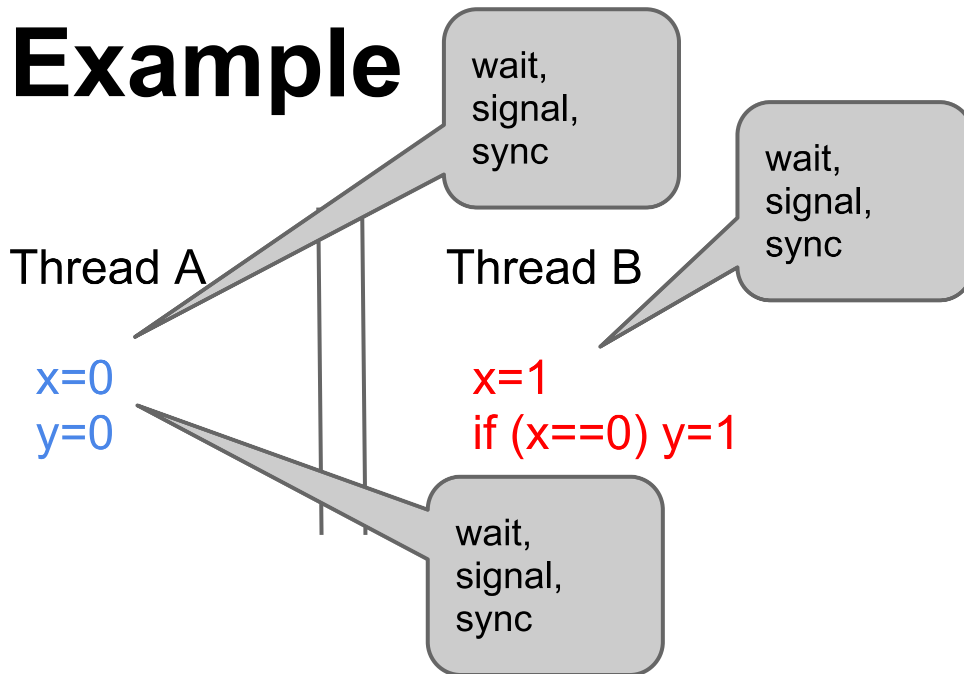
# Automatic Synthesis of Deterministic Concurrency

Veselin Raychev (ETH Zurich)
Martin Vechev (ETH Zurich)
Eran Yahav (Technion)

# Motivation

- Many parallel programs are meant to be deterministic
  - e.g. GPU programs
- Requires manual introduction of synchronization, which is difficult to get right
- Idea: let the tool discover the necessary synchronization automatically

# Example




Traces:

x=0; y=0; x=1; if (x==0) y=1; ➡ (x=1,y=0)

x=0; x=1; y=0; if (x==0) y=1; ➡ (x=1,y=0)

~~x=0; x=1; if (x==0) y=1; y=0; ➡ (x=1,y=0)~~

~~x=1; x=0; y=0; if (x==0) y=1; ➡ (x=0,y=1)~~

~~x=1; x=0; if (x==0) y=1; y=0; ➡ (x=0,y=0)~~

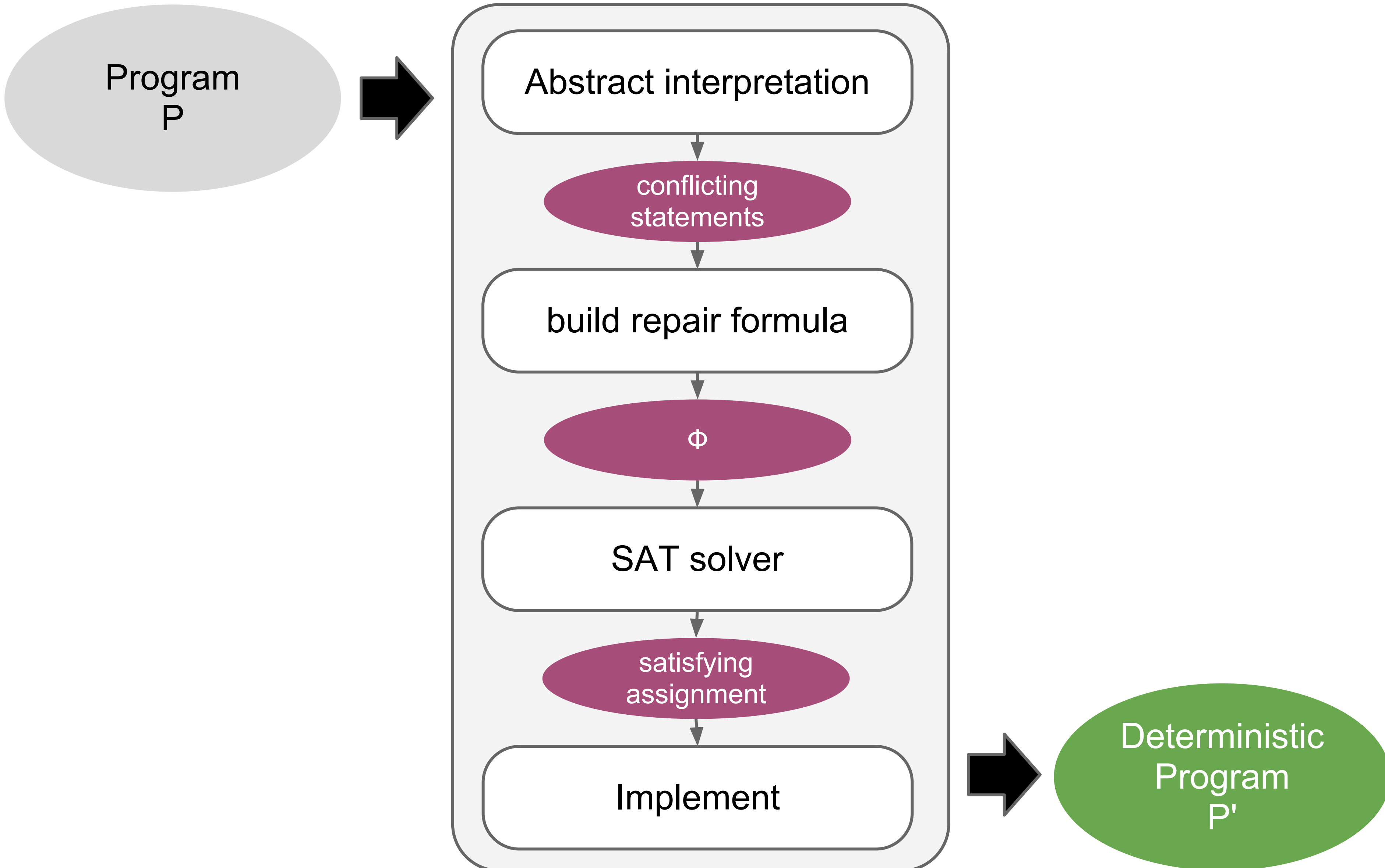
~~x=1; if (x==0) y=1; x=0; y=0; ➡ (x=0,y=0)~~

