

# Lecture 7b

## Atomizer: A Dynamic Atomicity Checker

# 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();  
        ...  
        ...  
        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

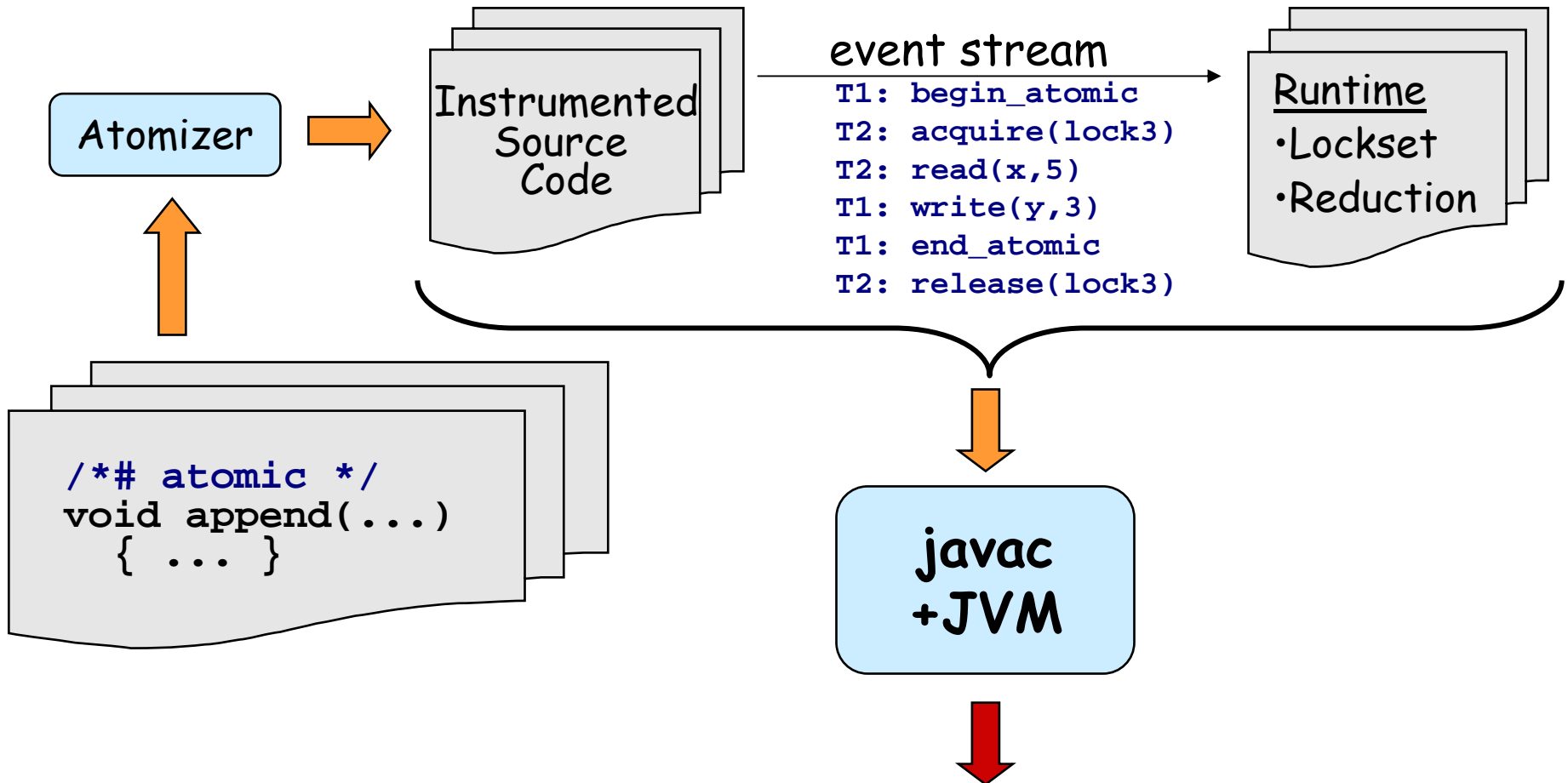
# The Atomizer

- Annotate program with atomicity requirements

```
/*# atomic */ synchronized StringBuffer append(...) ...  
/*# atomic */ synchronized int length() ...  
/*# atomic */ synchronized int getChars(...) ...
```

