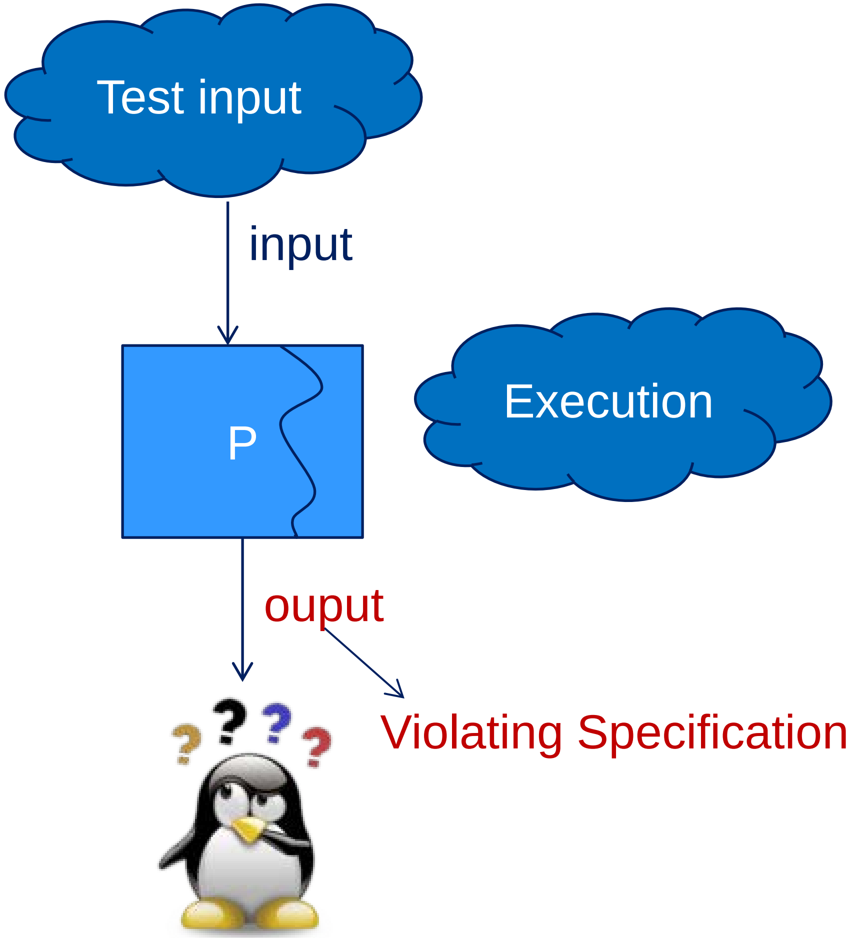
Explaining Concurrency Bugs with Interpolants

Andreas Holzer[2], Daniel Schwartz-Narbonne[3], **Mitra Tabaei[1]**, Georg Weissenbacher[1], Thomas Wies[3]

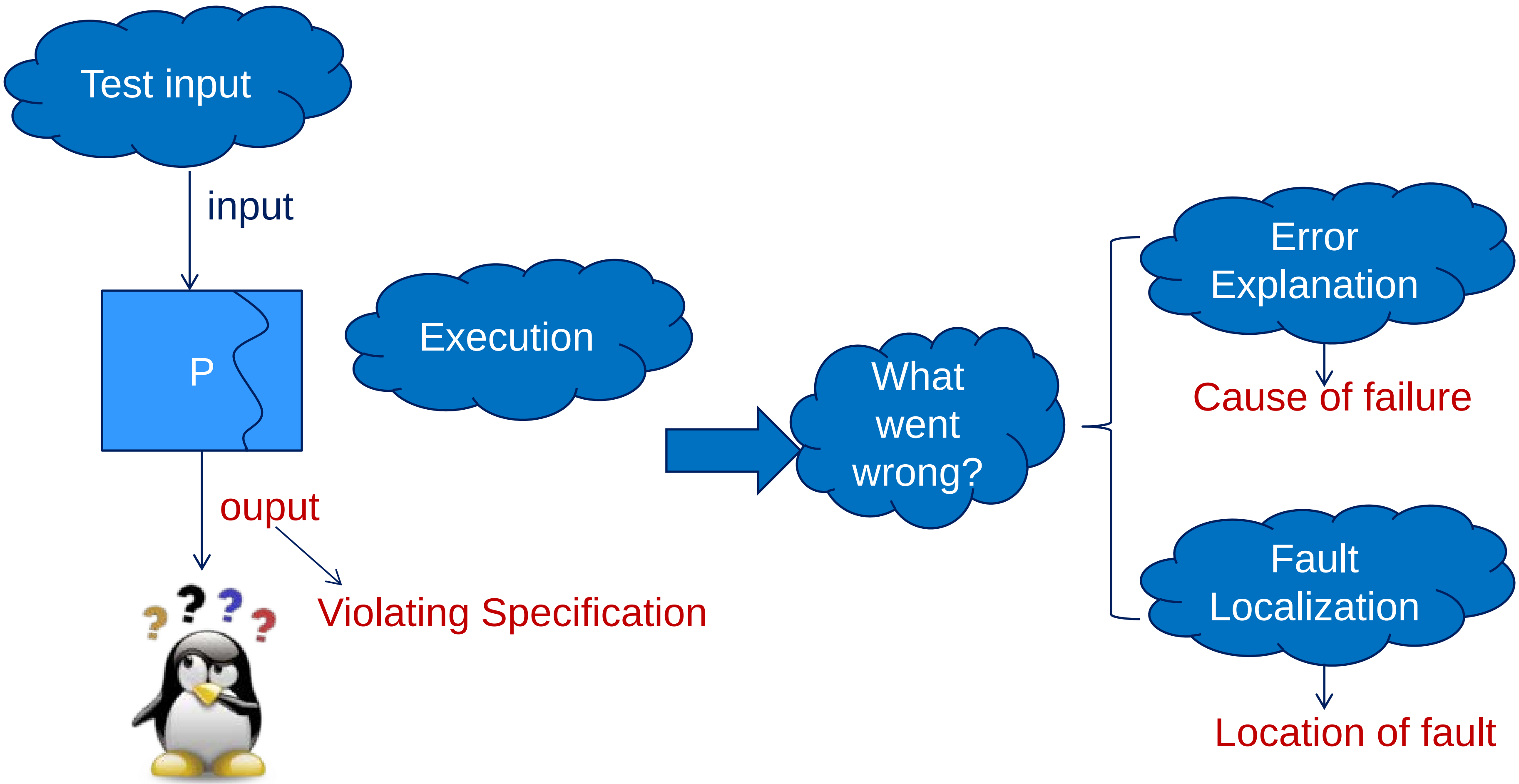
[1]Vienna University of Technology
[2]University of Toronto
[3]New York University

AVM, Austria

Debugging

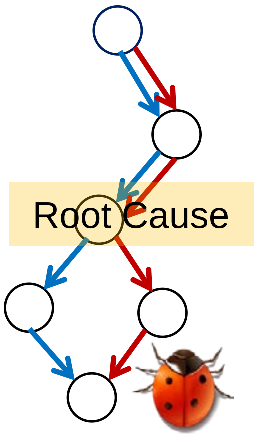

Test input
input
P
Execution
ouput
Violating Specification

Debugging


Test input
input
P
ouput
Violating Specification
Execution
What went wrong?
Error Explanation
Cause of failure
Fault Localization
Location of fault

Automatic Debugging Techniques

- Dynamic analysis:
  - Comparison of failing and passing traces
    - Quality of test suite



- Symbolic execution analysis:
  - Max-SAT
    - Cause clue clauses [PLDI11]
  - Interpolation
    - Error Invariant [FM12]
    - Flow-sensitive Fault Localization[VMCAI13]
    - Hybrid Algorithm [VSSTE14]

Overview of our Method

- A concurrency bug explanation technique:
  - Symbolic execution analysis
  - Interpolation
- A general framework for concurrency bug explanation
  - Not relying on specific bug characteristic
  - No given pattern templates or annotations

Outline

- Notion of Interpolant
- Interpolants for debugging sequential traces
  - Encoding control-dependencies
    - Flow-sensitive slices
- Interpolants for explaining concurrency bugs
  - Encoding:
    - Locks
    - Inter-thread data-dependencies
- Empirical Evaluation

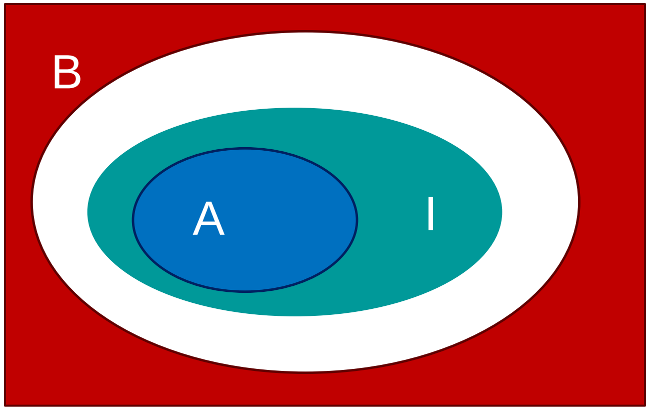
Interpolants

Given: an unsatisfiable conjunction of formulas $A \land B$:

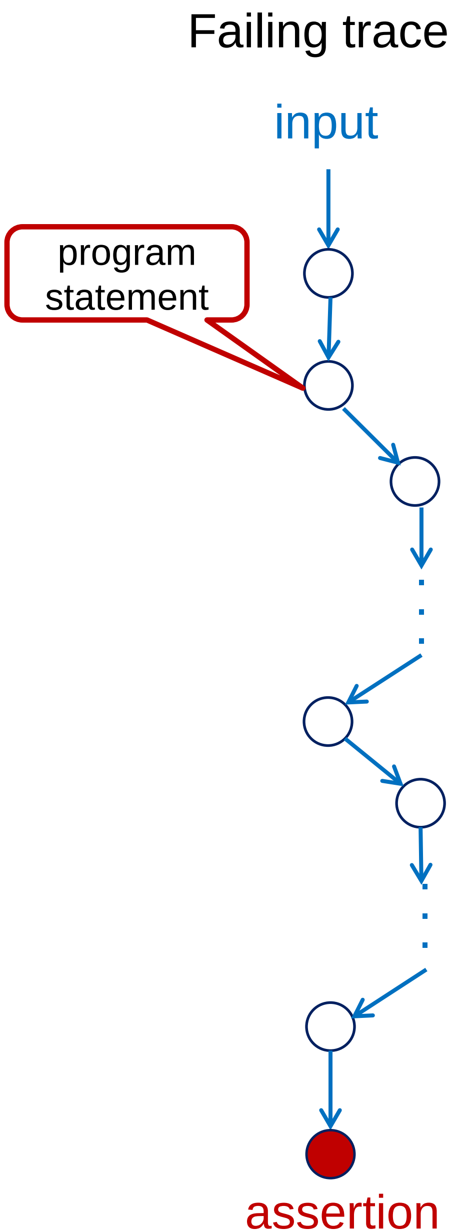
$$A \land B \equiv \text{false}$$

An Interpolant for $A \land B$ is a formula I s.t. :

