Analysis of Concurrent Software

Types for Atomicity

Cormac Flanagan
UC Santa Cruz

Stephen N. Freund
Williams College

Shaz Qadeer
Microsoft Research

Race Conditions

```
class Ref {
  int i;
  void inc() {
    int t;
    t = i;
    i = t+1;
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();     // two calls happen
    x.inc();     // in parallel
}
assert x.i == 2;
```

A race condition occurs if

- two threads access a shared variable at the same time
- at least one of those accesses is a write

Lock-Based Synchronization

```
class Ref {
  int i;                // guarded by this
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

- Field guarded by a lock
- Lock acquired before accessing field
- Ensures race freedom

Limitations of Race-Freedom

```
class Ref {
  int i;                    // guarded by this
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

Ref.inc()

- race-free
- behaves correctly in a multithreaded context

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
    }
    synchronized (this) {
      i = t+1;
    }
  }
  ...
}
```

Ref.inc()

- race-free
- behaves incorrectly in a multithreaded context

Race freedom does not prevent errors due to unexpected interactions between threads

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }

  synchronized
  void read() { return i; }
  ...
}
```

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }


  void read() { return i; }
  ...
}
```

Ref.read()

- has a race condition
- behaves correctly in a multithreaded context

Race freedom is not necessary to prevent errors due to unexpected interactions between threads

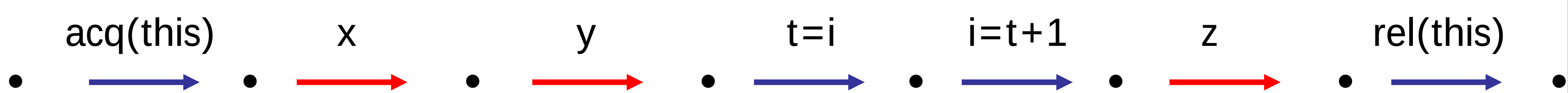
Race-Freedom

- Race-freedom is neither *necessary* nor *sufficient* to ensure the absence of errors due to unexpected interactions between threads

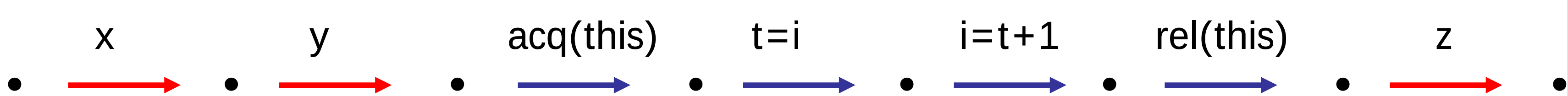
Atomicity

- The method inc() is atomic if concurrent threads do not interfere with its behavior

- Guarantees that for every execution



- there is a *serial* execution with same behavior



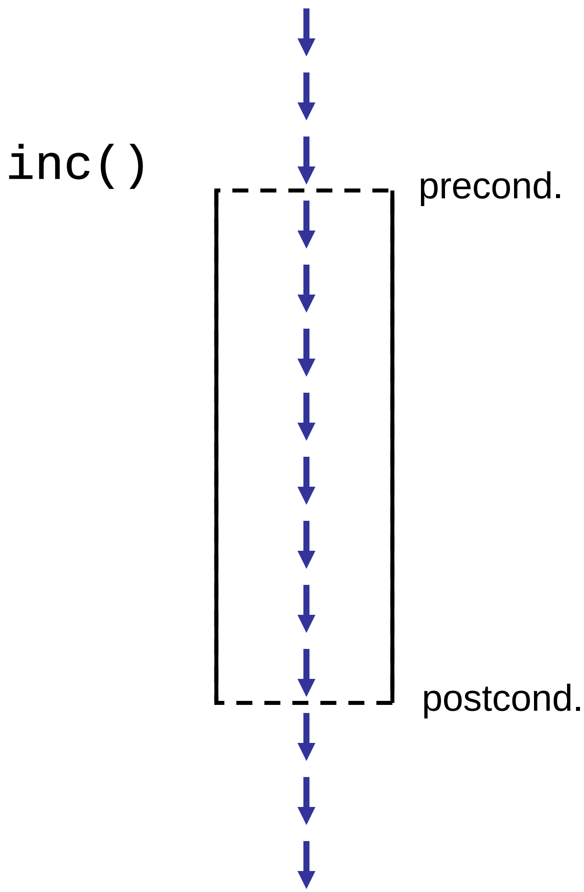
Motivations for Atomicity

1. Stronger property than race freedom

Motivations for Atomicity

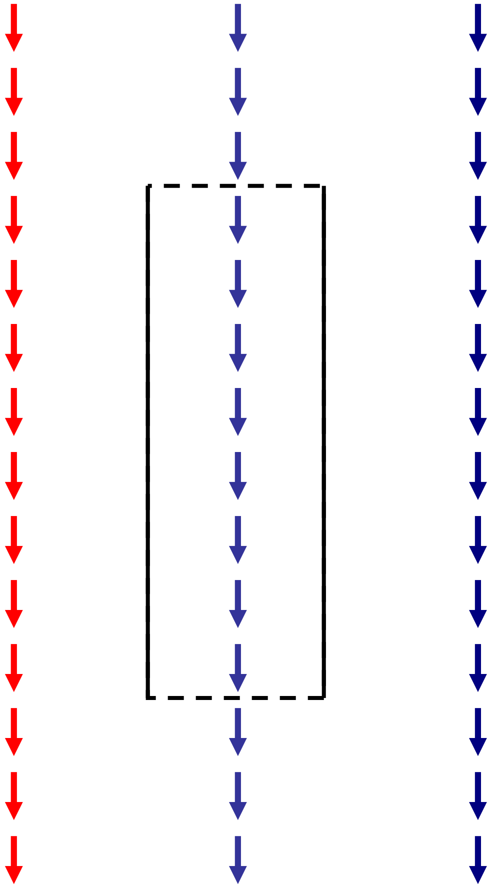
1. Stronger property than race freedom
2. Enables sequential reasoning

Sequential Program Execution



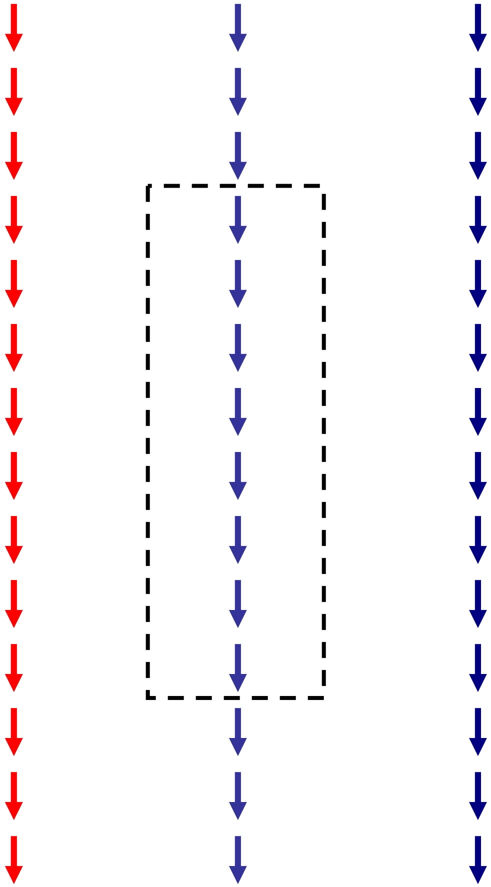

```
void inc() {

   ..


   ..


