Reasoning about Program Data Structure Shape: from the Heap to Distributed Systems

Mooly Sagiv

European Research Council

בית הספר למדעי המחשב על שם בלבטניק
The Blavatnik School of Computer Science

Credits

A. Benerjee

N. Immerman

S. Itzhaky

A. Karbyshev

O. Lahav

K. McMillan

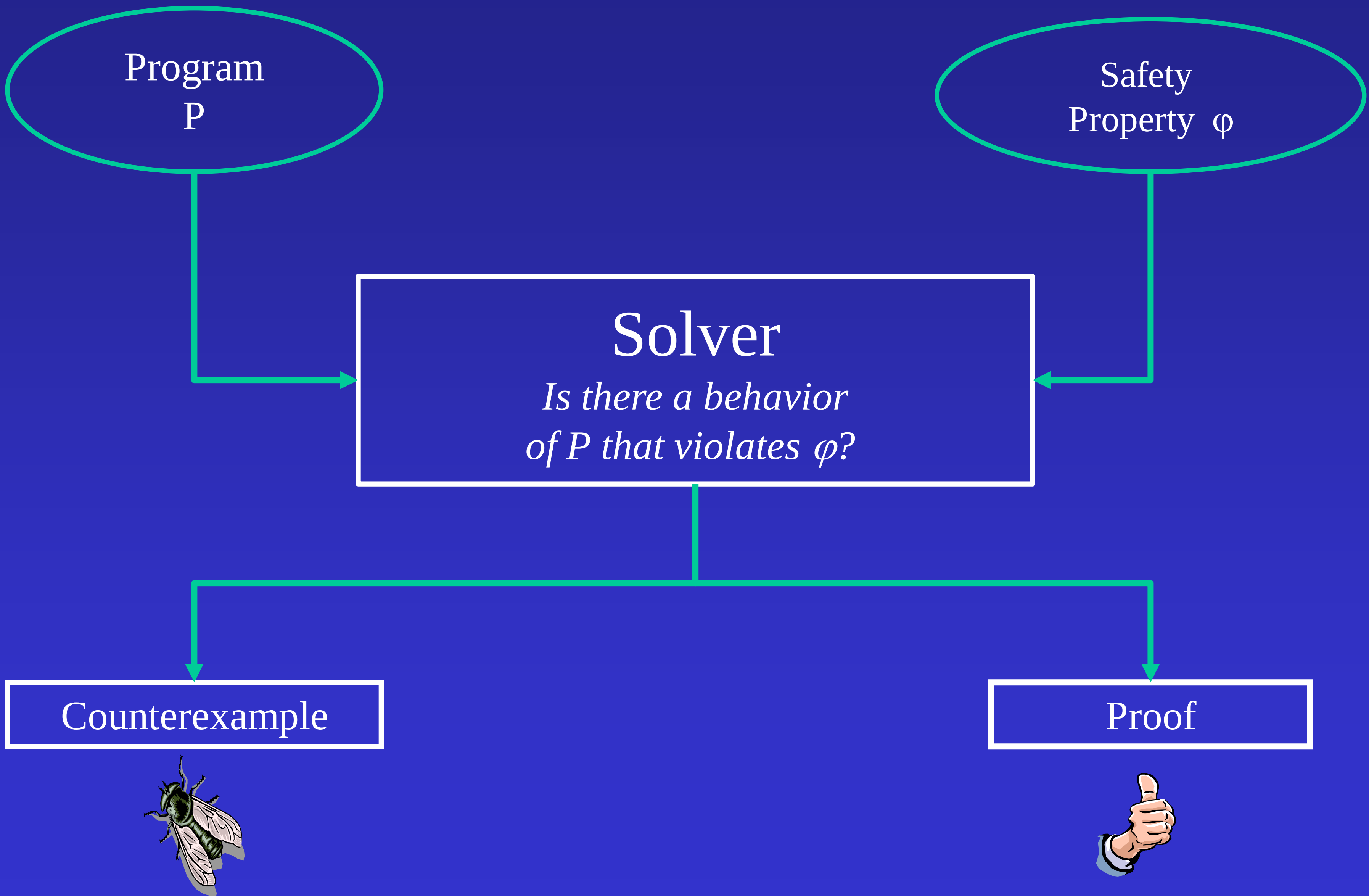
A. Nanevski

O. Padon

A. Panda

S. Shoham

Automatic Program Verification

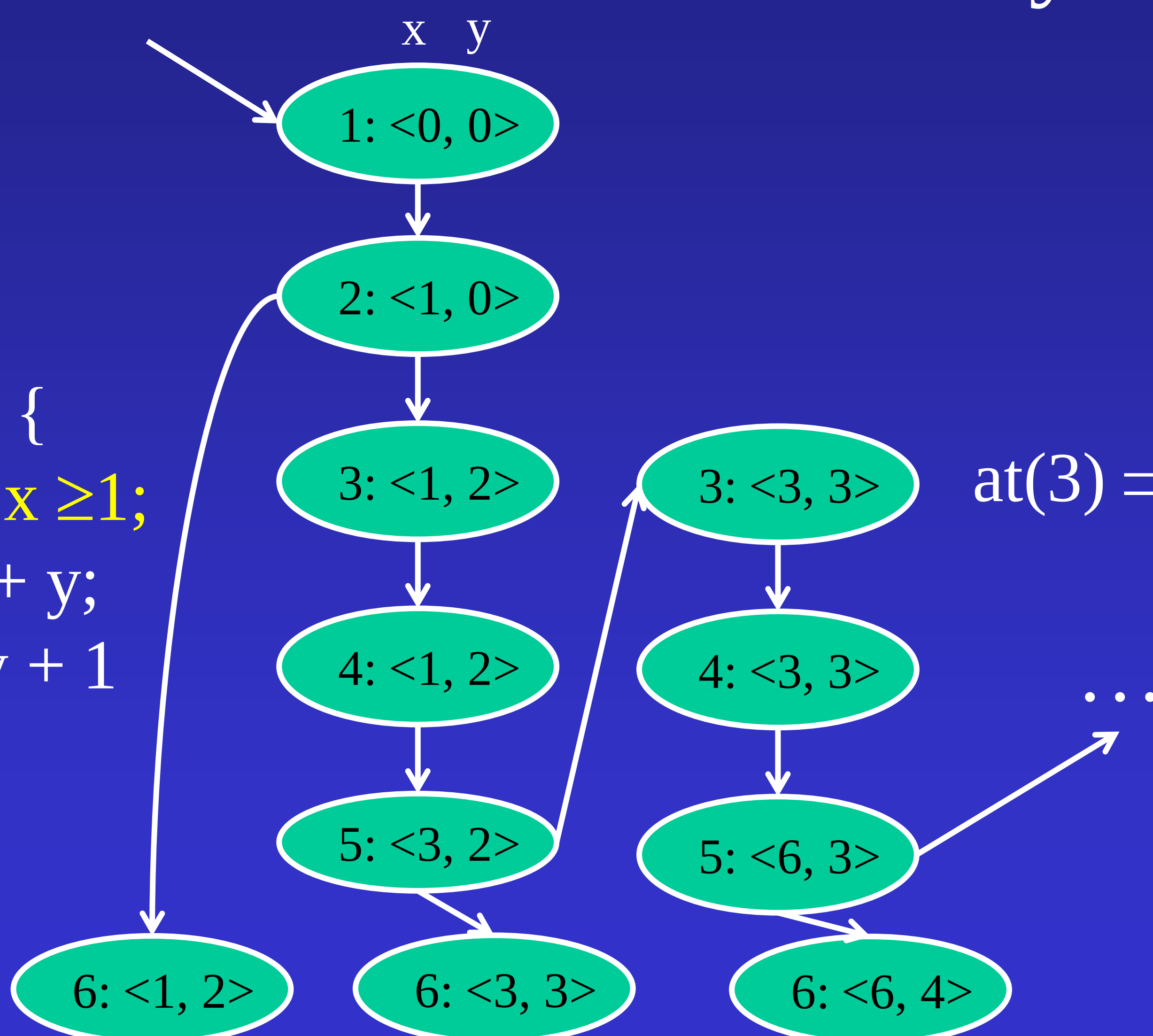


Challenges

1. Specifying safety properties
2. Undecidability of checking interesting properties
    1. The halting problem
    2. Rice theorem
    3. Simple programs can do complicated things

Programs ≈ Infinite Transition Systems

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



at(3) ⇒ x ≥1

Floyd'67

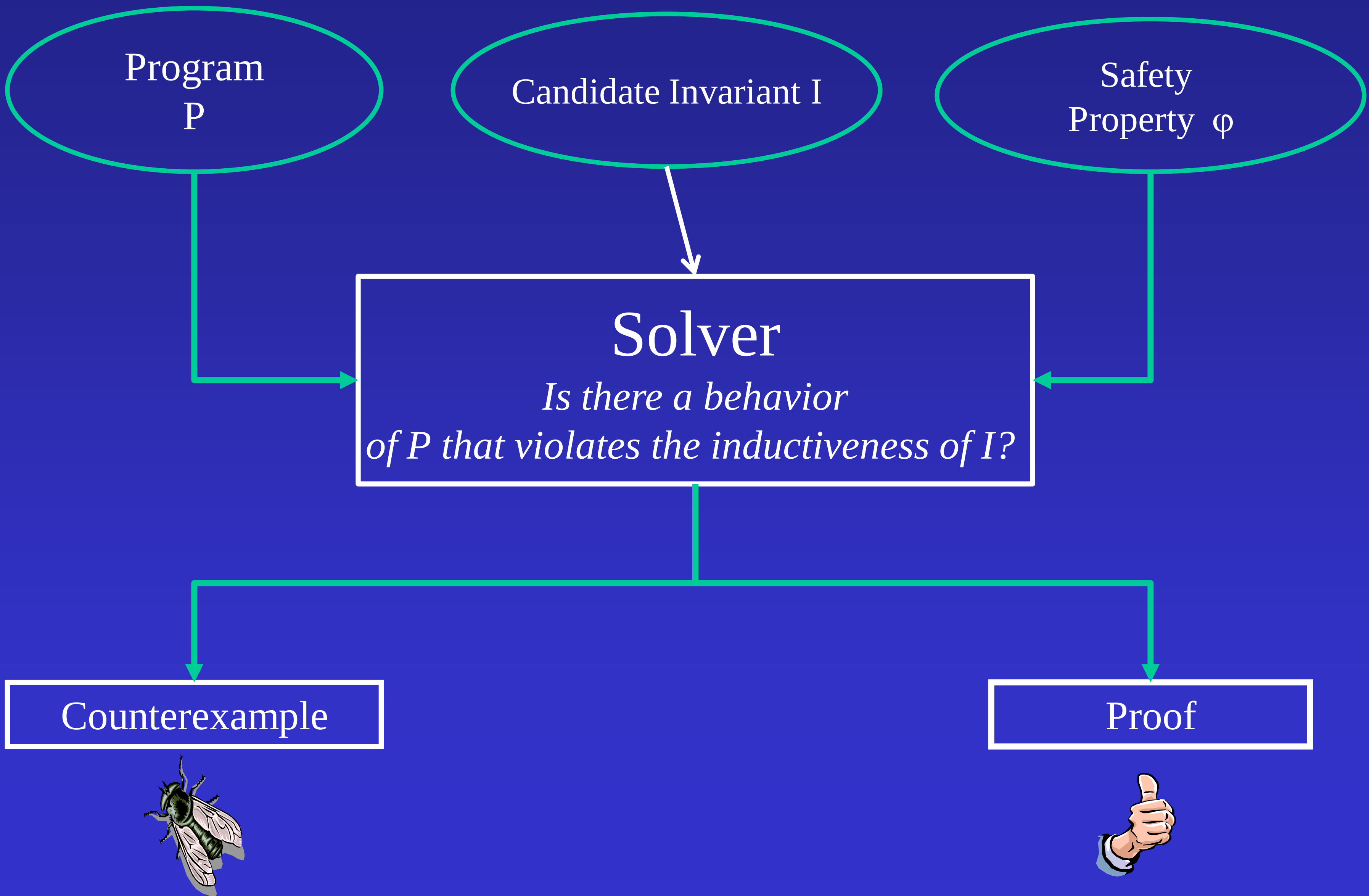
A safety property φ holds in a transition system τ if and only if there exists an inductive invariant I such that

