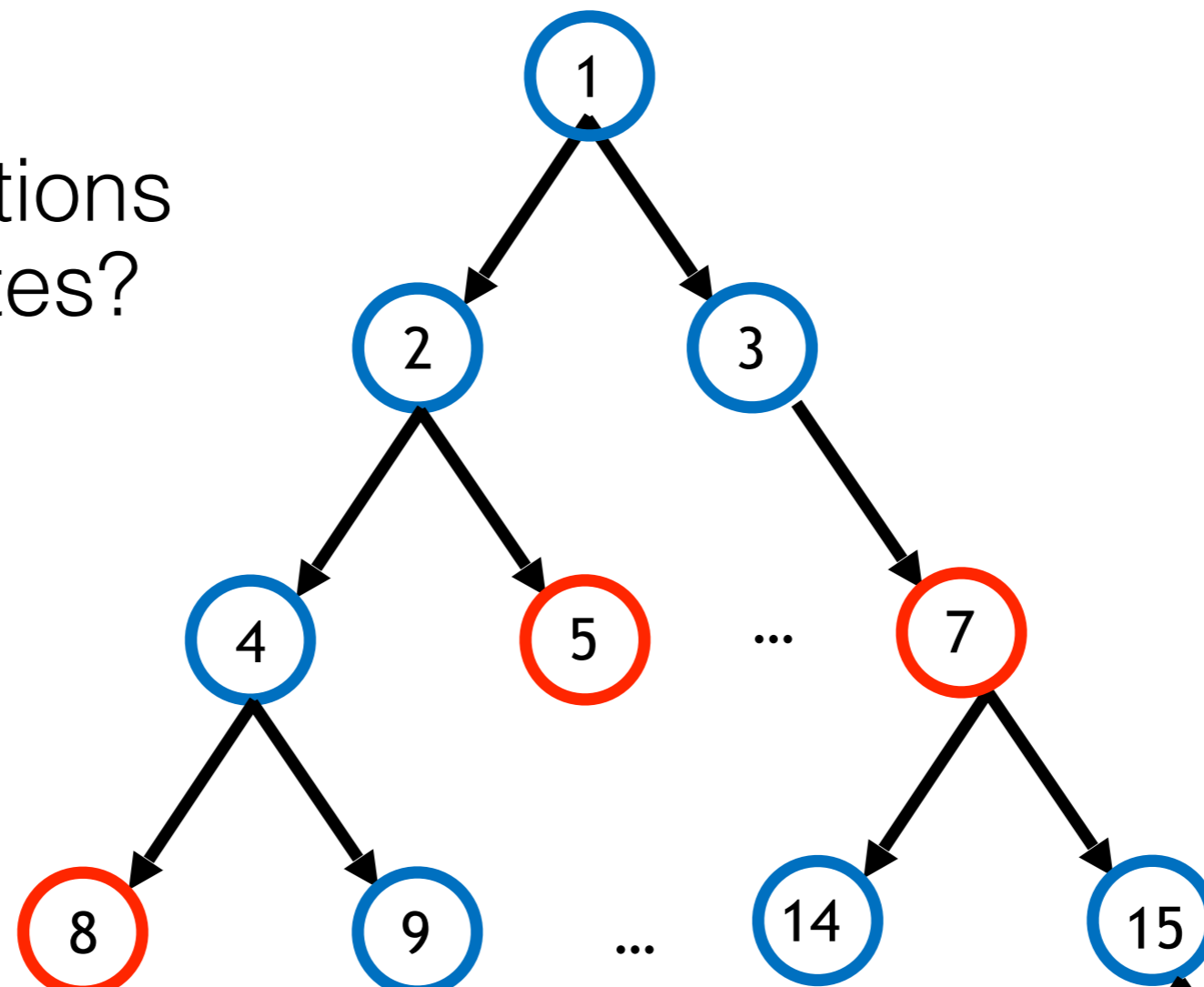
$AG \neg \bigcirc$



# Solving universal properties

Do all computations avoid **error** states?