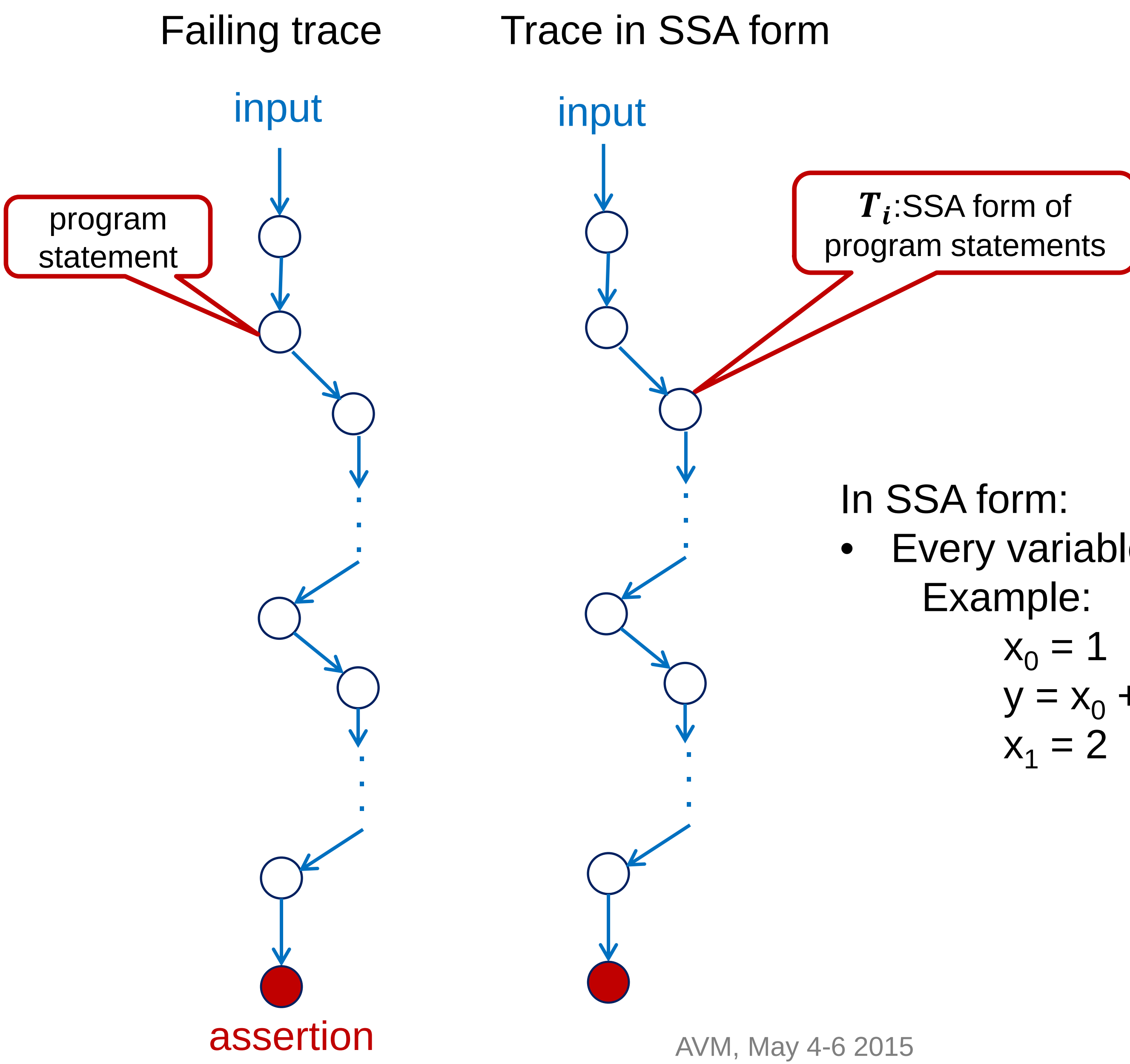
- $A \Rightarrow I$
- $I \land B \equiv \text{false}$
- I is only on common variables of A and B



Trace Formula



Trace Formula



In SSA form:

- Every variable is assigned once:

  Example:

  $x_0 = 1$

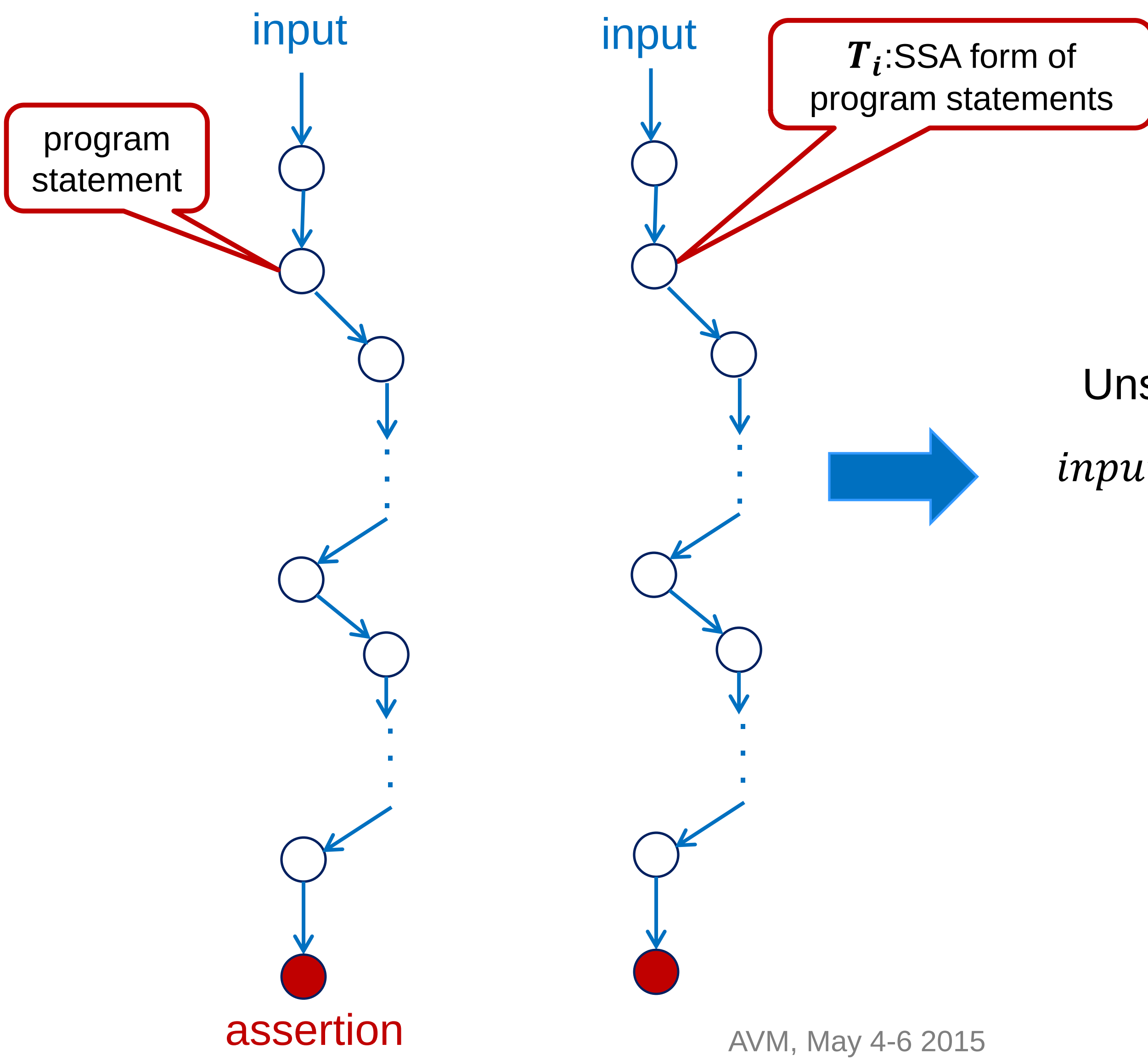
  $y = x_0 + 5$

  $x_1 = 2$

Trace Formula

Failing trace

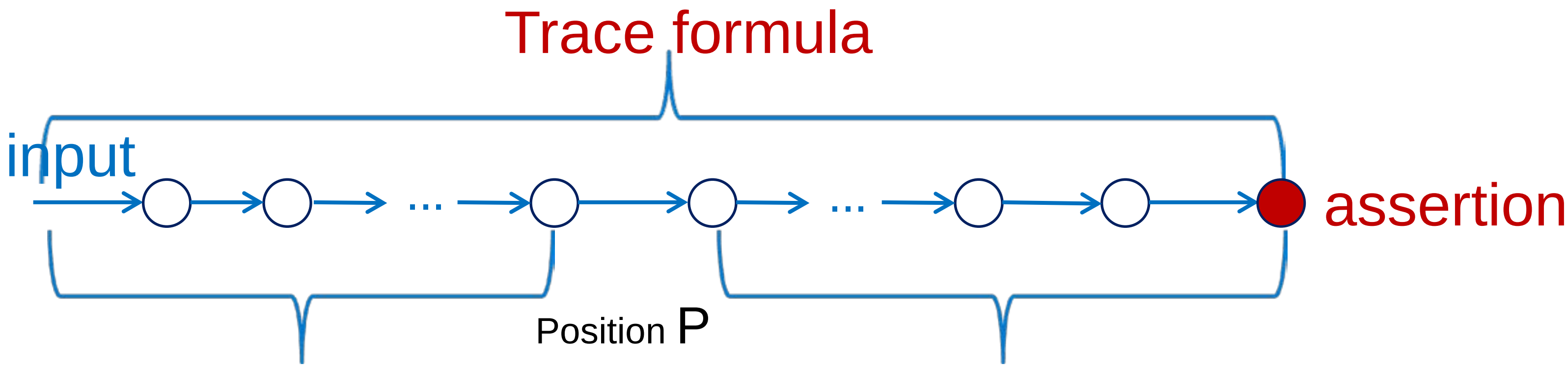
Trace in SSA form



Unsatisfiable trace formula:

$$input \wedge T_1 \wedge T_2 \ldots \wedge T_n \wedge assertion$$

Interpolants for Debugging



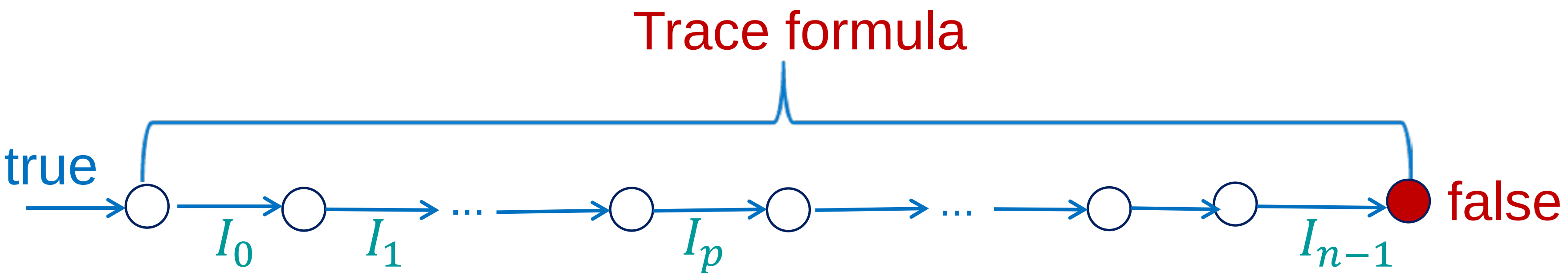
$$X \wedge Y \equiv \text{false}$$

$$X \Longrightarrow \neg Y$$

Interpolant at Position P:

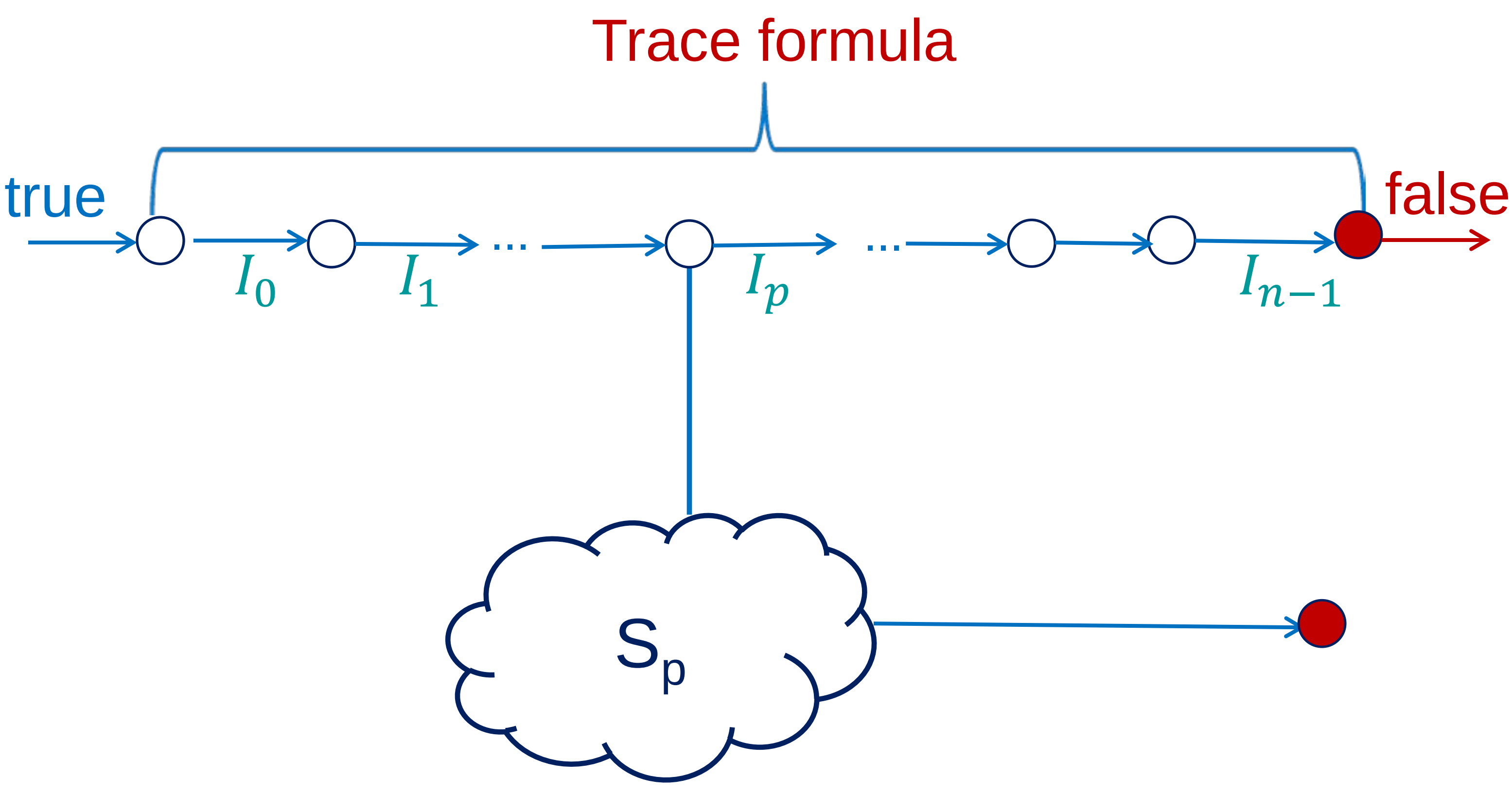
$$X \Longrightarrow I_P \Rightarrow \neg Y$$

Interpolants for Debugging



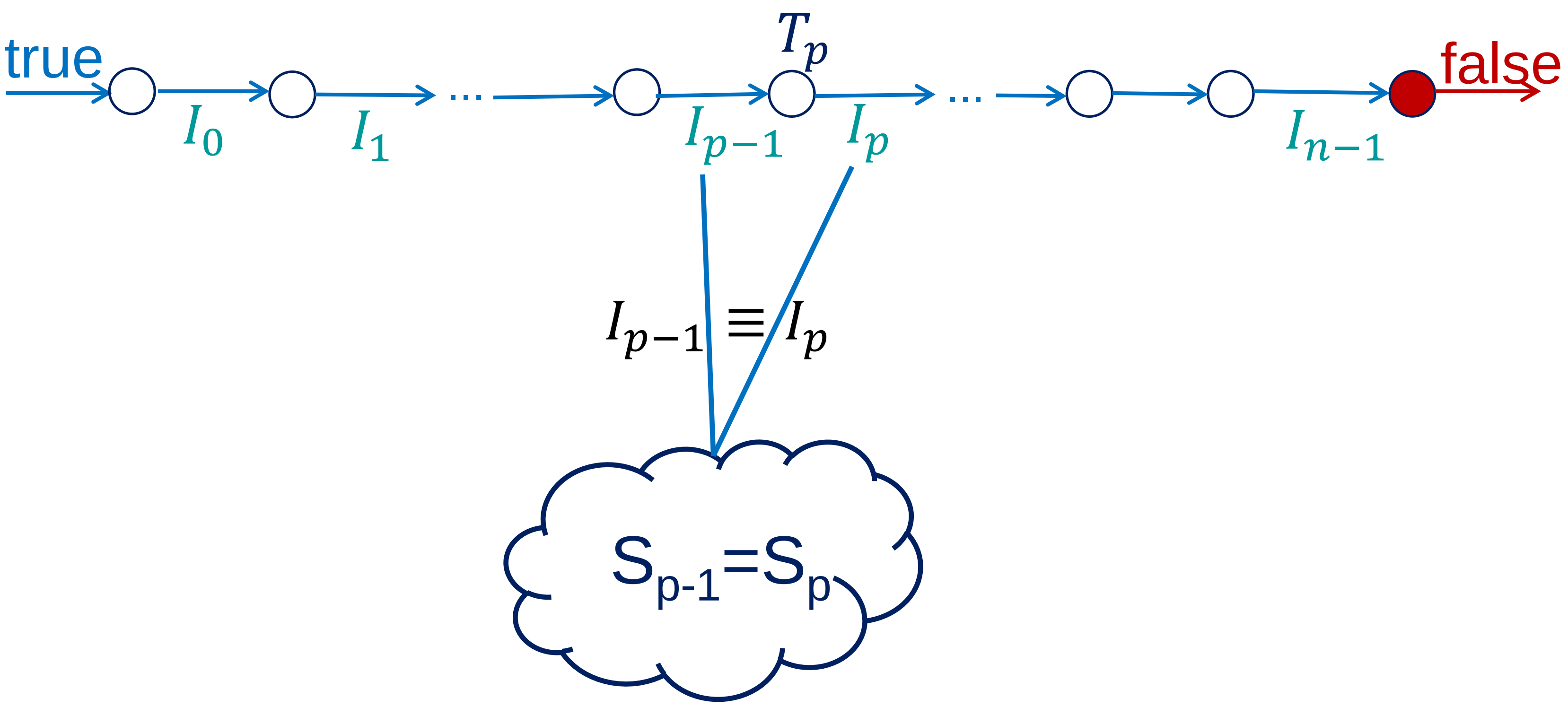
Interpolant Sequence

Interpolants for Debugging

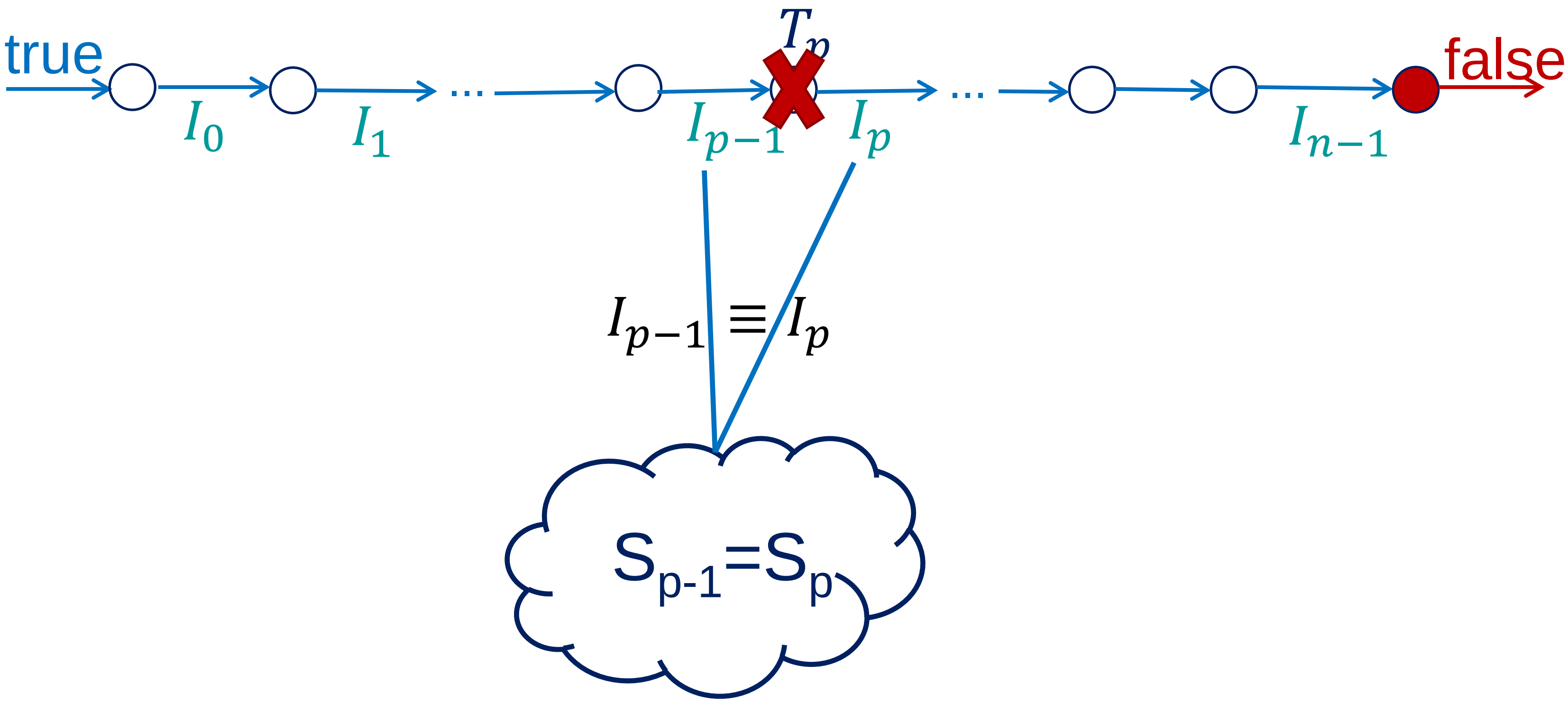


Over-approximaton of reachable states at p

Interpolants for Debugging



Interpolants for Debugging



Interpolants for Debugging

Sample Trace	Trace SSA encoding	Interpolants
		true
1. x=3;	$x_0=3$	
		$x_0=3$
2. y=5;	$y_0=5$	
		$x_0=3$
3. m=y+x;	$m_0=y_0+x_0$	
		$m_0=y_0+3, x_0=3$
4. n=y-x;	$n_0=y_0-x_0$	
		$m_0=n_0+6$
5. y=y+1;	$y_1=y_0+1$	
		$m_0=n_0+6$
6. assert(n>m);	$n_0>m_0$	
		false

