

Using Ordering Predicates in Concurrent Program Verification

Pratik Fegade

Supervisor : Ashutosh Gupta

Need for Verification

- Concurrent programs
 - are widely used
 - are harder to write and verify
 - require automated verification

A Difficulty in Concurrent Program Verification

Absence of Thread Modular Proofs

- Non-thread modular: proof refers to local variables of other threads
- Many programs lack thread modular proofs, e.g. Peterson's algorithm, Lamport's bakery algorithm
- State of art tools like Threader[GPR11] and SATABS[CKSY05] do not perform well on such programs
 - Because of no abstraction for program counters

An Example with non-thread modular proof:

Simple Lock

```
//Thread 0
0.0  await (~lock) lock = true;
.
.
.
0.n  lock = false;
```

```
//Thread 1
1.0  await (~lock) lock = true;
.
.
.
1.n  lock = false;
```

- Correctness conditions in Simple Lock
 - 0.n occurs before 1.0 or
 - 1.n occurs before 0.0.

Abstractions for Program Paths

- Currently,
 - The tools use too fine abstractions for program counters
 - The abstractions do not remember any path information
- Leads to large proofs and slow verification

Abstractions for Program Paths

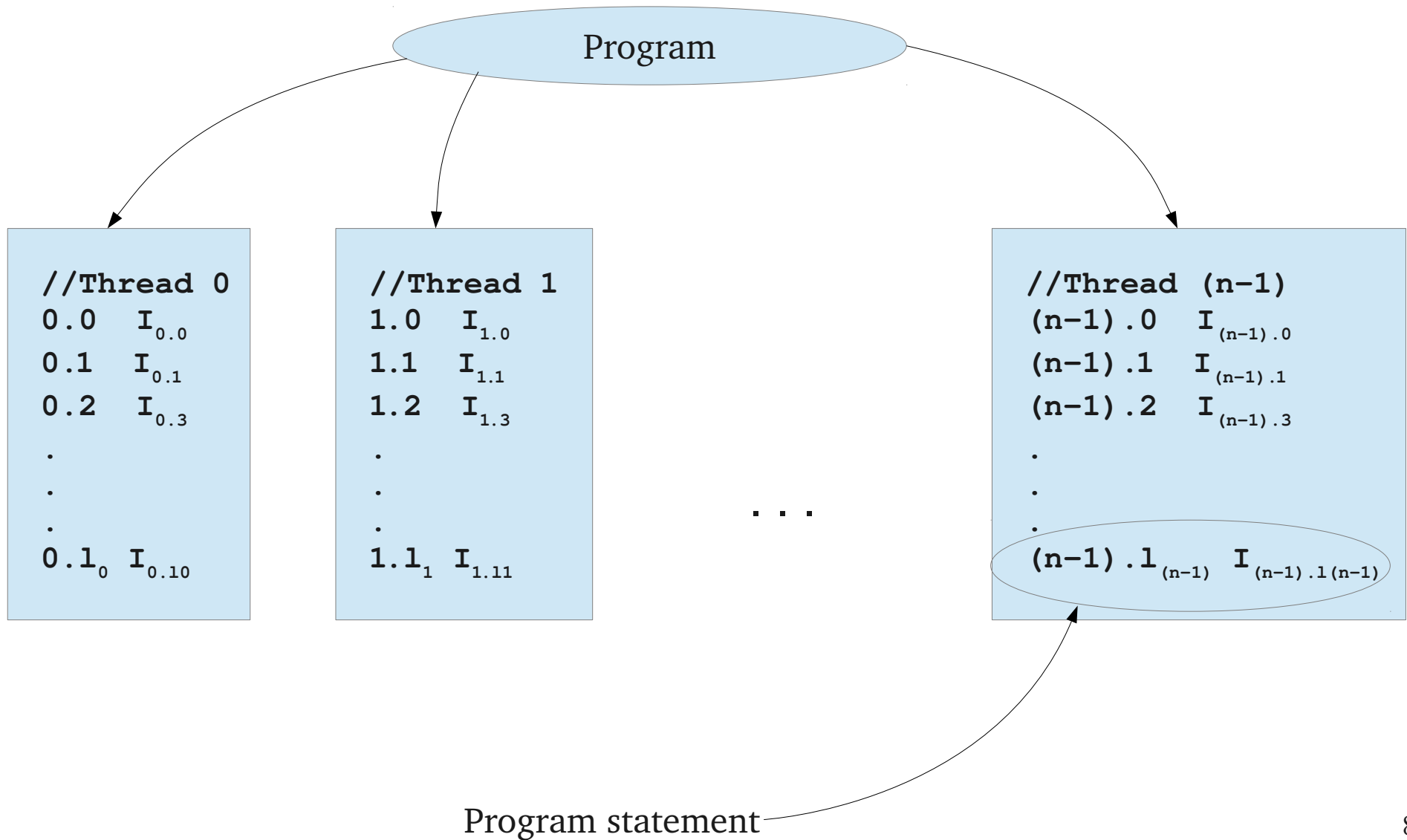
- Interesting correctness properties are expressible using **ordering predicates**, e.g., mutual exclusion
- Why not use these predicates in abstraction and computing fixed points?

