Decidable Deductive Verification

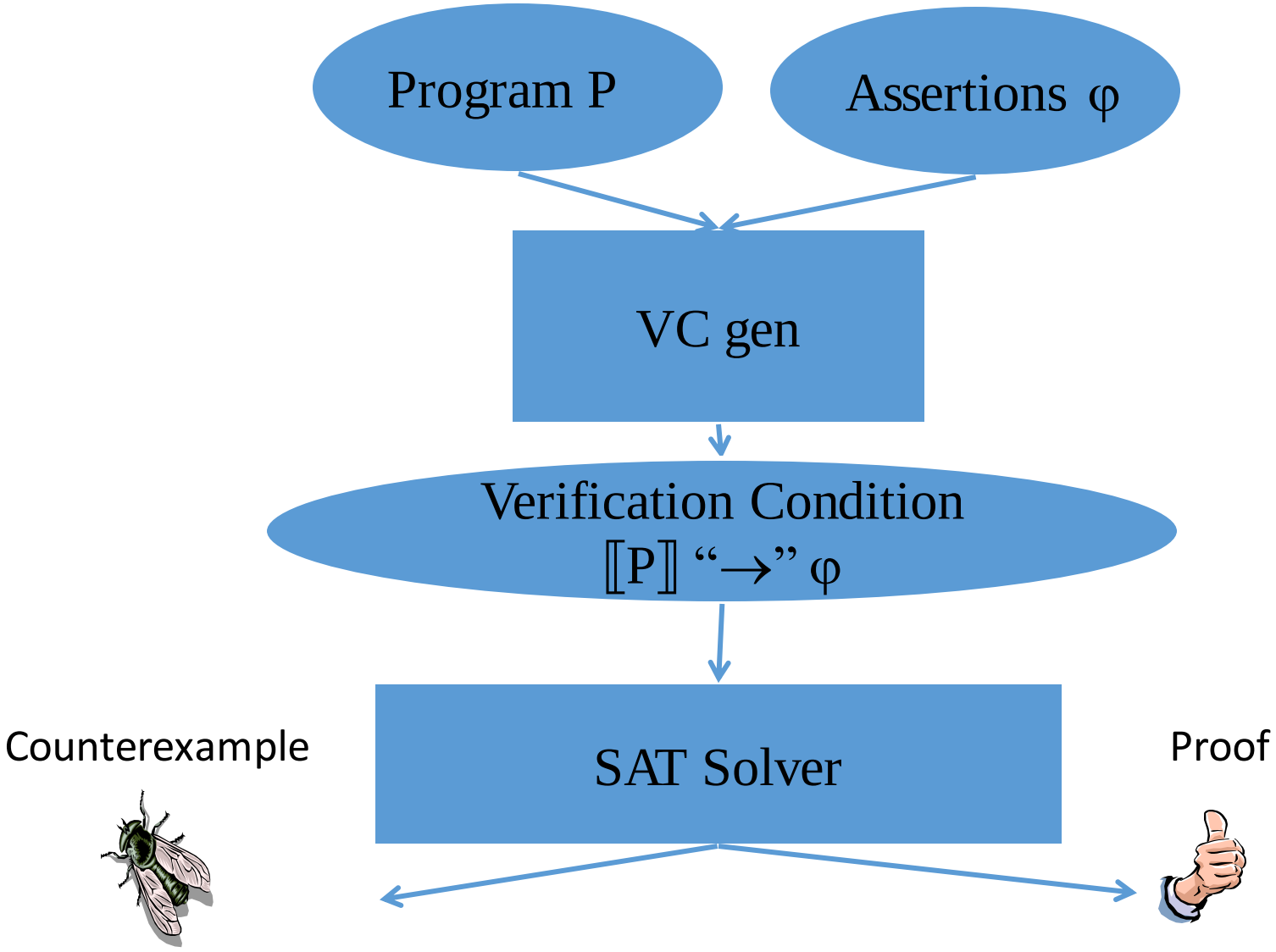
Mooly Sagiv

Shachar Itzhaky

Oded Padon

The Blavatnik School
of Computer Science
The Raymond and Beverly Sackler
Faculty of Exact Sciences
Tel Aviv University

Deductive Verification

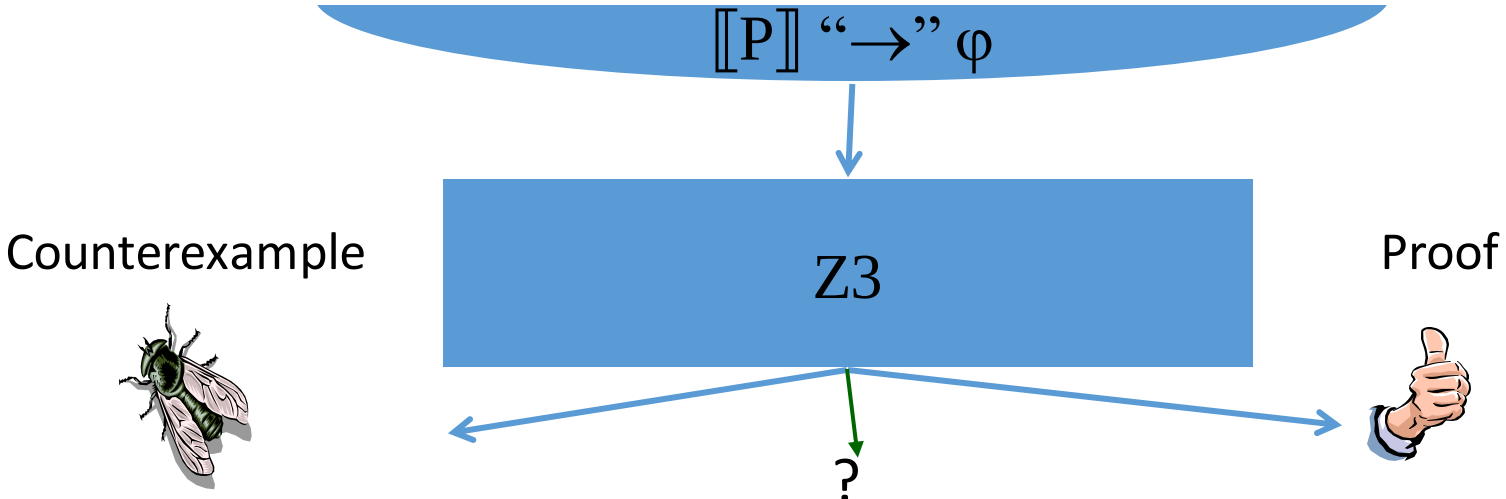


Where does undecidability occur?

- Interesting programmatic features
  - Unbounded integers
  - Floating points
  - Heaps
  - Unbounded arrays
  - Recursive procedures
  - Concurrency and distribution
- Interesting assertions
  - Quantifications
  - Numeric
  - Data types

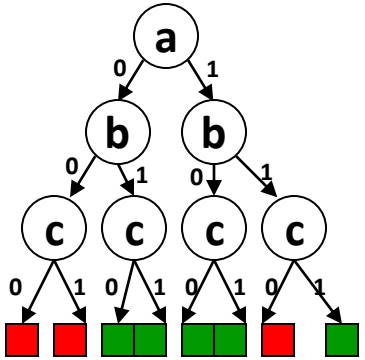
The IronFleet Project

The butterfly effect describes the phenomenon that a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program. When this change in outcome causes the verifier to hit a time limit or other resource limit, previously succeeding verifications turn into spurious verification failures. The butterfly effect thus leads to verification instability, user frustration, and overall a degraded user experience.

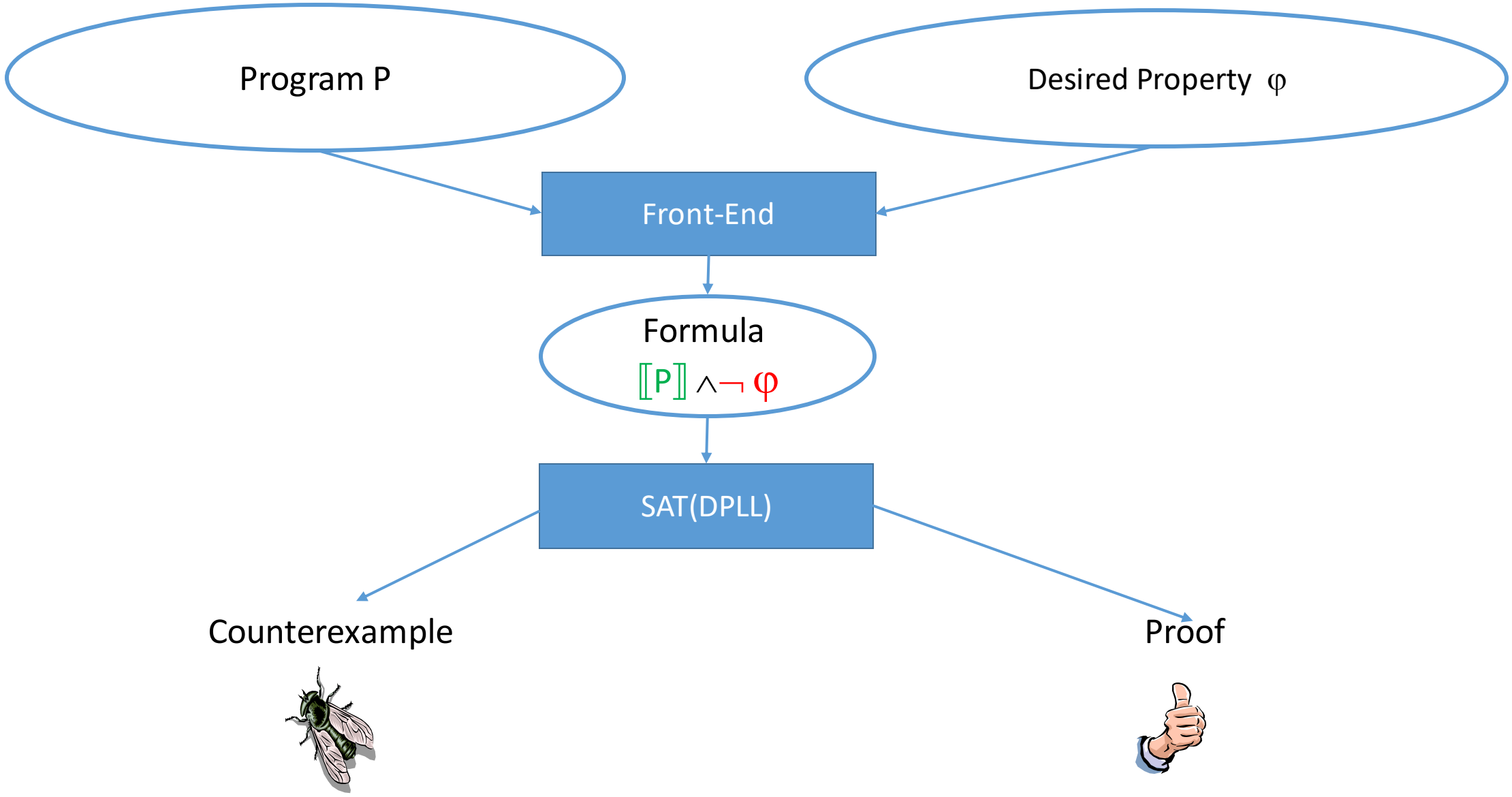


The SAT Problem

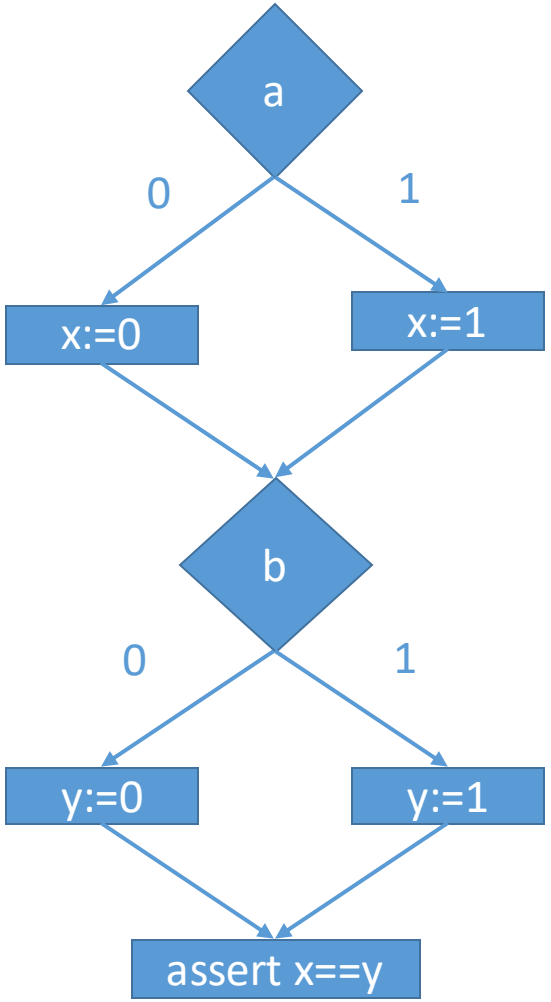
- Given a propositional formula (Boolean function)
  - $\varphi = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$
- Determine if φ is satisfiable
  - Find a satisfying assignment or report that such does not exist
- For *n* variables, there are 2^n possible truth assignments to be checked



