Workshop in Verification of Distributed Protocols

Mooly Sagiv, Oded Padon

08-March-2018

http://www.cs.tau.ac.il/~odedp/workshop18/

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

Administration

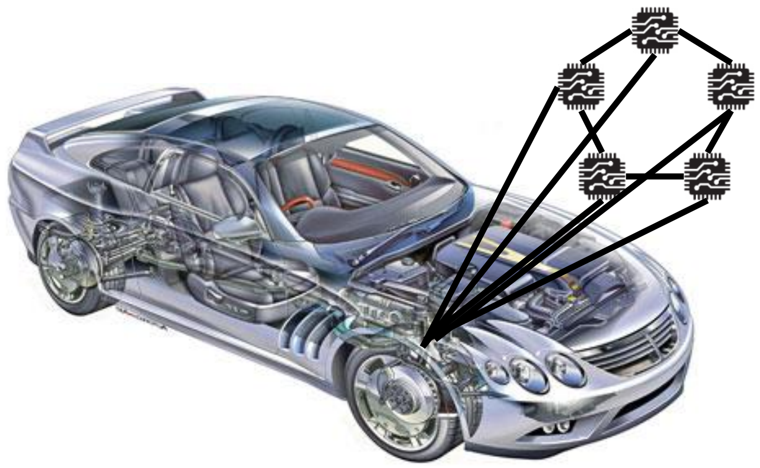
- Start-off meeting (today)
- Project teams:
  - 2-3 students
  - Each team will take different a project, and work independently during the semester
  - Meet with Oded / Mooly as needed
- If needed, we'll have more workshop meeting during the semester
- 14/6 – project presentation meeting
  - Each team will present project
  - Project must be finished and approved by Oded / Mooly before

Possible Projects

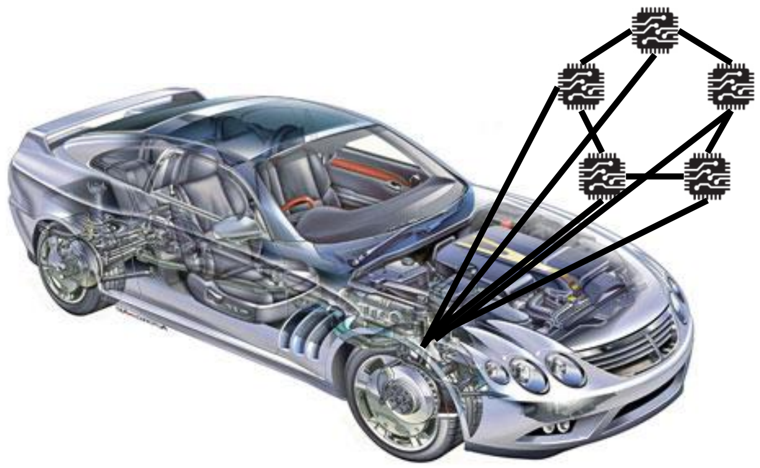
- Use Ivy to verify any distributed / shared memory algorithm
- Paxos variants
  - Disk Paxos, Generalised Paxos, EPaxos (see http://paxos.systems/variants.html for ideas)
  - Prove reconfiguration / failure recovery / log truncation / liveness
- Mutual Exclusion Algorithms
  - Knuth's Algorithm, Lamport's Bakery, Patterson, …
  - Prove safety and liveness
- Blockchain algorithms
  - Algorand, HoneyBadgerBFT, Bitcoin-NG, …
- Improve Ivy
  - Experiment with other SMT solvers (e.g. iProver, CVC4, Vampire, SPASS)

Why verify distributed protocols?

- Distributed systems are everywhere
  - Safety-critical systems
  - Cloud infrastructure
  - Blockchain
- Distributed systems are notoriously hard to get right
  - Even small protocols can be tricky
  - Bugs occur on rare scenarios
  - Testing is costly and not sufficient



Why verify distributed protocols?



- Distributed systems are everywhere
  - Safety-critical systems
  - Cloud infrastructure
  - Blockchain

- Distributed systems are notoriously hard to get right

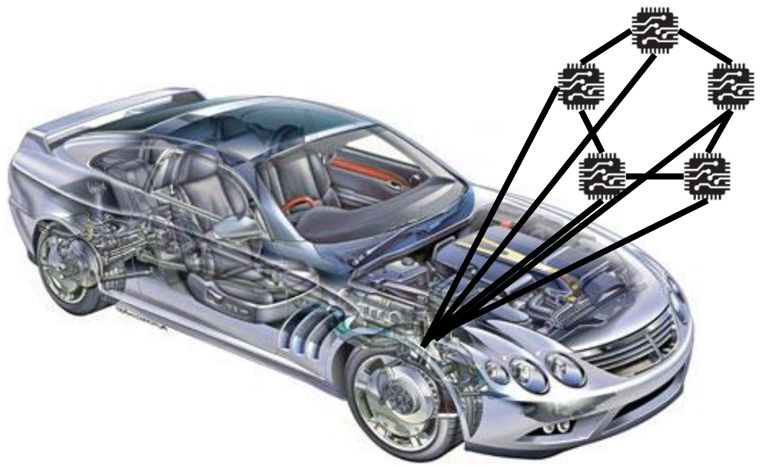
SIGCOMM'01

Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, *Member, IEEE*

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to func-

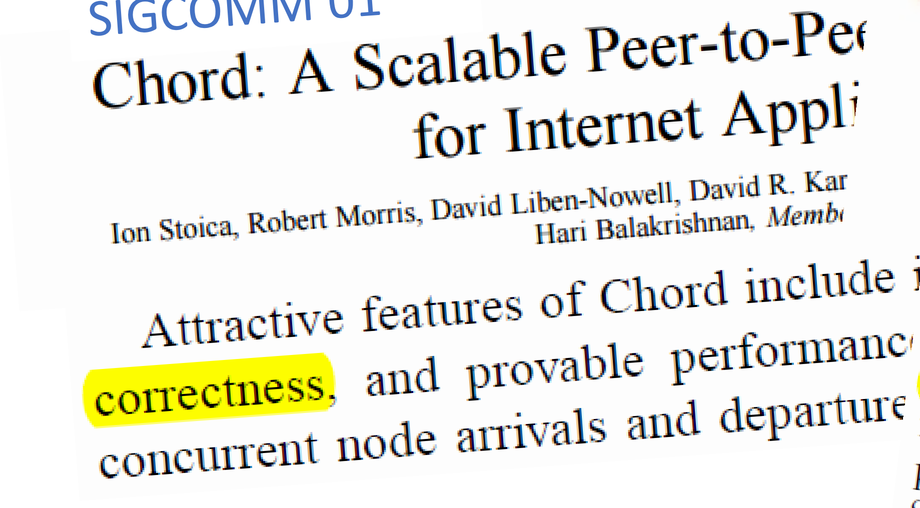
Why verify distributed protocols?



- Distributed systems are everywhere
  - Safety-critical systems
  - Cloud infrastructure
  - Blockchain

- Distributed systems are notoriously hard to get right

SIGCOMM'01



Chord: A Scalable Peer-to-Pee[...]
for Internet Appli[...]

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Kar[...]
Hari Balakrishnan, Memb[...]

Attractive features of Chord include [...]
correctness, and provable performanc[...]
concurrent node arrivals and departure[...]

CCR'12

Using Lightweight Modeling To Understand Chord

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey USA
pamela@research.att.com

Under the same assumptions made in the Chord papers, the [SIGCOMM] version of the protocol is not correct, and not one of the properties claimed invariant in [PODC] is actually invariantly true of it. The [PODC] version satisfies one invariant, but is still not correct. [...]
presented by means of [...]

SOSP'07 Best Paper Award

Zyzzyva: Speculative Byzantine Fault Tolerance

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong
Dept. of Computer Sciences
University of Texas at Austin

Zyzzyva is a state machine replication protocol based on ... protocols: (1) agreement, (2) view change, and (3) ... *agreement* protocol orders requests for exe- ... *view change* protocol coordinates ...

CACM'08

Zyzzyva: Sp... Fault ...

ACM Transactions on Computer Systems '09

Zyzzyva: Speculative Byzantine Fault Tolerance

RAMAKRISHNA KOTLA
Microsoft Research, Silicon Valley
and
LORENZO ALVISI, MIKE DAHLIN, ALLEN CLEMENT, and EDMUND WONG
The University of Texas at Austin

arXiv:1712.01367v1 [cs.DC] 4 Dec 2017

Revisiting Fast Practical Byzantine Fault Tolerance

Ittai Abraham, Guy Gueta, Dahlia Malkhi
VMware Research

with:
Lorenzo Alvisi (Cornell),
Rama Kotla (Amazon),
Jean-Philippe Martin (Verily)

We now proceed to demonstrate that the view-change mechanism in Zyzzyva does not guarantee safety.

Proving distributed systems is hard

- Amazon [CACM'15] uses TLA+ for testing protocols, but no proofs
- IronFleet [SOSP'15] – verification of Multi-Paxos in Dafny (3.7 person-years)
- Verdi [PLDI'15] – verification of Raft in Coq (50,000 lines of proofs)

Our goal: reduce human effort while maintaining flexibility

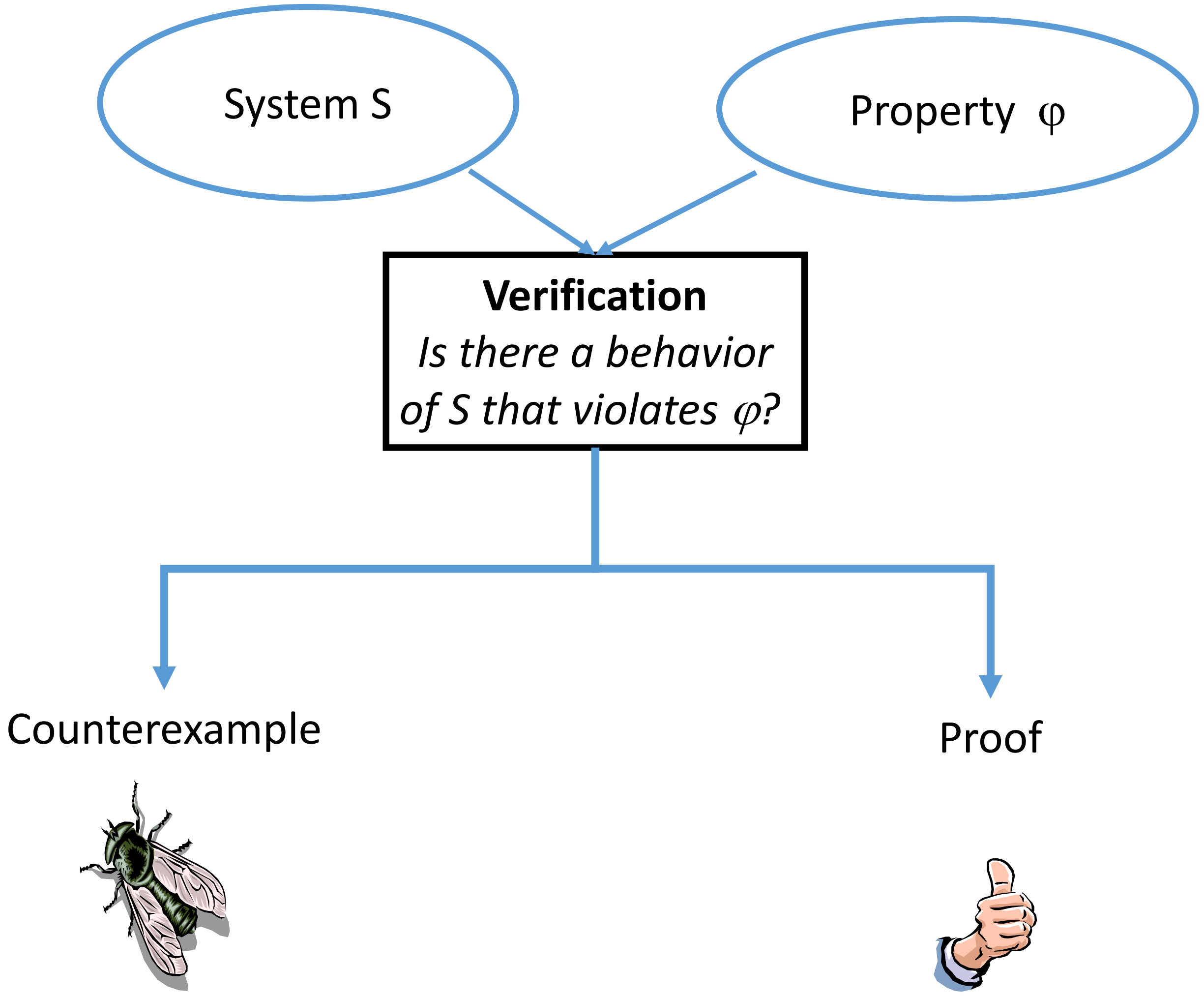
Our approach: decompose verification into decidable problems

