Reduction: A powerful technique for analyzing concurrent software

Shaz Qadeer

Microsoft Research

Collaborators:

- Cormac Flanagan, UC Santa Cruz
- Stephen Freund, Williams College
- Sriram Rajamani, Microsoft Research
- Jakob Rehof, Microsoft Research

Concurrent programs

- Operating systems, device drivers, databases, Java/C#, web services, …
- Reliability is important

Reliable Concurrent Software?

- Correctness Problem
  - does program behaves correctly for *all inputs* and *all interleavings*?
  - very hard to ensure with testing
- Bugs due to concurrency are insidious
  - non-deterministic, timing dependent
  - data corruption, crashes
  - difficult to detect, reproduce, eliminate
- Security attacks exploiting concurrency are the next frontier

Part 1: Atomicity analysis

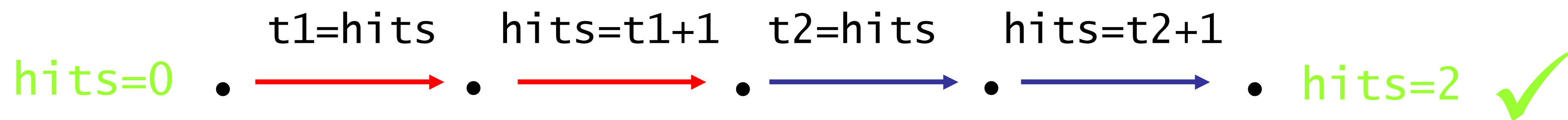
Multithreaded Program Execution

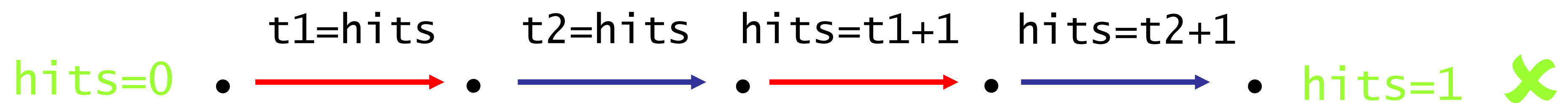
Thread 1

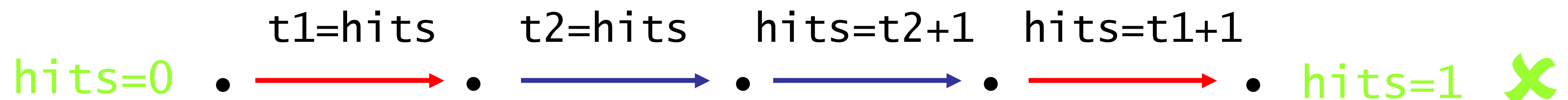
```
...
int t1 = hits;
hits = t1 + 1
...
```

Thread 2

```
...
int t2 = hits;
hits = t2 + 1
...
```







Race Conditions

A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write

Thread 1

```
...
int t1 = hits;
hits = t1 + 1
...
```

Thread 2

```
...
int t2 = hits;
hits = t2 + 1
...
```

Preventing Race Conditions Using Locks

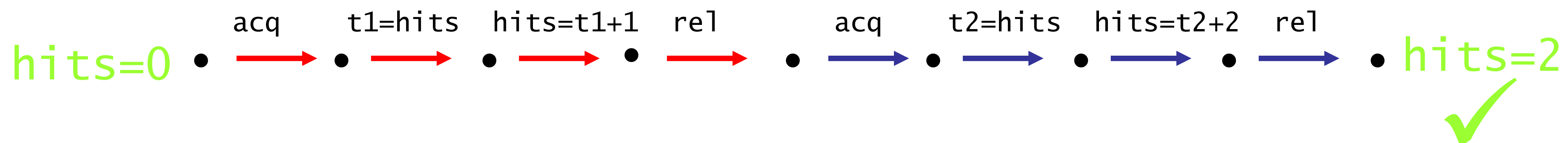
- Lock can be held by at most one thread
- Race conditions are prevented using locks
  - associate a lock with each shared variable
  - acquire lock before accessing variable