**Our idea: an Abstract Interpretation Tool
Using Ordering Predicates**

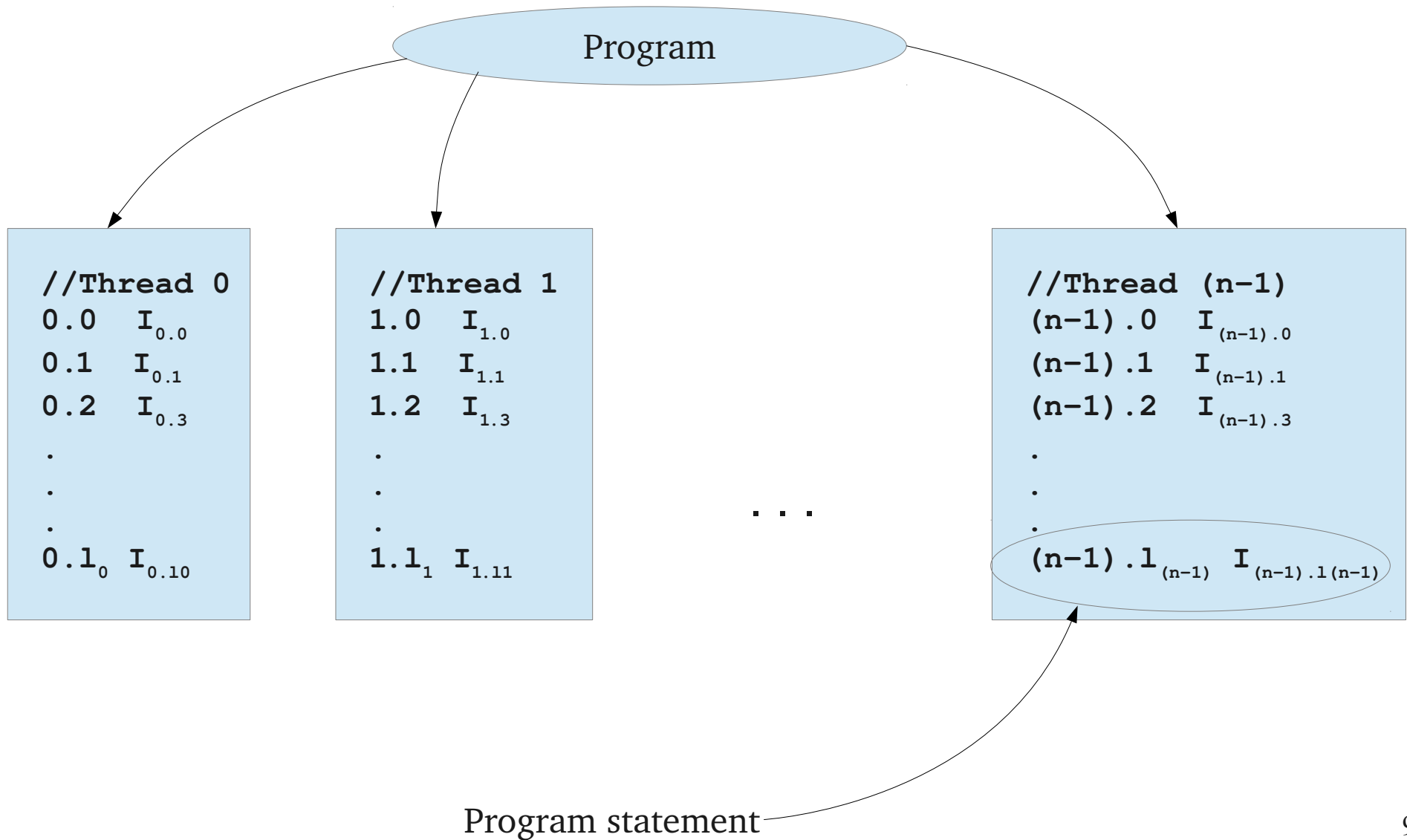
Our Programing Model

- Restricted set of programs
 - Loop free
 - Straight line
 - Only Boolean variables
- Instructions of the following types
 - `skip;`
 - `if (B) then x := E1;`
 - `if (B) then x := E1 else y := E2;`
 - `await (B) do x := E1;`
 - `await (B) do skip;`

