Algorithmic Verification of Concurrent Programs

Shaz Qadeer
Microsoft Research

Reliable concurrent software?

- Correctness problem
  - does program behave correctly for *all inputs* and *all interleavings*?
- Bugs due to concurrency are insidious
  - non-deterministic, timing dependent
  - data corruption, crashes
  - difficult to detect, reproduce, eliminate

Demo

- Debugging concurrent programs is hard

Program verification

- Program verification is undecidable
  - even for sequential programs
- Concurrency does not make the worst-case complexity any worse
- Why is verification of concurrent programs considered more difficult?

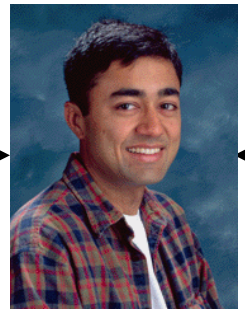
Undecidable problem!

P satisfies S

Assertions: Provide contracts to decompose problem into a collection of decidable problems

- pre-condition and post-condition for each procedure
- loop invariant for each loop

P satisfies S



Abstractions: Provide an abstraction of the program for which verification is decidable

- Finite-state systems
  - finite automata
- Infinite-state systems
  - pushdown automata, counter automata, timed automata, Petri nets

Assertions: an example

```
ensures \result >= 0 ==> a[\result] == n
ensures \result < 0   ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n

int find(int a[ ], int n) {
    int i = 0;


    while (i < a.length) {
        if (a[i] == n) return i;
        i++;
    }

    return -1;
}
```

Assertions: an example

```
ensures \result >= 0 ==> a[\result] == n
ensures \result < 0   ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n

int find(int a[ ], int n) {
    int i = 0;
    loop_invariant 0 <= i && i <= a.length
    loop_invariant forall j:int :: 0 <= j && j < i ==> a[j] != n
    while (i < a.length) {
        if (a[i] == n) return i;
        i++;
    }

    return -1;
}
```

```
{true}
i = 0
{0 <= i && i <= a.length  && forall j:int :: 0 <= j && j < i ==> a[j] != n}
```

```
{0 <= i && i <= a.length  && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume i < a.length;
assume !(a[i] == n);
i++;
{0 <= i && i <= a.length  && forall j:int :: 0 <= j && j < i ==> a[j] != n}
```

```
{0 <= i && i <= a.length  && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume i < a.length;
assume (a[i] == n);
{(i >= 0 ==> a[i] == n) && (i < 0   ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n)}
```

```
{0 <= i && i <= a.length  && forall j:int :: 0 <= j && j < i ==> a[j] != n}
assume !(i < a.length);
{(-1 >= 0 ==> a[-1] == n) && (-1 < 0   ==> forall j:int :: 0 <= j && j < a.length ==> a[j] != n)}
```

Abstractions: an example

```
requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {

        Acquire(m);
        while (x != y) {

            Release(m);
            Sleep(1);

            Acquire(m);
        }
        y = f (x);

        Release(m);
    }
}
```

```
requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {
        assert m == UNLOCK;
        m = LOCK;
        while (x != y) {
            assert m == LOCK;
            m = UNLOCK;
            Sleep(1);
            assert m == UNLOCK;
            m = LOCK;
        }
        y = f (x);
        assert m == LOCK;
        m = UNLOCK;
    }
}
```

Abstractions: an example

```
requires m == UNLOCK
void compute(int n) {
    for (int i = 0; i < n; i++) {
        assert m == UNLOCK;
        m = LOCK;
        while (x != y) {
            assert m == LOCK;
            m = UNLOCK;
            Sleep(1);
            assert m == UNLOCK;
            m = LOCK;
        }
        y = f (x);
        assert m == LOCK;
        m = UNLOCK;
    }
}
```

```
requires m == UNLOCK
void compute( ) {
    for ( ; * ; ) {
        assert m == UNLOCK;
        m = LOCK;
        while (*) {
            assert m == LOCK;
            m = UNLOCK;

            assert m == UNLOCK;
            m = LOCK;
        }

        assert m == LOCK;
        m = UNLOCK;
    }
}
```

Interference

pre x = 0;

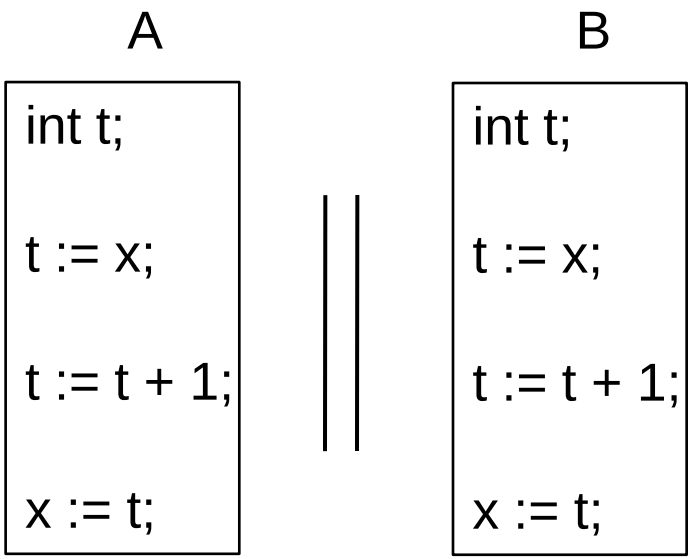
```
int t;

t := x;

t := t + 1;

x := t;
```

post x = 1;

Correct

Interference

pre x = 0;



post x = 2;

Incorrect!

Interference

pre x = 0;

A	B
int t;	int t;
acquire(l);	acquire(l);
t := x;	t := x;
t := t + 1;	t := t + 1;
x := t;	x := t;
release(l);	release(l);

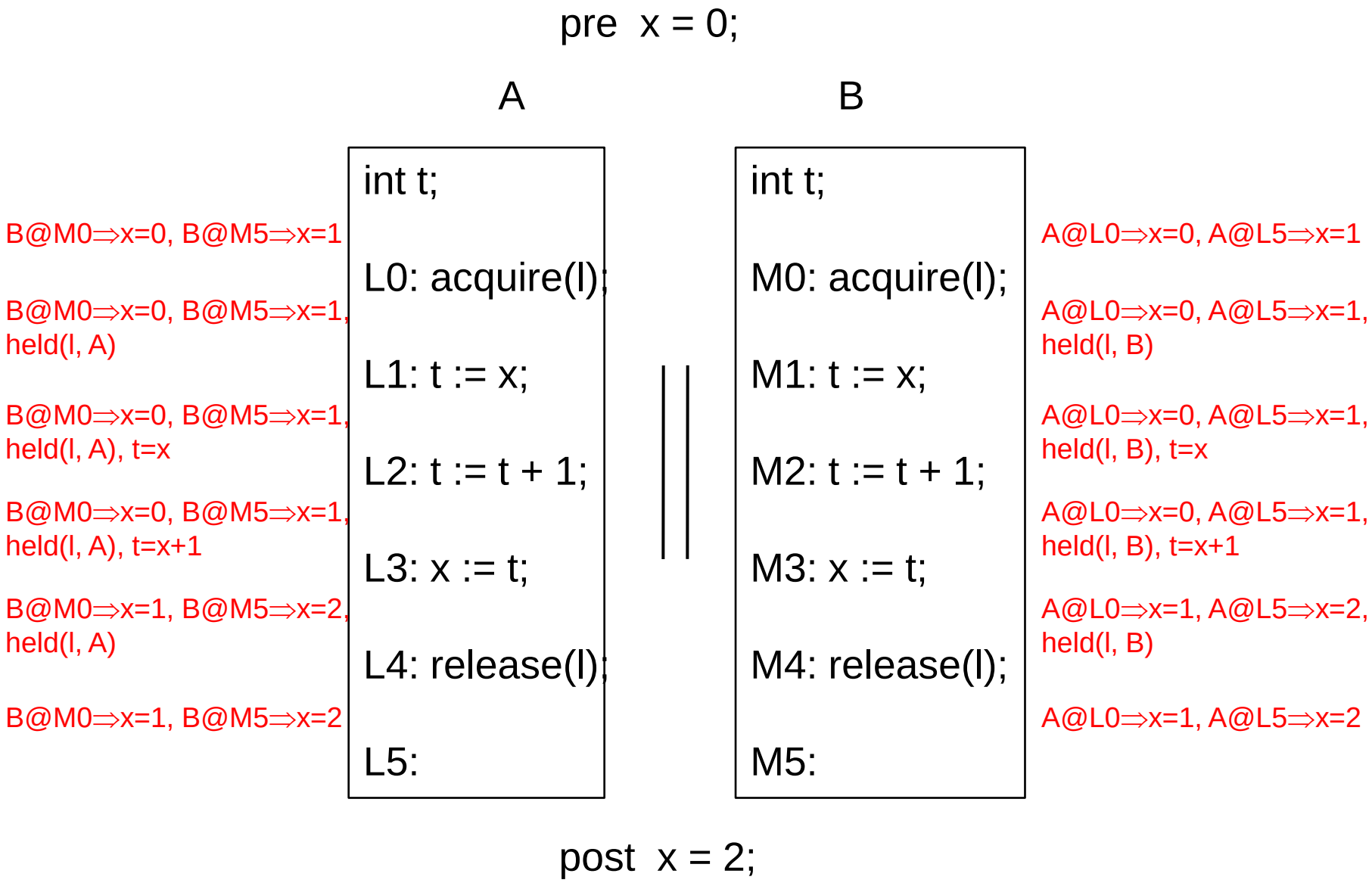
post x = 2;

Correct!

Compositional verification using assertions

Invariants

- Program: a statement S_p for each control location p
- Assertions: a predicate φ_p for each control location p
- Sequential correctness
  - If p is a control location and q is a successor of p, then $\{\varphi_p\}\ S_p\ \{\varphi_q\}$ is valid
- Non-interference
  - If p and q are control locations in different threads, then $\{\varphi_p \wedge \varphi_q\}\ S_q\ \{\varphi_p\}$ is valid


pre x = 0;
A
B
int t;
L0: acquire(l);
L1: t := x;
L2: t := t + 1;
L3: x := t;
L4: release(l);
L5:
int t;
M0: acquire(l);
M1: t := x;
M2: t := t + 1;
M3: x := t;
M4: release(l);
M5:
B@M0⇒x=0, B@M5⇒x=1
B@M0⇒x=0, B@M5⇒x=1, held(l, A)
B@M0⇒x=0, B@M5⇒x=1, held(l, A), t=x
B@M0⇒x=0, B@M5⇒x=1, held(l, A), t=x+1
B@M0⇒x=1, B@M5⇒x=2, held(l, A)
B@M0⇒x=1, B@M5⇒x=2
A@L0⇒x=0, A@L5⇒x=1
A@L0⇒x=0, A@L5⇒x=1, held(l, B)
A@L0⇒x=0, A@L5⇒x=1, held(l, B), t=x
A@L0⇒x=0, A@L5⇒x=1, held(l, B), t=x+1
A@L0⇒x=1, A@L5⇒x=2, held(l, B)
A@L0⇒x=1, A@L5⇒x=2
post x = 2;

Two other checks

- precondition $\Rightarrow (\varphi_{L0} \wedge \varphi_{M0})$

$$(A@L0 \wedge B@M0 \wedge x == 0) \Rightarrow \left[\begin{array}{l}(B@M0 \Rightarrow x=0) \wedge (B@M5 \Rightarrow x=1) \wedge \\ (A@L0 \Rightarrow x=0) \wedge (A@L5 \Rightarrow x=1)\end{array}\right]$$

- $(\varphi_{L0} \wedge \varphi_{M0}) \Rightarrow$ postcondition

$$\left[\begin{array}{l}A@L5 \wedge B@M5 \wedge \\ (B@M0 \Rightarrow x=1) \wedge (B@M5 \Rightarrow x=2) \wedge \\ (A@L0 \Rightarrow x=1) \wedge (A@L5 \Rightarrow x=2)\end{array}\right] \Rightarrow x == 2$$

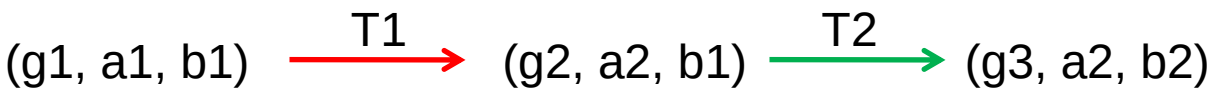
Annotation explosion!