[CACM'15] Newcombe et al. How Amazon Web Services Uses Formal Methods

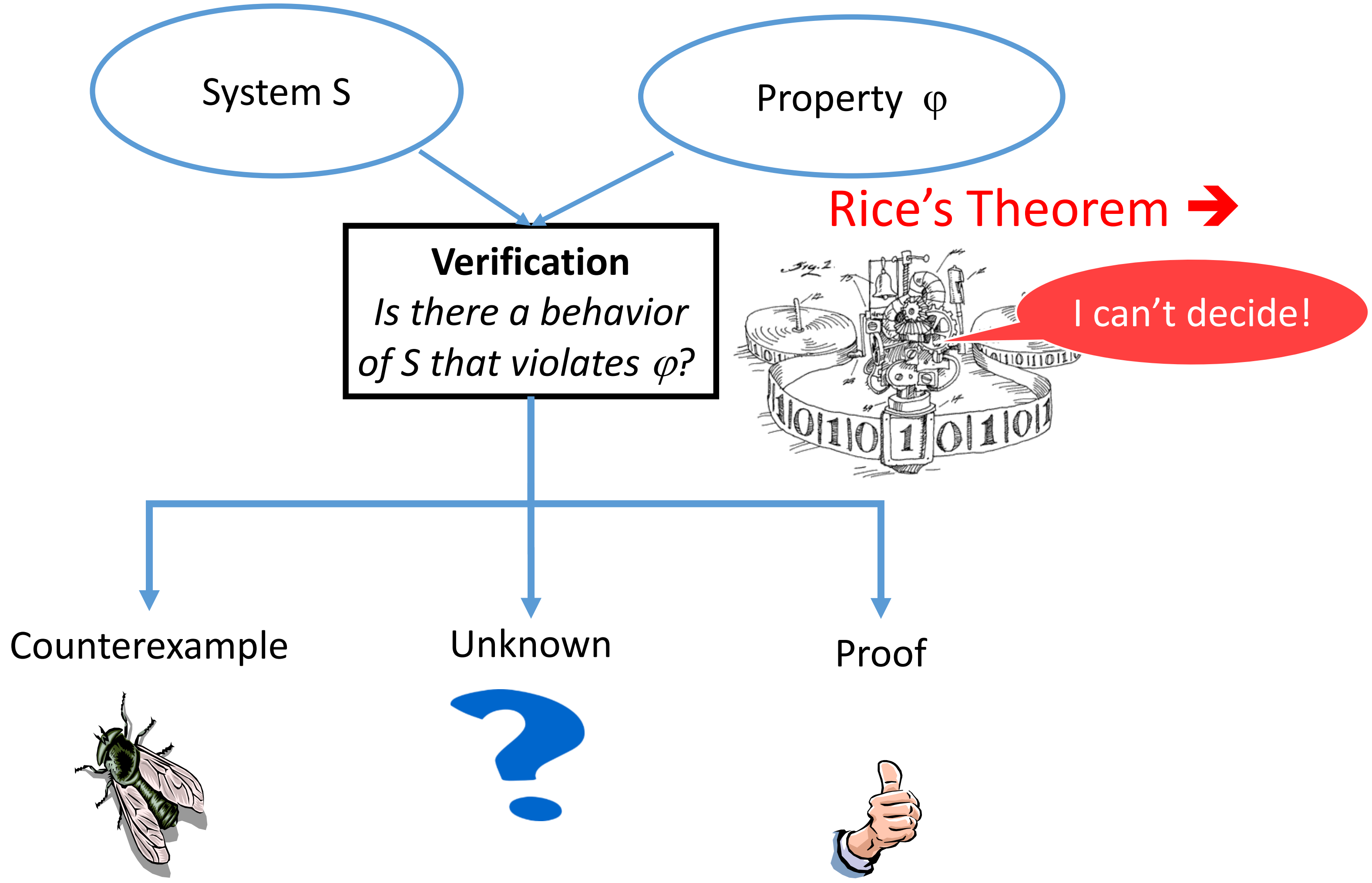
[SOSP'15] Hawblitzel et al. IronFleet: proving practical distributed systems correct

[PLDI'15] Wilcox et al. Verdi: a framework for implementing and formally verifying distributed systems

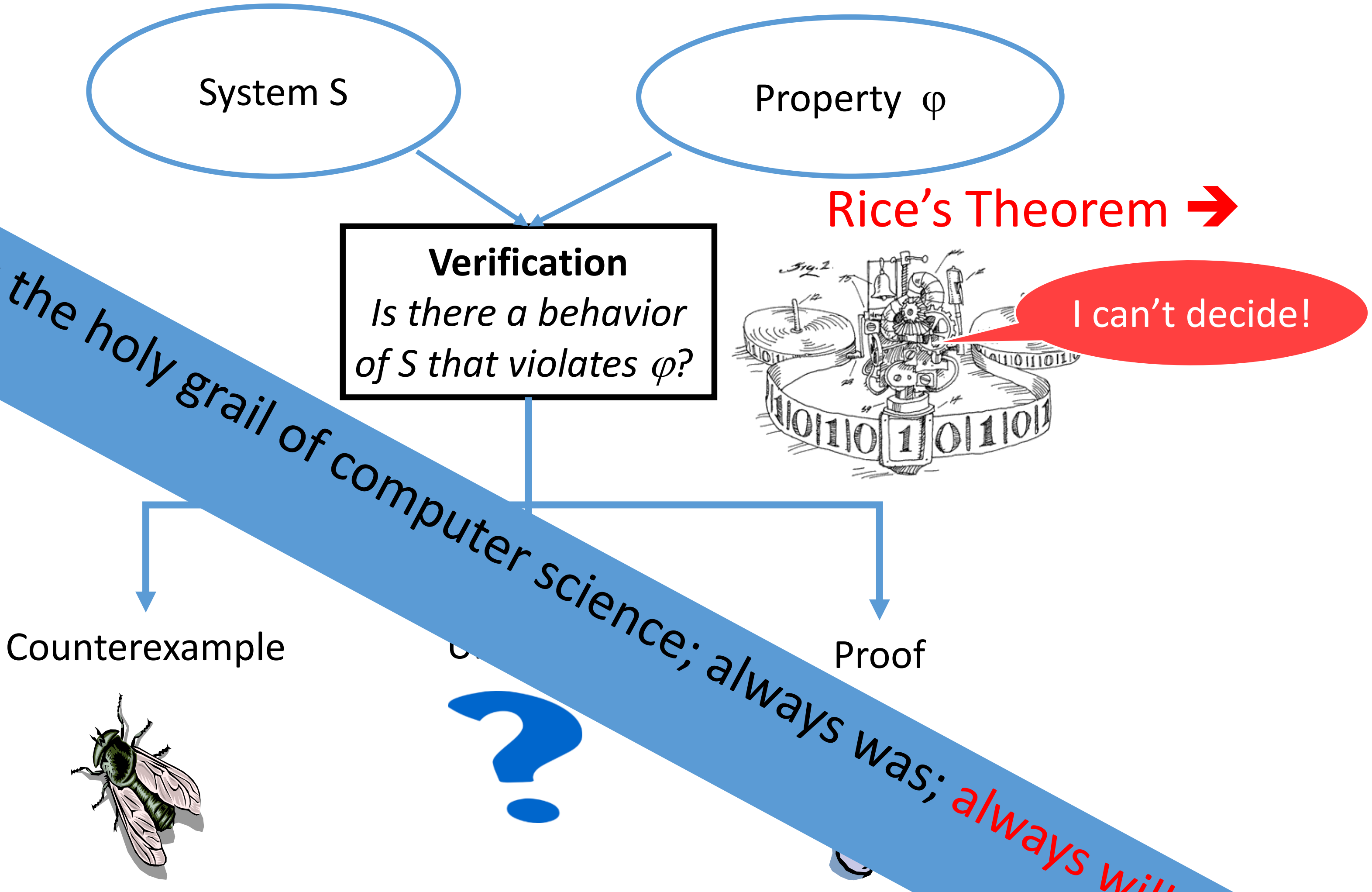
Automatic verification of infinite-state systems



Automatic verification of infinite-state systems

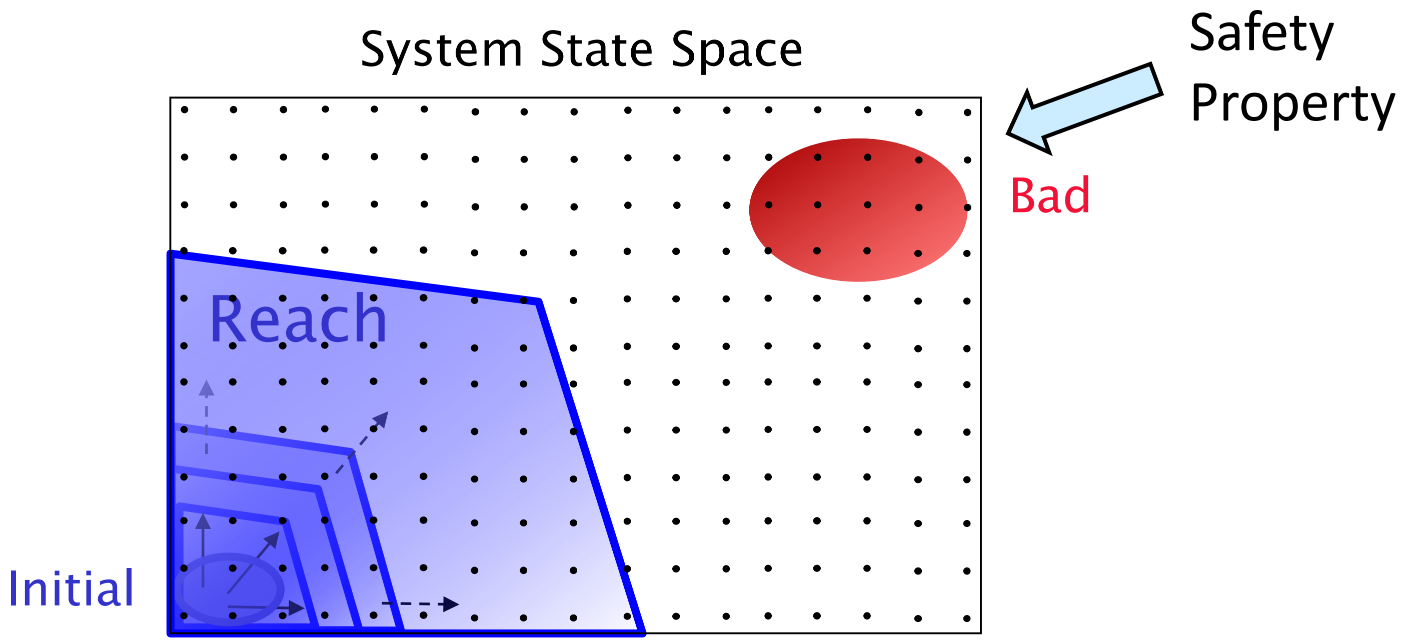


Automatic verification of infinite-state systems



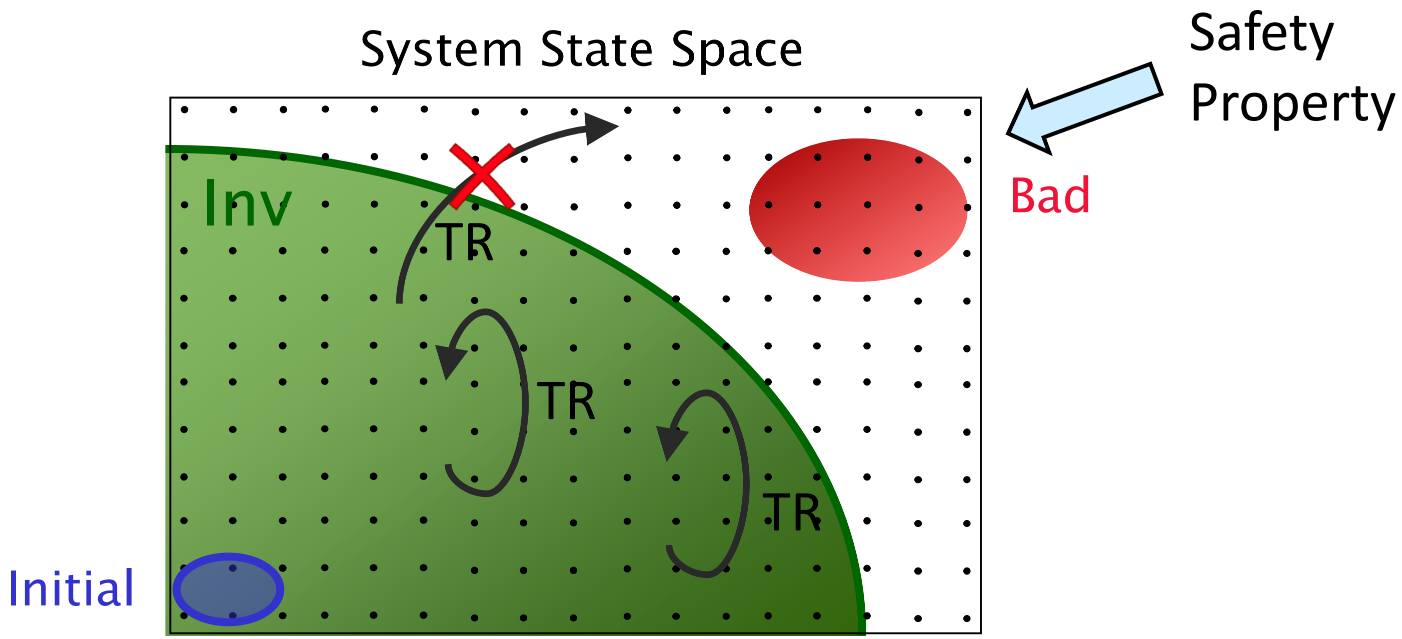
"Program verification is the holy grail of computer science; always was; always will be"
Chet Murthy

Inductive invariants



System S is **safe** if all the reachable states satisfy the property P = ¬Bad

Inductive invariants



System S is **safe** if all the reachable states satisfy the property P = ¬Bad

System S is safe iff there exists an **inductive invariant** Inv :

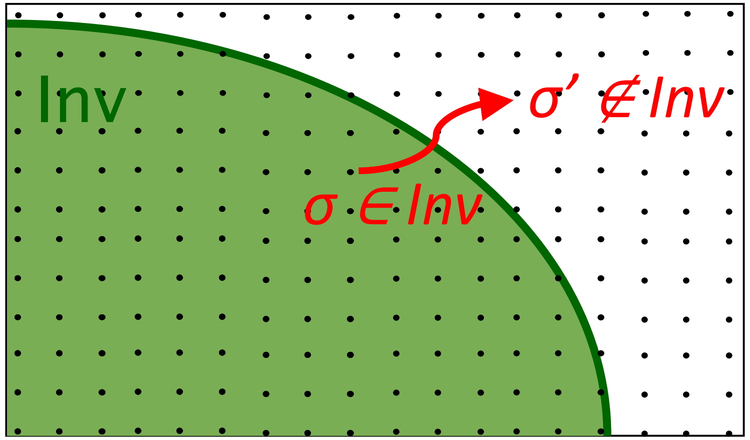
$Inv \cap Bad = \emptyset$ *(Safety)*

$Init \subseteq Inv$ *(Initiation)*

if $\sigma \in Inv$ *and* $\sigma \rightarrow \sigma'$ *then* $\sigma' \in Inv$ *(Consecution)*