more traces may be removed

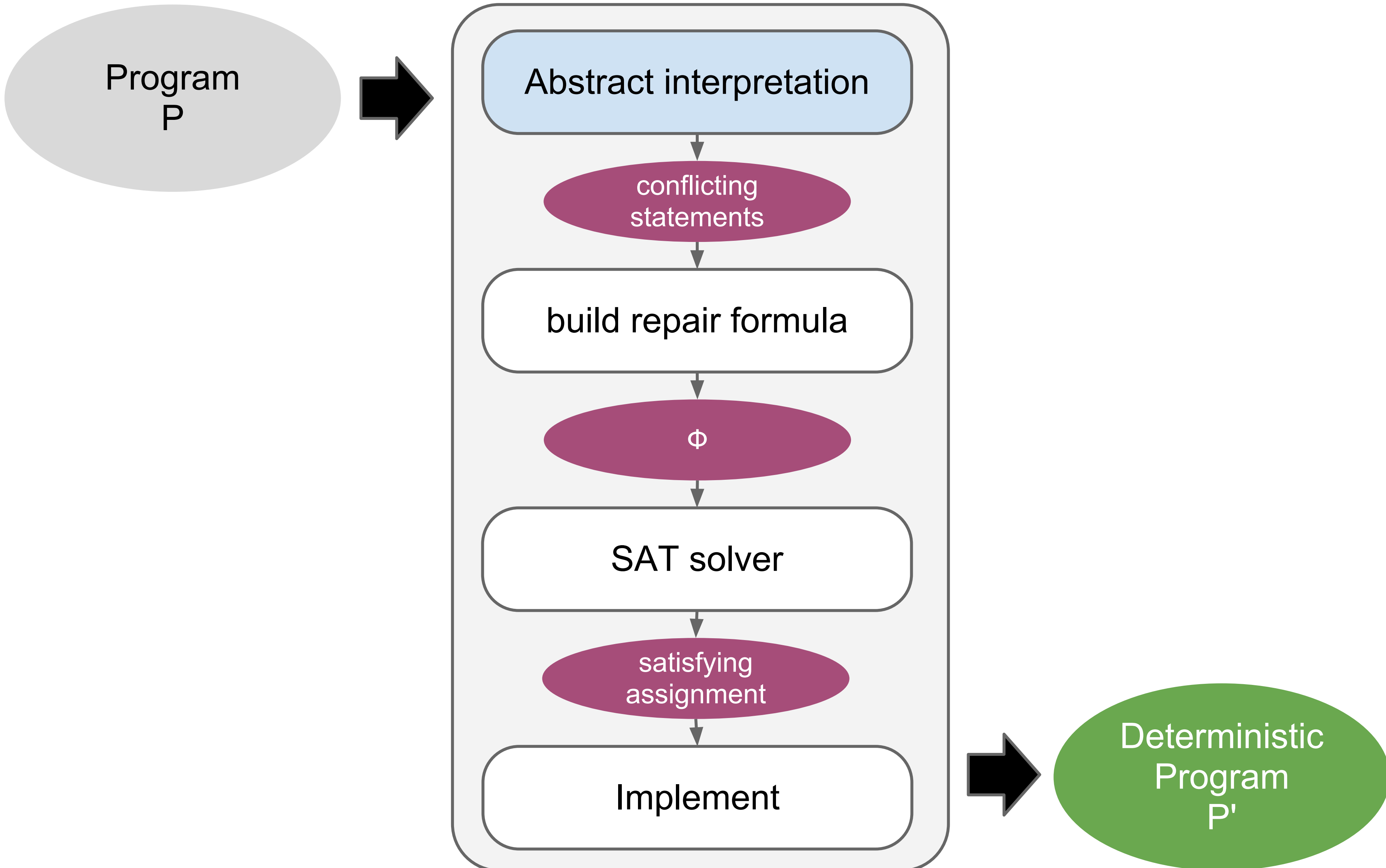
**Goal:**

Given a potentially non-deterministic parallel program, **statically introduce synchronization**, which transforms the program into a deterministic one

# Our approach - determinizing programs



# Our approach - determinizing programs



# Setting

Global state:
(PC1,PC2,x,y)

Thread A

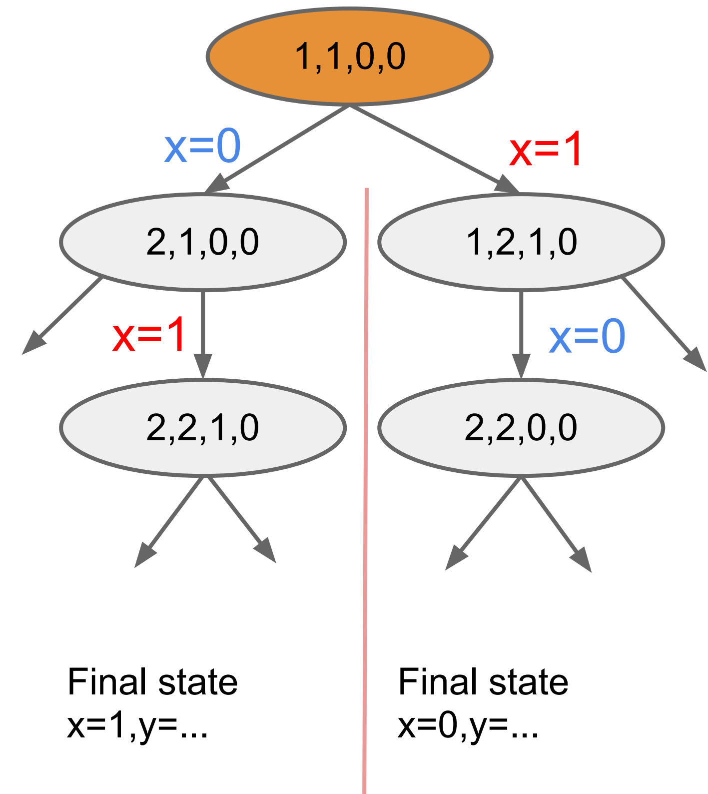
x=0
y=0

Thread B

x=1
if (x==0) y=1

A **conflict state** is one where transitions by different threads are enabled, both transitions access the same memory, and at least one access is a write

