EECS
Electrical Engineering and
Computer Sciences

BERKELEY PAR LAB

PARALLEL COMPUTING LABORATORY

Semantic Atomicity for Multithreaded Programs

Jacob Burnim, George Necula, Koushik Sen
Parallel Computing Laboratory
University of California, Berkeley

Parallel Correctness is Hard

EECS Electrical Engineering and Computer Sciences

BERKELEY PAR LAB

- Difficult to write correct parallel software.
  - **Key:** Interference between parallel threads.
  - **Atomicity** – freedom from harmful interference; a fundamental parallel correctness property.
- **Today**: **Semantic atomicity**.
  - Specifying atomicity with respect to user-defined, semantic equivalence.
  - Efficiently testing such specifications.
  - **Overall Goal:** Lightweight, useful specs to help programmers find and fix parallelism bugs.

Outline

- Overview + Motivation
- **Background: Atomicity**
- Specifying Semantic Atomicity
- Testing Semantic Atomicity
- Experimental Evaluation
- Conclusion

Background: Atomicity

- Atomicity a **non-interference property**.
  - Block of code is **atomic** if it behaves **as if** executed all-at-once and without interruption.
  - Interference from other threads is **benign** – cannot change overall program behavior.

Background: Atomicity

- Atomicity a **non-interference property**.
  - Block of code is **atomic** if it behaves **as if** executed all-at-once and without interruption.

```
int bal = 0;

deposit(int a) {
  @atomic {
    int t = bal;
    bal = t + a;
  }
}
```

Atomic **specification**.

Programmer **intends** that this code is atomic.

Want to **check** specification.

Is the code actually atomic?

Background: Atomicity

- Atomicity a **non-interference property**.
  - Block of code is **atomic** if it behaves **as if** executed all-at-once and without interruption.

```
int bal = 0;

deposit(int a) {
  @atomic {
    int t = bal;
    bal = t + a;
  }
}
```

Thread 1:

deposit(10)

t = 0

bal = 10

Thread 2:

deposit(5)

t = 0

bal = 5

Atomicity specification does **not** hold.

Background: Atomicity

- Atomicity a **non-interference property**.
  - Block of code is **atomic** if it behaves **as if** executed all-at-once and without interruption.

```
int bal = 0;

deposit(int a) {
  @atomic {
    int t = bal;
    while (!CAS(&bal, t, t+a))
      t = bal;
  }
}
```

With CAS, updates to balance are atomic.

Atomicity specification **does** hold.

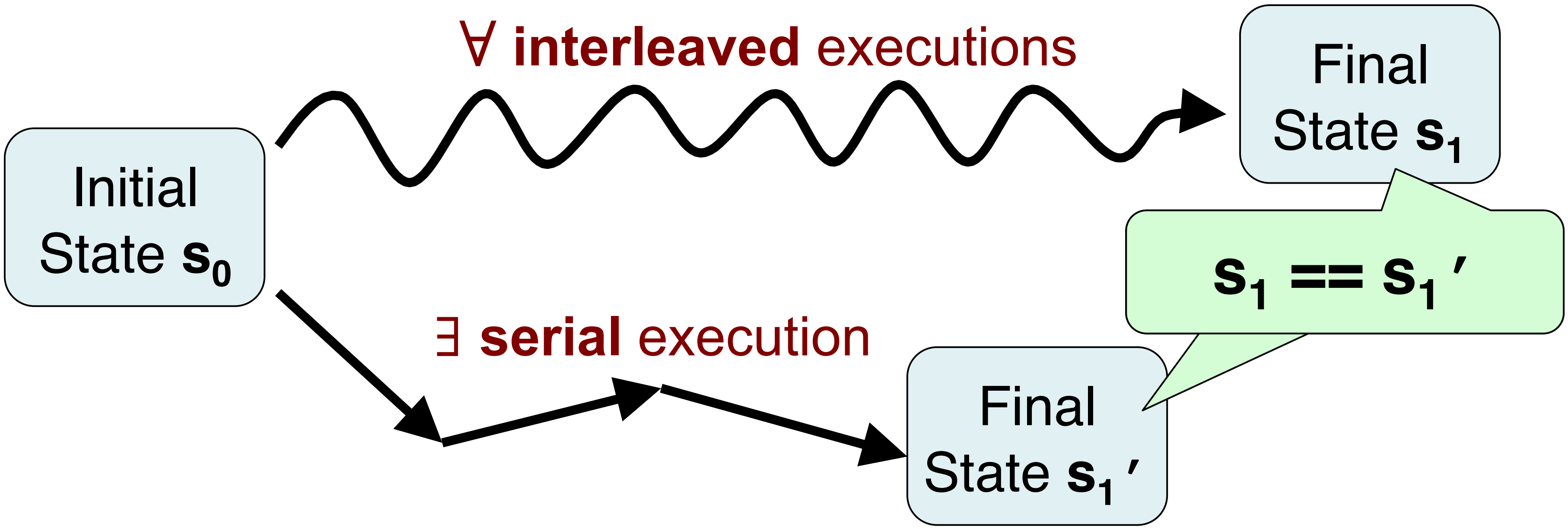
Background: Atomicity

- **Formally:** Two semantics for a program P with specified atomic blocks.
  - **Interleaved:** Threads interleave normally.