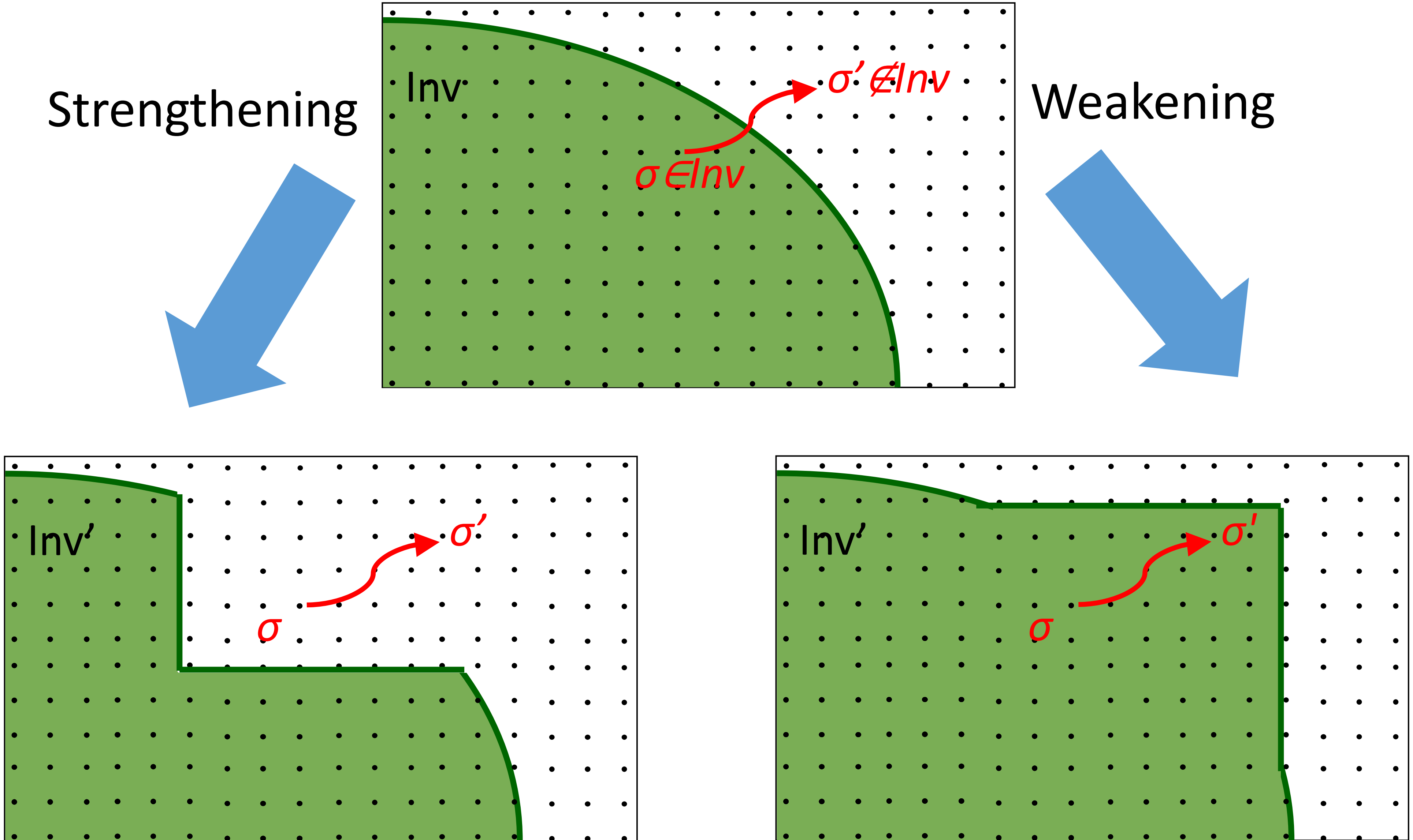
Counterexample To Induction (CTI)

- States σ,σ' are a CTI of Inv if:
- σ ∈ Inv
- σ' ∉ Inv
- σ → σ'



- A CTI may indicate:
  - A bug in the system
  - A bug in the safety property
  - A bug in the inductive invariant
    - Too weak
    - Too strong

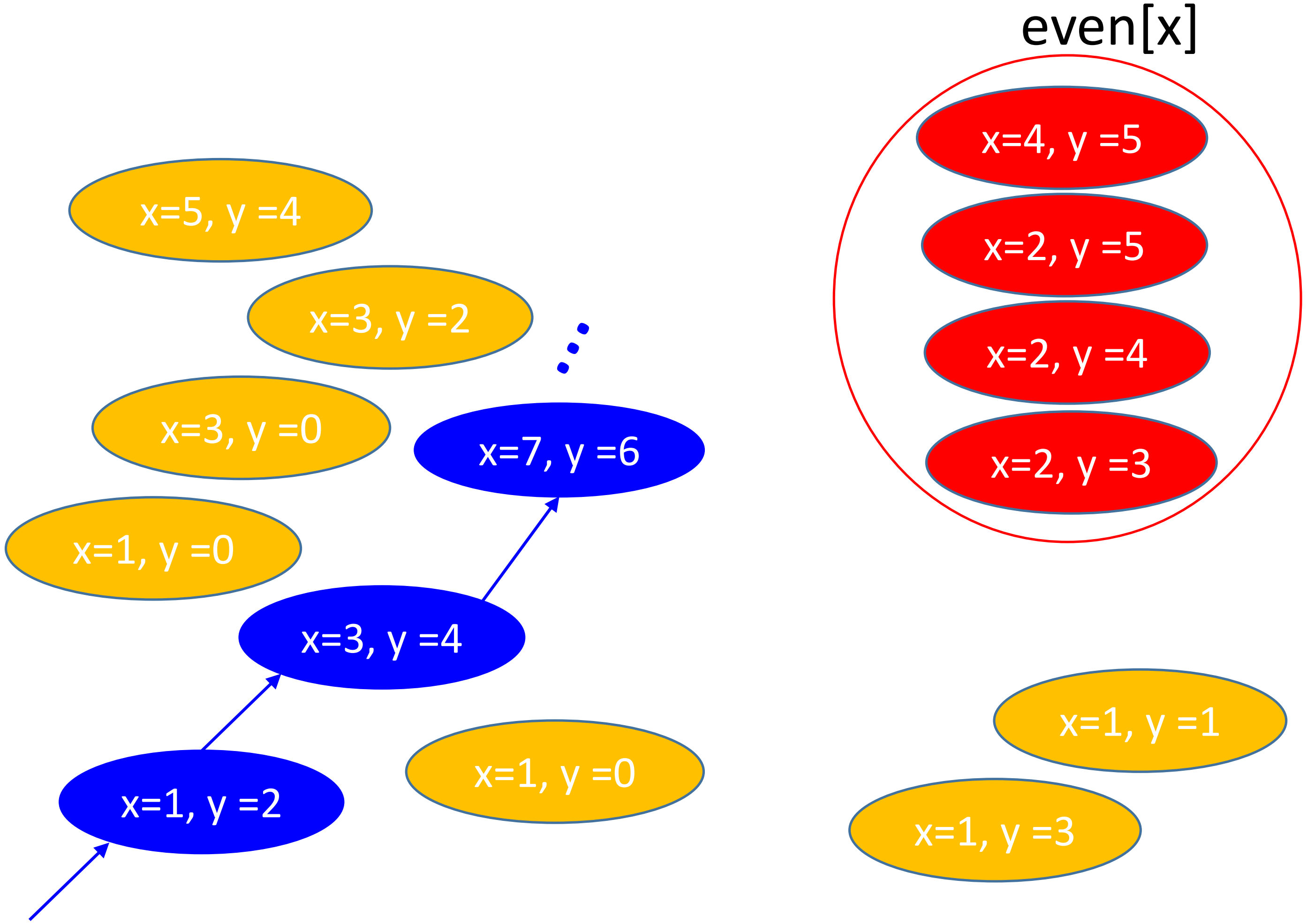
Strengthening & weakening from CTI



Simple example: loop invariants

```
x := 1;
y := 2;
while * do {
 assert ¬even[x];
 x := x + y;
 y := y + 2;
}
```