Verification by reductions to SAT



Verification by reduction to SAT



SAT Query:

$((a \wedge x) \vee (\neg a \wedge \neg x))$

$\wedge$

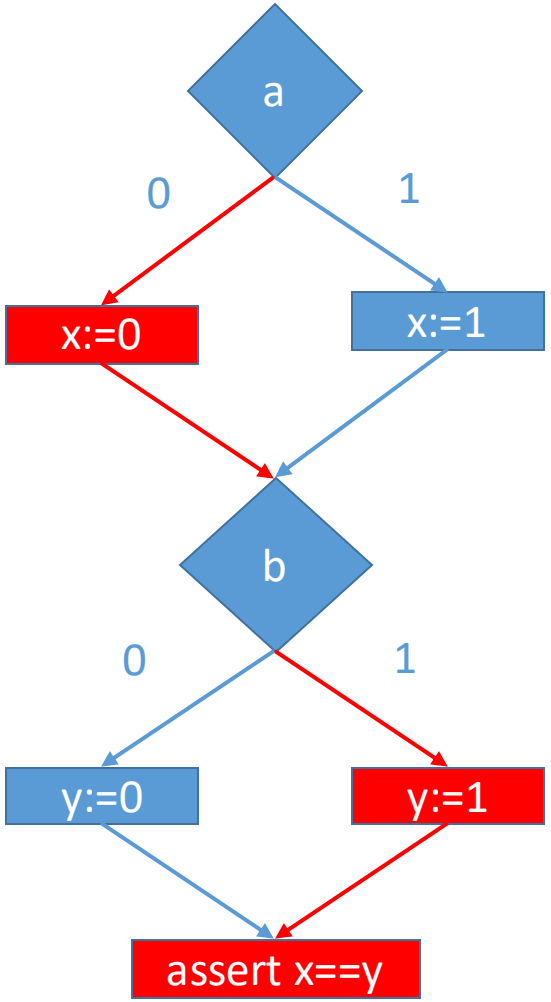
$((b \wedge y) \vee (\neg b \wedge \neg y))$

$\wedge$

$((x \wedge \neg y) \vee (\neg x \wedge y)$

?

Verification by reduction to SAT



SAT Query:

$((a \wedge x) \vee (\neg a \wedge \neg x))$

$\wedge$

$((b \wedge y) \vee (\neg b \wedge \neg y))$

$\wedge$

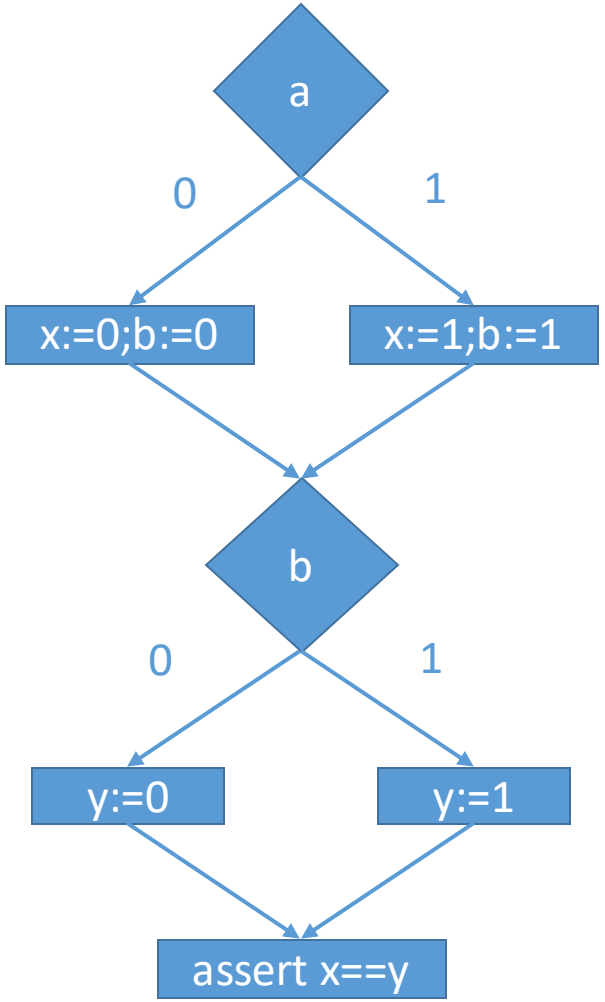
$((x \wedge \neg y) \vee (\neg x \wedge y)$

?

SAT Answer:

Satisfiable by a=0, b = 1

Verification by reduction to SAT



SAT Query:

$((a \wedge x \wedge b) \vee (\neg a \wedge \neg x \wedge \neg b))$

$\wedge$

$((b \wedge y) \vee (\neg b \wedge \neg y))$

$\wedge$

$((x \wedge \neg y) \vee (\neg x \wedge y)$

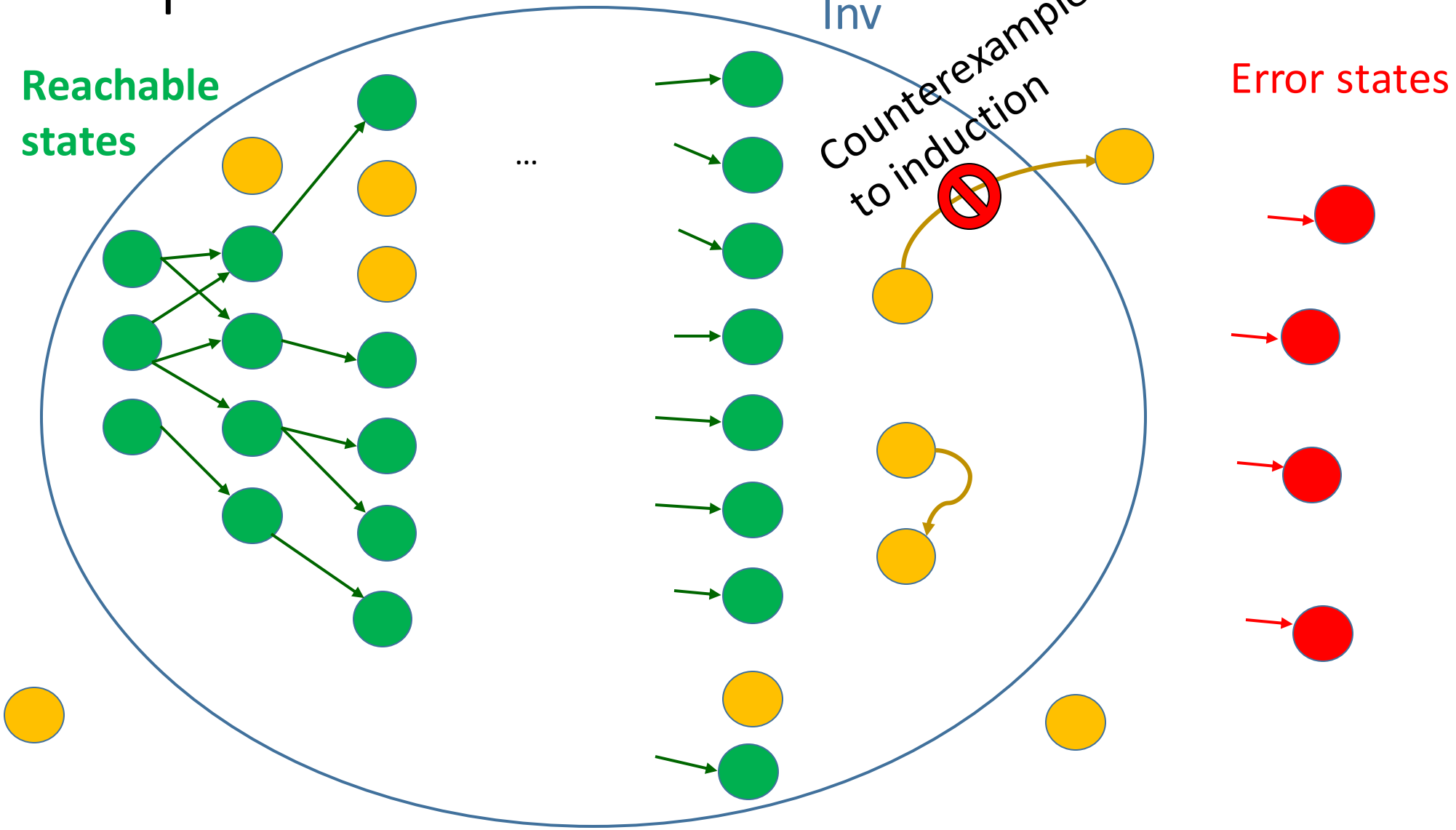
?

