Exploiting Synchrony and Symmetry in Relational Verification

Lauren Pick

Relational Verification

Relational Verification

Given:

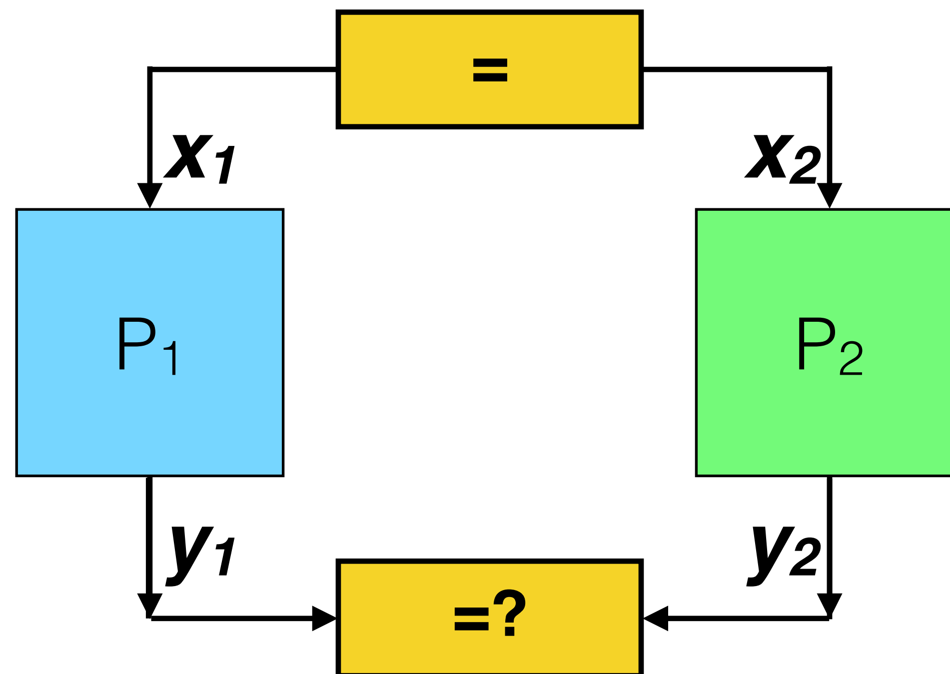
- k ($k > 1$) programs (renamed so that they have independent sets of variables)
- a *relational* specification (relating the variables) over the k programs

Prove that the relational specification holds for the programs

Example: Equivalence Checking

Given programs P_1, P_2, that respectively have inputs $\mathbf{x1}$, $\mathbf{x2}$ and outputs $\mathbf{y1}$, $\mathbf{y2}$, prove $\mathbf{x1} = \mathbf{x2} \Rightarrow \mathbf{y1} = \mathbf{y2}$.

Note: bold-faced variables are vectors



Hyperproperty Verification

- A *hyperproperty* is a relational property over *k* copies of the same program

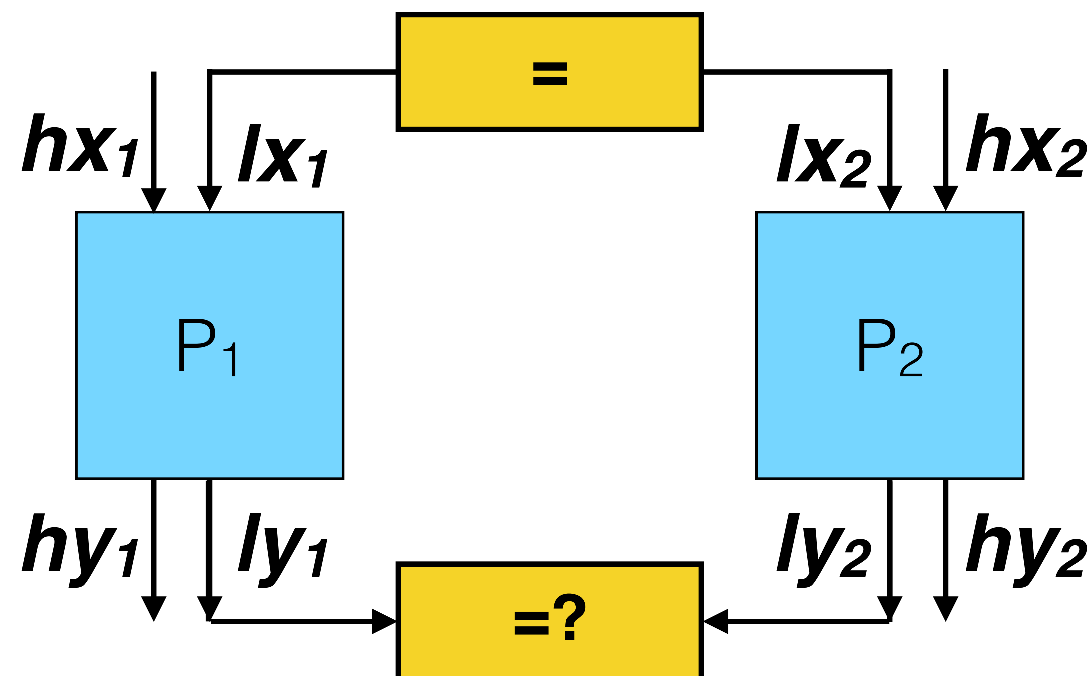
- The hyperproperty verification problem is the relational verification problem where all *k* programs are copies of the same program.

- E.g. noninterference, monotonicity, transitivity

Example: Noninterference

Security property for programs where variables have security types {low, high}

Given two *copies of the same program* P_1, P_2, that respectively have inputs (***lx1 : low, hx1 : high***), (***lx2 : low, hx2 : high***) and outputs (***ly1 : low, hy1 : high***), (***ly2 : low, hy2 : high***), prove ***lx1*** = ***lx2*** ⇒ ***ly1*** = ***ly2***.



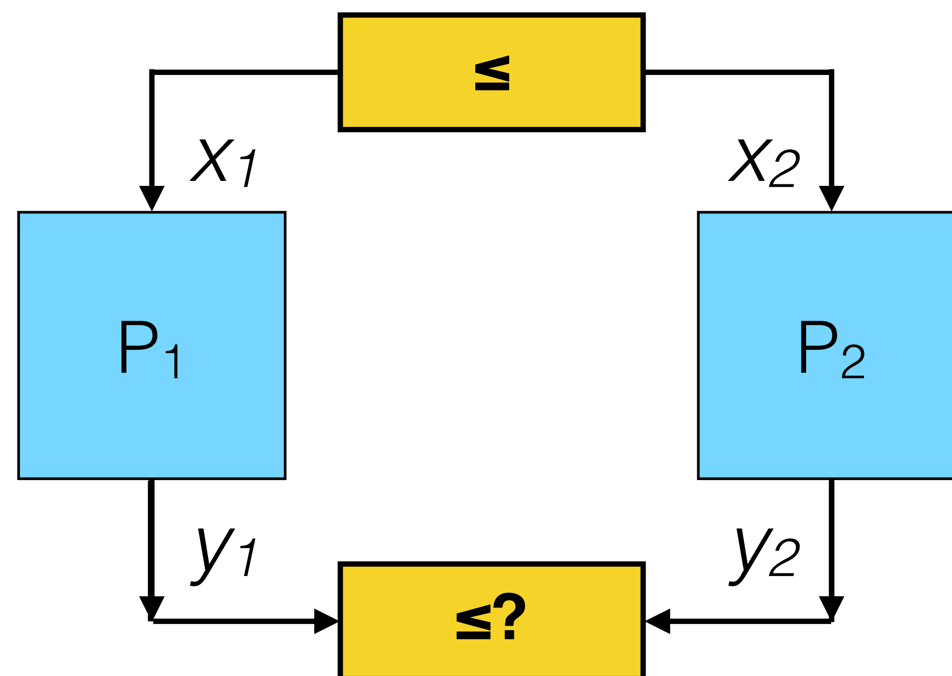
Example: Monotonicity

Given two *copies of the same program* P_1, P_2, that respectively have

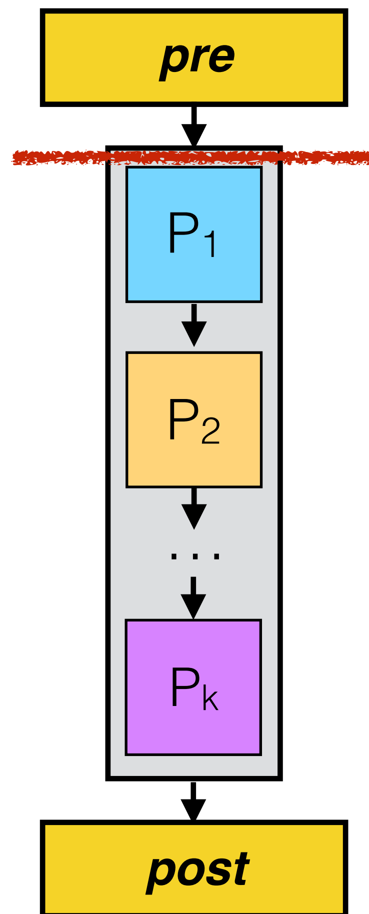
inputs $x1$, $x2$, and

outputs $y1$, $y2$, prove

$x1 \leq x2 \Rightarrow y1 \leq y2$.



Composition



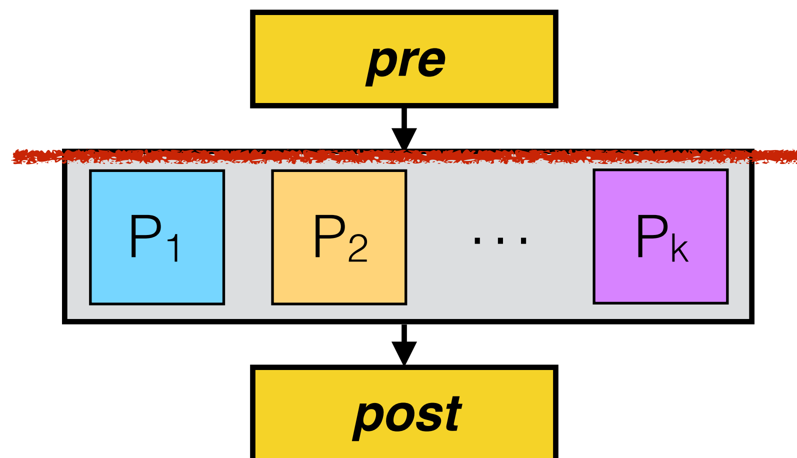
Sequential: {*pre*} P_1 ; … ; P_k {*post*}

- Pros: Can easily apply standard verification techniques

- Cons: Inflexible, can result in more difficult verification problems

[Barthe et al., 2004]
[Terauchi and Aiken, 2005]

Composition



Parallel: $\{pre\}\ P_1 \parallel \ldots \parallel P_k\ \{post\}$

- Pros: Flexibility can let us pick easier verification subproblems
- Cons: Need to come up with new techniques

[Barthe et al., 2004]
[Terauchi and Aiken, 2005]

Synchrony and Symmetry

Two new techniques: Synchrony, Symmetry

Let’s consider the challenges that motivate them….

Challenge 1: Loops

Challenge 1: Loops

$\{ x1 < x2 \;\wedge\; i1 = i2 \;\wedge\; x1 > 0 \;\wedge\; i1 > 0 \}$

L1 → `while (i1 < 10) { x1 *= i1; i1++; }`

;

L2 → `while (i2 < 10) { x2 *= i2; i2++; }`

$\{ x1 < x2 \;\wedge\; i1 = i2 \;\wedge\; x1 > 0 \;\wedge\; i1 > 0 \}$

Nonlinear Invariants:

L1: $x1 = x1_{init} \times i1! \,/\, i1_{init} \wedge \ldots$

L2: $x2 = x2_{init} \times i2! \,/\, i2_{init} \wedge \ldots$

Challenge 1: Loops

$\{ x1 < x2 \ \wedge \ i1 = i2 \ \wedge \ x1 > 0 \ \wedge \ i1 > 0 \}$

```
while (i1 < 10) { x1 *= i1; i1++; }

||

while (i2 < 10) { x2 *= i2; i2++; }
```

$\{ x1 < x2 \ \wedge \ i1 = i2 \ \wedge \ x1 > 0 \ \wedge \ i1 > 0 \}$

Consider the loops in parallel instead.

Challenge 1: Loops

$\{ x1 < x2 \ \wedge\ i1 = i2 \ \wedge\ x1 > 0 \ \wedge\ i1 > 0 \}$

```
while (i1 < 10 && i2 < 10) {

    x1 *= i1; i1++; x2 *= i2; i2++;

}
```

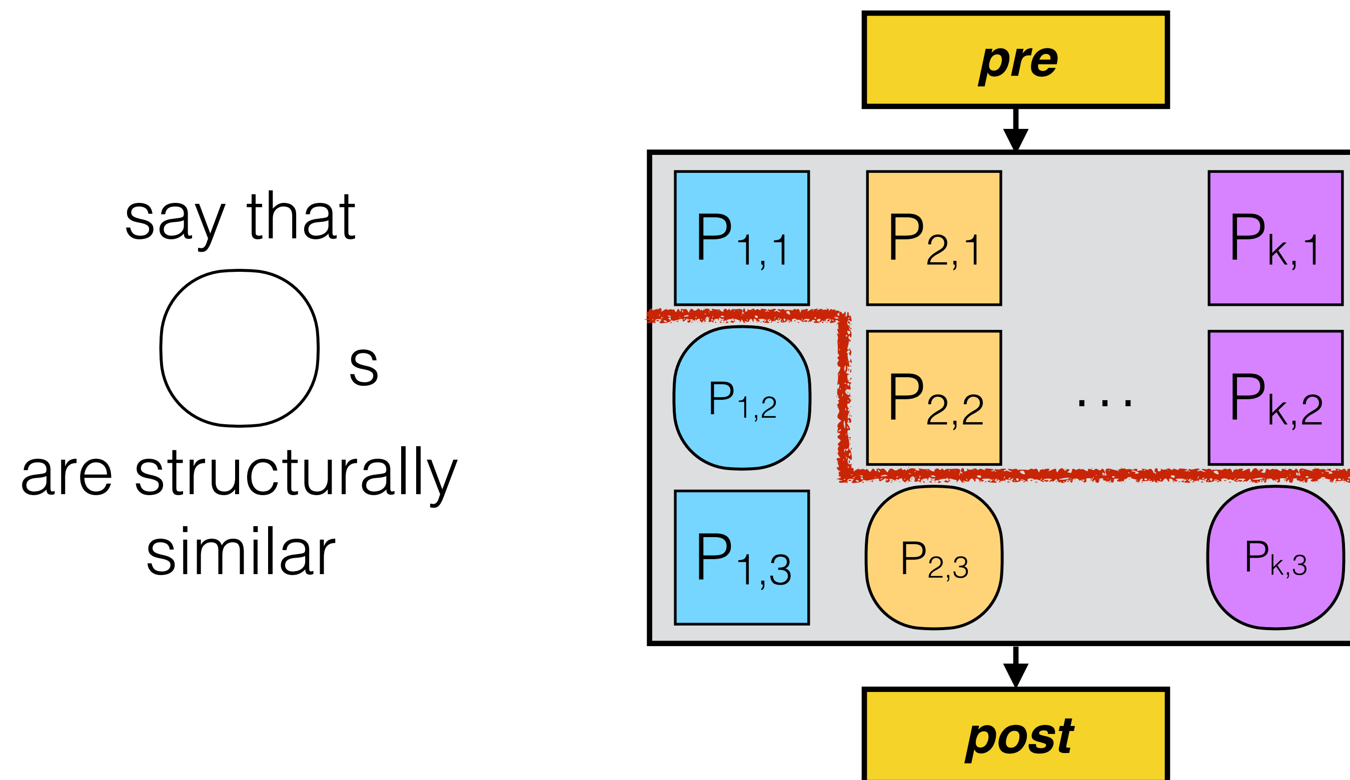
lockstep execution

$\{ x1 < x2 \ \wedge\ i1 = i2 \ \wedge\ x1 > 0 \ \wedge\ i1 > 0 \}$

(One) Relational Invariant: $x1 < x2 \wedge i1 = i2 \wedge x1 > 0 \wedge i1 > 0$

[Barthe et al., 2011]