- For sequential programs
  - assertion for each loop
  - assertion refers only to variables in scope
- For concurrent programs
  - assertion for each control location
  - assertion may need to refer to private state of other threads

Verification by analyzing abstractions

State-transition system

- Multithreaded program
  - Set of global states G
  - Set of local states L1, …, Ln
  - Set of initial states $I \subseteq G \times L1 \times \ldots \times Ln$
  - Transition relations T1, …, Tn
    - $Ti \subseteq (G \times Li) \times (G \times Li)$
  - Set of error states $E \subseteq G \times L1 \times \ldots \times Ln$



Example

- G = { (x, l) | x$\in${0,1,2}, l$\in${LOCK, UNLOCK} }

- L1 = { (t1, pc1) | t1$\in${0,1,2}, pc1$\in${L0,L1,L2,L3,L4,L5} }
- L2 = { (t2, pc2) | t2$\in${0,1,2}, pc2$\in${M0,M1,M2,M3,M4,M5} }

- I = { ((0, UNLOCK), (0, L0), (0, M0)) }

- E = { ((x, l), (t1, pc1), (t2, pc2)) | x $\neq$ 2 $\wedge$ pc1 == L5 $\wedge$ pc2 == M5 }

Reachability problem

- Does there exist an execution from a state in I to a state in E?

Reachability analysis

```
F = I
S = { }
while (F != ∅) {
    remove s from F
    if (s ∈ S) continue
    if (s ∈ E)
        return YES
    for every thread t:
        add every t-successor of s to F
    add s to S
}
return NO
```

Space complexity: $O(|G| \times |L|^n)$
Time complexity: $O(n \times |G| \times |L|^n)$

Reachability problem

- Does there exist an execution from a state in I to a state in E?
- PSPACE-complete
  - Little hope of polynomial-time solution in the general case

Challenge

- State space increases exponentially with the number of interacting components
- Utilize structure of concurrent systems to solve the reachability problem for programs with large state space

Tackling annotation explosion

- New specification primitive
  - "guarded_by *lock*" for data variables
  - "atomic" for code blocks
- Two layered proof
  - analyze synchronization to ensure that each atomic annotation is correct
  - do proof with assertions assuming atomic blocks execute without interruption

Bank account

```
Critical_Section l;
/*# guarded_by l */
int balance;
```

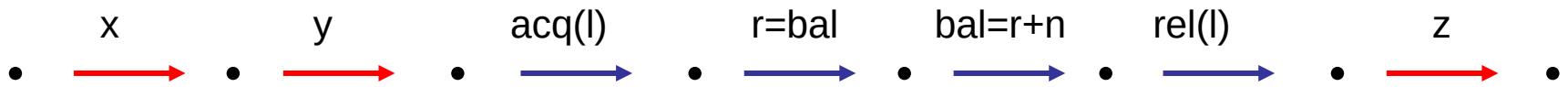
```
/*# atomic */
void deposit (int x)  {
   acquire(l);
   int r = balance;
   balance = r + x;
   release(l);
}
```

```
/*# atomic */
int read( )  {
   int r;
   acquire(l);
   r = balance;
   release(l);
   return r;
}
```

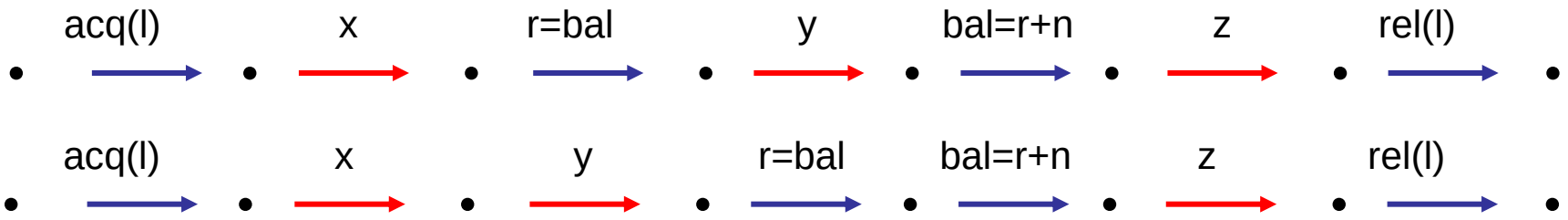
```
/*# atomic */
void withdraw(int x)  {
   acquire(l);
   int r = balance;
   balance = r – x;
   release(l);
}
```

Definition of atomicity

Serialized execution of deposit

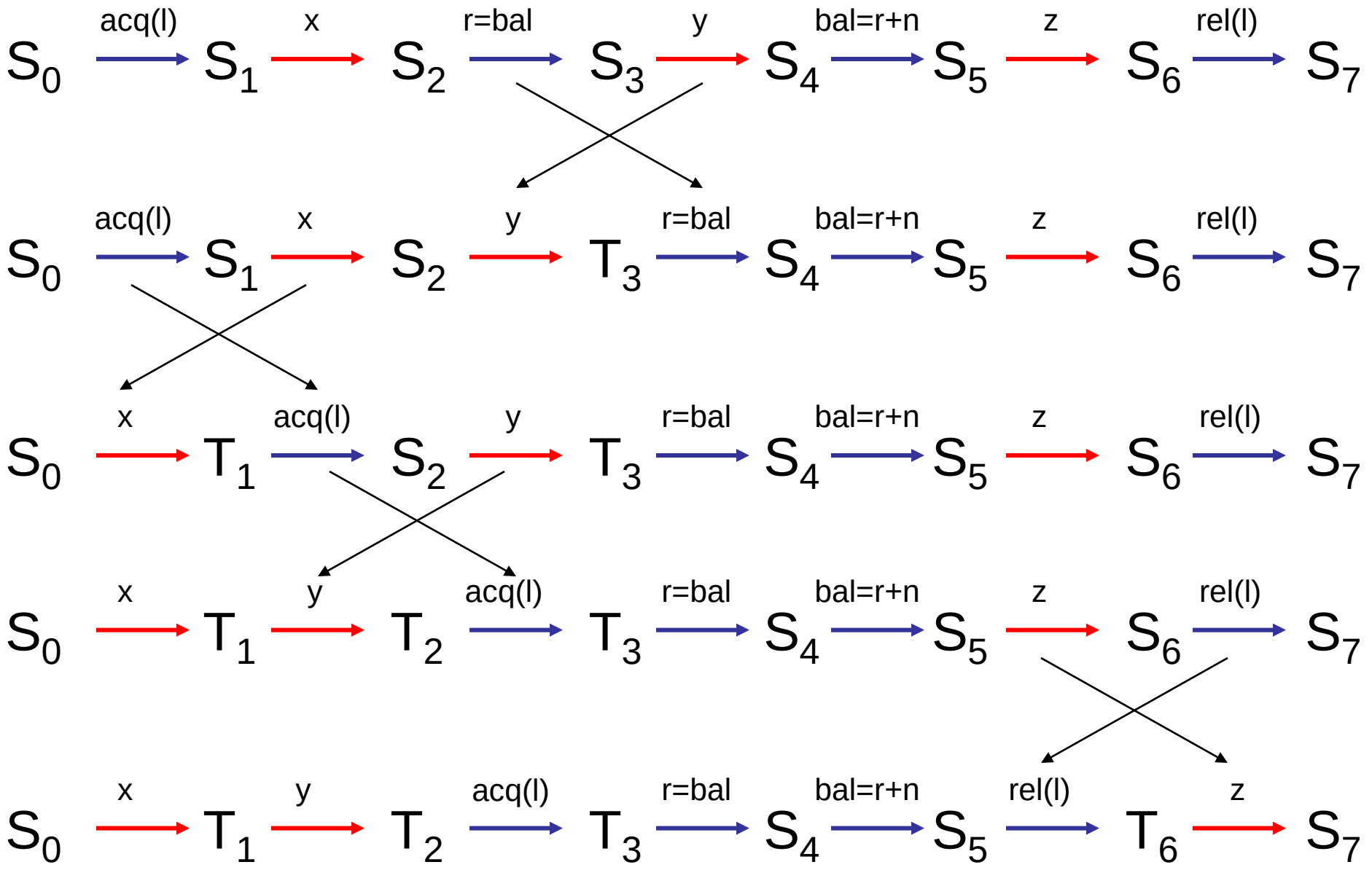


Non-serialized executions of deposit



- deposit is atomic if for every non-serialized execution, there is a serialized execution with the same behavior

Reduction


acq(l) x r=bal y bal=r+n z rel(l)
S0 S1 S2 S3 S4 S5 S6 S7
acq(l) x y r=bal bal=r+n z rel(l)
S0 S1 S2 T3 S4 S5 S6 S7
x acq(l) y r=bal bal=r+n z rel(l)
S0 T1 S2 T3 S4 S5 S6 S7
x y acq(l) r=bal bal=r+n z rel(l)
S0 T1 T2 T3 S4 S5 S6 S7
x y acq(l) r=bal bal=r+n rel(l) z
S0 T1 T2 T3 S4 S5 T6 S7