$AG \neg \bigcirc$

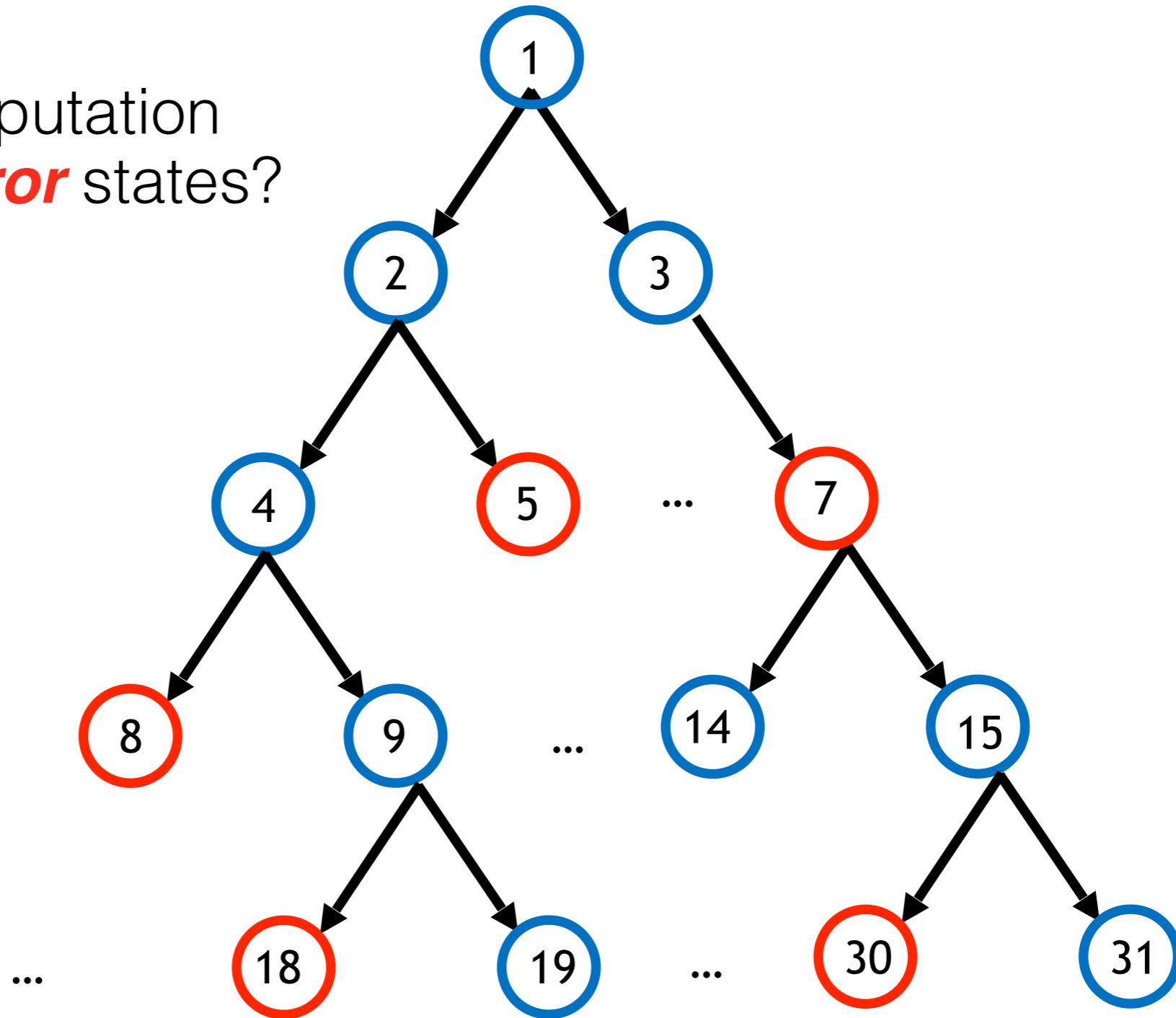


... a **success** story!

# Temporal existential properties

Is there a computation that avoids **error** states?