Relational Verification



- Relating (i.e. *synchronizing*) intermediate points in programs to get *intermediate relational specifications* can result in easier verification problems

- In particular, synchronizing structurally similar parts of the different programs can yield simpler relational specifications

[Barthe et al., 2011]
[Sousa and Dillig, 2016]
[De Angelis et al., 2016]
and more

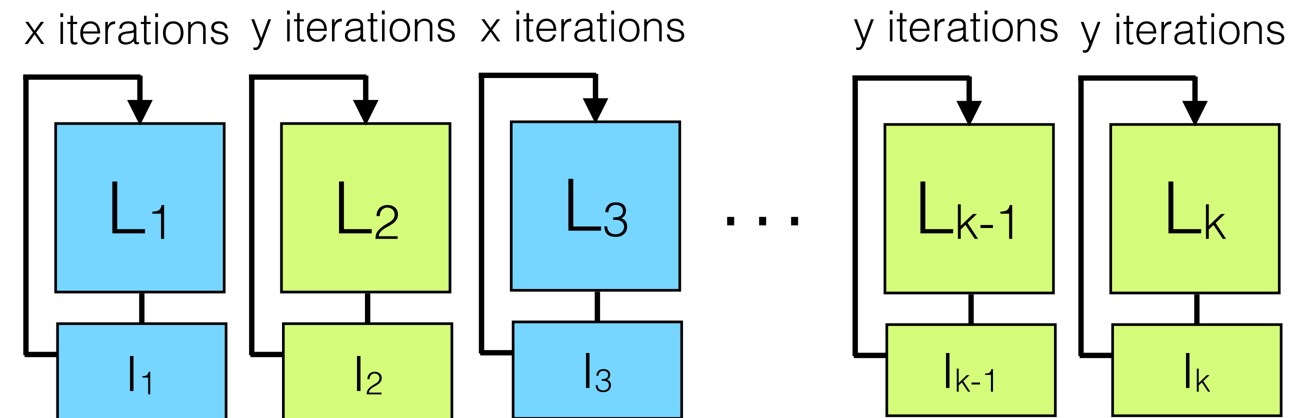
Lockstep Loops

Loops that iterate the same number of times are *able to be executed in lockstep*

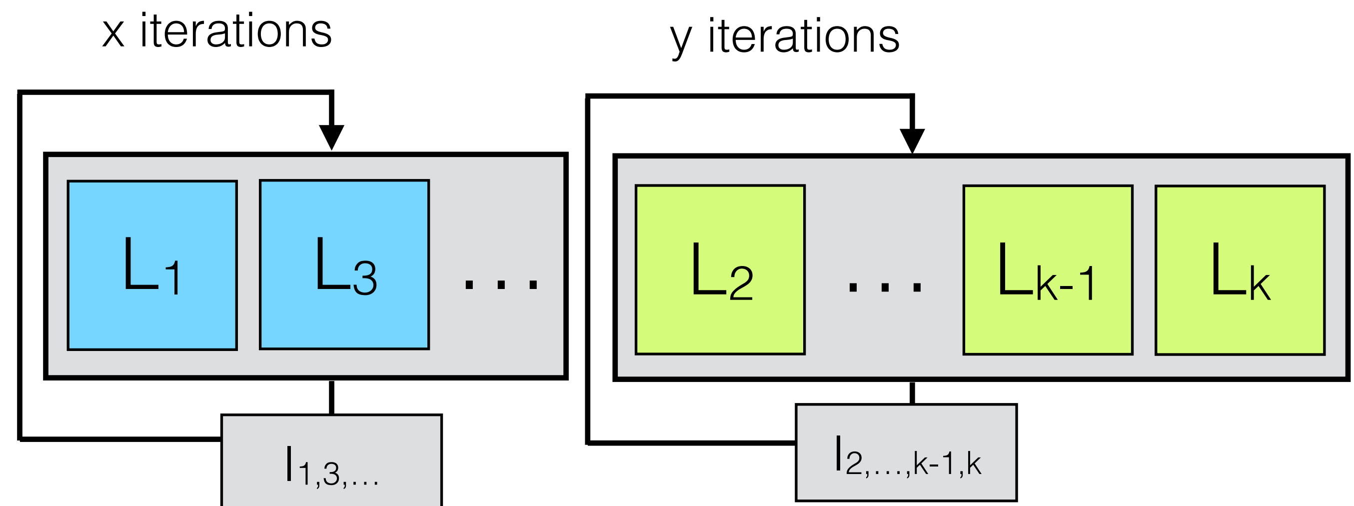
[Barthe et al., 2011]
[Sousa and Dillig, 2016]

Challenge 1: Loops

Handling each loop individually can require the generation of potentially complicated loop invariants.



How can we maximize the number of loops over which we can compute simpler relational invariants?



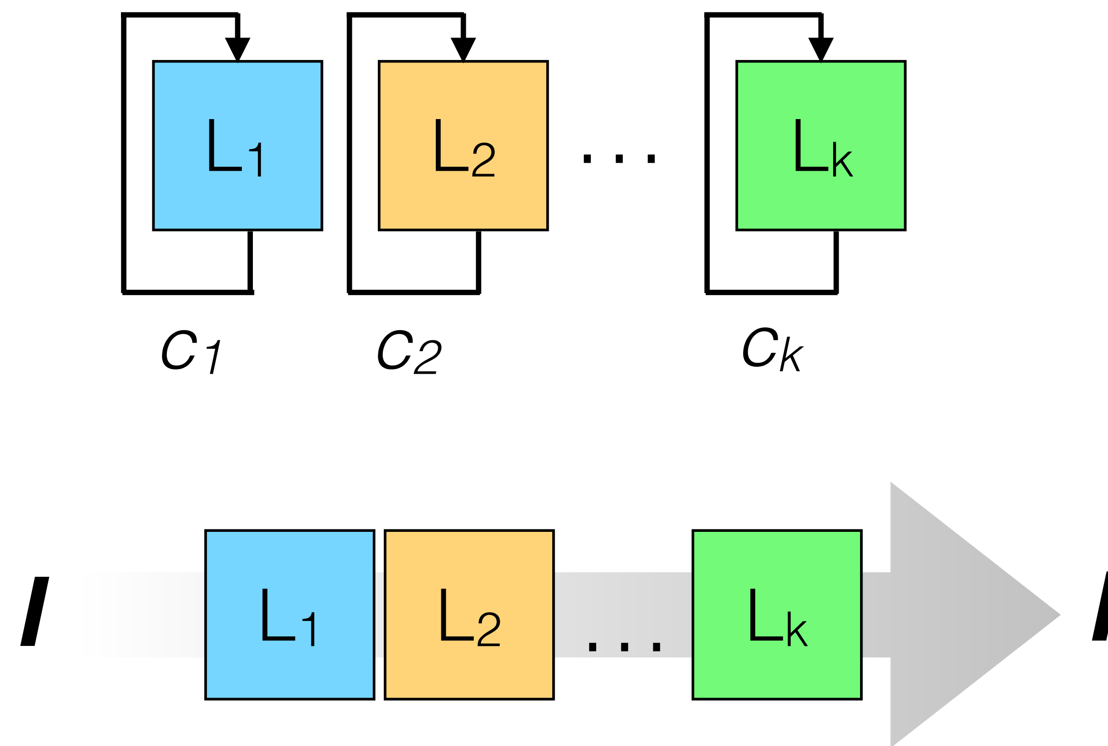
Synchrony

Partition a set of loops into *maximal* sets of loops that can be executed in lockstep

Synchrony

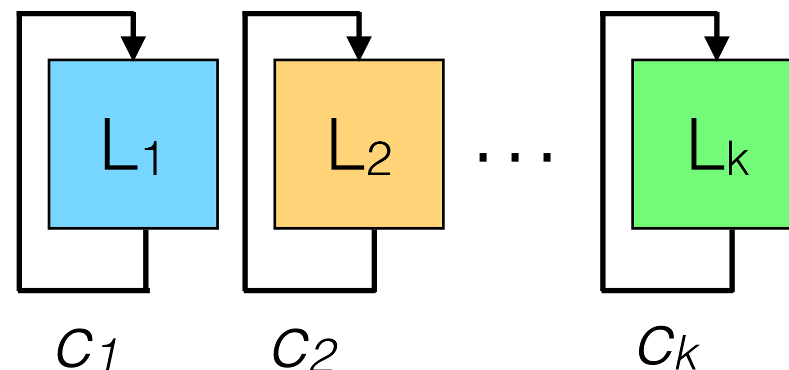
We assume we are given a relational invariant ***I***.

Note: You can use any of several existing techniques for invariant generation. The implementation (described later) uses a guess-and-check invariant generator.



Synchrony

When can we execute a set of loops in lockstep?



If any loop has terminated, all loops must have terminated.

$$\boldsymbol{I} \wedge (\neg c_1 \vee \neg c_2 \vee \ldots \vee \neg c_k) \Rightarrow (\neg c_1 \wedge \neg c_2 \wedge \ldots \wedge \neg c_k)$$

(check)

[Sousa and Dillig, 2016]

Maximal Lockstep Loop Detection

(check) $\neg(\boldsymbol{I} \wedge (\neg c_1 \vee \neg c_2 \vee \ldots \vee \neg c_k) \Rightarrow (\neg c_1 \wedge \neg c_2 \wedge \ldots \wedge \neg c_k))$

- If unsatisfiable, all loops can be executed in lockstep. (Done!)

- If satisfiable, then what?

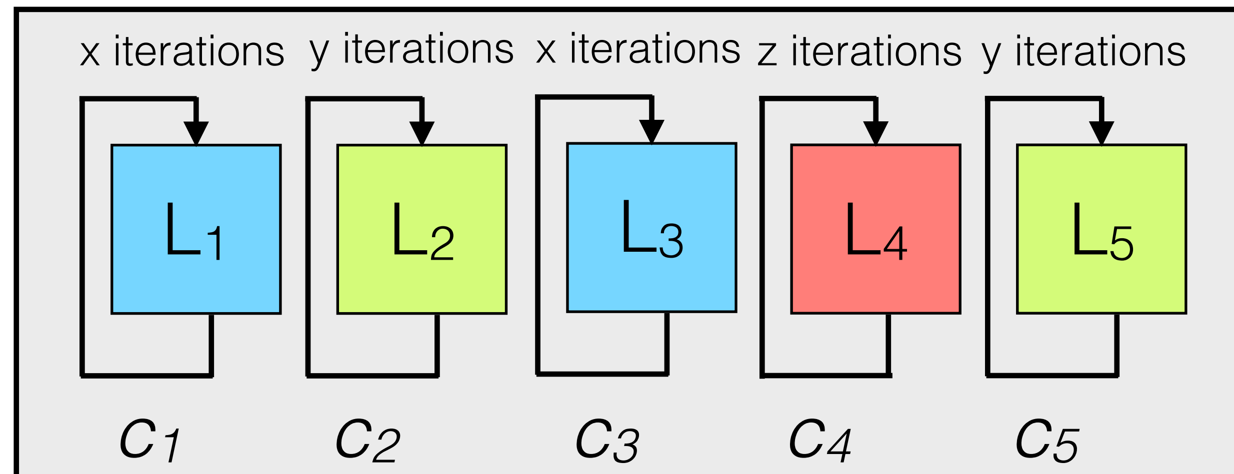
(partition)

- Use model to partition the set of loops into those that have terminated ($\neg c_i$ holds in the model) and those that have not (c_i holds in the model)

(recurse)

- Recurse on the two sets….

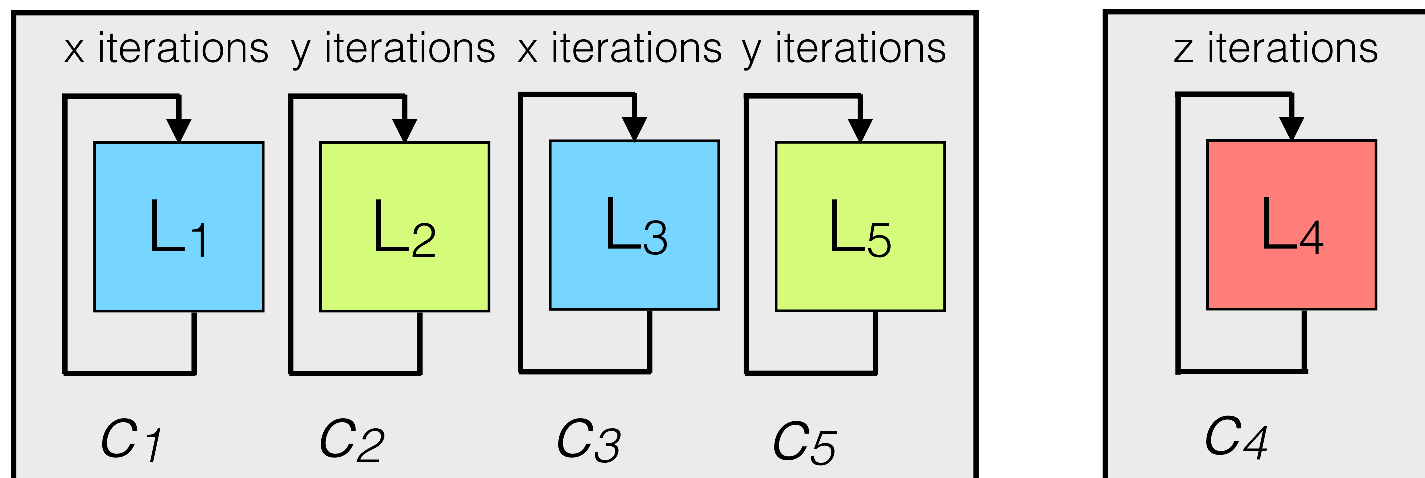
Maximal Lockstep Loop Example



$\neg(\boldsymbol{I} \wedge (\neg c_1 \vee \neg c_2 \vee \ldots \vee \neg c_5) \Rightarrow (\neg c_1 \wedge \neg c_2 \wedge \ldots \wedge \neg c_5))$