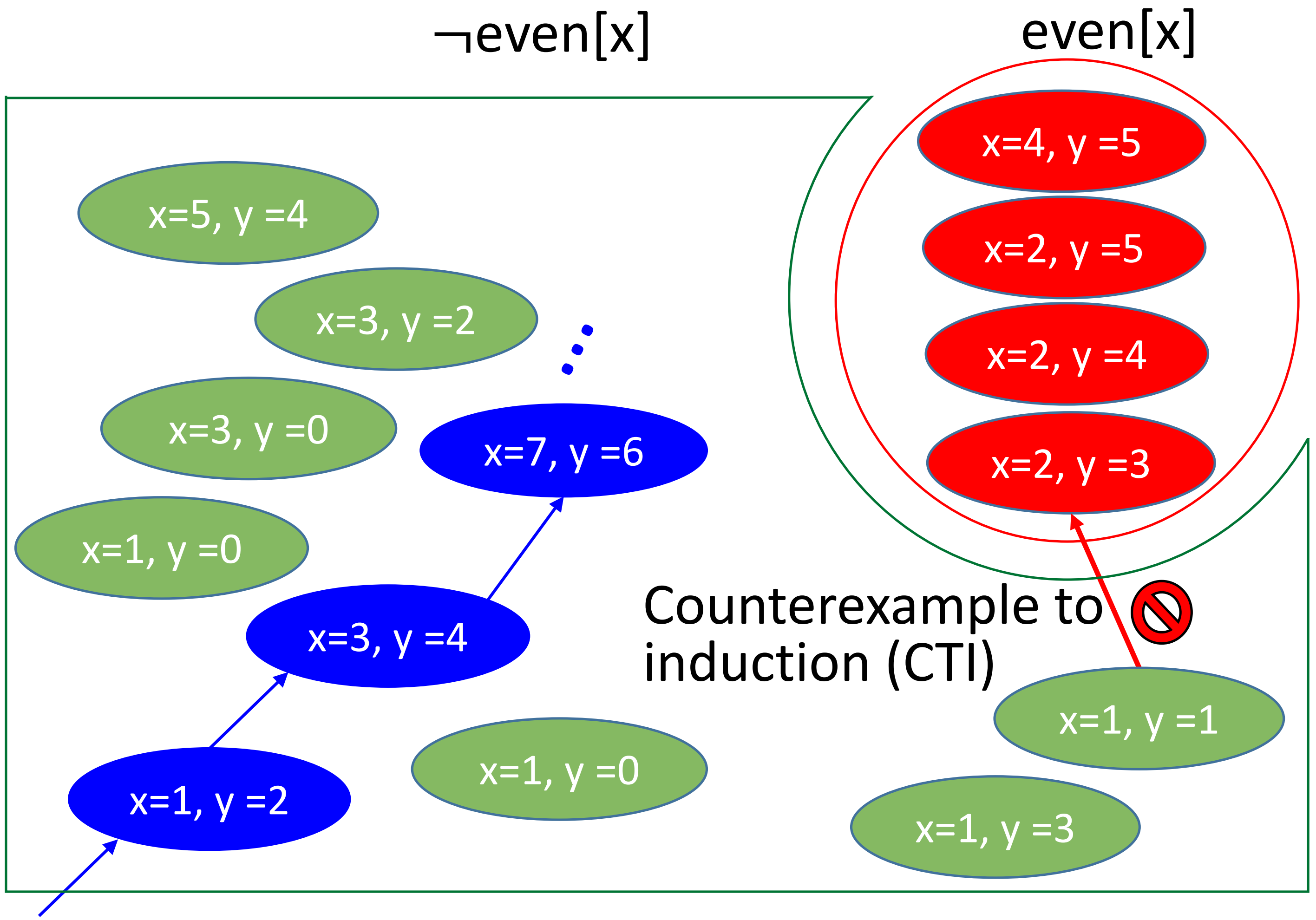
TR



Simple example: loop invariants

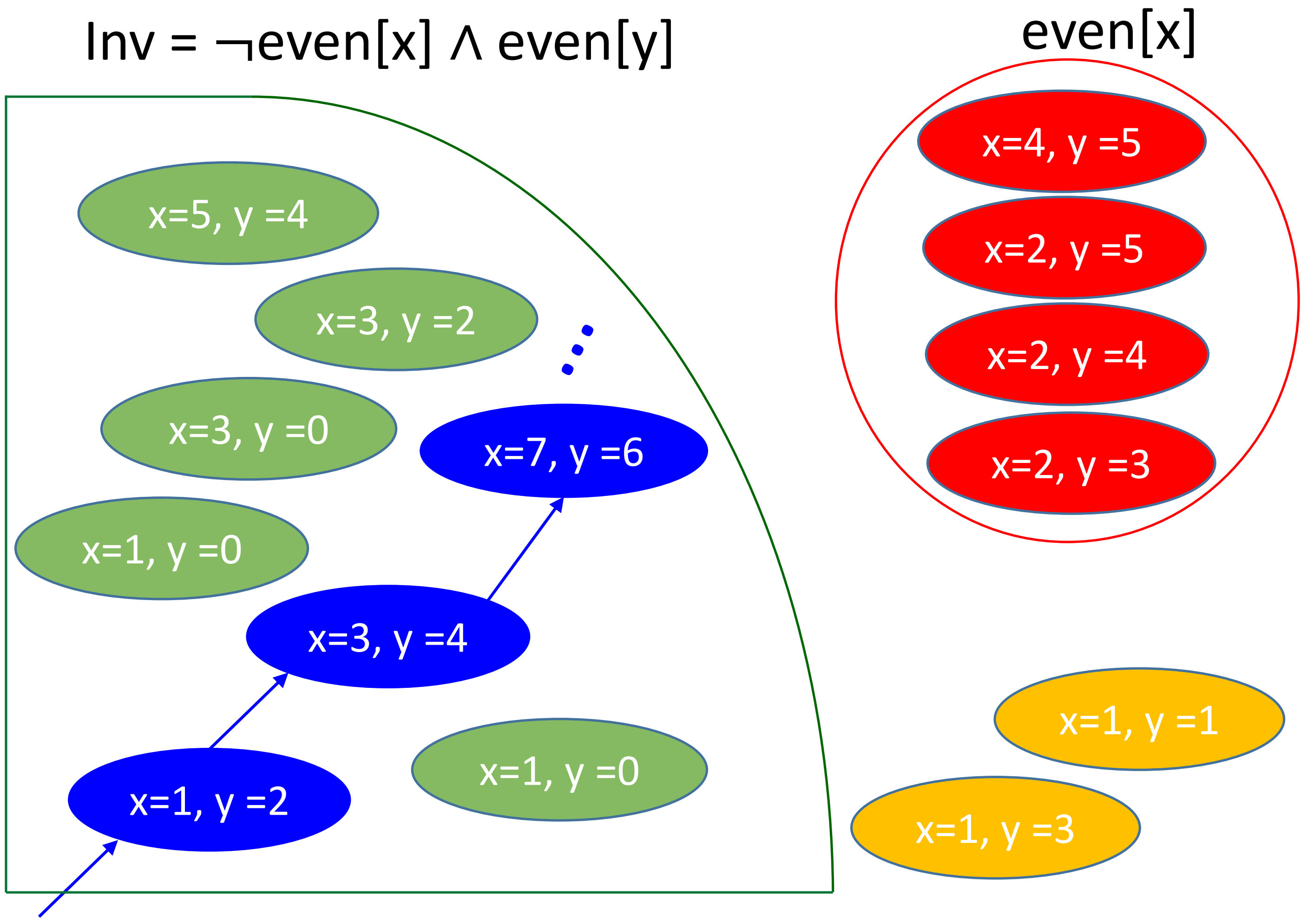
```
x := 1;
y := 2;
while * do {
 assert ¬even[x];
 x := x + y;
 y := y + 2;
}
```

TR



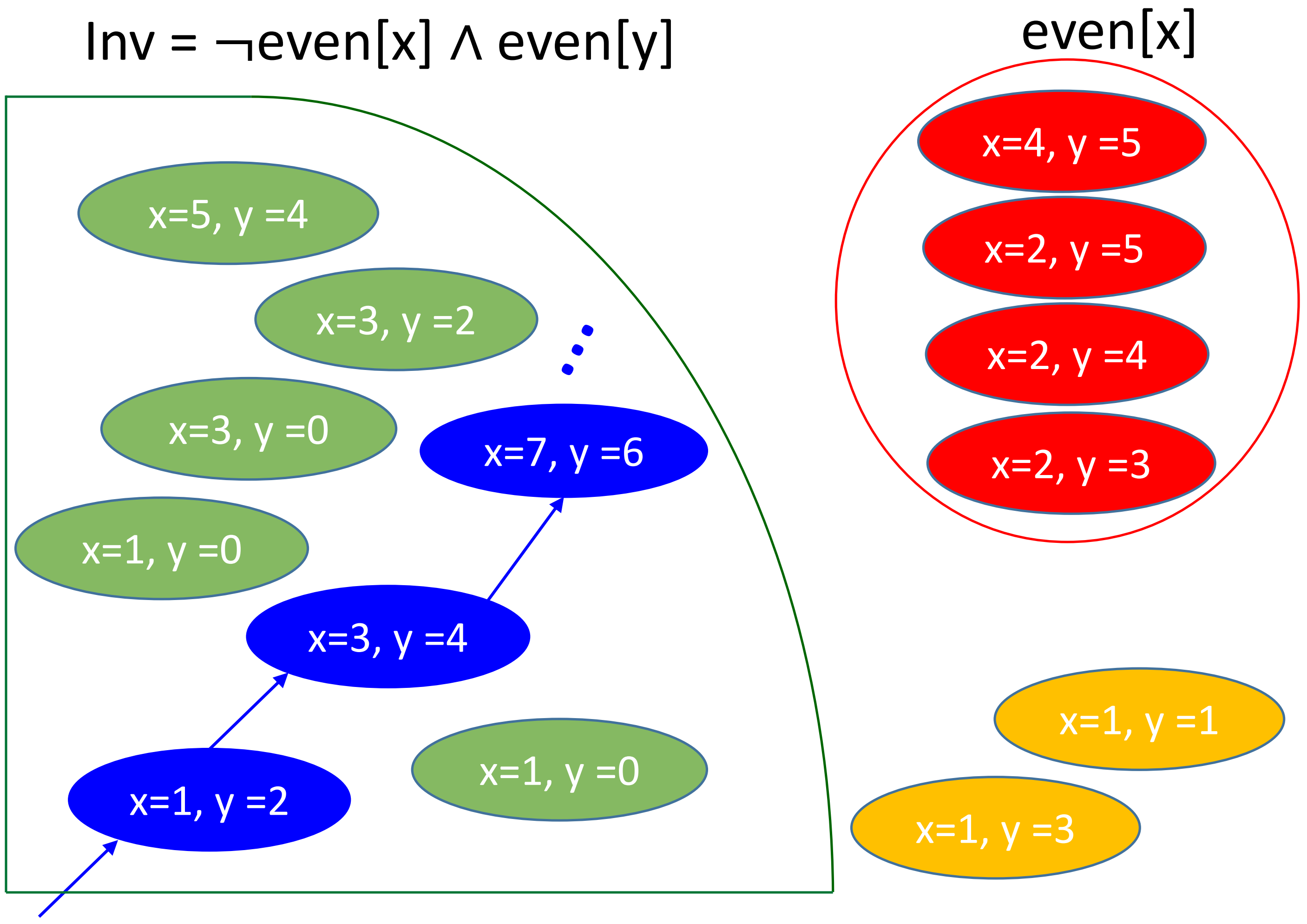
Simple example: loop invariants

```
x := 1;
y := 2;
while * do {
 assert ¬even[x];
TR | x := x + y;
   | y := y + 2;
}
```



Simple example: loop invariants

```
x := 1;
y := 2;
while * do {
 assert ¬even[x];
TR | x:=(x*x–y*y)/(x-y);
   | y := y + 2;
}
```



Challenges in Deductive Verification

1. Formal specification: formalizing infinite-state systems
   - Modeling the system and property (TR, Init, Bad)
2. Deduction: checking inductiveness
   - Undecidability of implication checking
     - Unbounded state (threads, messages), arithmetic, quantifier alternation
3. Inference: inferring inductive invariants (Inv)
   - Hard to specify
   - Hard to infer
     - Undecidable even when deduction is decidable

State of the art in formal verification

Expressiveness

Proof Assistants

Ultimately limited by human

proof/code:
Verdi: ~10
IronFleet: ~4

Automation

“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

Proof Assistants

Ultimately limited by human

proof/code:
Verdi: ~10
IronFleet: ~4

IVy

Decidable Reasoning
Finite Counterexamples
proof/code: ~0.2

Ultimately limited by undecidability

Model Checking
Static Analysis
Type Checking

Expressiveness

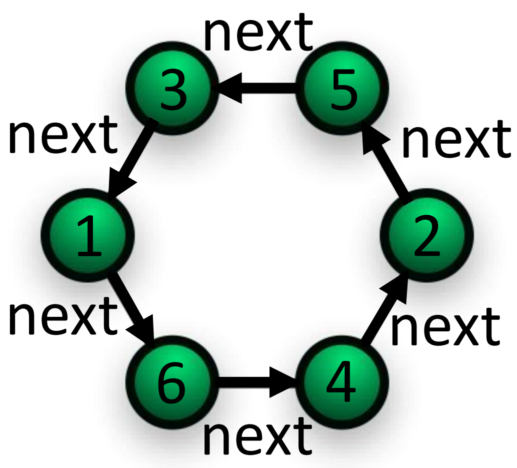
Automation

"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

- Specify systems and properties in decidable fragment of first-order logic (EPR)
  - Allows quantifiers to reason about unbounded sets
  - Decidable to check inductiveness
  - Finite counterexamples to induction, display graphically
  - Logic is mostly hidden
- Interact with the user to find inductive invariants
- Challenge: use restricted logic to verify interesting systems
  - Paxos, Reconfiguration, Byzantine Fault Tolerance
  - Liveness and Temporal Properties

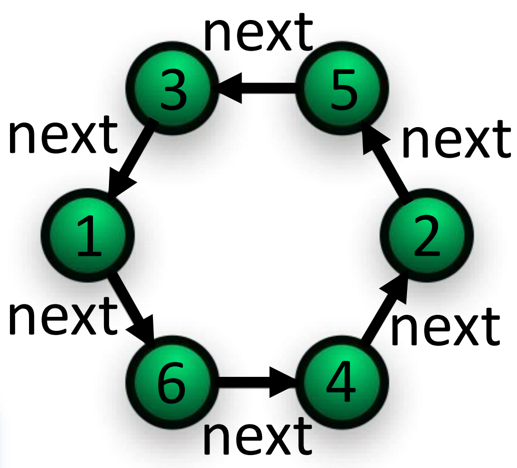
Example: Leader Election in a Ring



- 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*

Example: Leader Election in a Ring



- Nodes are organized in a ring
- Each node has a unique numeric id
- Protocol:
  - Each
  - A no ... if the id in the mes
  - A no
- Theore
  - The

Proposition: This algorithm detects one and only one highest number.

Argument: By the circular nature of the configuration and the consistent direction of messages, any message must meet all other processes before it comes back to its initiator. Only one message, that with the highest number, will not encounter a higher number on its way around. Thus, the only process getting its own message back is the one with the highest number.

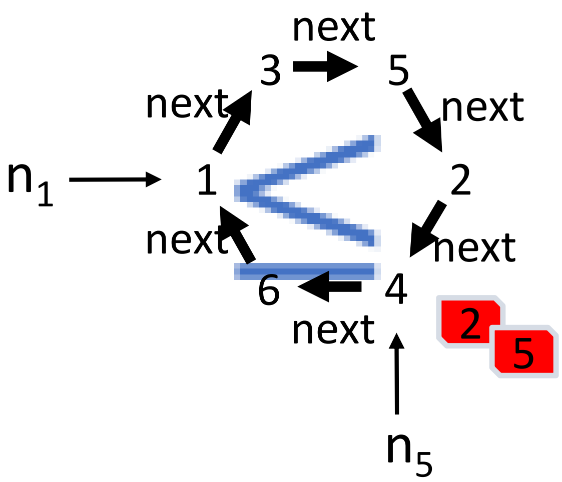
[CACM'79] E. Chang and R. Roberts. *An improved algorithm for decentralized extrema-finding in circular configurations of processes*

Leader Election Protocol (IVy)

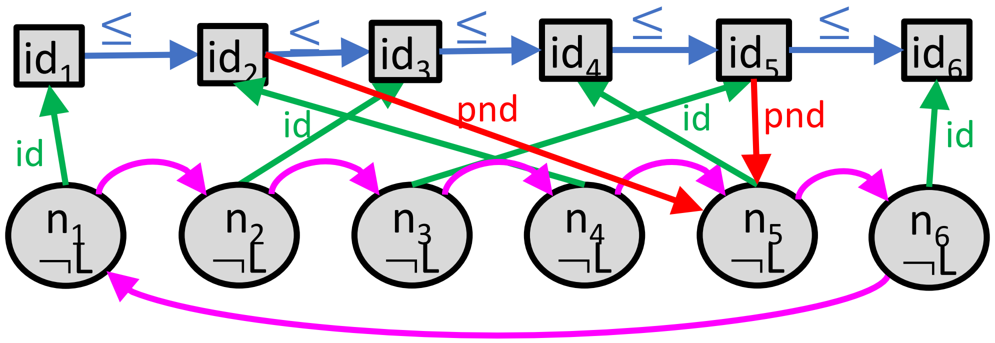
- ⩽ (ID, ID) – total order on node id's
- **btw** (Node, Node, Node) – the ring topology
- **id**: Node → ID – relate a node to its unique id
- **pending**(ID, Node) – pending messages
- **leader**(Node) – leader(n) means n is the leader