- Atomizer checks for violations at **run time**
- Core analysis
  - Reduction [Lipton 76]
  - *Eraser* [Savage et al. 97]
- Applied to 150,000+ lines of Java code

# Atomizer: Instrumentation Architecture



**Warning: method "append"  
may not be atomic at line 43**

# Lockset Algorithm

- Tracks *lockset* for each field
  - lockset = set of locks held on all accesses
  - empty lockset indicates possible race condition
- Algorithm:
  - On first access to  $o.f$ :
    - $\text{LockSet}(o.f) = \text{Held}(\text{curThread})$
  - On all subsequent accesses to  $o.f$ :
    - $\text{LockSet}(o.f) = \text{LockSet}(o.f) \cap \text{Held}(\text{curThread})$

# Lockset Example

Thread 1

```
synchronized(x) {  
o.f = 3;  
}
```



Thread 2

```
synchronized(y) {  
o.f = 2;  
}
```

- $\text{LockSet}(o.f) = \{ x \}$

# Lockset Example

Thread 1

```
synchronized(x) {  
    o.f = 3;  
}
```



Thread 2

```
synchronized(y) {  
    o.f = 2;  
}
```

- $\text{LockSet}(o.f) = \{x\} \cap \{y\} = \{\}$

# Problems

- This doesn't quite work
- We need to deal with
  - Uninitialized data
  - Read-Shared Data

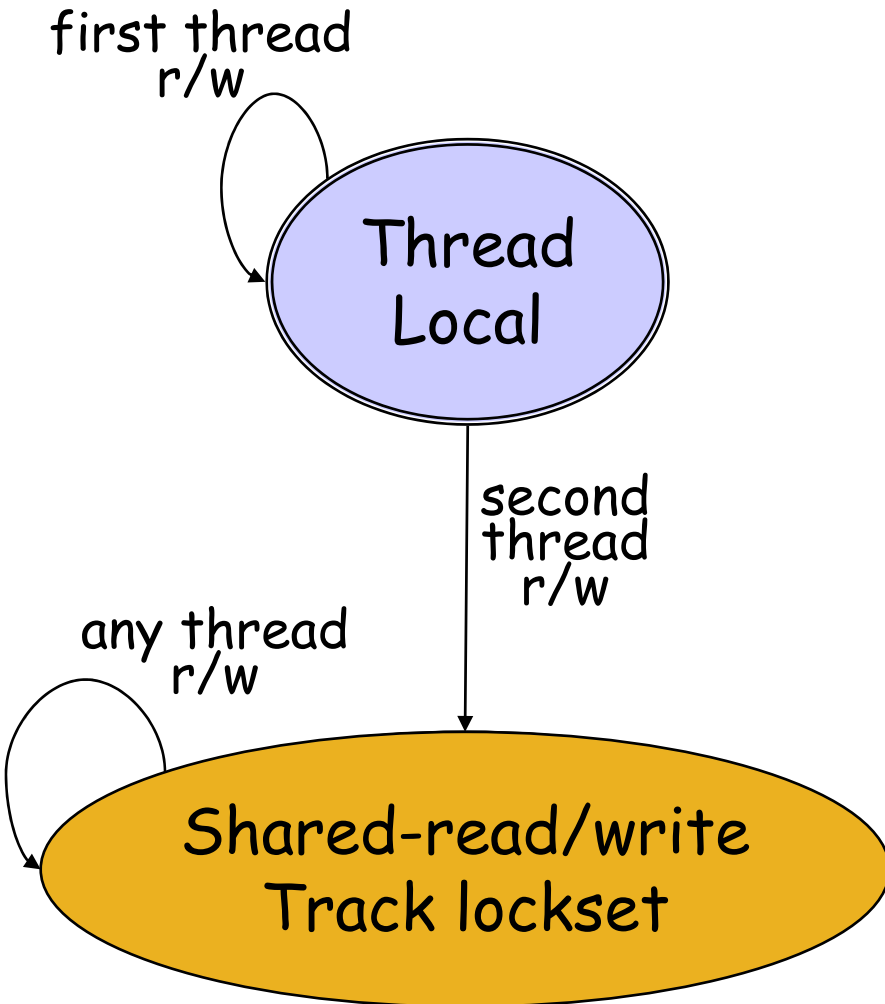
# Uninitialized Data

- Data often initialized by one owner
- No need to lock at this time
- How do we know when initialization is done?
  - Answer: We don't
  - But, we can tell when the value is accessed by a second thread