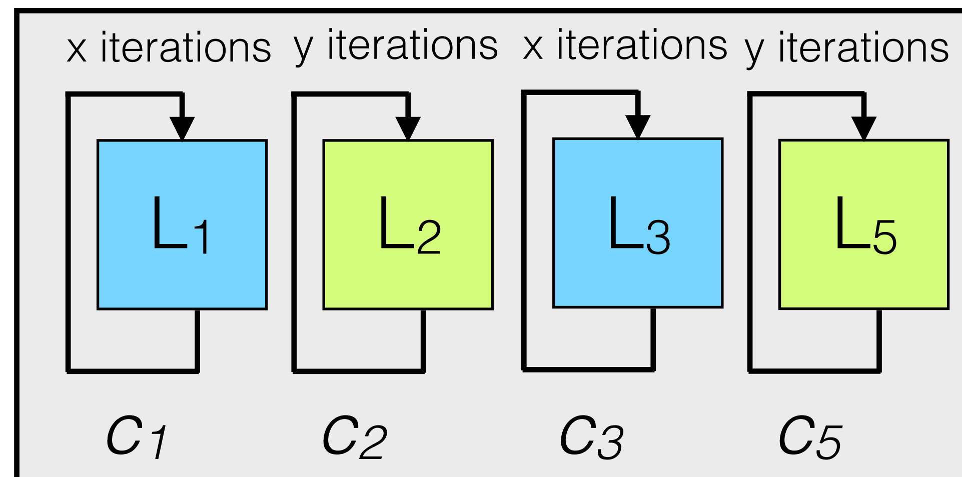
(check) SAT: c_1, c_2, c_3, $\neg c_4$, and c_5 hold in model

(partition)



Maximal Lockstep Loop Example

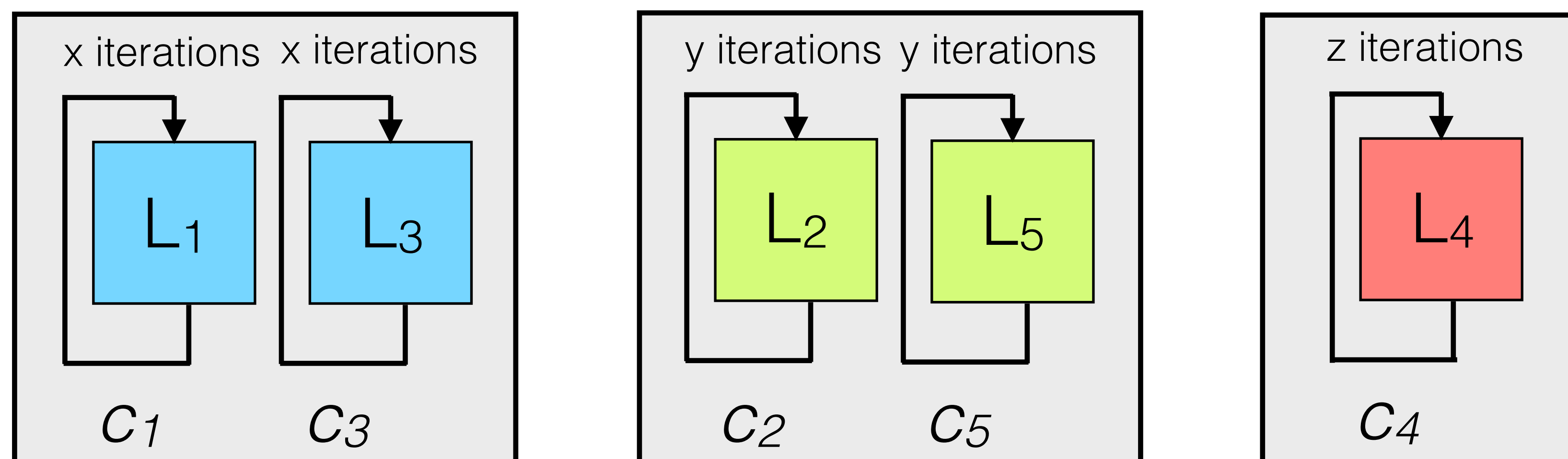
(recurse)



$$\neg(\boldsymbol{I} \wedge (\neg c_1 \vee \neg c_2 \vee \neg c_3 \vee \neg c_5) \Rightarrow (\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \neg c_5))$$

(check) SAT: c_1, $\neg c_2$, c_3, $\neg c_5$ hold in model

(partition)



Done!

Summary: Maximal Lockstep Loop Detection

Step 1. Check if current set can be executed in lockstep

Step 2. Partition according to model (if necessary)

Step 3. Recurse

Challenge 2: Redundancy

Challenge 2: Redundancy

$\{ x1 \neq x2 \}$

if (x1 > y1) then P1 else Q1

||

if (x2 > y2) then P2 else Q2

$\{ x1 \neq x2 \}$

Challenge 2: Redundancy

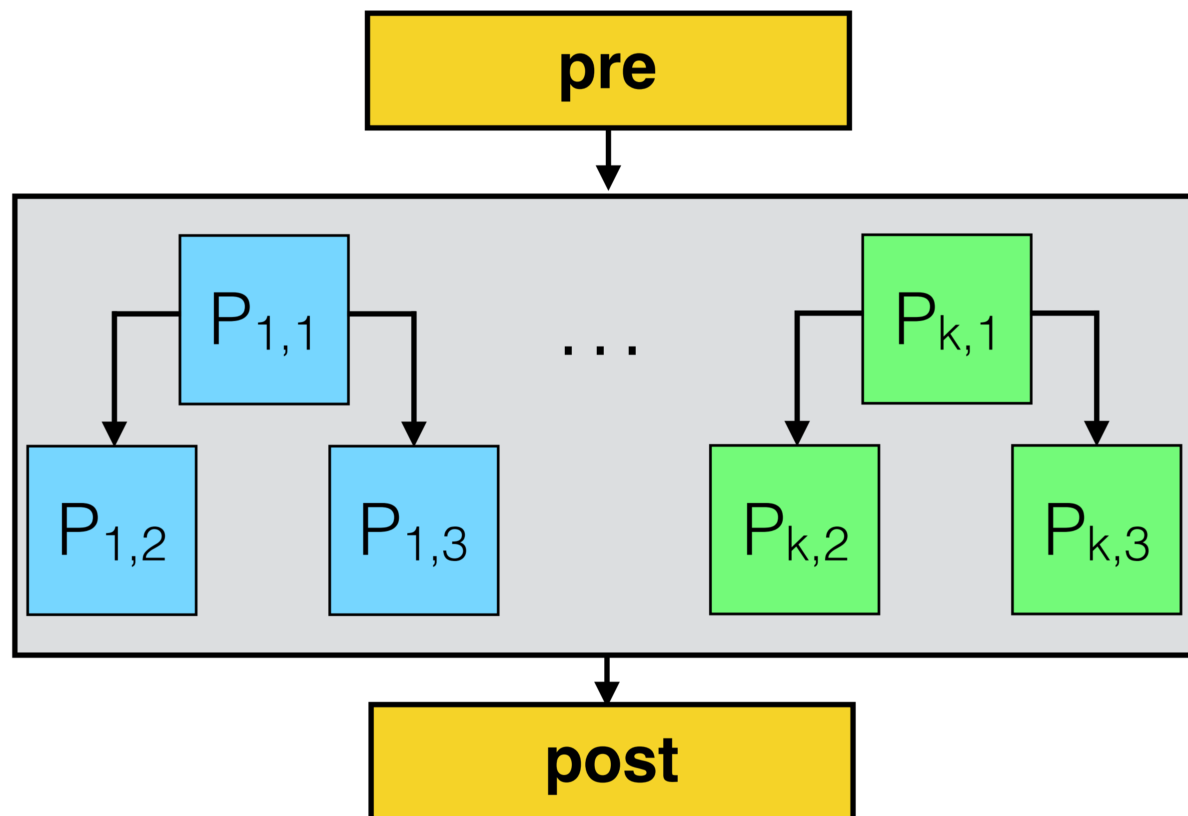
$\{ x1 \neq x2 \}$

if (x1 > y1) then P1 else Q1 || if (x2 > y2) then P2 else Q2

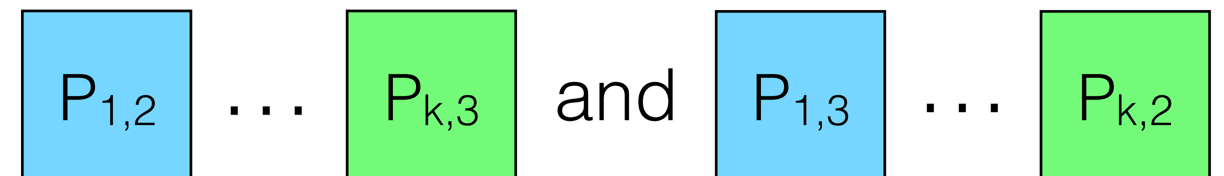
$\{ x1 \neq x2 \}$

RVP1	RVP2	RVP3	RVP4
$\{ x1 \neq x2 \wedge x1 > y1 \wedge x2 > y2 \}$	$\{ x1 \neq x2 \wedge x1 \leq y1 \wedge x2 > y2 \}$	$\{ x1 \neq x2 \wedge x1 > y1 \wedge x2 \leq y2 \}$	$\{ x1 \neq x2 \wedge x1 \leq y1 \wedge x2 \leq y2 \}$
P1 || P2	Q1 || P2	P1 || Q2	Q1 || Q2
$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$

Challenge 2: Redundancy



Maybe for the given relational specification,

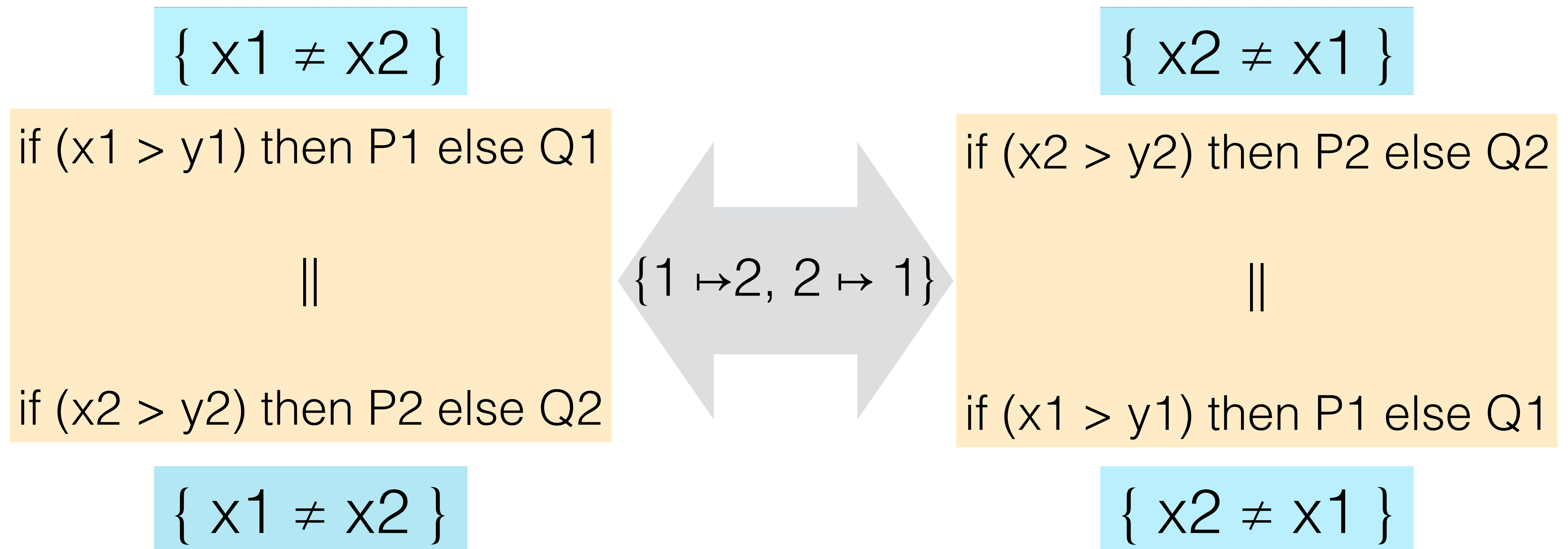


are symmetric over indices.

How can we identify and use symmetries
in programs and
in relational specifications
to avoid solving redundant verification problems?

Symmetric Relational Verification Problems (RVPs)

If you permute indices, you get the same problem.



Need a permutation π of indices that is a symmetry of the formulas (pre- and postconditions) and of the programs (can e.g. check if at same program point for hyperproperties)

Leveraging Symmetry to Reduce Redundancies

- Find symmetries in formulas (permutation π)
- Find symmetric RVPs (make sure programs are symmetric, i.e. π is a symmetry of the programs also)
- Prune (via symmetry-breaking, lifted from SAT)

Leveraging Symmetry to Reduce Redundancies

- **Find symmetries in formulas** (permutation π)
- Find symmetric RVPs (make sure programs are symmetric, i.e. π is a symmetry of the programs also)
- Prune (via symmetry-breaking, lifted from SAT)

Finding Symmetries of a Formula

- Prior work for SAT formulas: based on finding automorphisms of a colored graph
- Our work: Lift SAT techniques to first-order theories (with equality, linear integer arithmetic)

[Aloul et al., 2006]
[Crawford et al., 2005]

Example: Finding Symmetries of a Formula

Step 1. Canonicalize

$$\phi = x_1 \leq x_2 \land x_3 \leq x_4$$

$$\phi' = ((x_1 < x_2) \lor (x_1 = x_2)) \land ((x_3 < x_4) \lor (x_3 = x_4))$$

[Aloul et al., 2006]
[Crawford et al., 2005]