Four atomicities

R: right commutes
- lock acquire

L: left commutes
- lock release

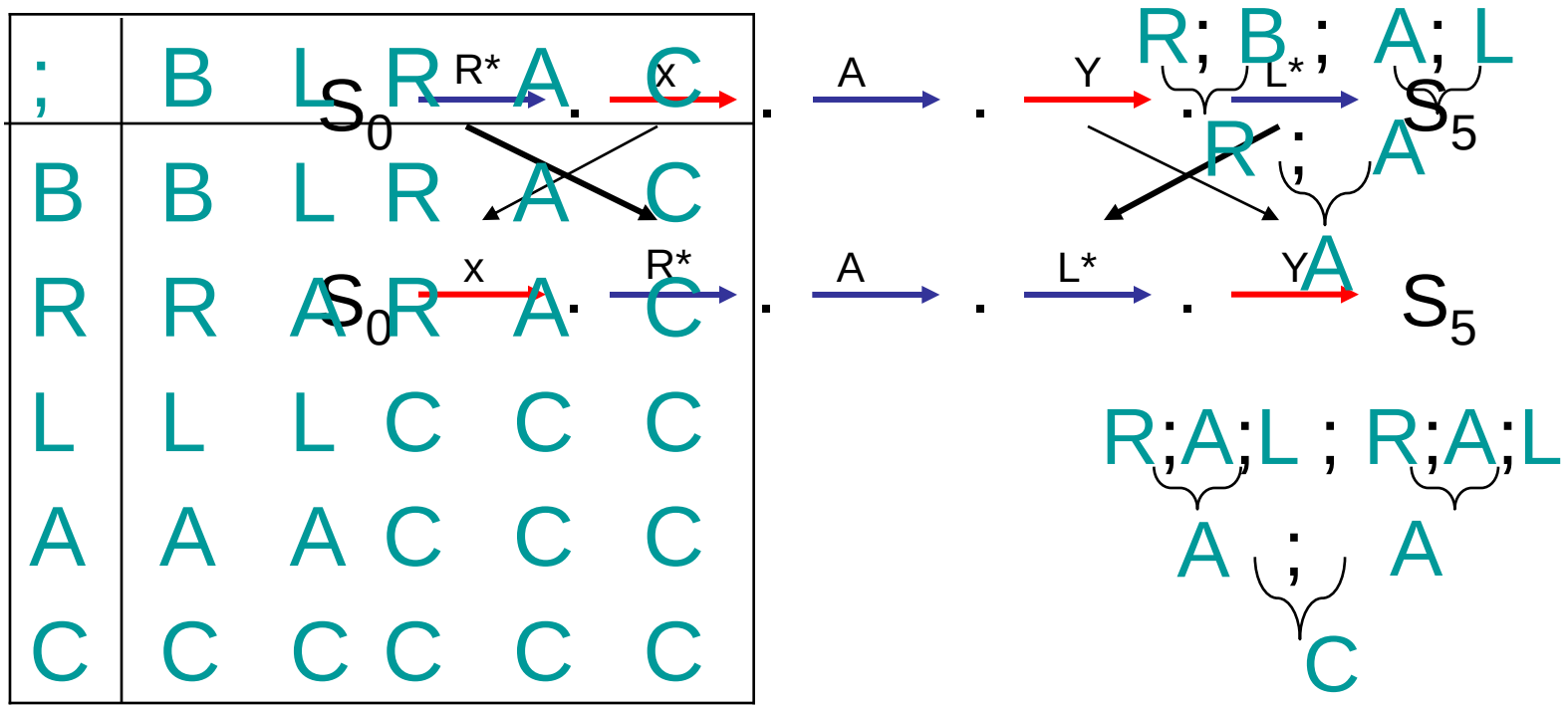
B: both right + left commutes
- variable access holding lock

A: atomic action, non-commuting
- access unprotected variable

Sequential composition

Use atomicities to perform reduction

Theorem: Sequence (R+B)*;(A+ε); (L+B)* is atomic



Bank account

```
Critical_Section l;
/*# guarded_by l */
int balance;
```

```
/*# atomic */
void deposit (int x)  {
A  R  acquire(l);
   B  int r = balance;
   B  balance = r + x;
   L  release(l);
}
```

```
/*# atomic */
int read( )  {
   int r;
A  R  acquire(l);
   B  r = balance;
   L  release(l);
   B  return r;
}
```

```
/*# atomic */
void withdraw(int x)  {
A  R  acquire(l);
   B  int r = balance;
   B  balance = r – x;
   L  release(l);
}
```

Correct!

Bank account

```
Critical_Section l;
/*# guarded_by l */
int balance;
```

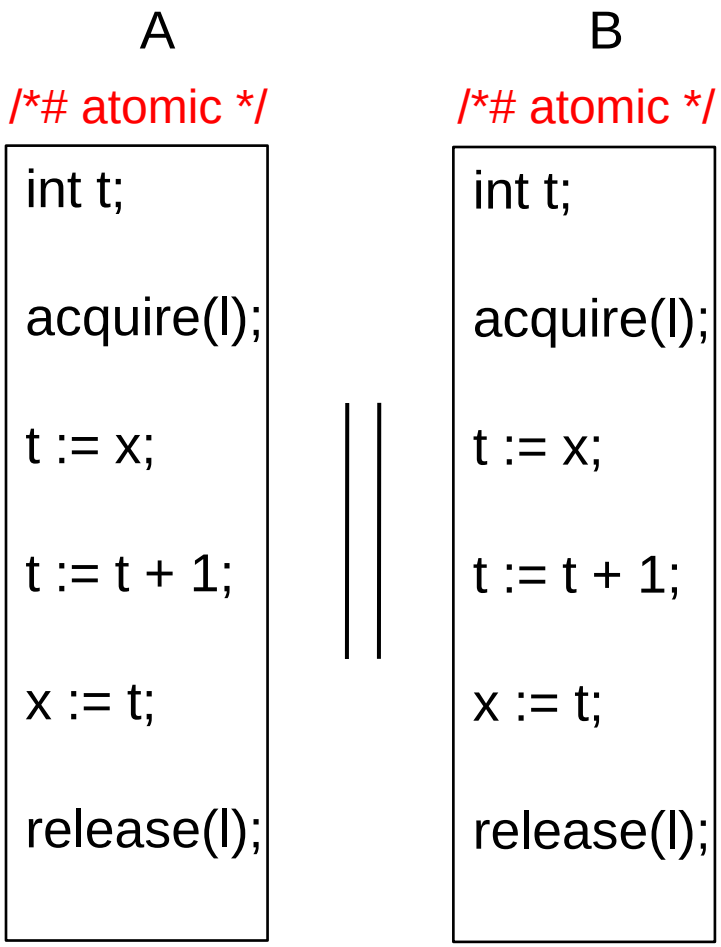
```
/*# atomic */
void deposit (int x)  {
A  R  acquire(l);
   B  int r = balance;
   B  balance = r + x;
   L  release(l);
  }
```

```
/*# atomic */
int read( )  {
   int r;
A  R  acquire(l);
   B  r = balance;
   L  release(l);
   B  return r;
  }
```

```
/*# atomic */
void withdraw(int x)  {
C  A  int r = read();
   R  acquire(l);
   B  balance = r – x;
   L  release(l);
  }
```

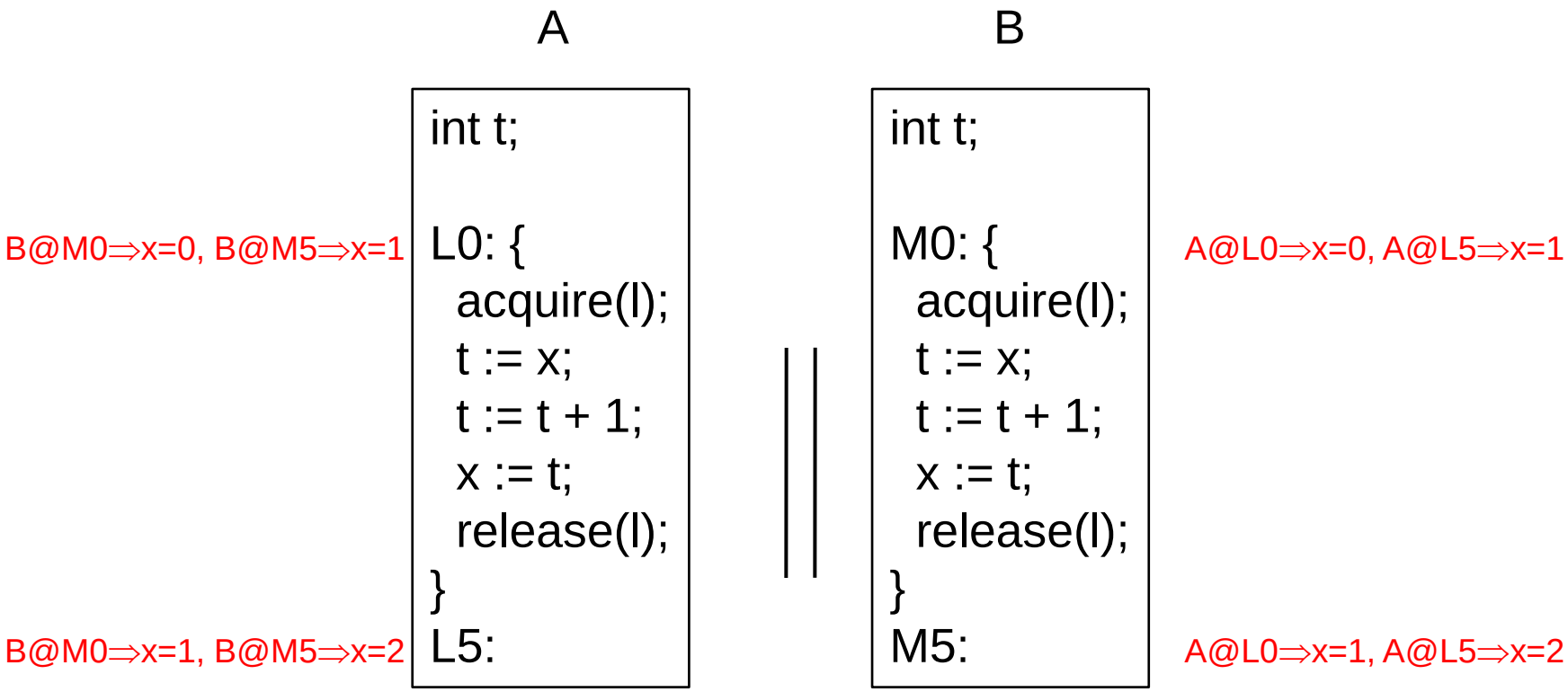
Incorrect!

pre x = 0;



post x = 2;

pre x = 0;



post x = 2;

Tackling state explosion

- Symbolic model checking
- Bounded model checking
- Context-bounded verification
- Partial-order reduction

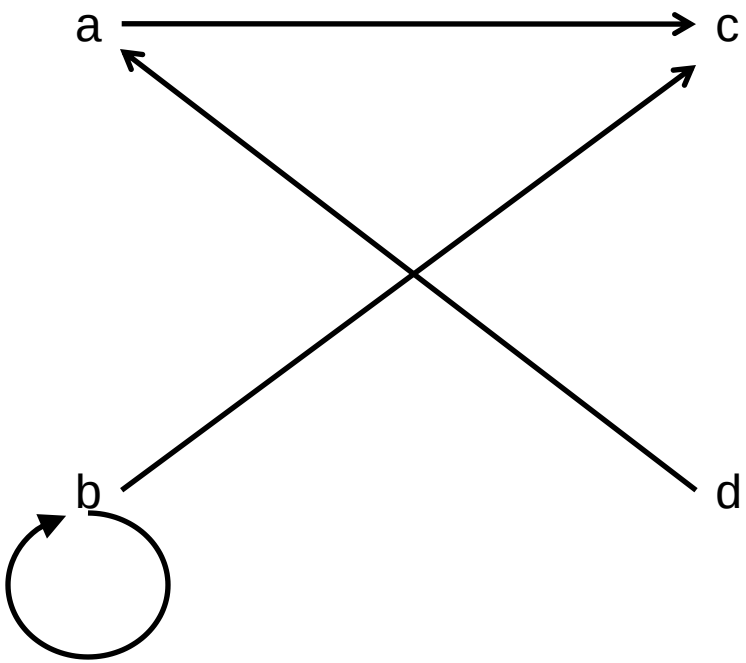
Recapitulation

- Multithreaded program
  - Set of global states G
  - Set of local states L1, …, Ln
  - Set of initial states $I \subseteq G \times L1 \times \ldots \times Ln$
  - Transition relations T1, …, Tn
    - $Ti \subseteq (G \times Li) \times (G \times Li)$
  - Set of error states $E \subseteq G \times L1 \times \ldots \times Ln$
- $T = T1 \cup \ldots \cup Tn$
- Reachability problem: Is there an execution from a state in I to a state in E?