Conflict freedom implies determinism*



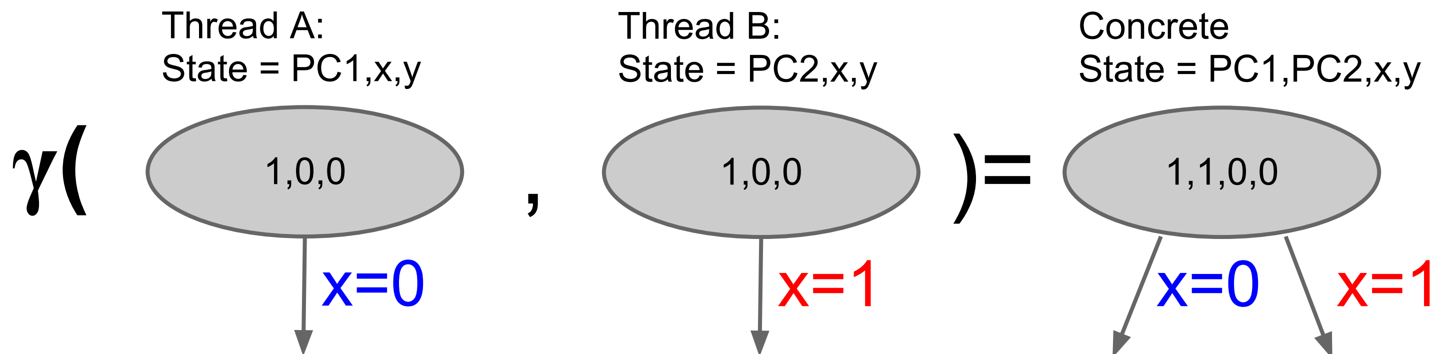
# Global transition system

Bad news: does not scale

# Thread-modular analysis

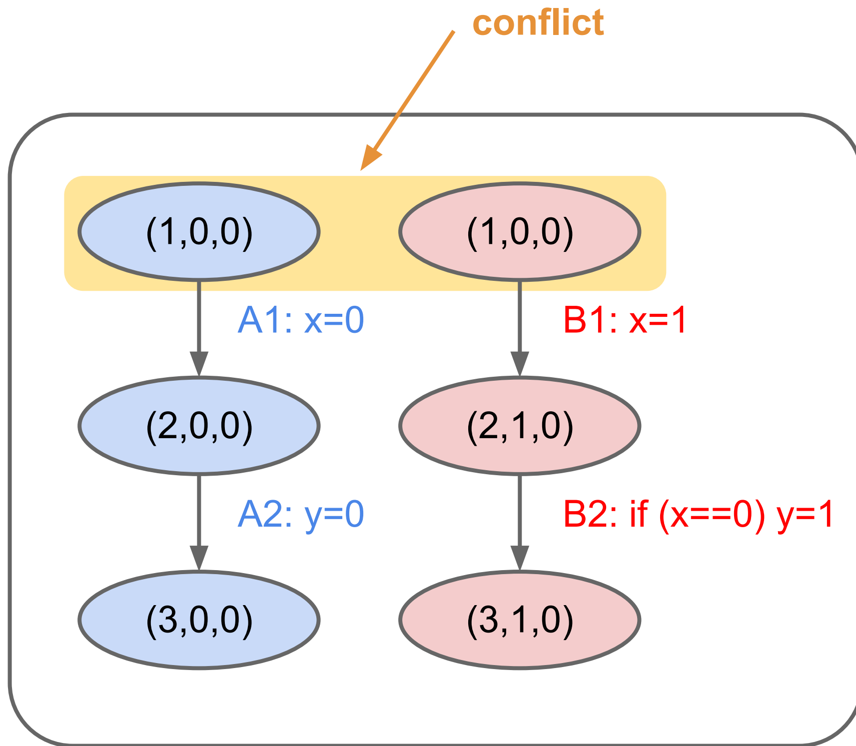
Analyze threads individually

A concrete state is obtained by combining compatible thread local states from different threads




If two threads have conflicting transitions, the combined state is a **conflict state**

# Thread modular analysis

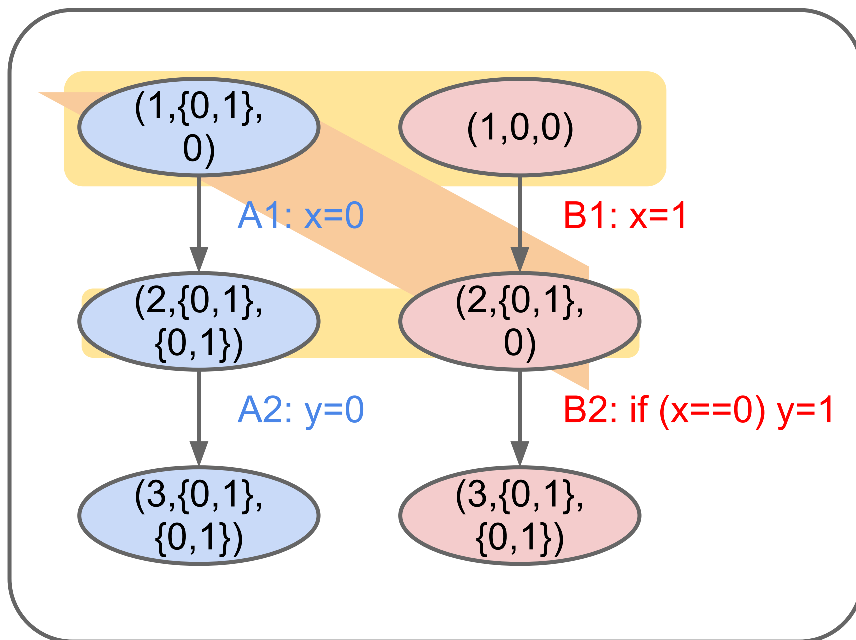



Need to take thread inference into account

Unsound. Did not consider traces like:

x=1; x=0; y=0; if (x==0) y=1; ➔ (x=0,y=1)

# Thread modular analysis

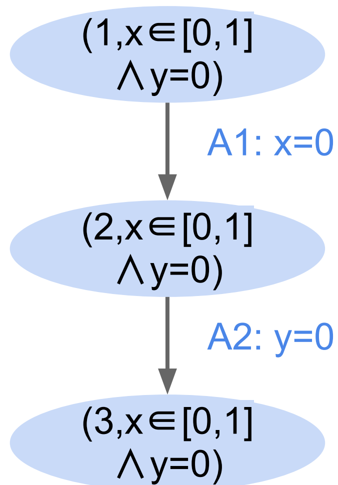



**Solution:**

Propagate values between potentially conflicting statements that may execute in parallel until the analysis stabilizes