# Lockset



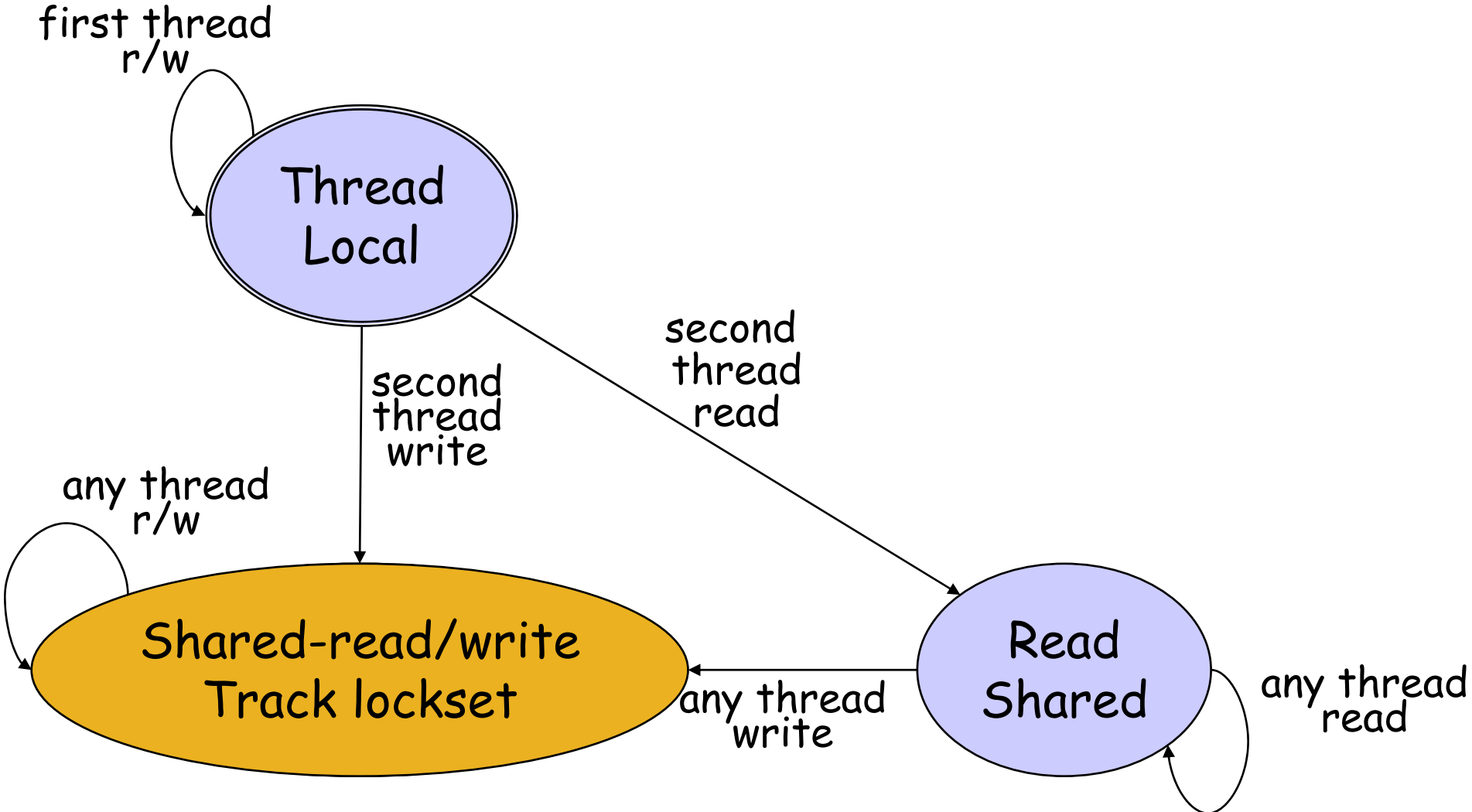
# Extending Lockset (Thread Local Data)