Program Structure



Program Structure



Some Definitions

State : A valuation of all program variables, including program counters

History : A tuple $((S_1, S_2, \dots, S_m), R)$ such that R is the state obtained by applying the statements $S_1; S_2; \dots; S_m$ the initial state

Abstraction Interpretation of Concurrent Programs via Owicki Gries Proof Rules

Owicki Gries Proof Rules

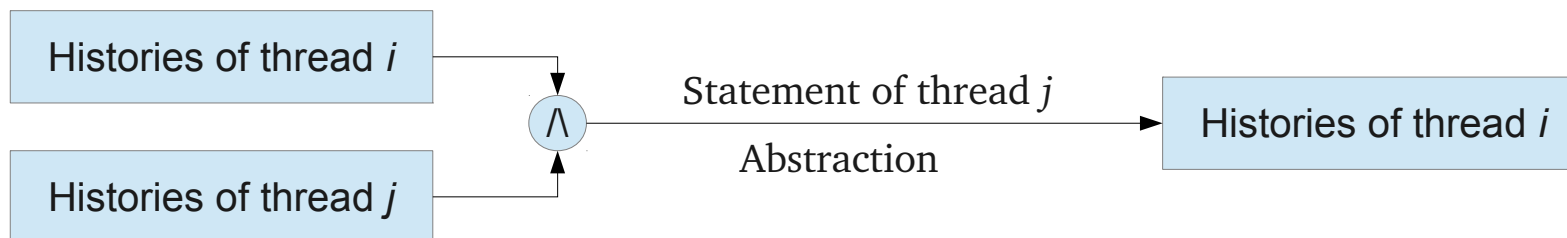
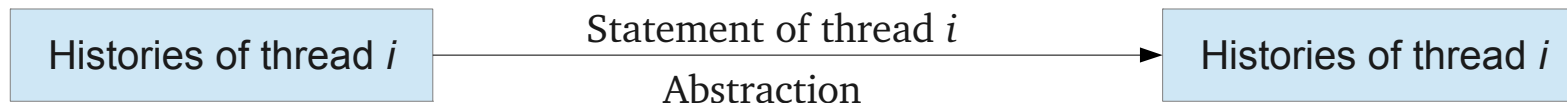
- Start with the initial state
- For each thread
 - Apply transitions i.e. execute program statements to get the post state(s)
 - Prove that the states obtained are interference free i.e. executing a statement of another thread does not lead to a new state