Thread 1

```
synchronized(lock) {
  int t1 = hits;
  hits = t1 + 1
}
```

Thread 2

```
synchronized(lock) {
  int t2 = hits;
  hits = t2 + 1
}
```



Race detection

- Static:
  - Sterling 93, Aiken-Gay 98, Flanagan-Abadi 99, Flanagan-Freund 00, Boyapati-Rinard 01, von Praun-Gross 01, Boyapati-Lee-Rinard 02, Grossman 03
- Dynamic:
  - Savage et al. 97 (Eraser tool)
  - Cheng et al. 98
  - Choi et al. 02

Race-free bank account

```
int balance;

void deposit (int n) {
  synchronized (this) {
    balance = balance + n;
  }
}
```

Bank account

```
int balance;

void deposit (int n) {
  synchronized (this) {
    balance = balance + n;
  }
}

int read( ) {
  int r;
  synchronized (this) {
    r = balance;
  }
  return r;
}

void withdraw(int n) {
  int r = read( );
  synchronized (this) {
    balance = r – n;
  }
}
```

balance = 10

Thread 1	Thread 2
deposit(10);	withdraw(10);

Race-freedom not sufficient!

Atomic bank account (I)

```
int balance;
```

```
void deposit (int n) {
  synchronized (this) {
    balance = balance + n;
  }
}
```

```
int read( ) {
  int r;
  synchronized (this) {
    r = balance;
  }
  return r;
}
```

```
void withdraw(int n) {
  synchronized (this) {
    balance = balance - n;
  }
}
```

java.lang.StringBuffer (jdk 1.4)

“String buffers are safe for use by multiple threads. The methods are synchronized so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.”

java.lang.StringBuffer is buggy!

```
public final class StringBuffer {
  private int count;
  private char[ ] value;
      .
      .
  public synchronized StringBuffer append (StringBuffer sb)  {
      if (sb == null) sb = NULL;
      int len = sb.length( );
      int newcount = count + len;
      if (newcount > value.length) expandCapacity(newcount);
      sb.getChars(0, len, value, count); //use of stale len !!
      count = newcount;
      return this;
  }

  public synchronized int length( ) { return count; }

  public synchronized void getChars(. . .) { . . . }
}
```

Atomic bank account (II)

```
int balance;

void deposit (int n) {
  synchronized (this) {
    balance = balance + n;
  }
}

int read( ) {
  return balance;
}

void withdraw(int n) {
  synchronized (this) {
    balance = balance – n;
  }
}
```

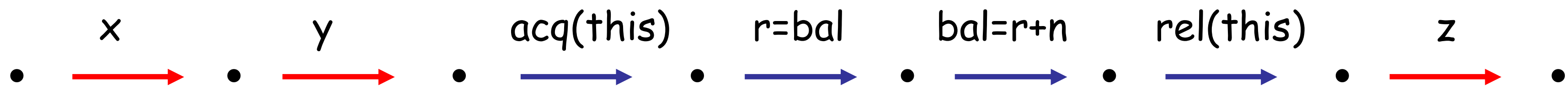
Race-freedom not necessary!

Atomicity

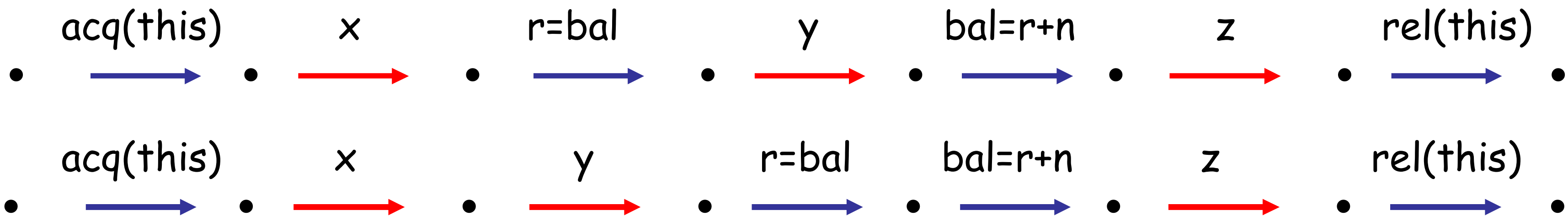
- A method is atomic if it seems to execute "in one step" even in presence of concurrently executing threads
- Common concept
  - "(strict) serializability" in databases
  - "linearizability" in concurrent objects
  - "thread-safe" multithreaded libraries
    - "`String buffers are safe for use by multiple threads.` …"
- Fundamental semantic correctness property