Axiomatized in first-order logic

protocol state



structure



$\langle n_5, n_1, n_3\rangle \in I(\textbf{btw})$

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

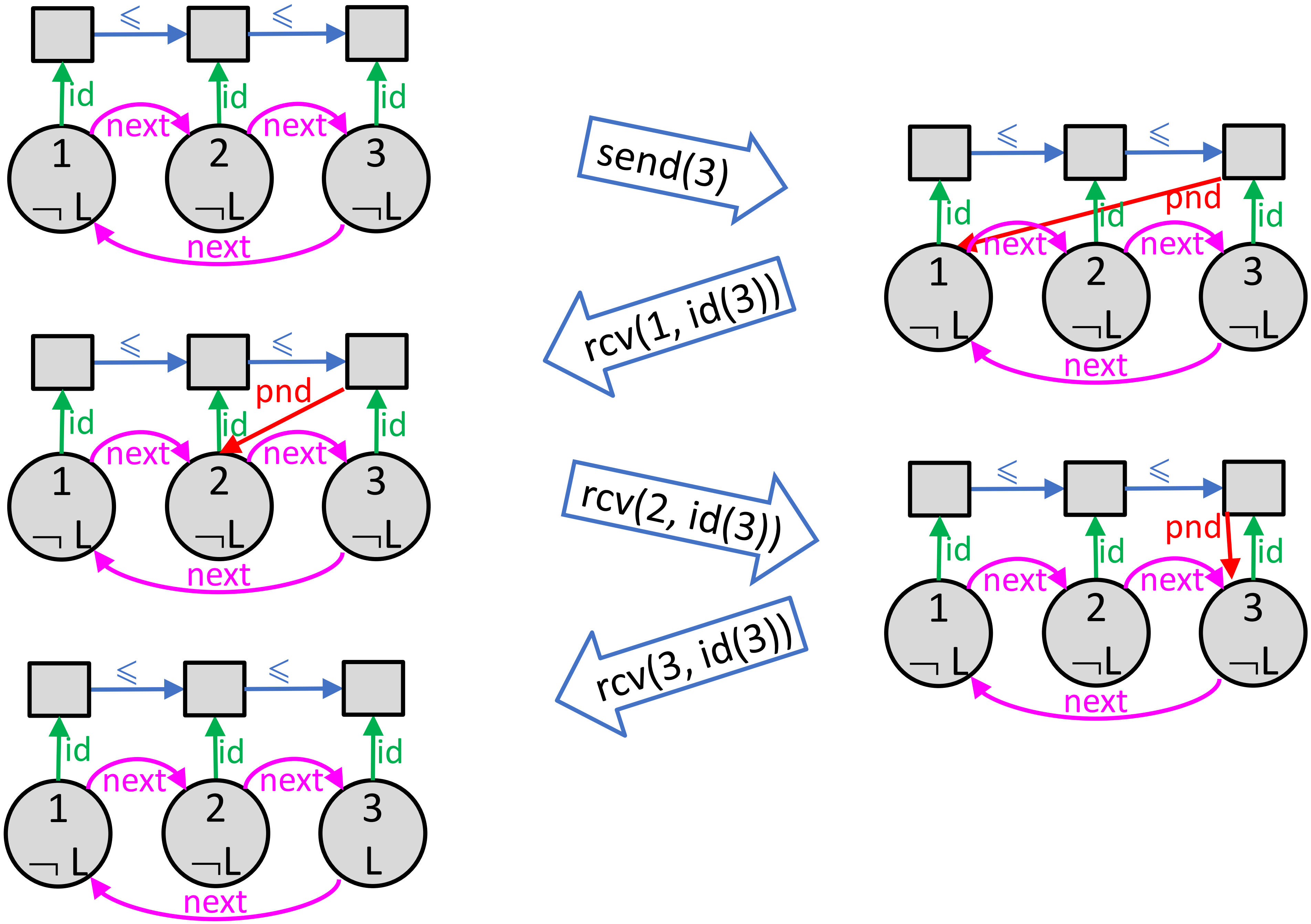
```
action send(n: Node) = {
  "s := next(n)";
  pending(id(n),s) := true
}
```

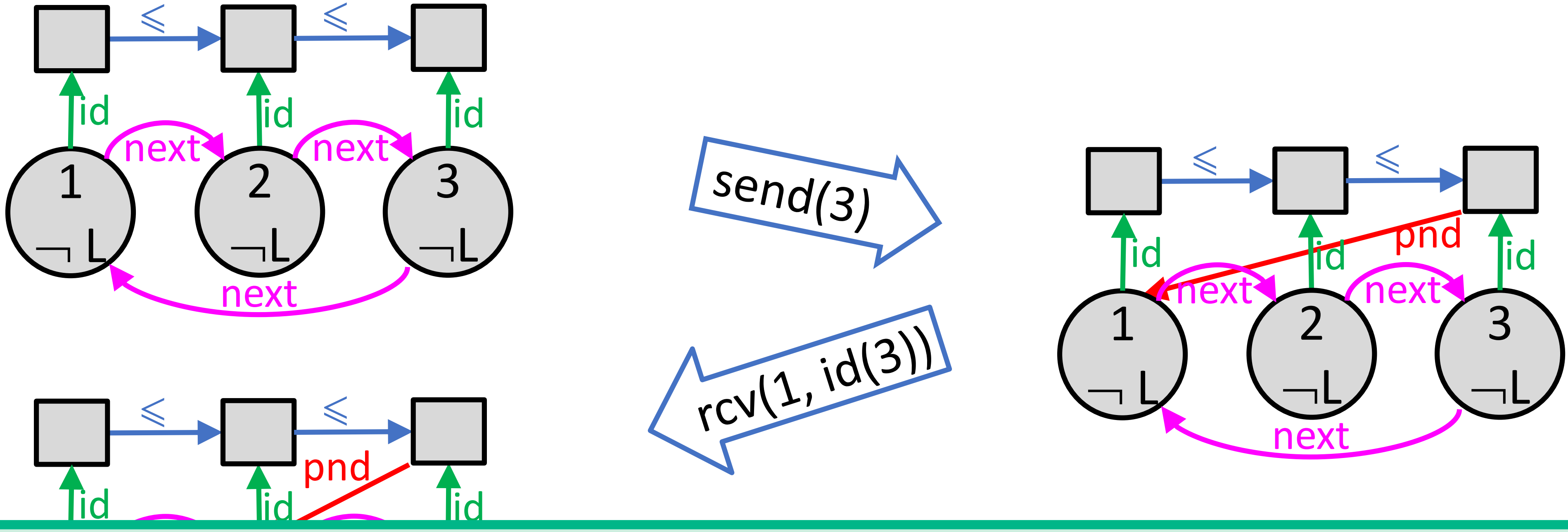
```
action receive(n: Node, m: ID) = {
  requires pending(m, n);
  if id(n) = m then
    // found leader
    leader(n) := true
  else if id(n) ⩽  m then
    // pass message
    "s := next(n)";
    pending(m, s) := true
}
```

∃n,s: Node. "s := next(n)" ∧ ∀x:ID,y:Node. pending'(x,y)↔(pending(x,y)∨(x=id(n)∧y=s))

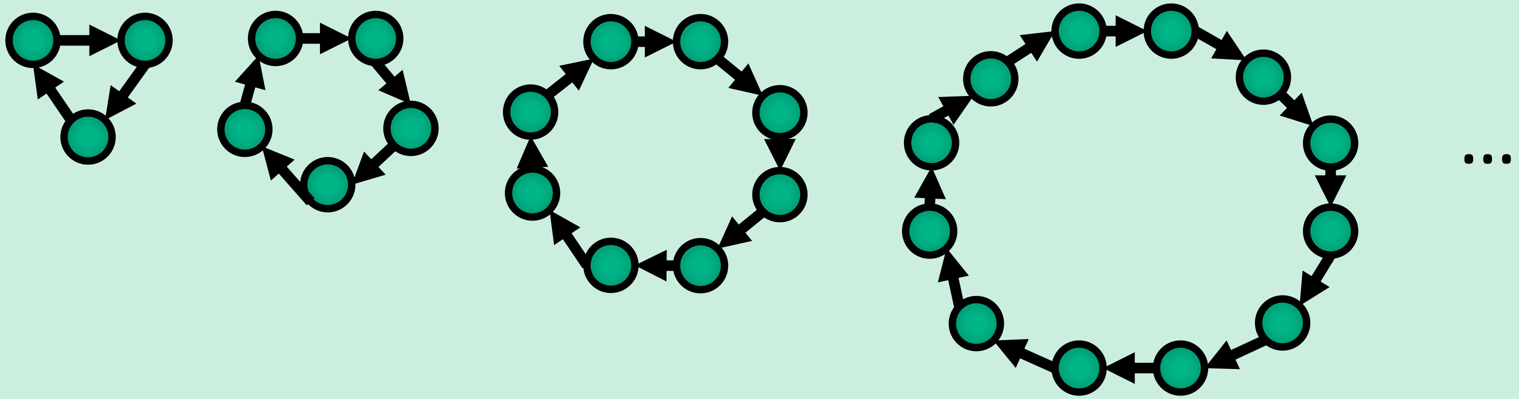
```
protocol = (send | receive)*
```

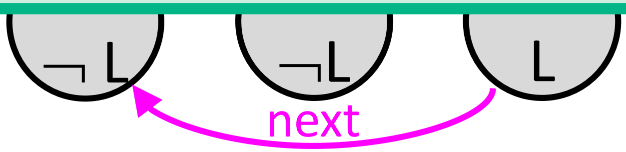
```
assert I0 = ∀ x,y: Node. leader(x)∧leader(y) → x = y
```


send(3)
rcv(1, id(3))
rcv(2, id(3))
rcv(3, id(3))
id
next
pnd
¬L
L
⩽



Specify and verify the protocol for **any** number of nodes in the ring





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

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

Inductive invariant: $Inv = I_0 \land I_1 \land I_2 \land I_3$

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

The leader has the highest ID

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

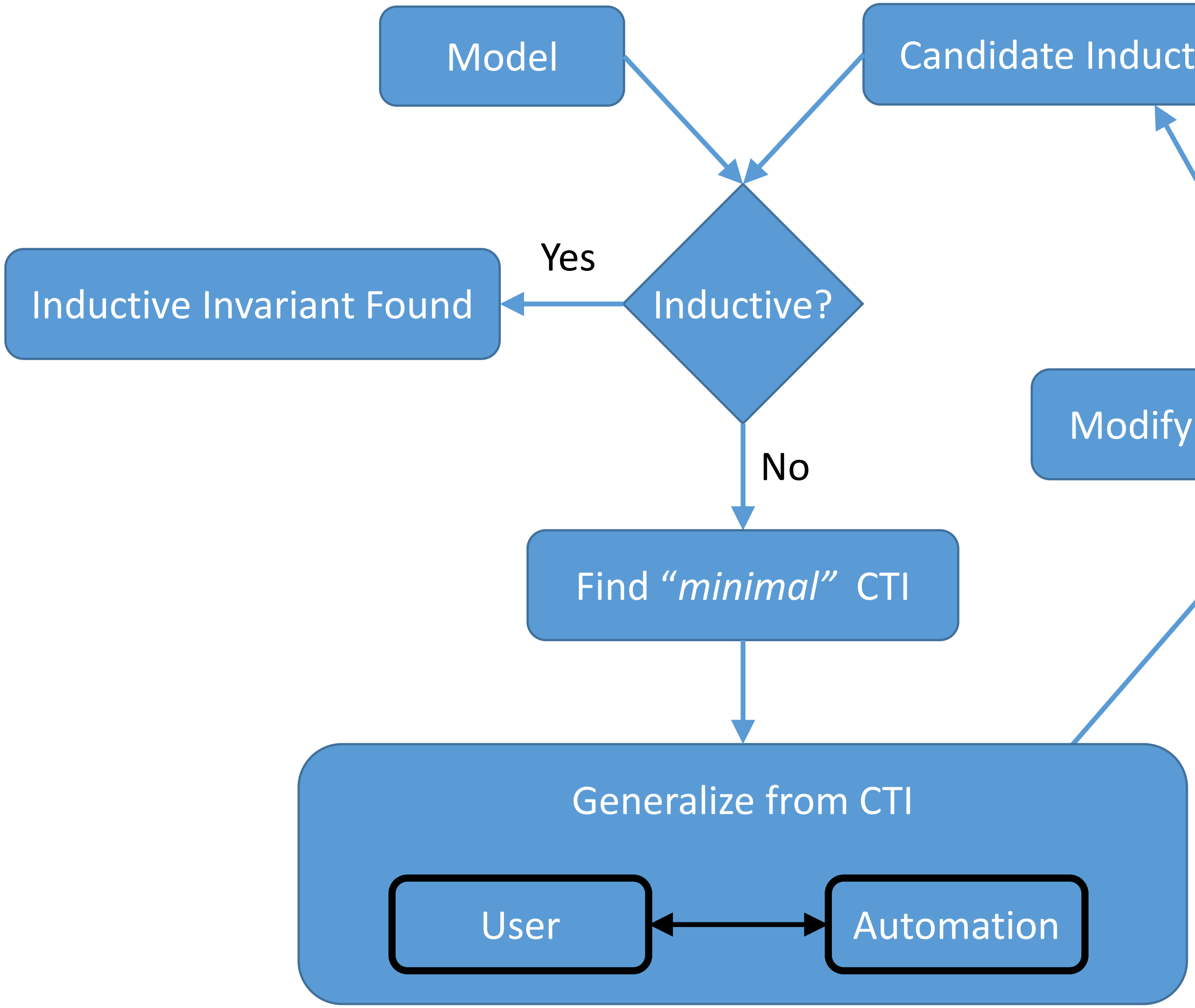
Only the leader can be self-pending

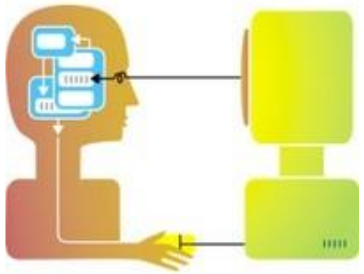
$$I_3 = \forall n_1, n_2, n_3: \text{Node}.\ btw(n_1, n_2, n_3) \land pending(id[n_2], n_1) \Rightarrow id[n_3] \leqslant id[n_2]$$