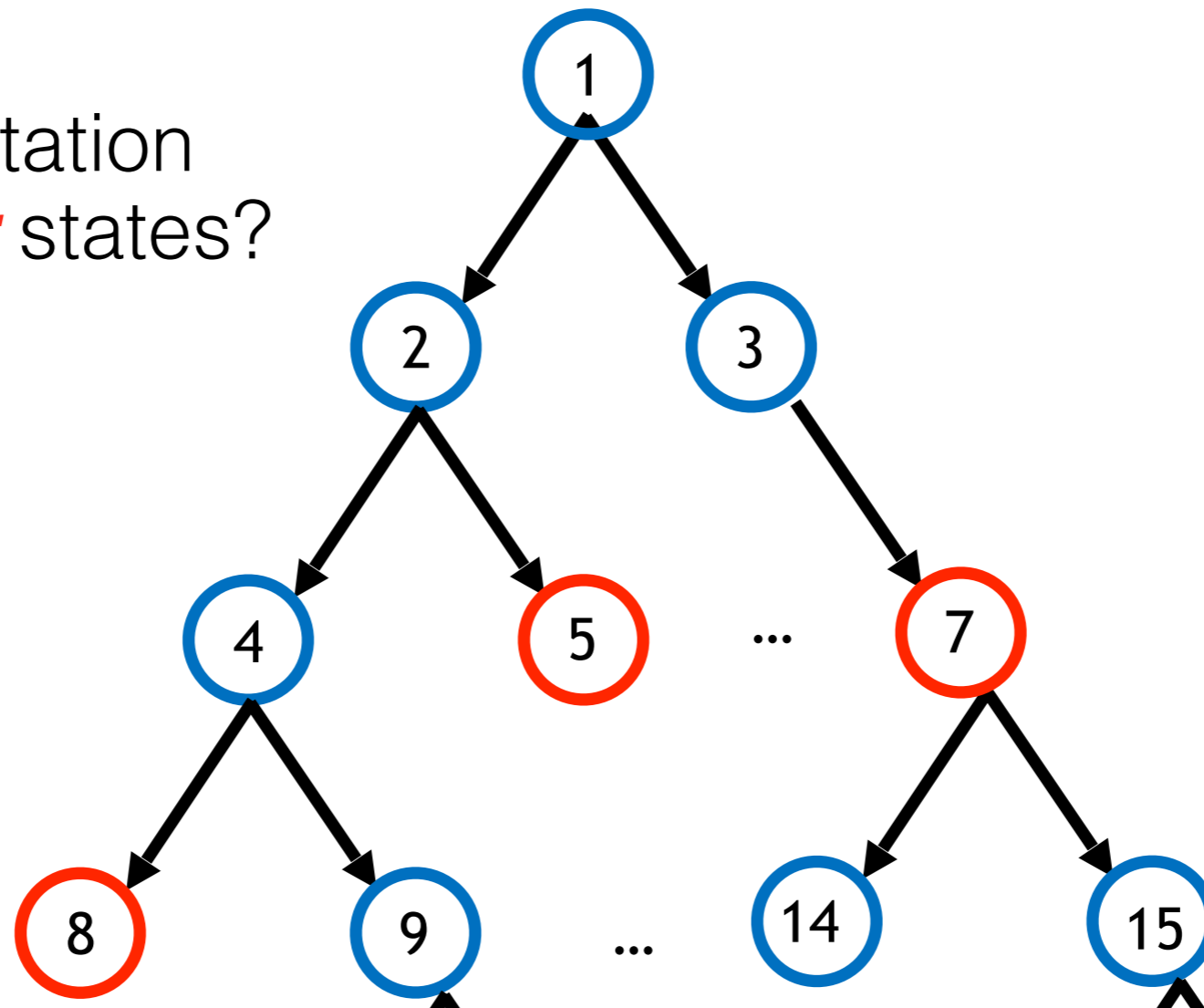
$EG \neg \bigcirc$



# Solving existential properties

Is there a computation that avoids **error** states?

$EG \neg \bigcirc$



... not as good as universal solvers!