SAT Answer:
Unsatisfiable

Challenge 1: Dealing with loops

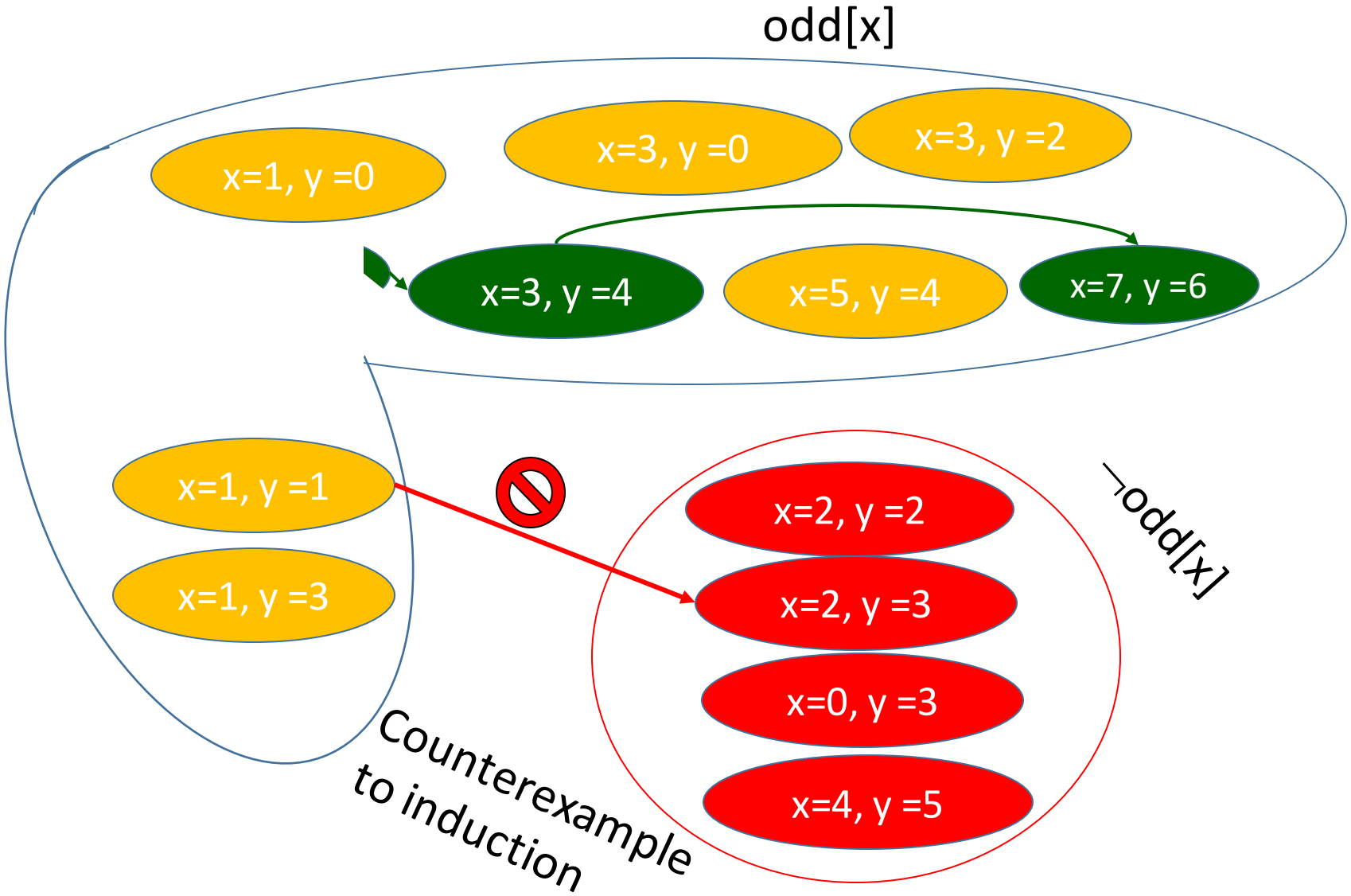
- Bounded model checking: Verify that there are no bugs up to some loop unfolding [BMC]
- Deductive verification: user specified candidate inductive loop invariants
  - The induction hypothesis: strong enough to imply preservation in the next step
- Automatically infer loop inductive invariants
  - Abstract interpretation

Loop invariants



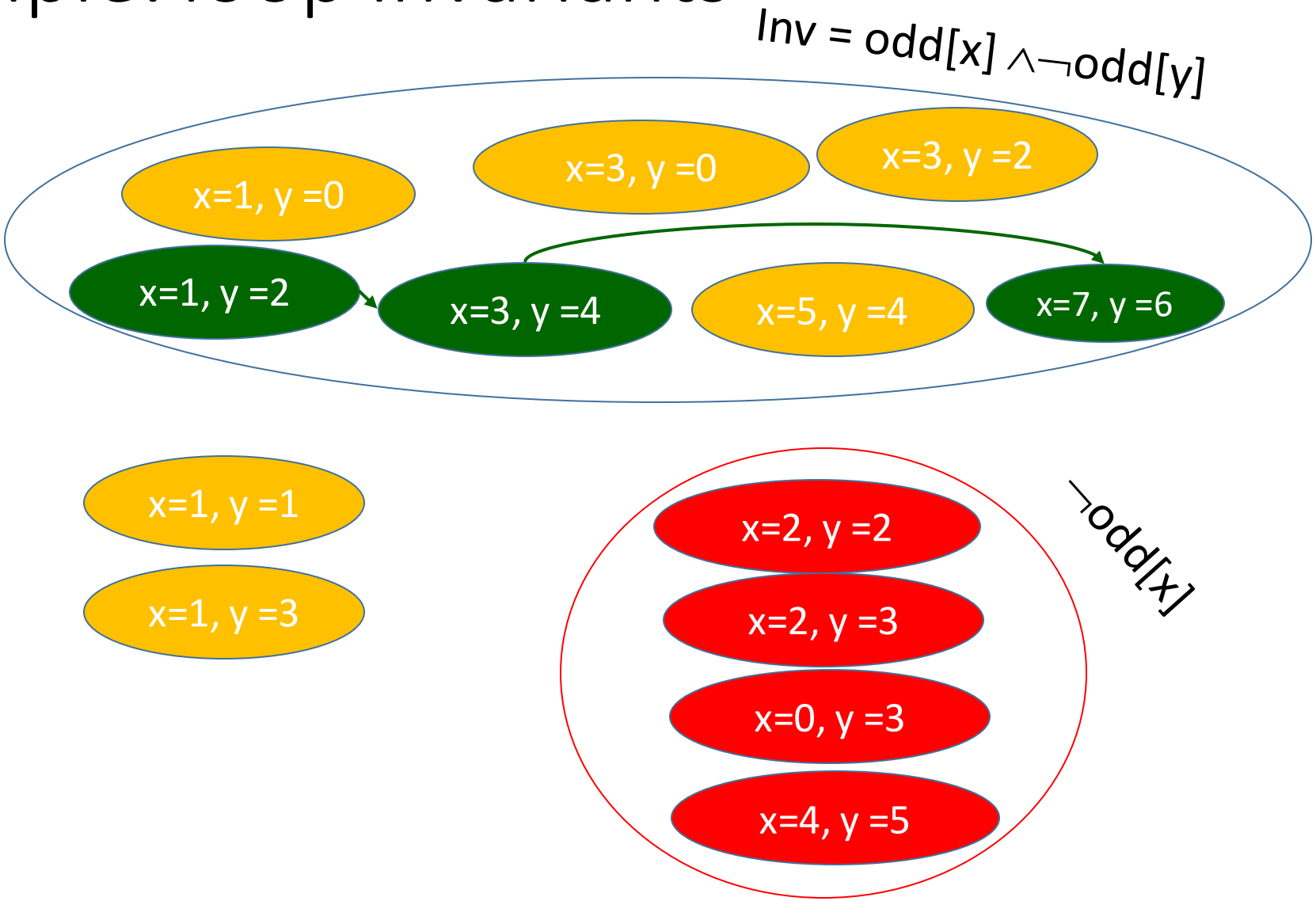
Simple Example: loop Invariants

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

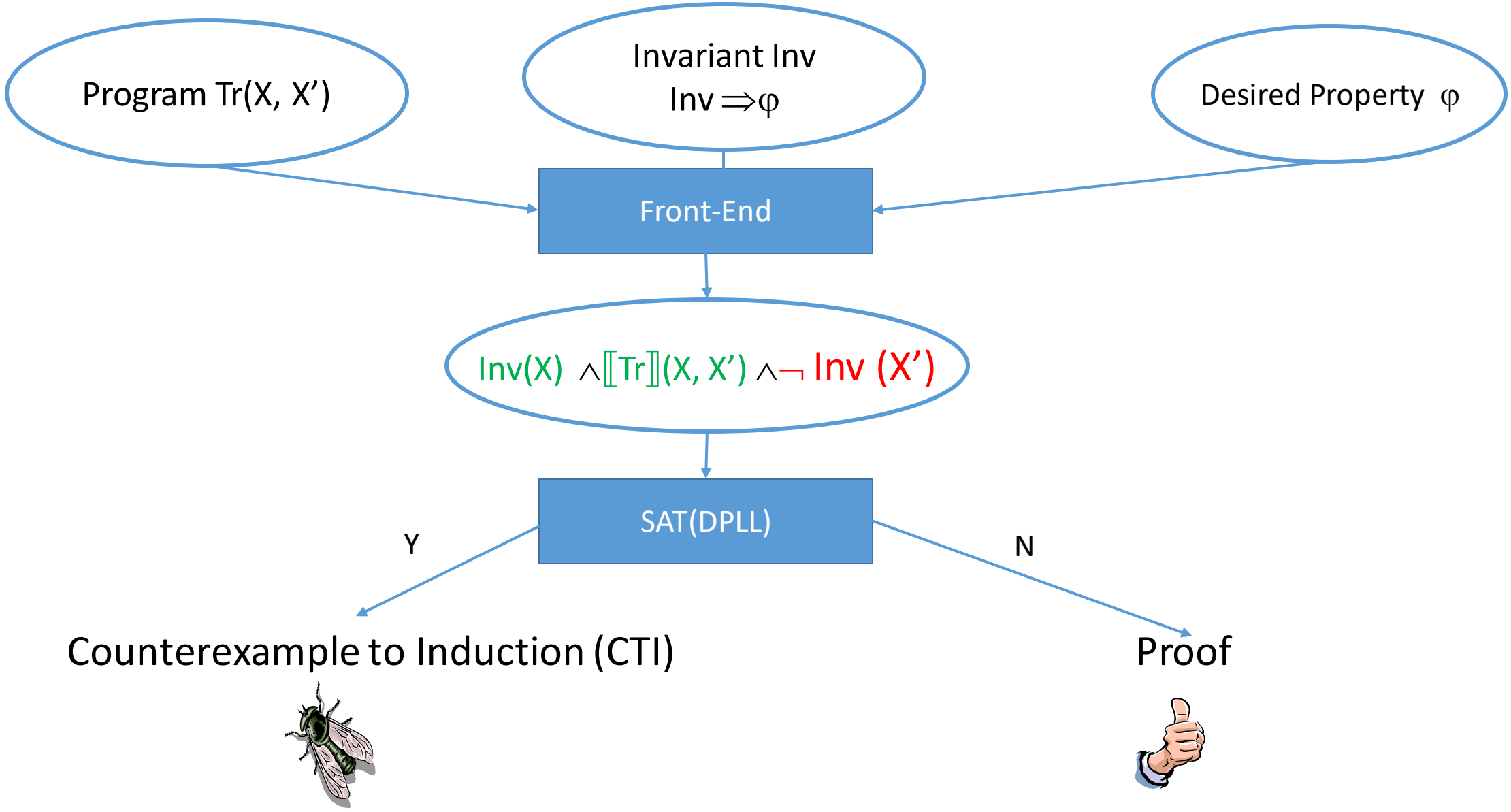


Simple Example: loop Invariants

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```



Deductive verification by reductions to SAT



Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

$Inv = at(3) \Rightarrow odd[x]$

$at(3) \Rightarrow odd[x]$

Solver

Is there a behavior of P that violates the inductiveness of I?

Y

Inv

$\neg$Inv

3: $odd[x]=1, odd[y]=1$

$odd[x]' = (odd[x] \wedge \neg odd[y]) \vee (\neg odd[x] \wedge odd[y])$
$odd[y]' = odd[y]$

$odd[x]'=0, odd[y]'=1$

Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

$Inv = at(3) \Rightarrow odd[x] \wedge \neg odd[y]$

$at(3) \Rightarrow odd[x]$

Solver

Is there a behavior of P that violates the inductiveness of I?