Example: Finding Symmetries of a Formula

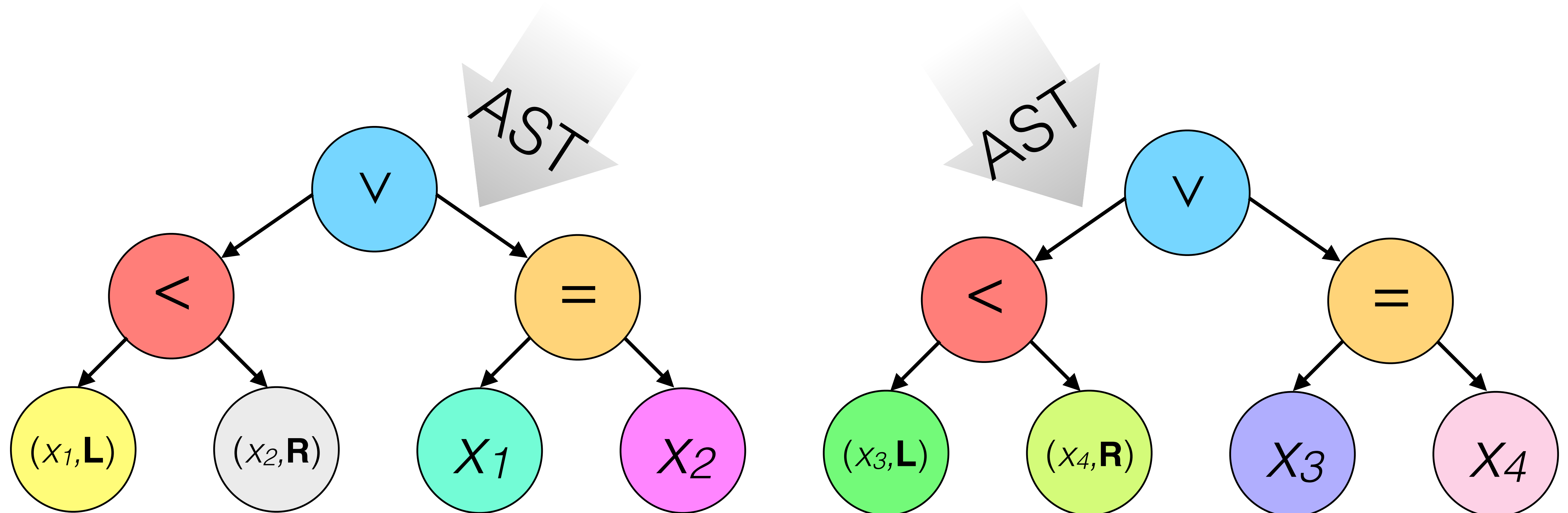
Step 2. Create colored graph from AST

$$\phi' = ((x_1 < x_2) \vee (x_1 = x_2)) \wedge ((x_3 < x_4) \vee (x_3 = x_4))$$

Example: Finding Symmetries of a Formula

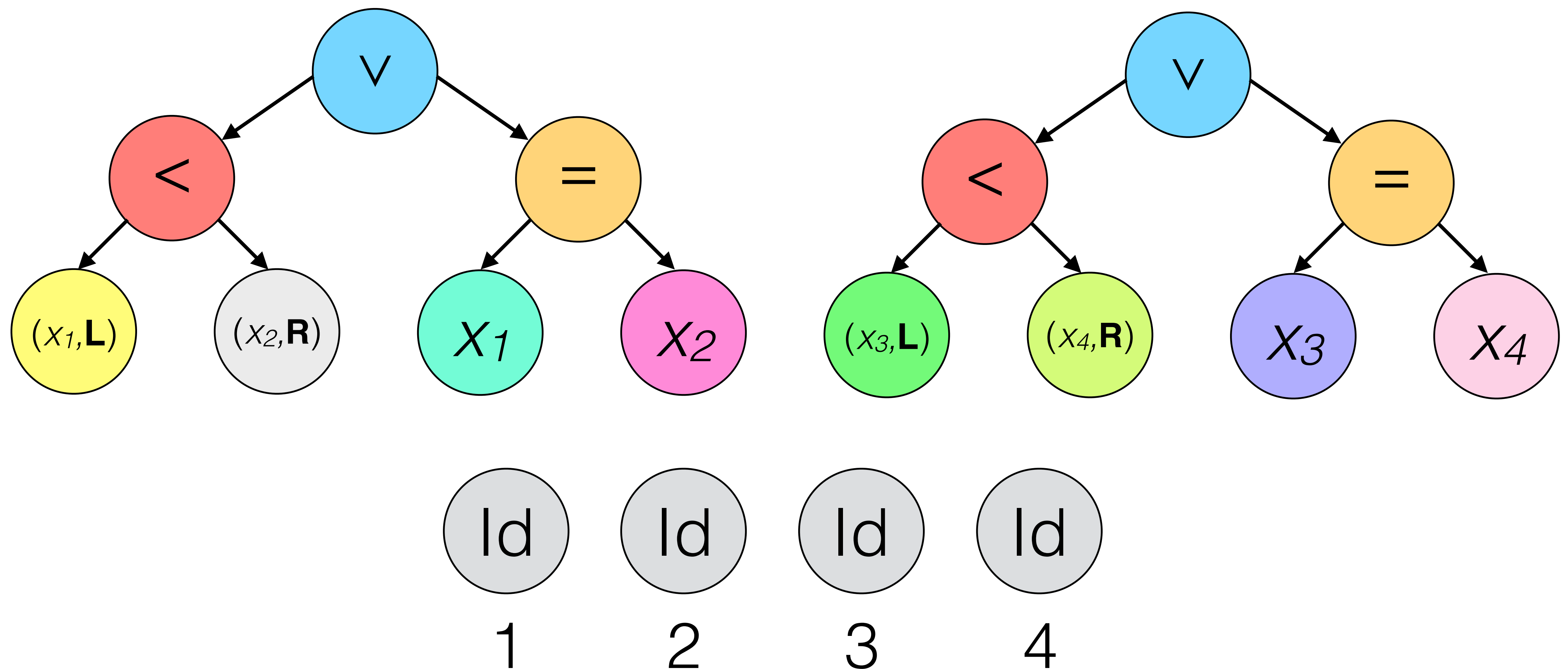
Step 2. Create colored graph from ASTs

Clauses: $\{(x_1 < x_2) \vee (x_1 = x_2), (x_3 < x_4) \vee (x_3 = x_4)\}$



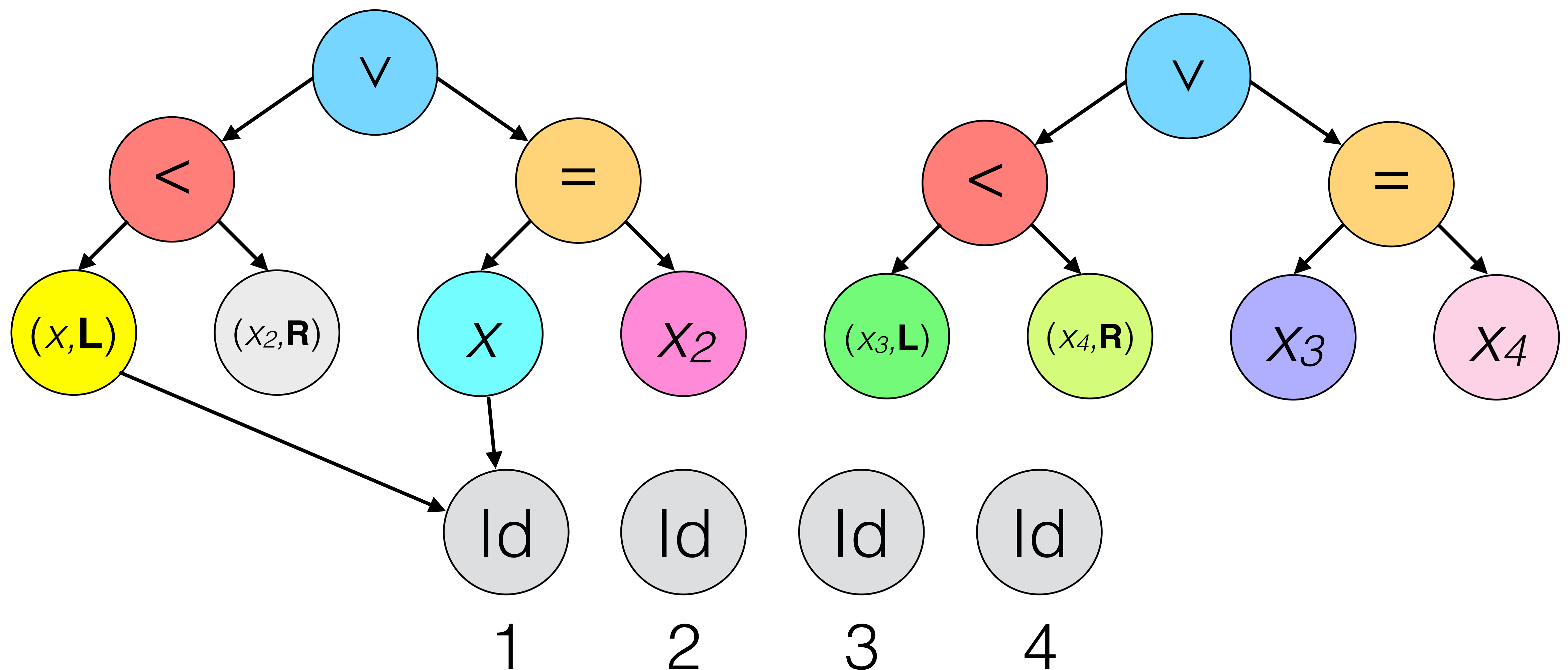
Example: Finding Symmetries of a Formula

Step 2. Create graph from ASTs



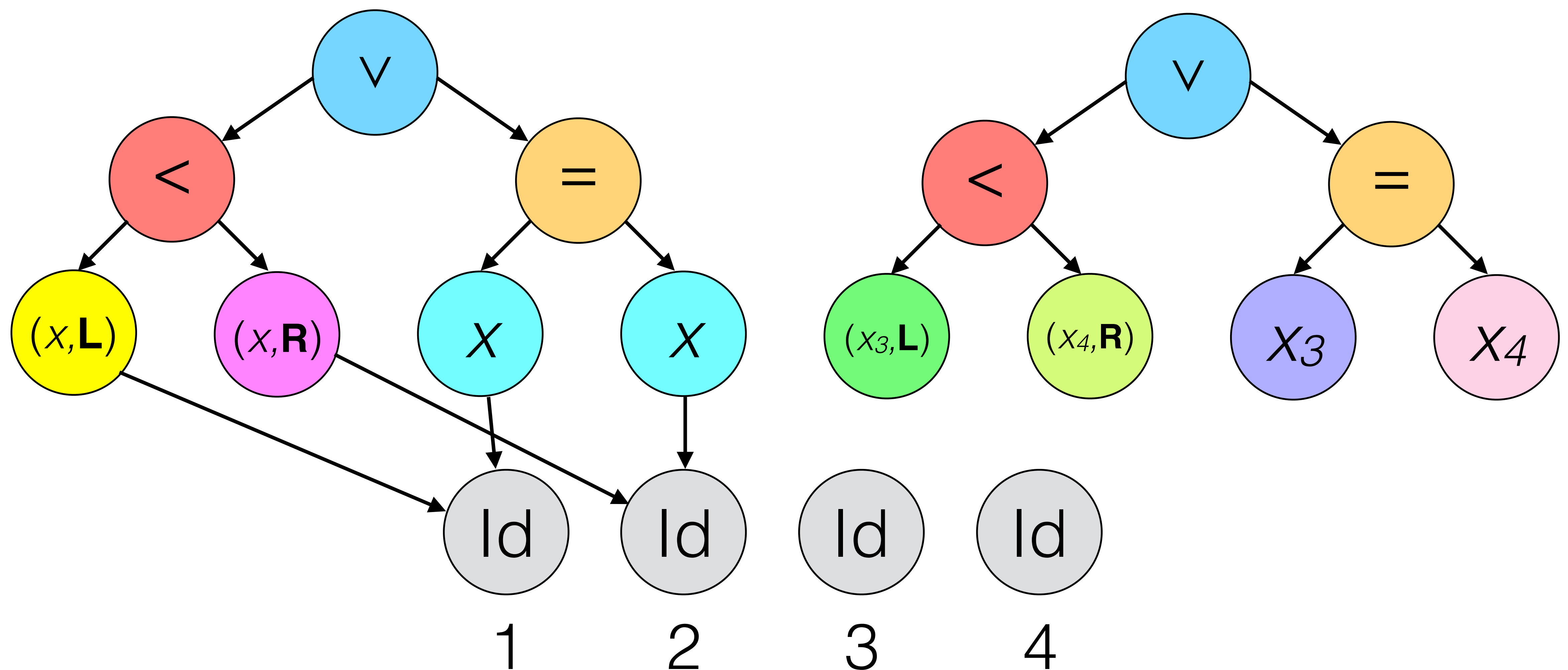
Example: Finding Symmetries of a Formula

Step 2. Create graph from ASTs



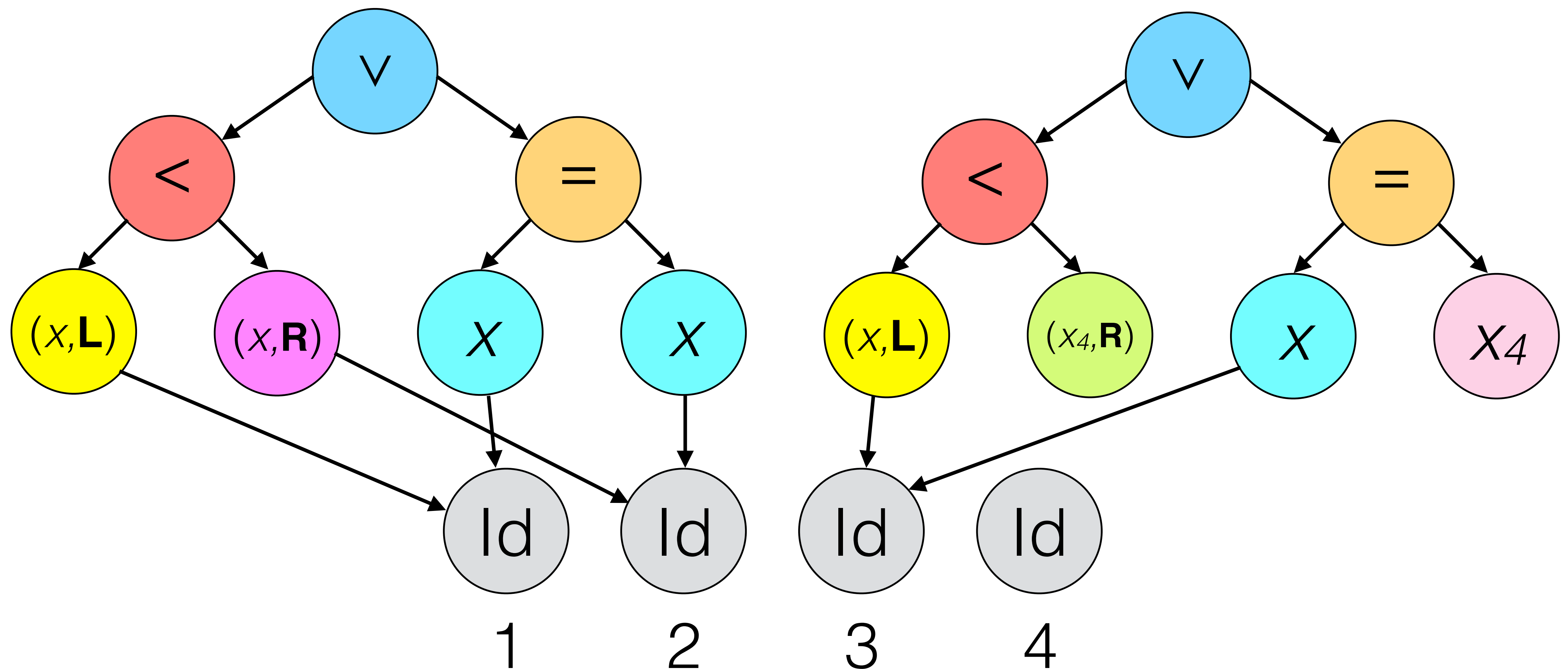
Example: Finding Symmetries of a Formula