$I \Rightarrow \varphi$ (Safety)

$Init \Rightarrow I$ (Initiation)

if $\sigma \models I$ and $\sigma\ \tau\ \sigma'$ then $\sigma' \models I$ (Consecution)

Semi-Automatic Program Verification



Semi-Automatic Program Verification

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

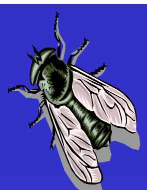
at(3) ⇒ x ≥1

at(3) ⇒ x ≥1

Solver

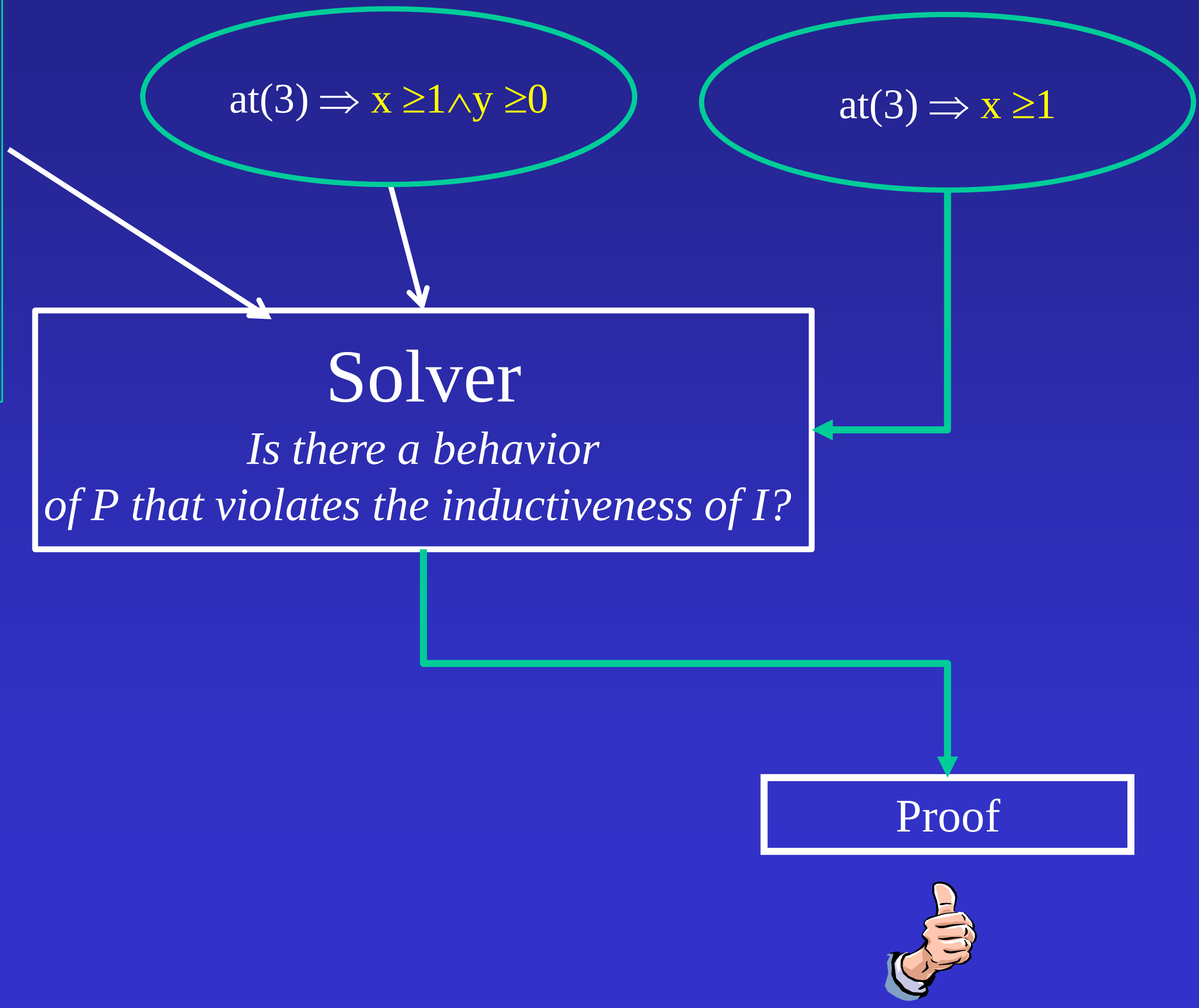
Is there a behavior of P that violates the inductiveness of I?

3: <1, -2>



Semi-Automatic Program Verification

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



Challenges

1. Specifying safety properties
2. Inductive Invariants for Floyd/Hoare style verification
    - Hard to express
    - Hard to change
    - Hard to infer
3. Deduction
    - Reasoning about inductive invariants
        - Undecidability of implication checking

Semi-Automatic Program Verification

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

at(3) ⇒ x ≥1∧y ≥0

at(3) ⇒ x ≥1

Solver

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

Proof

Hard Semi-Automatic Program Verification

```
1: x := 1;
2: y := 2;
while *do {
   3: assert x ≥1;
   4:  x= (x*x-y*y) / (x-y);
   5:  y := y + 1
}
6:
```

at(3) ⇒ x ≥1∧y ≥0

at(3) ⇒ x ≥1

Solver

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

Proof

Challenge 3: Deductive Verification about Reachability

Sound and complete Dafny w/o matching loops

[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
[IVY'15] O. Padon, K. McMillan, A. Panda, M.Sagiv, S. Shoham: Ivy: Interactive verification of parameterized systems via effectively propositional logic

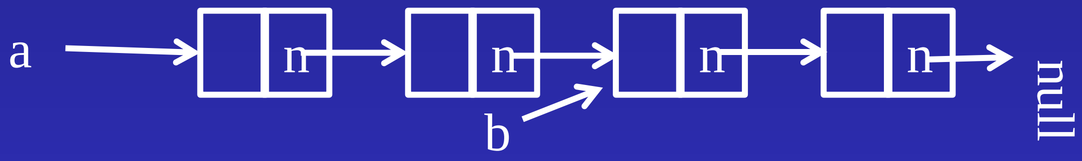
Reasoning about directed reachability in dynamically evolving graphs(relations)

- No garbage
- Preservation of data structure invariants
- Termination
- Reachability properties in distributed protocols
- Even sortedness

Program Termination

{n*(a, b)}

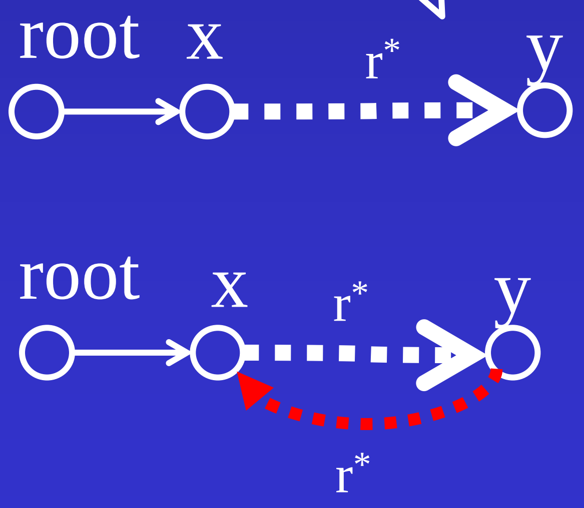
```
traverse(Node a, Node b) {
    for (t =a; t != b ; t = t->n) {
        ...
    }
}
```