}
```

Multithreaded Execution



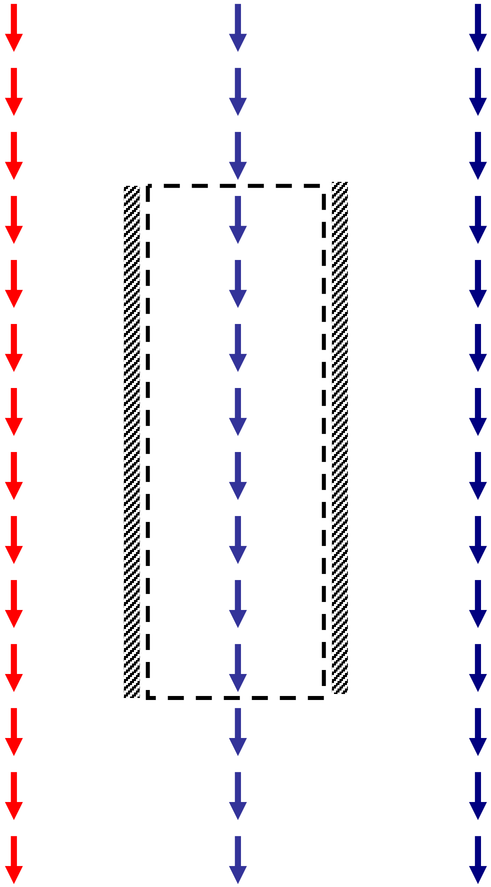
```
void inc() {
  ..
  ..
}
```

Multithreaded Execution



```
void inc() {

   ..


   ..