# Read Shared

- Once created, some data is only read
- No need to lock read-only data

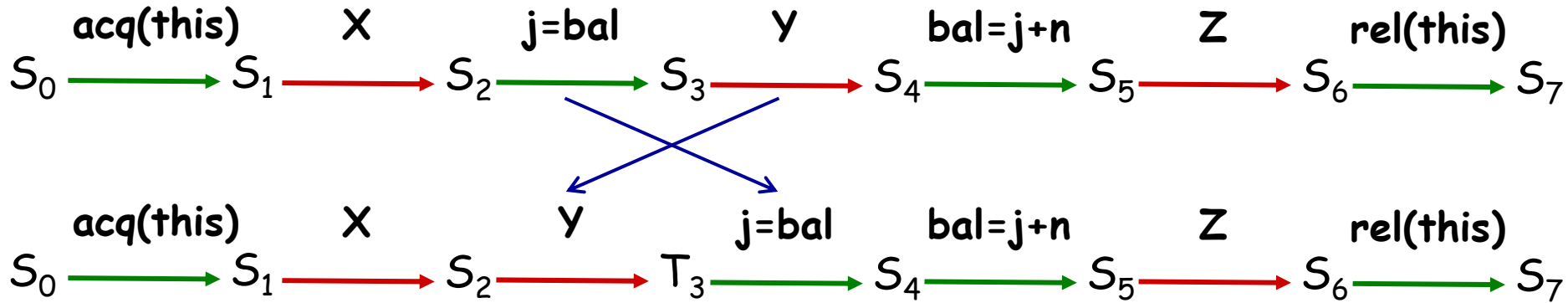
# Extending Lockset (Read Shared Data)



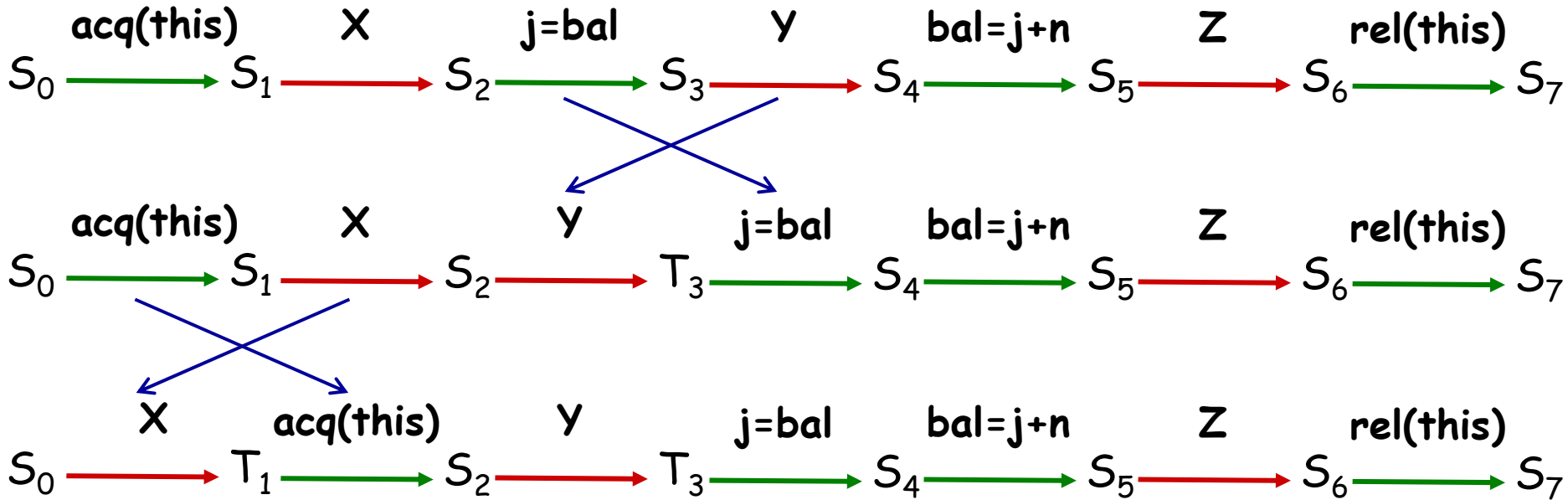
# Reduction [Lipton 75]