A More Conducive Form of Owicki Gries Rules

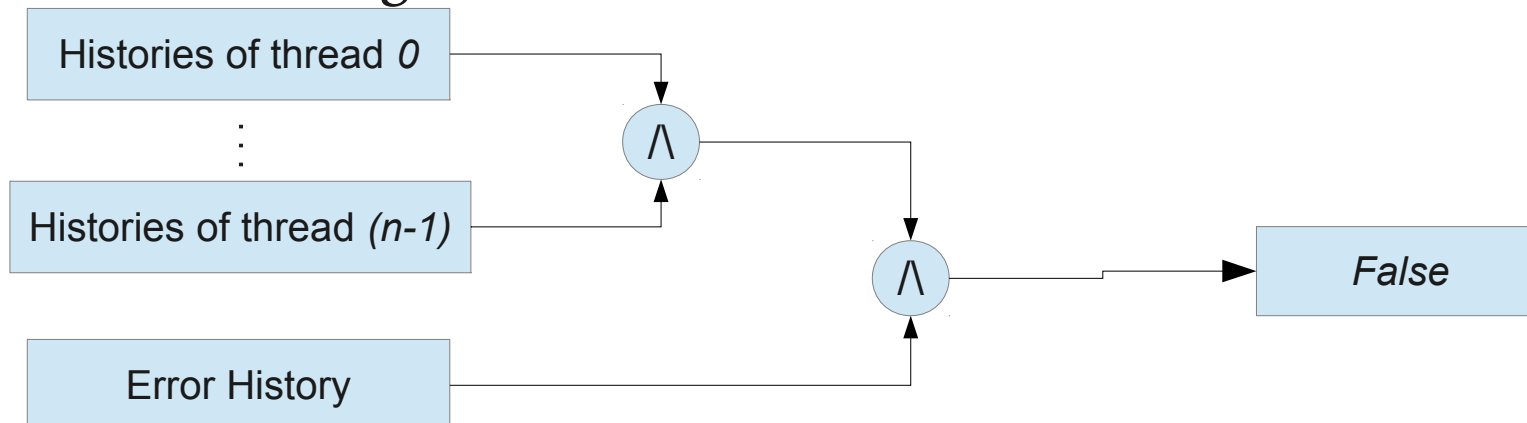
- Initialization



- Recursion



- Error Checking



Our Predicates

- We use the following predicates to describe “history”
 - Happens Before (*HB*)
 - Occurred (*Oc*)
 - Program Counters (*pc*)

Happens Before Predicate

- A binary relation over program statements
- Orders statement executions
- History H satisfies $HB(S_1, S_2)$ if S_1 is executed before S_2 in H
- An Example

History $H = ((S_1, S_2, S_3), R)$ satisfies $HB(S_1, S_2) \wedge HB(S_1, S_3)$ but not $HB(S_1, S_4) \vee HB(S_2, S_1)$

Occurred Predicate

- A predicate over the set of program statements
- States if a program statement has been executed
- An Example

History $H = ((S_1, S_2), R)$ satisfies $Oc(S_1) \wedge Oc(S_2)$

Abstraction Interpretation of Concurrent Programs via Owicki Gries

- We need ways to
 - Check entailment for our predicates
 - Compute the post given a set of histories and a statement.

Inference Rules

- Straight line programs imply linear program execution rules
- Happens Before predicate satisfies
 - Transitivity
 - Law of Trichotomy

Inference Rules: Some Examples

- Straight line programs
 - $(pc_0 = 0.7) \rightarrow Oc(0.3)$
- Transitivity of HB
 - $HB(1.2, 2.3) \wedge HB(2.3, 0.5) \rightarrow HB(1.2, 0.5)$
- Law of Trichotomy
 - $Oc(1.2) \wedge Oc(2.3) \rightarrow HB(1.2, 2.3) \vee HB(2.3, 1.2)$

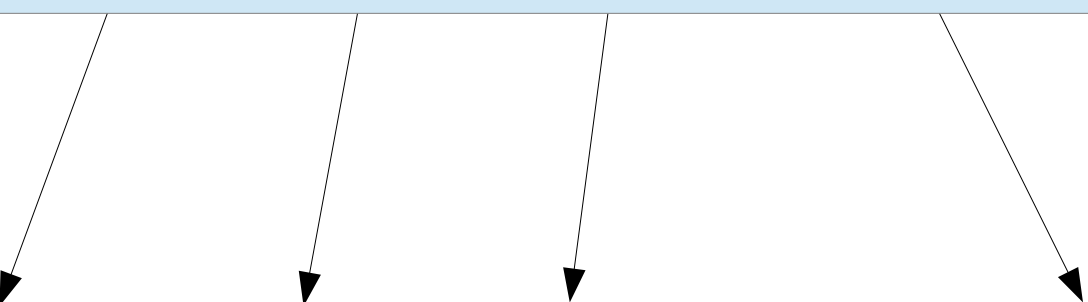
Strongest Postcondition Operator

- Takes as input a set of histories a and a program statement and outputs a set of histories
- In a way, executes the input statement
- Basically,
 - Increments the program counter (if any)
 - Replaces/modifies previous predicates to account for execution of input statement

Strongest Postcondition Operator: An Example

Execution of statement 1.6

$Oc(0.4) \wedge (pc_0=0.6) \wedge (pc_1=1.6) \wedge \sim HB(1.6, 2.3)$



$Oc(1.6) \wedge HB(0.4, 1.6) \wedge (pc_0=0.6) \wedge (pc_1=1.7) \wedge (Oc(1.6) \vee \sim Oc(2.3) \vee HB(2.3, 1.6))$

Prototype Implementation

- A prototype implemented to compute the fixed point given an abstraction
- Proved the following programs to be safe
 - Peterson's algorithm for 2 threads
 - Simple Lock
 - A simple producer-consumer program
- Correctly determined the following to be unsafe with fine abstractions
 - Wrong Lock
 - Wrong Peterson's algorithm for 2 threads

An Example: Simple Lock

- The Program

```
//Thread 0
0.0  await (~lock) lock = true;
0.1  skip;
0.2  skip;
0.3  skip;
0.4  lock = false;
```

```
//Thread 1
1.0  await (~lock) lock = true;
1.1  skip;
1.2  skip;
1.3  skip;
1.4  lock = false;
```