}
```

Multithreaded Execution



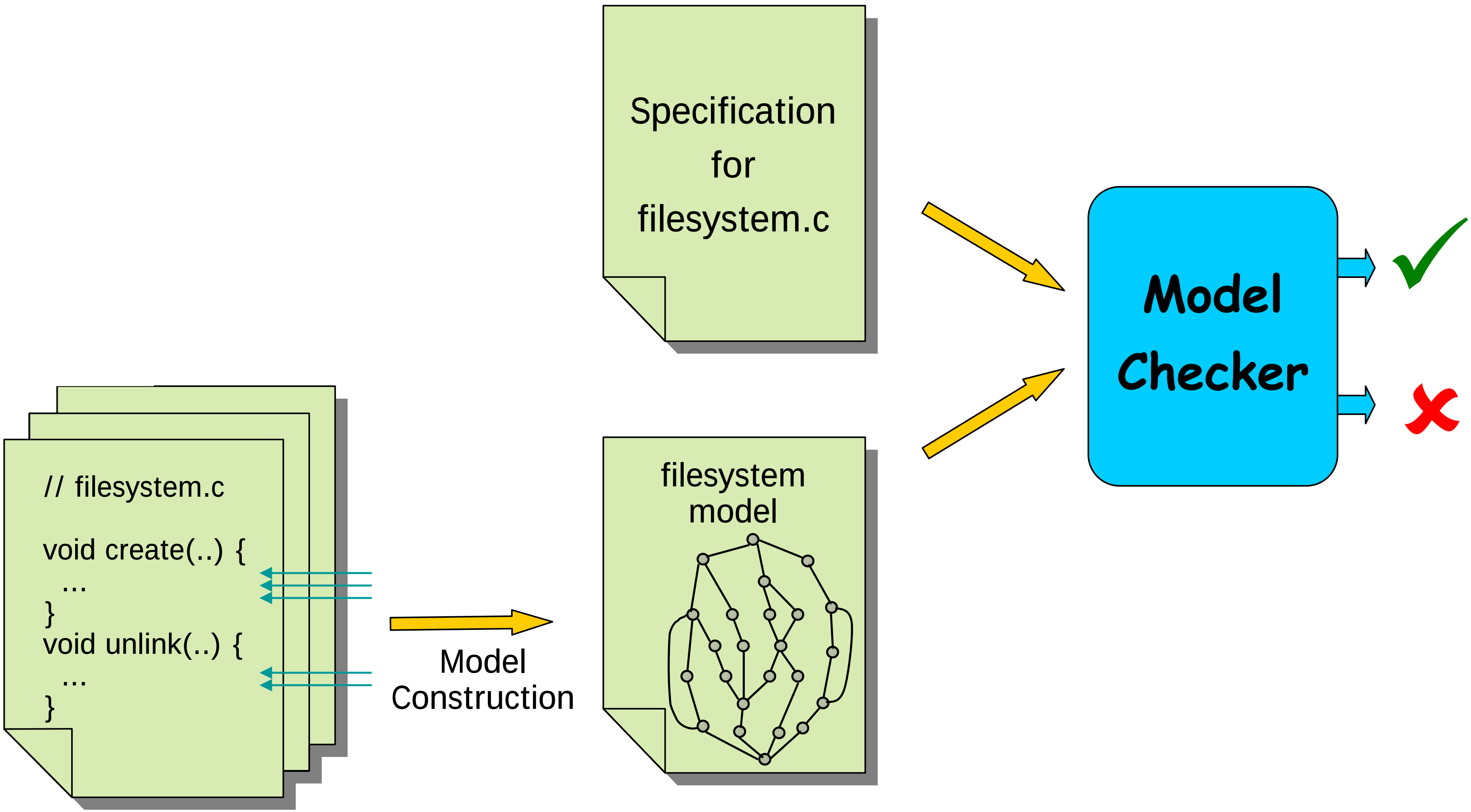
Atomicity

- guarantees concurrent threads do not interfere with atomic method

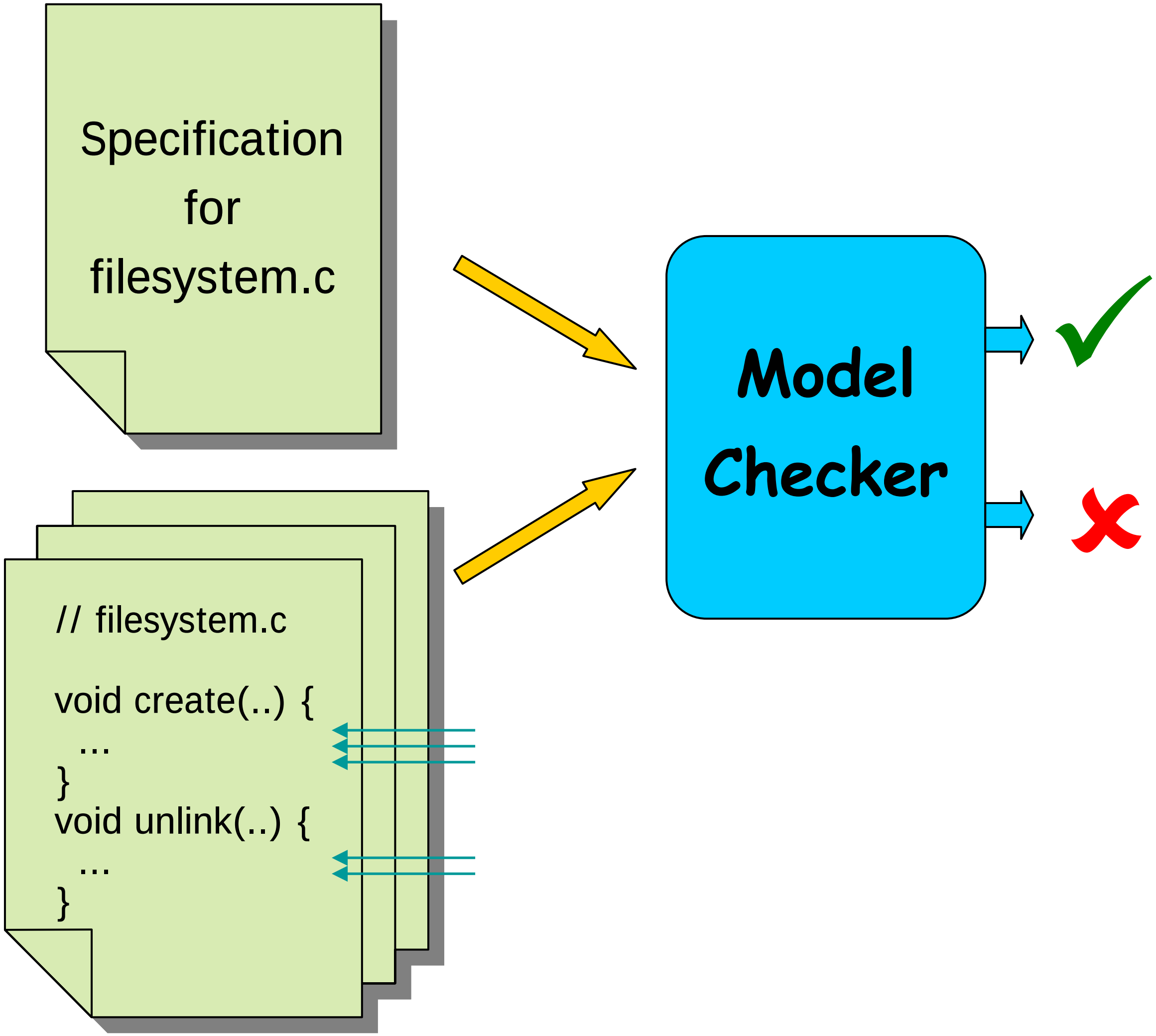
Motivations for Atomicity

1. Stronger property than race freedom
2. Enables sequential reasoning
3. Simple, powerful correctness property