- Interpolants
  - contain enough information to understand the failure

Interpolants for Debugging

Sample Trace	Trace SSA encoding	Interpolants	
		true	
1. x=3;	1. $x_0=3$		
		$x_0=3$	Unchanged
2. y=5;	2. $\cancel{y_0=5}$ Irrelevant		
		$x_0=3$	
3. m=y+x;	3. $m_0=y_0+x_0$		
		$m_0=y_0+3, x_0=3$	
4. n=y-x;	4. $n_0=y_0-x_0$		
		$m_0=n_0+6$	Unchanged
5. y=y+1;	5. $\cancel{y_1=y_0+1}$ Irrelevant		
		$m_0=n_0+6$	
6. assert(n>m);	6. $n_0>m_0$		
		false	

Error Explantion

Sample Trace	Error Explantion (Slice)	Error Invariants
		true
1. x=3;	1. x=3	
2. y=5;		x=3
3. m=y+x;	3. m=y+x	
		m=y+3,x=3
4. n=y-x;	4. n=y-x	
5. y=y+1;		m=n+6
6. assert(n>m);	6. assert(n>m)	
		false

Error Explanation:

- Slice: Isolating relevant *statements* for assertion violation
- Error Invariants: Revealing the relevant *variables*

Error Explantion

Sample Trace	Error Explantion (Slice)	Error Invariants
		true
1. x=3;	1. x=3	
2. y=5;		x=3
3. m=y+x;	3. m=y+x	
		m=y+3,x=3
4. n=y-x;	4. n=y-x	
5. y=y+1;		m=n+6
6. assert(n>m);	6. assert(n>m)	
		false

Error Explanation:

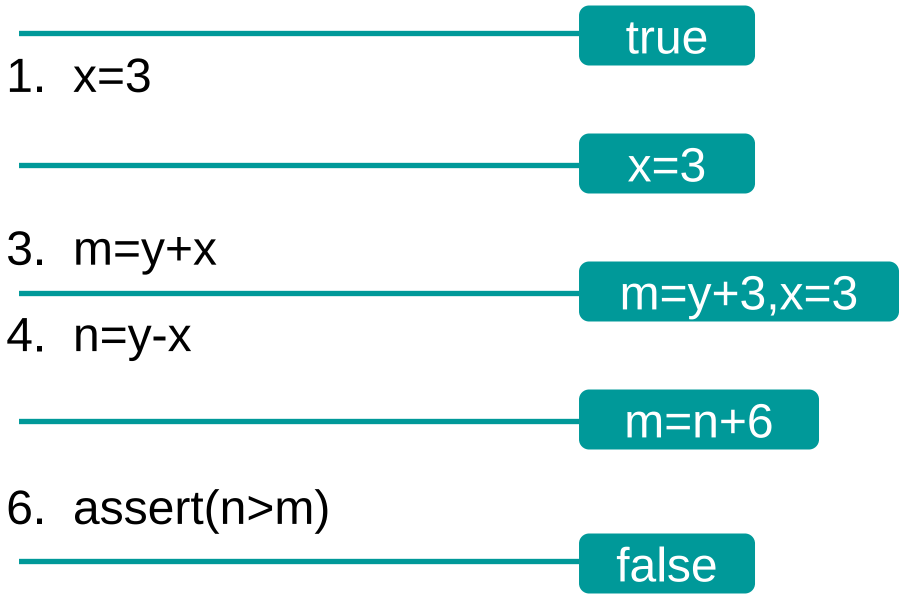
Interpolants are not unique: any formula between WP and SP

- Error Invariants: Revealing the relevant *variables*

Sound Error Explanation Slices

- Soundness of explanation
  - Slice forms an unsatisfiable formula

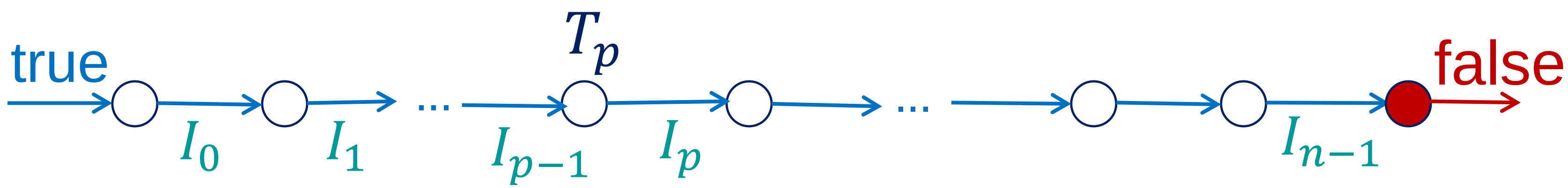
Sound Slice



Sound Error Explanation Slices

- Soundness of explanation
  - Achieved by *Inductive Interpolant Sequence* [VSSTE 2014]

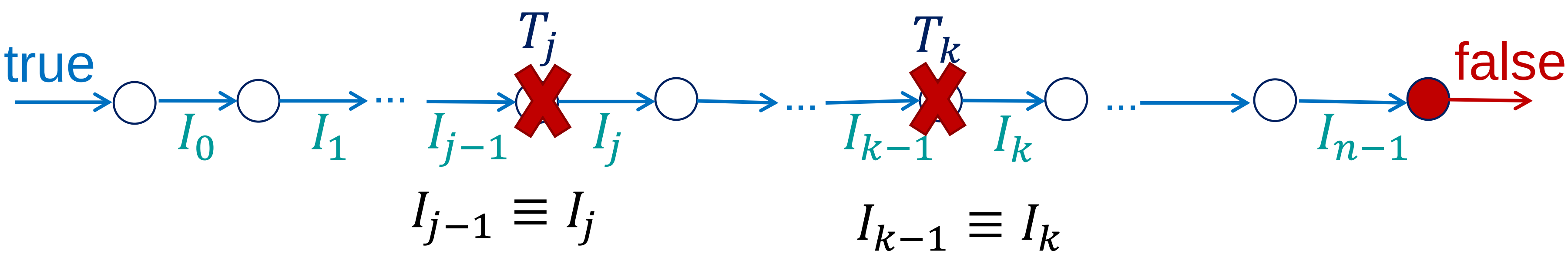
$$I_{p-1} \wedge T_p \Rightarrow I_p$$ Inductive property



Sound Error Explanation Slices

- Soundness of explanation
  - Achieved by *Inductive Interpolant Sequence* [VSSTE 2014]

$$I_{p-1} \wedge T_p \Rightarrow I_p$$ Inductive property



Interpolants for Debugging

- Generating unsatisfiable trace formula
  - by SSA encoding
- Computing inductive interpolants
  - for each position in the trace
- Excluding statements
  - with stationary surrounding interpolants

Encoding Conditions

Sample Code

1. x=1;
2. y=*;
3. if (y < 0)
4.   x=0;
5. assert(x!=0);

Sample Trace

true

1. x_0=1

true

2. y_0=-10

true

3. $(y_0<0) \land x_1=0$

x_1=0

4. $x_1 \neq 0$

false

Slice

4. x=0;

x=0

5. assert(x != 0)

Conditions are required for understanding the failure.

Encoding Conditions

SSA Trace

	true
1. $x_0=1;$	true
2. $y_0=-10;$	true
3. $(y_0<0) \wedge x_1=0;$	$x_1=0$
4. $x_1 \neq 0$	false

1. ~~x=1;~~
2. ~~y=*;~~
3. ~~if (y < 0)~~
4. x=0;
5. assert(x!=0);

Flow-sensitive SSA Trace

	true
1. $x_0=1;$	true
2. $y_0=-10;$	$y_0 <0$
3. $(y_0 <0) \Rightarrow x_1=0;$	$x_1=0$
4. $x_1 \neq 0$	false

1. ~~x=1;~~
2. y=*;
3. if (y < 0)
4. x=0;
5. assert(x!=0);

Flow-sensitive Slices

SSA Trace		Flow-sensitive SSA Trace	
	true		true
1. x_0=1;	true	1. x_0=1;	true
2. y_0=-10;	true	2. y_0=-10;	$y_0 < 0$
3. $(y_0<0) \wedge x_1=0$;	$x_1=0$	3. $(y_0 <0) \Rightarrow x_1=0$;	$x_1=0$
4. $x_1 \neq 0$	false	4. $x_1 \neq 0$	false

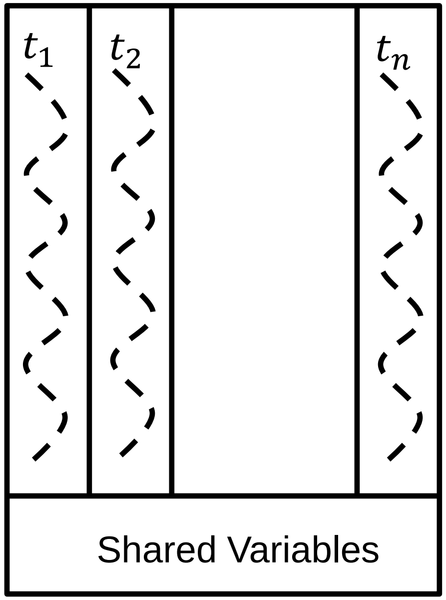
~~1. x=1;~~ ~~1. x=1;~~

- Encoding of control-dependency:
  - Conditions are encoded as implications in SSA traces:

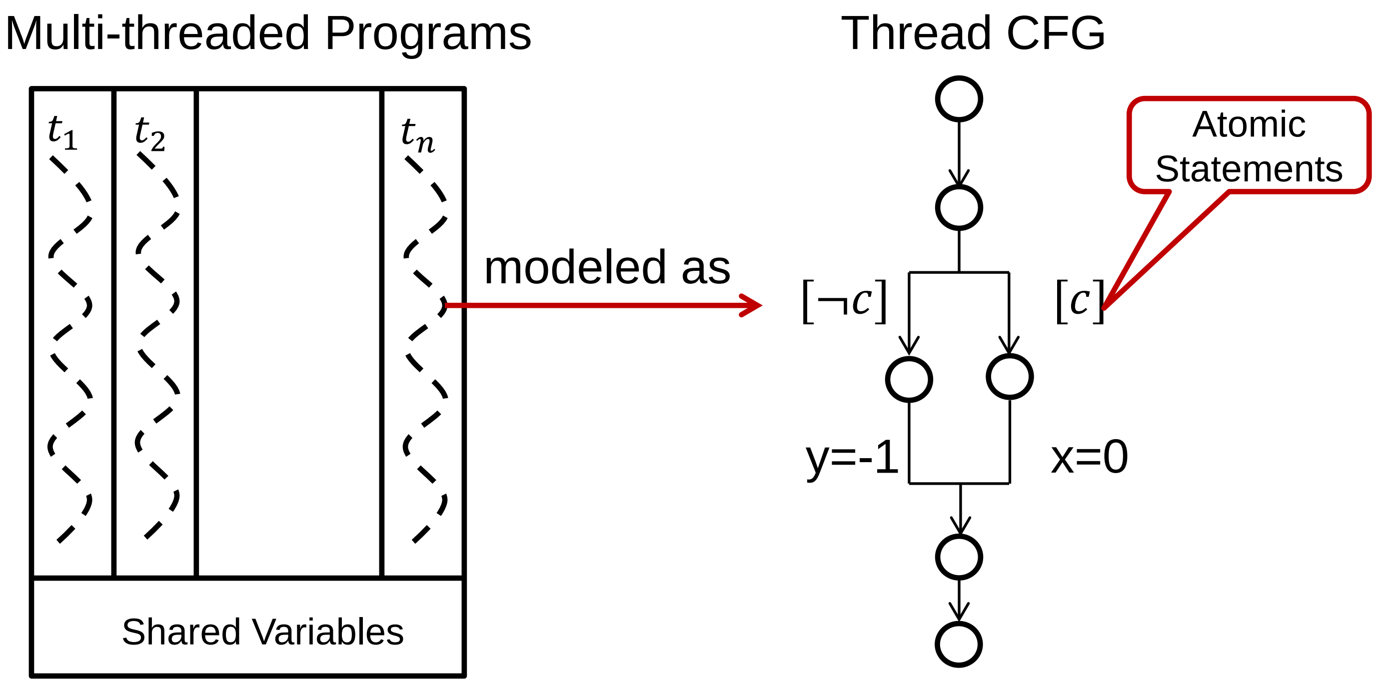
$$\left(\bigwedge_{c \in conds} c\right) \Rightarrow x = e$$