Directed Reachability

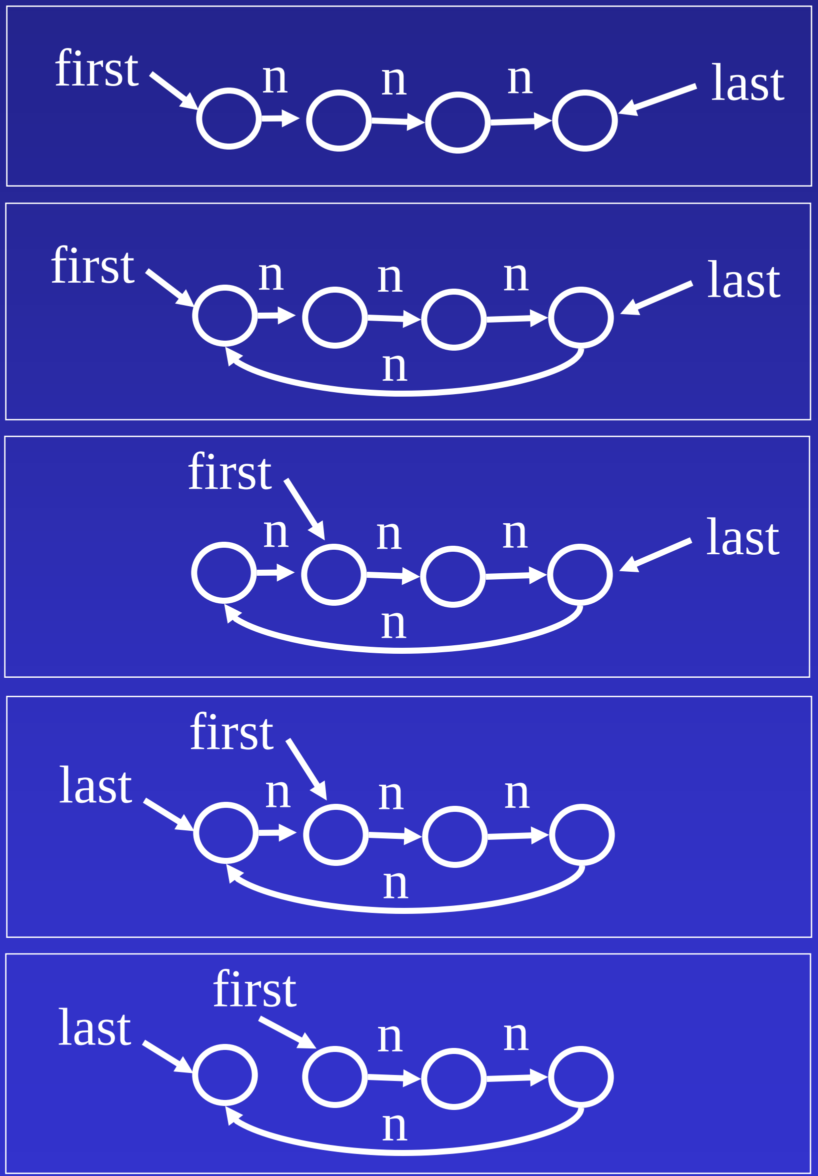
- Directed reachabiliy suffice to describe many properties of data structures
  - Absence of garbage
    - $\forall x: r^*(root, x)$
  - Acyclicity
    - $\forall x: \neg r^+(x, x)$
  - Data Structure Invariants
    - $\forall x: f^*(root, x) \Leftrightarrow b^*(root, x)$

r*(x, y) denotes a finite directed path of relation of r from x to y



Reachability in Dynamically Evolving Graphs

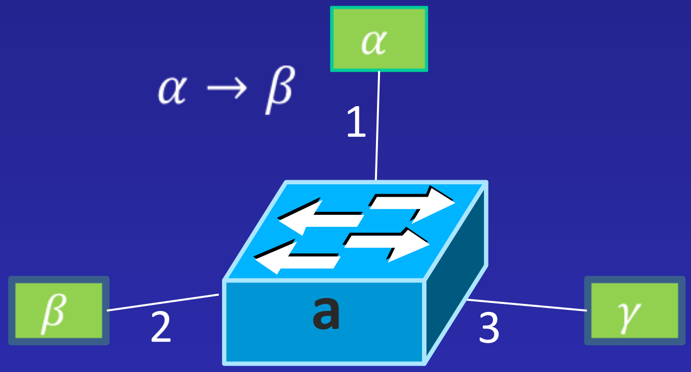
```
rotate(List first, List last) {
    assert acyclic first
    if ( first != NULL) {
        last → next = first;
        first = first → next;
        last = last → next;
        last → next = NULL;
    }
  assert acyclic first;
```



Reachability in Distributed Protocols

- The topology evolves over time
- Reason about evolving relations
- Prove safety
  - Absence of paths
    - Isolation
  - Absence of cycles

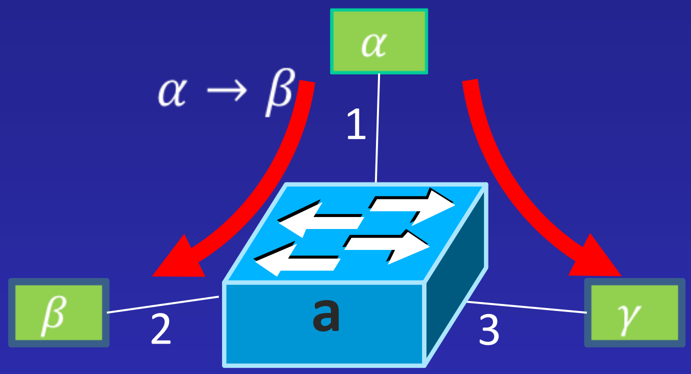
Learning Switch



Input Port	Packet	Output Port

Routing Table	
Dst	Prt

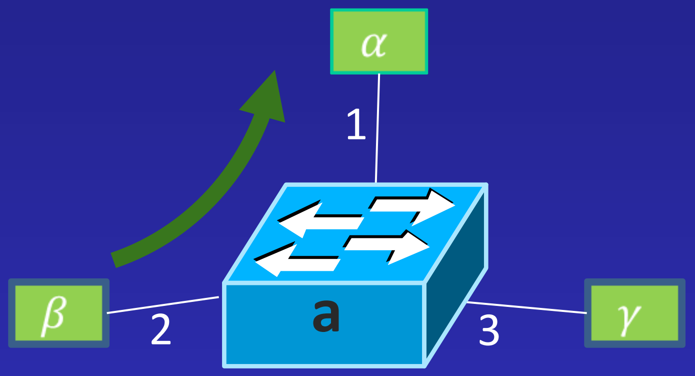
Learning Switch



Input Port	Packet	Output Port
1	α→β	2, 3

Routing Table	
Dst	Prt
α	1

Learning Switch



$\beta \rightarrow \alpha$

Input Port	Packet	Output Port
1	α→β	2, 3
2	β→α	1

Routing Table	
Dst	Prt
α	1

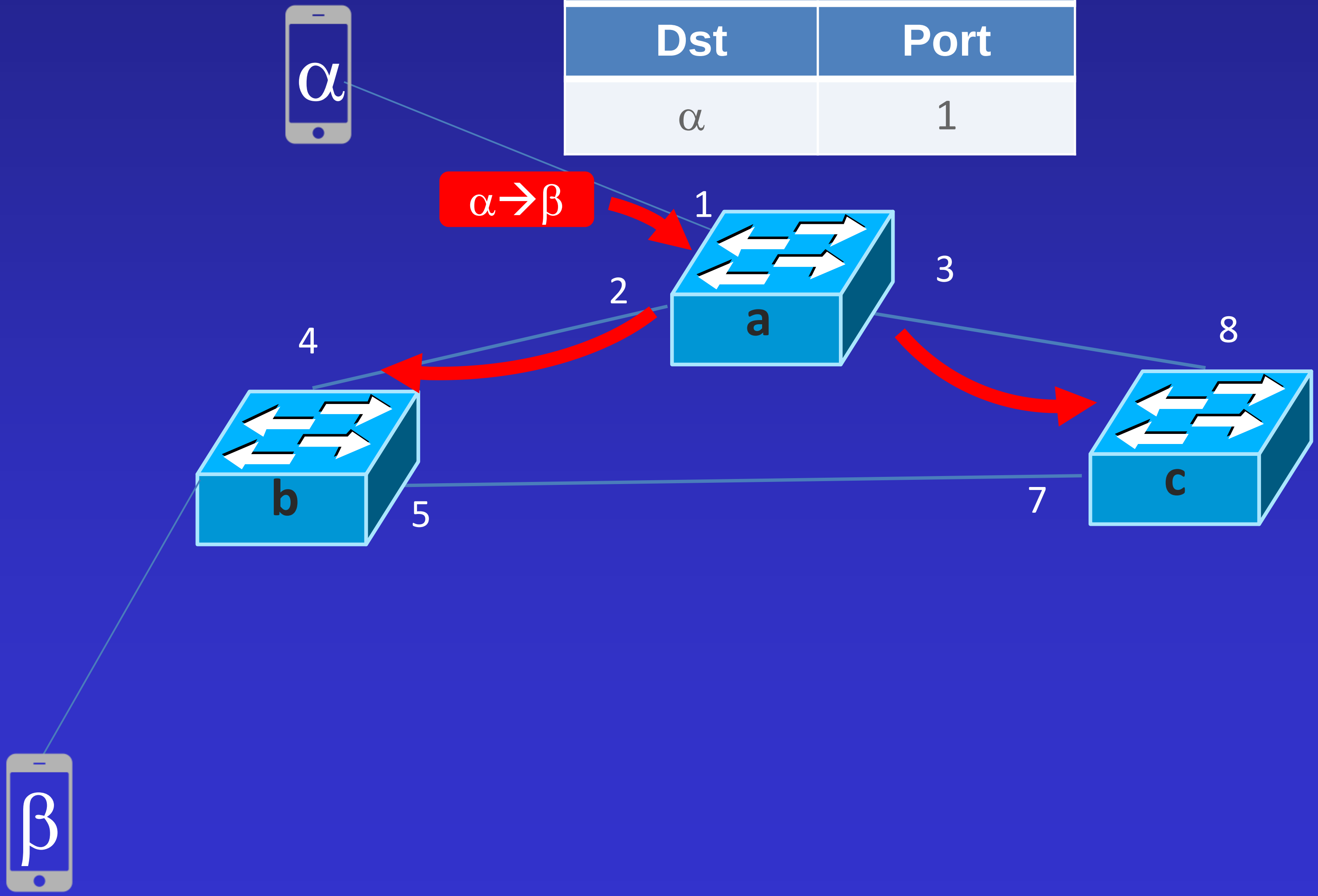
Learning Switch Code

```
event receive =
     <p: packet, m: node> ∊ pending ➔
        pending.remove <p, m>
        route[p.src] := {p.ingress}; // learn
        exists pr : route[p.dst] = {pr} ➔
             forward p to pr // adds new tuple to pending
        route[p.dst] = {} ➔
             flood p // adds new tuples to pending
        assert acyclic forall Dst: route[Dst];
```

Verification can identify a topology in which a forwarding loop in the routing table occur