Cannot bypass higher nodes

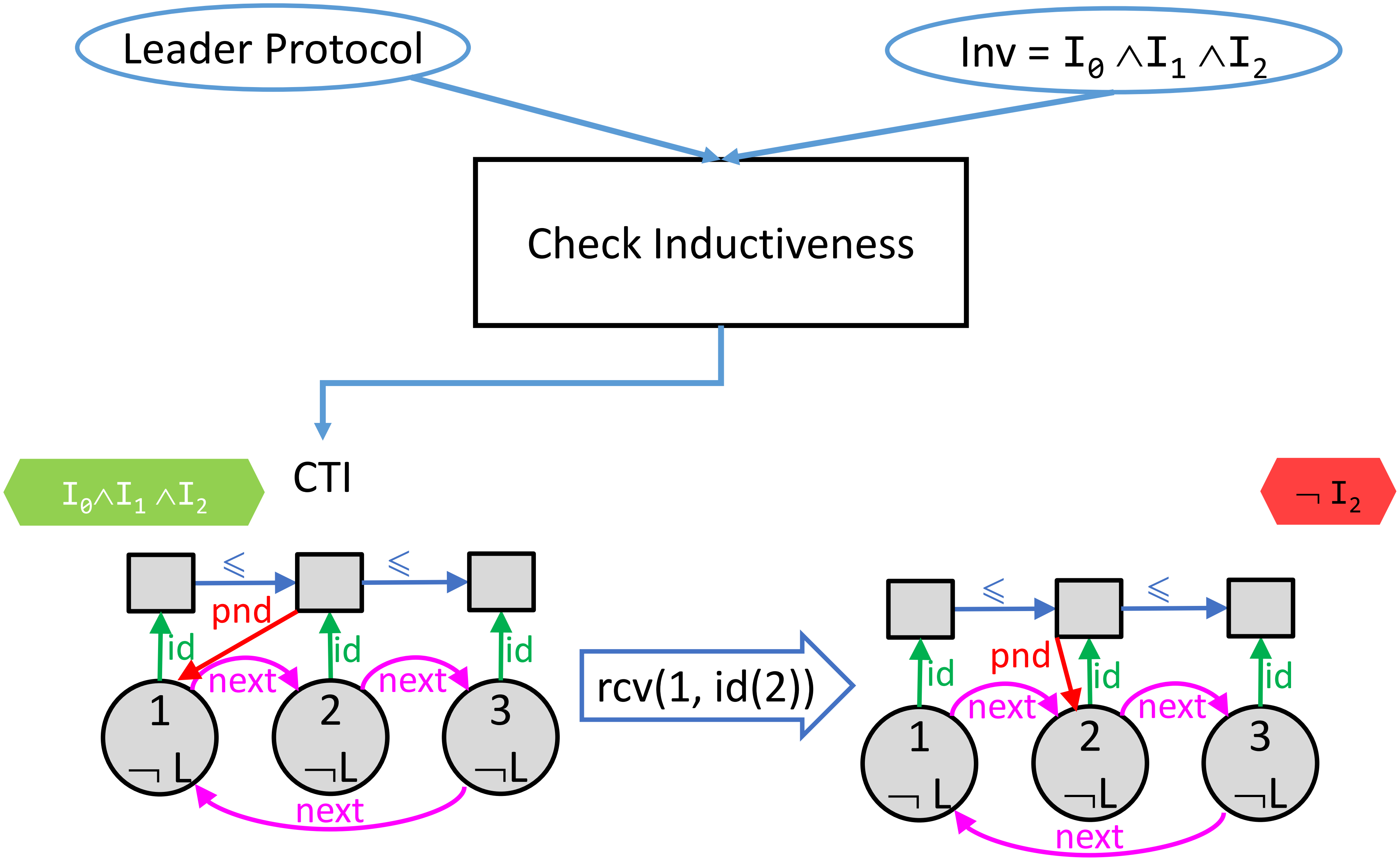
How can we find an inductive invariant without knowing it?

Invariant Inference in IVy

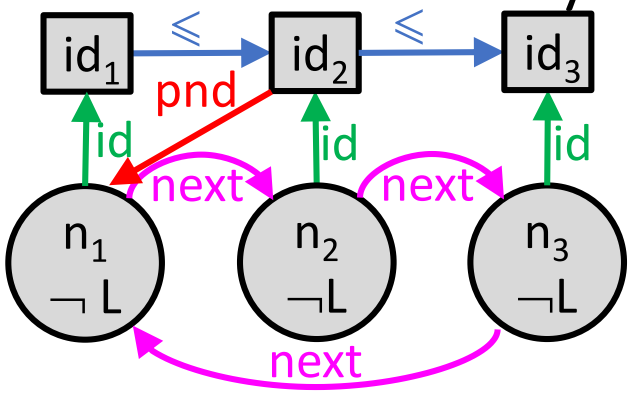




IVy: Check Inductiveness

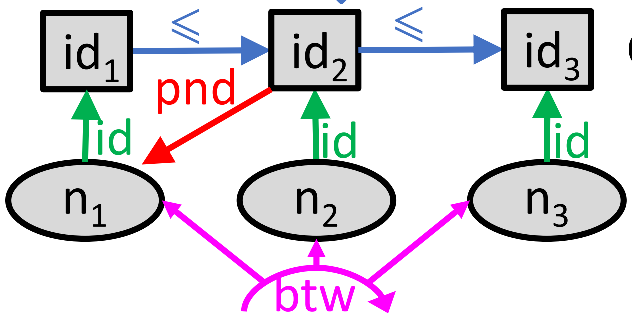


IVy: Generalize from CTI



Cannot bypass nodes with higher ids

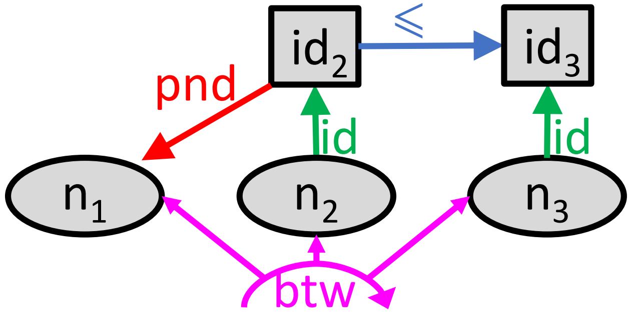
Project to {pnd,⩽, id,btw}



$C_3 = \neg\exists n_1, n_2, n_3 : \text{Node}.\ \neq(n_1,n_2,n_3) \wedge$
$\neq(id[n_1],id[n_2],id[n_3]) \wedge$
$id[n_1] \leqslant id[n_2] \leqslant id[n_3] \wedge$
$pnd(id[n_2], n_1) \wedge btw(n_1, n_2, n_3)$

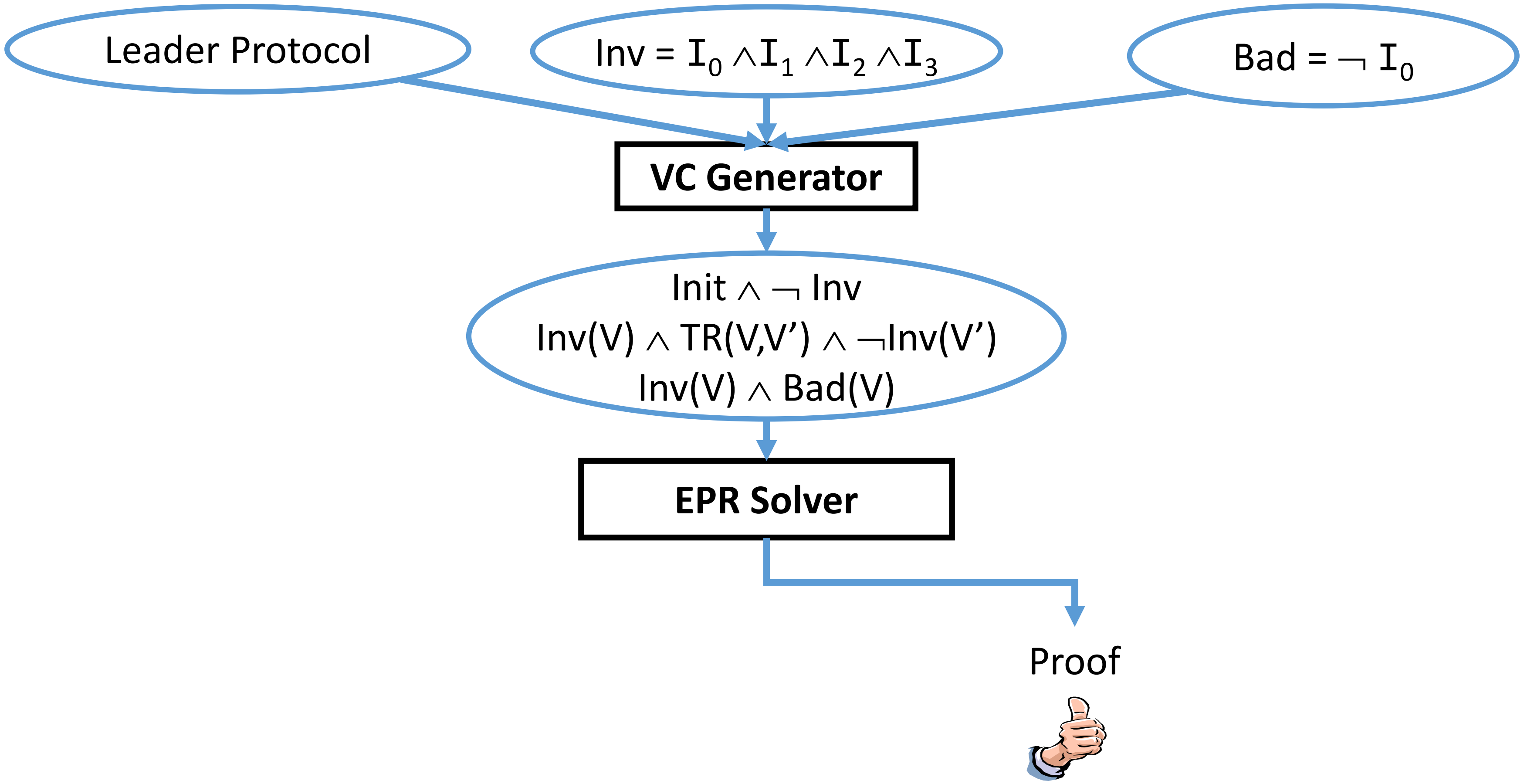
Interp(3)

This looks good, add to the invariant as I_3

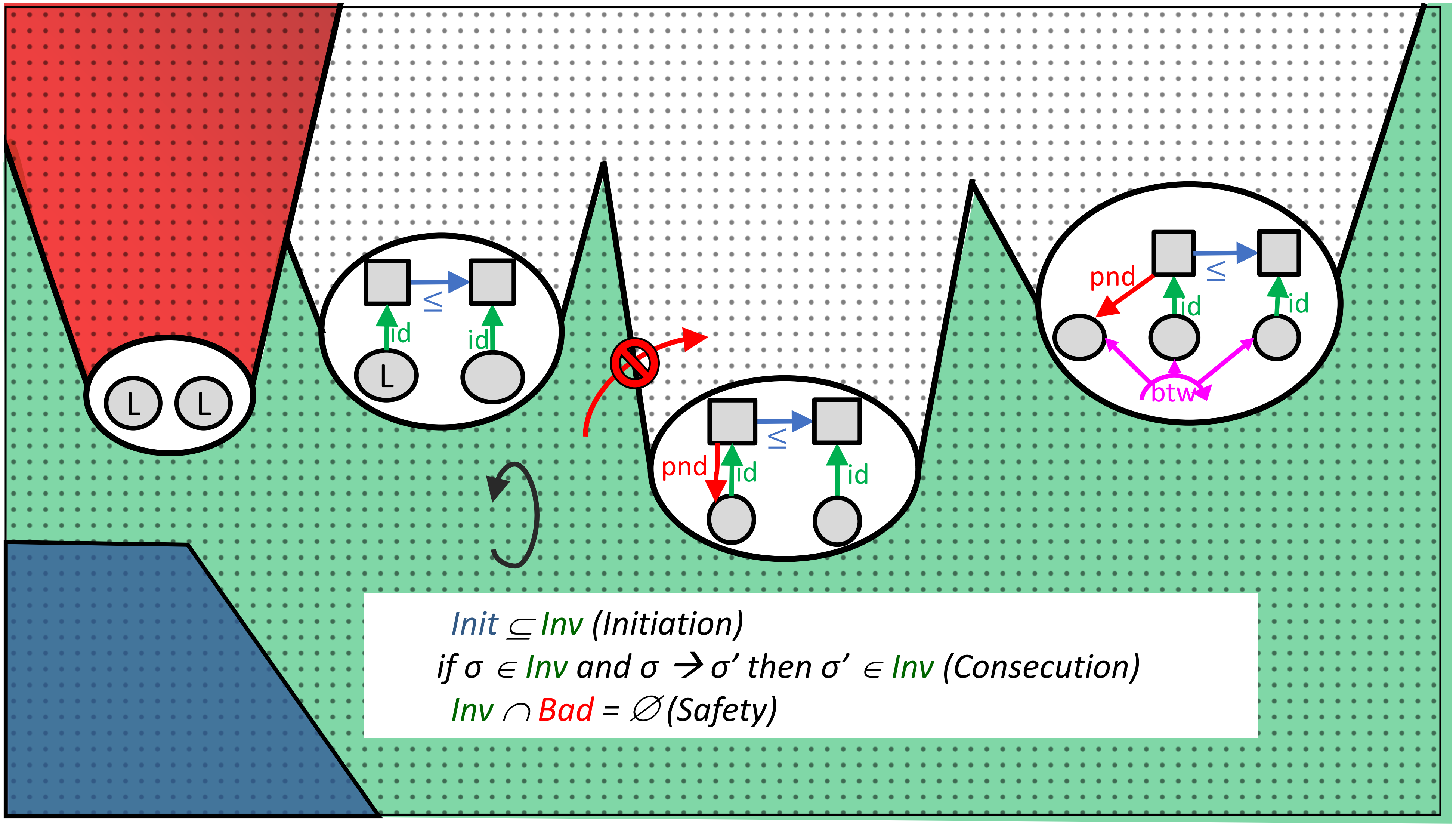


$I_3 = \neg\exists n_1, n_2, n_3 : \text{Node}.\ btw(n_1, n_2, n_3) \wedge$
$pnd(id[n_2], n_1) \wedge$
$id[n_2] \leqslant id[n_3]$

IVy: Check Inductiveness



$I_0 \wedge I_1 \wedge I_2 \wedge I_3$ is an inductive invariant for the leader protocol, which proves the protocol is safe