  - **Serial:** When one thread opens an atomic block, no other thread runs until it closes.

Background: Atomicity

- Formally, program P is **atomic** iff:
  - **For all interleaved** executions E yielding s_1, there **exists** a **serial** E′ yielding an identical final state.



Outline

- Overview
- Background: Atomicity
- **Specifying Semantic Atomicity**
- Testing Semantic Atomicity
- Experimental Evaluation
- Conclusion

Motivating Example

```
    ConcurrentLinkedQueue q;
    q.add(1);  q.add(1);

Thread 1:                    Thread 2:
 @atomic {                    @atomic {
   q.remove(1);                 q.remove(1);
 }                            }
```

- Michael & Scott non-blocking queue, in the Java standard library
- Internally, a linked list with lazy deletion.

EECS Electrical Engineering and Computer Sciences

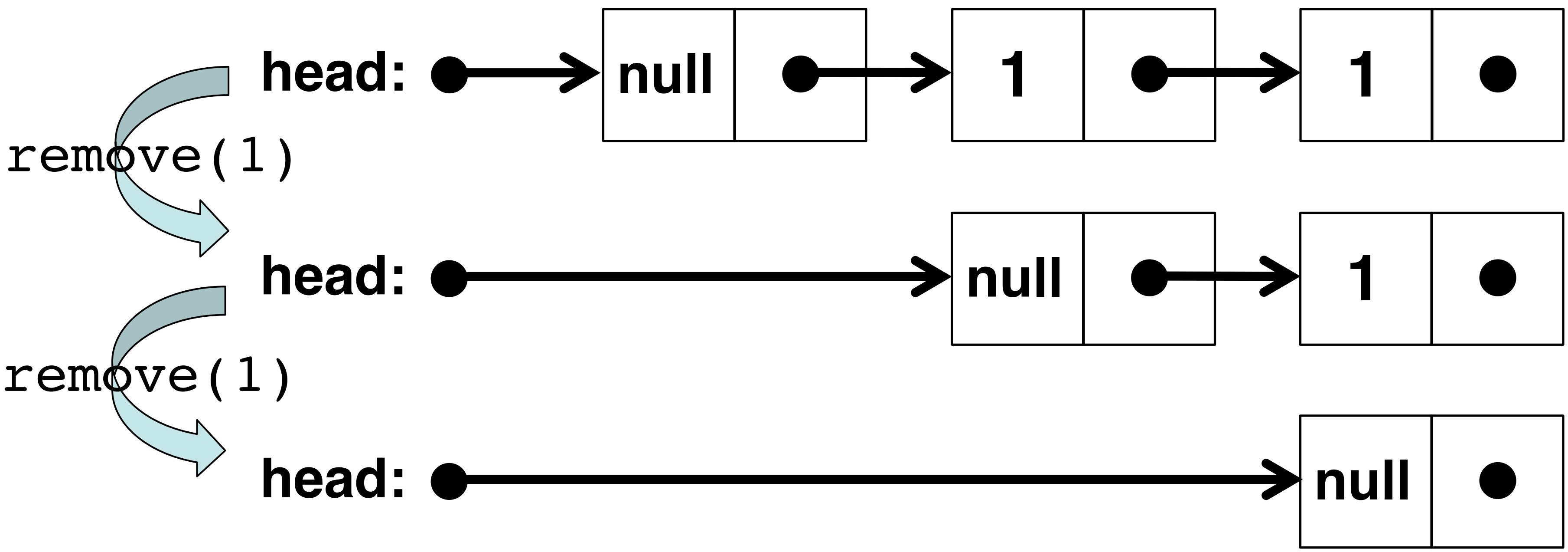
Motivating Example

BERKELEY PAR LAB

```
Thread 1:
 @atomic {
   q.remove(1);
 }
```

```
Thread 2:
 @atomic {
   q.remove(1);
 }
```

- In any **serial** execution:

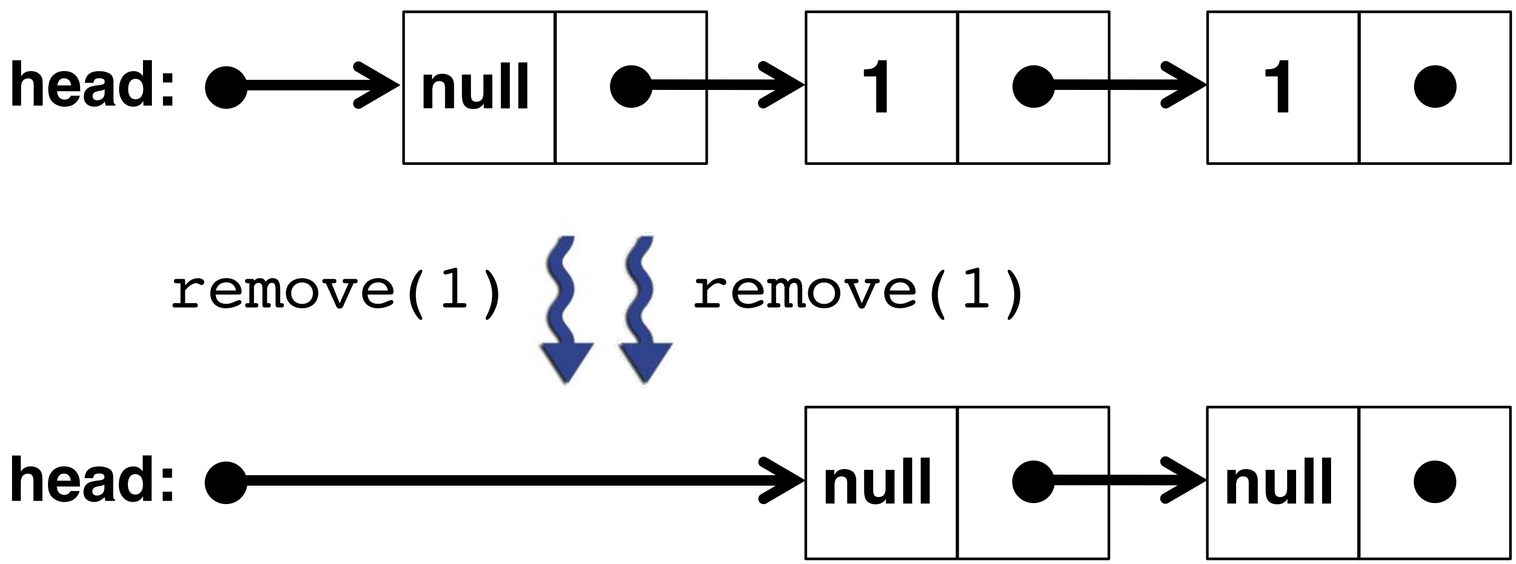


Motivating Example

```
Thread 1:
 @atomic {
   q.remove(1);
 }

Thread 2:
 @atomic {
   q.remove(1);
 }
```

- But in an **interleaved** execution:

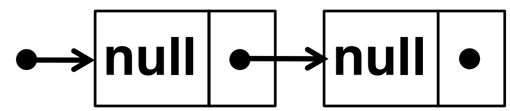


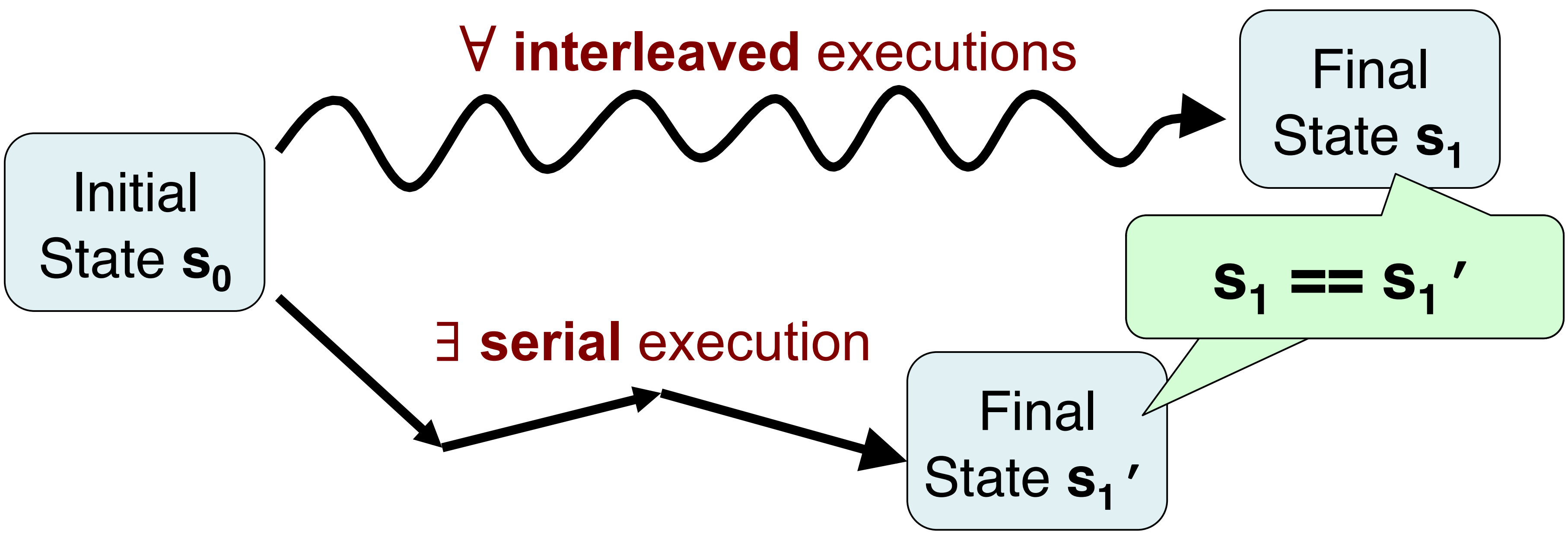
Motivating Example

```
Thread 1:
  @atomic {
    q.remove(1);
  }

Thread 2:
  @atomic {
    q.remove(1);
  }
```

- Traditional atomicity requires:

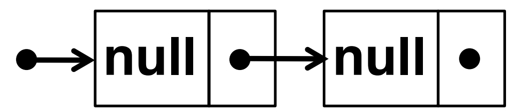


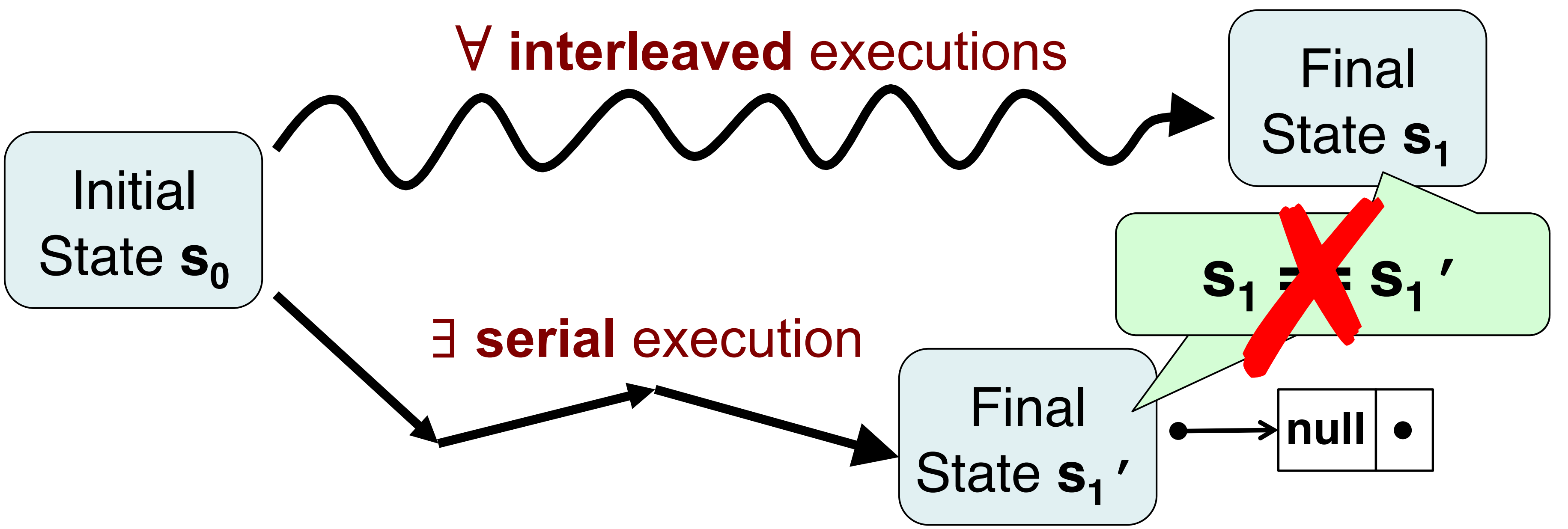


Motivating Example

```
Thread 1:                    Thread 2:
 @atomic {                    @atomic {
   q.remove(1);                 q.remove(1);
 }                            }
```

- Traditional atomicity requires:



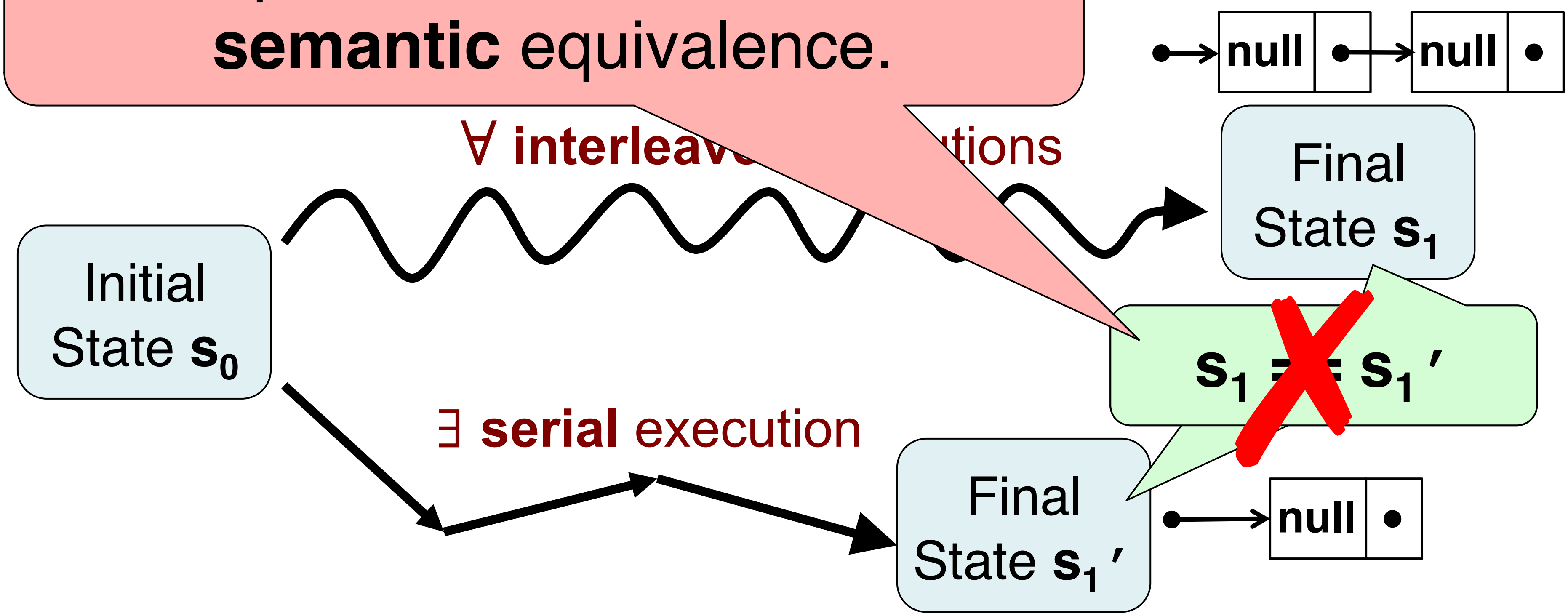


Motivating Example

```
Thread 1:
 @atomic {
   q.remove(1);
 }

Thread 2:
 @atomic {
   q.remove(1);
 }
```

Replace with user-defined **semantic** equivalence.



EECS
Electrical Engineering and Computer Sciences

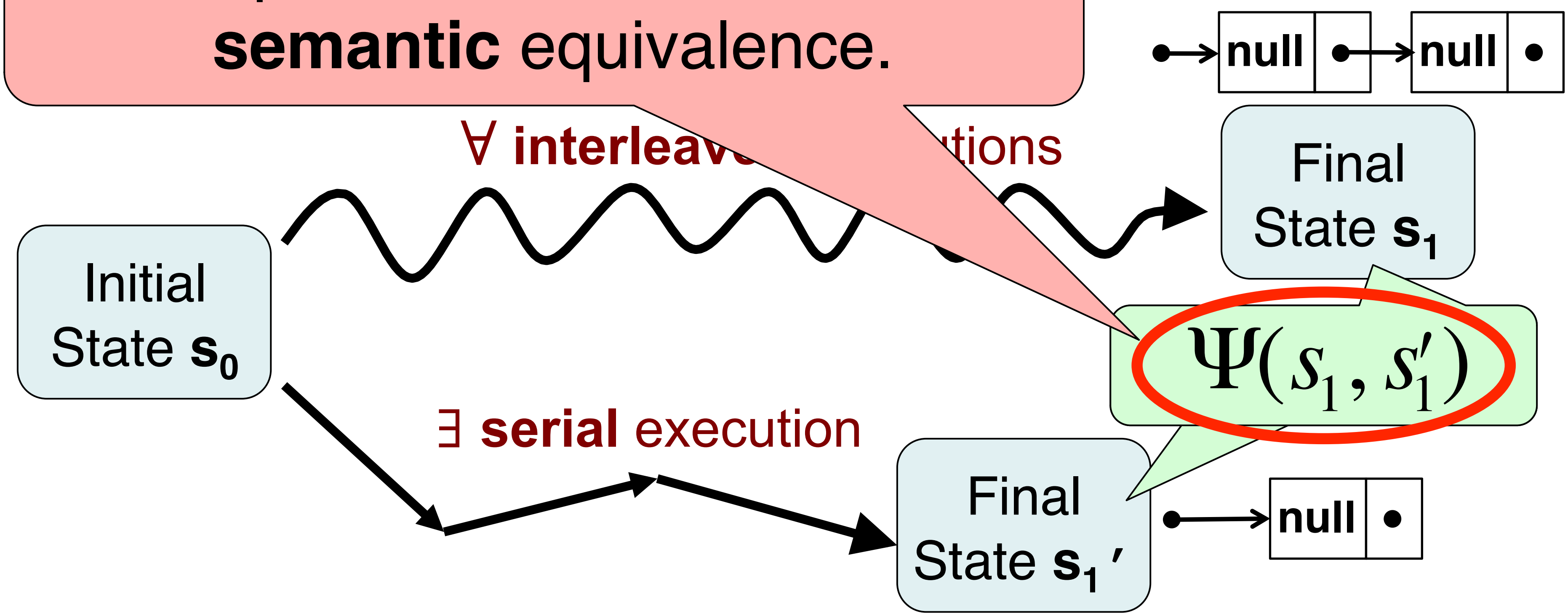
Semantic Atomicity

BERKELEY PAR LAB

```
Thread 1:
  @atomic {
    q.remove(1);
  }

Thread 2:
  @atomic {
    q.remove(1);
  }
```