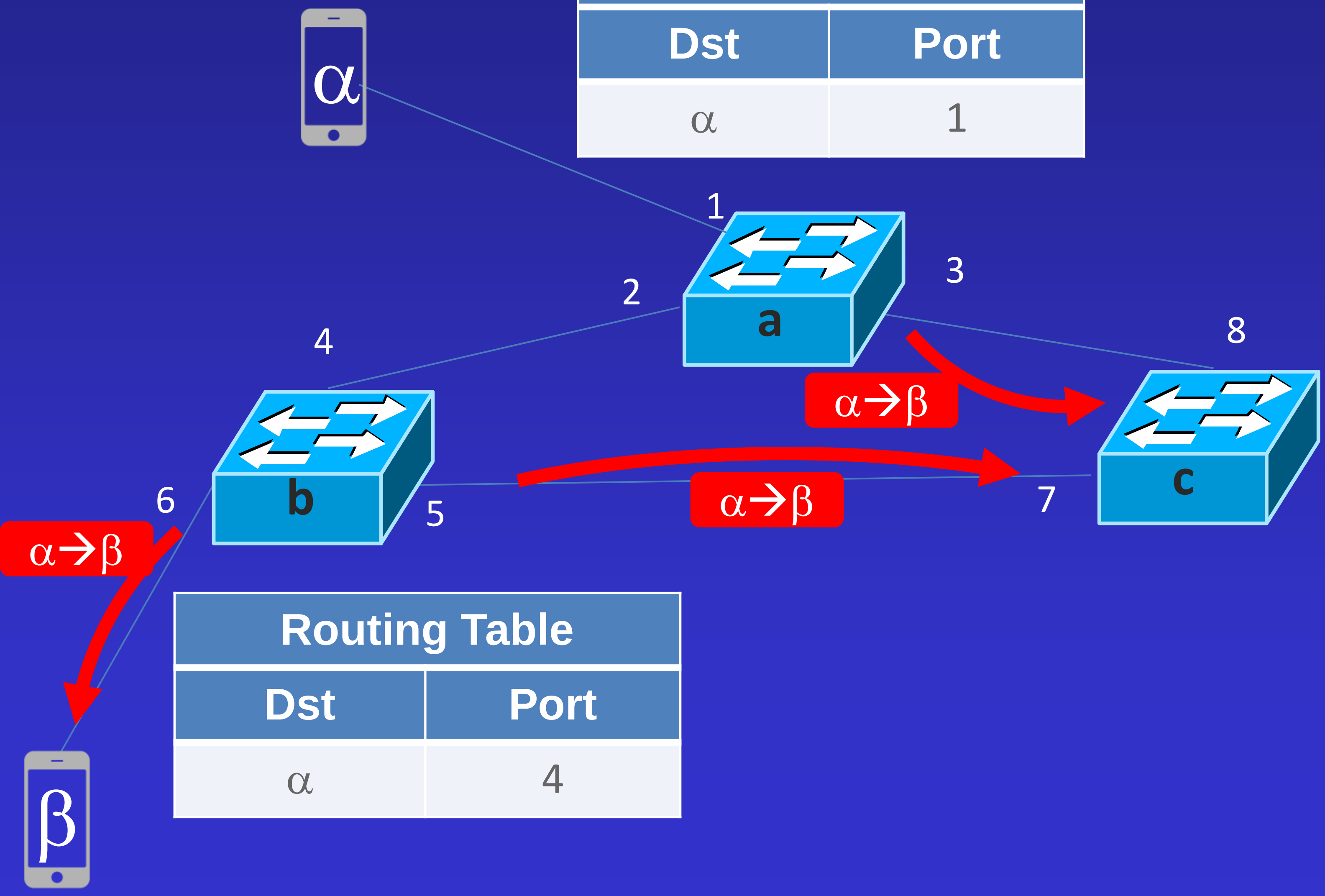
A Forwarding Loop

Routing Table	
Dst	**Port**
α	1



A Forwarding Loop

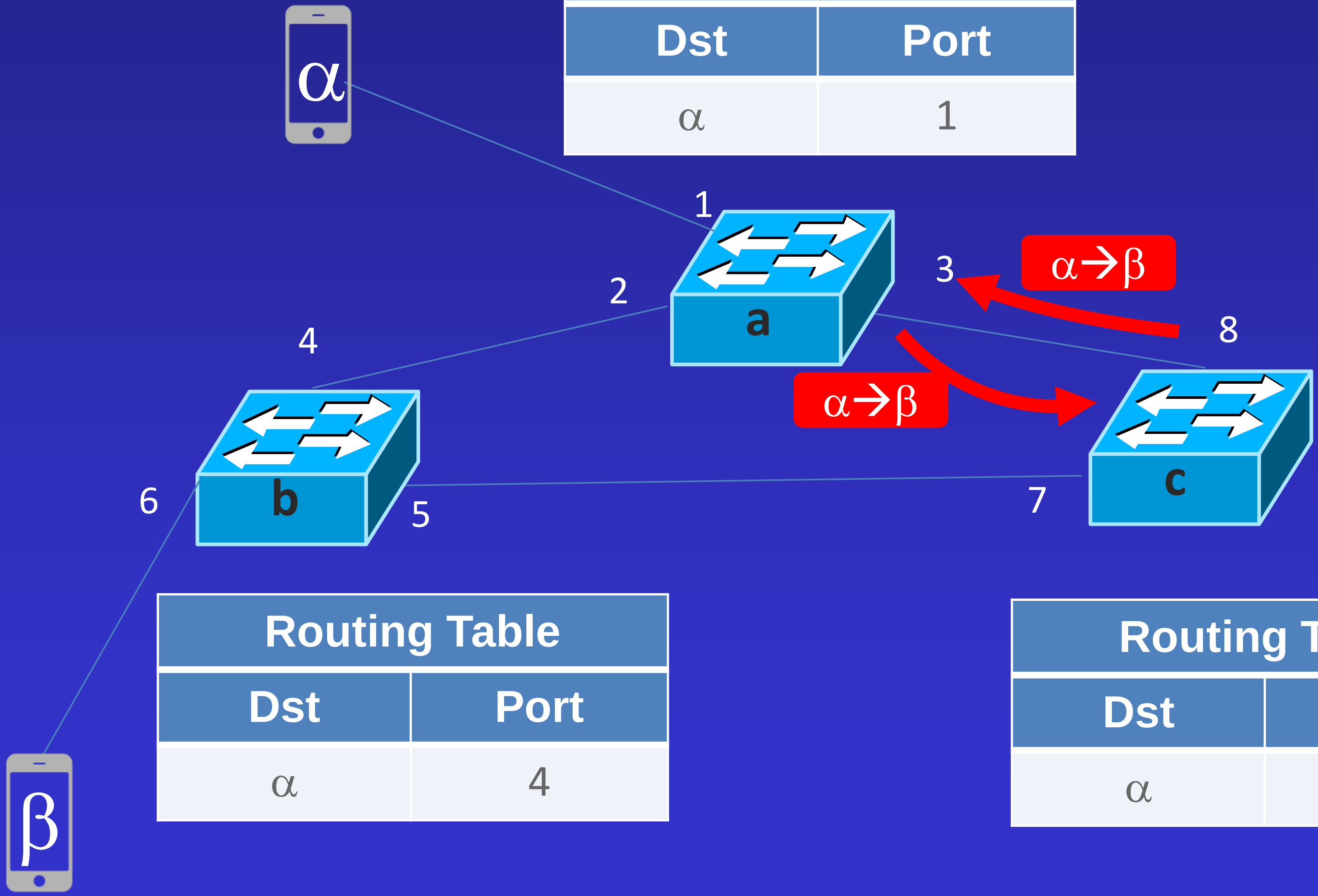
Routing Table	
Dst	**Port**
α	1



Routing Table	
Dst	**Port**
α	4

A Forwarding Loop

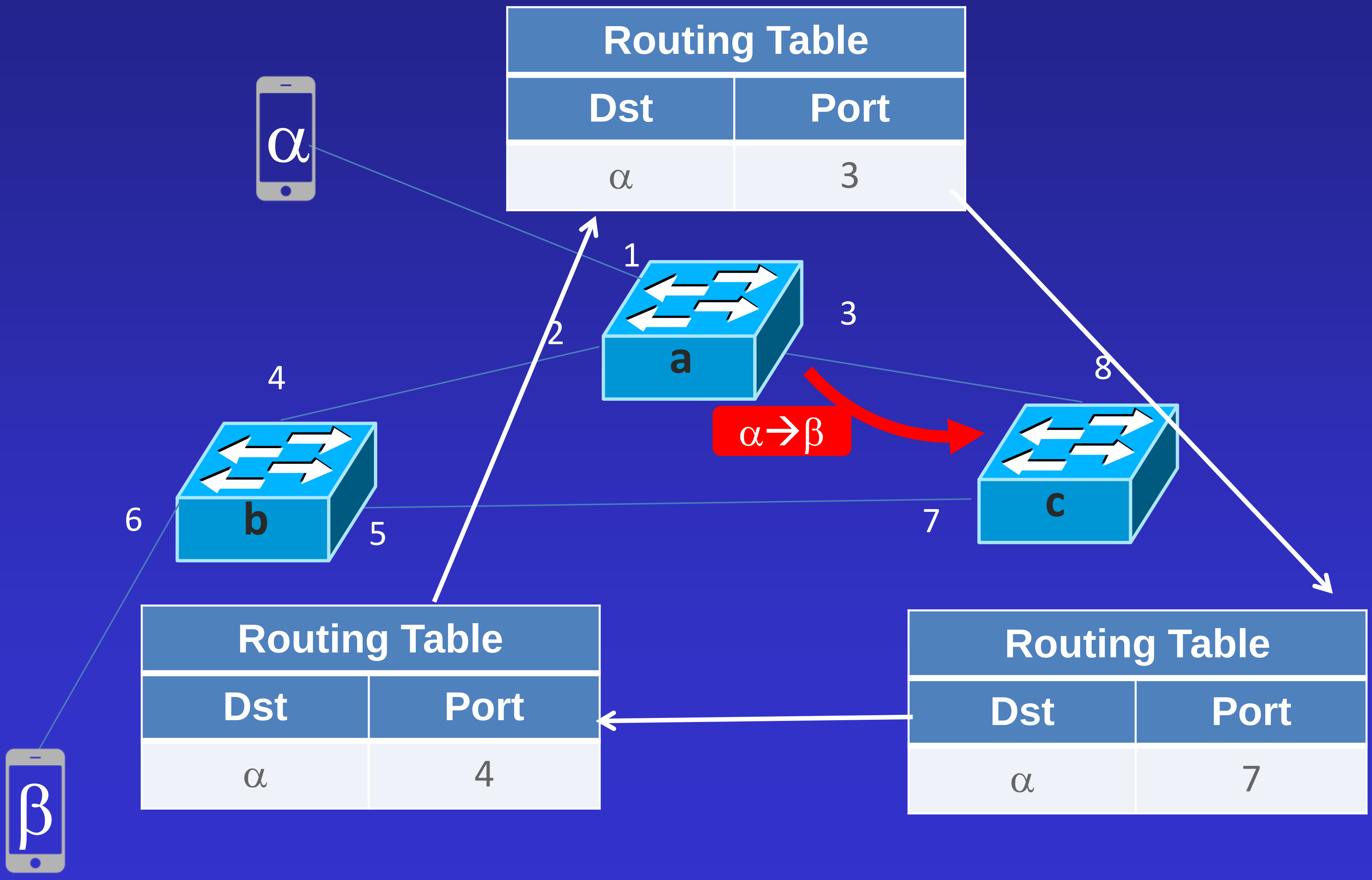
Routing Table	
Dst	**Port**
α	1



Routing Table	
Dst	**Port**
α	4

Routing Table	
Dst	**Port**
α	7

A Forwarding Loop



Loop-Free Learning Switch Code

```
event receive =
    <p: packet, m: node> ∊ pending ➔
      pending.remove <p, m>
      route[p.src] = {} ➔
           route[p.src] := {p.ingress} // learn
     exists pr : route[p.dst] = {pr} ➔
           forward p to pr // adds new tuple to pending
     route[p.dst] = {} ➔ // flood
           flood p // adds new tuples to pending
     assert acyclic forall Dst: route[Dst];
```