Model of Concurrent Traces

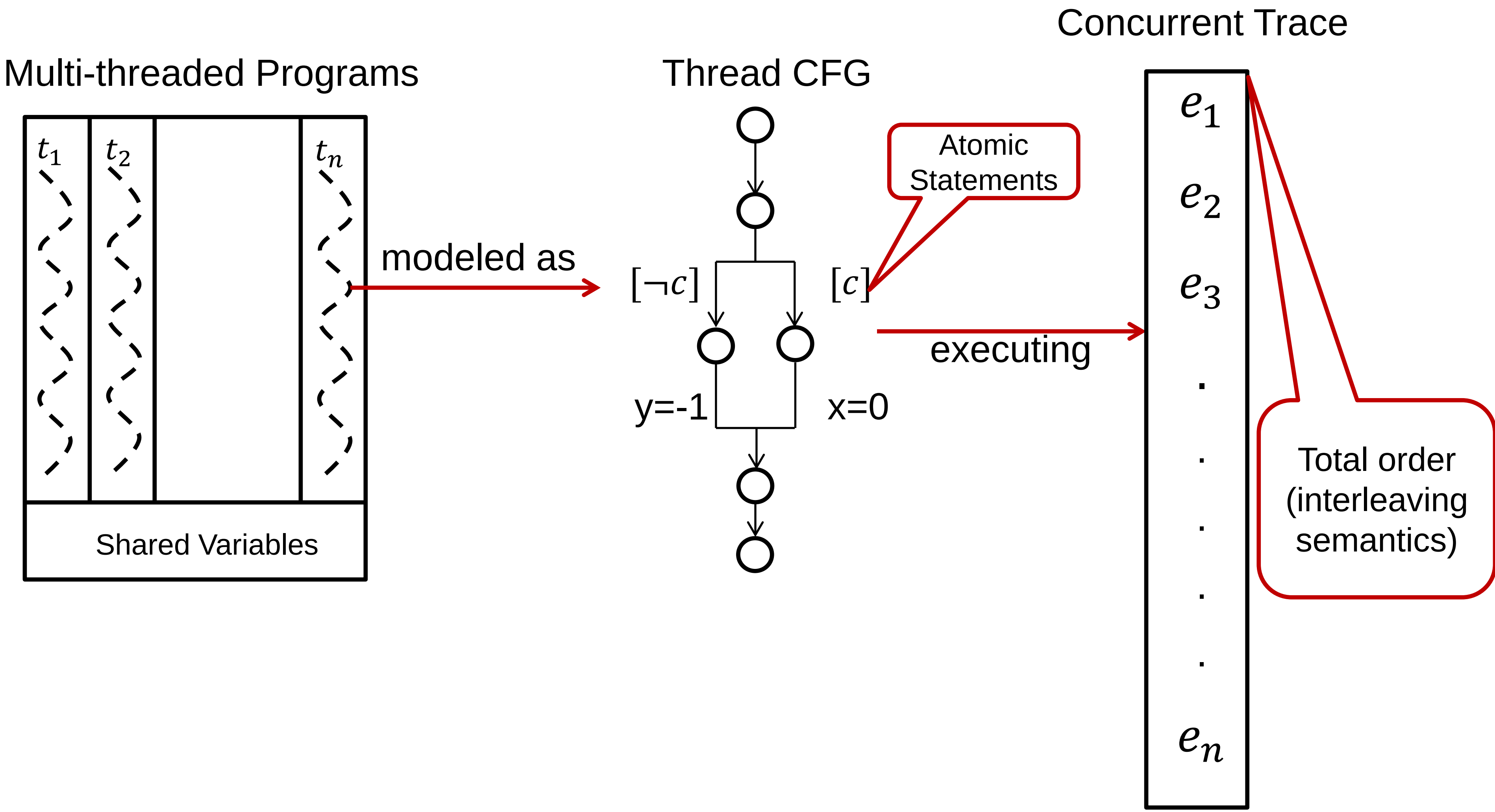
Multi-threaded Programs



Model of Concurrent Traces



Model of Concurrent Traces



Concurrent Trace Formula

- SSA encoding of variables
- Encoding control-dependency as implication

- Modeling Locks as:
  - Atomic guarded assignments:

    acquire ℓ: $(\ell = 0) \rhd \ell := tid$

    release ℓ: $(\ell = tid) \rhd \ell := 0$
  - Encoding locks as implications (similar to control-dependency)

Interpolants for Debugging Concurrent Bugs

Main Thread

balance := 40
withdrawal := 20
deposit := 10
...

Thread 1

acquire ℓ
bal = **balance**
release ℓ

bal = bal + *deposit*

acquire ℓ

balance = bal
release ℓ

Thread 2

acquire ℓ
bal = **balance**
release ℓ

bal = bal - *withdrawal*

acquire ℓ

balance = bal
release ℓ

assert(**balance** = 40 – 20 + 10)

Interpolants for Debugging Concurrent Bugs

Main Thread

(1)

balance := 40
withdrawal := 20
deposit := 10
...

Thread 1

acquire ℓ
bal = **balance**
release ℓ

bal = bal + *deposit*

acquire ℓ
balance = bal
release ℓ

Thread 2

acquire ℓ
bal = **balance**
release ℓ

bal = bal - *withdrawal*

acquire ℓ
balance = bal
release ℓ

assert(**balance** = 40 – 20 + 10)

Interpolants for Debugging Concurrent Bugs

Main Thread

(1)

balance := 40
withdrawal := 20
deposit := 10
...

Thread 1

acquire ℓ
bal = **balance**
release ℓ

bal = bal + *deposit*

acquire ℓ
balance = bal
release ℓ

Thread 2

(2)

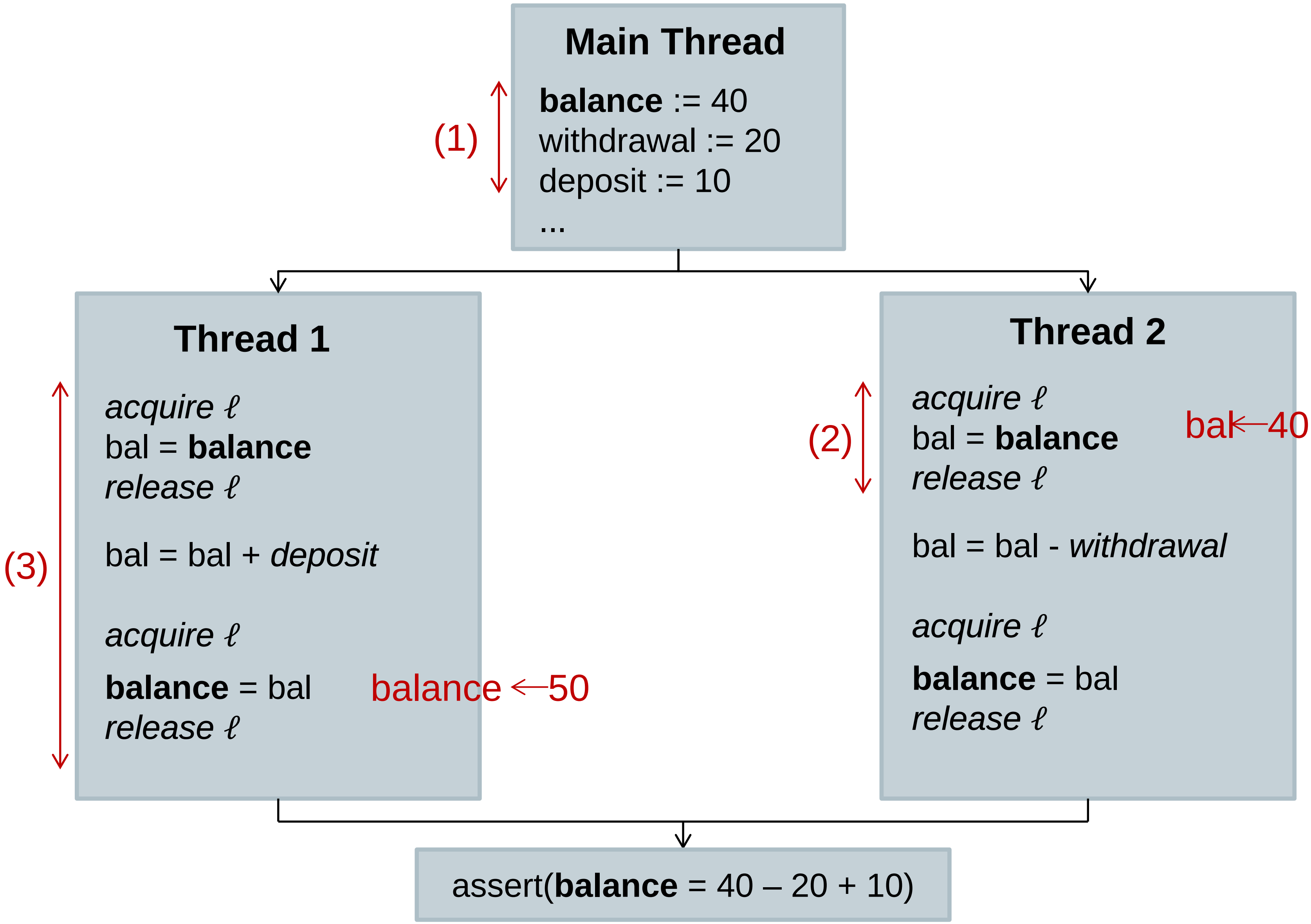
acquire ℓ
bal = **balance**   bal $\leftarrow$ 40
release ℓ