N

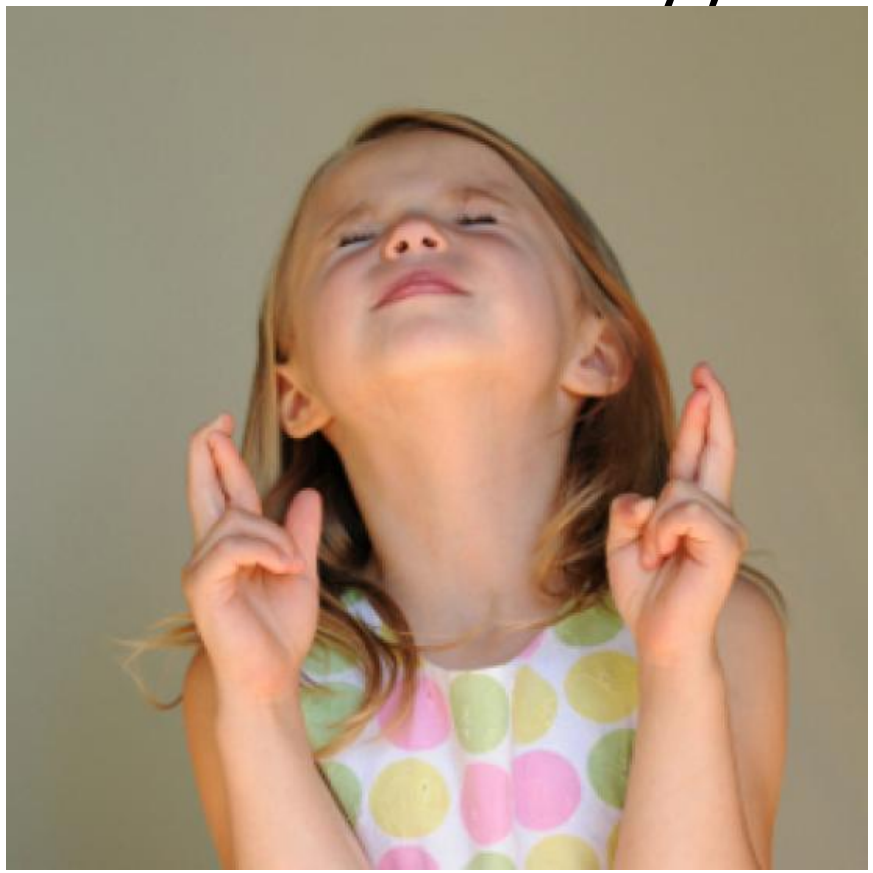
Proof

Challenge 2: Dealing with unbounded states

- Unbounded memory allocation
  - New objects
- Unbounded number of nodes in a distributed system
- Unbounded number of messages in the queue
- …

Common Solution: SMT (Sat Modulo Theory)

- Support rich logic
  - $\exists a, b, c, n: a, b, c > 0 \wedge n > 2 \wedge a^n + b^n = c^n$
  - Theories
  - Numeric
    - Non linear arithmetic
    - Strings
  - Quantifier alternation
  - Separation logic
  - Transitive closure
  - …
- Divergence
- Inconclusive & Infinite counterexamples



"The difficulty lies, not in new ideas, but in escaping from the old ones, which ramify, for those brought up as most of have been, into the corners of our minds." — John Maynard Keynes

Decidable Deductive Verification

- Express the assertions in a decidable logic
- For every assertion a:
  - There exists an algorithm to determine if a holds for some states (SAT)

Useful Decidable Logics

- Propositional logic
  - "odd[x] $\wedge$ odd[y] $\Rightarrow$ even[x']"
- Linear Arithmetic
  - $2 * x + 3 * y \geq 0 \wedge x \leq 2 * y$
- Effectively Propositional Logic

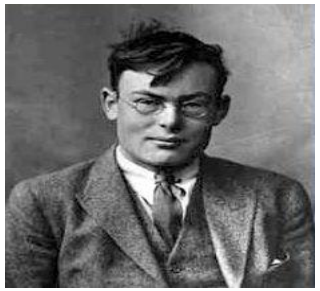
Simple Example

- $\exists a, b.\ r(a, b) \wedge \forall z.\ \neg r(z, z)$

Effectively PRopositional Logic – EPR

a.k.a. Bernays-Schönfinkel-Ramsey class

- Limited fragment of first-order logic
  - Restricted quantifier prefix: $\exists^* \forall^* \phi_{Q.F.}$
  - No $\forall^* \exists^*$
    - No recursive function symbols
    - No arithmetic
- Small model property
  - $\exists x_1,\ldots, x_n.\forall y_1,\ldots,y_m.\phi_{Q.F.}$ has a model iff it has a model of at most n+k elements (k - number of constant symbols)
- Satisfiability is decidable
  - NEXPTIME complete/Σ_2
- Support from Z3, Iprover, Vampire



F. Ramsey. *On a problem in formal logic.* Proc. London Math. Soc. 1930

Simple Example

$$\exists a, b: \forall x, y: \quad r(a, b) \wedge \neg r(b, a) \wedge (r(x, y) \quad \Rightarrow r(y, x))$$

$=_{SAT}$

$$\forall x, y: \quad r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(x, y) \Rightarrow r(y, x))$$

$=_{SAT}$

$$\forall x, y \in \{c_a, c_b\}: \quad r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(x, y) \Rightarrow r(y, x))$$

$=_{SAT}$

$$\begin{aligned}
&(r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(c_a, c_a) \Rightarrow r(c_a, c_a))) \wedge \\
&(r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(c_a, c_b) \Rightarrow r(c_b, c_a))) \wedge \\
&(r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(c_b, c_a) \Rightarrow r(c_a, c_b))) \wedge \\
&(r(c_a, c_b) \wedge \neg r(c_b, c_a) \wedge (r(c_b, c_b) \Rightarrow r(c_b, c_b)))
\end{aligned}$$

$=_{SAT}$

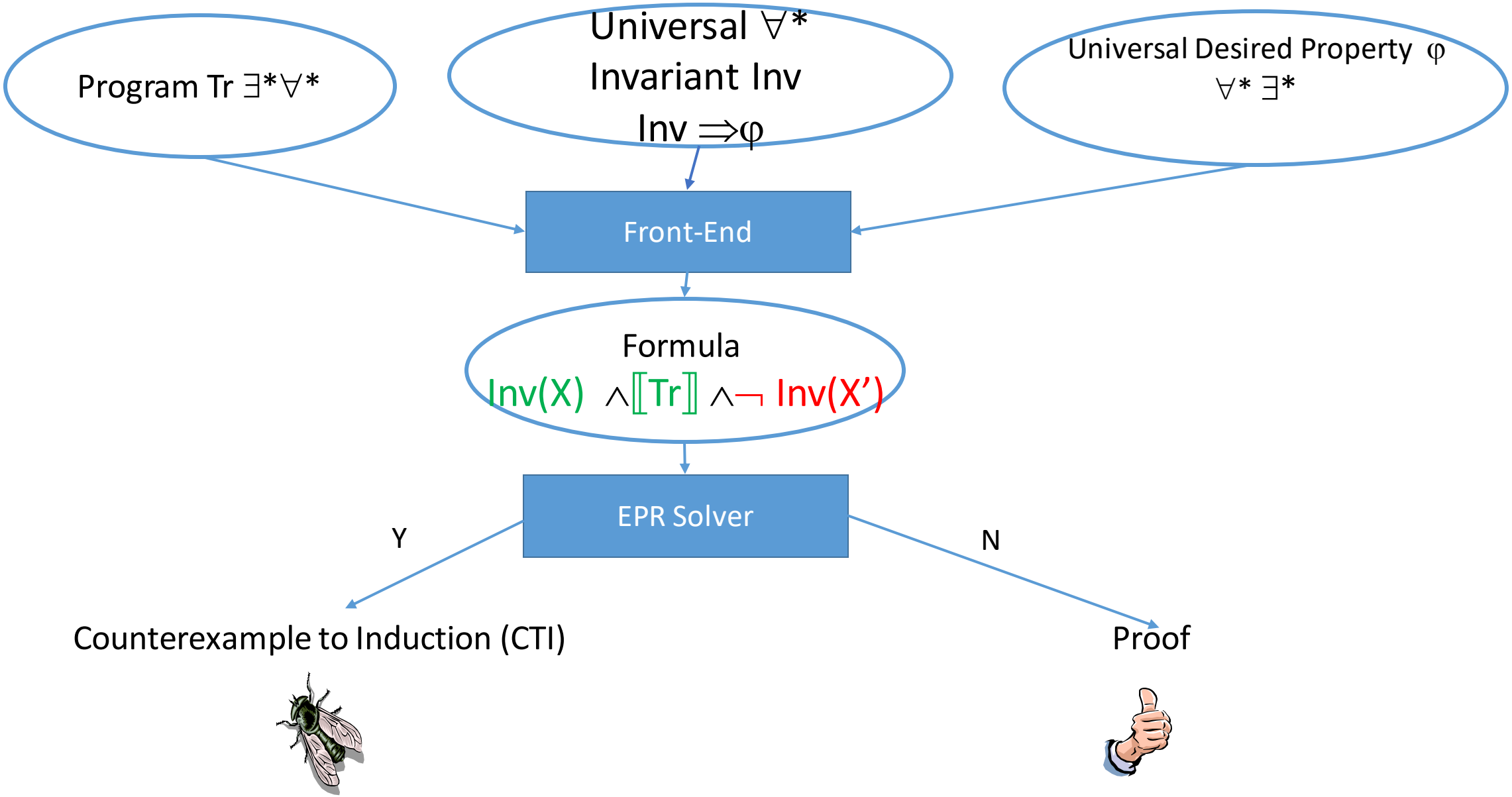
UNSAT

$$\begin{aligned}
&(p_{ab} \wedge \neg p_{ba} \wedge (p_{aa} \Rightarrow p_{aa})) \wedge (p_{ab} \wedge \neg p_{ba} \wedge (p_{ab} \Rightarrow p_{ba})) \wedge \\
&(p_{ab} \wedge \neg p_{ba} \wedge (p_{ba} \Rightarrow p_{ab)}) \wedge (p_{ab} \wedge \neg p_{ba} \wedge (p_{bb} \Rightarrow p_{bb}))
\end{aligned}$$

Why EPR?

- $\forall x.\ le(x, x)$ Reflexive
- $\forall x, y, z.\ le(x, y) \wedge le(y, z) \Rightarrow le(x, z)$ Transitive
- $\forall x, y.\ le(x, y) \wedge le(y,x) \Rightarrow x=y$ Antisymmetric
- $\forall x, y.\ le(x,y) \vee le(y, x)$ Total
- $\exists zero.\ \forall x.\ le(zero, x)$ Non-empty
- $\forall x.\ \exists y.\ le(x, y) \wedge x \neq y$

Deductive verification by reductions to EPR



Common Wisdom

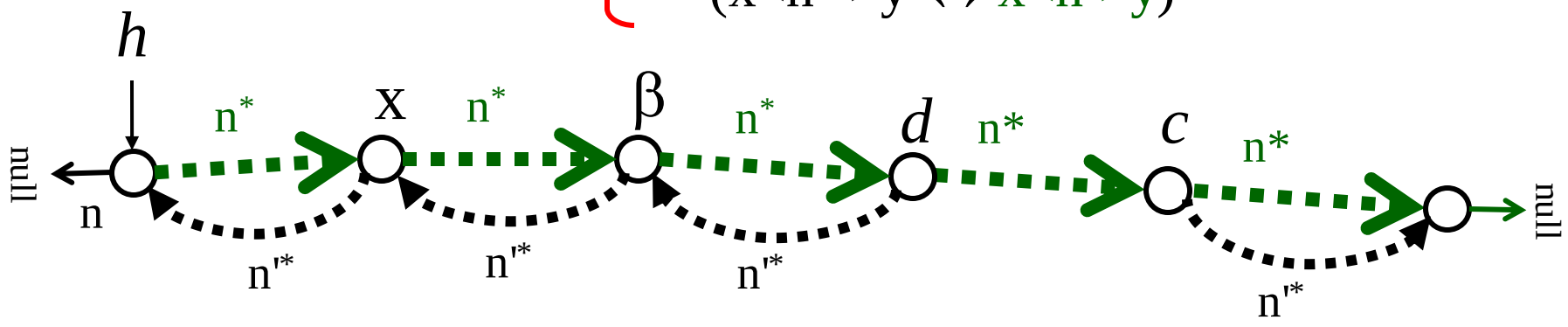
- EPR is too weak to express properties of software systems
- Cannot express theories like numerics
  - The number of nodes in a distributed systems
  - Quorums in consensus algorithms
- Reason about paths in a dynamically changing graphs
  - Not even first order expressible
  - What about modularity