Verification proves the absence of forwarding loops for arbitrary topologies 

Challenges

- Complexity of reasoning about reachability assertions
  - Not first order expressible
  - Undecidability of reachability (not even RE)

"*there is a mismatch between the simple intuitions about the way pointer operations work and the complexity of their axiomatic treatments*"

O'Hearn, Reynolds, Yang [CSL 2001]

- [Inferring reachability properties from the code]

Do I have to Solve Hilbert's 10^{th} problem?

```
count {
   List a =NULL, b=NULL, t;
   int c = 0 ; read(c);
   while (c > 0) {
        t = malloc();  t→next = a;   a = t ;
        t = malloc();  t→next = b;   b = t;
        c--; }
   while (a != null) {
        assert a!=null; print(a→d);
        assert b!=null; print(b→d);  }
}
```

Jackson's Thesis

- If a program has a bug $\Rightarrow$ it also occurs on small input k
  - True in many cases
  - But
    - ☹ What if not?
    - ☹ Hard to find k
    - ☹ Hard to scale checking to k

Itzhaky’s thesis: Linked list manipulations are simple

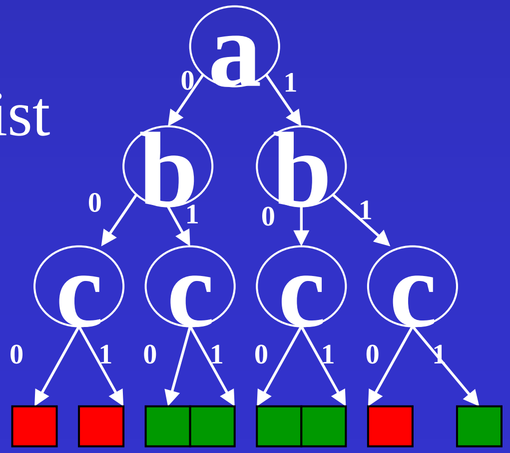
- Simple to reason about correctness
  - Small counterexamples
- Deterministic paths
- Even for doubly/circular/nested lists/distributed protocols
  - Sortedness
  - Size
- “Simple” inductive invariants suffice to show safety
  - Alternation Free + Reachability “$\subseteq$” $\exists^*\forall^*$

Do I have to Solve Hilbert's 10th problem?

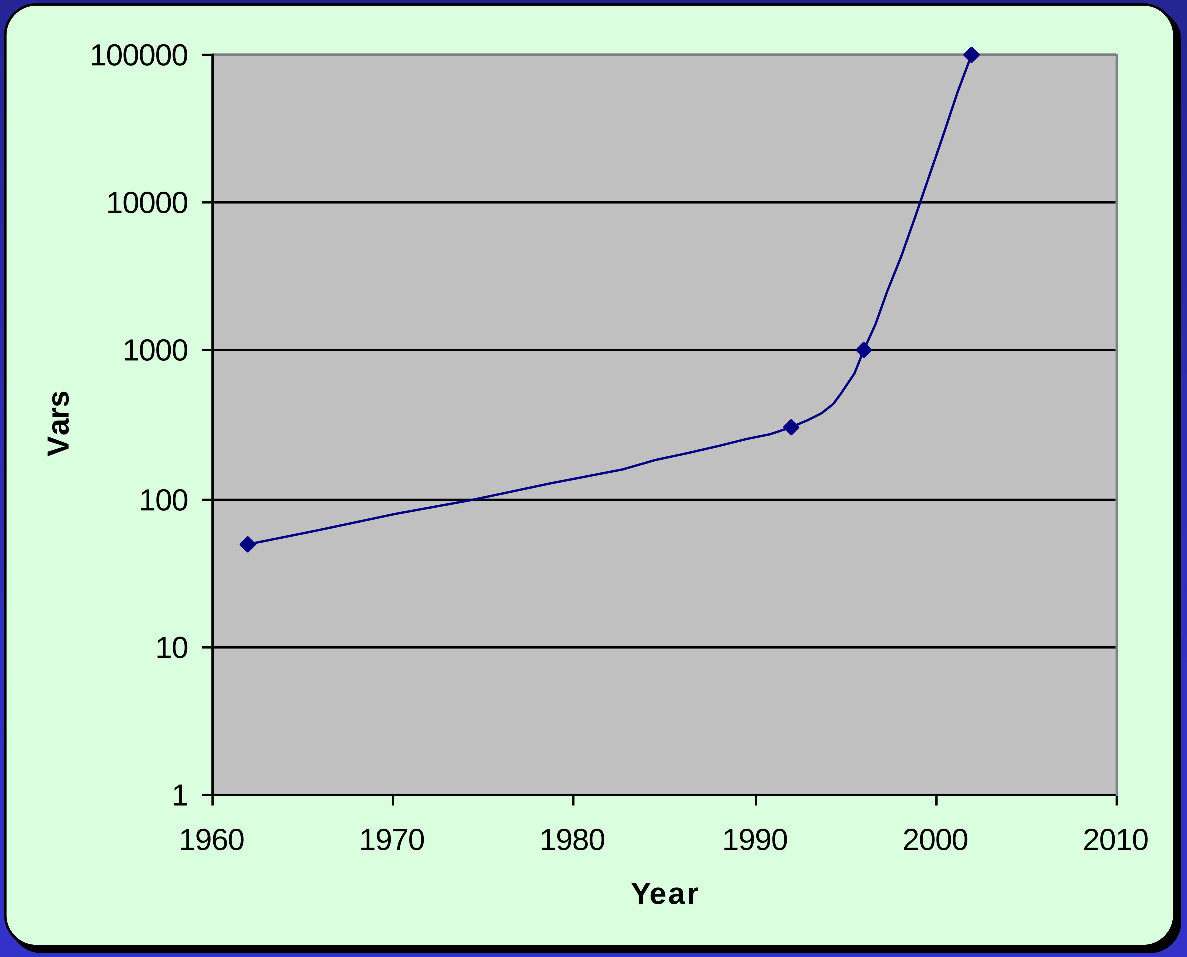
```
count {
  List a =NULL, b=NULL, t;
  int c = 0 ; read(c);
  while (c > 0) {
      t = malloc();  t→next = a;   a = t ;
      t = malloc();  t→next = b;   b = t;
      c--; }
  while (a != null) {
      assert a!=null; print(a→d);
      assert b!=null; print(b→d);  }
}
```

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 valid
- Determine if φ is satisfiable
  - Find a satisfying assignment or report that such does not exit
- For *n* variables, there are 2^n possible truth assignments to be checked
- But many practical tools exist

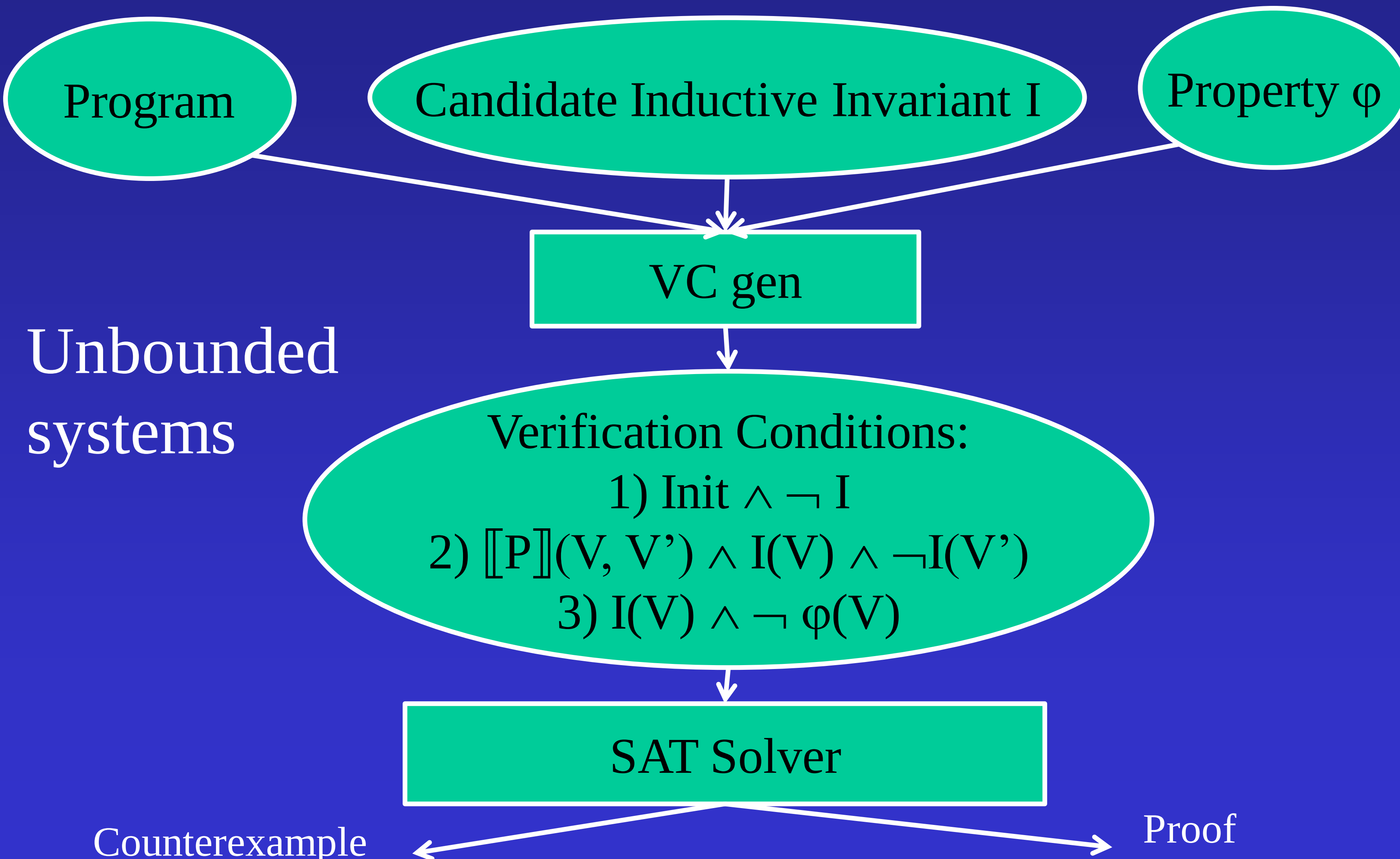


SAT made some progress…



Semi-Automatic Verification Process

Unbounded systems



(Uninterpreted Relational) First Order Logic w/o functions

$t ::= c$		Constant symbol
$\mid\ x$		Logical variable
$\varphi ::= r(t_1, t_2, \ldots, t_n)$		Relation
$\mid\ t_1 = t_2$		Equality
$\mid\ \exists x.\ \varphi$		Existential Quantification
$\mid\ \forall x.\ \varphi$		Universal Quantification
$\mid\ \varphi_1 \vee \varphi_2$		Disjunction
$\mid\ \varphi_1 \wedge \varphi_2$		Conjunction
$\mid\ \neg\varphi$		Negation