- Abstraction used

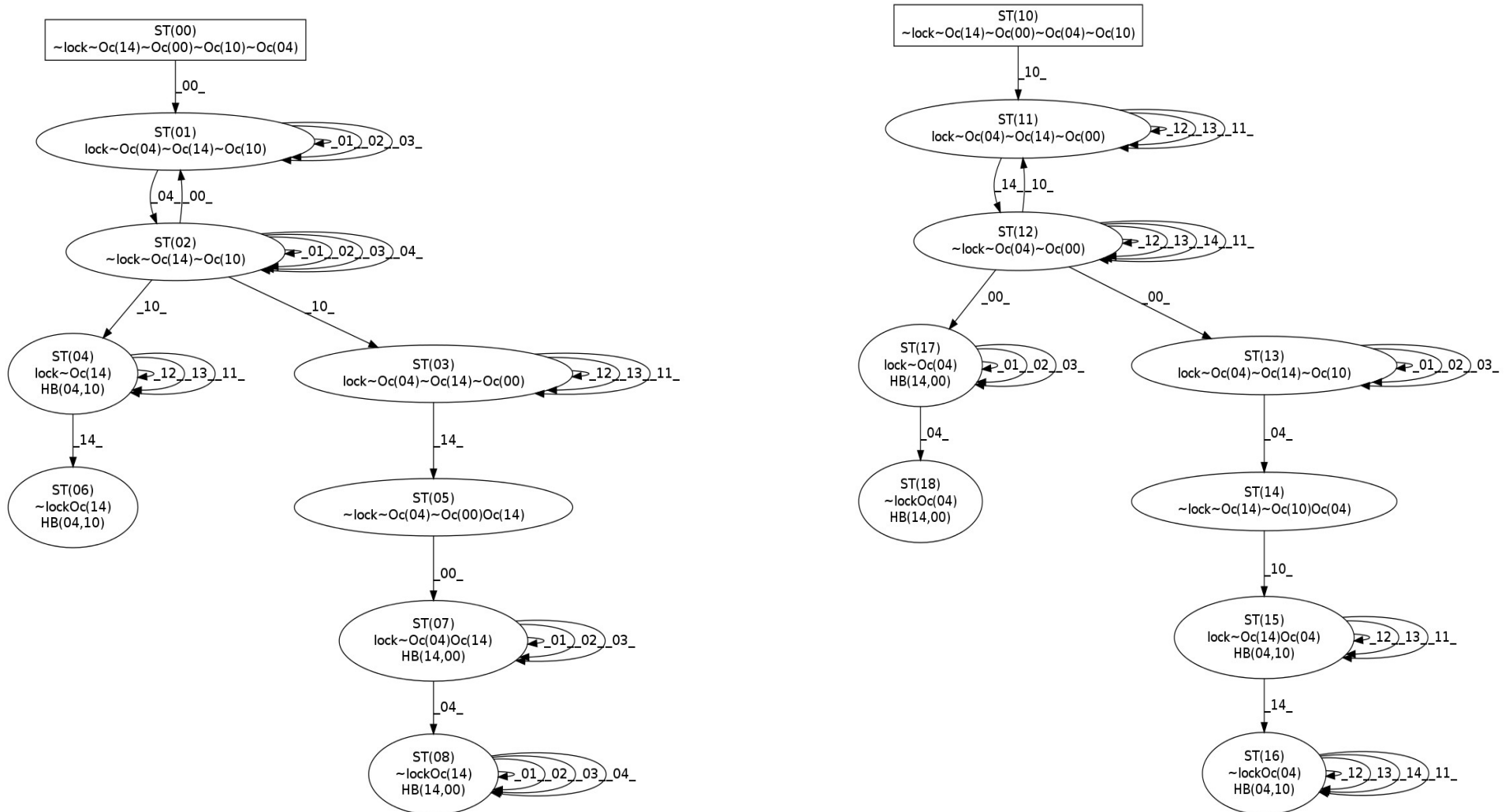
- Thread 0

$\{Oc(1.4), \sim Oc(0.0), \sim Oc(0.4), \sim Oc(1.0), \sim Oc(1.4), HB(0.4, 1.0), HB(1.4, 0.0)\}$

- Thread 1

$\{Oc(0.4), \sim Oc(1.0), \sim Oc(1.4), \sim Oc(0.0), \sim Oc(0.4), HB(1.4, 0.0), HB(0.4, 1.0)\}$

Simple Lock: Counter Example Free Fixed Point



Thank You!