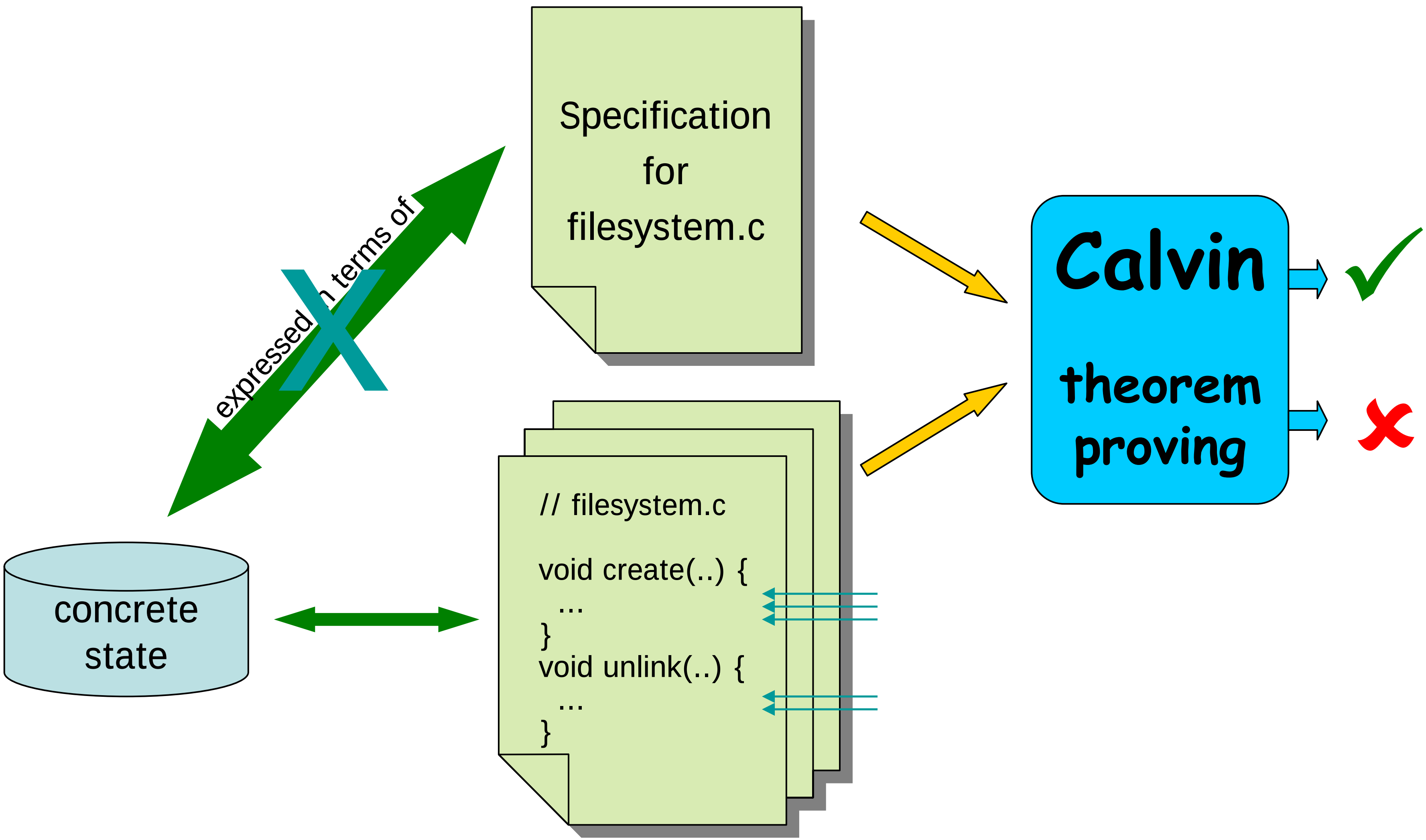
Model Checking of Software *Models*



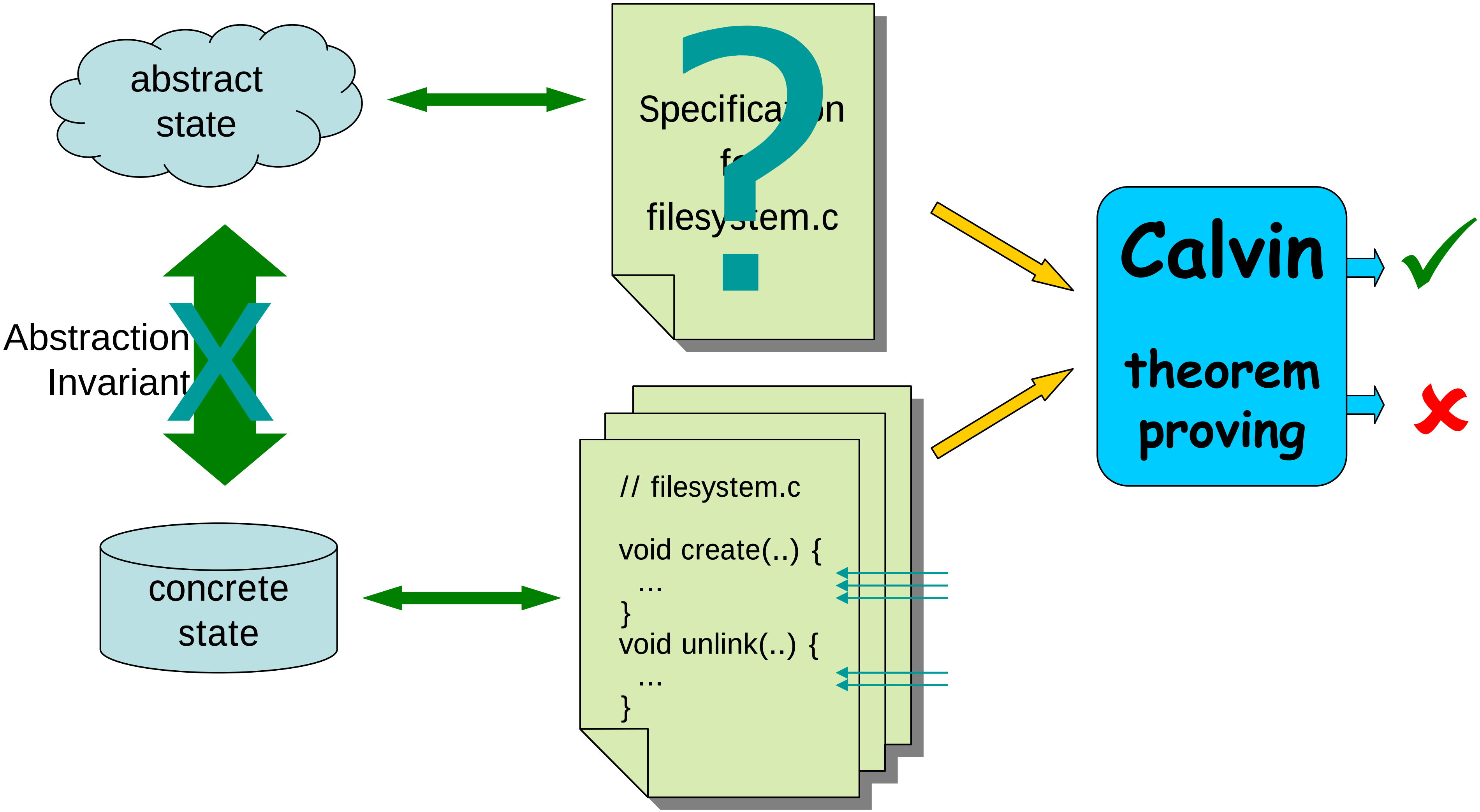
Model Checking of Software



Experience with Calvin Software Checker



Experience with Calvin Software Checker

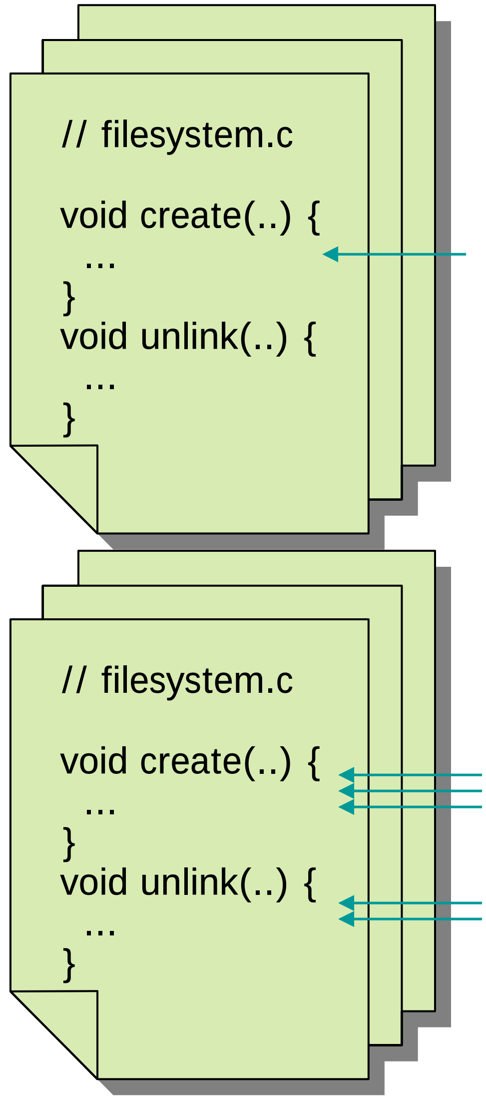


Experience with Calvin Software Checker