Invariant List Reversal

assume ac [h]$\wedge\forall$x: h <n*>x

```
Node reverse(Node h) {
  Node c = h;   Node d = null;
   while  {I}  (c != null)  {
      Node t = c.next;
      c.next =  d;
      d = c;
      c = t;
      }
return d
}
```

$$I = \forall x, y: \begin{cases} x\langle n'^{*}\rangle y \leftrightarrow y\langle n^{*}\rangle x & d\langle n'^{*}\rangle x \\ c\langle n'^{*}\rangle x \wedge & \neg d\langle n'^{*}\rangle x \\ \quad (x\langle n'^{*}\rangle y \leftrightarrow x\langle n^{*}\rangle y) & \end{cases}$$



Linked List Manipulations can count

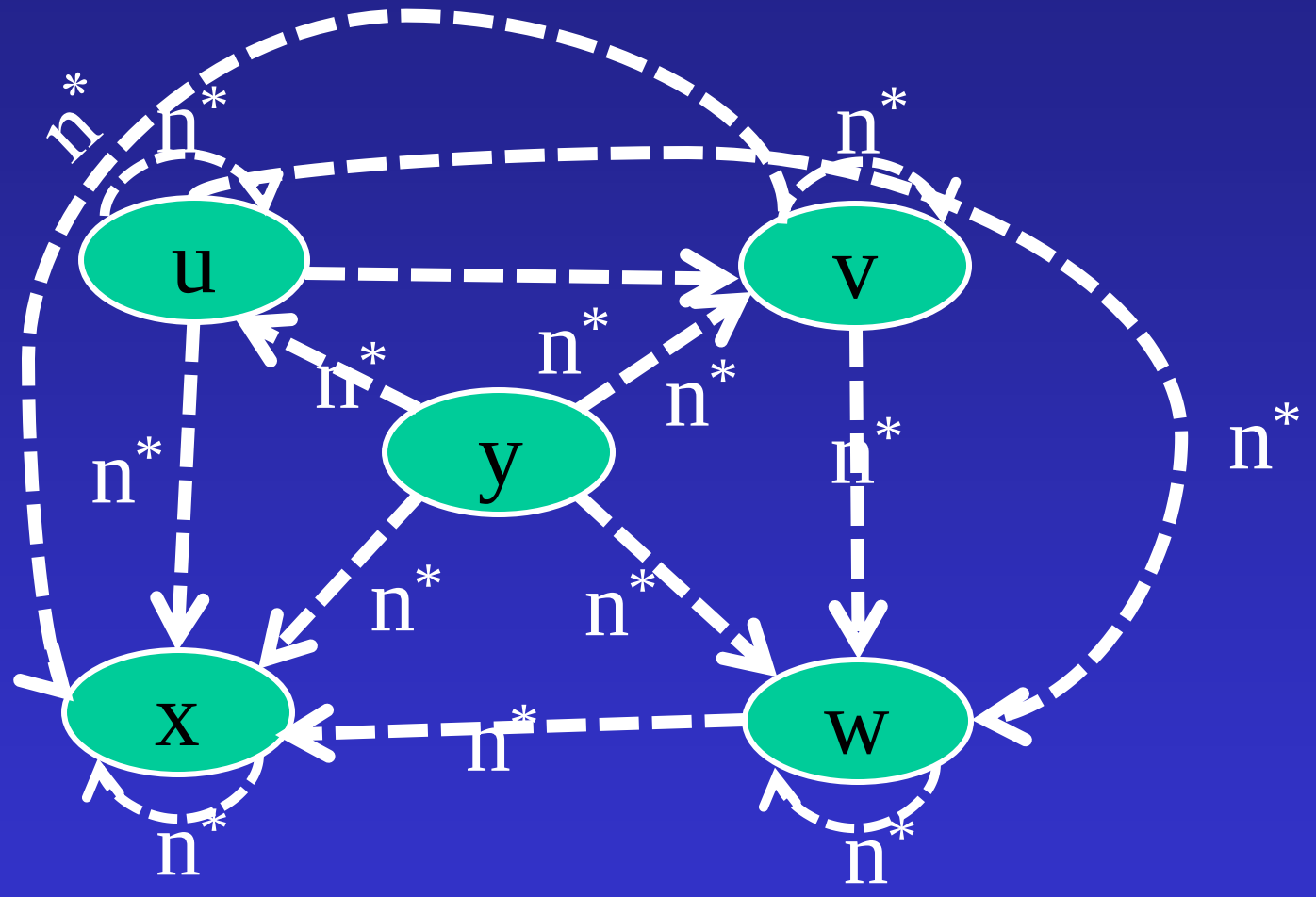
```
void foo() {
  Node l1, l2, t ;
  l1 = null;
  l2 = null ;
  while (*) {
      new(t);       t.next = l1; l1 = t
      new(t);       t.next = l1 ; l2 =t;
       }
while (l1 != null) {
          t = l1.next ;
          assert l2 != null ;
         t = l2.next ;
}
```

Sound & Complete Reasoning about Deterministic Paths [CAV'13, POPL'14]

- $\forall x: n^*(x, x)$ — Reflexivity
- $\forall x, y: n^*(x, y) \wedge n^*(y, z) \Rightarrow n^*(x, z)$ — Transitivity
- $\forall x, y: n^*(x, y) \wedge n^*(y, x) \Rightarrow x = y$ — Acyclicity
- $\forall x, y, z: n^*(x, y) \wedge n^*(x, z) \Rightarrow n^*(y, z) \vee n^*(z, y)$ — Linearity

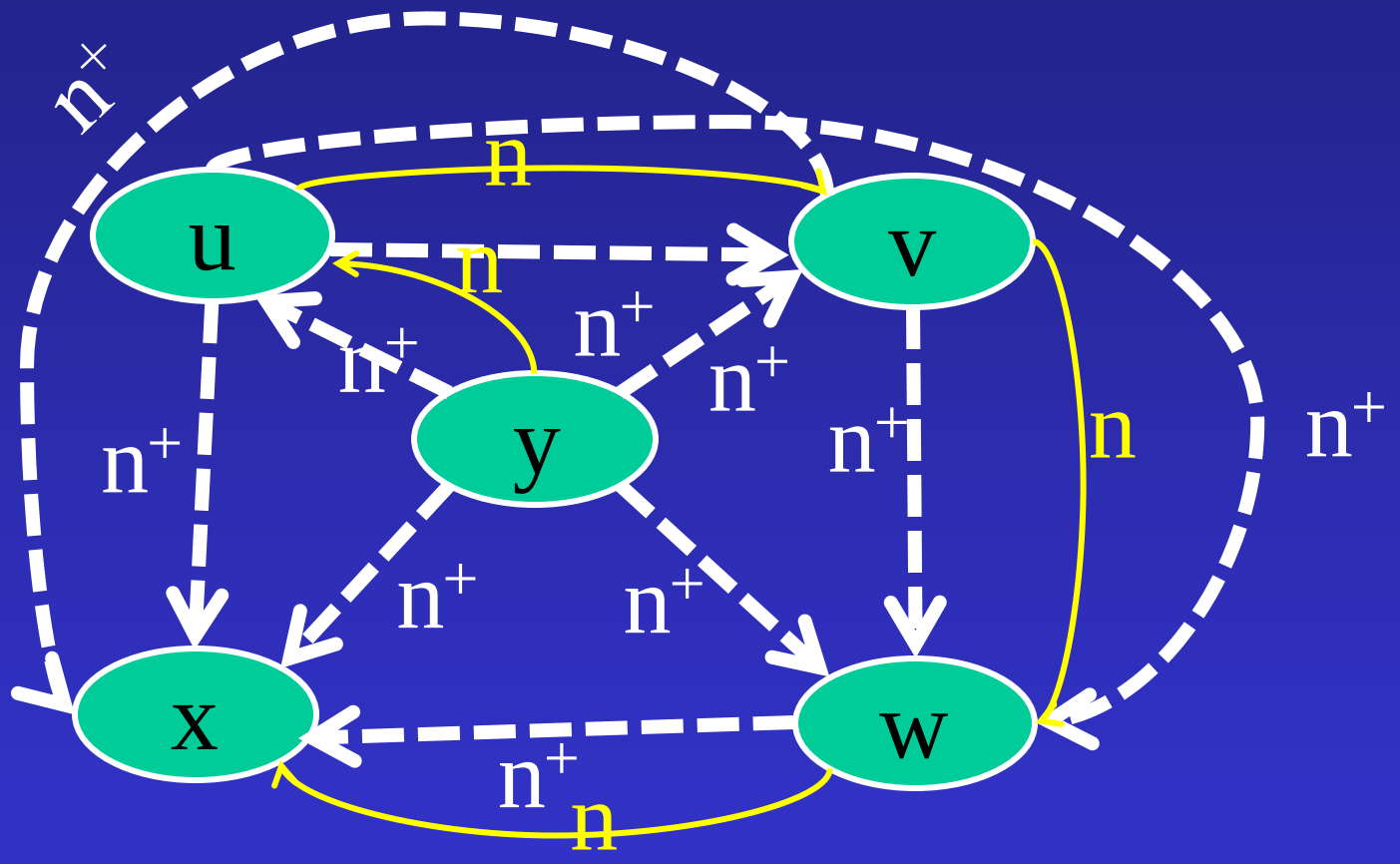
[CAV'13] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, M. Sagiv: Effectively-Propositional Reasoning about Reachability in Linked Data Structures
[POPL'14] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, M. Sagiv: Modular reasoning about heap paths via effectively propositional formulas

Inverting $n^* \Rightarrow n$



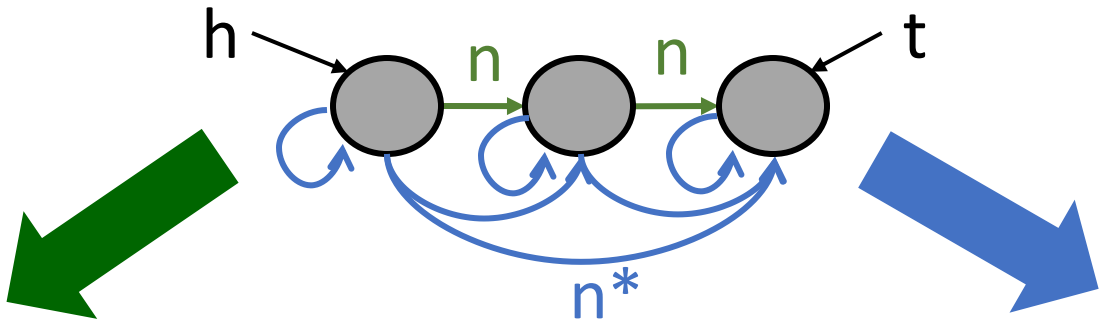
$$n+(\alpha,\beta) \leftrightarrow n^*(\alpha,\beta) \wedge \alpha \neq \beta$$

Inverting $n^* \Rightarrow n$



$$n(\alpha) = \beta \leftrightarrow n+(\alpha,\beta) \wedge \forall\gamma: n+(\alpha,\gamma) \rightarrow n^*(\beta,\gamma)$$