Definition of Atomicity

- Serialized execution of deposit

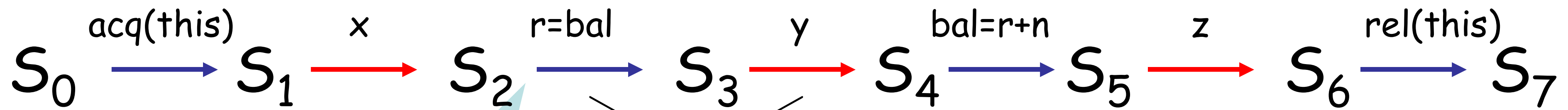


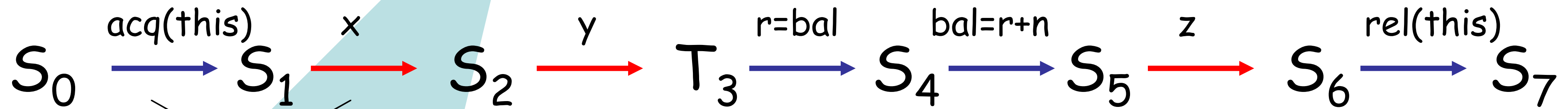
- Non-serialized executions of deposit



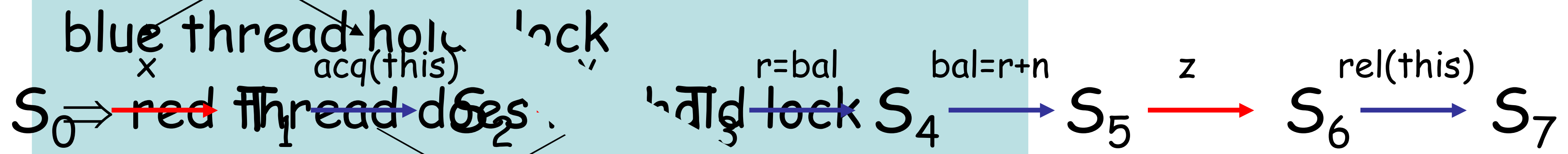
- deposit is atomic if for every non-serialized execution, there is a serialized execution with the same behavior

Reduction (Lipton 75)





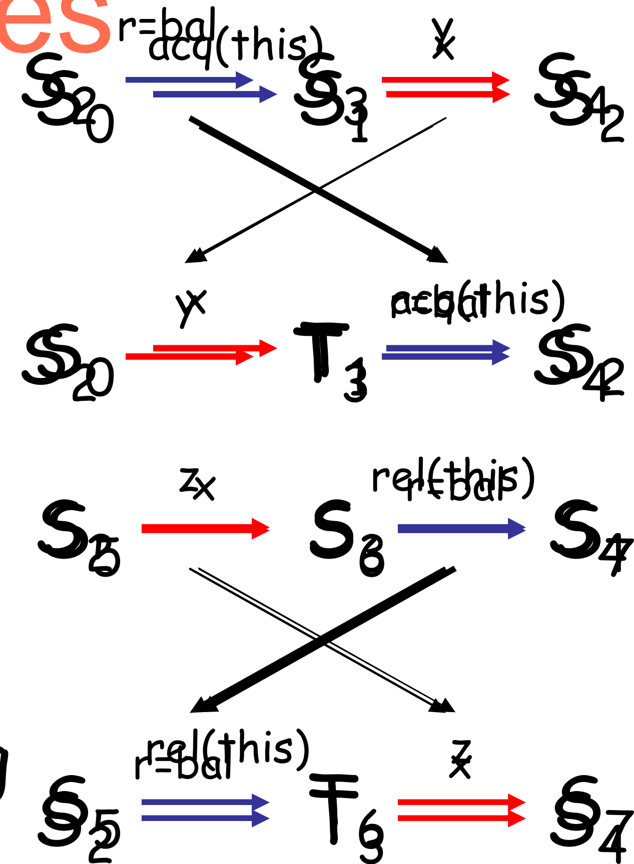
blue thread hol... lock
⇒ red thread does ... hold lock
⇒ operation y does no... balance
⇒ operations commute