It is an over-approximation of all pairs of conflicting transitions

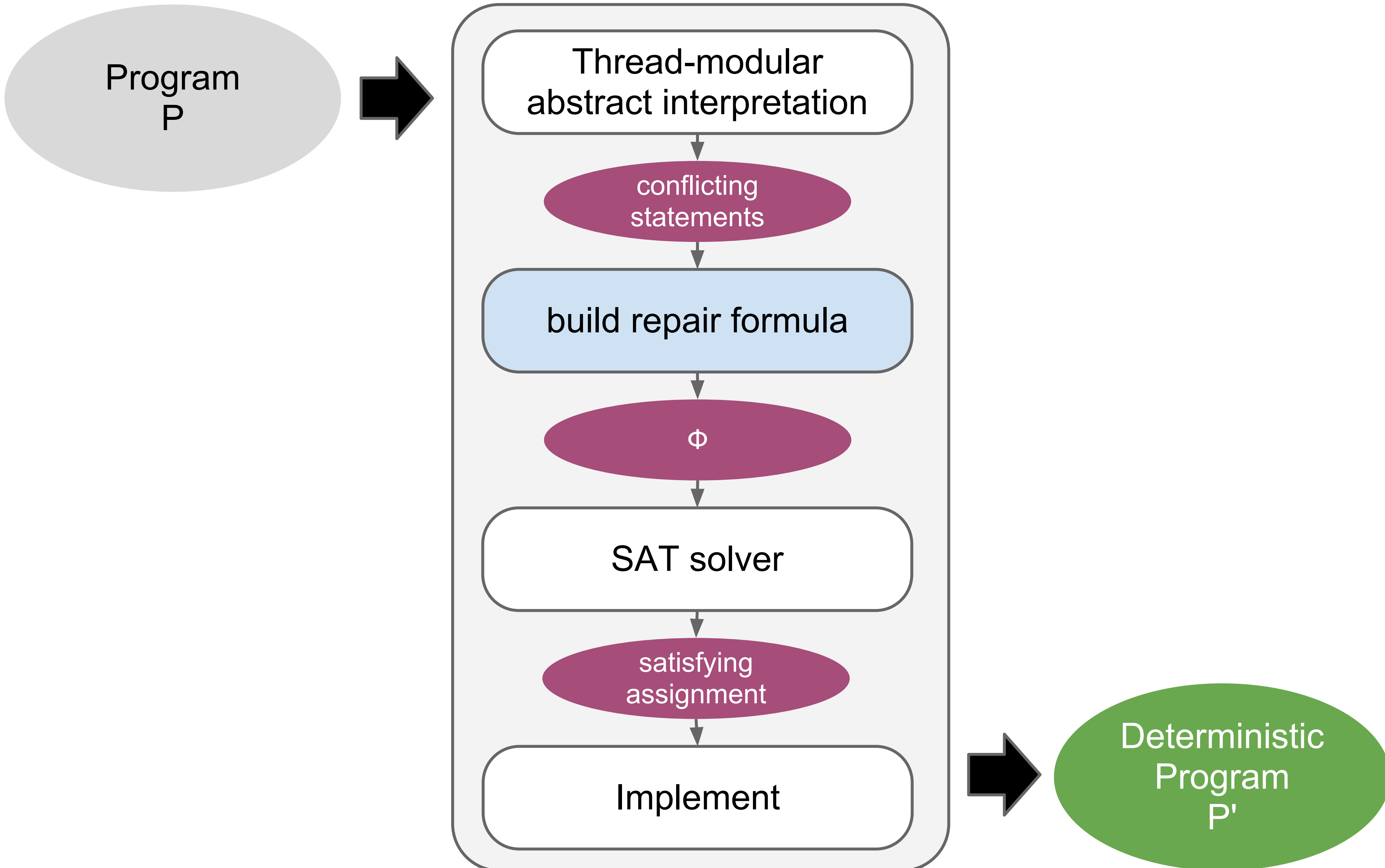
# Analysis of each thread




The programs we handle in our experiments involve the heap and arrays

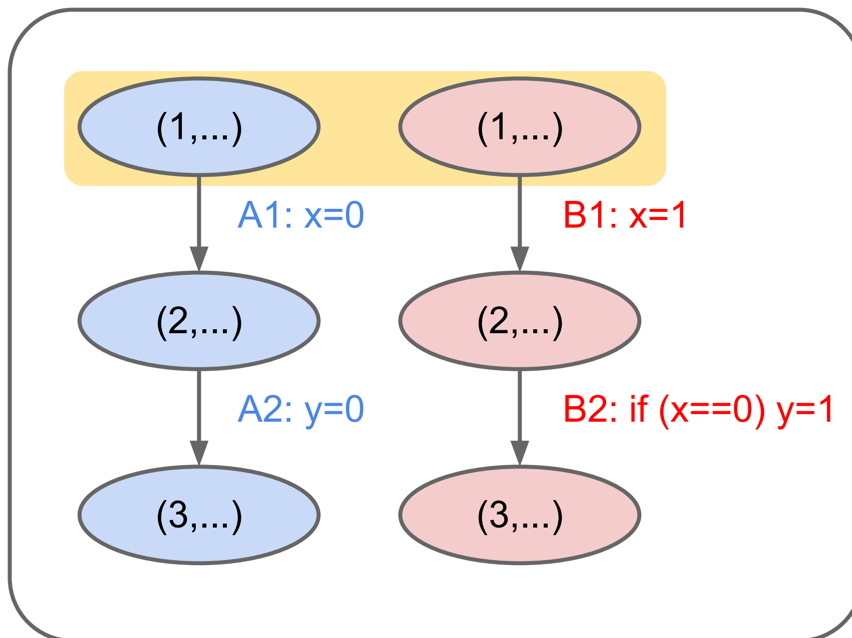
We use a combination of pointer information and numerical abstract domains to capture this information in a precise enough manner for detecting conflicts