SAT becomes undecidable

- $\forall x.\ le(x, x)$ 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
- $\forall x.\ \exists y.\ le(x, y) \wedge x \neq y$

SAT becomes ~~un~~decidable

- $\forall x.\ le(x, x)$ 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
- ~~$\forall x.\ \exists y.\ le(x, y) \wedge x \neq y$~~

Effectively Propositional Logic – EPR
a.k.a. Bernays-Schönfinkel-Ramsey class

- Fragment of first-order logic
  - Restricted quantifier prefix: $\exists^* \forall^* \varphi_{Q.F.}$
  - No function symbols
- Small model property
  - $\exists x_1, \ldots, x_n. \forall y_1, \ldots, y_m. \varphi_{Q.F.}$ has a model iff it has a model of at most n+k elements (k - number of constant symbols)

- Satisfiability is decidable
  - NEXPTIME
- Support from Z3

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

Can we reason about interesting properties with EPR?

Some parts have to be provided by domain experts for a class of programs

Axioms provided by domain experts

Semi-Automatic Program Verification

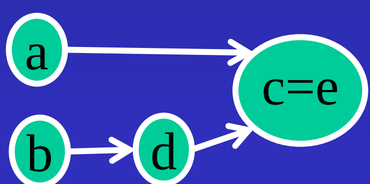
```
assume ∀x. ¬ ( n*(a,x) ∧ n*(b,x) )
c := a->n;
d := b->n;
e := d->n;
assert c != e;
```

c != e

Is there a behavior of P in which c=e?

$\forall x.\ \neg\ (\ n^*(a,x) \wedge n^*(b,x)\ ) \wedge$
$n(a, c) \wedge n(b, d) \wedge n(d, e) \wedge c{=}e$

SAT Solver (Z3)



n = {(a,c), (b,d), (d,c)}
n* = { }

Counterexample

Complete Reasoning about Deterministic Paths

- $n^*(x, x)$ Reflexivity
- $n^*(x, y) \wedge n^*(y, z) \Rightarrow n^*(x, z)$ Transitivity
- $n^*(x, y) \wedge n^*(y, x) \Rightarrow x = y$ Acyclicity
- $n^*(x, y) \wedge n^*(x, z) \Rightarrow n^*(y, z) \vee n^*(z, y)$ Linearity
- $n^+(x, y) \equiv n^*(x, y) \wedge x \neq y$
- $n(a, b) \equiv n^+(a, b) \wedge \forall x: n^+(a, x) \Rightarrow n^*(b, x)$

[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

Semi-Automatic Program Verification

```
assume ∀x. ¬ ( n*(a,x) ∧ n*(b,x) )
c := a->n;
d := b->n;
e := d->n;
assert c != e;
```

c != e

Is there a behavior of P in which c=e?

axioms $\wedge$
$\forall x. \neg (n^*(a,x) \wedge n^*(b,x)) \wedge$
“n(a, c)” $\wedge$ “n(b, d)” $\wedge$ “n(d, e)” $\wedge$ c=e

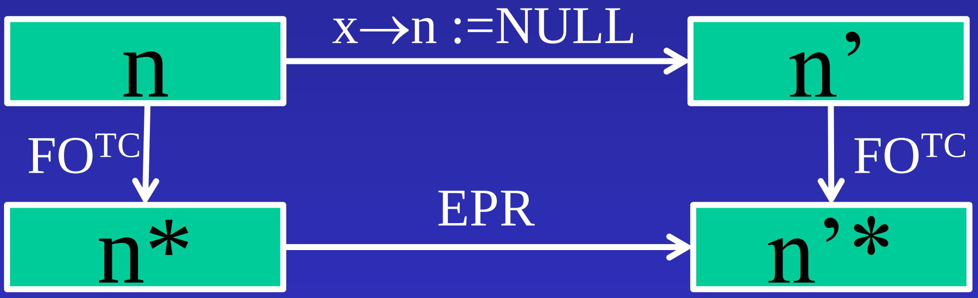
SAT Solver (Z3)

Proof

But how can we model the program in EPR?

- The program updates edge relations
- The compiler generates EPR formulas to update paths
- This can always be done

Incremental
Simple updates



Benchmark	Formula Size						Solving time (Z3)
	P,Q		I		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

Summary thus far

- Reduced the undecidable problem of checking inductiveness to the NEXPTIME problem of checking EPR satisfibility
  - Efficient in practice
  - Useful for bounded model checking
  - Useful for synthesis
- But what about inferring EPR invariants?

Automatically Inferring EPR Invariants

- PDR/IC3 procedure for inferring universal invariants [CAV'15]
- Inferring universal invariants for linked-lists is decidable [POPL'16]
- Systematic extensions for decidability of some distributed protocols [POPL'16]
- Inferring general universal invariants is undecidable [POPL'16]
- Inferring alternation-free invariants for linked-lists is undecidable [POPL'16]

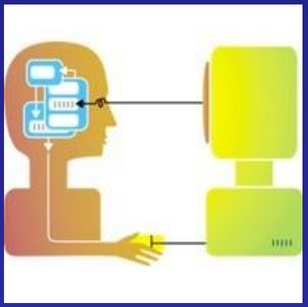
[CAV'15] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, S. Shoham: Property-directed inference of universal invariants or proving their absence

[POPL'16] O. Padon, N. Immerman, A. Karbyshev, S. Shoham, M. Sagiv Decidability of inferring inductive invariants

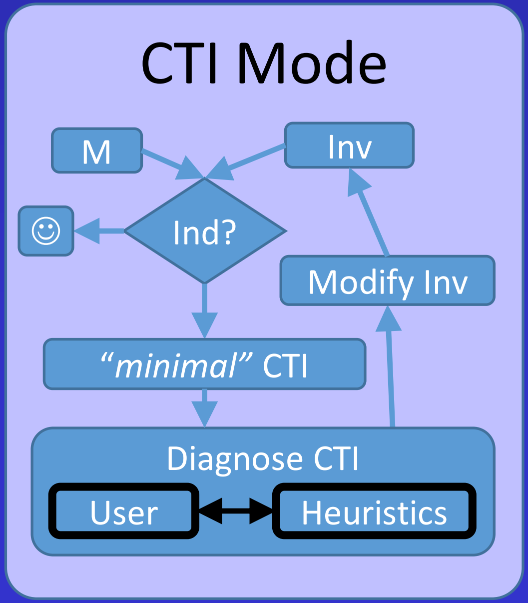
Ivy: Interactive Verification via EPR

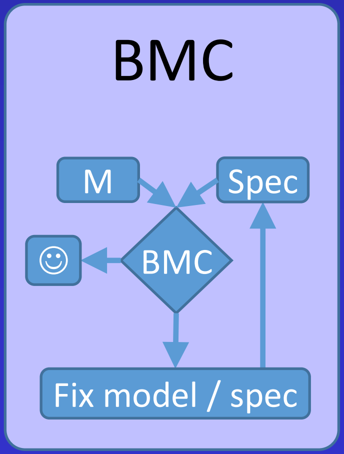
Goal: Engage the user in automated verification

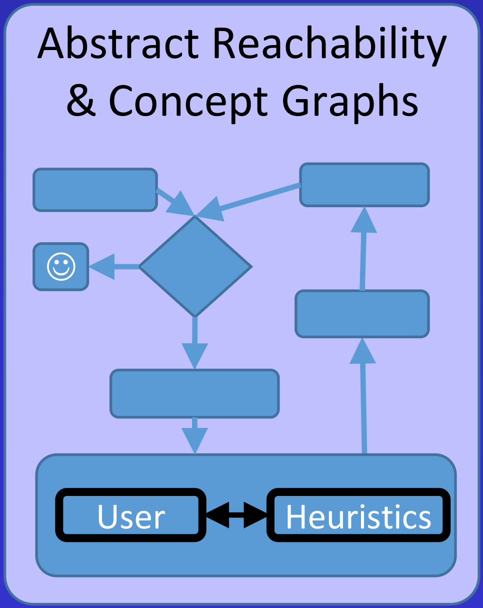
- Use powerful invariant generation heuristics interactively
- Bidirectional feedback between user and heuristics

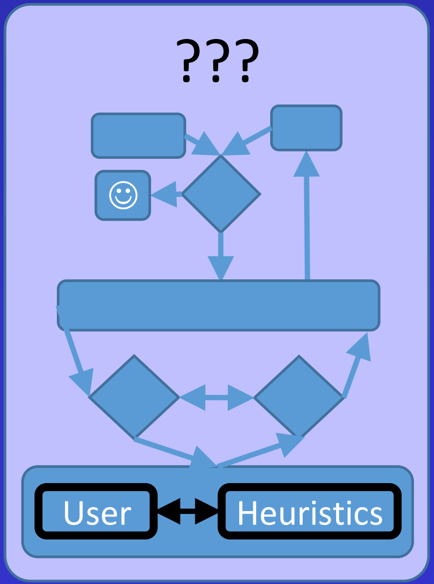


- Questions:
  - What *decidable problem* should we let the machine solve?
  - What is a useful *interaction mode* between the user and the machine heuristics?