```
 /*@ global_invariant (\forall int i; inodeLocks[i] == null ==>
                        0 <= inodeBlocknos[i] && inodeBlocknos[i] < Daisy.MAXBLOCK) */
//@ requires 0 <= inodenum && inodenum < Daisy.MAXINODE;
//@ requires i != null
//@ requires DaisyLock.inodeLocks[inodenum] == \tid
//@ modifies i.blockno, i.size, i.used, i.inodenum
//@ ensures i.blockno  == inodeBlocknos[inodenum]
//@ ensures i.size     == inodeSizes[inodenum]
//@ ensures i.used     == inodeUsed[inodenum]
//@ ensures i.inodenum == inodenum
//@ ensures 0 <= i.blockno && i.blockno < Daisy.MAXBLOCK

static void readi(long inodenum, Inode i) {
        i.blockno = Petal.readLong(STARTINODEAREA + (inodenum * Daisy.INODESIZE));
        i.size    = Petal.readLong(STARTINODEAREA +  (inodenum * Daisy.INODESIZE) + 8);
        i.used    = Petal.read(STARTINODEAREA + (inodenum * Daisy.INODESIZE) + 16) == 1;
        i.inodenum = inodenum;
        // read the right bytes, put in inode
}
```

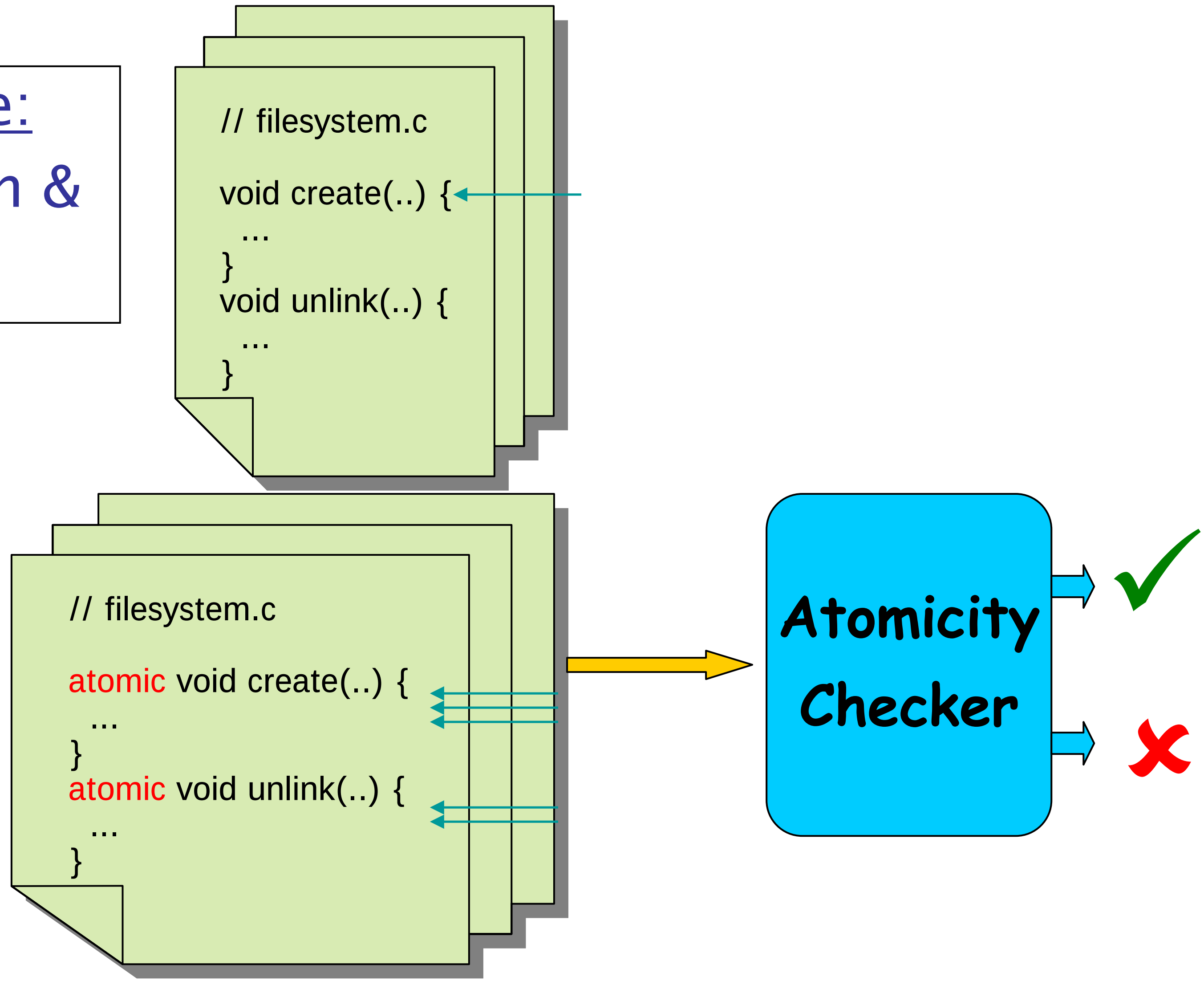
The Need for Atomicity

<u>Sequential case:</u>
code inspection & testing mostly ok



The Need for Atomicity

Sequential case:
code inspection & testing ok



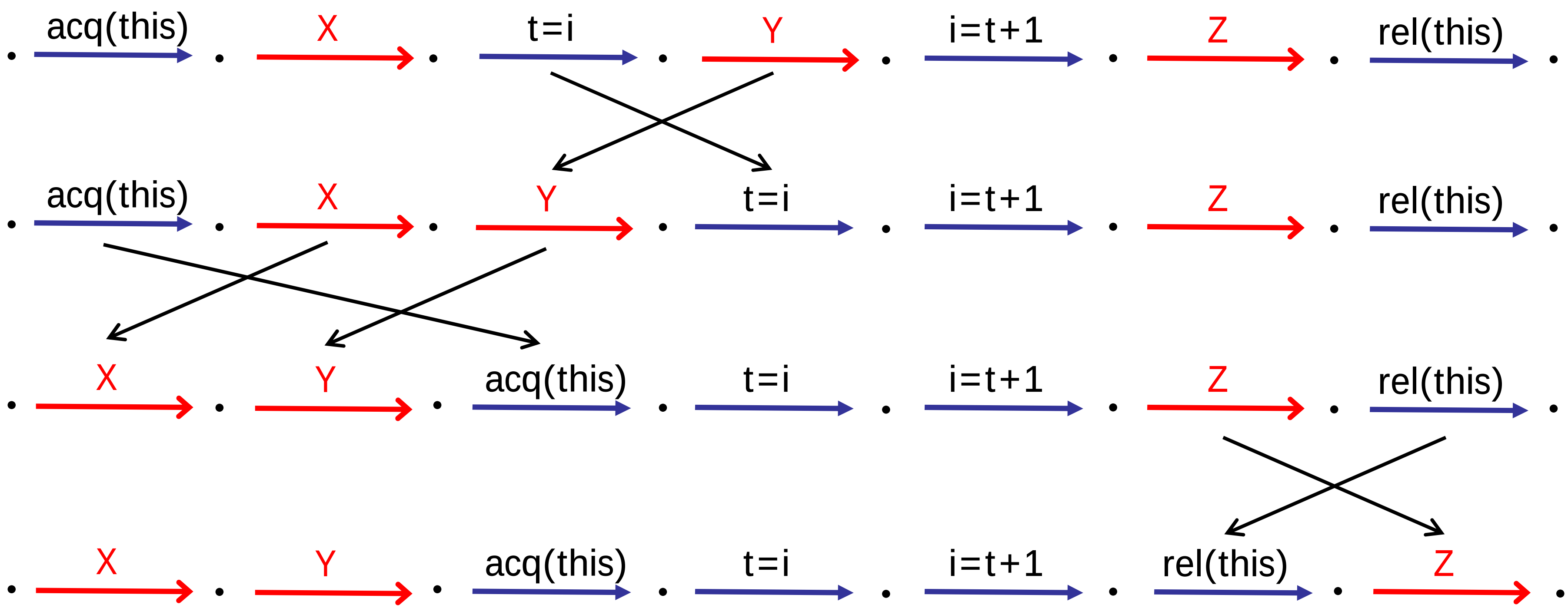
Motivations for Atomicity

1. Stronger property than race freedom
2. Enables sequential reasoning
3. Simple, powerful correctness property

Atomicity

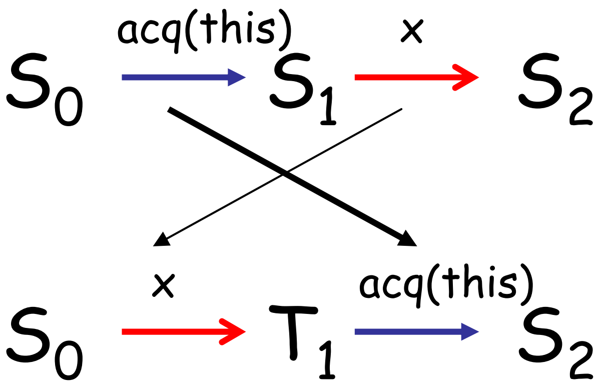
- Canonical property
  - (cmp. serializability, linearizability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (80%+) are atomic
  - many interfaces described as "thread-safe"
- Can verify atomicity statically or dynamically
  - atomicity violations often indicate errors
  - leverages Lipton's theory of reduction

Reduction [Lipton 75]



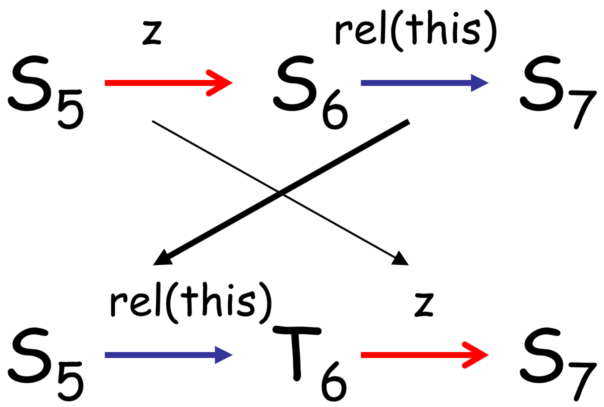
Movers

- right-mover
  - lock acquire



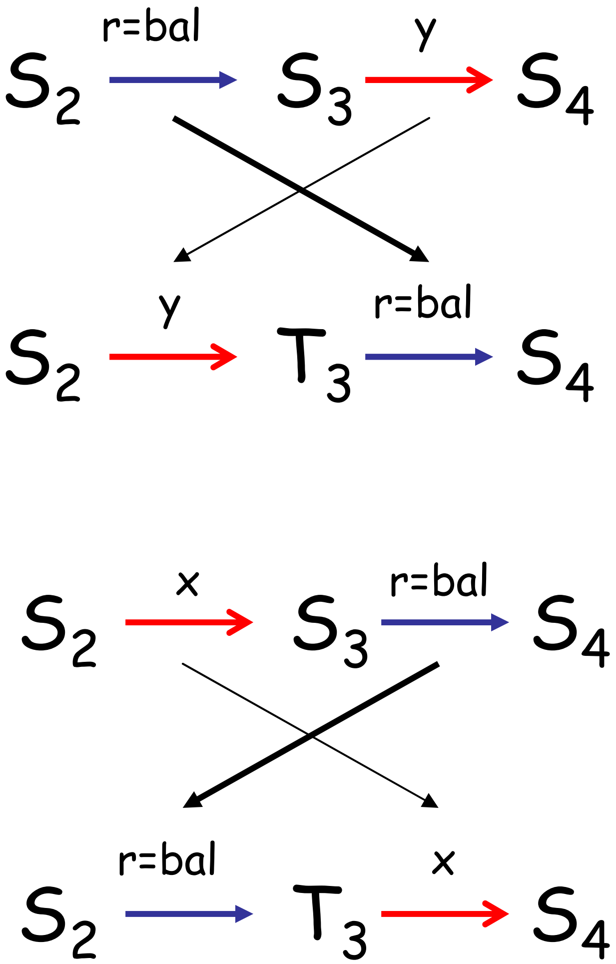
Movers

- right-mover
  - lock acquire
- left-mover
  - lock release



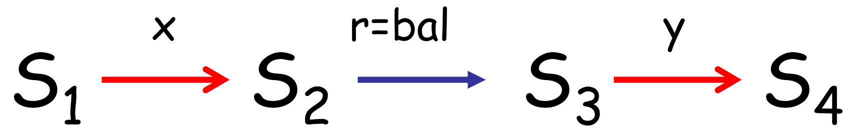
Movers

- right-mover
  - lock acquire
- left-mover
  - lock acquire
- both-mover
  - race-free field access



Movers

- right-mover
  - lock acquire
- left-mover
  - lock acquire
- both-mover
  - race-free field access
- non-mover (atomic)
  - access to "racy" fields



Code Classification

right: lock acquire
left: lock release
(both) mover: race-free variable access
atomic: conflicting variable access

acq(this) j=bal bal=j+n rel(this)

right mover mover left

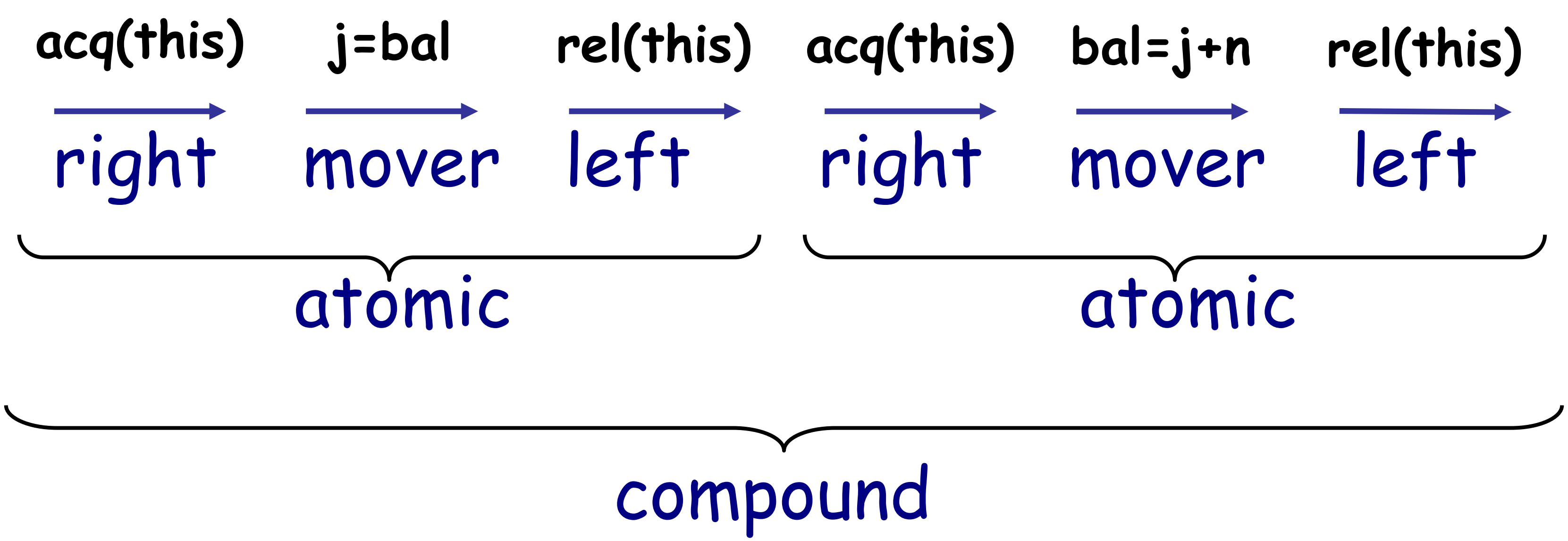
atomic

- composition rules:

right; mover = right right; left = atomic
right; atomic = atomic atomic; atomic = cmpd

Composing Atomicities

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



Conditional Atomicity