Symbolic model checking

- Symbolic domain
  - universal set U
  - represent sets $S \subseteq U$ and relations $R \subseteq U \times U$
  - compute $\cup$, $\cap$, $\subset$, \, etc.
  - compute post(S, R) = $\{ s \mid \exists s' \in S.\ (s',s) \in R \}$
  - compute pre(S, R) = $\{ s \mid \exists s' \in S.\ (s,s') \in R \}$

R = {(a,c), (b,b), (b,c), (d, a)}



Post({a,b}, R) = {b, c}

Pre({a, c}, R) = {a, b, d}

Boolean logic as symbolic domain

- $U = G \times L1 \times \ldots \times Ln$
- Represent any set $S \subseteq U$ using a formula over $\log |G| + \log |L1| + \ldots + \log |Ln|$ Boolean variables
- $R_i = \{ ((g,l_1,\ldots,l_i,\ldots,l_n), (g',l_1,\ldots,l_i',\ldots,l_n)) \mid ((g,l_i), (g',l_i')) \in Ti \}$
- Represent Ri using $2 \times (\log |G| + \log |L1| + \ldots + \log |Ln|)$ Boolean variables
- $R = R1 \vee \ldots \vee Rn$

Forward symbolic model checking

```
S := ∅
S' := I
while  S ⊂ S' do {
    if (S' ∩ E ≠ ∅)
        return YES
    S := S'
    S' := S ∪ post(S, R)
}
return NO
```

Backward symbolic model checking

$S := \emptyset$

$S' := E$

while $S \subset S'$ do {

    if $(S' \cap I \neq \emptyset)$

        return YES

    $S := S'$

    $S' := S \cup pre(S, R)$

}

return NO

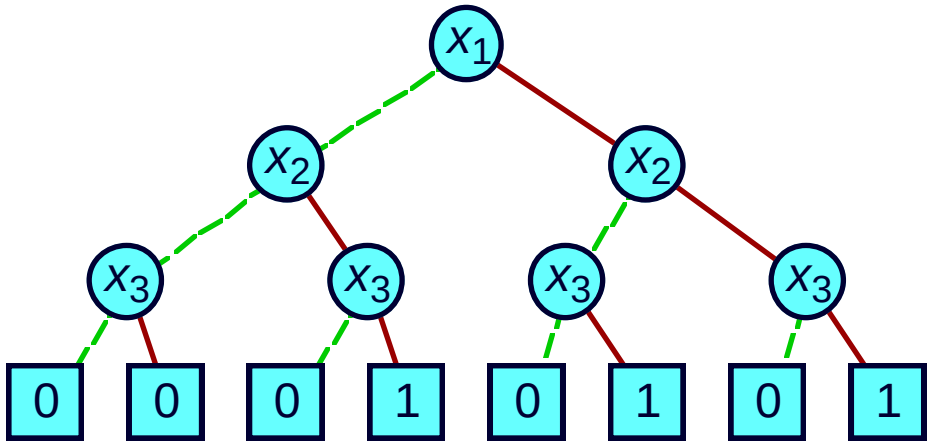
Symbolic model checking

- Symbolic domain
  - represent a set S and a relation R
  - compute $\cup$, $\cap$, $\subset$, etc.
  - compute post(S, R) = $\{ s \mid \exists s' \in S.\ (s', s) \in R \}$
  - compute pre(S, R) = $\{ s \mid \exists s' \in S.\ (s, s') \in R \}$
- Often possible to compactly represent large number of states
  - Binary decision diagrams

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value
- Along each path, variables occur in the variable order
- Along each path, a variable occurs exactly once

(Reduced Ordered) Binary Decision Diagram

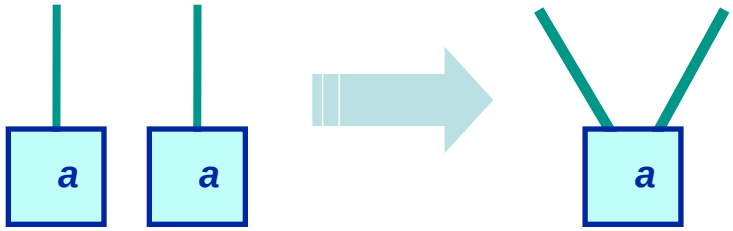
1. Identify isomorphic subtrees (this gives a dag)
2. Eliminate nodes with identical left and right successors
3. Eliminate redundant tests

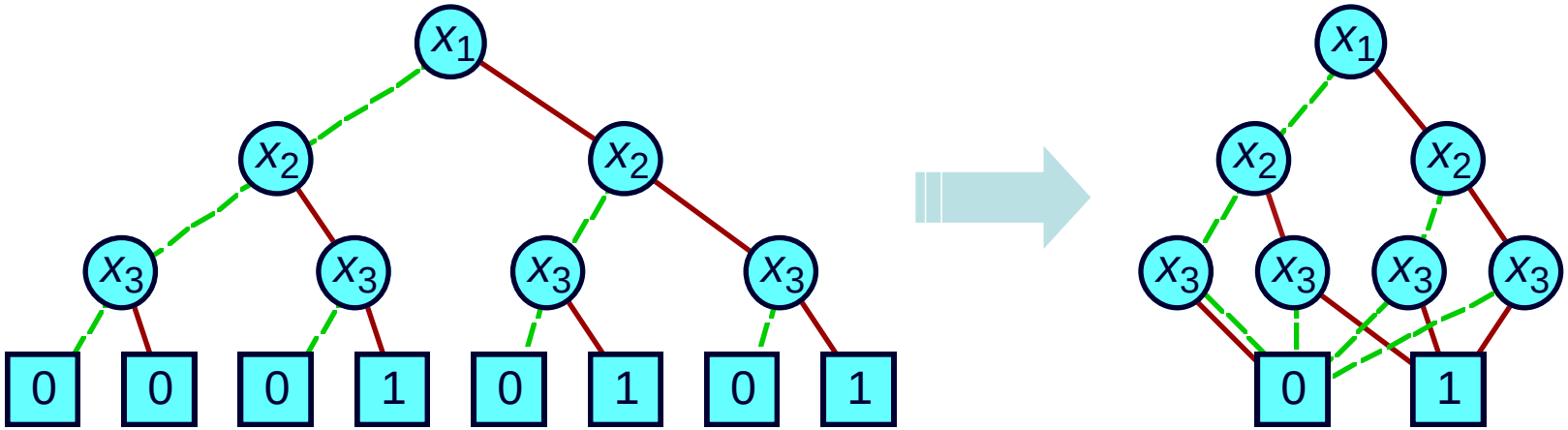
For a given boolean formula and variable order, the result is unique.

(The choice of variable order may make an exponential difference!)