bal = bal - *withdrawal*

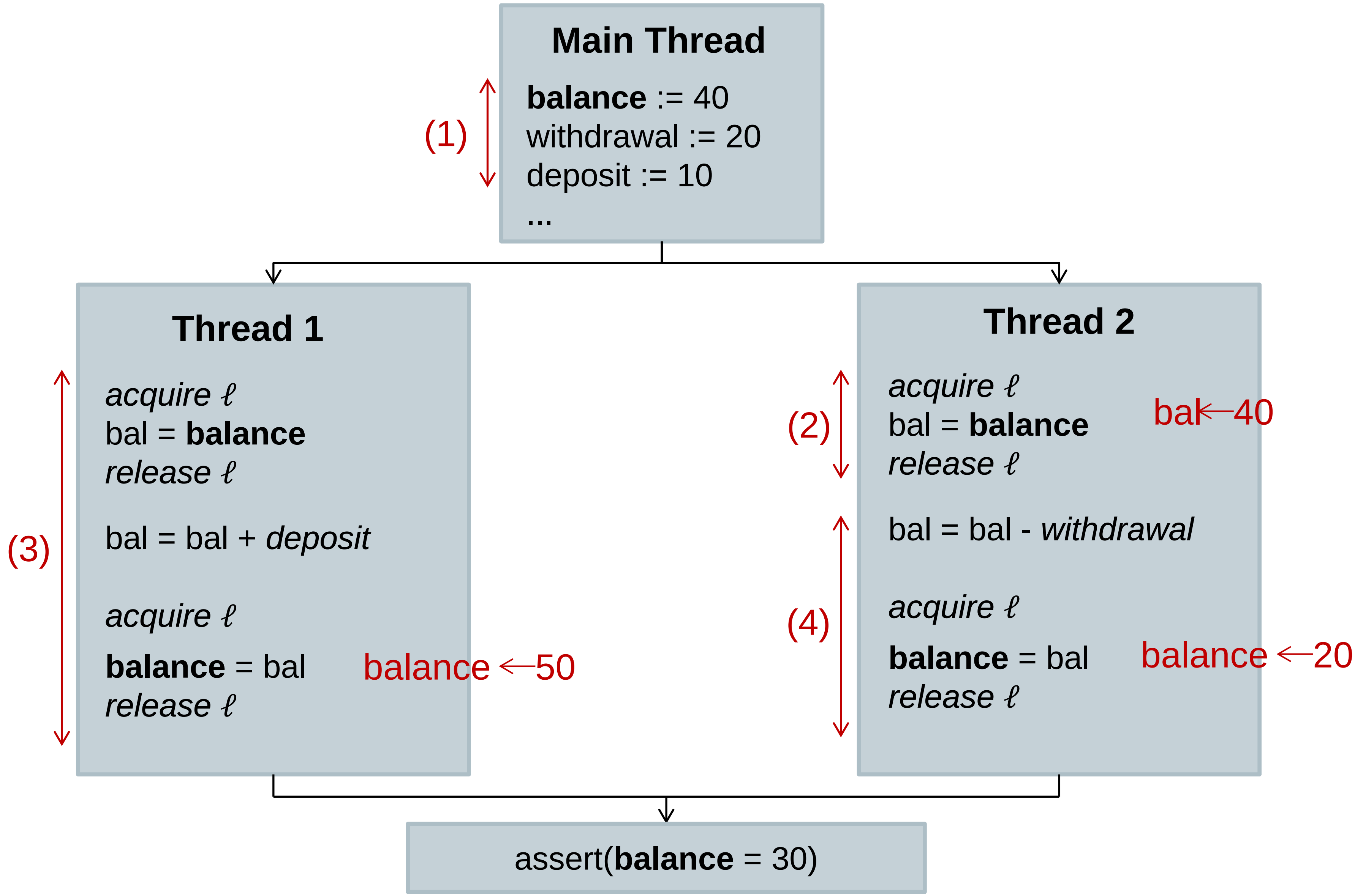
acquire ℓ
balance = bal
release ℓ

assert(**balance** = 40 – 20 + 10)

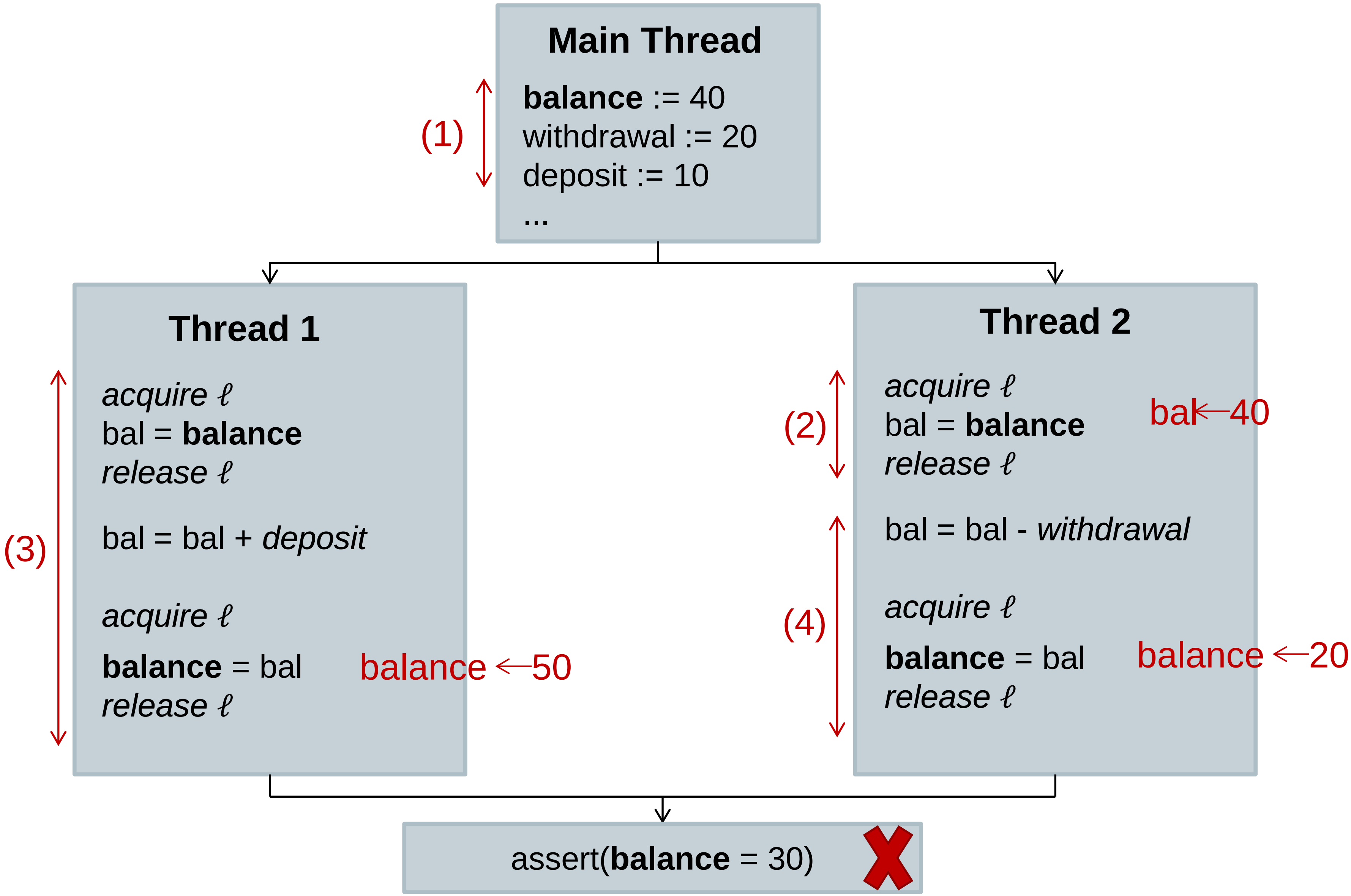
Interpolants for Debugging Concurrent Bugs



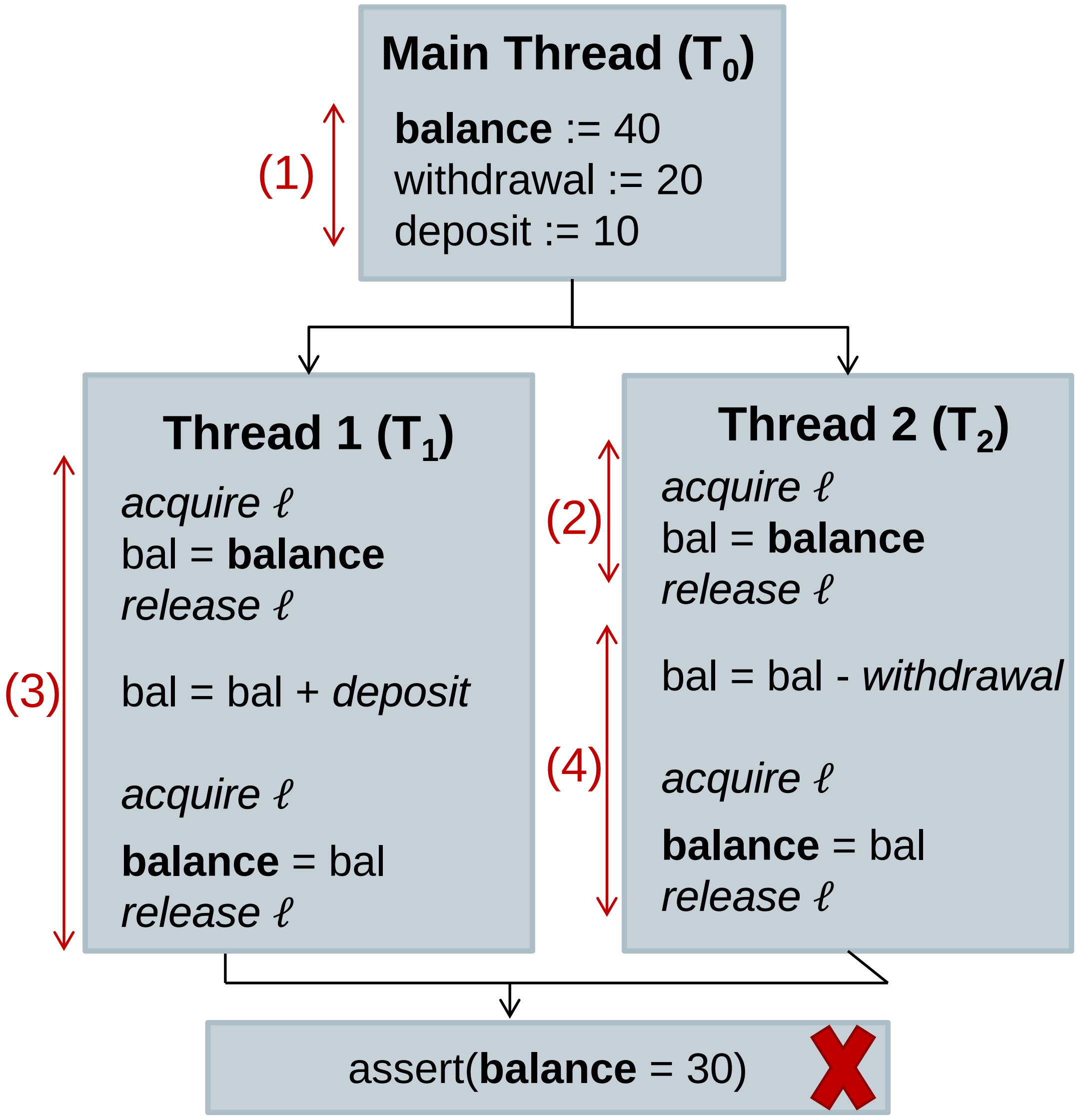
Interpolants for Debugging Concurrent Bugs