Step 2. Create graph from ASTs



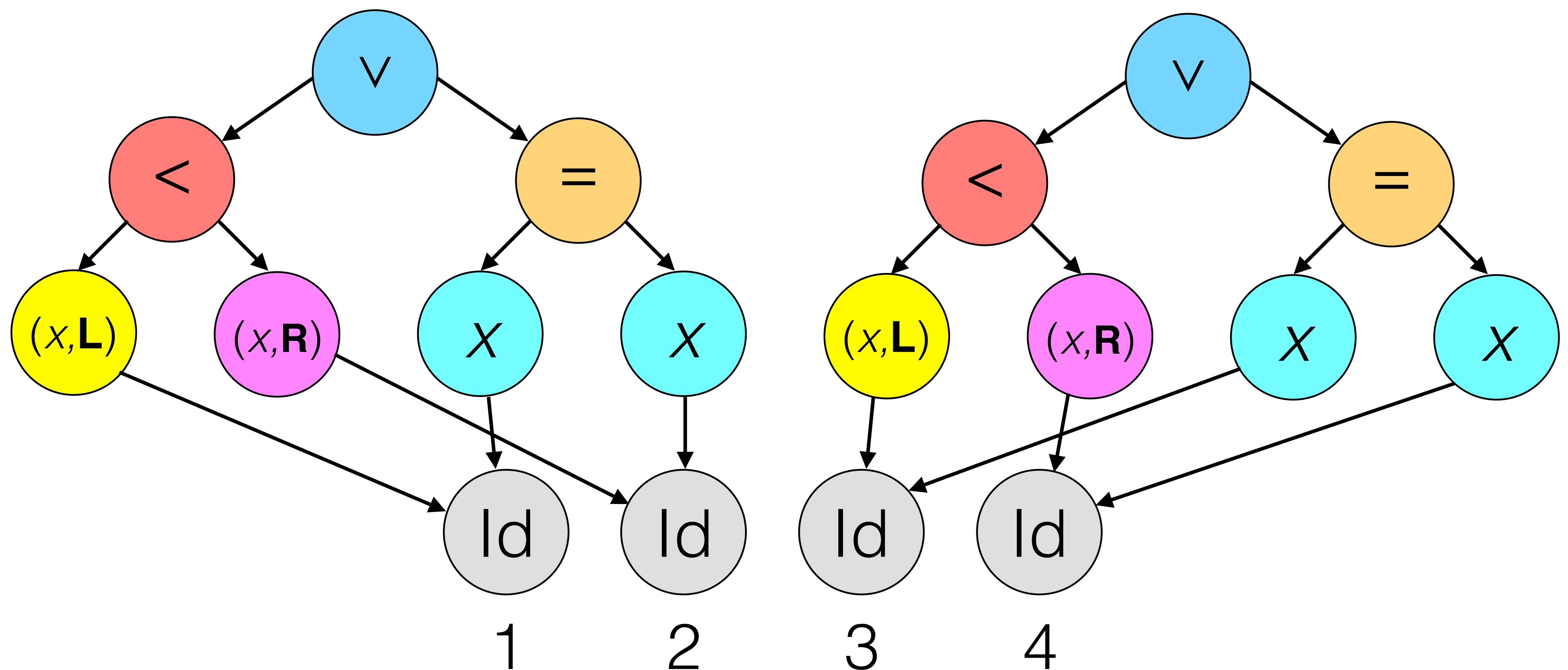
Example: Finding Symmetries of a Formula

Step 2. Create graph from ASTs



Example: Finding Symmetries of a Formula

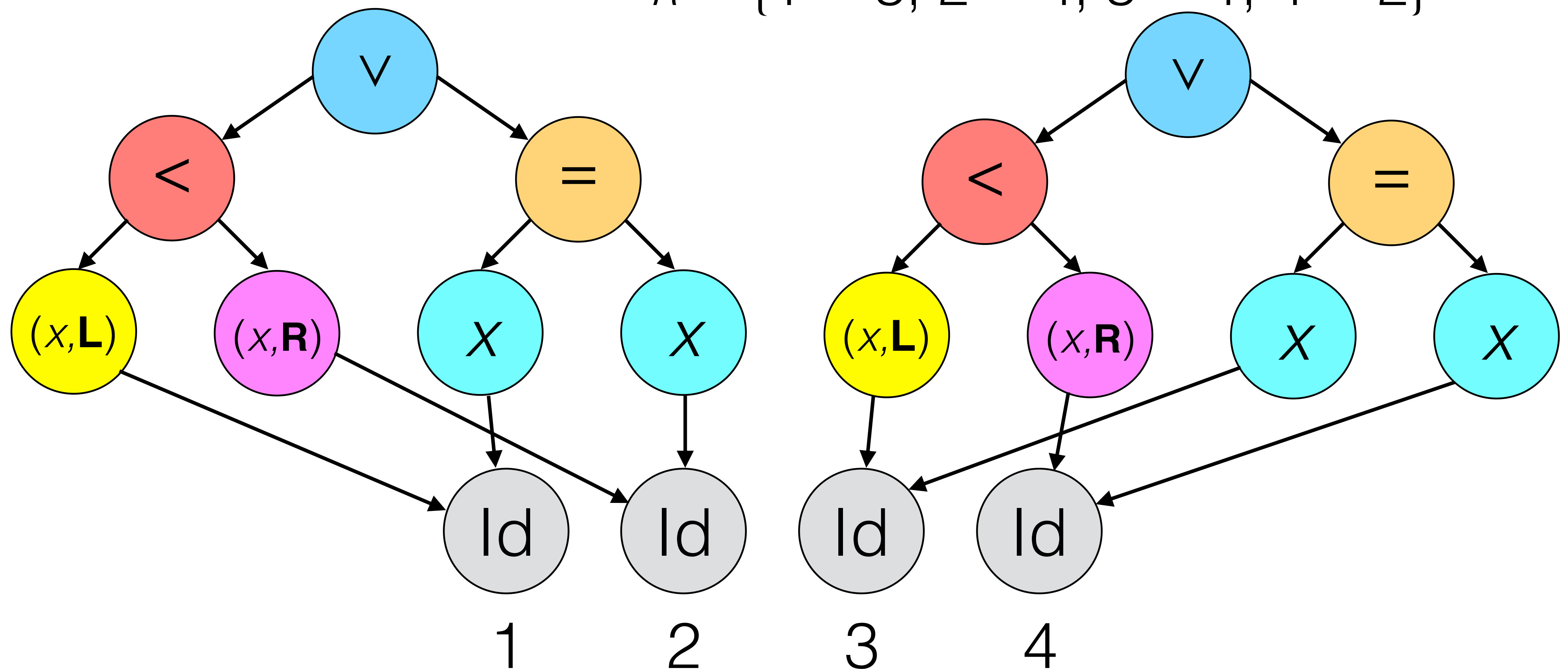
Step 2. Create graph from ASTs



Example: Finding Symmetries of a Formula

Step 3. Find graph automorphisms

$\pi = \{1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2\}$



Example: Finding Symmetries of a Formula

$$\pi = \{1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2\}$$

$$\phi = x_1 \leq x_2 \wedge x_3 \leq x_4$$

$$\pi(\phi) = x_3 \leq x_4 \wedge x_1 \leq x_2$$

$$\phi \Leftrightarrow \pi(\phi)$$

Summary: Finding Symmetries of a Formula Automatically

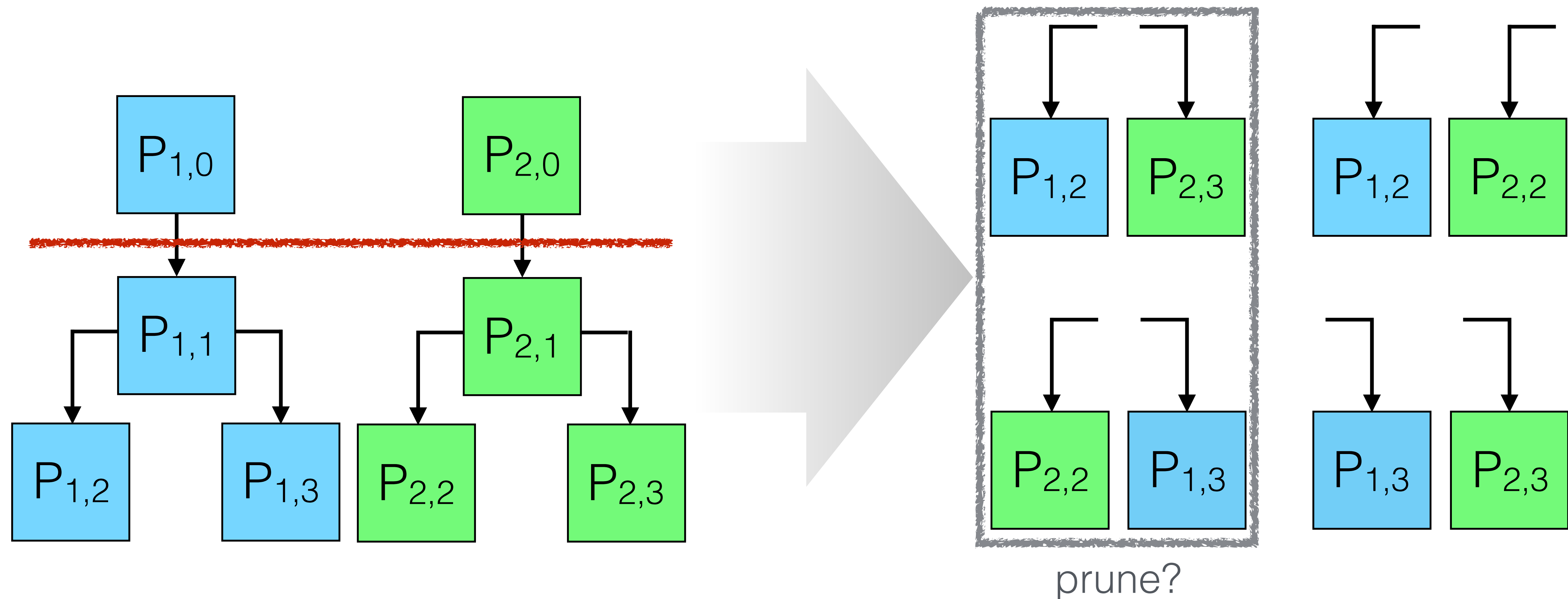
Step 1. Canonicalize

Step 2. Create graph from AST

Step 3. Find graph automorphisms

How to apply symmetry?

- Aligning conditionals
- So far we have considered trying to align loops in a particular way
- We also would like to align conditional statements in order to give us more opportunities to exploit symmetry



Aligning Conditionals: Example

{ post(post(x1 ≠ x2, R1), S2) }

R if (x1 > y1) then P1 else Q1 1

||

S if (x2 > y2) then P2 else Q2 2

{ x1 ≠ x2 }

Instantiation: Hyperproperty Verification

Instantiation: Hyperproperty Verification

Algorithm based on forward analysis where we maintain Hoare triples.

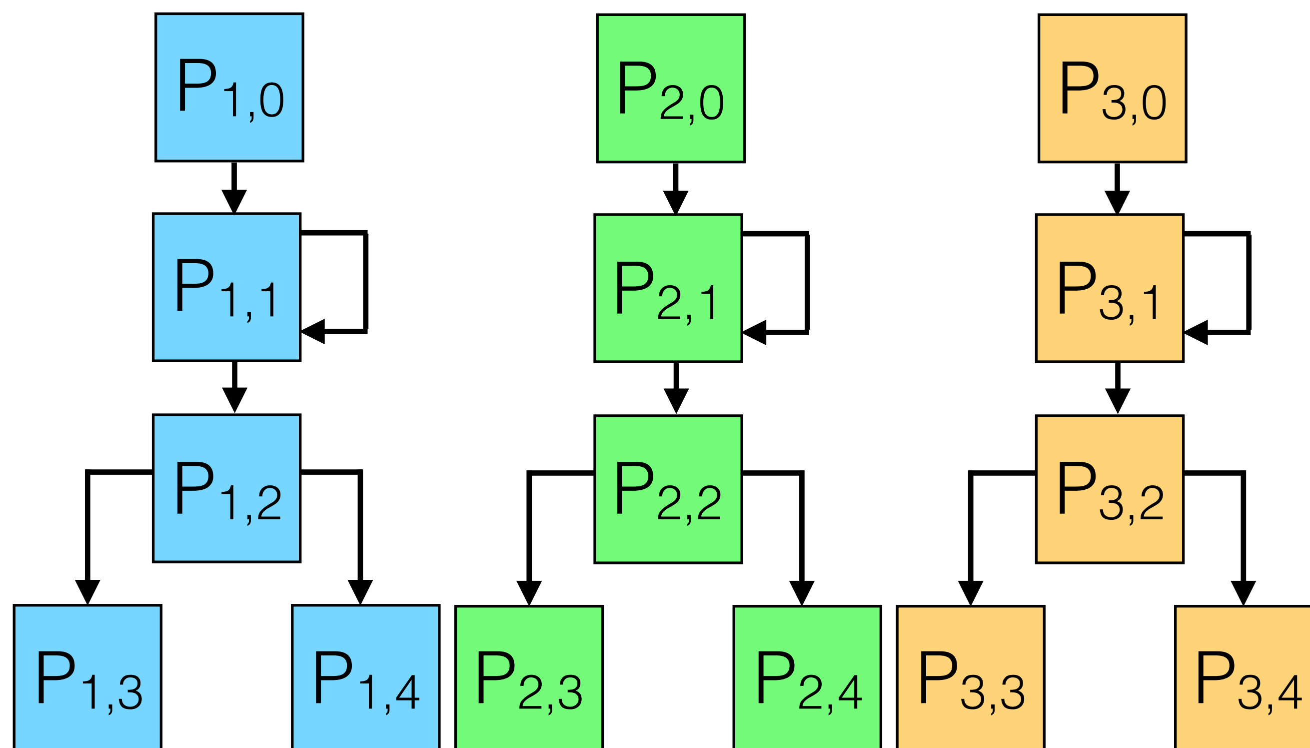
```
1: procedure VERIFY(pre, Current, Ifs, Loops, post)
2:     while Current ≠ ∅ do
3:         if PROCESSSTATEMENT(pre, P_i, Ifs, Loops, post) = safe then return safe
4:     if Loops ≠ ∅ then HANDLELOOPS(pre, Loops, post)
5:     else if Ifs ≠ ∅ then HANDLEIFS(pre, Ifs, Loops, post)
6:     else return unsafe
```

Maintain sets of program copies that begin with conditionals (*Ifs*) and loops (*Loops*).