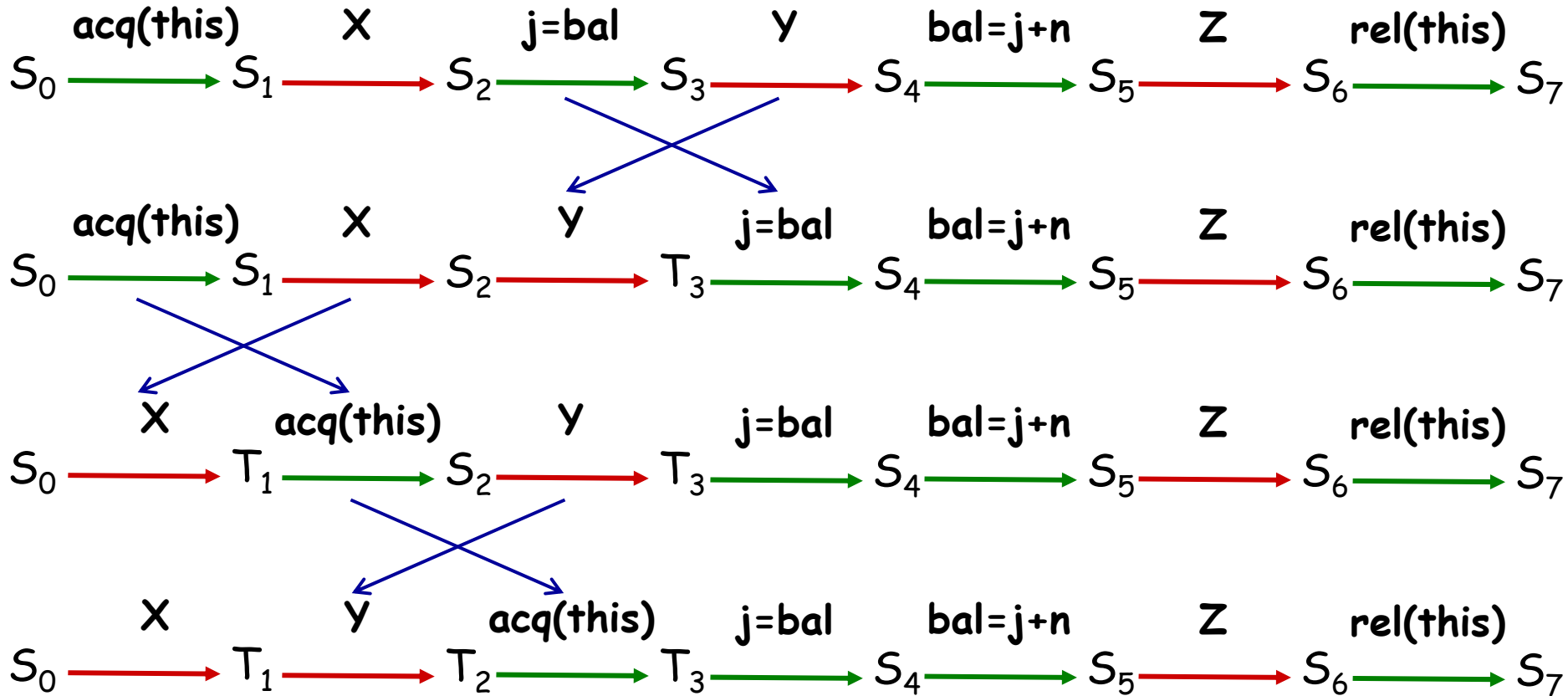
# Reduction [Lipton 75]