Interpolants for Debugging Concurrent Bugs



Interpolants for Debugging Concurrent Bugs



Flow-insensitive Slice

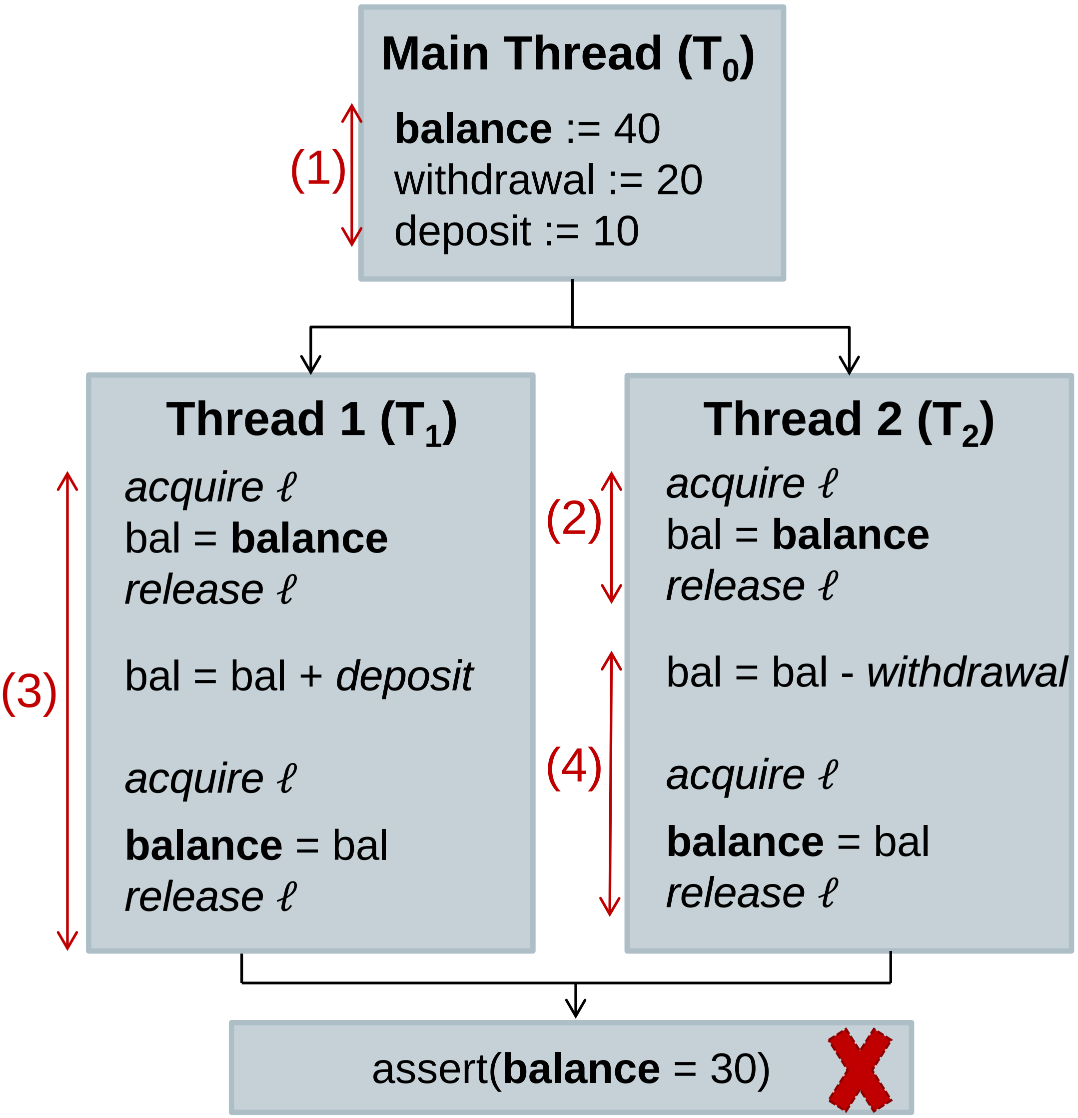
T_0 : **balance** := 40
T_0 : withdrawal := 20

T_2 : bal = **balance**
T_2 : bal = bal - *withdrawal*
T_2 : **balance** = bal

T_0 : assert(**balance** = 30)

Ignoring Thread 1 altogether

Interpolants for Debugging Concurrent Bugs



Flow-sensitive Slice

T_0 : **balance** := 40
T_0 : withdrawal := 20
T_2 : *acquire ℓ*
T_2 : bal = **balance**
T_2 : *release ℓ*
T_1 : *acquire ℓ*
T_1 : *release ℓ*
T_1 : *acquire ℓ*
T_1 : *release ℓ*
T_2 : bal = bal - *withdrawal*
T_2 : *acquire ℓ*
T_2 : **balance** = bal
T_2 : *release ℓ*
T_0 : assert(**balance** = 30)

T_1 : **balance** = bal is missing

Data dependencies

- Data dependency:
  - Flow of data between statements
- Types of data dependency (in general)
  - Read-after-write

    a = x;

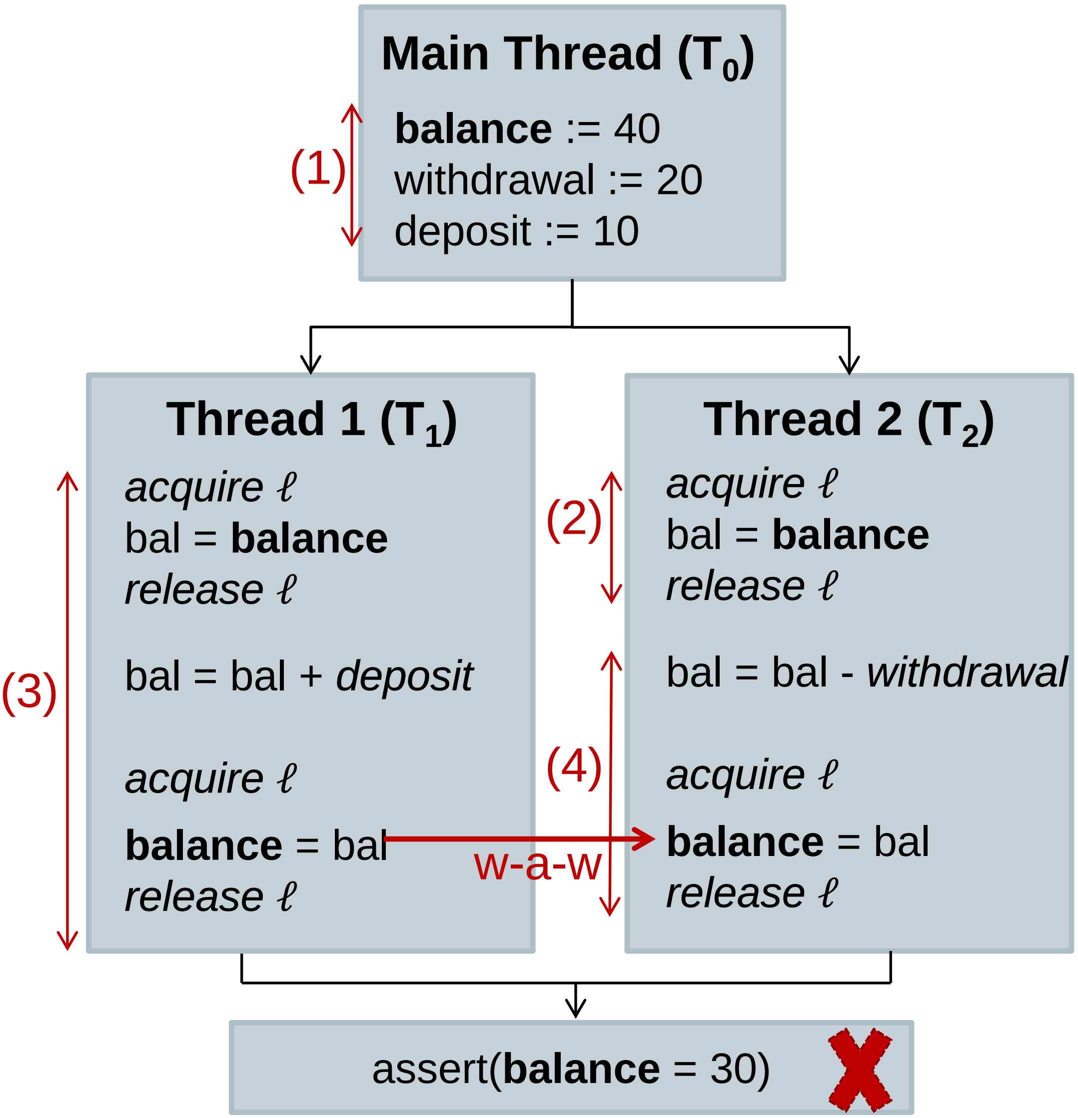
    y = **a** + 10;
  - Write-after-read

    x = **a**;

    a = y + 10;
  - Write-after-write
    - **a** = x + 10;
    - **a** = y + 10;

- Inter-thread data dependencies (in multi threaded programs)
  - Being able to indicate
    - conflicting accesses or hazards

Interpolants for Debugging Concurrent Bugs



Flow-sensitive Slice

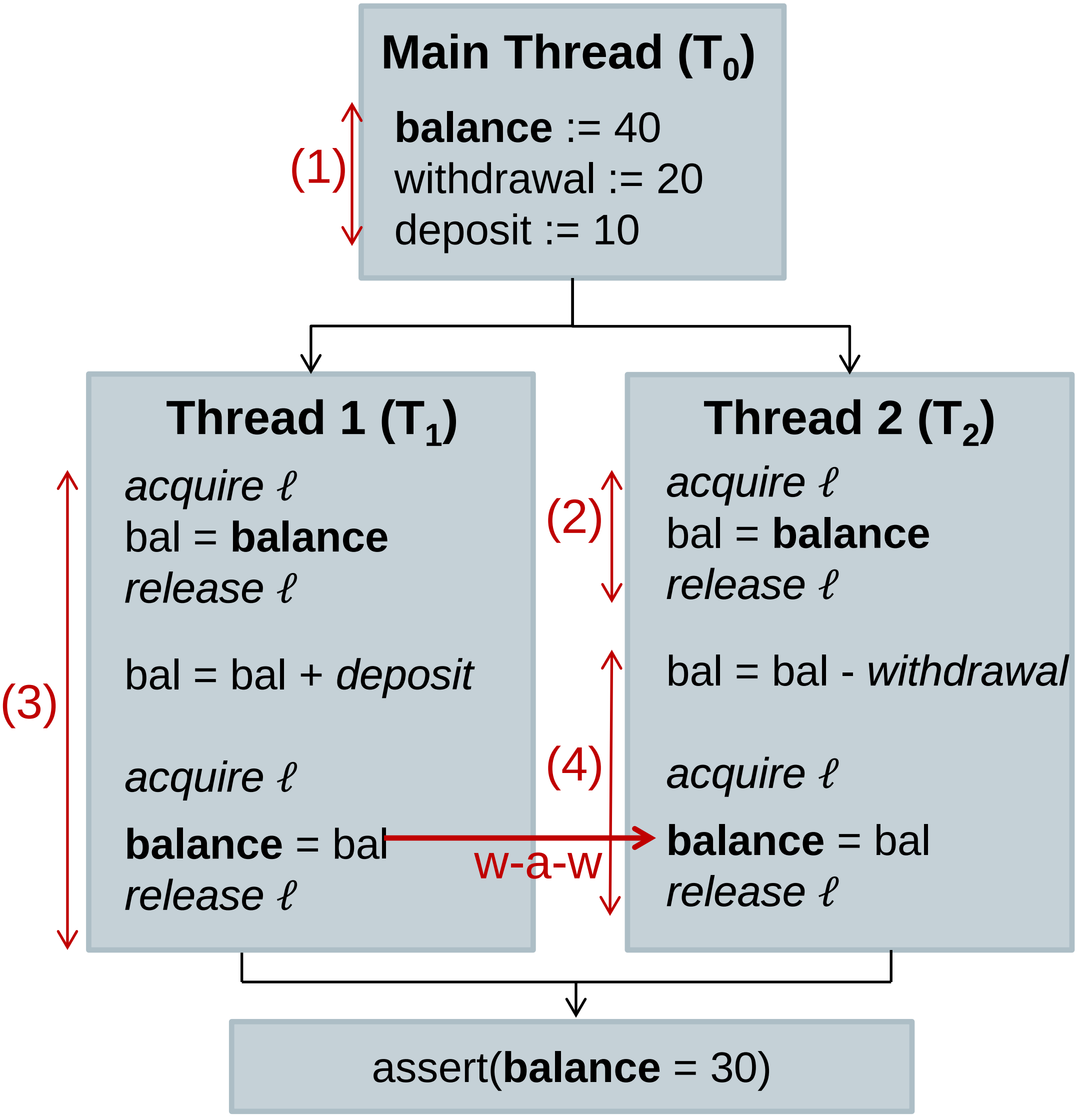
T_0 : **balance** := 40
T_0 : withdrawal := 20
T_2 : *acquire ℓ*
T_2 : bal = **balance**
T_2 : *release ℓ*
T_1 : *acquire ℓ*
T_1 : *release ℓ*
T_1 : *acquire ℓ*
T_1 : *release ℓ*
T_2 : bal = bal - *withdrawal*
T_2 : *acquire ℓ*
T_2 : **balance** = bal
T_2 : *release ℓ*
T_0 : assert(**balance** = 30)