Reduction rule #1

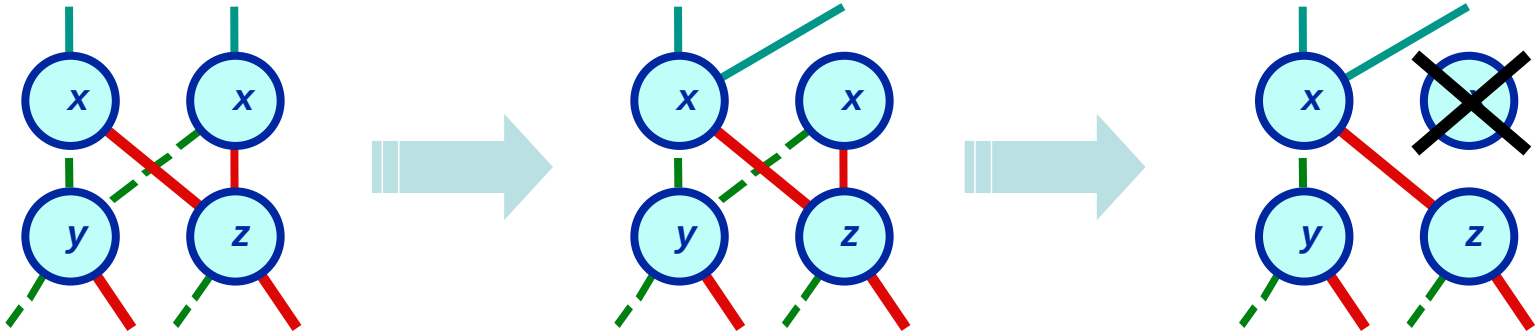
Merge equivalent leaves

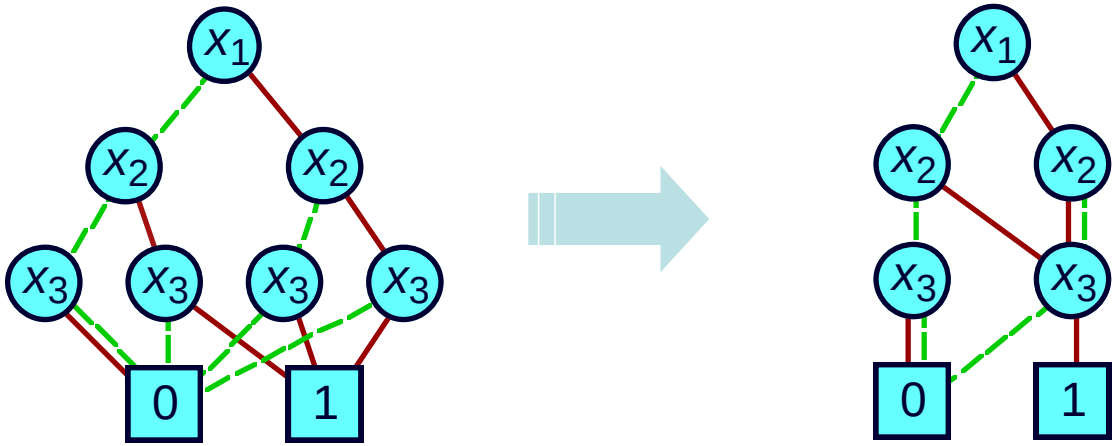




Reduction rule #2

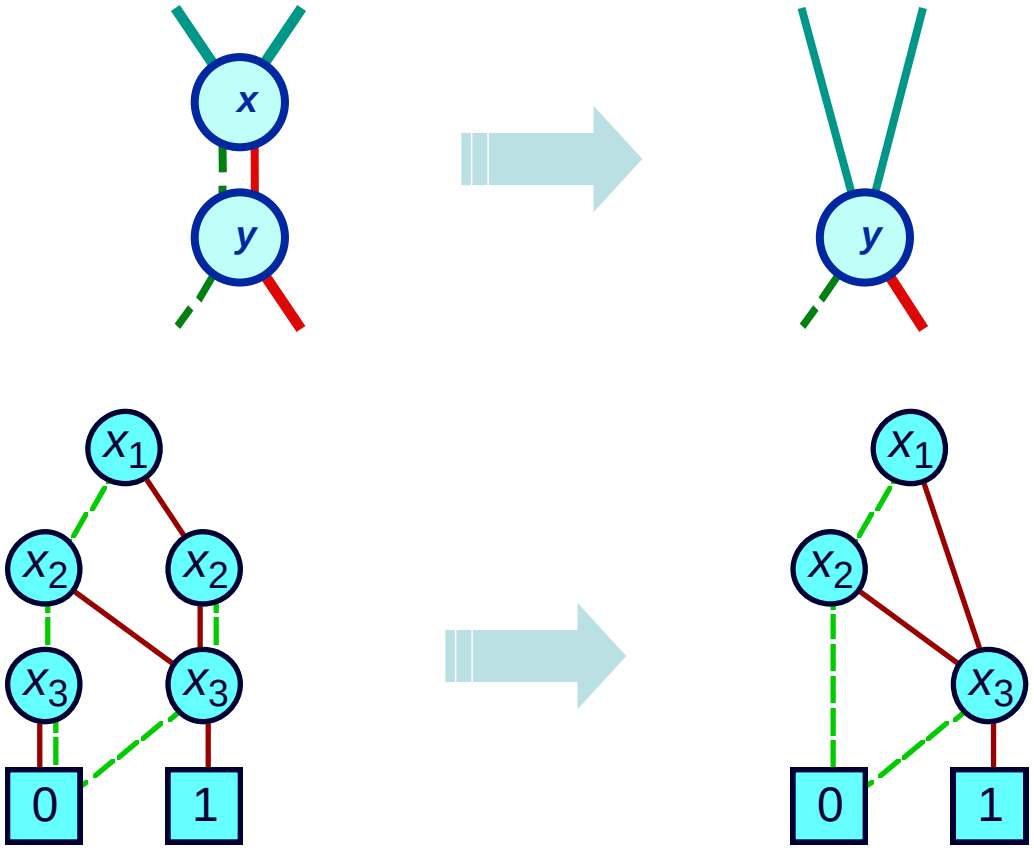
Merge isomorphic nodes



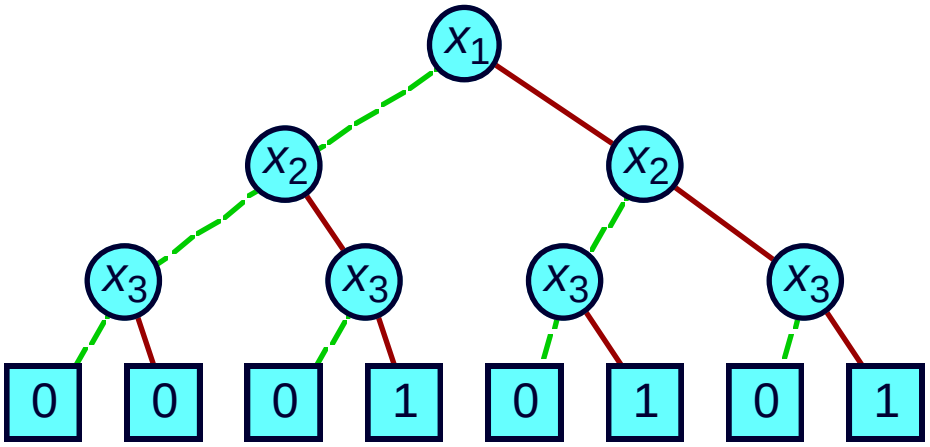


Reduction rule #3

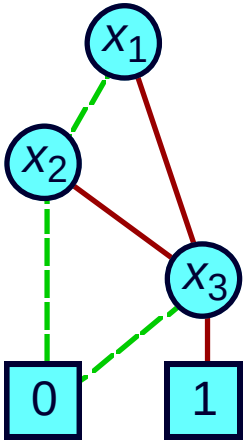
Eliminate redundant tests



Initial graph



Reduced graph

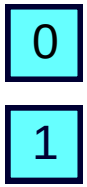


$(x_1 \vee x_2) \wedge x_3$

- Canonical representation of Boolean function
- For given variable ordering, two functions equivalent if and only if their graphs are isomorphic
- Test in linear time

Examples

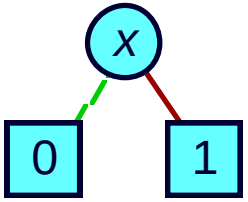
Constants



Unique unsatisfiable function

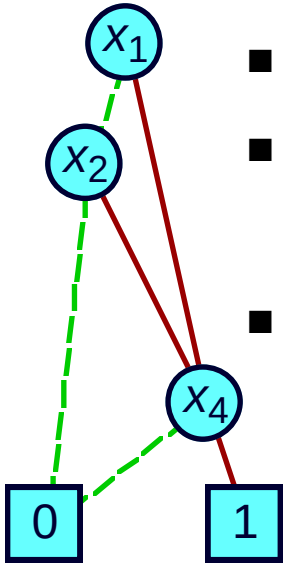
Unique tautology

Variable



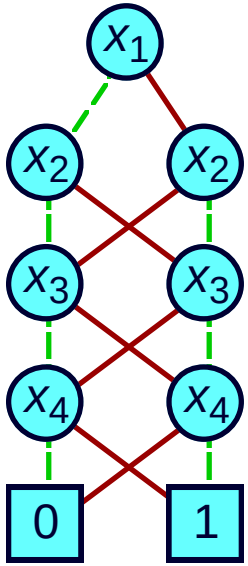
Treat variable as function

Typical function



- $(x_1 \vee x_2) \wedge x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Odd parity

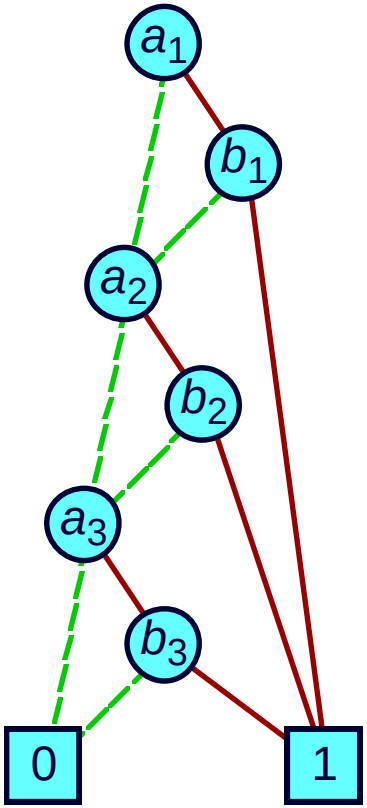


Linear representation

Effect of variable ordering

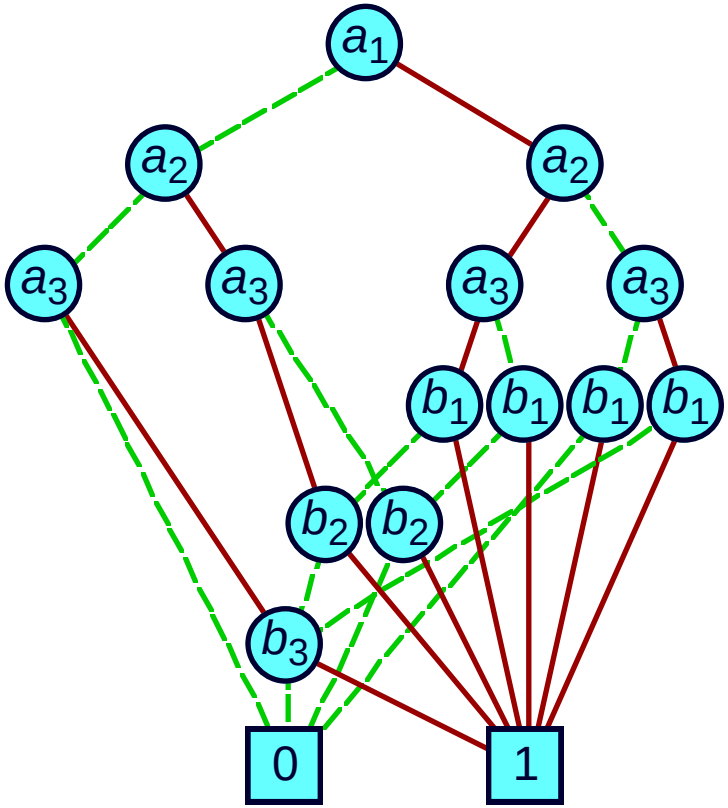
$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

Good ordering



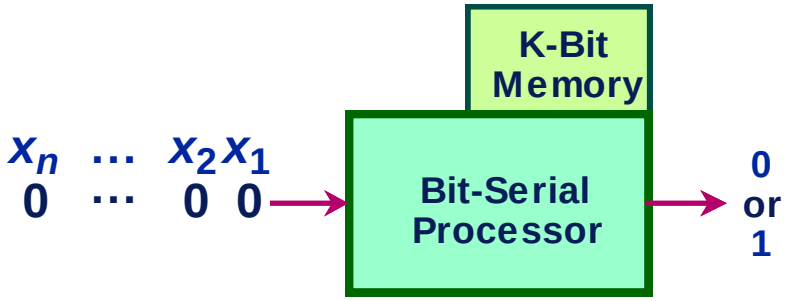
Linear growth

Bad ordering



Exponential growth

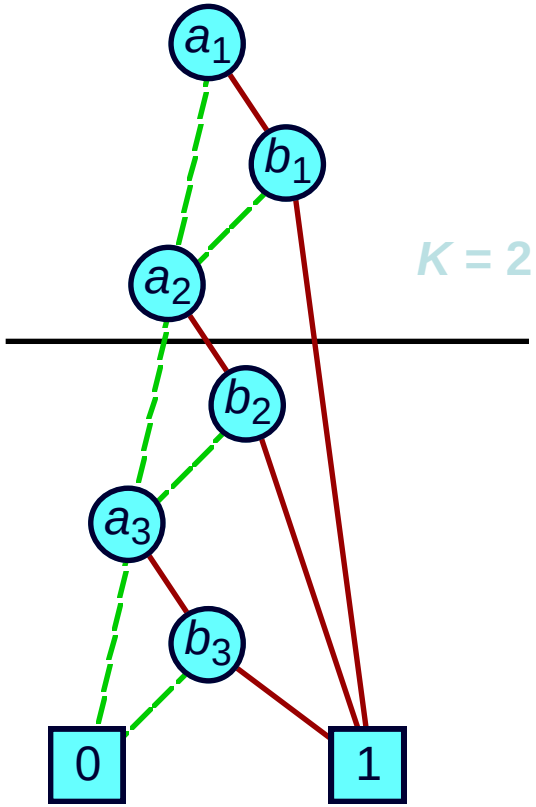
Bit-serial computer analogy



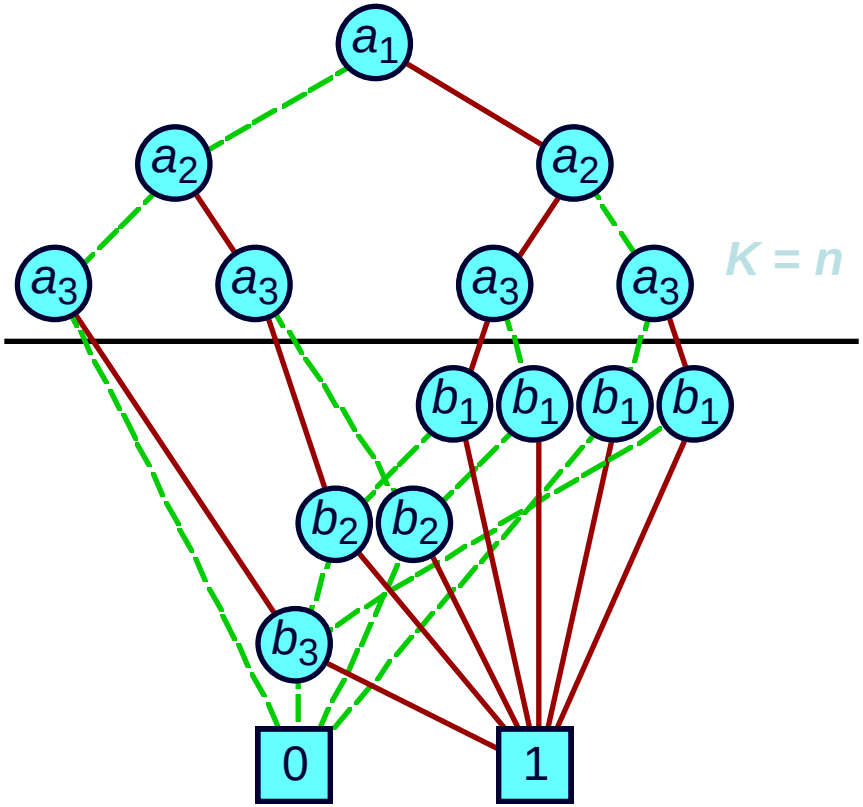
- Operation
  - Read inputs in sequence; produce 0 or 1 as function value.
  - Store information about previous inputs to correctly deduce function value from remaining inputs.
- Relation to BDD Size
  - Processor requires K bits of memory at step i.
  - BDD has ~2^K branches crossing level i.