L
L
L
id
id
≤
pnd
id
id
≤
pnd
id
id
≤
btw
Init ⊆ Inv (Initiation)
if σ ∈ Inv and σ → σ' then σ' ∈ Inv (Consecution)
Inv ∩ Bad = ∅ (Safety)

Leader Election Protocol (axioms)

- $\leqslant$ (ID, ID) – total order on node id's
- **btw** (a: Node, b: Node, c: 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

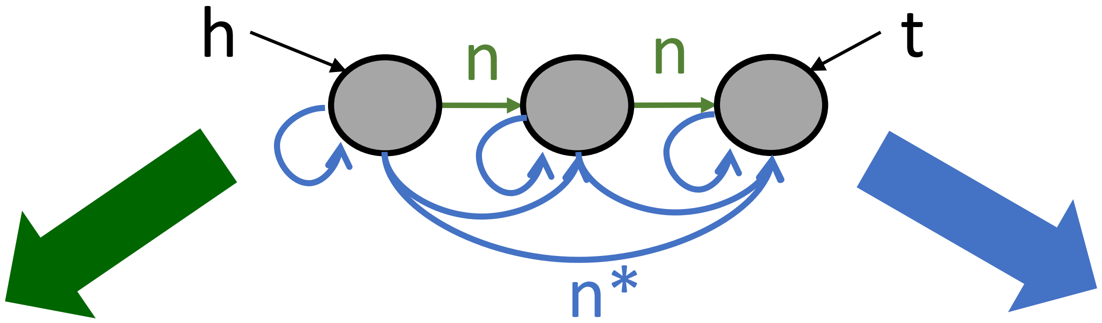
	Natural Interpretation	EPR Modeling
Node ID's	Integers	**∀i:ID.** $i \leqslant i$ **Reflexive** **∀i, j, k: ID.** $i \leqslant j \wedge j \leqslant k \Rightarrow i \leqslant k$ **Transitive** **∀i, j: ID.** $i \leqslant j \wedge j \leqslant i \Rightarrow i = j$ **Anti-Symmetric** **∀i, j: ID.** $i \leqslant j \vee j \leqslant i$ **Total** **∀x, y: Node.** $id(x) = id(y) \Rightarrow x = y$ Injective
Ring Topology	Next edges + Transitive closure	**∀x, y, z: Node.** $btw(x, y, z) \Rightarrow btw(y, z, x)$ Circular shifts **∀x, y, z, w: Node.** $btw(w, x, y) \wedge btw(w, y, z) \Rightarrow btw(w, x, z)$ Transitive **∀x, y, w: Node.** $btw(w, x, y) \Rightarrow \neg btw(w, y, x)$ A-Symmetric **∀x, y, z, w: Node.** $distinct(x, y, z) \Rightarrow btw(w, x, y) \vee btw(w, y, x)$
		"`next(a)=b`" ≡ **∀x: Node.** $X=a \vee X=b \vee$ `btw(a,b,x)`

Challenge: How to use restricted first-order logic to verify interesting systems?

- Expressing transitive closure
  - Linked lists
  - Ring protocols
- Expressing Consensus
  - Paxos, Multi-Paxos
  - Reconfiguration
  - Byzantine Fault Tolerance
- Liveness and temporal Properties

Key idea: representing deterministic paths

[Itzhaky SIGPLAN Dissertation Award 2016]



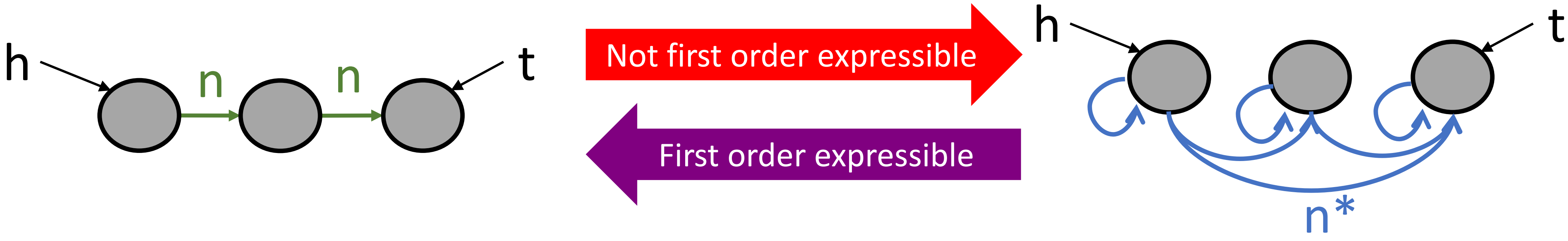
$n^* \approx$ btw

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 to outdegree ≤ 1
- Definable in first order logic (for roots)
  - $n^+(a,b) \equiv n^*(a, b) \land a \neq b$
  - $n(a, b) \equiv n^+(a,b) \land \forall z: n^+(a, z) \rightarrow n^*(b, z)$



Paxos made EPR

Methodology for decidable verification of infinite-state systems

Protocol → (1) Modeling → Formal specification *in first-order logic* → (2) Transforming → Formal specification with *decidable VC*

Abstraction
Domain knowledge

Z3

Paxos

- Single decree Paxos – consensus
  lets nodes make a common decision despite node crashes and packet loss
- Paxos family of protocols – state machine replication
  variants for different tradeoffs, e.g., Fast Paxos is optimized for low contention, Vertical Paxos is reconfigurable, etc.
- Pervasive approach to fault-tolerant distributed computing
  - Google Chubby
  - VMware NSX
  - AWS
  - Many more…

Challenge: reasoning about Paxos in FOL

- Consensus algorithms use set cardinalities
  - Wait for messages from more than N / 2 nodes
- Insight: set cardinalities are used to get a simple effect

  Can be modeled in first-order logic!

- Solution: axiomatize quorums in first-order logic

  sort quorum

  relation member (node, quorum)

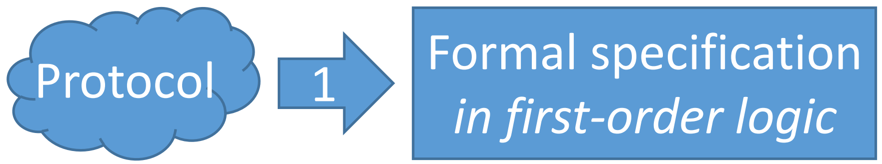
  – set membership (2^{nd}-order logic in first-order)

  axiom $\forall q_1,q_2$: quorum. $\exists n$: node. member(n, q_1) $\land$ member(n, q_2)

```
action propose(r:round) {
 requires "›N/2 join_msg's"
 …
}
```

```
action propose(r:round) {
 requires ∃q.∀n.member(n,q) →
   ∃r',v'.join_msg(n,r,r',v')
 …
}
```

Principle: domain knowledge



Concept	Intention	First-order abstraction
Quorums	Majority sets	**relation** member (node, quorum) **axiom** $\forall q_1, q_2{:}quorum \exists n{:}node.\ member(n, q_1) \land member(n, q_2)$
Rounds	Natural numbers	**relation** $\leq$(round,round) **axiom** $\forall x{:}round.\ \ x \leq x$ *reflexive* **axiom** $\forall x,y,z{:}round.\ x \leq y \land y \leq z \rightarrow x \leq z$ *transitive* **axiom** $\forall x,y{:}round.\ x \leq y \land y \leq x \rightarrow x = y$ *anti-symmetric* **axiom** $\forall x,y{:}round.\ x \leq y \lor y \leq x$ *total*
Messages	Network with: dropping duplication reordering	**relation** start_msg(round) **relation** join_msg(node,round,round,value) **relation** propose_msg(round,value) **relation** vote_msg(node,round,value)

Paxos in first-order logic

```
sort node, quorum, round, value

relation ≤ : round, round
axiom total_order(≤)
constant ⊥ : round

relation member : node, quorum
axiom ∀q1, q2 : quorum. ∃n : node. member(n, q1) ∧ member(n, q2)

relation start_round_msg : round
relation join_ack_msg : node, round, round, value
relation propose_msg : round, value
relation vote_msg : node, round, value
relation decision : node, round, value

init ∀r. ¬start_round_msg(r)
init ∀n, r1, r2, v. ¬join_ack_msg(n, r1, r2, v)
init ∀r, v. ¬propose_msg(r, v)
init ∀n, r, v. ¬vote_msg(n, r, v)
init ∀n, r, v. ¬decision(n, r, v)

action START_ROUND(r : round) {
  assume r ≠ ⊥
  start_round_msg(r) := true
}
action JOIN_ROUND(n : node, r : round) {
  assume r ≠ ⊥
  assume start_round_msg(r)
  assume ¬∃r′, r″, v. r′ > r ∧ join_ack_msg(n, r′, r″, v)
  # find maximal round in which n voted, and the corresponding vote.
  # maxr = ⊥ and v is arbitrary when n never voted.
  local maxr, v := max {(r′, v′) | vote_msg(n, r′, v′) ∧ r′ < r}
  join_ack_msg(n, r, maxr, v) := true
}
action PROPOSE(r : round, q : quorum) {
  assume r ≠ ⊥
  assume ∀v. ¬propose_msg(r, v)
  # 1b from quorum q
  assume ∀n. member(n, q) → ∃r′, v. join_ack_msg(n, r, r′, v)
  # find the maximal round in which a node in the quorum reported
  # voting, and the corresponding vote.
  # v is arbitrary if the nodes reported not voting.
  local maxr, v := max {(r′, v′) | ∃n. member(n, q)
                          ∧join_ack_msg(n, r, r′, v′) ∧ r′ ≠ ⊥}
  propose_msg(r, v) := true  # propose value v
}
action VOTE(n : node, r : round, v : value) {
  assume r ≠ ⊥
  assume propose_msg(r, v)
  assume ¬∃r′, r″, v. r′ > r ∧ join_ack_msg(n, r′, r″, v)
  vote_msg(n, r, v) := true
}
action LEARN(n : node, r : round, v : value, q : quorum) {
  assume r ≠ ⊥
  # 2b from quorum q
  assume ∀n. member(n, q) → vote_msg(n, r, v)
  decision(n, r, v) := true
}
```

$\forall n_1, n_2 : \text{node}, r_1, r_2 : \text{round}, v_1, v_2 : \text{value}.\ decision(n_1, r_1, v_1) \land decision(n_2, r_2, v_2) \to v_1 = v_2$

$\forall r : \text{round}, v_1, v_2 : \text{value}.\ propose\_msg(r, v_1) \land propose\_msg(r, v_2) \to v_1 = v_2$

$\forall n : \text{node}, r : \text{round}, v : \text{value}.\ vote\_msg(n, r, v) \to propose\_msg(r, v)$

$\forall r : \text{round}, v : \text{value}.(\exists n : \text{node}.\ decision(n, r, v)) \to \exists q : \text{quorum}.\forall n : \text{node}.\ member(n, q) \to vote\_msg(n, r, v)$

$\forall n : \text{node},\ r, r' : \text{round},\ v, v' : \text{value}.\ join\_ack\_msg(n, r, \bot, v) \land r' < r \to \neg vote\_msg(n, r', v')$

$\forall n : \text{node},\ r, r' : \text{round},\ v : \text{value}.\ join\_ack\_msg(n, r, r', v) \land r' \neq \bot \to r' < r \land vote\_msg(n, r', v)$

$\forall n : \text{node},\ r, r', r'' : \text{round},\ v, v' : \text{value}. join\_ack\_msg(n, r, r', v) \land r' \neq \bot \land r' < r'' < r \to \neg vote\_msg(n, r'', v')$

$\forall n : \text{node}, v : \text{value}.\ \neg vote\_msg(n, \bot, v)$

$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, q : \text{quorum}.\ propose\_msg(r_2, v_2) \land r_1 < r_2 \land v_1 \neq v_2 \to$
$\quad \exists n : \text{node}, r', r'' : \text{round}, v : \text{value}.\ member(n, q) \land \neg vote\_msg(n, r_1, v_1) \land r' > r_1 \land join\_ack\_msg(n, r', r'', v)$

VC's in first-order logic

Step 2: Obtaining decidable VC's

Challenge : quantifier alternation cycles

- Axiom

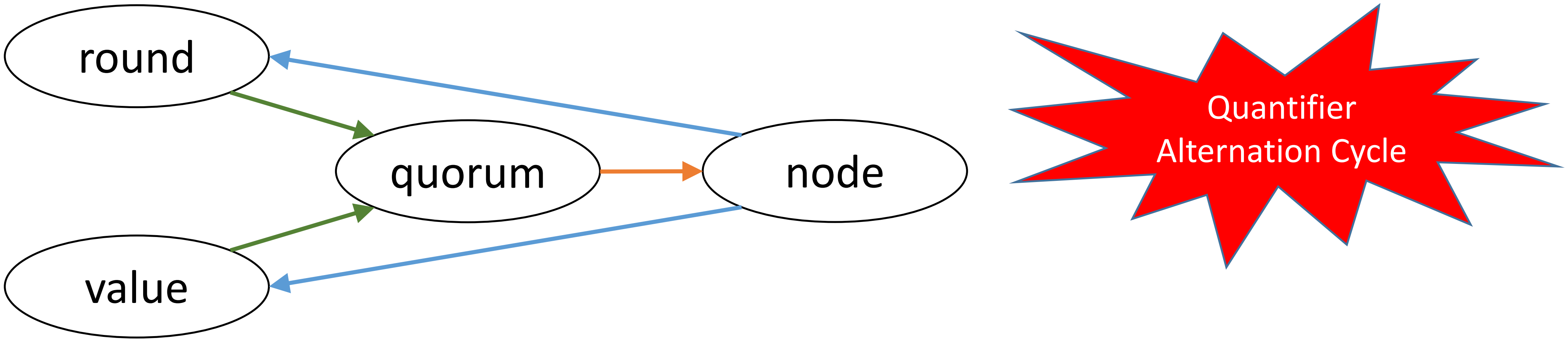
  ∀q1,q2: quorum. ∃n: node. member(n, q1) ∧ member(n, q2)