# State of the art

- Specialised efforts, e.g., non-termination = ***EG true***
- Direct efforts: e.g., ***Cook-Koskinen PLDI2013***,  
***Cook-Khlaaf-Piterman FMCAD2014***.
  - program transformation specific to CTL

# In this talk ...

**CTL Verification**



**Proof rules**

- *forall-exists quantified Horn constraints*
- *well-foundedness constraints*

**E-HSF engine**



# Context

- Existentially quantified Horn constraints (CAV13)
- Proof rules for 2-player games and automation (POPL14, VSTTE15)
- CTL+FO verification (SPIN14)

## Solving Existentially Quantified Horn Clauses

Tewodros A. Beyene<sup>1</sup>, Corneliu Popescu<sup>2</sup>, and Andrey Rybalchenko<sup>1,2</sup>

<sup>1</sup> Technische Universität München  
<sup>2</sup> Microsoft Research Cambridge

**Abstract.** Temporal verification of universal (i.e., valid for all computation paths) properties of various kinds of programs, e.g., procedural, multi-threaded, or functional, can be reduced to finding solutions for equations in form of universally quantified Horn clauses extended with well-foundedness conditions. Dealing with existential properties (e.g., whether there exists a particular computation path), however, requires solving forall-exists quantified Horn clauses, where the conclusion part of some clauses contains existentially quantified variables. For example, a deductive approach to CTL verification reduces to solving such clauses. In this paper we present a method for solving forall-exists quantified Horn clauses extended with well-foundedness conditions. Our method is based on a counterexample-guided abstraction refinement scheme to discover witnesses for existentially quantified variables. We also present an application of our solving method to automation of CTL verification of software, as well as its experimental evaluation.

## A Constraint-Based Approach to Solving Games on Infinite Graphs

Tewodros A. Beyene  
Technische Universität München

Swarat Chaudhuri  
Rice University

Corneliu Popescu  
Technische Universität München

Andrey Rybalchenko  
Microsoft Research Cambridge and  
Technische Universität München

### Abstract

We present a constraint-based approach to computing winning strategies in two-player graph games over the state space of infinite-state programs. Such games have numerous applications in program verification and synthesis, including the synthesis of infinite-state reactive programs and branching-time verification of infinite-state programs. Our method handles games with winning conditions given by safety, reachability, and general Linear Temporal Logic (LTL) properties. For each property class, we give a deductive proof rule that — provided a symbolic representation of the game states — describes a winning strategy for a player.

a graph, and a player wins if the sequence of nodes visited by the player satisfies a certain  $\omega$ -regular winning condition. For example,

- To synthesize a reactive system from a temporal specification  $\varphi$ ,  $\exists X, \forall Y$ , one constructs a graph game where the goal of one player is to satisfy the specification and the goal of the other is to violate it. The desired system is realizable if and only if the first player has a winning strategy in this game.
- The problem of verifying a branching-time property  $\varphi$  of a system is naturally framed as a graph game [10]. Here, one player models the existential path quantifier in the property, the other

## Recursive Games for Compositional Program Synthesis

Tewodros A. Beyene<sup>1</sup>, Swarat Chaudhuri<sup>2</sup>,  
Corneliu Popescu<sup>1</sup>, and Andrey Rybalchenko<sup>3</sup>

<sup>1</sup> TU München, Munich, Germany  
beyene@in.tum.de

<sup>2</sup> Rice University, Texas, USA

<sup>3</sup> Microsoft Research, Cambridge, UK

**Abstract.** Compositionality, i.e., the use of procedure summarization instead of code inlining, is key to scaling automated verification to large code bases. In this paper, we present a way to exploit compositionality in the context of program synthesis.

The goal in our synthesis problem is to instantiate missing expressions in a procedural program so that the resulting program satisfies a safety or termination requirement in spite of an adversarial environment. The synthesis is modeled as a recursive game between two players.