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good ordering

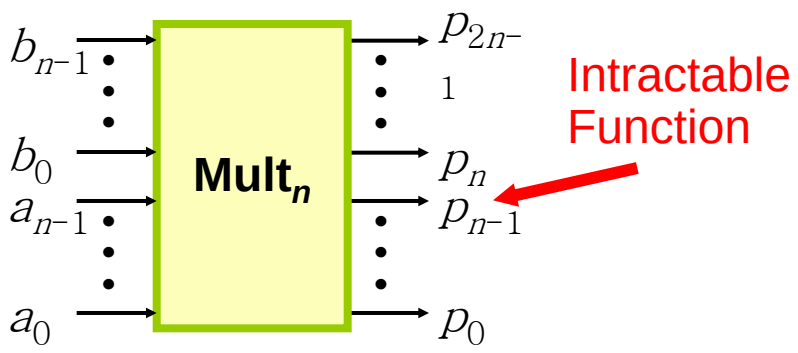


Bad ordering



Lower bound for multiplication (Bryant 1991)

- Integer multiplier circuit
  - *n*-bit input words *A* and *B*
  - 2*n*-bit output word *P*
- Boolean function
  - Middle bit (*n*-1) of product
- Complexity
  - Exponential BDD for all possible variable orderings

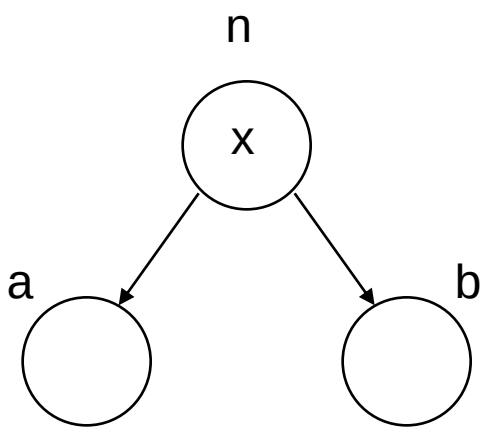


Actual Numbers

- 40,563,945 BDD nodes to represent all outputs of 16-bit multiplier
- Grows 2.86x per bit of word size

BDD operations ¬, ∧, ∨, ∃, ∀

BDD node



n.var = x
n.false = a
n.true = b

- BDD manager maintains a directed acyclic graph of BDD nodes
- ite(x,a,b) returns
  - if a = b, then a (or b)
  - if a ≠ b, then a node with variable x, left child a, and right child b

and(a,b)

```
if (a = false ∨ b = false) return false
if (a = true) return b
if (b = true) return a
if (a = b) return a
if (a.var < b.var)
    return ite(a.var, and(a.false,b), and(a.true,b))
if (b.var < a.var)
    return ite(b.var, and(a,b.false), and(a,b.true))
// a.var = b.var
return ite(a.var, and(a.false,b.false), and(a.true,b.true))
```

Complexity: $O(|a| \times |b|)$

not(a)

if (a = true) return false
if (a = false) return true
return ite(a.var, not(a.false), not(a.true))

Complexity: O(|a|)

cofactor(a,x,p)

```
if (x < a.var) return a
if (x > a.var)
    return ite(a.var, cofactor(a.false,x,p),
                      cofactor(a.true,x,p))
// x = a.var
if (p)
    return a.true
else
    return a.false
```

Complexity: O(|a|)

substitute(a,x,y)

Assumptions
- a is independent of y
- x and y are adjacent in variable order

```
if (a = true ∨ a = false) return a
if (a.var > x) return a
if (a.var < x)
    return ite(a.var, substitute(a.false,x,y),
                      substitute(a.true,x,y))
if (a.var = x) return ite(y,a.false,a.true)
```

Derived operations

or(a,b) ≡ not(and(not(a),not(b)))

exists(a,x) ≡ or(cofactor(a,x,false), cofactor(a,x,true))

forall(a,x) ≡ and(cofactor(a,x,false), cofactor(a,x,true))

implies(a,b) ≡ (or(not(a),b) = true)

iff(a,b) ≡ (a = b)

Tools that use BDDs

- Finite-state machine
  - SMV, VIS, …
- Pushdown machine
  - Bebop, Moped, …
- …

Applying symbolic model checking is difficult

- Symbolic representation for successive approximations to the reachable set of states may explode
  - hardware circuits, cache-coherence protocols, etc.

Bounded model checking

Problem: Is there an execution from a state in I to a state in E of length at most d?

```
for each (s ∈ I) {
    Explore(s, d)
}
exit(NO)

Explore(s, x) {
    if (s ∈ E) exit(YES)
    if (x == 0) return
    for each thread t:
        for each t-successor s' of s:
            Explore(s', x-1)
    }
}
```

Space complexity: O(d)
Time complexity: $O(n^d)$

NP-complete

Symbolic bounded model checking

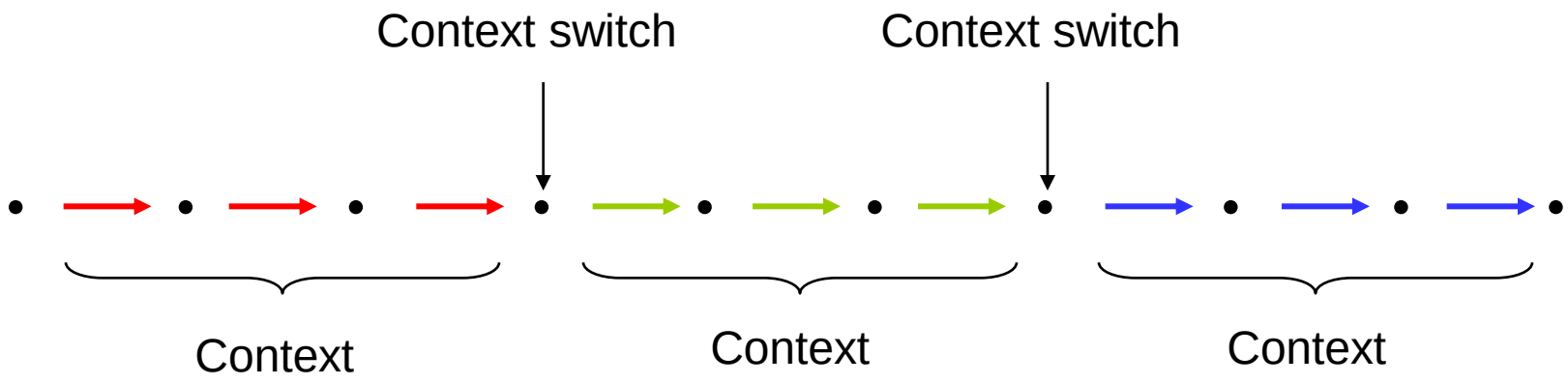
- Construct Boolean formula $\varphi(d)$
  - $I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{d-1}, s_d) \wedge (E(s_0) \vee \ldots \vee E(s_d))$
- φ is satisfiable iff there is an execution from a state in I to a state in E of length at most d

- Set k to 0
- Check satisfiability of $\varphi(k)$
- If unsatisfiable, increment k and iterate

Symbolic bounded model checking

- Pros
  - Eliminates existential quantification and state caching
  - Leverage years of research on SAT solvers
  - Finds shallow bugs (low k)
- Cons
  - Difficult to prove absence of bugs
  - May not help in finding deep bugs (large k)

Context-bounded verification



- Many subtle concurrency errors are manifested in executions with few context switches
- Analyze all executions with few context switches
- Unbounded computation within each context
  - Different from bounded model checking

Context-bounded reachability problem

- An execution is c-bounded if every thread has at most c contexts
- Does there exist a c-bounded execution from a state in I to a state in E?

Context-bounded reachability (I)

A work item (s, p) indicates that from state s,
p(i) contexts of thread i remain to be explored

```
for each thread t {
    ComputeRTClosure(t)
}
for each (s ∈ I) {
    Explore(s, λt. c)
}
exit(NO)

Explore(s, p) {
    if (s ∈ E) exit(YES)
    for each u such that (p(u) > 0) {
        for each s' such that (s, s') ∈ RT(u)
            Explore(s', p[u := p(u)-1])
    }
}
```

```
ComputeRTClosure(u) {
    F = {(s, s) | s ∈ G × L1 × … × Ln}
    while (F ≠ ∅) {
        Remove (s, s') from F
        for each u-successor s'' of s'
            add (s, s'') to F
        Add (s, s') to RT(u)
    }
}
```

Context-bounded reachability (II)

A work item (s, p) indicates that from state s,
p(i) contexts of thread i remain to be explored

```
for each (s ∈ I) {
    Explore(s, λt. c)
}
exit(NO)

Explore(s, p) {
    if (s ∈ E) exit(YES)
    for each u such that (p(u) > 0) {
        ComputeRTClosure(u, s)
        for each s' such that (s, s') ∈ RT(u)
            Explore(s', p[u := p(u)-1])
    }
}
```

```
ComputeRTClosure(u, s) {
    if (s, s) ∈ RT(u) return
    F = {(s, s)}
    while (F ≠ ∅) {
        Remove (s, s') from F
        for each u-successor s'' of s'
            add (s, s'') to F
        Add (s, s') to RT(u)
    }
}
```

Complexity analysis

Space complexity: $n \times (|G| \times |L|)^2$

Time complexity: $(nc)!/(c!)^n \times |I| \times (|G| \times |L|)^{nc}$

Although space complexity becomes polynomial, time complexity is still exponential.

Context-bounded reachability is NP-complete

Membership in NP: Witness is an initial state and nc sequences each of length at most $|G \times L|$

NP-hardness: Reduction from the CIRCUIT-SAT problem

Complexity of safety verification

	Unbounded	Context-bounded
Finite-state systems	PSPACE complete	NP-complete
Pushdown systems	Undecidable	NP-complete

P = # of program locations
G = # of global states

n = # of threads
c = # of contexts

Applying symbolic model checking is difficult

- Symbolic representation for successive approximations to the reachable set of states may explode
  - hardware circuits, cache-coherence protocols, etc.
- State might be complicated
  - finding suitable symbolic representation might be difficult
  - stacks, heap, queues, etc.

Enumerative model checking

- A demonic scheduler
  - Start from the initial state
  - Execute some enabled transition repeatedly
- Systematically explore choices
  - breadth-first, depth-first, etc.

Stateful vs. Stateless

- Stateful: capture and cache visited states
  - an optimization on terminating programs
  - required for termination on non-terminating programs
- Stateless: avoid capturing state
  - capturing relevant state might be difficult
  - depth-bounding for termination on non-terminating programs

Simplifying assumption

- Multithreaded program
  - Set of global states G
  - Set of local states L1, …, Ln
  - Unique initial state init $\in$ G × L1 × … × Ln
  - Partial transition functions T1, …, Tn
    - Ti : (G × Li) $\rightarrow$ (G × Li)
  - Set of error states E $\subseteq$ G × L1 × … × Ln
- T = T1 $\cup$ … $\cup$ Tn
- State transition graph defined by T is acyclic
- Reachability problem: Is there an execution from init to a state in E?