- Propose action precondition

  ∃q:quorum. ∀n:node. member(n,q) → ∃r':round,v':value. join_msg(n,r,r',v')

- Inductive invariant

  ∀r:round,v:value. decision(r,v) → ∃q:quorum. ∀n:node. member(n,q) → vote_msg(n,r,v)

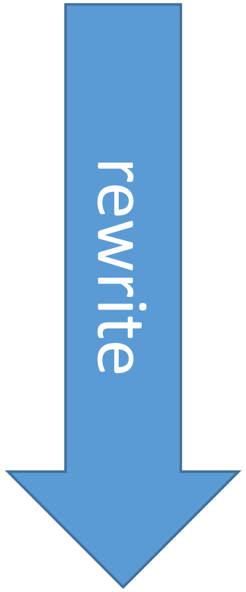


Solution: derived relations and rewrites

$\exists q{:}quorum.\ \forall n{:}node.\ member(n,q) \rightarrow \exists r'{:}round, v'{:}value.\ join\_msg(n,r,r',v')$

Solution: derived relations and rewrites

∃q:quorum. ∀n:node. member(n,q) → ∃r’:round,v’:value. join_msg(n,r,r’,v’)



new relation: joined(n:node,r:round) ≡ ∃r’:round,v’:value. join_msg(n,r,r’,v’)

update code:

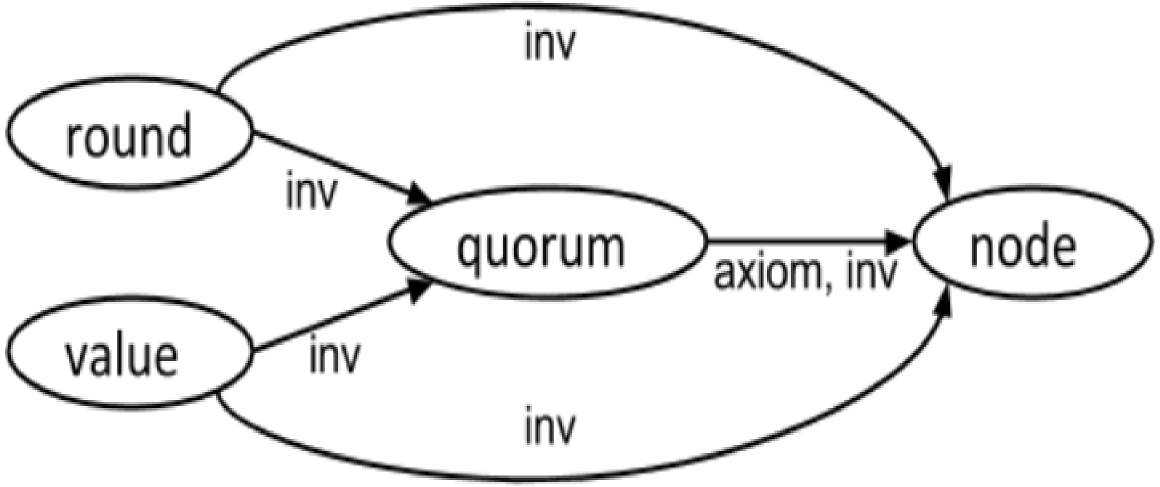
```
action join(n:node, r:round) {
  requires start_round_msg(r)
  let maxr,v := …
  join_msg(n,r,maxr,v) := true
  joined(n,r) := true
}
```

∃q:quorum. ∀n:node. member(n,q) → joined(n,r)

Solution: derived relations and rewrites

joined(n:node,r:round) ≡ ∃r’:round,v’:value. join_msg(n,r,r’,v’)

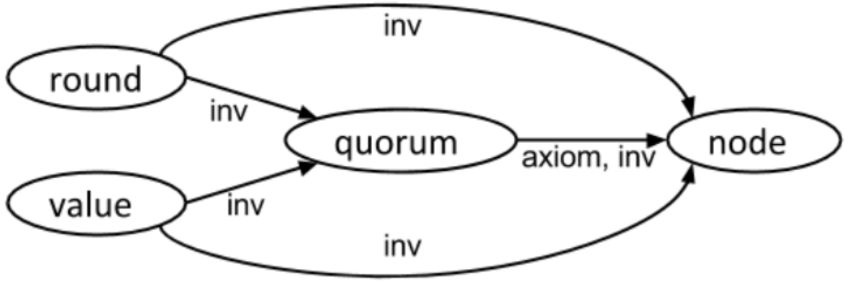
left(n:node,r:round) ≡ ∃r’,r’’:round,v’:value. join_msg(n,r’,r’’,v’) ∧ r’>r



VC’s are decidable!

Principle: decomposing into decidable checks

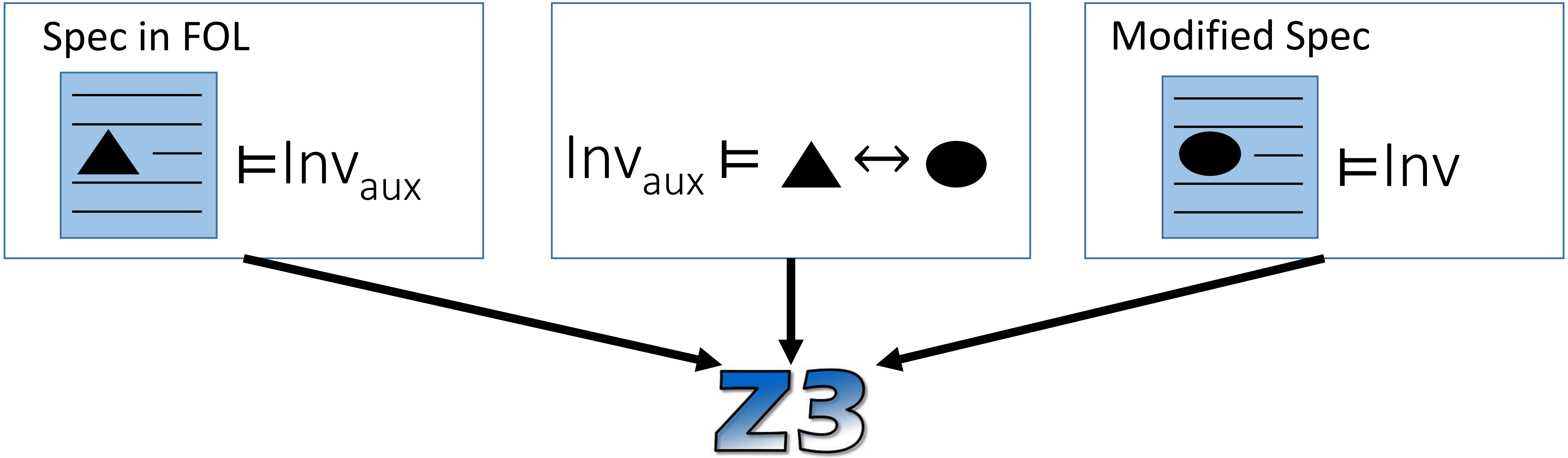
- User defines:
  - Derived relations
  - Rewrites
  - Inductive invariants
- Decidable checks:



Formal specification *in first-order logic*

2

Formal specification with *decidable VC*



Inductive Invariant of Paxos

```
# safety property
conjecture decision(N1,R1,V1) & decision(N2,R2,V2) -> V1 = V2
# proposals are unique per round
conjecture proposal(R,V1) & proposal(R,V2) -> V1 = V2
# only vote for proposed values
conjecture vote(N,R,V) -> proposal(R,V)
# decisions come from quorums of votes:
conjecture forall R, V. (exists N. decision(N,R,V)) -> exists Q. forall N. member(N, Q) -> vote(N,R,V)
# properties of one_b_max_vote
conjecture one_b_max_vote(N,R2,none,V1) & ~le(R2,R1) -> ~vote(N,R1,V2)
conjecture one_b_max_vote(N,R,RM,V) & RM ~= none -> ~le(R,RM) & vote(N,RM,V)
conjecture one_b_max_vote(N,R,RM,V) & RM ~= none & ~le(R,RO) & ~le(RO,RM) -> ~vote(N,RO,VO)
# property of choosable and proposal
conjecture ~le(R2,R1) & proposal(R2,V2) & V1 ~= V2 -> exists N. member(N,Q) & left_rnd(N,R1) & ~vote(N,R1,V1)
# property of one_b, left_rnd
conjecture one_b(N,R2) & ~le(R2,R1) -> left_rnd(N,R1)
```

Experimental Evaluation

Protocol	Model [LOC]	Invariant [Conjectures]	EPR [sec] μ	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!

Stoppable Paxos

Dahlia Malkhi   Leslie Lamport   Lidong Zhou

April 28, 2008

have been chosen as the j^{th} command for some $j < i$. Although the basic idea of the algorithm is not complicated, getting the details right was not easy.

⟨1⟩7. $NoneChoosableAfter(i, b, v)'$

PROOF: We assume $v \in StopCmd$, $j > i$, $c < b$, and w any command and we prove $NotChoosable(j, c, w)'$. By Lemma 1.7, it suffices to prove $NotChoosable(j, c, w)$. We split the proof into two cases.

⟨2⟩1. CASE: $sval2a(i, b, Q) = \top$

PROOF: Assumption ⟨1⟩1.3 implies $E4(i, b, Q, v)$, so the assumption $v \in StopCmd$ implies $E4b(i, b, Q, v)$. The case assumption, the assumption $j > i$, and $E4b(i, b, Q, v)$ imply $sval2a(j, b, Q) = \top$. The assumption $c < b$ and step ⟨1⟩4 then imply $NotChoosable(j, c, w)$.

⟨2⟩2. CASE: $sval2a(i, b, Q) \neq \top$

⟨3⟩1. $sval2a(i, b, Q) = val2a(i, b, Q) = v$

PROOF: Assumption ⟨1⟩1.3 implies $E3(i, b, Q, v)$, which implies $sval2a(i, b, Q) = v$. The case assumption and the definition of $sval2a$ then implies $val2a(i, b, Q) = v$.

⟨3⟩2. $Done2a(i, mbal2a(i, b, Q), v)$

PROOF: ⟨3⟩1, assumption ⟨1⟩1.4, and the definition of $val2a$ imply $vote_i[a][mbal2a(i, b, Q)] = v$ for some acceptor a in Q, which by Lemma 1.3 implies $Done2a(i, mbal2a(i, b, Q), v)$.

By the assumption $c < b$, it suffices to consider the following two cases.

⟨3⟩3. CASE: $c < mbal2a(i, b, Q)$

PROOF: Step ⟨3⟩2 and assumption ⟨1⟩1.1 imply $NoneChoosableAfter(i, mbal2a(i, b, Q), v)$. By the case assumption and the assumptions $v \in StopCmd$ and $j > i$, this implies $NotChoosable(j, c, w)$.

⟨3⟩4. CASE: $mbal2a(i, b, Q) \leq c < b$

⟨4⟩1. $mbal2a(j, b, Q) < mbal2a(i, b, Q)$

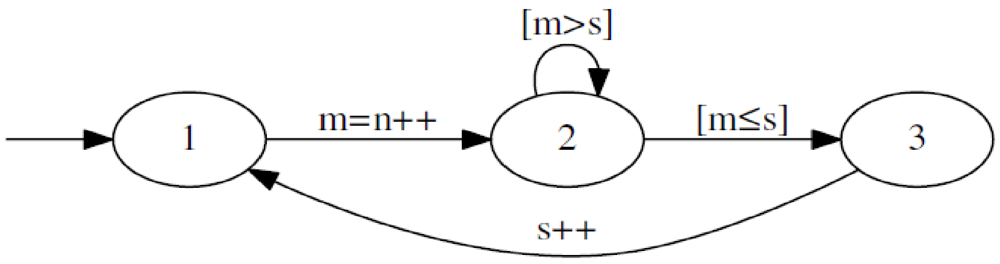
PROOF: The assumption $v \in StopCmd$ and ⟨3⟩1 imply $sval2a(i, b, Q) \in StopCmd$. Case assumption ⟨2⟩2 and the definition of $sval2a$ then imply $mbal2a(k, b, Q) < mbal2a(i, b, Q)$ for all $k > i$.

⟨4⟩2. $NotChoosable(j, c, w)$

PROOF: ⟨4⟩1 and case assumption ⟨3⟩4 imply $mbal2a(j, b, Q) < c < b$. By assumption ⟨1⟩1.4, Lemma 3 implies $NotChoosable(j, c, w)$. □

Verification of Temporal Properties

```
   global nat s, n
   local nat m
1: while (true) {
     m=n++; // Acquire a ticket
2:   while (m>s) { // Busy wait
       skip;
     }
     // Critical section
3:   s++; // Exit critical
   }
```



Liveness Property	$\forall x : \text{thread}.\ \Box\,(pc_2(x) \rightarrow \Diamond\, pc_3(x))$
Fairness Assumption	$\forall x : \text{thread}.\ \Box\Diamond\, scheduled(x)$
Temporal Spec. (*spec*)	$(\forall x : \text{thread}.\ \Box\Diamond\, scheduled(x)) \rightarrow \forall x : \text{thread}.\ \Box\,(pc_2(x) \rightarrow \Diamond\, pc_3(x))$

Possible Projects

- Verify any distributed / shared memory algorithm
- Paxos variants
  - Disk Paxos, Generalised Paxos, EPaxos (see http://paxos.systems/variants.html for ideas)
  - Prove reconfiguration / failure recovery / log truncation / liveness
- Mutual Exclusion Algorithms
  - Knuth's Algorithm, Lamport's Bakery, Patterson, …
  - Prove safety and liveness
- Blockchain algorithms
  - Algorand, HoneyBadgerBFT, Bitcoin-NG, …
- Improve Ivy
  - Experiment with other SMT solvers (e.g. iProver, CVC4, Vampire, SPASS)