## CTL+FO Verification as Constraint Solving

Tewodros A. Beyene  
Technische Universität München, Germany

Marc Brockschmidt  
Microsoft Research Cambridge, UK

Andrey Rybalchenko  
Microsoft Research Cambridge, UK

### ABSTRACT

Expressing program correctness often requires relating program data throughout (different branches of) an execution. Such properties can be represented using CTL+FO, a logic that allows mixing temporal and first-order quantification. Verifying that a program satisfies a CTL+FO property is a challenging problem that requires both temporal and data reasoning. Temporal quantifiers require discovery of reachability and ranking functions, while first-order quantifiers demand instantiation techniques. In this paper, we present a constraint-based method for proving CTL+FO properties automatically. Our method encodes the integrity between the temporal and first-order quantification required by a constraint-solving that combines recursion and existential quantification. By integrating this constraint-solving with an off-the-shelf solver we obtain an automatic verifier

the current system state, and temporal quantifiers allow to relate this data to system states reached at a later point.

While CTL+FO and similar logics have been identified as a specification language before, no fully automatic method to check CTL+FO properties on infinite-state systems was developed. Hence, the current state of the art is to either produce verification results specific to small subclasses of programs, or using more general program modifications that explicitly introduce and instantiate ghost variables, which are then used in (standard) CTL quantification.

In this paper, we present a fully automatic procedure to transform a CTL+FO verification problem into a system of existentially quantified recursive Horn clauses. Such systems can be solved by leveraging recent advances in constraint solving [2], allowing to blend first-order and temporal reasoning. Our method benefits from the simplicity of

# Outline

- Illustration
- Solving engine: E-HSF
- Proof rules
- Experimental evaluation

# Outline

- **Illustration**
- Solving engine: E-HSF
- Proof rules
- Experimental evaluation

# Example

```
assume(y=0)
while(1) {
    x = x+y
    y = nondet()
}
```

**EF (x ≥ 0) ?**

# Example

```
assume(y=0)
while(1) {
    x = x+y
    y = nondet()
}
```

**EF (x ≥ 0) ?**

$v = (x, y)$

$\text{init}(v) = (y=0)$

$\text{next}(v, v') = (x'=x+y \wedge y'=?)$

$(\text{init}(v), \text{next}(v, v')) \models \text{EF } (x \geq 0) ?$

# Constraint Generation

1.  $\forall v: \text{init}(v) \rightarrow \text{inv}(v)$
2.  $\forall v: \text{inv}(v) \wedge \neg (x \geq 0) \rightarrow \exists v': \text{next}(v, v') \wedge$   
 $\text{inv}(v') \wedge \text{round}(v, v')$
3.  $\text{well-founded}(\text{round}(v, v'))$

---

$$(\text{init}(v), \text{next}(v, v')) \models \text{EF } (x \geq 0)$$

- solve for  $\text{inv}(v')$  and  $\text{round}(v, v')$

# Solving challenges

1.  $\forall v: \text{init}(v) \rightarrow \text{inv}(v)$
2.  $\forall v: \text{inv}(v) \wedge \neg (x \geq 0) \rightarrow \exists v': \text{next}(v, v') \wedge$   
 $\text{inv}(v') \wedge \text{round}(v, v')$
3.  $\text{well-founded}(\text{round}(v, v'))$

---

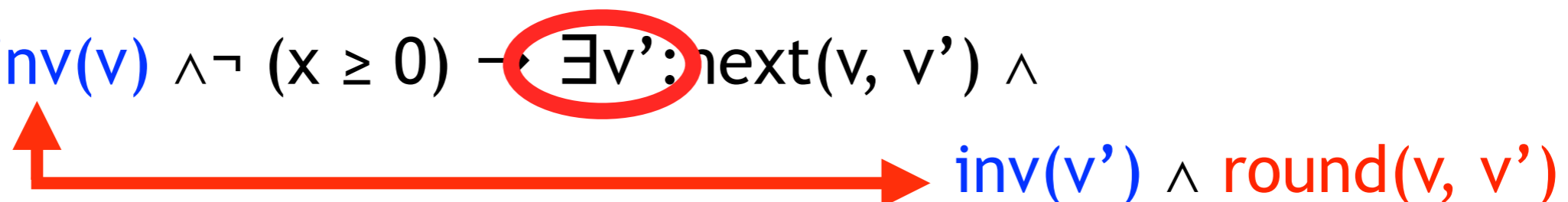
$$(\text{init}(v), \text{next}(v, v')) \models \text{EF } (x \geq 0)$$