Observations

- A path is a sequence of thread ids
- Some paths are executable and called executions
- An execution results in a unique final state
- A multithreaded program with acyclic transition graph has a finite number of executions

T1	T2
acquire(m)	acquire(m)
x := 1	x := 2
release(m)	release(m)
acquire(n)	
y := 1	
release(n)	

- T1, T2 is not an execution
- T1, T1, T1, T2, T2, T2 is an execution
- T1, T1, T1, T2, T2, T2, T1, T1, T1 is a terminating execution

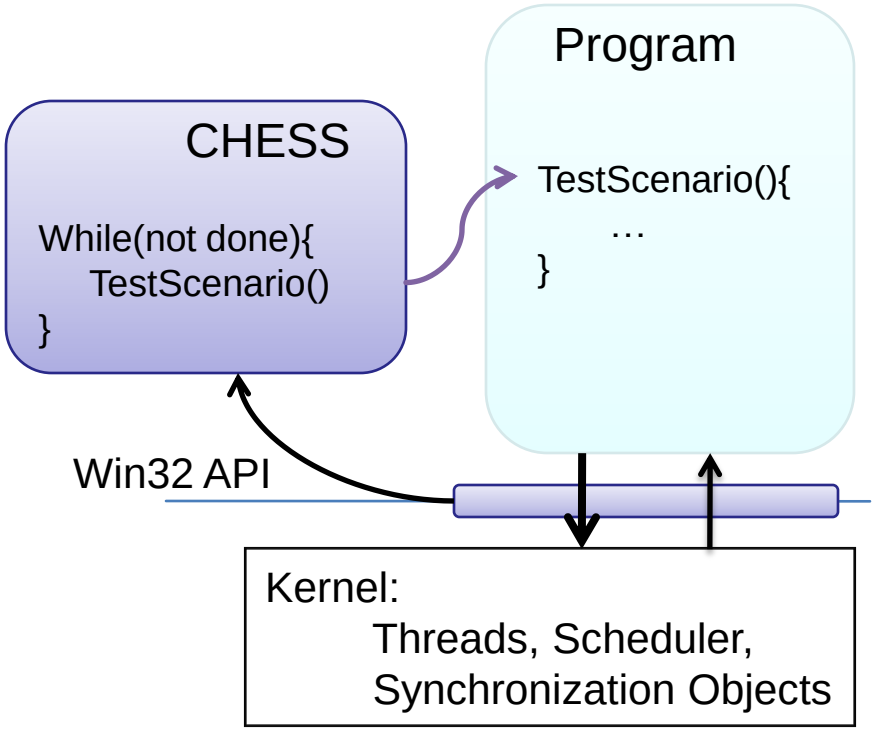
A simple stateless search algorithm

```
Explore(ε)
exit(NO)

Explore(e) {
    if (e ∈ E) exit(YES)
    for every thread t enabled in e {
        Explore(e•t)
    }
}
```

Tools

- Stateful: SPIN, Murphi, Java Pathfinder, CMC, Bogor, Zing, …
- Stateless: Verisoft, CHESS

CHESS: Systematic testing for concurrency

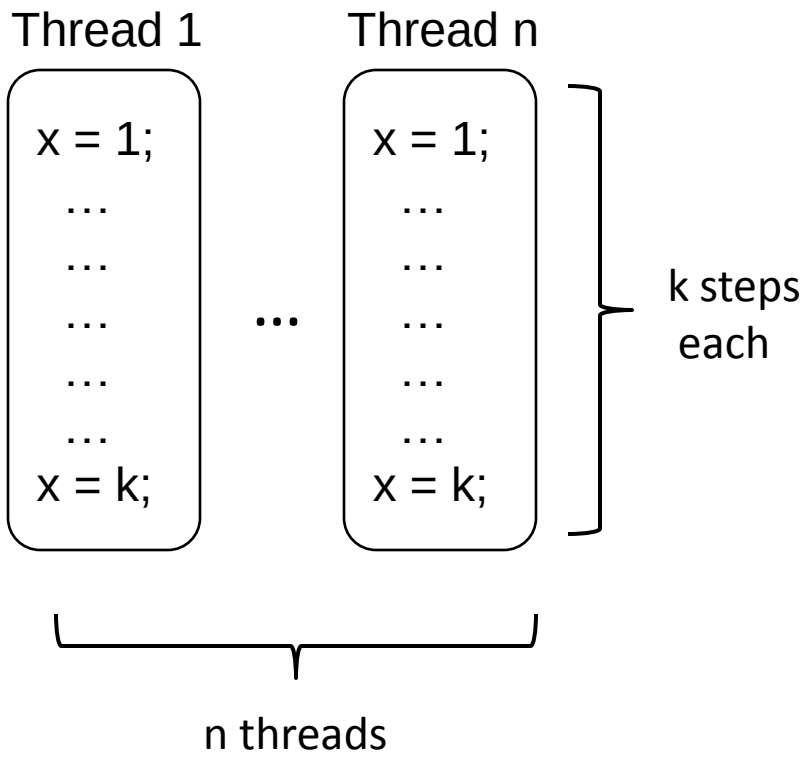


Tester Provides a Test Scenario

CHESS runs the scenario in a loop

- Each run is a different interleaving
- Each run is repeatable

State-space explosion

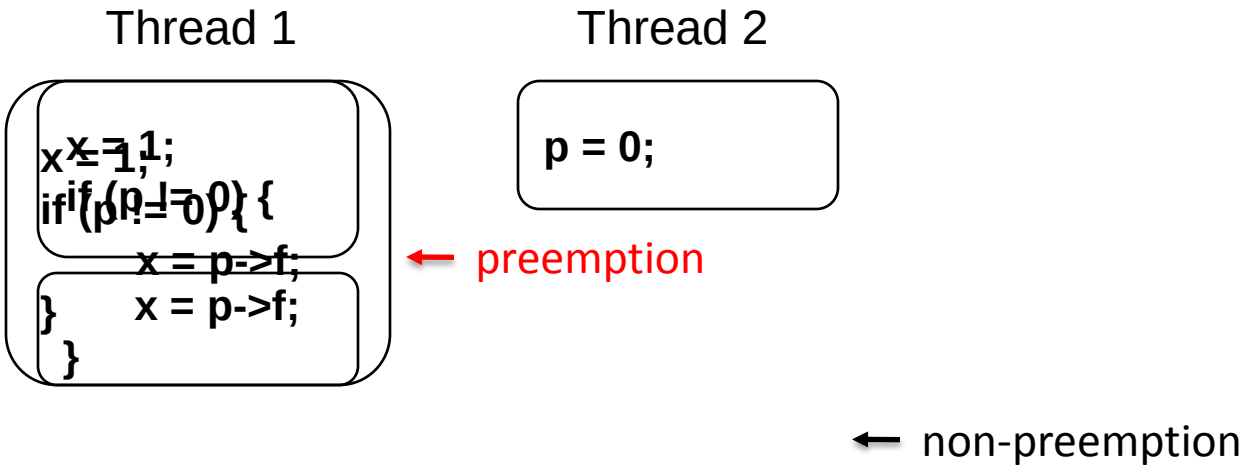


- Number of executions
  $= O(n^{nk})$
- Exponential in both n and k
  - Typically: $n < 10$ $k > 100$
- Limits scalability to large programs

Goal: Scale CHESS to large programs (large k)

Preemption-bounding

- Prioritize executions with small number of preemptions
- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end



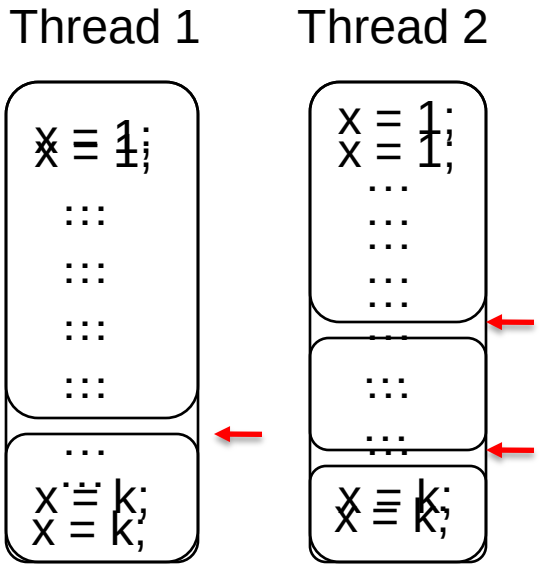
Preemption-bounding in CHESS

- The scheduler has a budget of c preemptions
  - Nondeterministically choose the preemption points
- Resort to non-preemptive scheduling after c preemptions
- Once all executions explored with c preemptions
  - Try with c+1 preemptions

Property 1: Polynomial bound

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
- Number of executions <= $_{nk}C_c$. (n+c)!
  = O($(n^2k)^c$. n!)

Exponential in n and c, but not in k



- Choose c preemption points
- Permute n+c atomic blocks

Property 2: Simple error traces

- Finds smallest number of preemptions to the error
- Number of preemptions better metric of error complexity than execution length

Property 3: Coverage metric

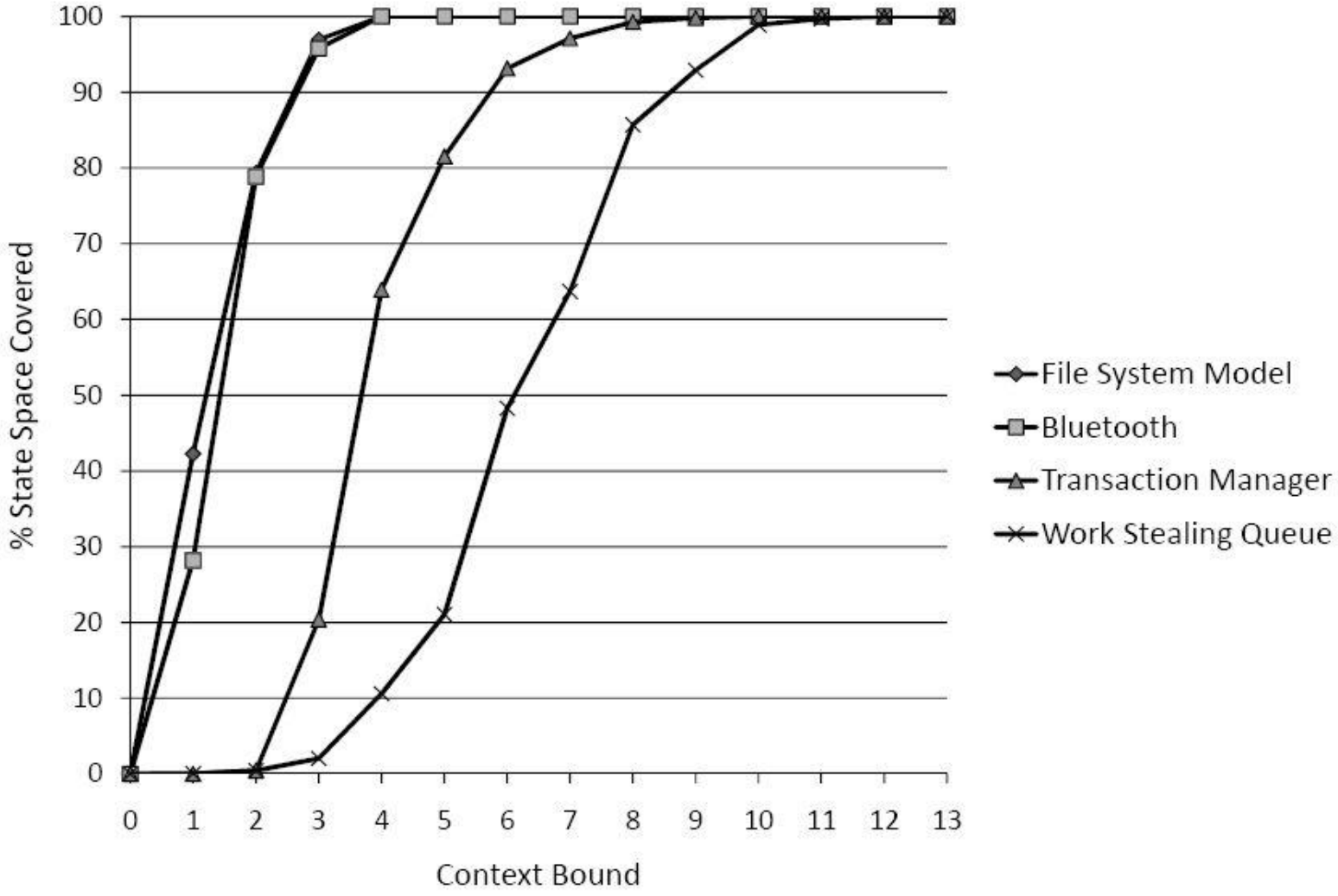
- If search terminates with preemption-bound of c, then any remaining error must require at least c+1 preemptions