Key idea: representing deterministic paths

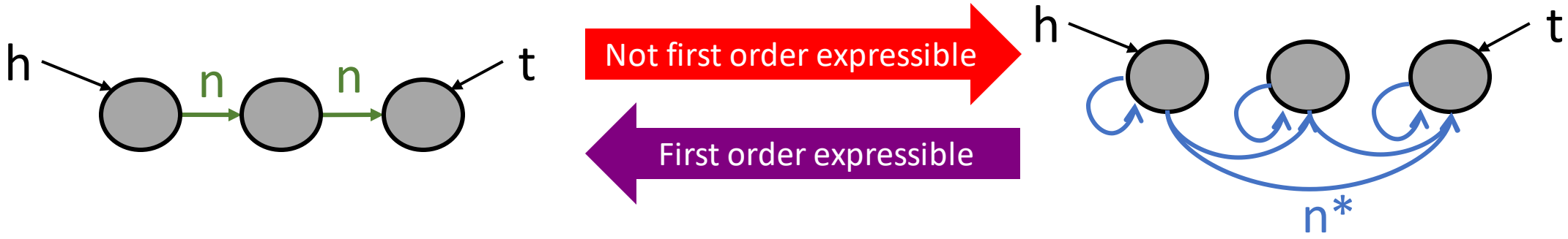


Alternative 1: maintain n

- n^* defined by transitive closure of n
- not definable in first-order logic

Alternative 2: maintain n^*

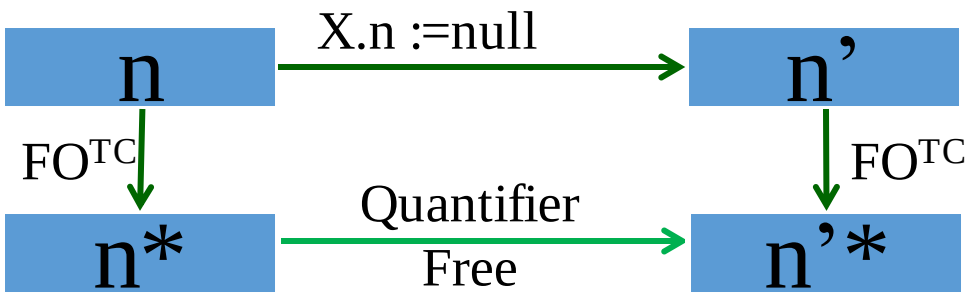
- n defined by transitive reduction of n^*
- Unique due acyclicity and outdegree at most 1
- Definable in first order logic (for roots)
  - $n^+(a,b) \equiv n^*(a, b) \wedge a \neq b$
  - $n(a, b) \equiv n^+(a,b) \wedge \forall z: n^+(a, z) \rightarrow n^*(b, z)$



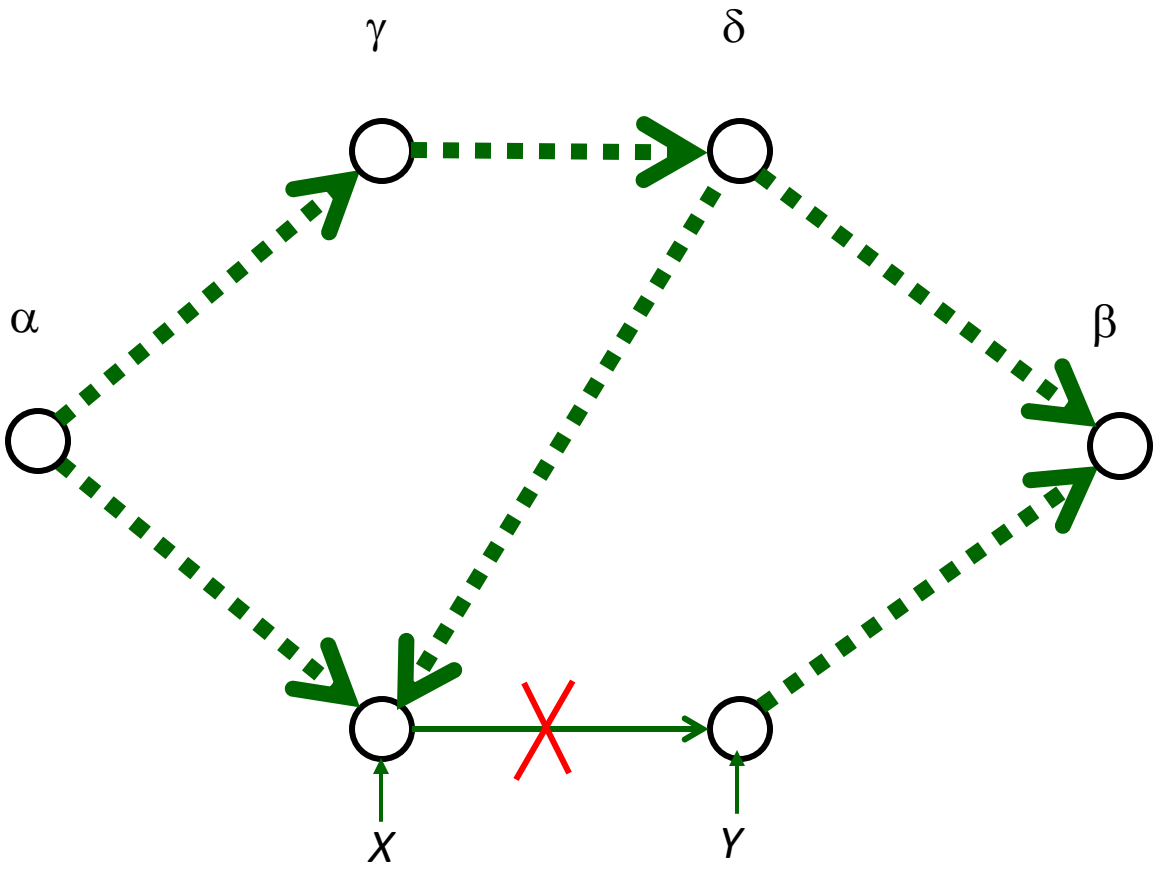
Exploring Locality

- The program updates edge relations
- The compiler generates formulas to update paths
- This can always be done
  - Even in distributed systems
  - Uniform updates
- EPR formulas are closed under WP

Incremental
Simple updates

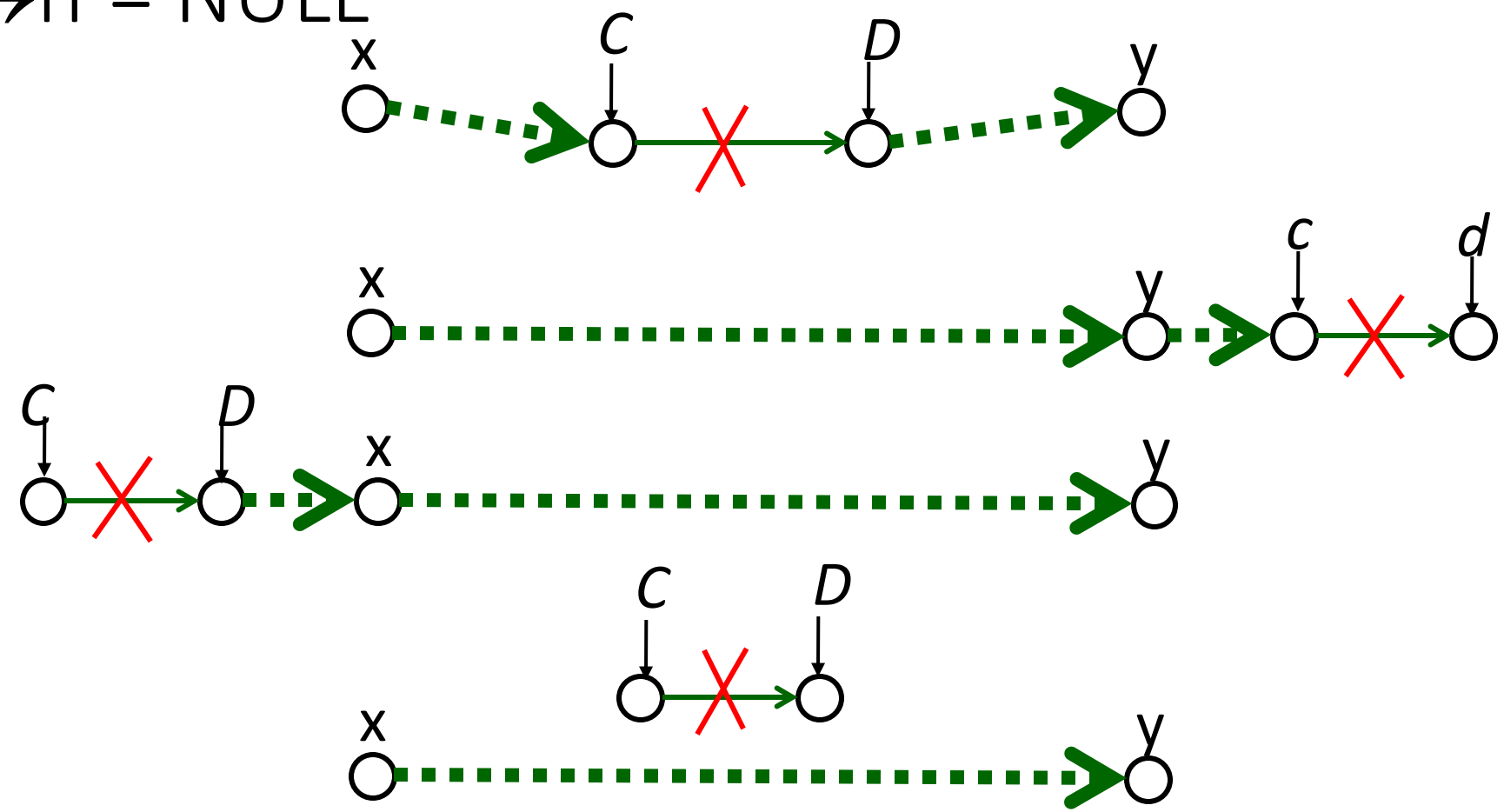


Updating directed reachabilty in general graph is hard



Removing an edge in deterministic graphs (destructive update)

C→n = NULL



$$n'^{*}(x,y) \leftrightarrow n^{*}(x,y) \wedge \neg\, n^{*}(x,C) \wedge n^{+}(C,y)$$

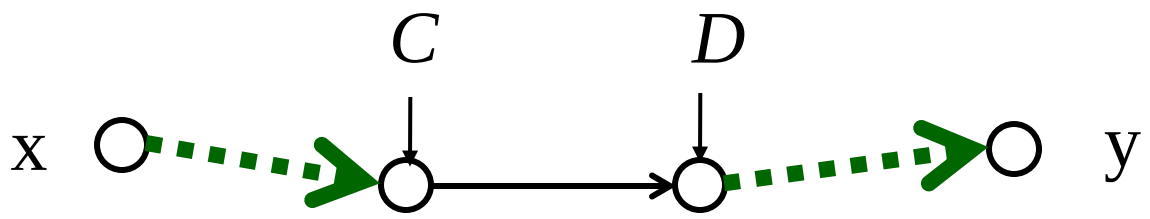
Mutating Single Linked Lists

- wp(C.n := null,Q) = $Q[(n^*(\alpha,\beta)\wedge(\neg n^*(\alpha,C)\vee n^*(\beta,C)) \,/\, n^*(\alpha,\beta)]$
- Can also enforce absence of null dereferences
  $C \neq null$
- Can deal with nested cyclic lists
- But not arbitrary DAGs
  - Requires qunatifyer alternations [Dong&Su 95]

Single Mutation C.n := D (assuming C.n ==null)

- Simple for general graphs
- AF^R for arbitrary data structures
- wp(C.n := D,Q) =
  $Q[(n^*(\alpha,C) \wedge n^*(D,\beta)))/ n^*(\alpha,\beta)]$
- Can also enforce acyclicity $\neg n^*(y,C)$