- solve for  $\text{inv}(v')$  and  $\text{round}(v, v')$

# Solving challenges

1.  $\forall v: \text{init}(v) \rightarrow \text{inv}(v)$

2.  $\forall v: \text{inv}(v) \wedge \neg (x \geq 0) \rightarrow \exists v': \text{next}(v, v') \wedge$   
 $\text{inv}(v') \wedge \text{round}(v, v')$



3.  $\text{well-founded}(\text{round}(v, v'))$

---

$$(\text{init}(v), \text{next}(v, v')) \models \text{EF } (x \geq 0)$$

- solve for  $\text{inv}(v')$  and  $\text{round}(v, v')$



# Solving challenges

1.  $\forall v: \text{init}(v) \rightarrow \text{inv}(v)$
2.  $\forall v: \text{inv}(v) \wedge \neg (x \geq 0) \rightarrow \exists v': \text{next}(v, v') \wedge$   
 $\text{inv}(v') \wedge \text{round}(v, v')$
3.  $\text{well-founded}(\text{round}(v, v'))$

---

$(\text{init}(v), \text{next}(v, v')) \models \text{EF } (x \geq 0)$

- solve for  $\text{inv}(v')$  and  $\text{round}(v, v')$

# Example: one solution

```
assume(y=0)
while(1) {
  x = x+y
  y = nondet()
}
```

EF (x ≥ 0) ?

v = (x, y)

init(v) = (y=0)

next(v, v') = (x'=x+y ∧ y'=?)

(init(v), next(v, v')) ⊨ EF (x ≥ 0) ?

1.  $\forall v: \text{init}(v) \rightarrow \text{inv}(v)$
2.  $\forall v: \text{inv}(v) \wedge \neg (x \geq 0) \rightarrow \exists v': \text{next}(v, v') \wedge \text{inv}(v') \wedge \text{round}(v, v')$
3. well-founded(round(v, v'))

$\text{sol}(\text{inv}(v)) := (x \geq 0 \vee x < 0 \wedge y = 0 \vee x < 0 \wedge y > 0)$

$\text{sol}(\text{round}(v, v')) := (x < 0 \wedge y = 0 \wedge x' = x \wedge y' > 0 \vee$   
 $x < 0 \wedge y > 0 \wedge x' = x + y \wedge y' > 0)$

# Outline

- Illustration
- **Solving engine: E-HSF**
- Proof rules
- Experimental evaluation

# E-HSF: inputs and outputs

- Inputs: **Horn constraints** and **Skolem Templates**

- e.g., 
$$\left\{ \begin{array}{l} \dots \rightarrow \text{aux}(x), \\ \text{aux}(x) \rightarrow \exists y: \dots \wedge \text{aux}(y) \end{array} \right\}$$

- Output: model for each **auxiliary predicate**
  - instantiations for **existentially** quantified variables

# E-HSF: skolemization

- Reformulates problem as witness finding
- Given  $\forall v: \text{body}(v) \rightarrow \exists w : \text{head}(v, w)$ 
  1.  $\forall v, w: \text{body}(v) \wedge \text{wit}(v, w) \rightarrow \text{head}(v, w)$
  2.  $\text{body}(v) \rightarrow \text{domain}(\text{wit}(v, w))$
- (2) *ensures* that for any state in  $\text{body}(v)$ , a witness is defined by  $\text{wit}(v, w)$ .
- **Skolem Templates** define space of witness relations
- applies **HSF**

# E-HSF: basic features

- A **CEGAR**-like scheme
- Refines **skolem template** instantiation
  - unlike in **CEGAR**, no monotonicity!
- A **global constraint** ensures progress of template refinement
  - by keeping track of **counter-examples** from previous template instantiations

# Outline

- Illustration
- Solving engine: E-HSF
- **Proof rules**
- Experimental evaluation

# Proof rules

- Inspired by ***Compositional proof system for CTL\**** [***Kesten, Pnueli, TCS 05***]
- 2 set of proof rules
  - **Constraint generation**: for basic CTL state formulas.
  - **Decomposition**: for non-basic CTL state formulas.