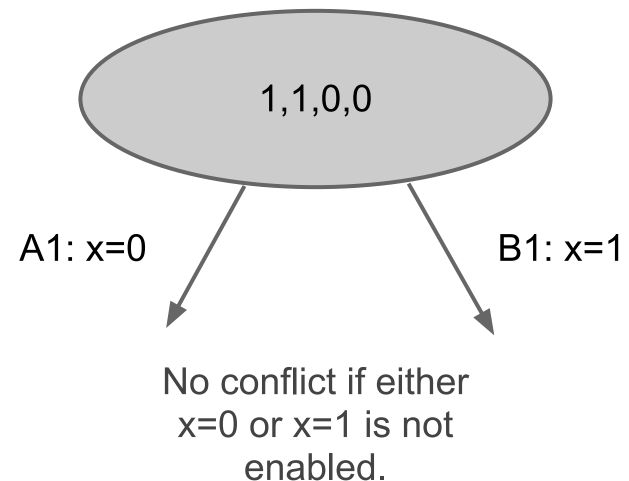
# Our approach - determinizing programs



# Repair conflicts

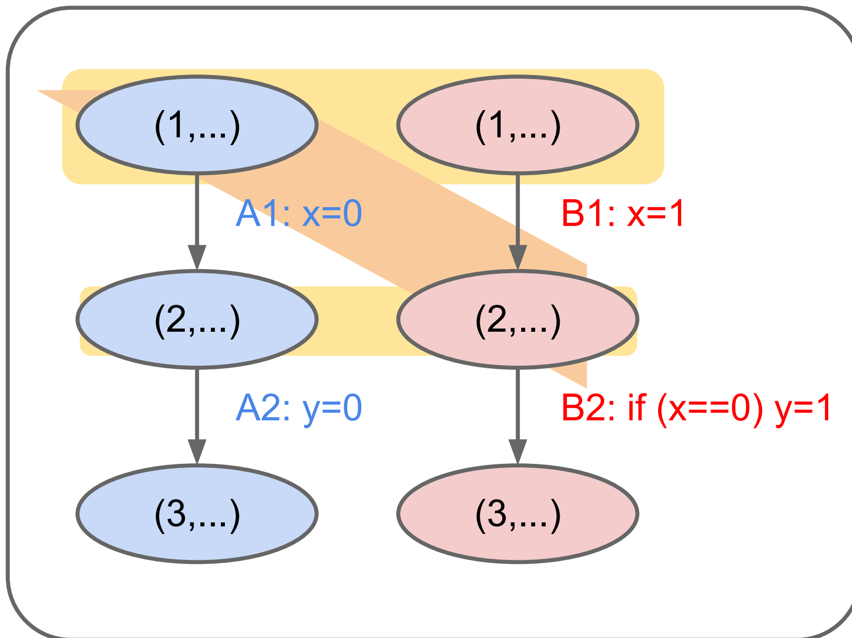
Concrete state:
(PC1,PC2,x,y)




Goal:
Remove the conflict state by removing conflicting transitions (i.e. adding ordering constraints)

# Repair formula




Conjunction of terms fixing each conflict

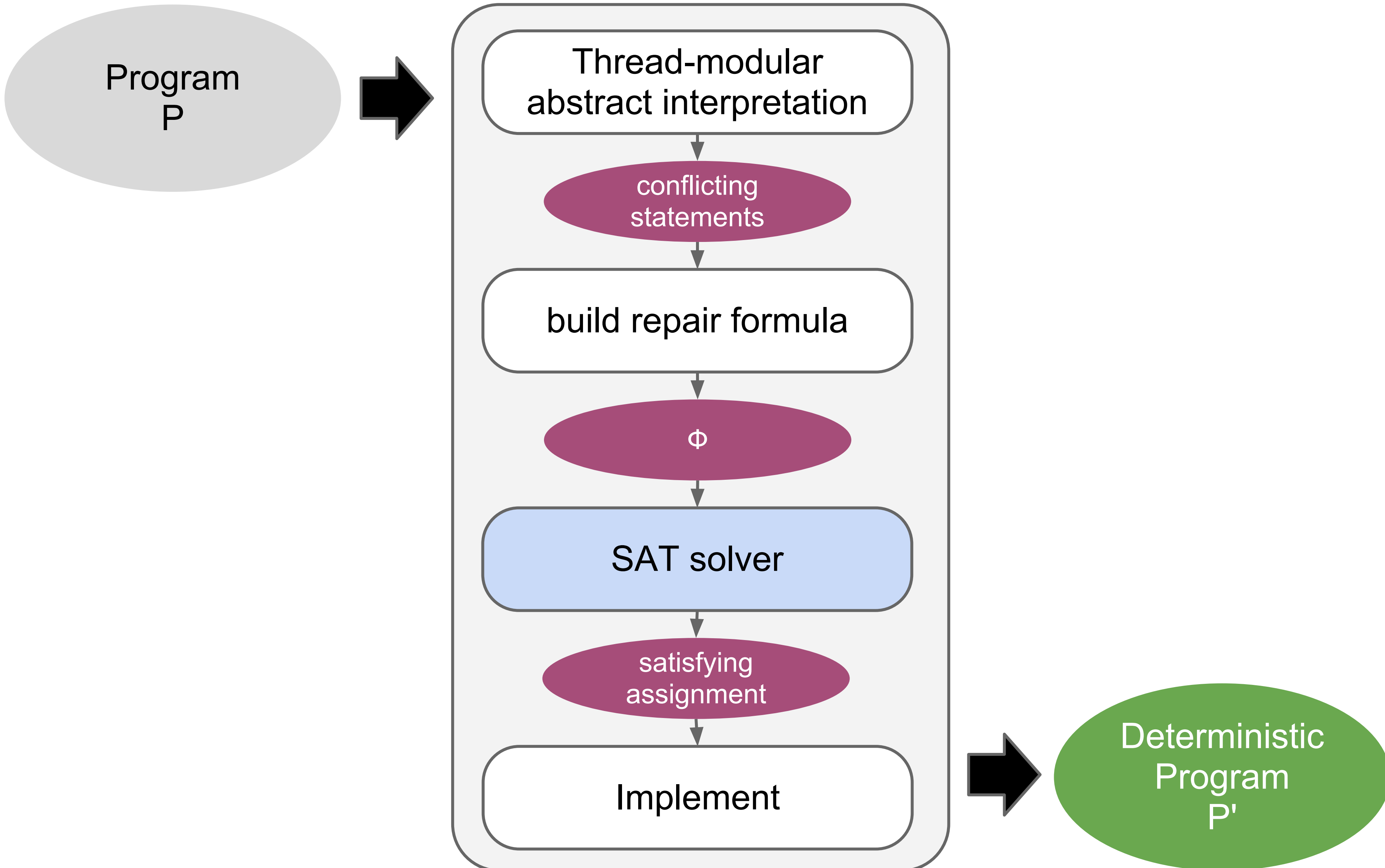
E.g. either B1 is not enabled before A1 executes, or A1 is not enabled before B1 executes.

$$A1<B1 \oplus B1<A1$$

This leads to a formula:

$$\Phi = (A1<B1 \oplus B1<A1) \wedge (A1<B2 \oplus B2<A1) \wedge (A2<B2 \oplus B2<A2)$$

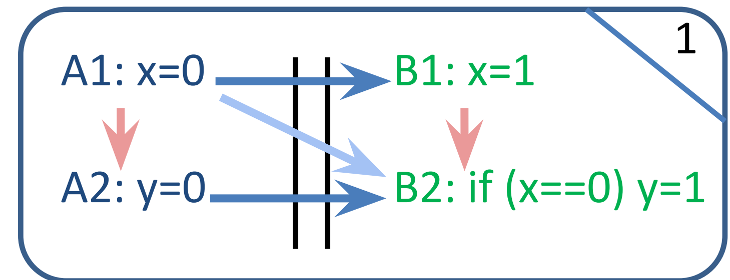
# Our approach - determinizing programs

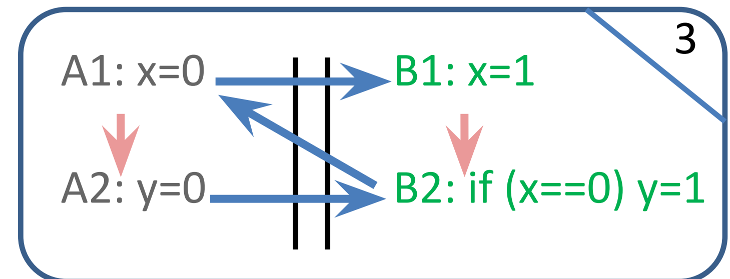


# Satisfying assignments

$$\Phi = (A1<B1 \oplus B1<A1) \wedge (A1<B2 \oplus B2<A1) \wedge (A2<B2 \oplus B2<A2)$$

| Num | Satisfied Terms | |
|---|---|---|
| 1 | A1<B1, A1<B2, A2<B2 | |
| 2 | B1<A1, A1<B2, A2<B2 | ✗ |
| 3 | A1<B1, B2<A1, A2<B2 | ✗ |
| 4 | B1<A1, B2<A1, A2<B2 | |
| 5 | A1<B1, A1<B2, B2<A2 | |
| 6 | B1<A1, A1<B2, B2<A2 | ✗ |
| 7 | A1<B1, B2<A1, B2<A2 | |
| 8 | B1<A1, B2<A1, B2<A2 | |