blue thread holds lock after acquire
⇒ operation x does not modify lock
⇒ operations commute

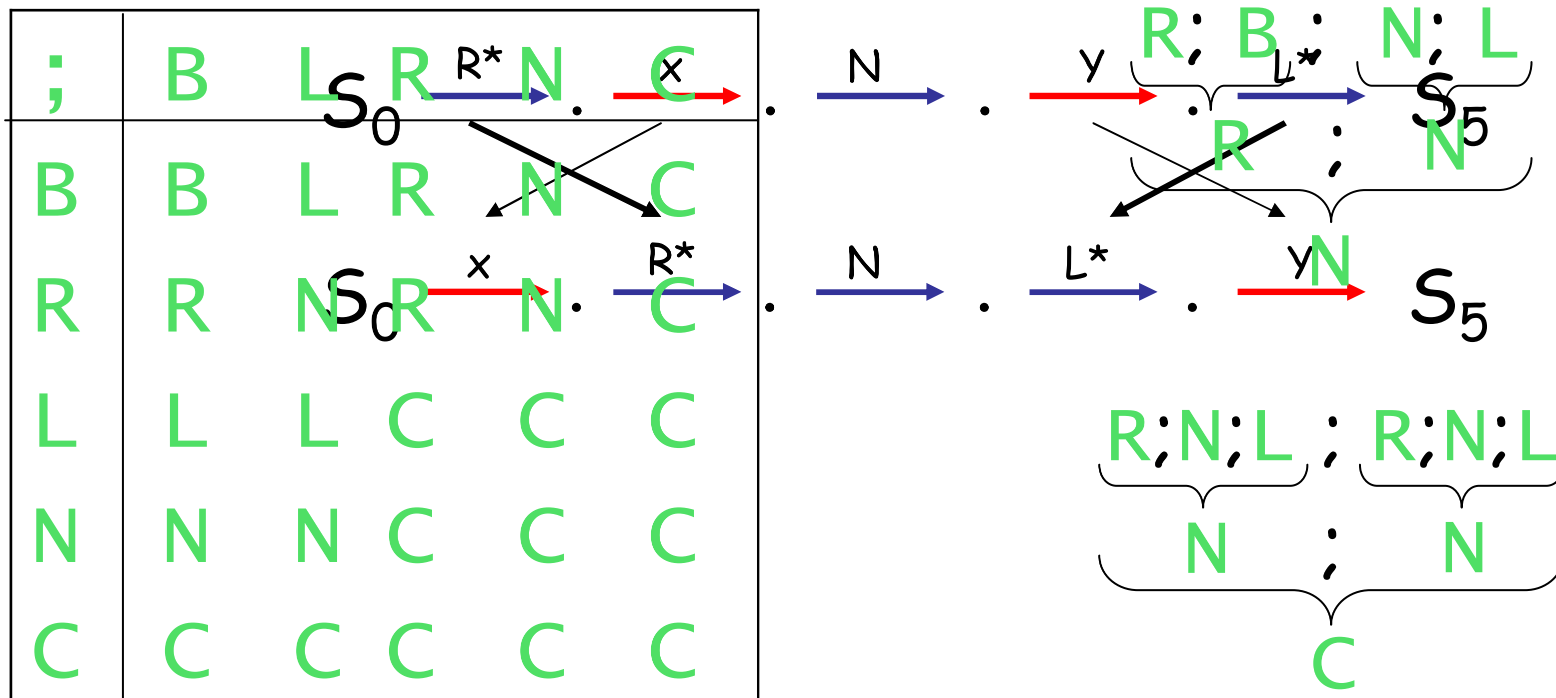
Four Atomicities

- R: right commutes
  – lock acquire
- L: left commutes
  - lock release
- B: both right + left commutes
  - variable access holding lock
- N: atomic action, non-commuting
  - access unprotected variable