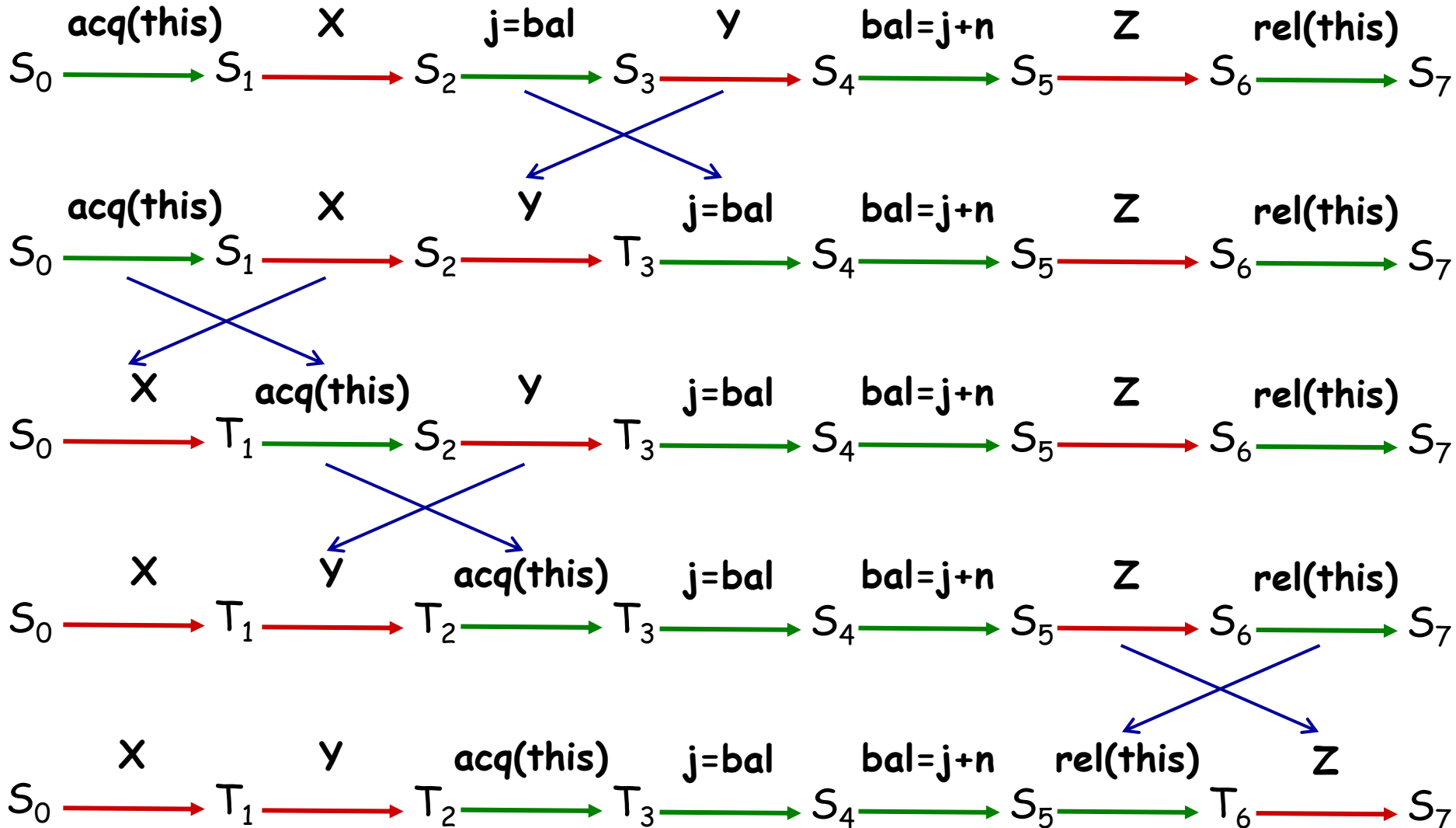
# Reduction [Lipton 75]



# Reduction [Lipton 75]



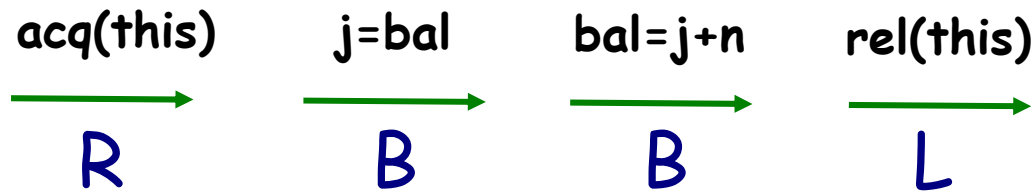
# Reduction [Lipton 75]