- Intuitive estimate for
  - The complexity of the bugs remaining in the program
  - The chance of their occurrence in practice

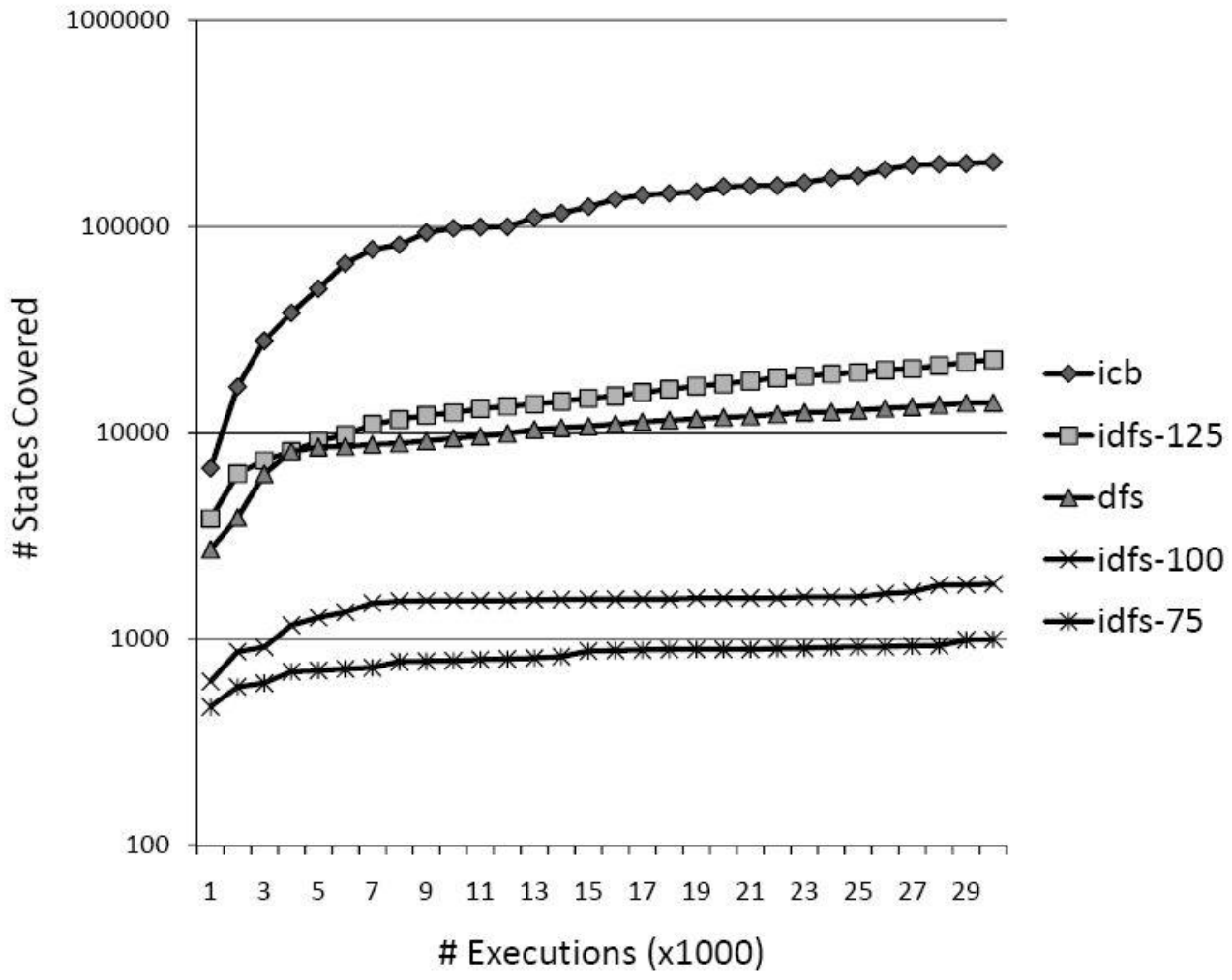
Property 4: Many bugs with few preemptions

Program	KLOC	Threads	Preemptions	Bugs
Work-Stealing Queue	1.3	3	2	3
CDS	6.2	3	2	1
CCR	9.3	3	2	2
ConcRT	16.5	4	3	4
Dryad	18.1	25	2	7
APE	18.9	4	2	4
STM	20.2	2	2	2
PLINQ	23.8	8	2	1
TPL	24.1	8	2	9

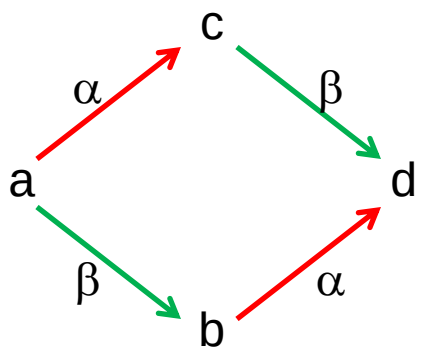
Coverage vs. Preemption-bound



Dryad (coverage vs. time)



Partial-order reduction



- Commuting actions
  - accesses to thread-local variables
  - accesses to different shared variables
- Useful in both stateless and stateful search

Goal: Explore only one interleaving of commuting actions by different threads

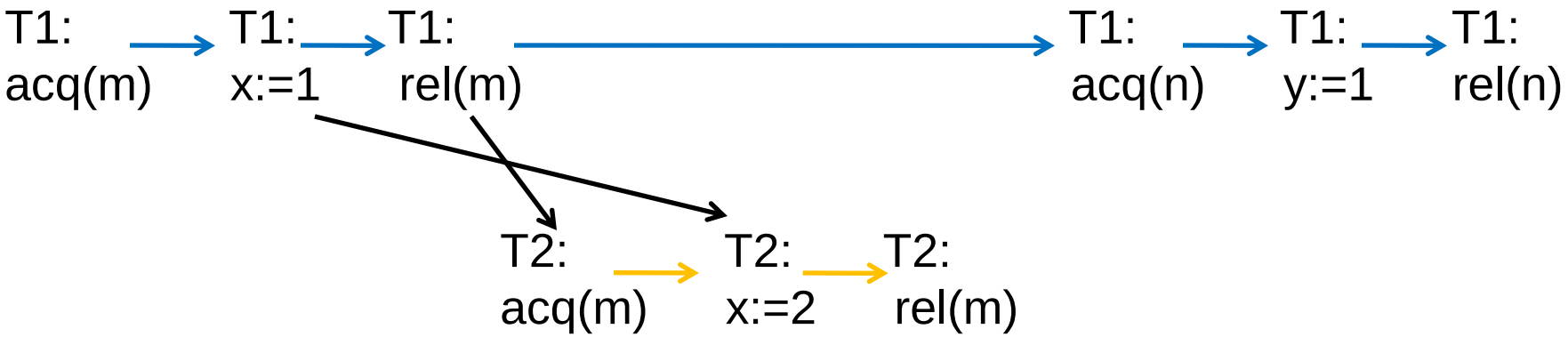
Happens-before graph

T1	T2
acquire(m)	acquire(m)
x := 1	x := 2
release(m)	release(m)
acquire(n)	
y := 1	
release(n)	

T1, T1, T1, T2, T2, T2, T1, T1, T1

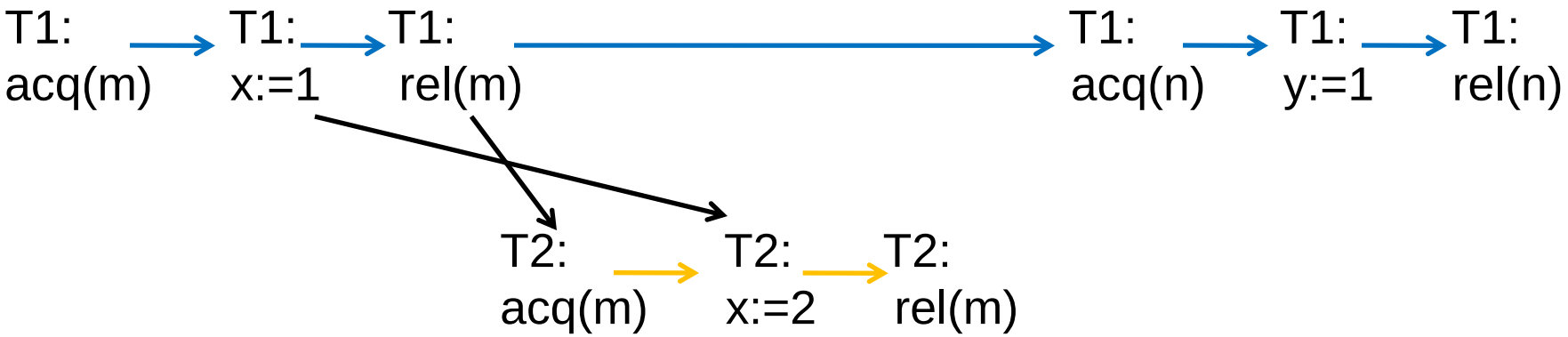
Happens-before graph

T1	T2
acquire(m)	acquire(m)
x := 1	x := 2
release(m)	release(m)
acquire(n)	
y := 1	
release(n)	



Happens-before graph

- An execution is a total order over thread actions
- The happens-before graph of an execution is a partial order over thread actions
  - All linearizations of a happens-before graph are executions and generate the same final state
- The happens-before graph is a canonical representative of an execution



A partial-order reduction algorithm

```
Explore(ε)
exit(NO)

Explore(e) {
    if (e ∈ E) exit(YES)
    add HB(e) to S
    for every thread t enabled in e {
        if (HB(e•t) ∈ S) continue
        Explore(e•t)
    }
}
```

Theorem: The algorithm explores exactly one linearization of each happens-before graph

Eliminating the HB-cache

- Order on thread ids: t1 < … < tn
- Extend < to the dictionary order on executions
- Rep(e) ≡ min {e' | HB(e) = HB(e')}

```
Explore(e) {
    if (e ∈ E) exit(YES)
    for every thread t enabled in e {
        if (Rep(e•t) < e•t) continue
        Explore(e•t)
    }
}
```

Iteration performed in order t1 < … < tn

Checking Rep(e) < e

- Straightforward
  - compute Rep(e) (linear time)
  - check Rep(e) < e (linear time)

- Incremental
  - We know that Rep(e) < e
  - What about Rep(e•t) < e•t?

Checking Rep(e) < e

$e \bullet t$

$t_1 \bullet \ldots \bullet t_p \bullet t_{p+1} \bullet \ldots \bullet t_q \bullet t$

t_p is the latest action with a happens-before edge to t

$$Rep(e \bullet t) < e \bullet t$$
iff
$t < t_k$ for some k in (p, q]

Comments

- Algorithm equivalent to the “sleep sets” algorithm
- Particularly useful for stateless search
  - converts the search *tree* into a search *graph*