Sequential Composition

- Use atomicities to perform reduction
- Lipton: sequence (R+B)*;(N+ε); (L+B)* is atomic



Bank account

```
int balance;  /*# guarded_by this */
```

```
    /*# atomicity N */
    void deposit (int x) {
  ⎧ R   acquire(this);
N ⎨ B   int r = balance;
  ⎪ B   balance = r + x;
  ⎩ L   release(this);
    }
```

```
    /*# atomicity N */
    int read( ) {
        int r;
  ⎧ R   acquire(this);
  ⎪ B   r = balance;
N ⎨ L   release(this);
  ⎩ B   return r;
    }
```

```
    /*# atomicity N */
    void withdraw(int x) {
    N   int r = read( );
  ⎧ R   acquire(this);
N ⎨ B   balance = r - x;
  ⎩ L   release(this);
    }
```

Bank account

```
int balance;  /*# guarded_by this */

    /*# atomicity N */
    void deposit (int x)  {
  ⎧ R   acquire(this);
N ⎨ B   int r = balance;
  ⎪ B   balance = r + x;
  ⎩ L   release(this);
    }

    /*# atomicity N */
    int read( )  {
        int r;
  ⎧ R   acquire(this);
  ⎪ B   r = balance;
N ⎨ L   release(this);
  ⎩ B   return r;
    }

    /*# atomicity N */
    void withdraw(int x)  {
  ⎧ R   acquire(this);
N ⎨ B   int r = balance;
  ⎪ B   balance = r - x;
  ⎩ L   release(this);
    }
```

Soundness Theorem

- Suppose a *non-serialized* execution of a well-typed program reaches state S in which no thread is executing an atomic method
- Then there is a *serialized* execution of the program that also reaches S

Atomicity Checker for Java

- Leverage Race Condition Checker to check that protecting lock is held when variables accessed
- Found several atomicity violations
  - java.lang.StringBuffer
  - java.lang.String
  - java.net.URL

Experience with Atomicity Checker

Class	Size (lines)	Annotations per KLOC					
		total	guard	req.	atomic	array	esc.
Inflater	296	20	17	0	3	0	0
Deflater	364	25	20	0	5	0	0
PrintWriter	557	36	5	0	25	0	5
Vector	1029	14	3	1	4	3	3
URL	1269	33	10	1	10	0	13
StringBuffer	1272	19	2	4	5	7	1
String	2399	22	0	0	1	19	1
Total	**7366**	**24**	**8**	**1**	**8**	**4**	**3**

~~“String buffers are safe for use by multiple threads. The methods are synchronized so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.”~~

“String buffers are atomic”

Part 2: Summarizing procedures

Summarization for sequential programs

- Procedure summarization (Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95) is the key to efficiency

```
int x;

void incr_by_2() {
    x++;
    x++;
}
```

```
void main() {
    …
    x = 0;
    incr_by_2();
    …
    x = 0;
    incr_by_2();
    …
}
```

- Bebop, ESP, Moped, MC, Prefix, …

Assertion checking for sequential programs

- Boolean program with:
  - g = number of global vars
  - m = max. number of local vars in any scope
  - k = size of the CFG of the program
- Complexity is $O(\ k \times 2^{O(g+m)}\ )$, linear in the size of CFG
- Summarization enables termination in the presence of recursion

Assertion checking for concurrent programs

Ramalingam 00:

There is no algorithm for assertion checking of concurrent boolean programs, even with only two threads.

Our contribution

- Precise semi-algorithm for verifying properties of concurrent programs
  - based on model checking
  - procedure summarization for efficiency
- Termination for a large class of concurrent programs with recursion and shared variables
- Generalization of precise interprocedural dataflow analysis for sequential programs

What is a summary in sequential programs?