Instantiation: Hyperproperty Verification

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2: **while** $Current \neq \varnothing$ **do**
3: **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) = safe **then return** safe
4: **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5: **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6: **else return** unsafe

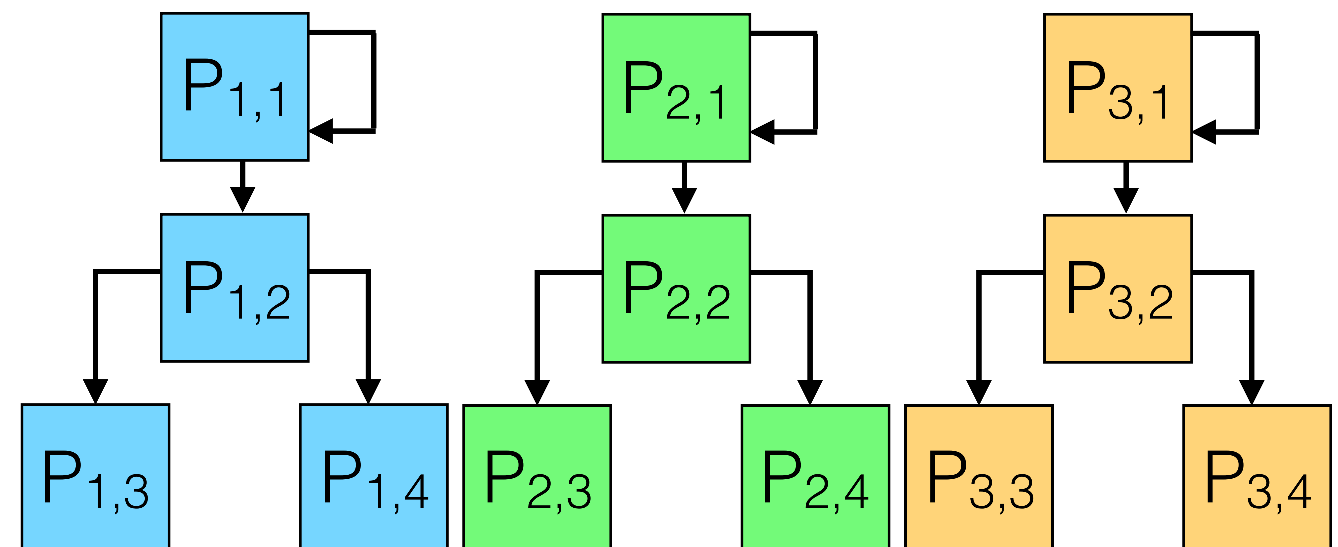
Current *Ifs* *Loops*



Instantiation: Hyperproperty Verification

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2: **while** $Current \neq \varnothing$ **do**
3: **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) $=$ safe **then return** safe
4: **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5: **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6: **else return** unsafe

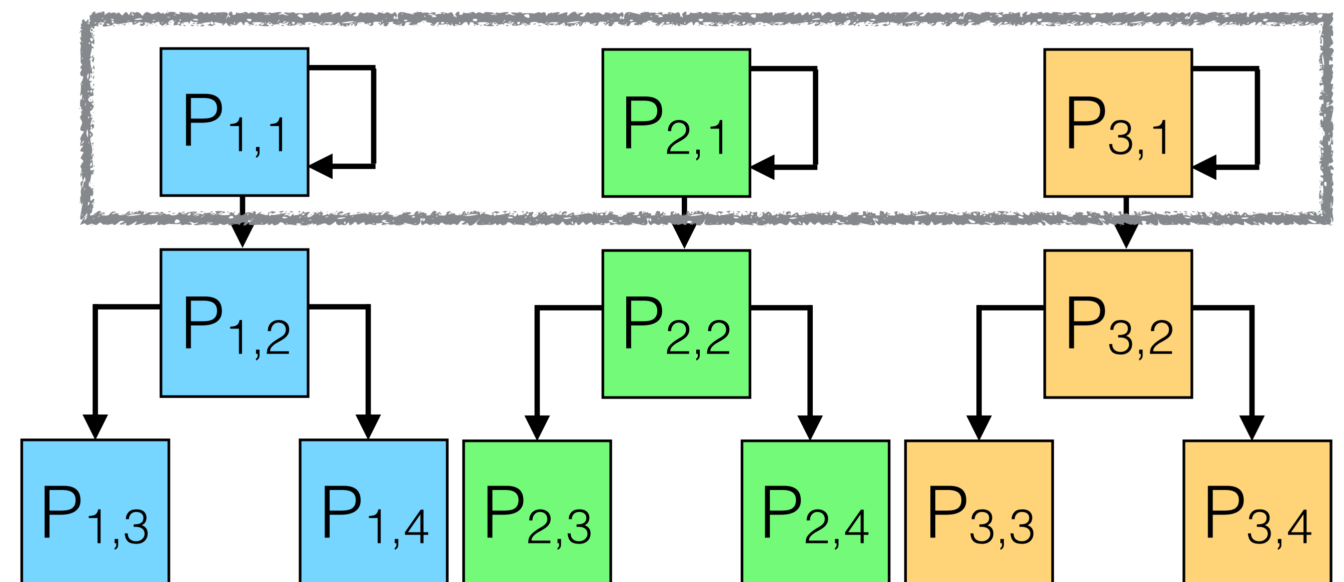
Current *Ifs* *Loops*



Instantiation: Hyperproperty Verification

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2: **while** $Current \neq \varnothing$ **do**
3: **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) = safe **then return** safe
4: **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5: **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6: **else return** unsafe

Current *Ifs* *Loops*

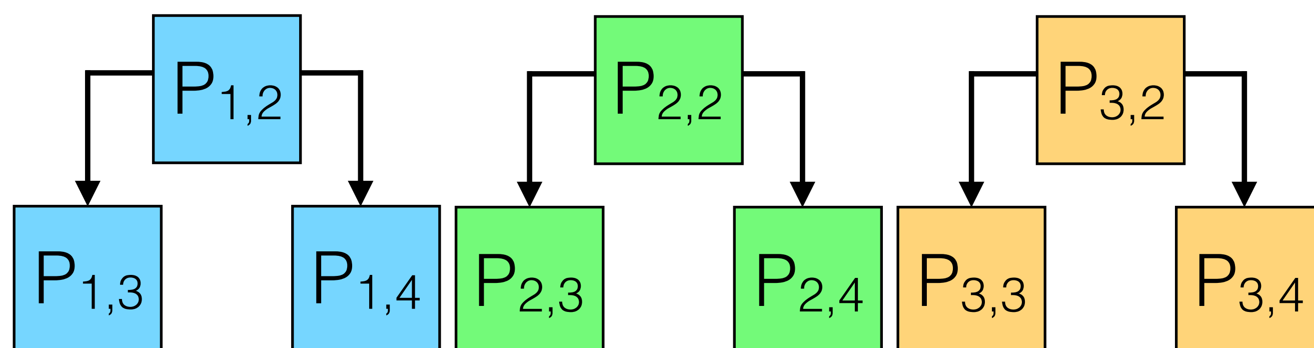


handle maximally in lockstep

Instantiation: Hyperproperty Verification

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2: **while** $Current \neq \varnothing$ **do**
3: **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) = safe **then return** safe
4: **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5: **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6: **else return** unsafe

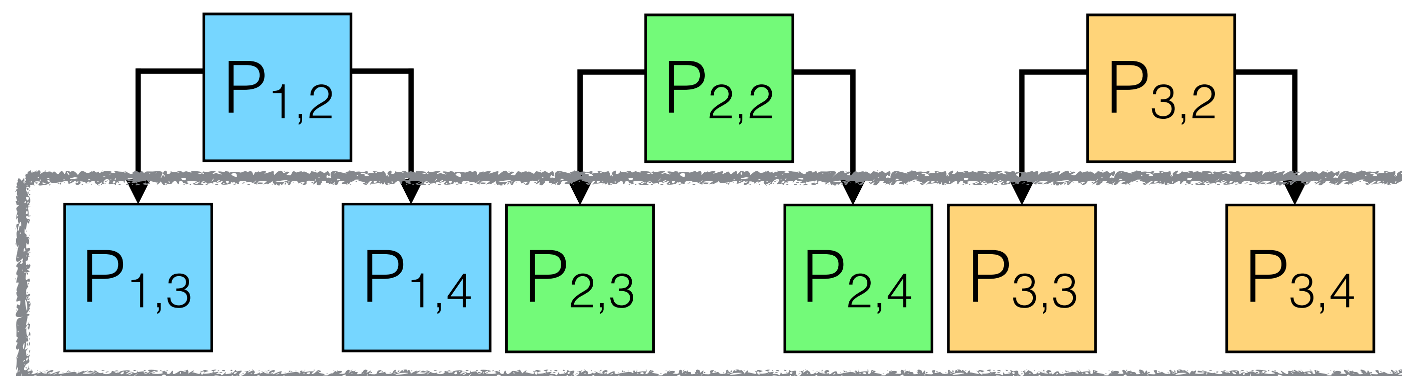
Current *Ifs* *Loops*



Instantiation: Hyperproperty Verification

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2: **while** $Current \neq \varnothing$ **do**
3: **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) = safe **then return** safe
4: **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5: **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6: **else return** unsafe

Current *Ifs* *Loops*



Evaluation

Prototype

- Built on top of *Descartes* **[Sousa and Dillig, 2016]**
- Two variants:
  - Syn - uses synchrony
  - Synonym - uses synchrony and symmetry

Benchmarks

- 33 small Stackoverflow Java benchmarks (21-107 LOC) from original *Descartes* evaluation **[Sousa and Dillig, 2016]**
- 16 larger, modified Stackoverflow Java benchmarks (62-301 LOC)

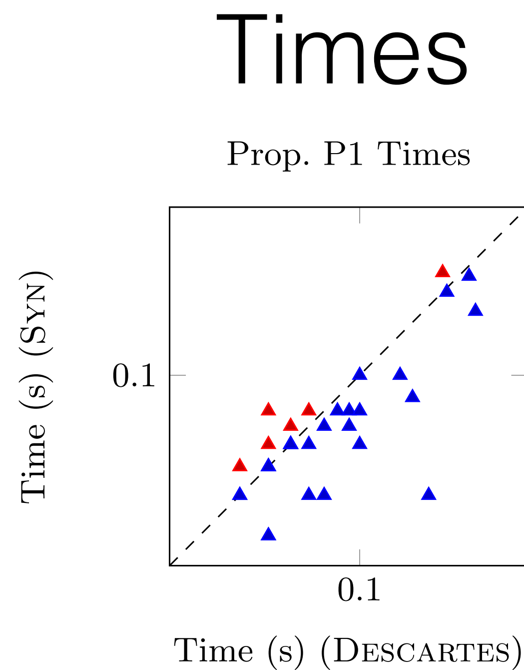
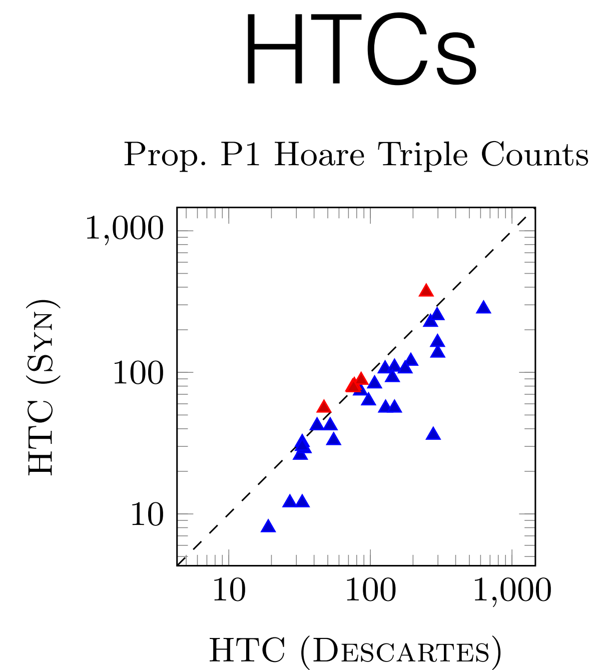
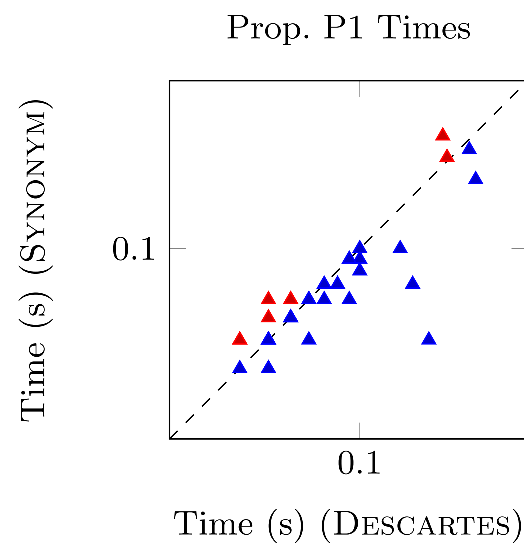
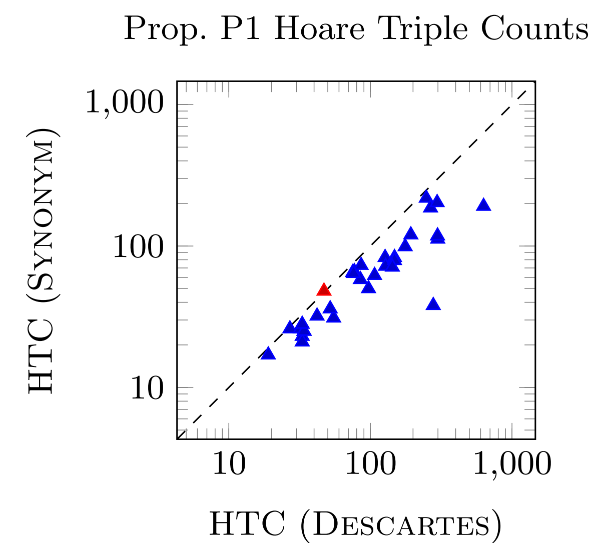
All experiments conducted on a MacBook Pro with a 2.7GHz Intel Core i5 processor and 8GB RAM.