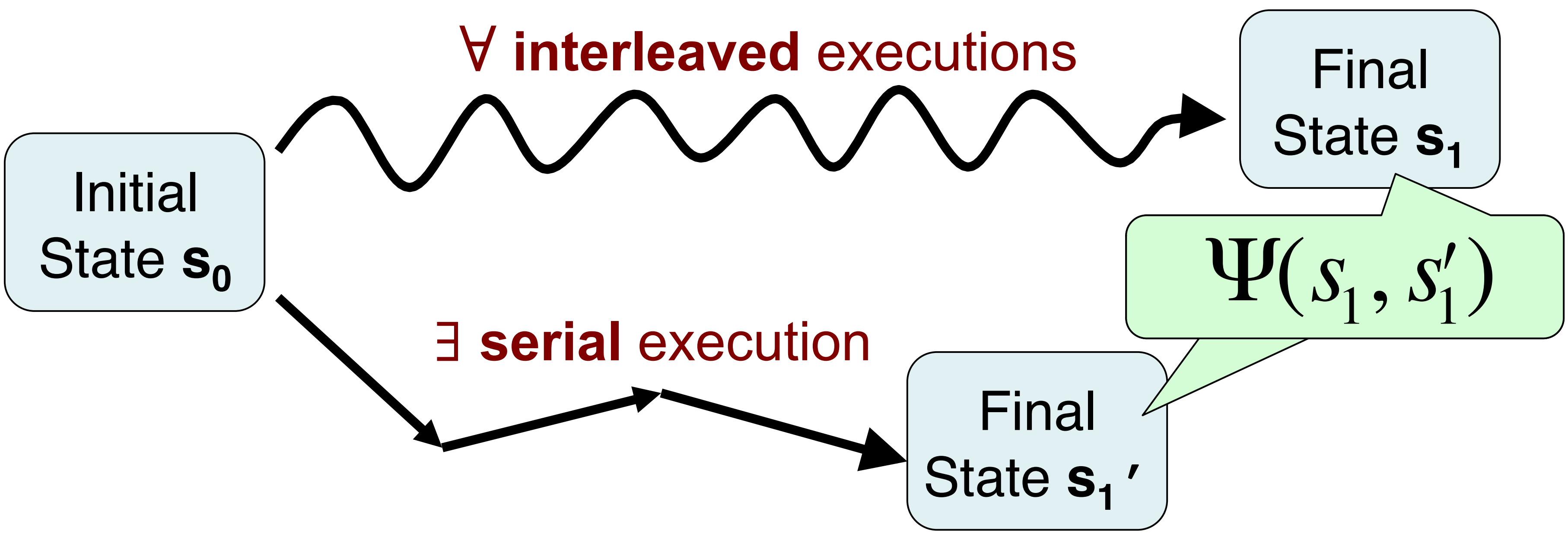
Replace with user-defined **semantic** equivalence.



Semantic Atomicity Example

```
Thread 1:                     Thread 2:
 @atomic {                     @atomic {
   q.remove(1);                  q.remove(1);
 }                             }

   Atomicity predicate: q.equals(q')
```



Bridge Predicates

Thread 1:

```
@atomic {
  q.remove(1);
}
```

Thread 2:

```
@atomic {
  q.remove(1);
}
```

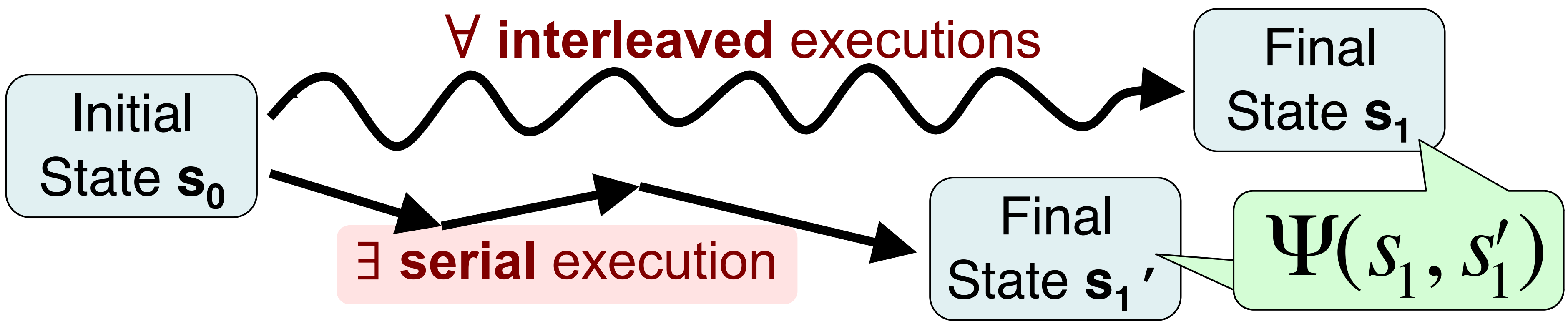
Atomicity predicate: **q.equals(q')**

Bridge predicate.