- Summary of a procedure P = Set of all (pre-state → post-state) pairs obtained by invocations of P

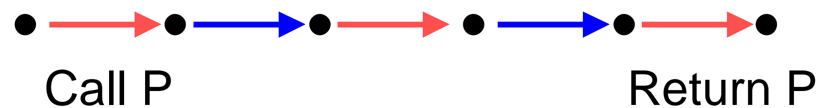
```
int x;

void incr_by_2() {
    x++;
    x++;
}
```

```
void main() {
    …
    x = 0;
    incr_by_2();
    …
    x = 0;
    incr_by_2();
    …
    x = 1;
    incr_by_2();
    …
}
```

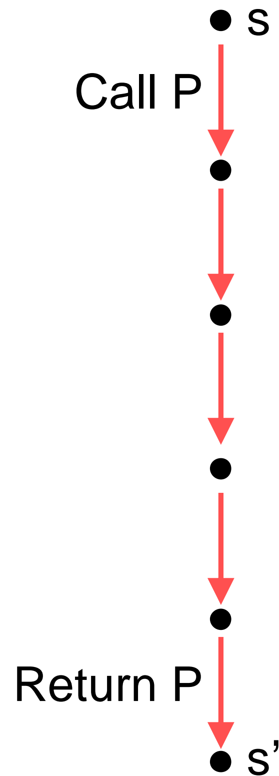
x	→	x'
0	→	2
1	→	3

What is a summary in concurrent programs?

- Unarticulated so far
- Naïve extension of summaries for sequential programs do not work



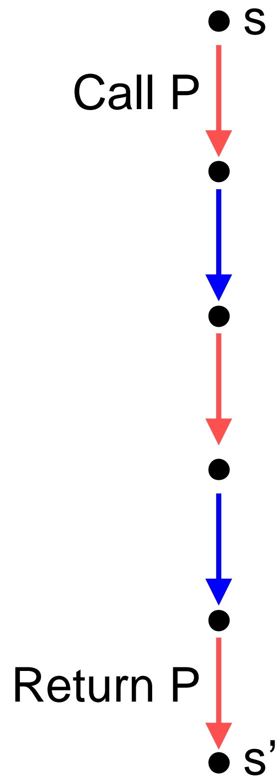
Attempt 1



Advantage: summary computable as in a sequential program

Disadvantage: summary not usable for executions with interference from other threads

Attempt 2



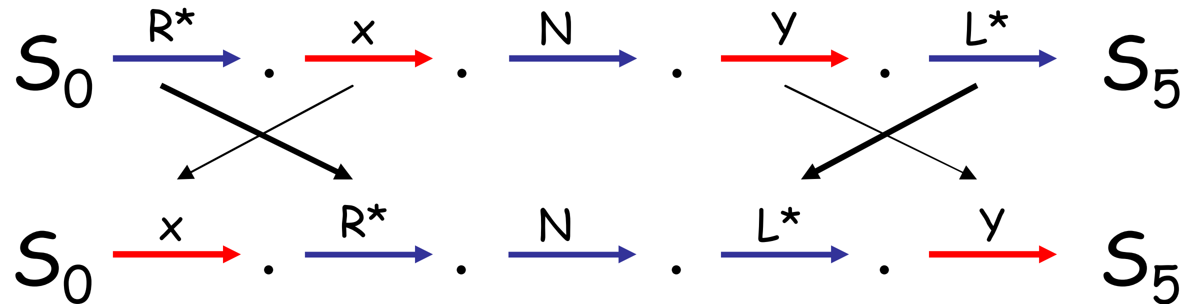
Advantage: Captures all executions

Disadvantage: s and s' must comprise full program state

- summaries are complicated
- do not offer much reuse

Transaction

Lipton: any sequence (R+B)*; (N+ε) ; (L+B)* is a transaction



Other threads need not be scheduled in the middle of a transaction

⇒ Transactions may be summarized

If a procedure body is a single transaction, summarize as in a sequential program

```
bool available[N];
mutex m;

int getResource() {
      int i = 0;
L0:   acquire(m);
L1:   while (i < N) {
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m);
L5:       return i;
        }
L6:     i++;
      }
L7:   release(m);
L8:   return i;
}
```