Example Benchmark

```
public class Match implements Comparator<Match>{
    int score;
    int seq1start;
    int seq2start;

    @Override
    public int compare(Match o1, Match o2) {
      // first compare scores
      if (o1.score > o2.score) return -1; /* higher score for o1 -> o1 */
      if (o1.score < o2.score) return 1; /* higher score for o2 -> o2 */

      // scores are equal, go on with the position
      if ((o1.seq1start + o1.seq2start) < (o2.seq1start+o2.seq2start)
          return -1; /* o1 farther left */
      if ((o1.seq1start + o1.seq2start) > (o2.seq1start+o2.seq2start))
          return 1; /* o2 farther left */

      // they're equally good
      return 0;
    }
}
```

Results: Small Stackoverflow Benchmarks

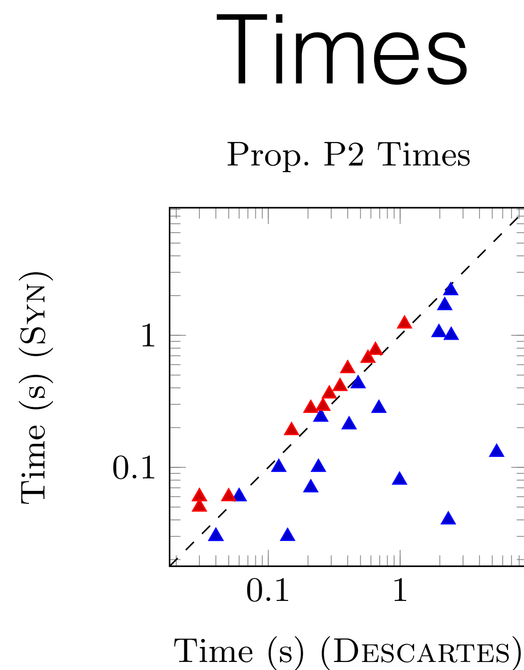
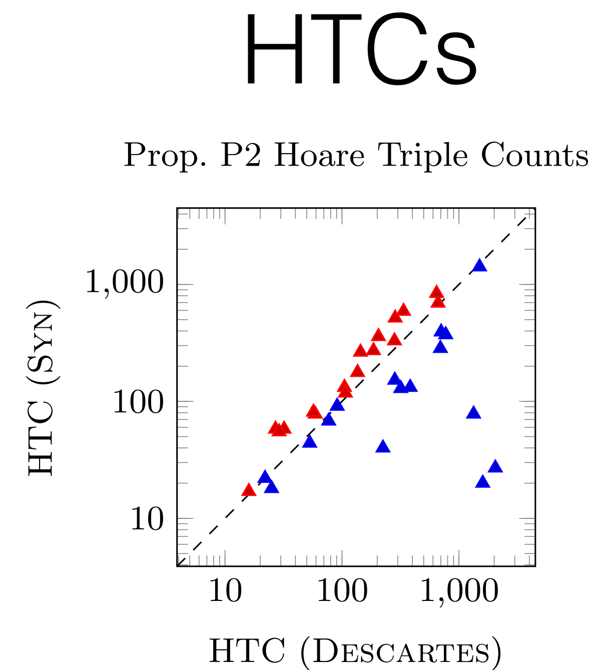
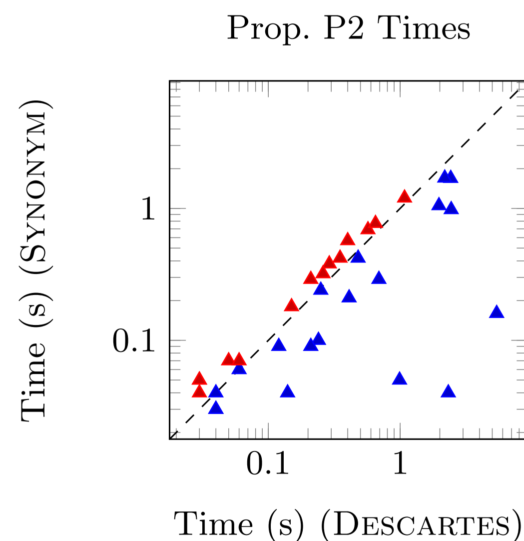
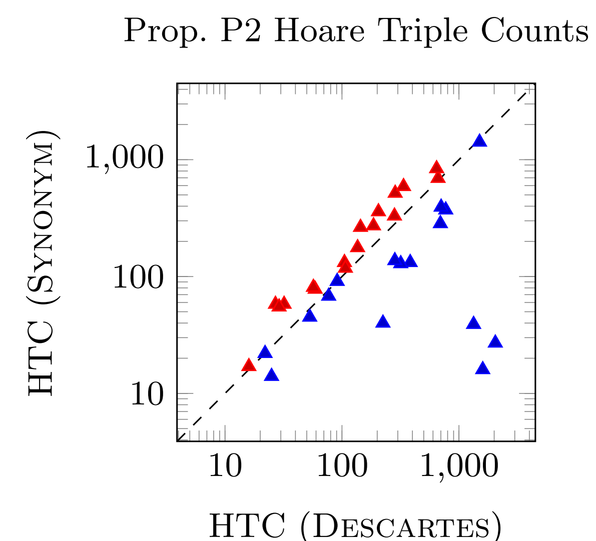
P1: ∀*x*,*y*. *sgn*(*compare*(*x*,*y*)) = -*sgn*(*compare*(*y*,*x*))

	Times	HTCs
Syn vs. Descartes		
Synonym vs. Descartes		

HTC - Hoare Triple Count

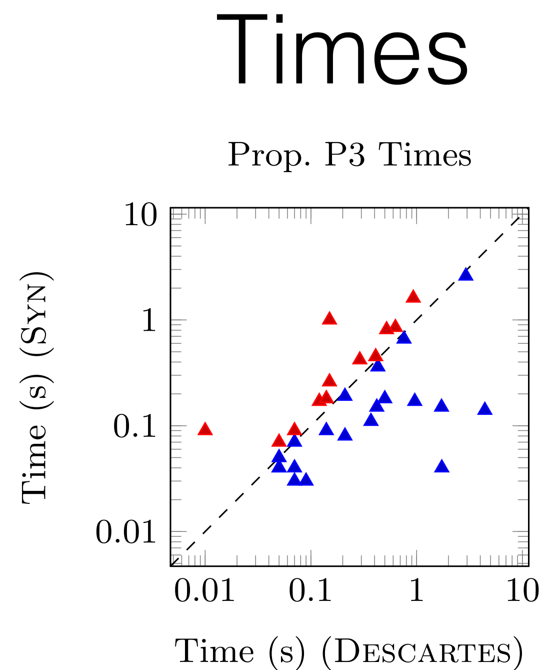
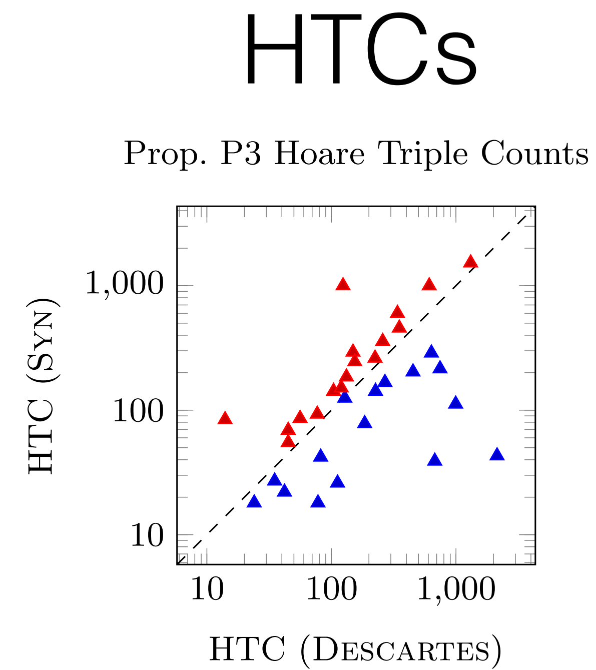
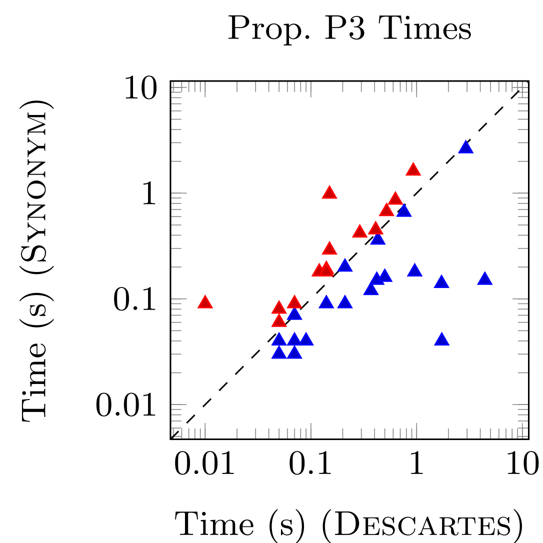
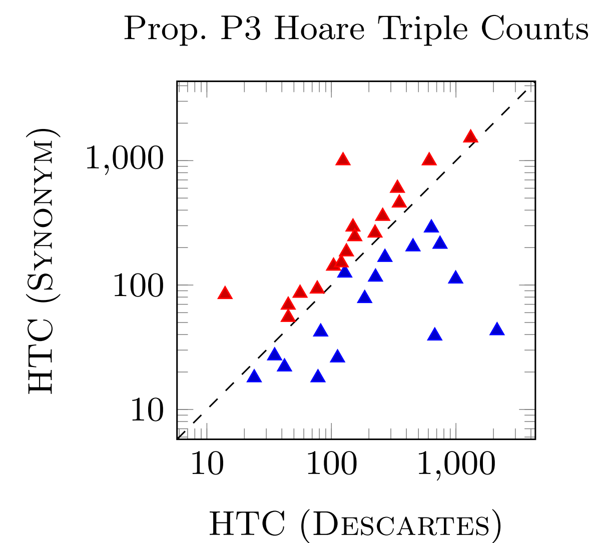
Results: Small Stackoverflow Benchmarks

P2: $\forall x,y,z.\ (compare(x,y) > 0 \wedge compare(y,z) > 0) \Rightarrow compare(x,z) > 0$

	Times	HTCs
Syn vs. Descartes		
Synonym vs. Descartes		

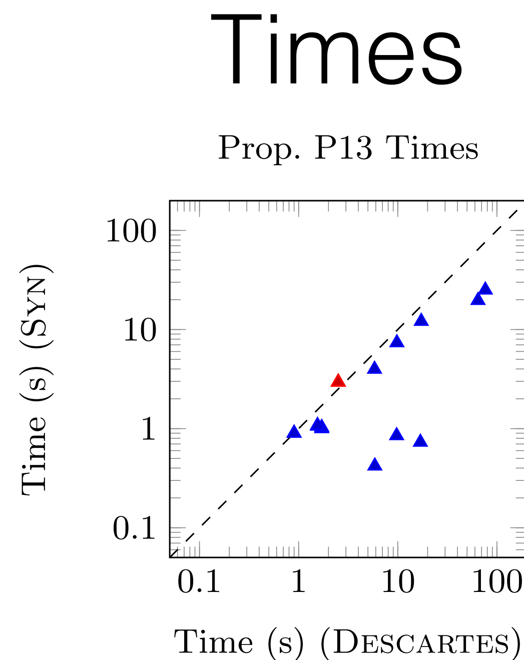
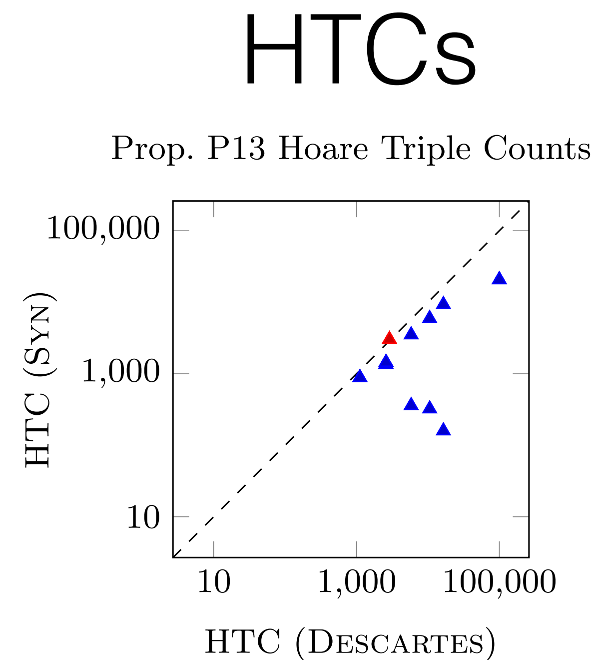
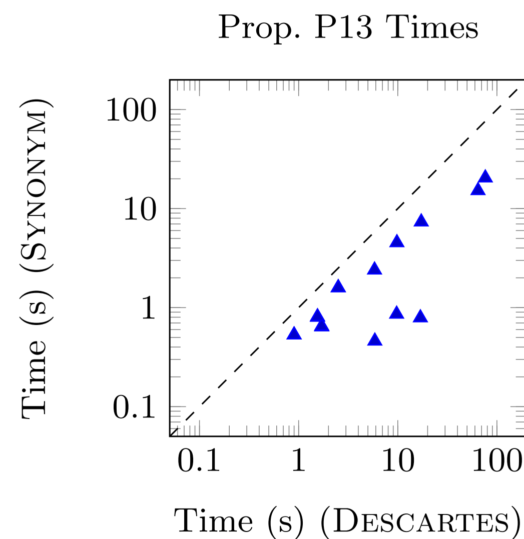
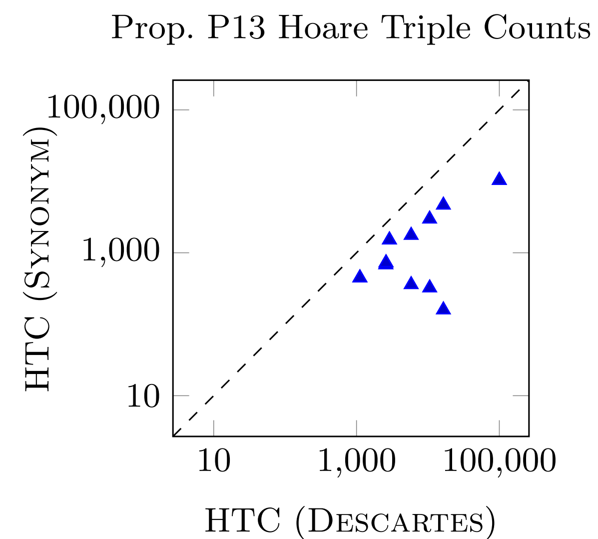
Results: Small Stackoverflow Benchmarks

P3: $\forall x,y,z.\ compare(x,y) = 0 \Rightarrow (sgn(compare(x,z)) = sgn(compare(y,z)))$

	Times	HTCs
Syn vs. Descartes		
Synonym vs. Descartes		

Results: Modified Benchmarks

P13: $\forall x,y,z.pick(x,y,z) = pick(y,x,z)$

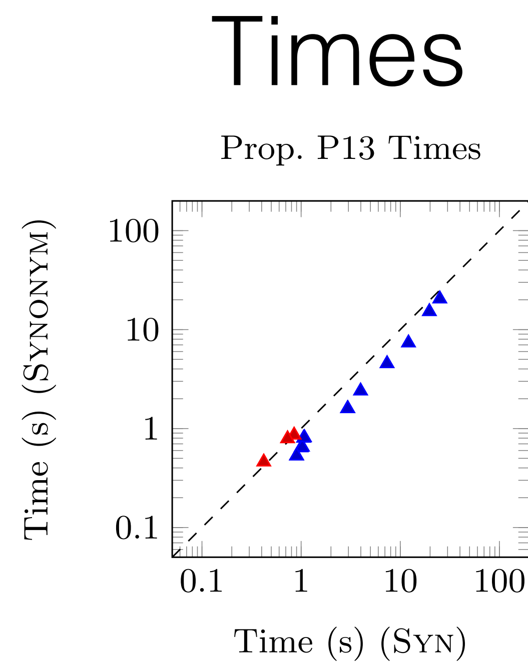
	Times	HTCs
Syn vs. Descartes		
Synonym vs. Descartes		