Some solutions deadlock

# Updating the formula Φ

$\Phi = (A1<B1 \oplus B1<A1) \land (A1<B2 \oplus B2<A1) \land (A2<B2 \oplus B2<A2)$

Add constraints to the formula to disable cycles

$\Phi = (A1<B1 \oplus B1<A1) \land (A1<B2 \oplus B2<A1) \land (A2<B2 \oplus B2<A2) \land (\neg A1<B1 \lor \neg B2<A1) \land (\neg A2<B2 \lor \neg B2<A1)$

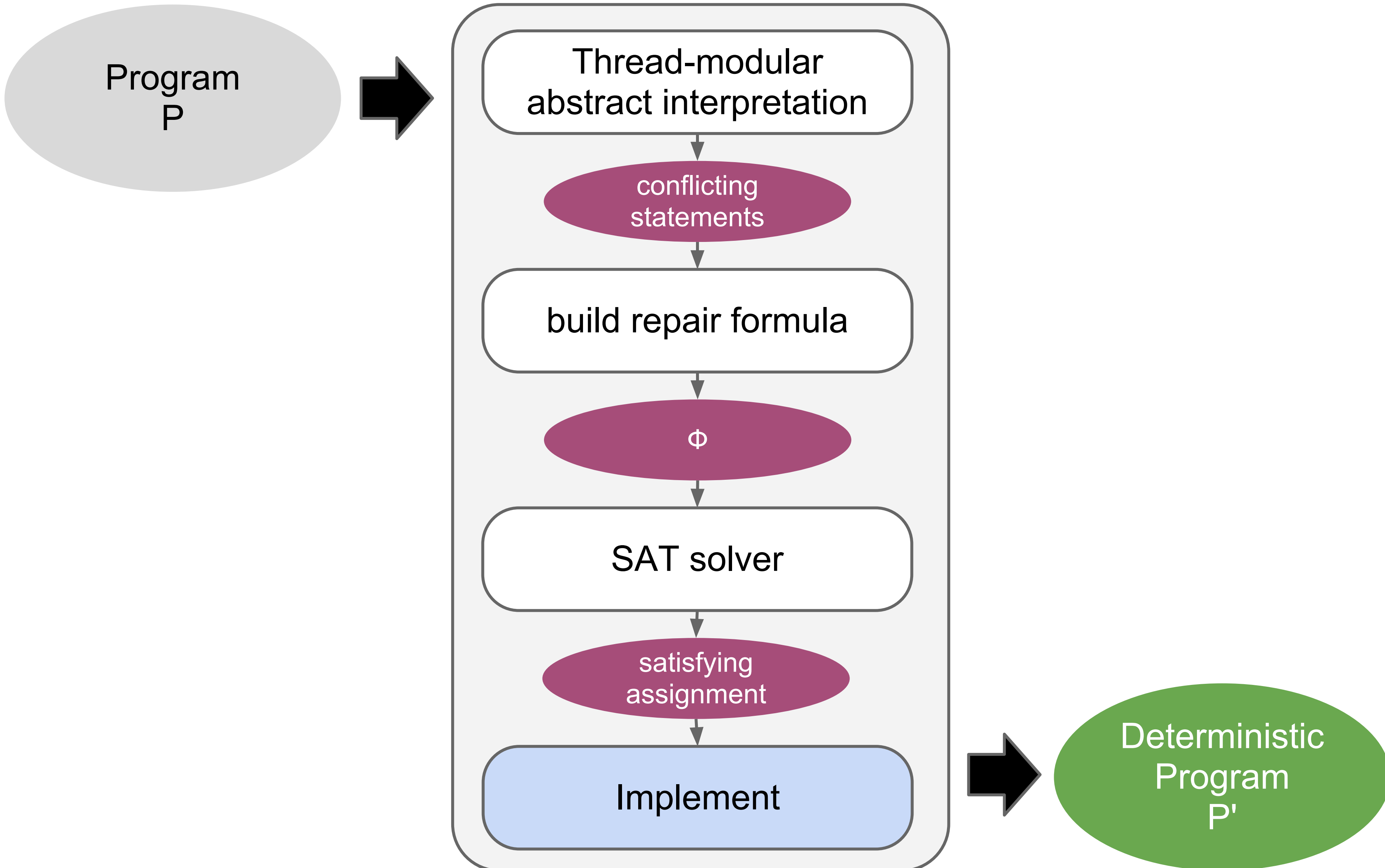
| Num | New Satisfied Terms |
|---|---|
| 1 | A1<B1, A1<B2, A2<B2 |
| 2 | B1<A1, A1<B2, A2<B2 |
| 5 | A1<B1, A1<B2, B2<A2 |
| 6 | B1<A1, A1<B2, B2<A2 |
| 8 | B1<A1, B2<A1, B2<A2 |
| 6 | B1<A1, A1<B2, B2<A2 |
| 7 | A1<B1, B2<A1, B2<A2 |
| 8 | B1<A1, B2<A1, B2<A2 |

# Termination Theorem

Termination is preserved if

- enforced constraints are between labels that execute exactly once
- all threads terminate on their own (no synchronization that is not encoded in the assignment of the formula)
- no cycles

# Our approach - determinizing programs



# Synchronization primitives

- signal/wait
  - Popular thread synchronization construct
  - e.g. present in POSIX
- sync
  - Used in structured parallel programming frameworks such as IBM's X10, MIT's Cilk, etc.