Choose N = 2

Summaries:

⟨ m, (a[0],a[1]) ⟩ → ⟨ i', m', (a[0]',a[1]') ⟩

⟨ 0, (0, 0) ⟩ → ⟨ 2, 0, (0,0) ⟩
⟨ 0, (0, 1) ⟩ → ⟨ 1, 0, (0,0) ⟩
⟨ 0, (1, 0) ⟩ → ⟨ 0, 0, (0,0) ⟩
⟨ 0, (1, 1) ⟩ → ⟨ 0, 0, (0,1) ⟩

Transactional procedures

- In the Atomizer benchmarks (Flanagan-Freund 04), a majority of procedures are transactional

What if a procedure body comprises multiple transactions?

```
bool available[N];
mutex m[N];

int getResource() {
      int i = 0;
L0:   while (i < N) {
L1:     acquire(m[i]);
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m[i]);
L5:       return i;
        } else {
L6:       release(m[i]);
        }
L7:     i++;
      }
L8:   return i;
}
```

Choose N = 2

Summaries:

⟨ pc,i,(m[0],m[1]),(a[0],a[1]) ⟩ → ⟨ pc',i',(m[0]',m[1]'),(a[0]',a[1]') ⟩

⟨ L0, 0, (0,*), (0,*) ⟩ → ⟨ L1, 1, (0,*), (0,*) ⟩

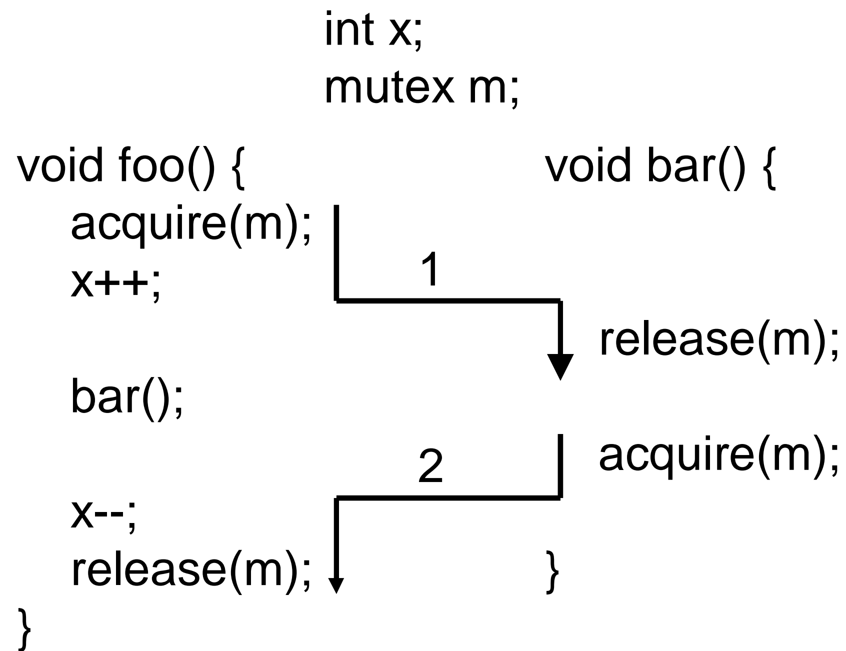
⟨ L0, 0, (0,*), (1,*) ⟩ → ⟨ L5, 0, (0,*), (0,*) ⟩