Results: Modified Benchmarks

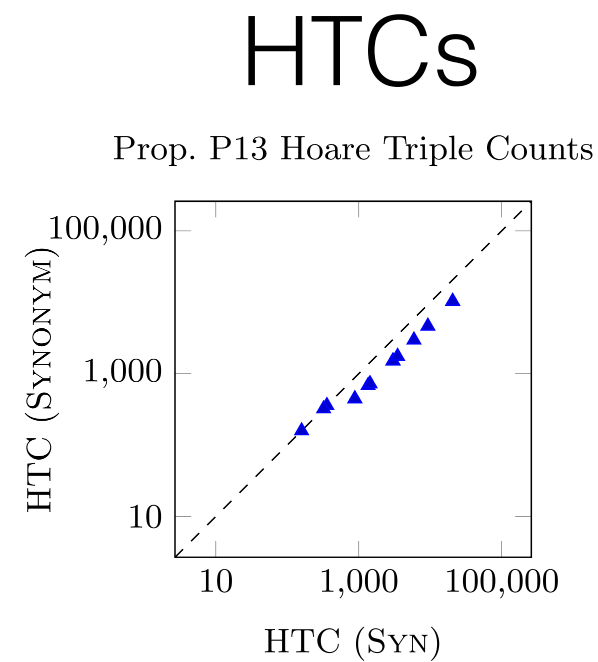
P13: $\forall x,y,z.pick(x,y,z) = pick(y,x,z)$

Synonym vs. Syn

Times



HTCs



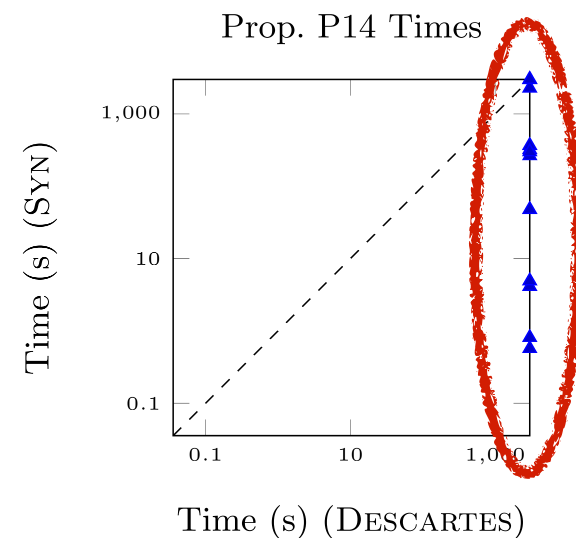
Results: Modified Benchmarks

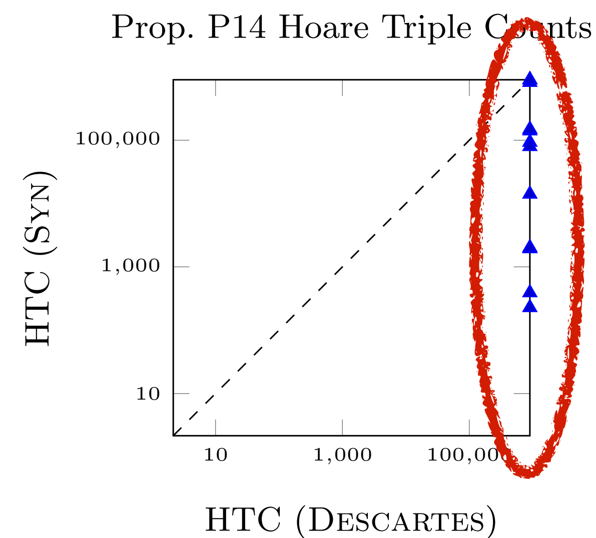
P14: $\forall x,y,z.\ pick(x,y,z) = pick(y,x,z) \wedge pick(x,y,z) = pick(z,y,x)$

Times

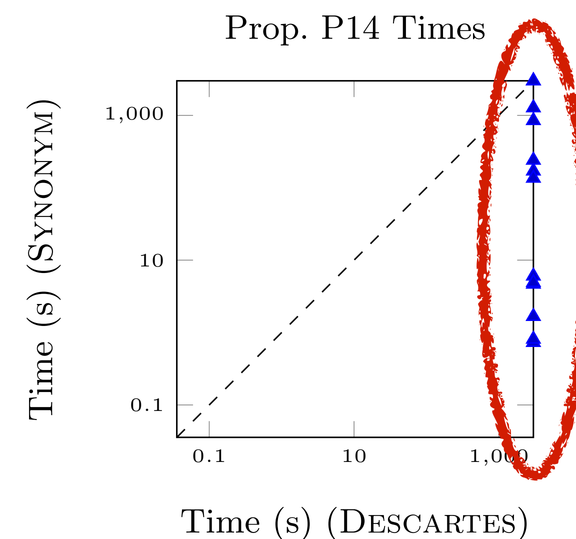
HTCs

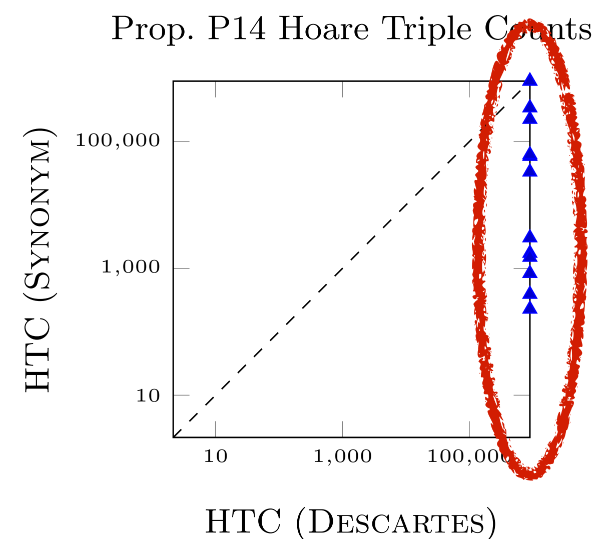
Syn vs. Descartes





Synonym vs. Descartes





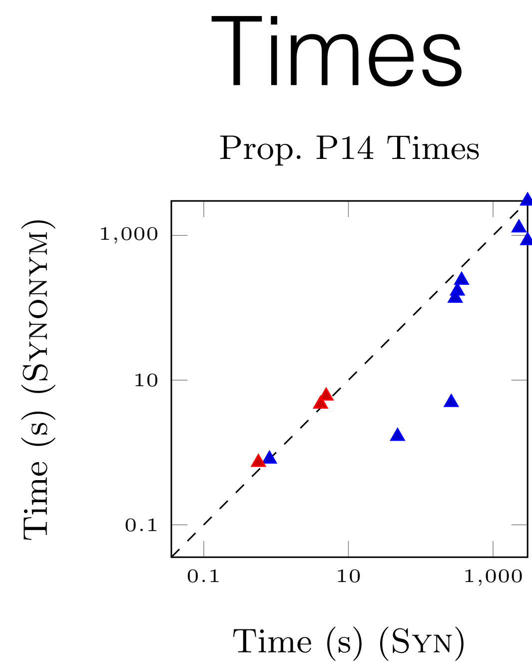
Descartes times out on all examples.

Results: Modified Benchmarks

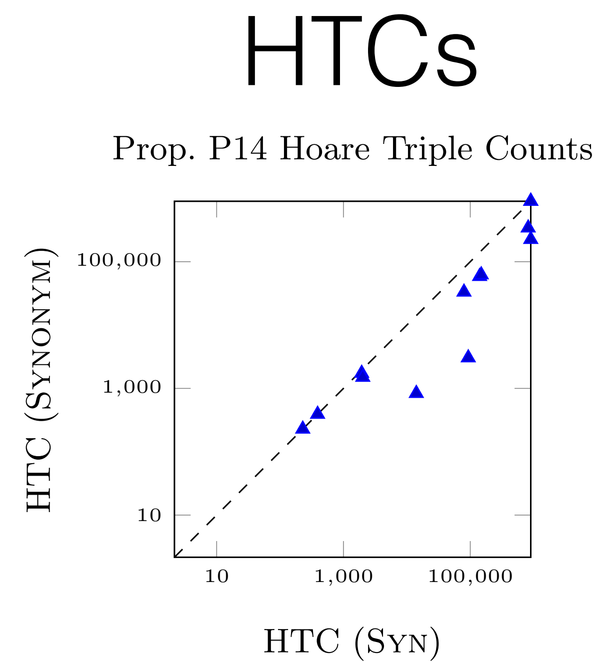
P14: $\forall x,y,z.\ pick(x,y,z) = pick(y,x,z) \land pick(x,y,z) = pick(z,y,x)$

Synonym vs. Syn

Times



HTCs



Related Work

- Cartesian Hoare Logic and Cartesian Loop Logic for relational verification (most closely related)
  - [Sousa and Dillig, 2016]
- Exploiting synchrony (by constructing [some kind of] product program)
  - [Barthe et al. 2011; Lahiri et al. 2013; Strichman and Veitsman 2016; Felsing et al. 2014; Kiefer et al., 2016; De Angelis et al., 2016; Mordvinov and Fedyukovich, 2017]
- Exploiting symmetry in model checking
  - [Emerson and Sistla, 1993; Clarke et al., 1993; Ip and Dill, 1996; Donaldson et al., 2011]
- Without self-composition
  - [Antonopoulos et al., 2017]

Summary

We have seen approaches to addressing the following two challenges in relational verification:

1

How can we maximize the number of loops over which we can compute simpler relational invariants?

2

How can we identify and use symmetries in programs and relational specifications to avoid solving redundant verification problems?

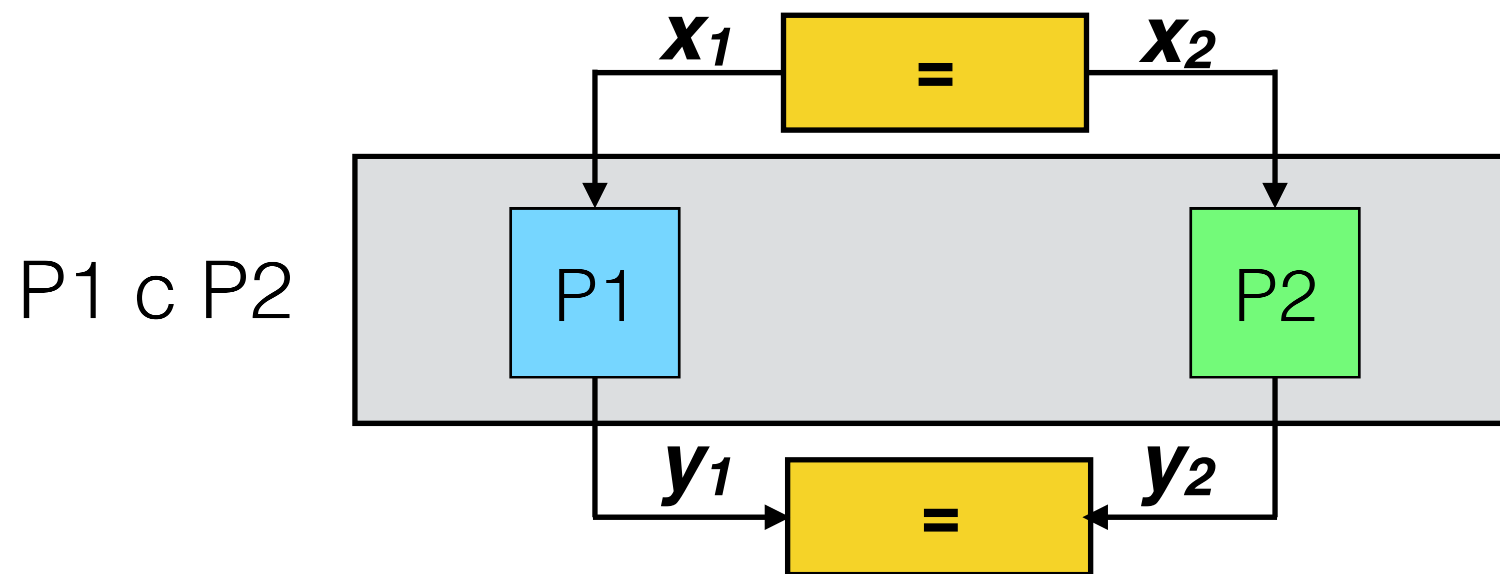
Extra Slides

Composition

Can use standard verification techniques by applying *composition.*
E.g. for equivalence-checking:

{***x1 = x2***} *P1* c *P2* {***y1 = y2***}

where c is a composition operator (e.g. sequential composition or parallel composition)



Challenge: Loops

$\{ x1 < x2 \land i1 = i2 \land i3 > i1 \land x1 > 0 \land i1 > 0 \}$

```
while (i1 < 10) { x1 *= i1; i1++; } ||

while (i2 < 10) { x2 *= i2; i2++; } ||

while (i3 < 10) { x3 *= i3; i3++; }
```

$\{ x1 < x2 \land i1 = i2 \land x1 > 0 \land i1 > 0 \}$

In this case, not all loops can be executed in lockstep, but we still want to execute the first and second loops together.

Symmetric Relational Verification Problems

Two relational verification problem {*pre*} *Ps* {*post*} and {*pre*} *Ps'* {*post*} are symmetric under a permutation π iff

1. π is a symmetry of formula $pre \wedge post'$

2. for every $P_i \in Ps$ and $P_j \in Ps'$, if $\pi(i) = j$, then P_i and P_j have the same number of inputs and outputs and have logically equivalent encodings for the same set of input variables and output variables

Symmetric Formulas

Let $\boldsymbol{x_1},\ldots,\boldsymbol{x_k}$ be vectors of the same size over disjoint sets of variables.

A symmetry π of a formula $F(\boldsymbol{x_1},\ldots,\boldsymbol{x_k})$ is a permutation of set $\{\ \boldsymbol{x_i} \mid 1 \leq i \leq k\ \}$ s.t.

$$F(\boldsymbol{x_1},\ldots,\boldsymbol{x_k}) \Leftrightarrow F(\pi(\boldsymbol{x_1}),\ldots,\pi(\boldsymbol{x_k}))$$

Symmetry-Breaking

We can construct the following symmetry-breaking predicate (SBPs) for the condition (xi > 5)

$$p1 \wedge$$

$$(p1 \Rightarrow (((x1 > 5) \Rightarrow (x3 > 5)) \wedge p2))) \wedge$$

$$(p2 \Rightarrow ((x3 > 5) \Rightarrow (x1 > 5)) \Rightarrow ((x2 > 5) \Rightarrow (x4 > 5)))$$

$x1 > 5 \wedge x3 \leq 5$
not allowed

This is an adaptation of the SBPs constructed for propositional logic in earlier work.

[Aloul et al., 2006]
[Crawford et al., 1996]

Synchrony on Conditionals: Pruning Bonus

$\{ x1 = x2 \}$

if (x1 > 0) then P1 else Q1 || if (x2 > 0) then P2 else Q2

$\{ x1 = x2 \}$

$\{ x1 = x2 \wedge x1 > 0 \wedge x2 > 0\}$	$\{ x1 = x2 \wedge x1 \leq 0 \wedge x2 > 0\}$	$\{ x1 = x2 \wedge x1 > 0 \wedge x2 \leq 0\}$	$\{ x1 = x2 \wedge x1 \leq 0 \wedge x2 \leq 0\}$
P1 || P2	Q1 || P2	P1 || Q2	Q1 || Q2
$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$	$\{ x1 \neq x2 \}$