Hazard-sensitive Slices

- Encoding inter-thread data dependencies:
  - as implication (using auxiliary variables)

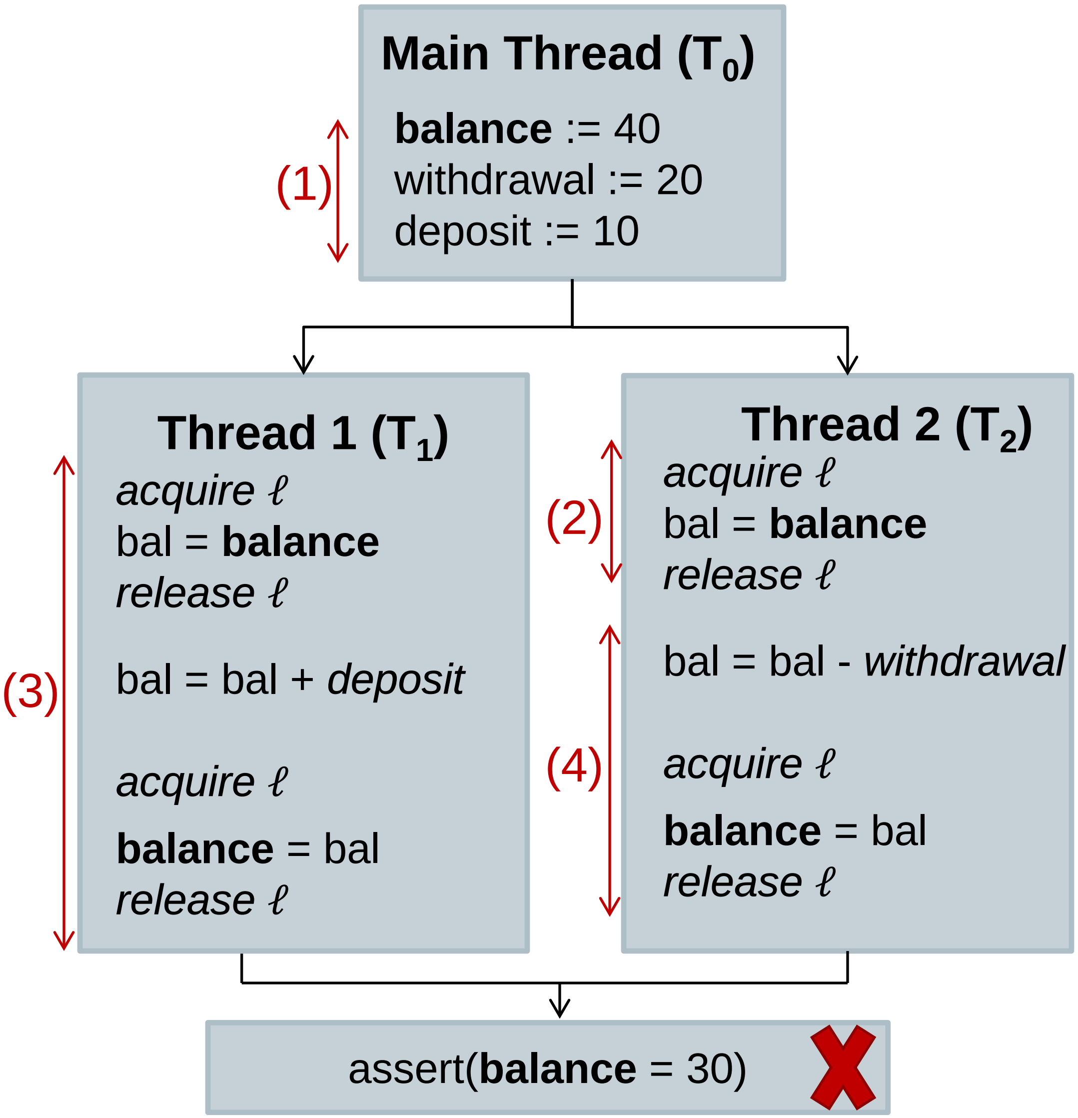
- The resulting slice:
  - Hazard-sensitive slice

Hazard-sensitive Slice



T1: $v \wedge balance = bal$
T2: $v \Rightarrow balance = bal$

Interpolants for Debugging Concurrent Bugs



Hazard-sensitive Slice

T_0 : **balance** := 40
T_0 : withdrawal := 20

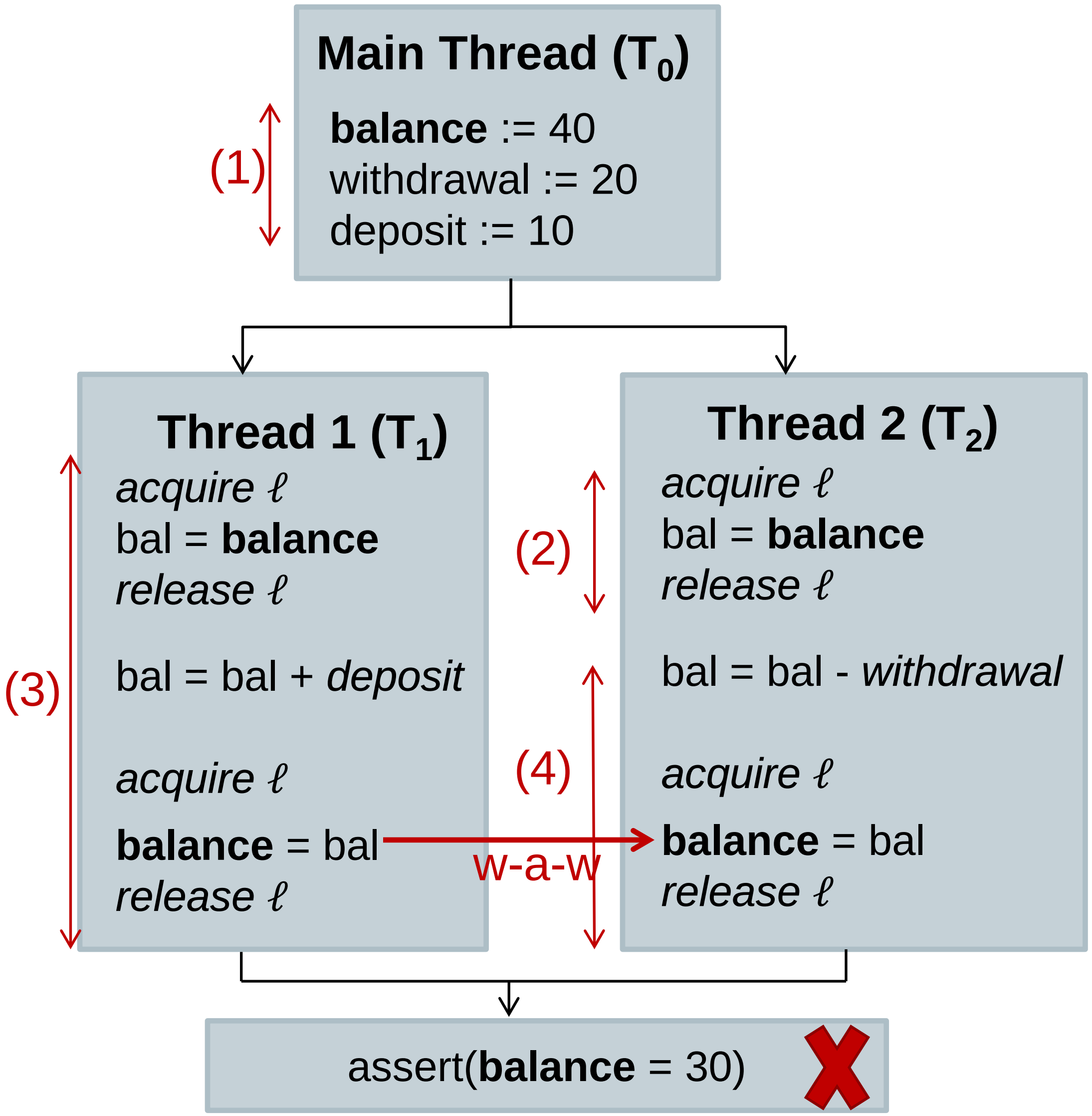
T_2 : bal = **balance**

T_1 : **balance** = bal

T_2 : bal = bal – *withdrawal*
T_2 : **balance** = bal

T_0 : assert(**balance** = 30)

Interpolants for Debugging Concurrent Bugs



Hazard-sensitive Slice

T_0 : **balance** := 40

$\{balance \le 40\}$

T_0 : withdrawal := 20

$\{balance \le withdrawal + 20\}$

T_2 : bal = **balance**

$\{bal \le withdrawal + 20\}$

T_1 : **balance** = bal

$\{bal \le withdrawal + 20\}$

w-a-w

T_2 : bal = bal – *withdrawal*

$\{bal \le 20\}$

T_2 : **balance** = bal

$\{balance \le 20\}$

T_0 : assert(**balance** = 30)

Fine-Tuning Explanations

- Adding different levels of detail to the explanations
  - Encoding:
    - control- and inter-thread data-dependency (fs+hs)
    - control-dependency (fs)
    - inter-thread data-dependency (hs)
    - no dependency (∅)
  - Leading to different reductions in number of:
    - variables
    - statements

Fine-Tuning Explanations

∅

T_0 : **balance** := 40
T_0 : withdrawal := 20
T_2 : bal = **balance**
T_2 : bal = bal - *withdrawal*
T_2 : **balance** = bal
T_0 : assert(**balance** = 30)

fs

T_0 : **balance** := 40
T_0 : withdrawal := 20
T_2 : *acquire* ℓ
T_2 : bal = **balance**
T_2 : *release* ℓ
T_1 : *acquire* ℓ
T_1 : *release* ℓ
T_1 : *acquire* ℓ
T_1 : *release* ℓ
T_2 : bal = bal - *withdrawal*
T_2 : *acquire* ℓ
T_2 : **balance** = bal
T_2 : *release* ℓ
T_0 : assert(**balance** = 30)

hs

T_0 : **balance** := 40
T_0 : withdrawal := 20

T_2 : bal = **balance**

T_1 : **balance** = bal

T_2 : bal = bal – *withdrawal*
T_2 : **balance** = bal

T0 : assert(**balance** = 30)

Empirical Evaluation

Quality + Quantity results:

Program	Concurrency bug	Number of traces	Type of slice	Avg. reduction of statements(%)	Avg. reduction of variables
Lock free pool	Linearizability problem	8	fs	61%	34%
			fs+hs	60%	34%
Bank account	Atomicity violation	5	fs+hs	46%	23%
			hs	88%	33%

Conclusion

- A general framework for concurrency bug explantion
  - Interpolation
  - Symbolic execution analysis
    - Encoding of :
      - Control-dependency
      - Inter-thread data-dependency
  - Implementation
    - Interpolant computation
      - VERMEER [ICSE15]
    - Tracing failing concurrent traces
      - ConCrest [FSE13]


Thank you!