```
atomic void deposit(int n) {
  synchronized(this) {     right  ⎫
    int j = bal;           mover  ⎪
    bal = j + n;           mover  ⎬ atomic
  }                        left   ⎪
}                                 ⎭

X atomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);            atomic
    deposit(n);            atomic
  }
}
```

Conditional Atomicity

```
atomic void deposit(int n) {
  synchronized(this) {    right
    int j = bal;          mover
    bal = j + n;          mover
  }                       left
}
```

(right, mover, mover, left) → atomic

if this already held: (mover, mover, mover, mover) → mover

```
atomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);           atomic
    deposit(n);           atomic
  }
}
```

Conditional Atomicity

```
(this ? mover : atomic) void deposit(int n) {
  synchronized(this) {
    int j = bal;
    bal = j + n;
  }
}

atomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);            (this ? mover : atomic)
    deposit(n);            (this ? mover : atomic)
  }
}
```

Conditional Atomicity Details

- In conditional atomicity $(x?b_1:b_2)$, x must be a lock expression (ie, constant)

- Composition rules

  $$a ; (x?b_1:b_2) = x ? (a;b_1) : (a;b_2)$$

java.lang.StringBuffer

```
/**
  ... used by the compiler to implement the binary
  string concatenation operator ...

  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.
*/
public atomic class StringBuffer { ... }
```

FALSE

java.lang.StringBuffer is *not* Atomic!

```
  public atomic StringBuffer {
    private int count guarded_by this;
A   public synchronized int length() { return count; }
A   public synchronized void getChars(...) { ... }

    public synchronized void append(StringBuffer sb){

  ⎧A   int len = sb.length();
  ⎪      ...
C ⎨      ...
  ⎩A   sb.getChars(...,len,...);
         ...
    }
  }
```

sb.length() acquires the lock on sb, gets the length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

- append(...) is *not* atomic

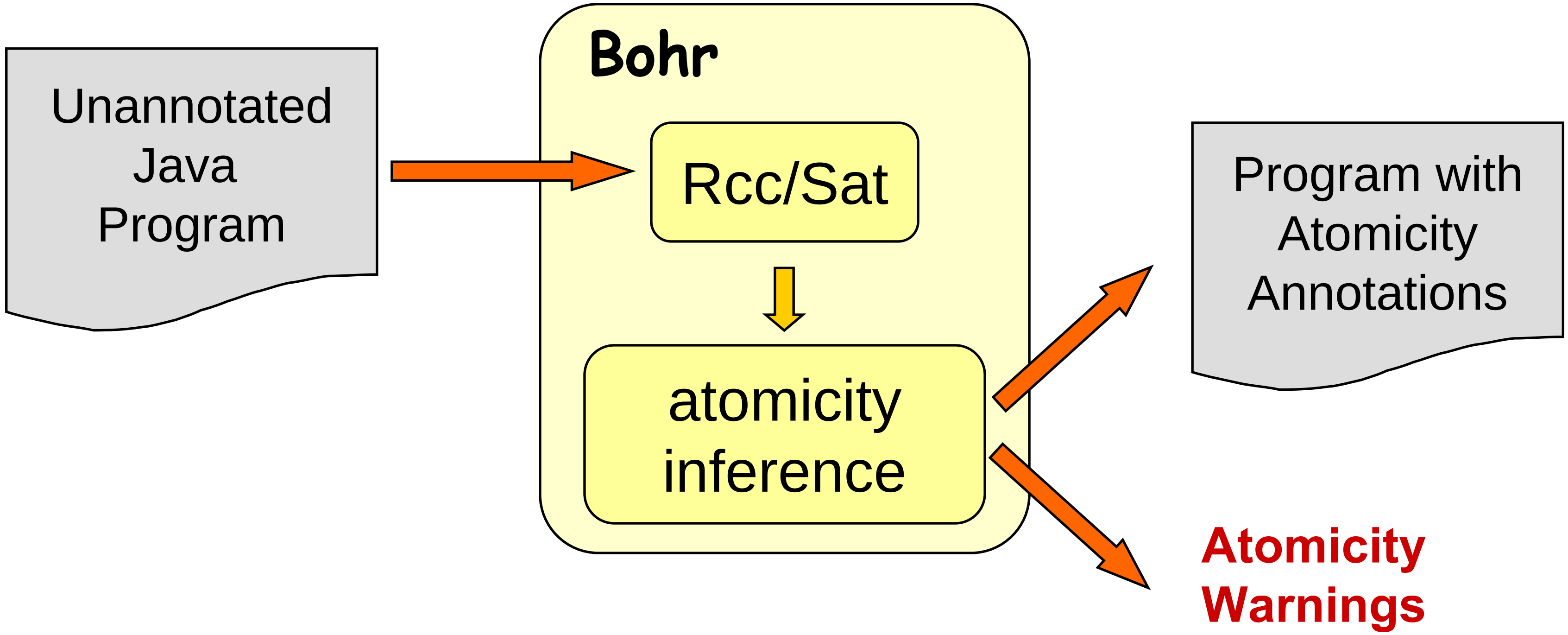
java.lang.Vector

```
interface Collection {
  atomic int length();
  atomic void toArray(Object a[]);
}

class Vector {
  int count;
  Object data[];