# signal/wait construct

Each object starts non-signalled, wait() on non-signalled object blocks until another thread calls signal()

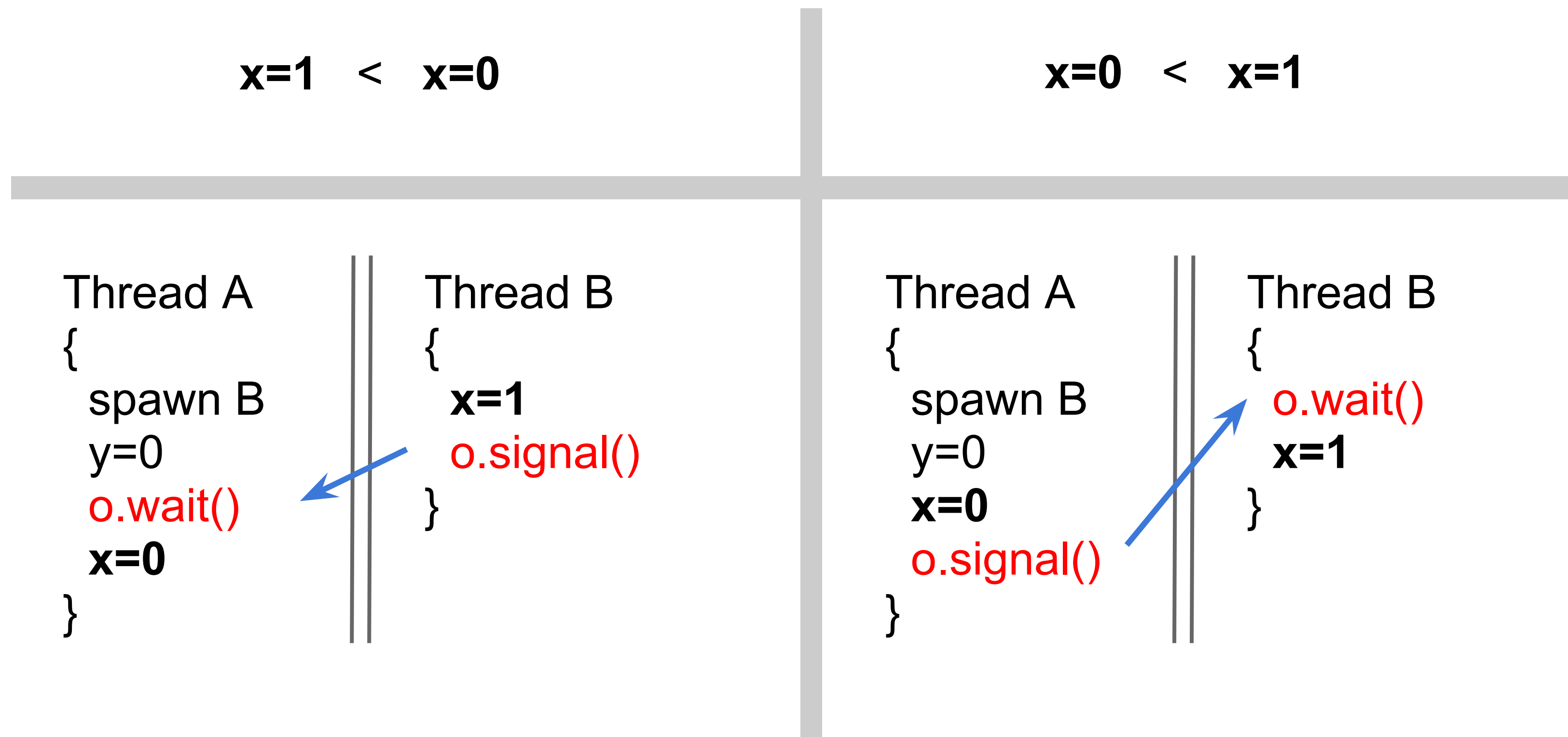
Example program:

| Thread A | Thread B |
| --- | --- |
| { | { |
| spawn B | **x=1** |
| y=0 | } |
| **x=0** | |
| } | |

**x=0** || **x=1** is a conflict

# signal/wait construct

Each object starts non-signalled, wait() on non-signalled object blocks until another thread calls signal()



# sync construct

Blocks a thread until all child threads terminate

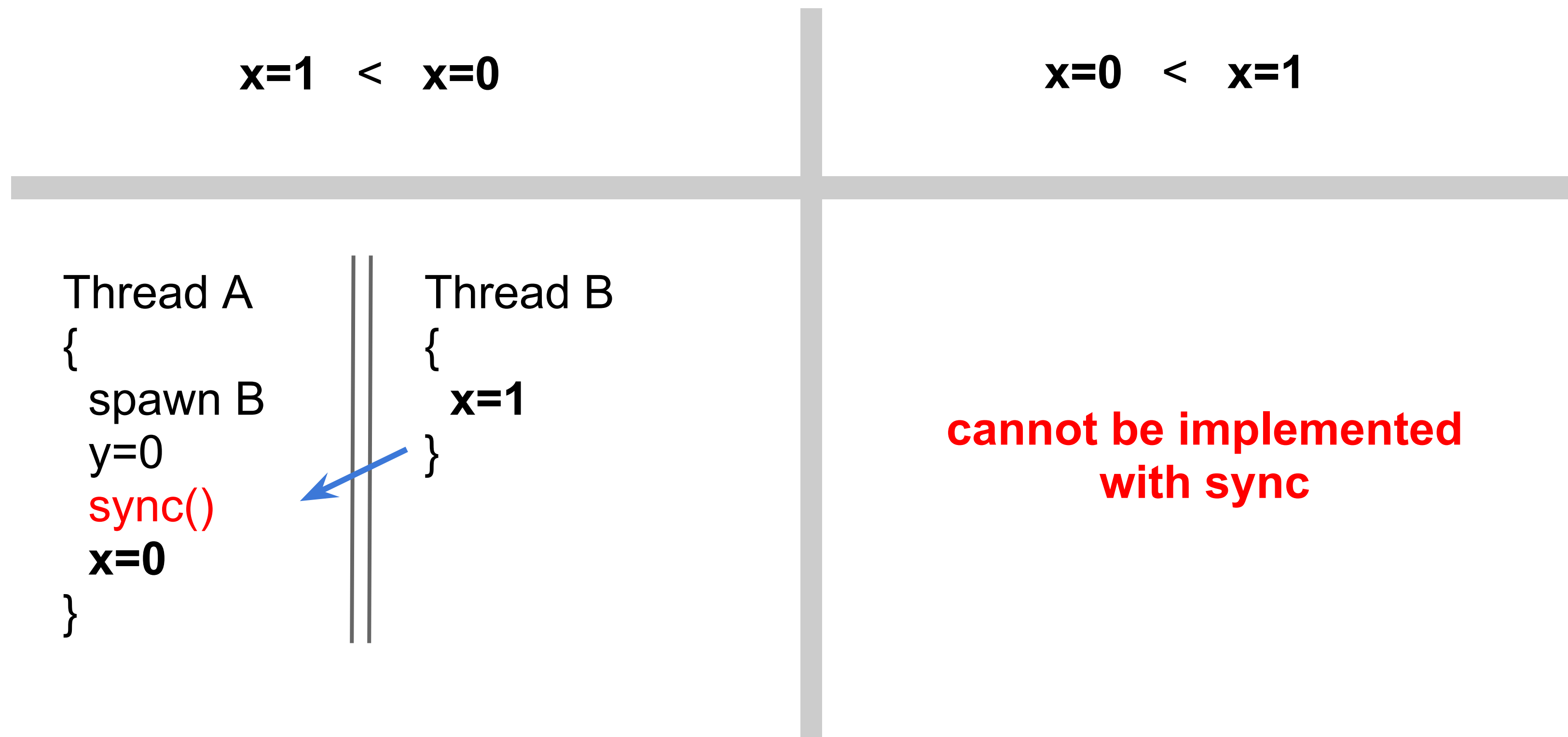
Example program:

```
Thread A          ||  Thread B
{                 ||  {
  spawn B         ||    x=1
  y=0             ||  }
  x=0             ||
}                 ||
```