# Movers

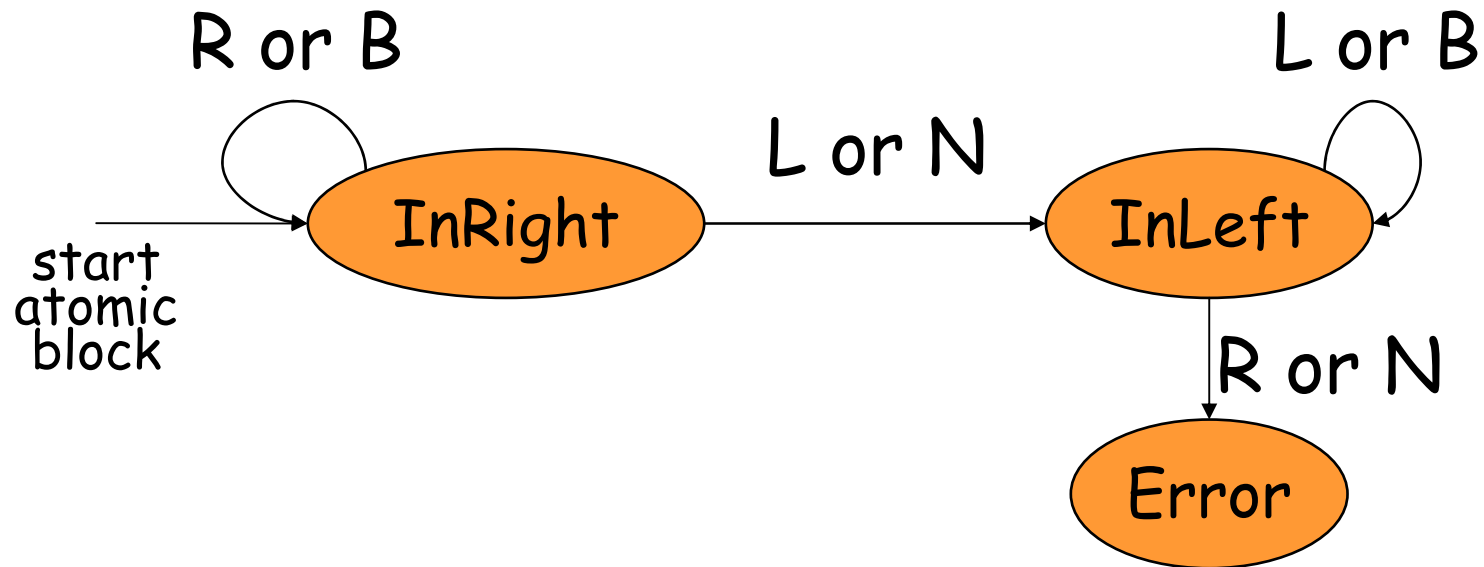
- **R**: right-mover
  - lock acquire
- **L**: left-mover
  - lock release
- **B**: both-mover
  - race-free field access
- **N**: non-mover
  - access to "racy" fields

• Reducible blocks have form  $(R|B)^* [N] (L|B)^*$



# Dynamic Reduction

- Automata to check  $(R|B)^* [N] (L|B)^*$



# java.lang.StringBuffer

```
public class StringBuffer {
    private int count;
    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
    /*# atomic */
    public synchronized void append(StringBuffer sb){

        int len = sb.length();
        ...
        ...
        ...
        sb.getChars(..., len, ...);
        ...
    }
}
```

StringBuffer.append is not atomic:

Start:

at StringBuffer.append(StringBuff  
at Thread1.run(Example.java:17)

Commit: Lock Release

at StringBuffer.length(StringBuff  
at StringBuffer.append(StringBuff  
at Thread1.run(Example.java:17)

Error: Lock Acquire

at StringBuffer.getChars(StringBu  
at StringBuffer.append(StringBuff  
at Thread1.run(Example.java:17)

# Summary

- Race conditions are a *heuristic* for detecting errors
- Atomicity is a deeper, more semantic property
  - sequential reasoning ok on atomic methods
- Can verify atomicity
  - statically (we saw a type system for this)
  - dynamically (by inspecting an event trace)