X atomic Vector(Collection c) {
    count = c.length();            atomic
    data = new Object[count];      mover
    ...                                      compound
    c.toArray(data);               atomic
  }
}
```

Atomicity Inference

Bohr

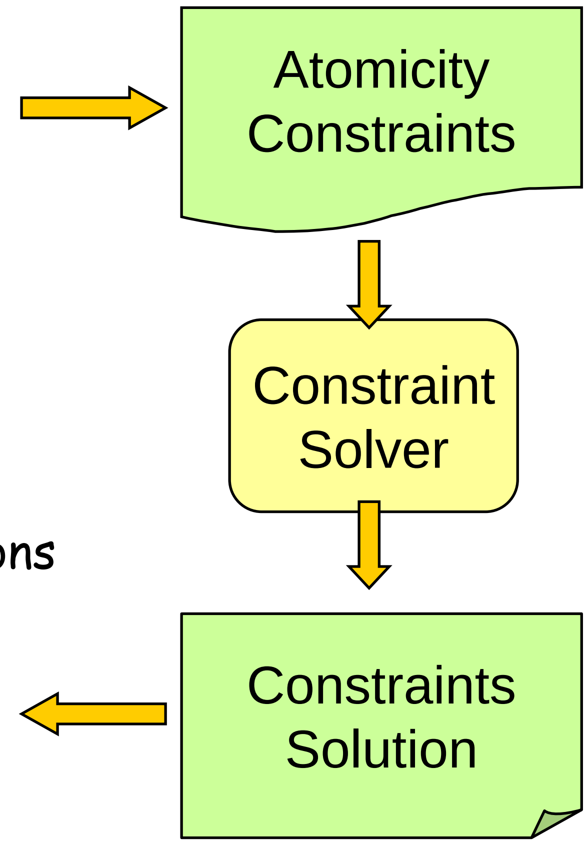
- Type inference for atomicity
  - finds smallest atomicity for each method



Atomicity Inference

Program w/ Locking Annotations

```
class A<ghost x> {
  int f guarded_by this;
  int g guarded_by x;
  void m() {...}
}
```

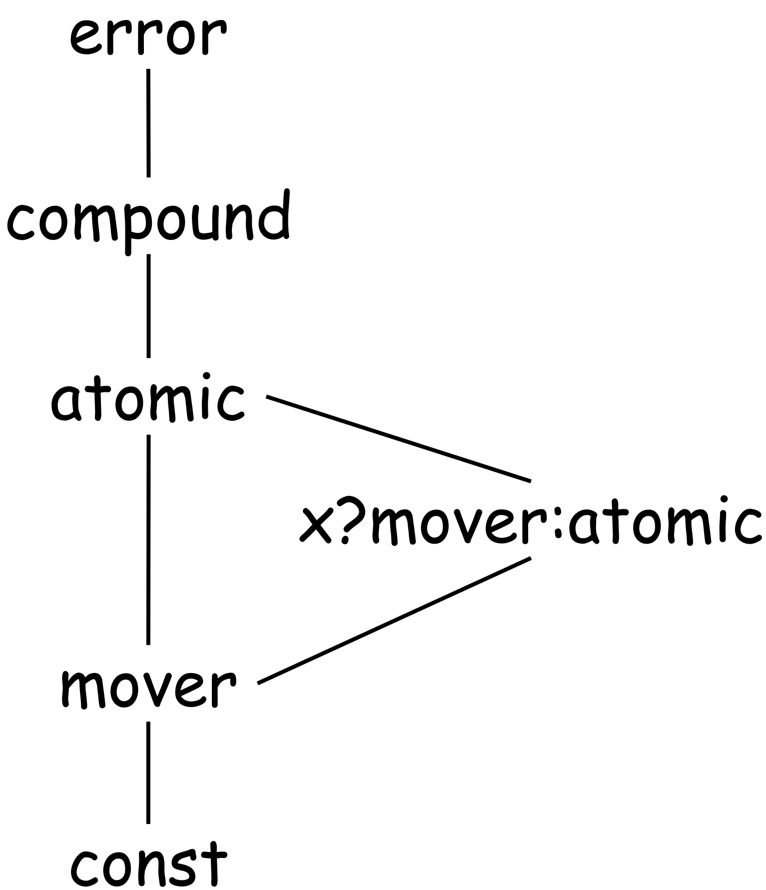


Program w/ Atomicity Annotations

```
class A<ghost x> {
  int f guarded_by this;
  int g guarded_by x;
  atomic void m() {...}
}
```

Atomicity Details

- Partial order of *atomicities*



1. Add atomicity variables

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {



  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

2. Generate constraints over atomicity variables

$$s \leq \alpha_i$$

Atomicity expression

$$
\begin{aligned}
s ::= & \; \text{const} \mid \text{mover} \mid \text{atomic} \\
& \mid \text{cmpd} \mid \text{error} \\
& \mid \alpha \\
& \mid s_1 ; s_2 \\
& \mid x \; ? \; s_1 : s_2 \\
& \mid S(l, s) \\
& \mid WFA(E, s)
\end{aligned}
$$

3. Find assignment A

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;        →   (const; this?mover:error)
      j = this.bal + n;
    }
  }
}

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {



  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

→ ((const; this?mover:error); (const; this?mover:error))

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}
```

→ S(this,
((const; this?mover:error);
(const; this?mover:error)))

S(l,a): atomicity of synchronized(l) { e }
where e has atomicity a

S(l, mover)	=	l ? mover : atomic
S(l, atomic)	=	atomic
S(l, compound)	=	compound
S(l, $l?b_1:b_2$)	=	S(l,b_1)
S(l, $m?b_1:b_2$)	=	m ? S(l,b_1) : S(l,b_2) if $l \neq m$

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error)))

$\leq \alpha_1$

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}
```

$S(this,$
$((const;\ this?mover:error);$
$(const;\ this?mover:error)))$

$\leq \alpha_1$

```
class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

replace **this** with name of receiver

$(const;\ (this?mover:error)[this:=c])$

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}
```

S(this,
((const; this?mover:error);
(const; this?mover:error)))

$\leq \alpha_1$

```
class Bank {

  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

((const; c?mover:error);
(const; α_1[this := c]))

Delayed Substitution

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this,
  ((const; this?mover:error);
  (const; this?mover:error)))

$\leq \alpha_1$

S(c,
  ((const; c?mover:error);
  (const; α_1[this := c])))

$\leq \alpha_2$

Delayed Substitutions

- Given α[x := e]
  - suppose α becomes (x?mover:atomic) and e does not have const atomicity
  - then (e?mover:atomic) is not valid

- WFA(E, b) = smallest atomicity b' where
  - b ≤ b'
  - b' is well-typed and constant in E

- WFA(E, (e?mover:atomic)) = atomic

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}
```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error)))

$\leq \alpha_1$

```
class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α_1[this:=c]))))

$\leq \alpha_2$

3. Compute Least Fixed Point

- Initial assignment A: $\alpha_1 = \alpha_2 = \text{const}$

- Algorithm:
  - pick constraint $s \leq \alpha$ such that $A(s) \not\leq A(\alpha)$
  - set $A(\alpha)$ to $A(\alpha) \sqcup A(s)$
  - repeat until quiescence