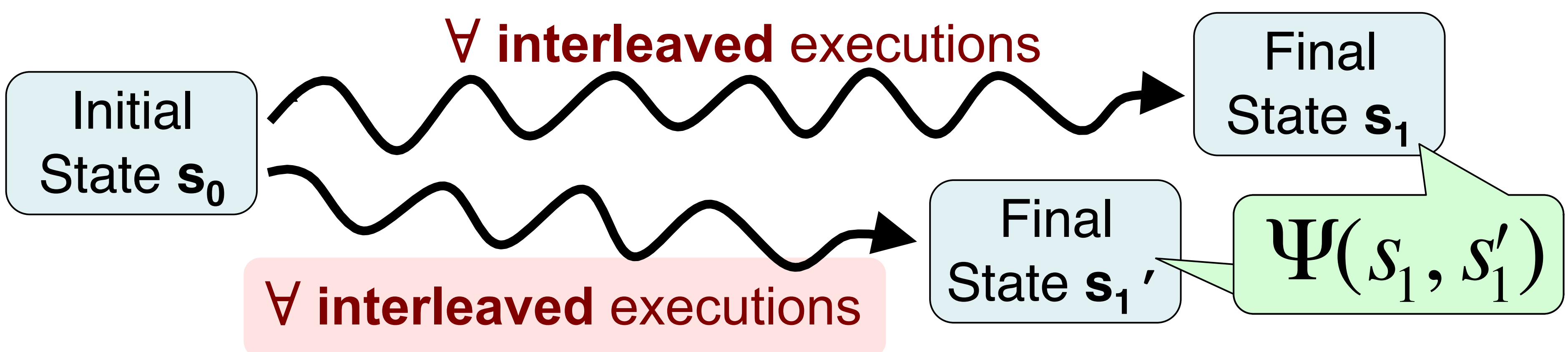
- Burnim, Sen, "Asserting and Checking Determinism for Multithreaded Programs", FSE 2009, CACM 2010.

Atomicity vs. Determinism

- Semantic **Atomicity**:



- Semantic **Determinism**:



EECS Electrical Engineering and Computer Sciences

Semantic Atomicity Example

BERKELEY PAR LAB

```
int bal = 0;
int conflicts = 0;

deposit(int a) {
  @atomic {
    int t = bal;
    while (!CAS(&bal, t, t+a)) {
      t = bal;
      conflicts += 1;
    }
  }
}
```

With CAS, updates to balance are atomic.

"Performance counter" of # of CAS failures.

Atomicity predicate: bal == bal'

EECS Electrical Engineering and Computer Sciences

Semantic Atomicity Example

BERKELEY PAR LAB

```
        ConcurrentList list;

Thread 1:                     Thread 2:
 @atomic {                     @atomic {
    ...                           ...
    list.add(1);                  list.add(3);
    ...                           ...
    list.add(2);                  list.add(4);
 }                             }

Atomicity predicate: eqSets(list,list')
```

- If `list` is [1,3,2,4], an atomicity violation?
  - User must specify **intended** atomicity.

Outline

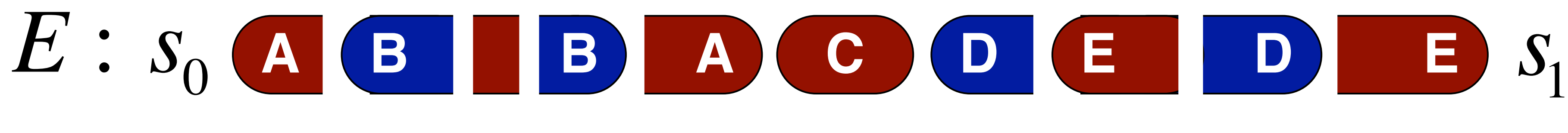
- Introduction + Motivation
- Background: Atomicity
- Specifying Semantic Atomicity
- **Testing Semantic Atomicity**
- Experimental Evaluation
- Conclusion

EECS Electrical Engineering and Computer Sciences

Testing Semantic Atomicity

BERKELEY PAR LAB

- **Interleaved** run E is **semantically atomic** w.r.t. Ψ iff there exists a **serial** run E' s.t.:
  - The final states of E, E' satisfy $\Psi(s_1, s_1')$.



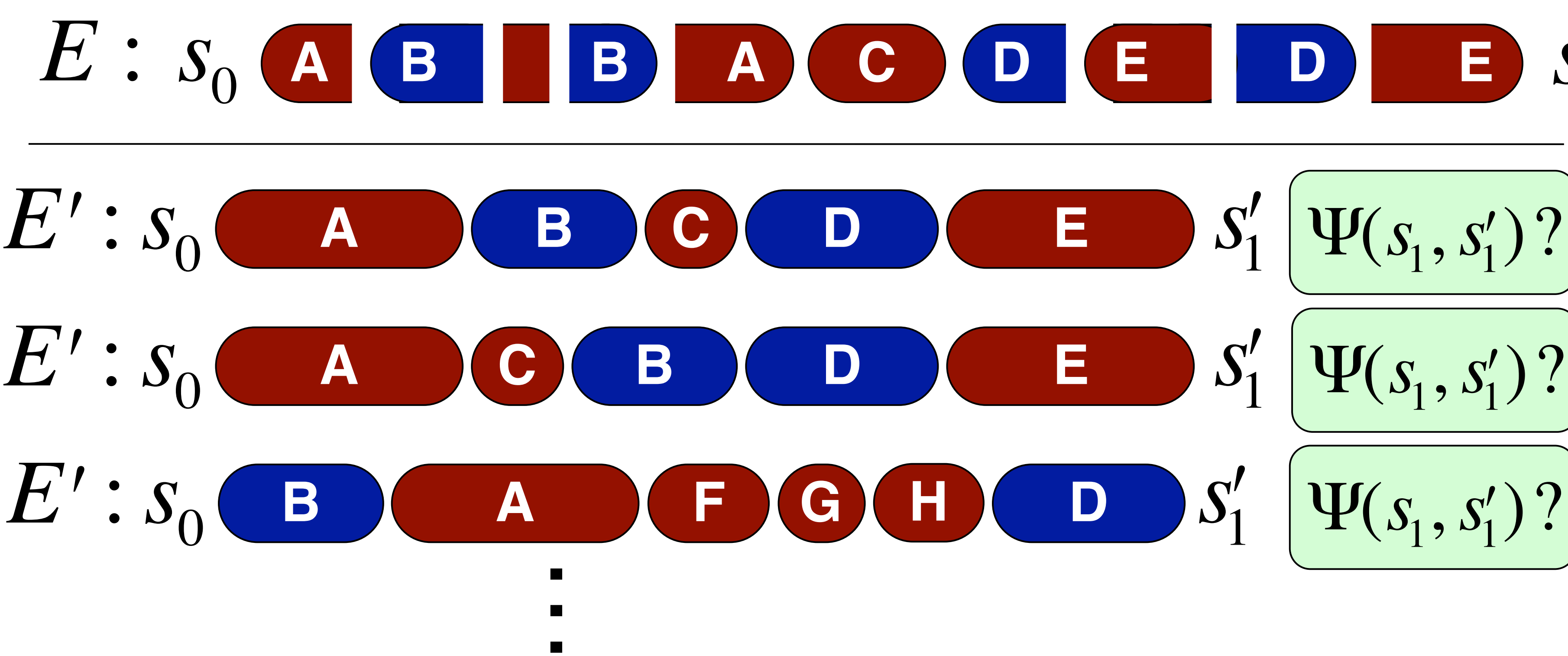
Is E semantically atomic w/ respect to Ψ?