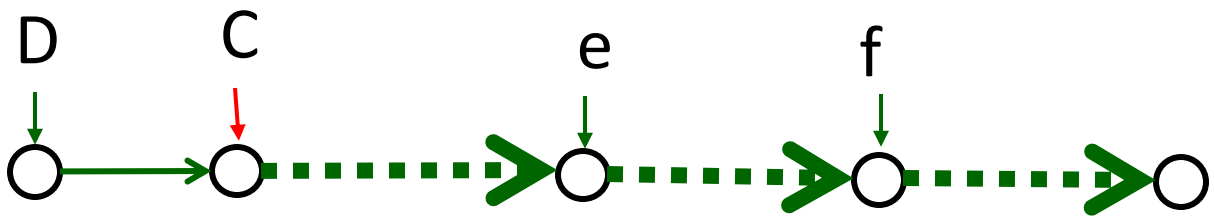
# Adding an edge
C.n= D



assert $\neg n^*(D, C)$

$n'^*(x,y) \leftrightarrow n^*(x,y) \vee (n^*(x, C) \wedge n^*(D,y))$

Traversing an edge
C = D.n (C is fresh)



$n^{+}(D,C) \ \wedge$

$\forall x: \ n^{+}(D,x) \Rightarrow n^{*}(C,x)$

WP Compound statements

- wp (skip,Q) = Q
- wp(X := Y, Q) = Q[Y/ X]
- wp(S_1; S_2,Q) = wp(S_1,(wp(S_2,Q))
- wp(if B then S_1 else S_2, Q) =
  $(B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q))$
- wp(while B do {I} S, Q) = I

VC rules

- $VC_{gen}(\{P\}\ S\ \{Q\}) = P \rightarrow wp(S,Q) \wedge \bigwedge VC_{aux}(S, Q)$
- $VC_{aux}(S, Q) = \{\}$ (for any atomic statement)
- $VC_{aux}(S_1; S_2, Q) =$
  $VC_{aux}(S_1, wp(S_2, Q)) \cup VC_{aux}(S_2, Q)$
- $VC_{aux}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) =$
  $VC_{aux}(S_1, Q) \cup VC_{aux}(S_2, Q)$
- $VC_{aux}(\text{while } B \text{ do } S, Q) = VC_{aux}(S, I) \cup$
  $\{I \wedge [\![B]\!] \rightarrow wp(S,I)\} \cup$
  $\{I \wedge \neg [\![B]\!] \rightarrow Q\}$

But what about pointer traversals?

- Hoare assignment rule goes outside EPR
- wp(X := Y.n,Q) = Q[n(y) / x]
  - Outside EPR
- Reason about list segments
- Coincides with complications in pointer and shape analysis

Pointer Traversals

- Observe that wp is only used positively in VCs (unlike invariants and preconditions)
- Allows EA formulas with reachability
- wp (C := D.n, Q) = $\forall\alpha$: ‘n(D)=α’$\rightarrow$Q[α/C]
  - Replace n with n^* using reachability inversions
- Universal quantifications are also used for allocation C := new()

Benchmark	*Formula Size*						*Solving time* (Z3)
	P,Q		Inv		VC		
	#	∀	#	∀	#	∀	
SLL: reverse	2	2	11	2	133	3	57ms
SLL: filter	5	1	14	1	280	4	39ms
SLL: create	1	0	1	0	36	3	13ms
SLL: delete	5	0	12	1	152	3	23ms
SLL: deleteAll	3	2	7	2	106	3	32ms
SLL: insert	8	1	6	1	178	3	17ms
SLL: find	7	1	7	1	64	3	15ms
SLL: last	3	0	5	0	74	3	15ms
SLL: merge	14	2	31	2	2255	3	226ms
SLL: rotate	6	1	-	-	73	3	22ms
SLL: swap	14	2	-	-	965	5	26ms
DLL: fix	5	2	11	2	121	3	32ms
DLL: splice	10	2	-	-	167	4	27ms

Disproving with SAT

Benchmark	*Nature of defect*	*Formula Size*						*Solving time*	*C.e. Size*
		P,Q		I		VC			
		#	∀	#	∀	#	∀	(Z3)	(vertices)
SLL: find	null pointer dereference	7	1	7	1	64	3	18ms	2
SLL: deleteAll	Loop invariant in annotation is too weak to prove the desired property	3	2	5	2	68	3	58ms	5
SLL: rotate	Transient cycle introduced during execution	6	1	-	-	109	3	25ms	3
SLL: insert	Unhandled corner case when an element with the same value already exists in the list --- ordering violated	8	1	6	1	178	3	33ms	4

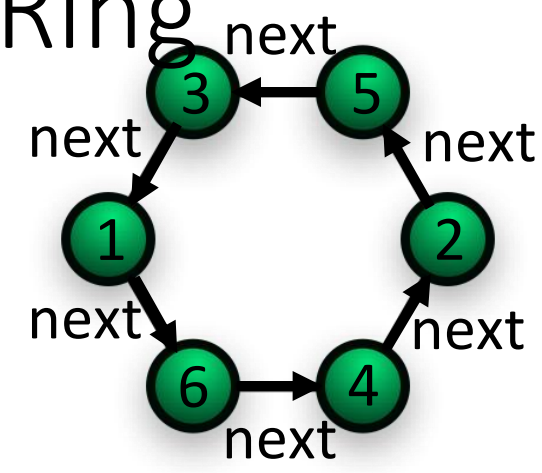
Open Questions

- What kind of data structure manipulations cannot be expressed in EPR?
  - Correlated list length
- Can we use a more limited logic?

From linked lists to distributed protocols

- Represent the global state of the system using labeled directed hypergraph
- The user specifies global invariants
- Each of the actions can be executed non-deterministically

Example: Leader Election in a Ring



- Nodes are organized in a ring
- Each node has a unique numeric id
- Protocol:
  - Each node sends its id to the next
  - A node that receives a message passes it (to the next) if the id in the message is higher than the node's own id
  - A node that receives its own id becomes the leader
- Theorem:
  - The protocol selects at most one leader

[CACM'79] E. Chang and R. Roberts. *An improved algorithm for decentralized extrema-finding in circular configurations of processes*

Inductive Invariant for Leader Election

- $\leqslant$ (ID, ID) – total order on node id's
- **btw** (Node, Node, Node) – the ring topology
- **id**: Node → ID – relate a node to its id
- **pending**(ID, Node) – pending messages
- **leader**(Node) – leader(n) means n is the leader

Safety property: I_0

$$I_0 = \forall x, y: \text{Node}.\ \textbf{leader}(x) \wedge \textbf{leader}(y) \Rightarrow x = y$$

Inductive invariant: $\text{Inv} = I_0 \wedge I_1 \wedge I_2 \wedge I_3$

$$I_1 = \forall n_1, n_2: \text{Node}.\ \textbf{leader}(n_2) \wedge n_1 \neq n_2 \Rightarrow id[n_1] \leqslant id[n_2]$$

$$I_2 = \forall n_1, n_2: \text{Node}.\ pending(id[n_2], n_2) \wedge n_1 \neq n_2 \Rightarrow id[n_1] \leqslant id[n_2]$$

$$I_3 = \forall n_1, n_2, n_3: \text{Node}.\ btw(n_1, n_2, n_3) \wedge pending(id[n_2], n_1) \Rightarrow id[n_3] \leqslant id[n_2]$$

How can we find an inductive invariant without knowing it?