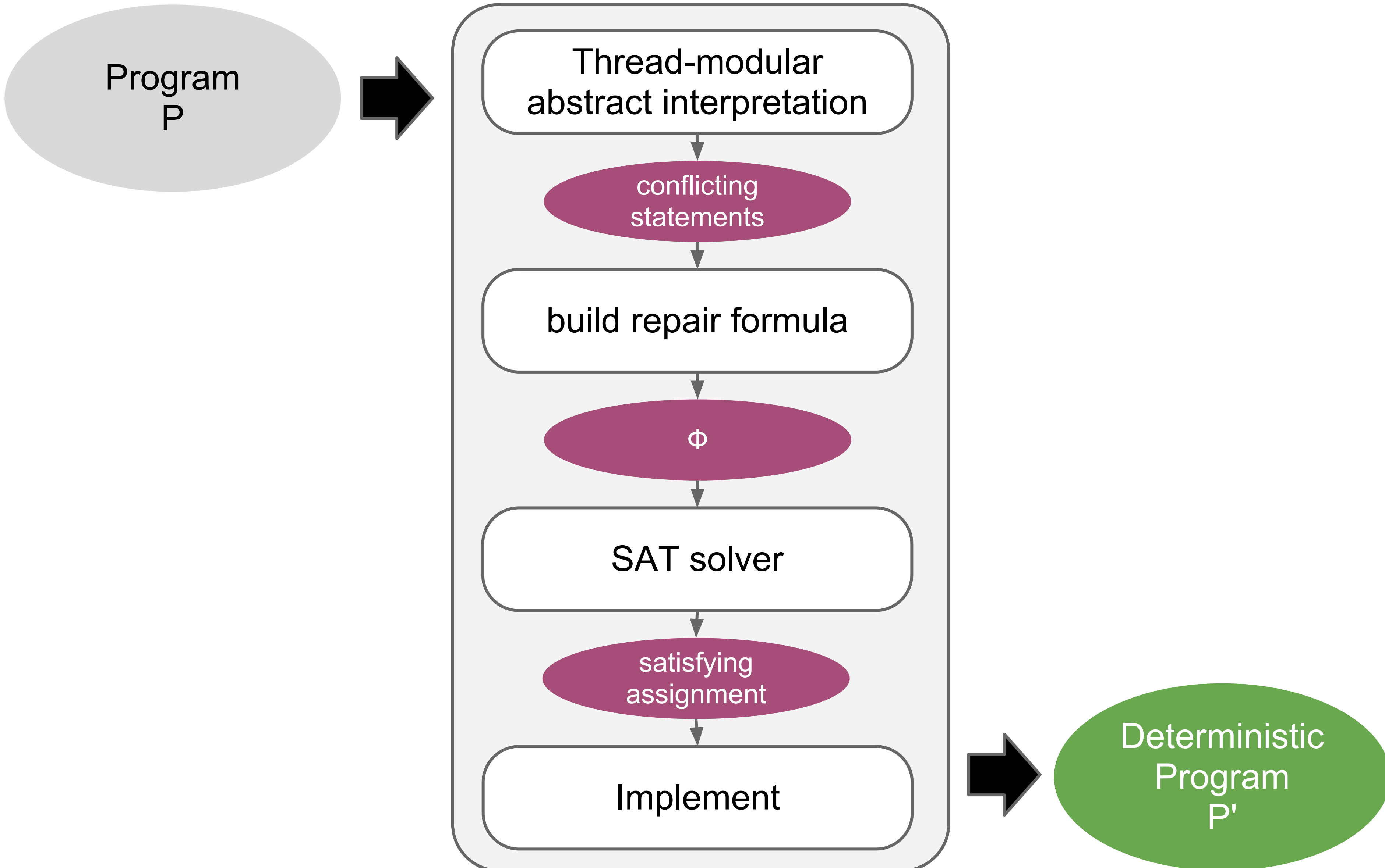
**x=0** || **x=1** is a conflict

# sync construct

Blocks a thread until all child threads terminate



# Our approach - determinizing programs



# Implementation

- Program analysis
  - Thread-modular
  - Pointer analysis: Flow-insensitive
  - Numerical domain: Octagon or Polyhedra

- Implement with given synchronization primitives
  - spawn/signal/wait or spawn/sync

# Implementation

- Handles Java programs

- Analysis framework: Soot  (Jimple), Apron

- SAT Solver: sat4j

- Evaluate on modified Habanero benchmarks
  - High performance community benchmarks by Rice University
  - Original synchronization was removed
  - Number of threads made constant

# Experimental results

## Ability to reconstruct original synchronization

Original synchronization is using sync

| Program | Description | LOC | Octagon | Polyhedra |
|---|---|---|---|---|
| CRYPT | IDEA encryption | 249 | ✗ | ✓ |
| MOLDYN | Molecular dynamics simulation | 579 | ✗ | ✓ |
| SOR | Successive over-relaxation | 119 | ✗ | ✓ |
| LUFACT | LU Factorization | 265 | ✓ | ✓ |
| SERIES | Fourier coefficient analysis | 228 | ✓ | ✓ |
| SPARSE | Sparse matrix multiplication | 93 | ✗ | ✗ |

# Future Work

- Allow more fine-grained constraints
- Refine stabilization procedure to interact with repairs
- Handle other synchronization constructs

# Conclusion

- We statically enforce determinism of infinite-state parallel programs
- We preserve termination
- Our approach uses abstract interpretation for analysis and encodes the repairs as a SAT problem

# Experimental results (2)

Number of determinizations depending on given specification (with Polyhedra)

| Program | Infer Signal/Wait | Infer Signal/Wait W < R |
|---|---|---|
| CRYPT | 6 | 1 |
| MOLDYN | 922 | 72 |
| SOR | 2 | 1 |
| LUFACT | 7 | 4 |
| SERIES | 3 | 2 |
| SPARSE | 2 | 1 |

# Single-transition labels

If A1 and B1 are labels that execute exactly once per run, **enforce for all traces** either A1 to always execute before B1 or B1 to always execute before A1.

If conflicting transition may execute multiple times, add constraints between labels before and after the conflict.

Example: Add to the repair formula

$$A1<B1 \oplus B1<A1$$