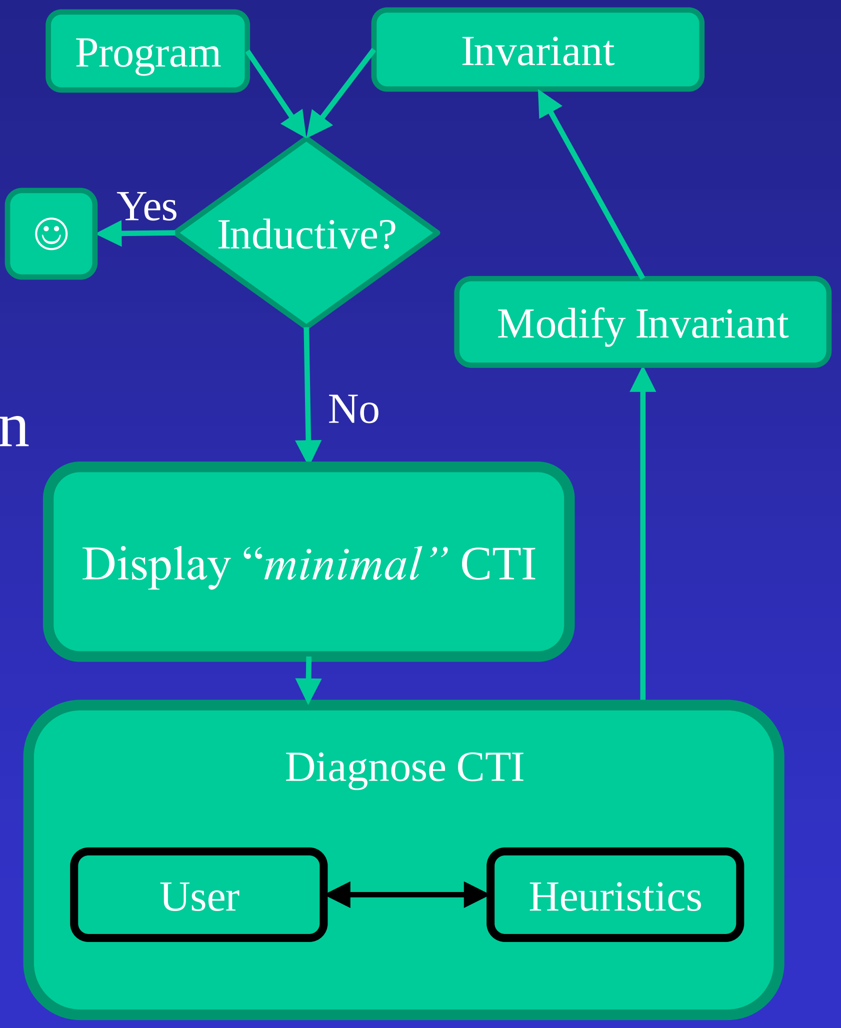




Heuristics for User Interaction

Exploit EPR

- Carefully select CTI
  - Minimize certain "metrics"
- Interactive Generalization
  - Select visible relations
  - Gather facts from user selection
  - BMC
    - Check conjecture
    - Minimize conjecture
  - Sufficiency for current failure
  - Relative inductiveness



Summary

- EPR is useful to reason about infinite state systems
  - BMC
  - Inductive invariants
  - Effective reasoning about TC
- Exploit simplicity of quantifier free updates in distributed systems
- The next challenge is invariant inference

BACKUP SLIDES

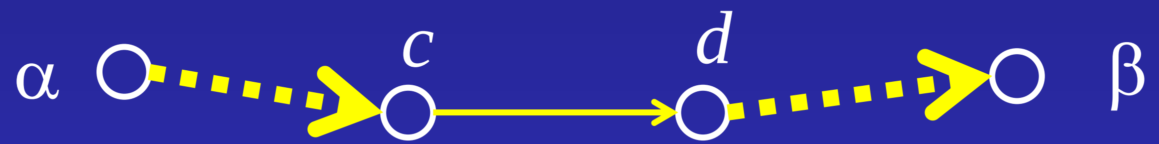
Some Related Work

- Monadic second order logic [CIAA’00] [SAS’11]
- Decidable separation logic
- Sound first order axioms

[CIAA’00] N. Klarlund, A. Møller, M. I. Schwartzbach:
MONA implementation secrets. CIAA 2000
[SAS’11] P. Madhusudan, X. Qiu:
Efficient decision procedures for heaps using STRAND. SAS 2011

Updating Reachability

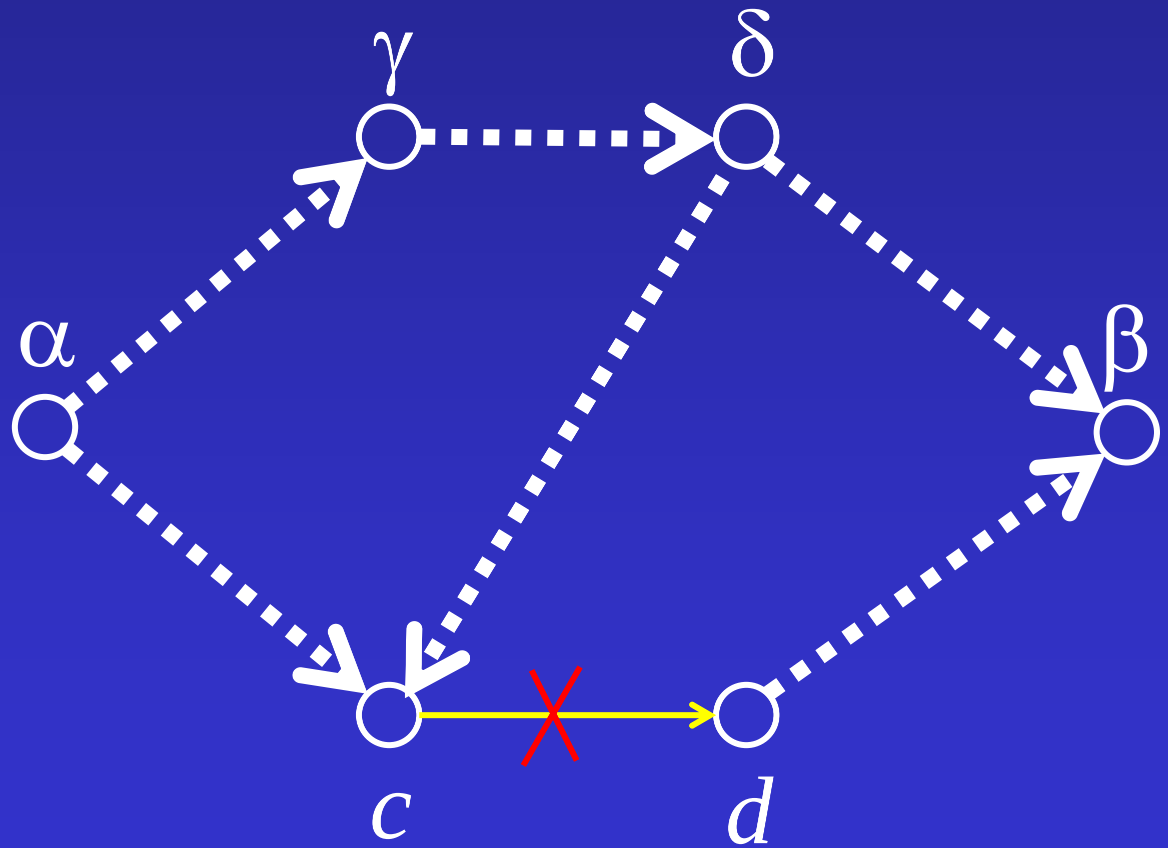
Adding an edge
$c \rightarrow n = d$



assert $\neg n^*(\beta, \alpha)$

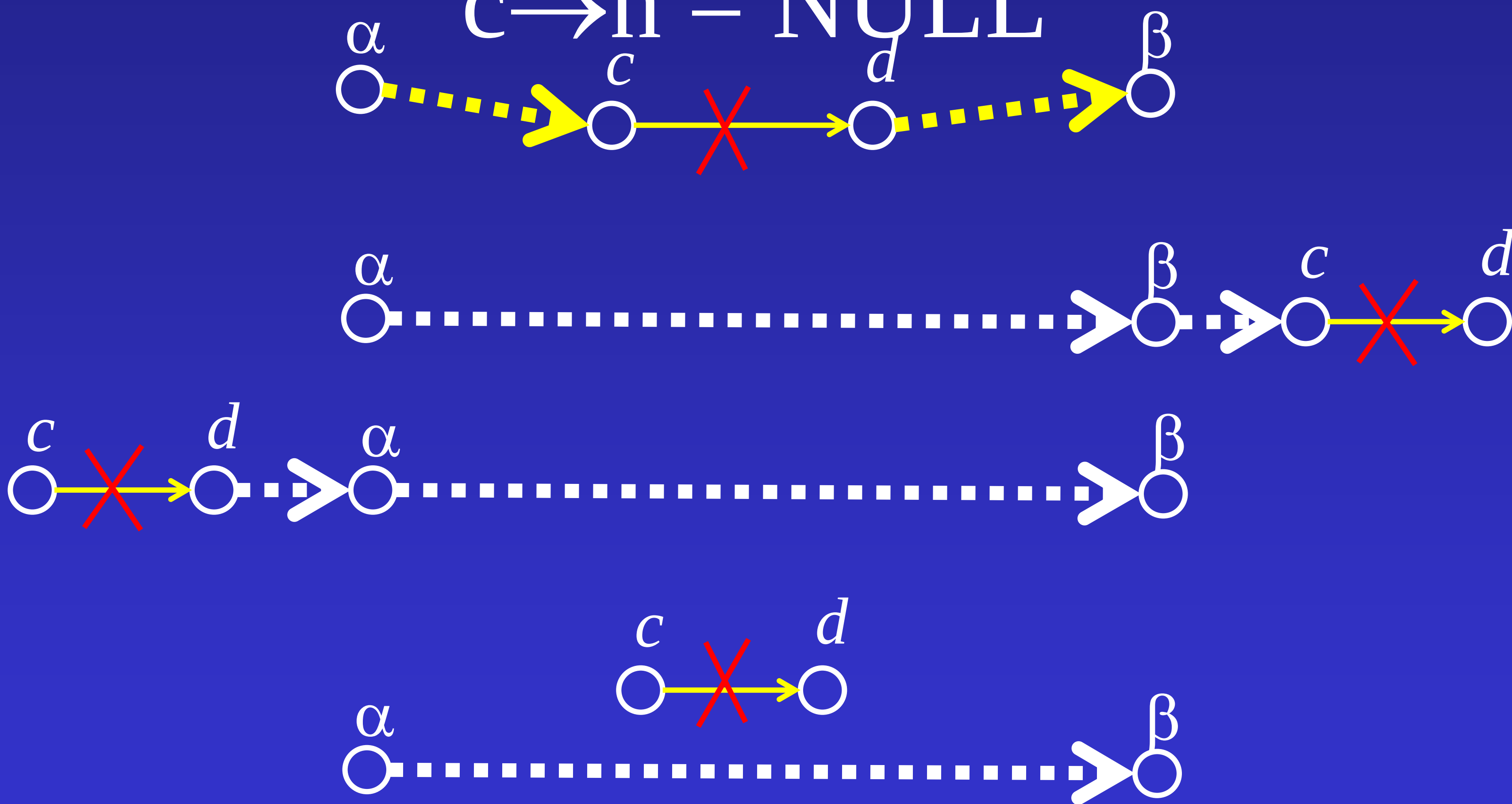
$n'^*(\alpha,\beta) \leftrightarrow n^*(\alpha,\beta) \vee (n^*(\alpha, c) \wedge n^*(d,\beta))$