⟨ L1, 1, (*,0), (*,0) ⟩ → ⟨ L8, 2, (*,0), (*,0) ⟩

⟨ L1, 1, (*,0), (*,1) ⟩ → ⟨ L5, 1, (*,0), (*,0) ⟩

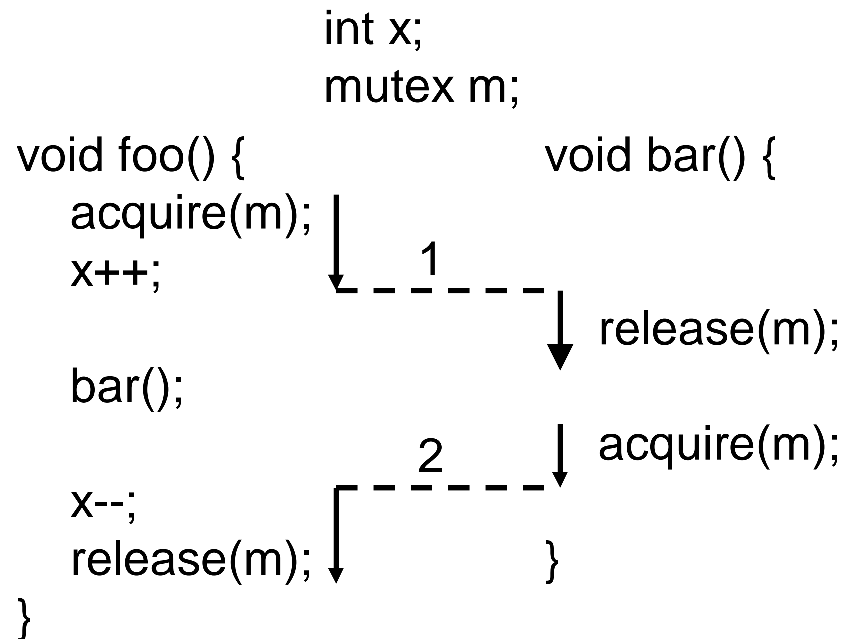
What if a transaction

1. starts in caller and ends in callee?
2. starts in callee and ends in caller?



What if a transaction

1. starts in caller and ends in callee?
2. starts in callee and ends in caller?



Solution:

1. Split the summary into pieces
2. Annotate each piece to indicate whether transaction continues past it

Two-level model checking

- Top level performs state exploration
- Bottom level performs summarization
- Top level uses summaries to explore reduced set of interleavings
  - Maintains a stack for each thread
  - Pushes a stack frame if annotated summary edge ends in a call
  - Pops a stack frame if annotated summary edge ends in a return

Termination

- Theorem:
  - If all recursive functions are transactional, then our algorithm terminates.
  - The algorithm reports an error iff there is an error in the program.

Concurrency + recursion

```
int g = 0;
mutex m;

void foo(int r) {
L0:   if (r == 0) {
L1:       foo(r);
      } else {
L2:       acquire(m);
L3:       g++;
L4:       release(m);
      }
L5:   return;
}

void main() {
      int q =
      choose({0,1});
M0:   foo(q);
M1:   acquire(m)
M2:   assert(g >= 1);
M3:   release(m);
M4:   return;
}

Prog = main() || main()
```

Summaries for foo:

$\langle$ pc,r,m,g $\rangle \rightarrow \langle$ pc',r',m',g' $\rangle$

$\langle$ L0,1,0,0 $\rangle \rightarrow \langle$ L5,1,0,1 $\rangle$

$\langle$ L0,1,0,1 $\rangle \rightarrow \langle$ L5,1,0,2 $\rangle$

Summary (!)

- Transactions enable summarization
- Identify transactions using the theory of movers
- Transaction boundaries may not coincide with procedure boundaries
  - Two level model checking algorithm
  - Top level maintains a stacks for each thread
  - Bottom level maintains summaries

Sequential programs

- For a sequential program, the whole execution is a transaction
- Algorithm behaves exactly like classic interprocedural dataflow analysis

Related work

- Summarizing sequential programs
  - Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95, Ball-Rajamani 00, Esparza-Schwoon 01
- Concurrency+Procedures
  - Duesterwald-Soffa 91, Dwyer-Clarke 94, Alur-Grosu 00, Esparza-Podelski 00, Bouajjani-Esparza-Touili 02
- Reduction
  - Lipton 75, Freund-Qadeer 03, Flanagan-Qadeer 03, Stoller-Cohen 03, Hatcliff et al. 03

- Model checker for concurrent software
- Joint work with Tony Andrews
- http://www.research.microsoft.com/zing