Simple invariants for proving the safety of distributed protocols

Oded Padon, Kenneth McMillan, Aurojit Panda, Sharon Shoham




TEL AVIV UNIVERSITY

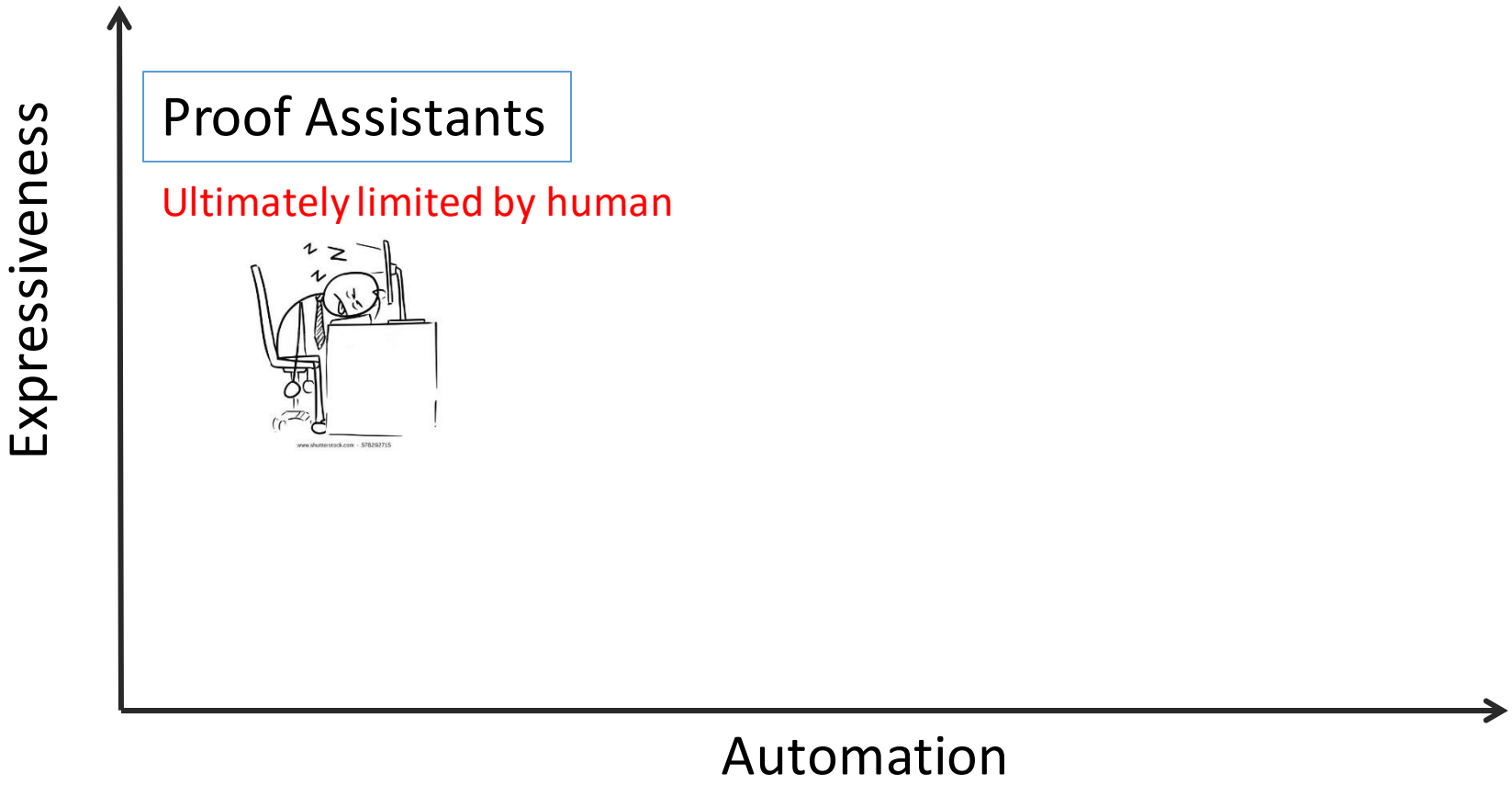
Microsoft Research

Berkeley UNIVERSITY OF CALIFORNIA

TEL AVIV UNIVERSITY

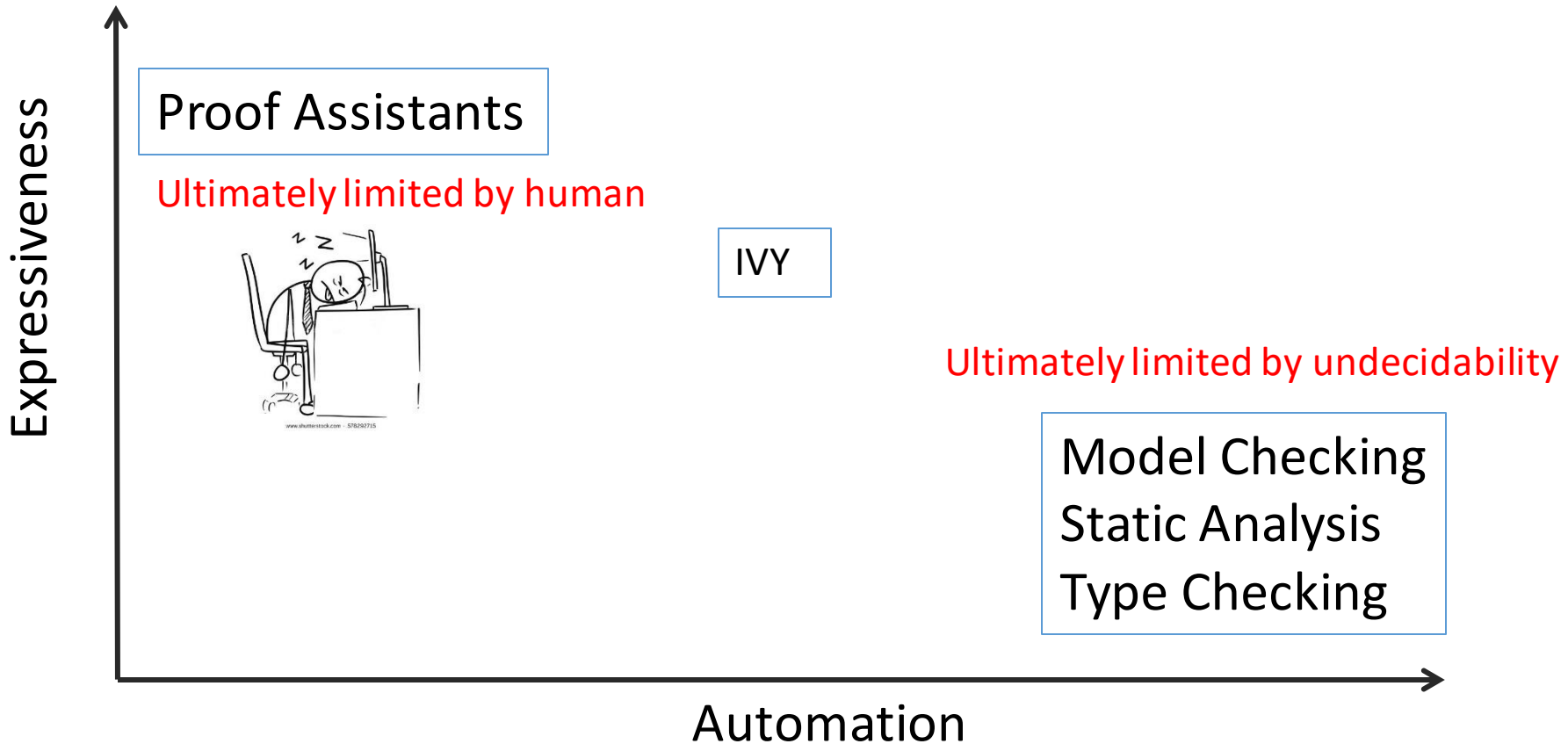
http://microsoft.github.io/ivy/

State of the art in formal verification



"the proofs consisted of about 5000 lines and assumed several nontrivial invariants of the Raft protocol. This paper discusses the verification of Raft as a whole, including all the invariants from the original Raft paper [32]. These new proofs consist of about 45000 additional lines" [Verdi, CPP'16]

State of the art in formal verification



"but our input language cannot compete in generality with mechanized proof methods that rely heavily on human expertise, e.g., IVY [55], Verdi [68], IronFleet [38], TLAPS [16]" [Konnov et al, POPL'17]

IVY's Principles

- Express the program transition system and the safety property using limited first order logic
  - Logic is mostly hidden
  - Deduction is decidable
  - Display Counter Examples to Inductions graphically
- Assist the user in writing inductive invariants

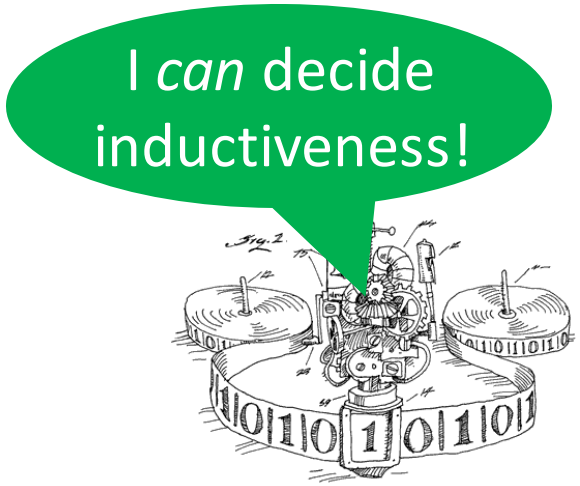
Effectively Propositional Logic – EPR

a.k.a. Bernays-Schönfinkel-Ramsey class

- Limited fragment of first-order logic
  - Restricted quantifier prefix: $\exists^* \forall^* \phi_{Q.F.}$
  - No $\forall^* \exists^*$
  - No function symbols
    - Possible to add stratified function symbols
  - No arithmetic
- Small model property
  - $\exists x_1,\ldots, x_n.\forall y_1,\ldots,y_m.\phi_{Q.F.}$ has a model iff it has a model of at most n+k elements (k - number of constant symbols)
- Satisfiability is decidable
  - NEXPTIME / Σ_2
- Supported by theorem provers (e.g., Z3, iProver, Vampire)

F. Ramsey. *On a problem in formal logic.* Proc. London Math. Soc. 1930

Relational Modeling Language (RML)



- Designed to make verification tasks decidable (EPR)
  - Yet expressive enough to model systems
- Turing-Complete
- Universally quantified inductive invariants are decidable to check
- System state described by finite (unbounded) relations
- No numerics
- Simple (quantifier-free) updates
- Universally quantified axioms (domain specific)
  - Total orders, partial orders, lists, trees, rings, quorums, …

Languages and verification

Language	Executable	Expressiveness	Inductiveness
C, Java, Python…	☑	Turing-Complete	Undecidable
SMV	☒	Finite-state	Temporal Properties
Ivy	C++	Turing-Complete	Decidable(EPR)
Dafny	☑	Turing-Complete	Undecidable with lemmas
Coq, Isabelle/HOL	☑	Turing-Complete	Manual with tactics
TLA+	☒	Turing-Complete	Manual

Leader Election Protocol (Ivy)

- ⩽ (ID, ID) – total order on node id's
- **btw** (Node, Node, Node) – the ring topology
- **id**: Node → ID – relate a node to its unique id
- **pending**(ID, Node) – pending messages
- **leader**(Node) – leader(n) means n is the leader

```
action send(n: Node) = {
  "s := next(n)";
  pending(id(n),s) := true
}
```

```
action receive(n: Node, m: ID) = {
  requires pending(m, n);
  pending(m, n) := false;
  if id(n) = m then
    // found leader
    leader(n) := true
  else if id(n) ⩽  m then
    // pass message
    "s := next(n)";
    pending(m, s) := true
}
```

```
protocol = (send | receive)*
```

```
assert I0 = ∀ x,y: Node. leader(x)∧leader(y) → x = y
```

Inductive Invariant for Leader Election

- $\leqslant$ (ID, ID) – total order on node id's
- **btw** (Node, Node, Node) – the ring topology
- **id**: Node → ID – relate a node to its id
- **pending**(ID, Node) – pending messages
- **leader**(Node) – leader(n) means n is the leader

Safety property: I_0

$$I_0 = \forall x, y: \text{Node}.\ \textbf{leader}(x) \wedge \textbf{leader}(y) \Rightarrow x = y$$

Inductive invariant: $\text{Inv} = I_0 \wedge I_1 \wedge I_2 \wedge I_3$

$$I_1 = \forall n_1, n_2: \text{Node}.\ \textbf{leader}(n_2) \wedge n_1 \neq n_2 \Rightarrow id[n_1] \leqslant id[n_2]$$

The leader has the highest ID

$$I_2 = \forall n_1, n_2: \text{Node}.\ pending(id[n_2], n_2) \wedge n_1 \neq n_2 \Rightarrow id[n_1] \leqslant id[n_2]$$

Only the leader can be self-pending

$$I_3 = \forall n_1, n_2, n_3: \text{Node}.\ btw(n_1, n_2, n_3) \wedge pending(id[n_2], n_1) \Rightarrow id[n_3] \leqslant id[n_2]$$

Cannot bypass higher nodes

How can we find an inductive invariant without knowing it?

Verified Protocols

Protocol	Model (# LOC)	Property (# Literals)	Invariant (# Literals)
Leader in Ring	59	3	12
Learning Switch	50	11	18
DB Chain Replication	143	11	35
Chord	155	35	46
Lock Server (500 Coq lines [Verdi])	122	3	21
Distributed Lock (1 week [IronFleet])	41	3	26
Single Decree Paxos	85	3	32
Multi Paxos	102	3	38
Vertical Paxos	123	3	65
Fast Paxos	117	3	59
Flexible Paxos	88	3	32
Virtually Synchronous Paxos	Work in progress		
Practical Byzantine Fault Tolerance			

Consensus [OOPSLA'17]

Protocol	Model [LOC]	Invariant [Conjectures]	EPR [sec] μ	EPR [sec] σ	RW [sec]
Paxos	85	11	1.0	0.1	1.2
Multi-Paxos	98	12	1.2	0.1	1.4
Vertical Paxos*	123	18	2.2	0.2	-
Fast Paxos*	117	17	4.7	1.6	1.5
Flexible Paxos	88	11	1.0	0	1.2
Stoppable Paxos*	132	16	3.8	0.9	1.6

*first mechanized verification

Transformation to EPR reusable across all variants!

Why do people hate First Order Logic?

- Hard to understand and error prone
  - Nested quantifiers and negations
- Too weak: Cannot express
  - Parity
  - Numeric
  - Quorums
  - Finiteness
  - Paths in a graph
- Hard for automation
  - Satisfiability is undecidable
  - NP-complete for fixed size

Our Solution

- Limited EPR formulas $\exists^*\forall^*$
  - Finite model guaranteed to exit
  - Display graphs graphically
- Define axioms per of domain of programs
  - Total orders
  - Paths in deterministic graphs
- Use a Turing complete programming language for updates

- *Satisfiability is NEXPTIME complete/Σ_2*
  - *Support from Z3, Iprover, Vampire*
- Rely on user provided loop invariants or sound inferred invariants

Main contributions

- Showing the linked lists can be verified in a sound and complete way using EPR solver [CAV'13, POPL'14]
- Characterizing the absence of universal invariants relaxed traces [CAV'15, JACM]
- Characterizing the decidability of inferring universal invariants [POPL'16]
- A system for interactively inferring universal invariants [PLDI'16]

[PLDI'16] O. Padon, K.L. McMillan, A. Panda, M. Sagiv, S. Shoham:
Ivy: safety verification by interactive generalization.
[POPL'16] O. Padon, N. Immerman, A. Karbyshev, S. Shoham, M. Sagiv Decidability of inferring inductive invariants
[POPL'14] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, M. Sagiv: Modular reasoning about heap paths via effectively propositional formulas
[CAV'15, JACM] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, S. Shoham:
Property-directed inference of universal invariants or proving their absence
[CAV'13] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, M. Sagiv: Effectively-Propositional Reasoning about Reachability in Linked Data Structures