Updating Directed Reachabilty in General Graph is Hard



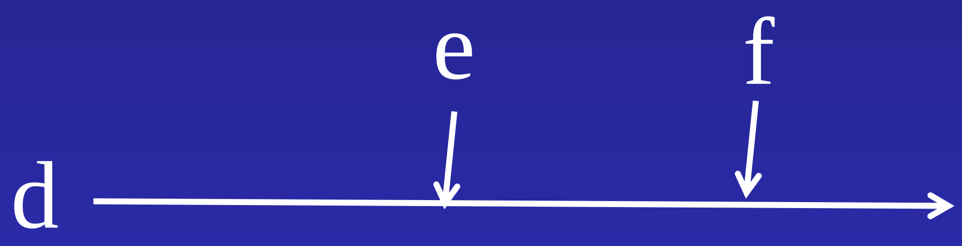
Removing an edge (destructive update)

c→n = NULL



$$n'^*(\alpha,\beta) \leftrightarrow n^*(\alpha,\beta) \wedge \neg(n^*(\alpha,c) \wedge n^+(c,\beta))$$

Traversing an edge
c = d→n (c is fresh)



$n^+(d,c) \wedge$

$\forall x\colon\ n^+(d,x) \Rightarrow n^*(c,x)$

Reasoning about Distributed Protocols

- The correctness of very simple distributed protocol can be tricky
  - Safety, Consensus, Serializability, Liveness
  - Widely used
- Examples: Raft, Paxos, Chord
- Unlimited resources
- Counterintuitive reasoning
- Topology affects correctness

Beyond EPR

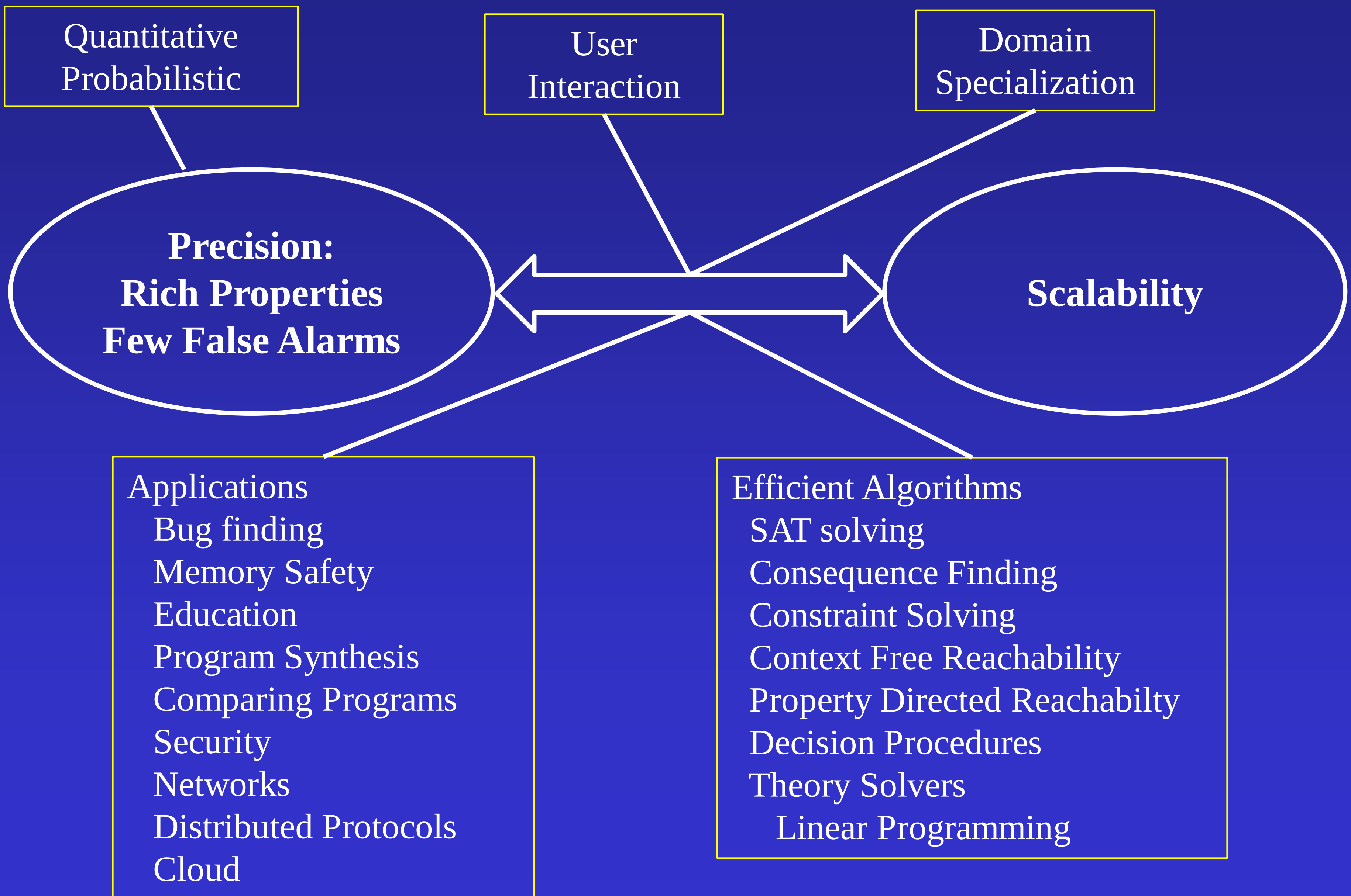
- EPR cannot force the existence of unbounded sets
- Non-emptyness of the routing relations
- Hole-punching firewall

The Instrumentation Principle

- Users define extra derived relations
- Expressible outside EPR
- The system generates update formulas
- Guaranteed soundness
- Completeness no longer guaranteed
  - But concrete states are precise

[TOPLAS'10] T.W. Reps, M. Sagiv, A. Loginov:
Finite differencing of logical formulas for static analysis

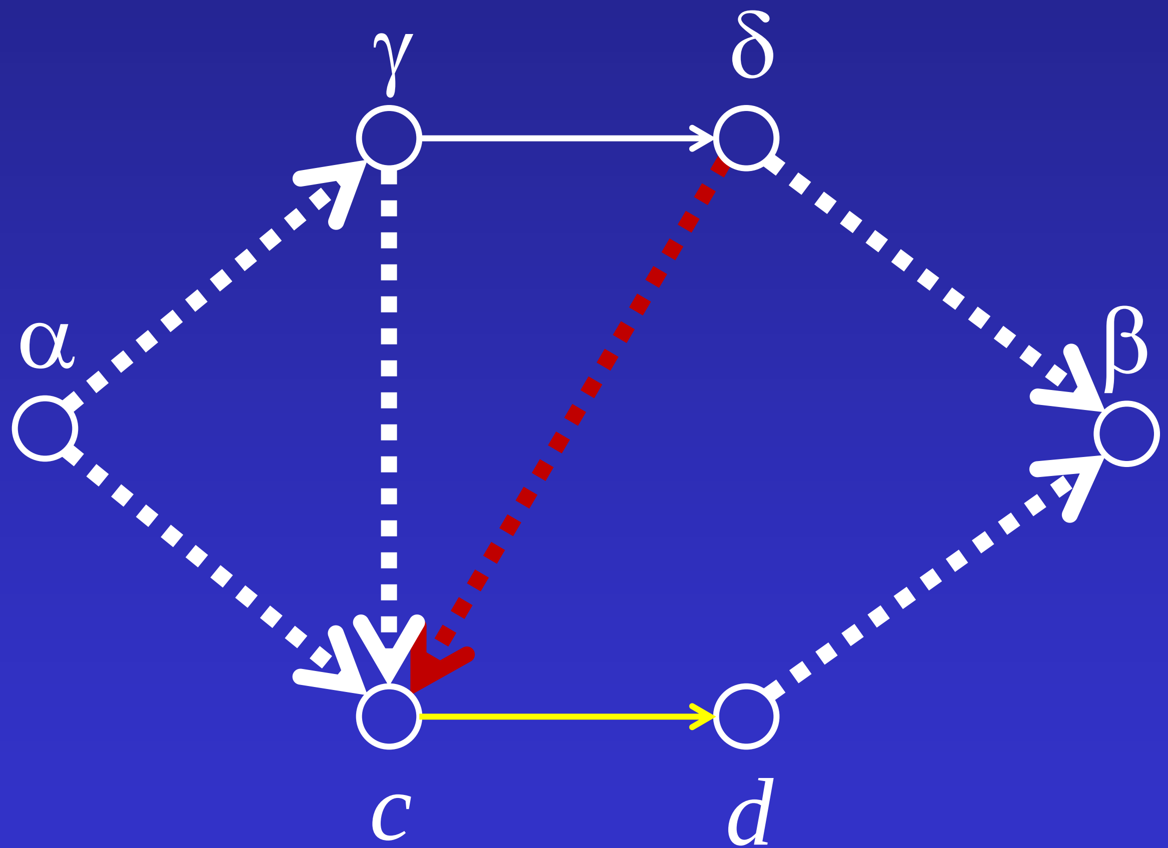
The Static Analysis Tradeoff



Summary

- Domain specific verification/static analysis
- Symbolic reasoning on directed reachability can be useful for verification and bug finding in
  - Linked data structures
  - Distributed systems
- Much more need to be done
  - Invariant Inference
  - Efficient decision procedures

Dong & Su [SIGMOD'00] DAG



$\exists\gamma$: $\alpha\langle n^*\rangle\gamma \wedge \gamma\langle n^*\rangle c \wedge$

$n(\gamma)=\delta \wedge \delta\langle n^*\rangle\beta \wedge \neg\delta\langle n^*\rangle c$

Loop-Free Learning Switch Code

```
event receive =
     <p: packet, m: node> ∈ pending ➔
        pending.remove <p, m>
        route[p.src] = {} ➔
             route[p.src] := {p.ingress} // learn
       exists pr : route[p.dst] = {pr} ➔
             forward p to pr // adds new tuple to pending
       route[p.dst] = {} ➔ // flood
             flood p // adds new tuples to pending
      assert acyclic forall Dst: route[Dst];
```

$\forall$dst, node1, node2:

route[node2, dst] $\neq$ {} $\rightarrow$ $\neg$path[dst](node1, node2)

Expressible in a weak decidable logic $\exists^*\forall^*$