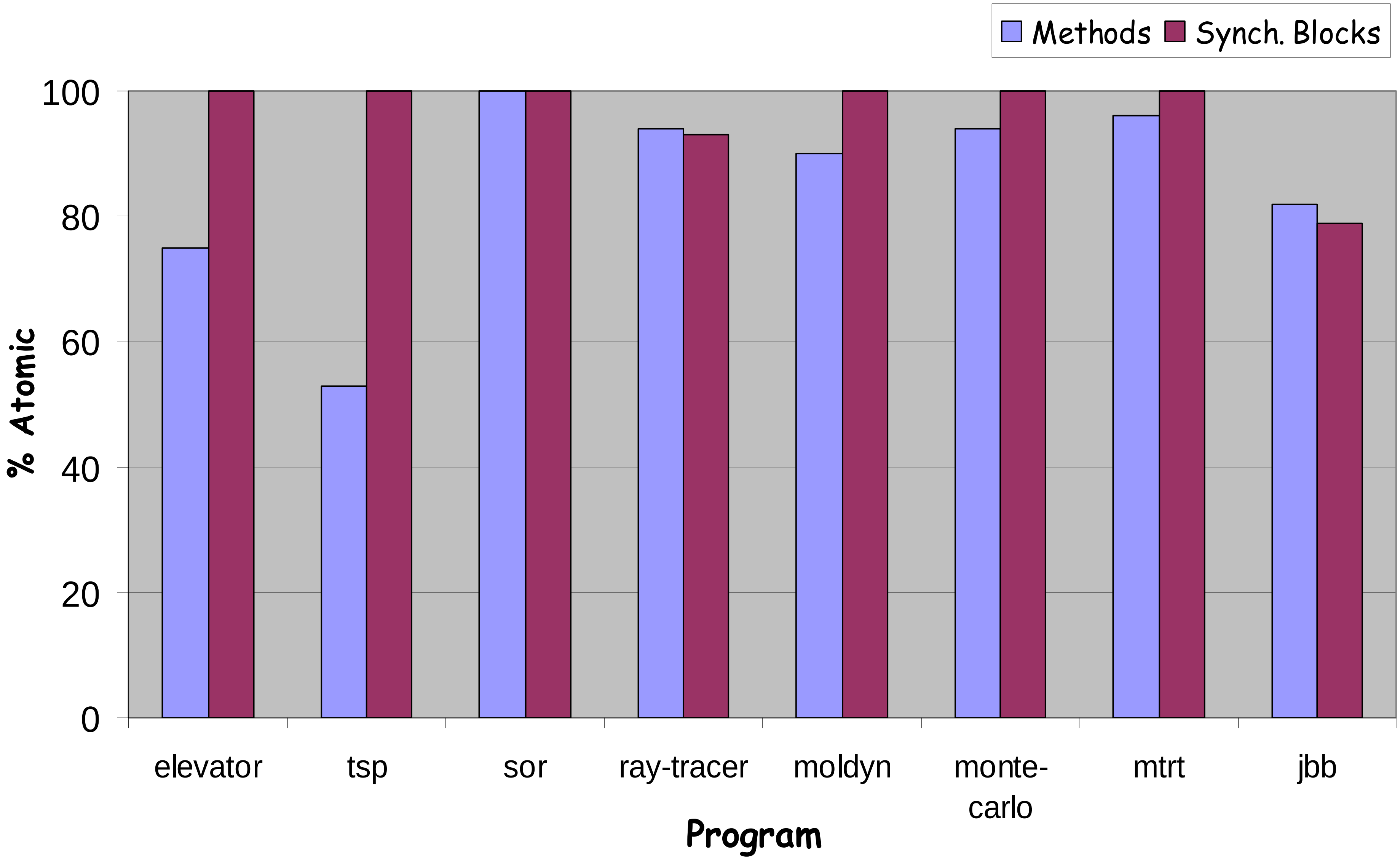
```
class Account {
  int bal guarded_by this;

  (this ? mover : atomic) void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  (c ? mover : atomic) void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

Validation

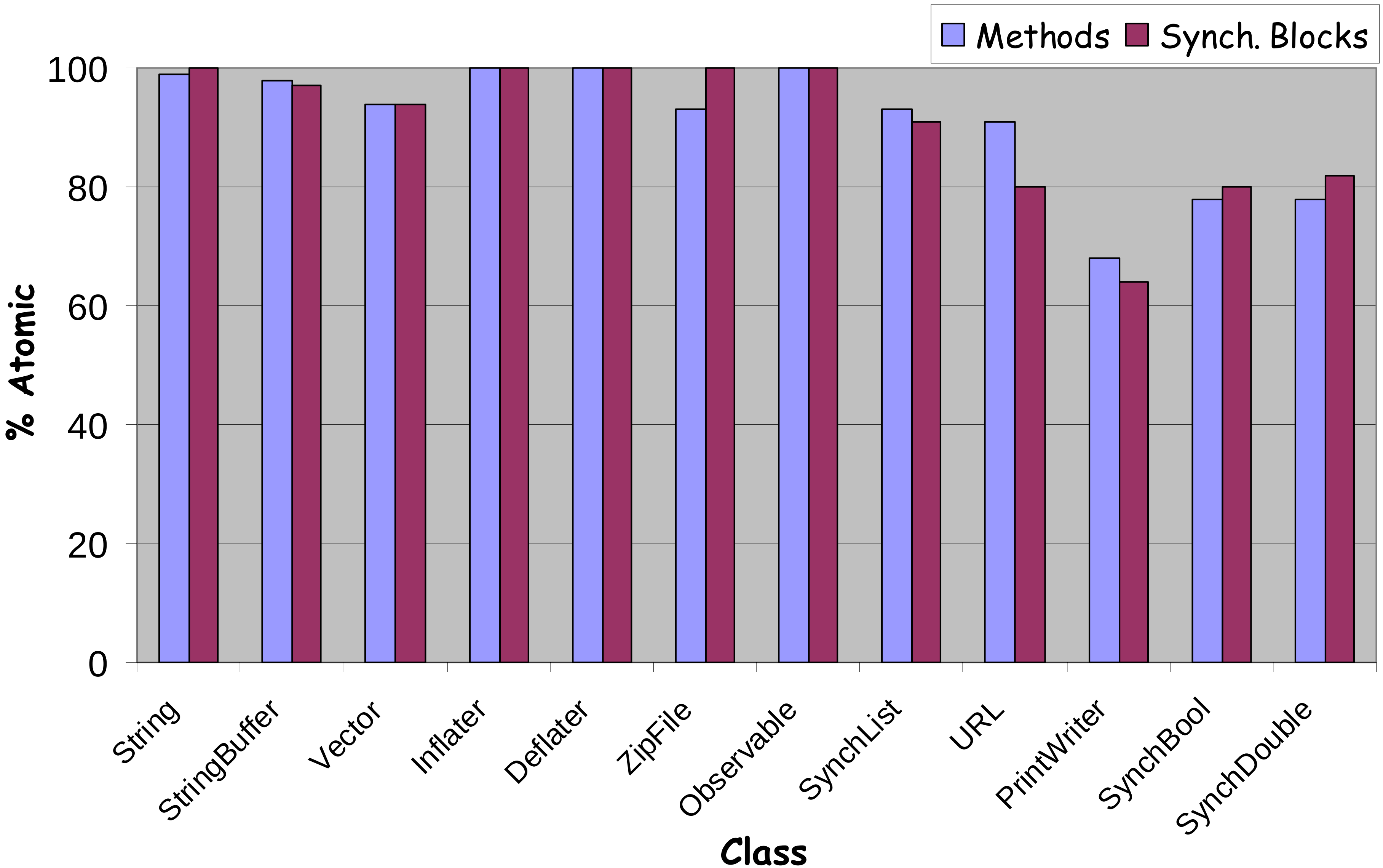
Program	Size (KLOC)	Time (s)	Time (s/KLOC)
elevator	0.5	0.6	1.1
tsp	0.7	1.4	2.0
sor	0.7	0.8	1.2
raytracer	2.0	1.7	0.9
moldyn	1.4	4.9	3.5
montecarlo	3.7	1.5	0.4
mtrt	11.3	7.8	0.7
jbb	30.5	11.2	0.4

(excludes Rcc/Sat time)

Inferred Atomicities



Thread-Safe Classes



Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - other applications:
    - model checking [Stoller-Cohen 03, Flanagan-Qadeer 03]
    - dynamic analysis [Flanagan-Freund 04, Wang-Stoller 04]
- Atomicity inference
  - type and effect inference [Talpin-Jouvelot 92,...]
  - dependent types [Cardelli 88]
  - ownership, dynamic [Sastakur-Agarwal-Stoller 04]

Conclusions And Future Directions

- Atomicity a fundamental concept
  - improves over race freedom
  - matches programmer intuition and practice
  - simplifies reasoning about correctness
  - enables concise and trustable documentation

- Many approaches for verifying atomicity
  - static type systems
  - dynamic checking (tomorrow)
  - ... hybrid checkers ...
  - ... model checkers ...