# Proof rule RuleCtIEG

Find an assertion  $inv(v)$  such that:

$$\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \\ inv(v) \rightarrow q(v) \end{array}$$

---

$$(p(v), next(v, v')) \models_{CTL} EG q(v)$$

- for basic CTL formula with EG outer operator

# Proof rule RuleCtlDecompBin

Given a CTL formula  $f(\psi_1(v), \psi_2(v))$  where  $f \in \{AU, EU, \wedge, \vee\}$ , and a transition system  $(p(v), next(v, v'))$ , find assertions  $q_1(v)$  and  $q_2(v)$  such that:

$$\begin{array}{c} p(v) \rightarrow f(q_1(v), q_2(v)), \\ (q_1(v), next(v, v')) \models_{CTL} \psi_1(v) \quad (q_2(v), next(v, v')) \models_{CTL} \psi_2(v) \end{array}$$

---

$$(p(v), next(v, v')) \models_{CTL} f(\psi_1(v), \psi_2(v))$$

- for non-basic CTL state formulas with binary outer operator

# Proof rules

1. RuleCtlDecompUni
2. RuleCtlDecompBin
3. RuleCtlEX
4. RuleCtlEG
5. RuleCtlEU
6. RuleCtlAX
7. RuleCtlAG
8. RuleCtlAU
9. RuleCtlEF
10. RuleCtlAF

# Outline

- Illustration
- Solving engine: E-HSF
- Proof rules
- **Experimental evaluation**

# Experiment setup

- **Challenging benchmarks**: *Windows OS fragment and PostgreSQL pgarch* [**Cook-Koskinen PLDI 13, Cook-Khlaff-Piterman FMCAD 14**]
- Linear **Skolem Templates** were sufficient

# Evaluation

Program $P$	Property $\varphi$	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook [37]	E-HSF	Cook [37]
Windows OS fragment 1 (29 LOC)	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
Windows OS fragment 2 (58 LOC)	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
Windows OS fragment 3 (370 LOC)	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
Windows OS fragment 4 (380 LOC)	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
Windows OS fragment 5 (43 LOC)	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
PostgreSQL pgarch (70 LOC)	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4

# Evaluation

Program $P$	Property $\varphi$	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook [37]	E-HSF	Cook [37]
Windows OS fragment 1 (29 LOC)	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
Windows OS fragment 2 (58 LOC)	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
Windows OS fragment 3 (370 LOC)	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
Windows OS fragment 4 (380 LOC)	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
Windows OS fragment 5 (43 LOC)	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
PostgreSQL pgarch (70 LOC)	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4

# Evaluation

Program $P$	Property $\varphi$	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook [37]	E-HSF	Cook [37]
Windows OS fragment 1 (29 LOC)	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
Windows OS fragment 2 (58 LOC)	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
Windows OS fragment 3 (370 LOC)	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
Windows OS fragment 4 (380 LOC)	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
Windows OS fragment 5 (43 LOC)	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
PostgreSQL pgarch (70 LOC)	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4
		<b>26.4</b>	<b>85.0</b>	<b>20.8</b>	<b>62.6</b>



# Evaluation

Program $P$	Property $\varphi$	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook [37]	E-HSF	Cook [37]
Windows OS fragment 1 (29 LOC)	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
Windows OS fragment 2 (58 LOC)	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
Windows OS fragment 3 (370 LOC)	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
Windows OS fragment 4 (380 LOC)	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
Windows OS fragment 5 (43 LOC)	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
PostgreSQL pgarch (70 LOC)	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4

 **70% time**

<b>26.4</b>	<b>85.0</b>	<b>20.8</b>	<b>62.6</b>
-------------	-------------	-------------	-------------

# Summary

- contrasting success of temporal reasoning for existential and universal fragments
- **Horn-constraint** based solution: **CTL proof rules** and **E-HSF engine**
- experimental evaluation on a set of challenging benchmarks
- generic approach performs better!