EECS Electrical Engineering and Computer Sciences

Testing Semantic Atomicity

BERKELEY PAR LAB

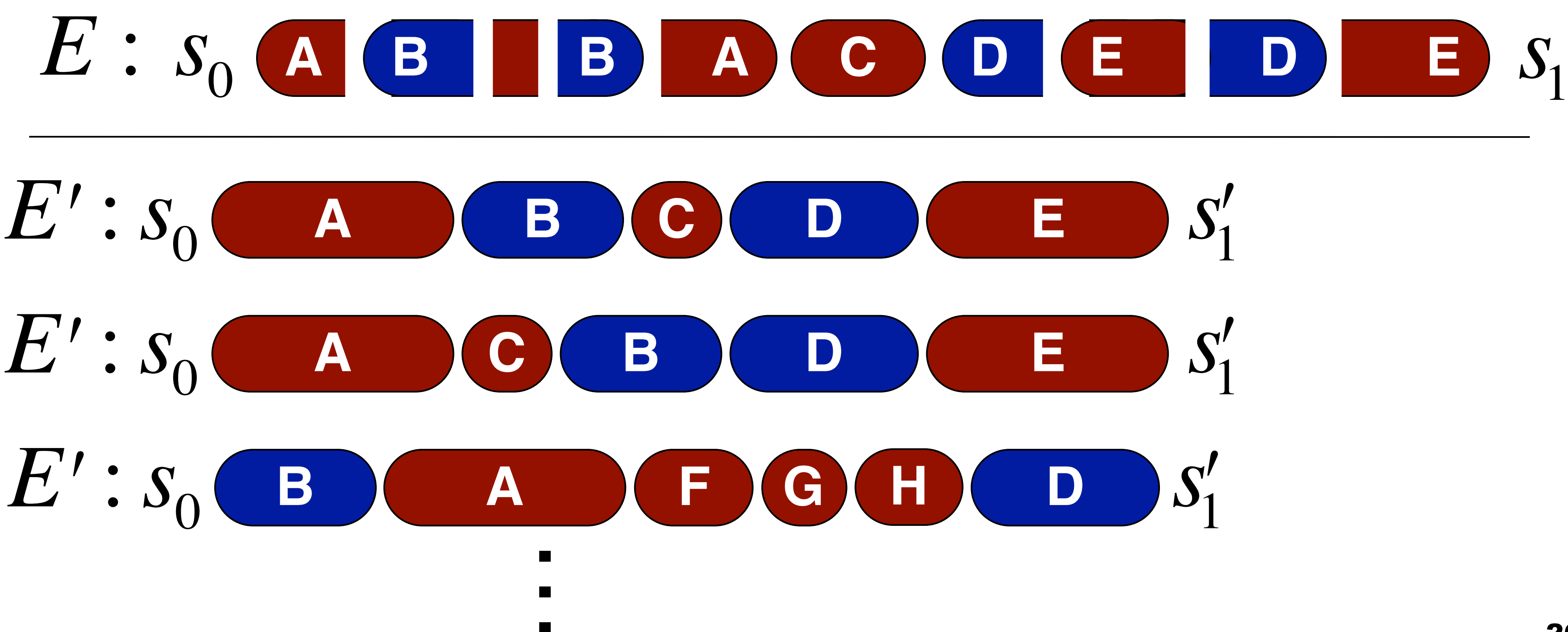
- **Interleaved** run E is **semantically atomic** w.r.t. Ψ iff there exists a **serial** run E' s.t.:
  - The final states of E, E' satisfy $\Psi(s_1, s_1')$.



Testing Semantic Atomicity

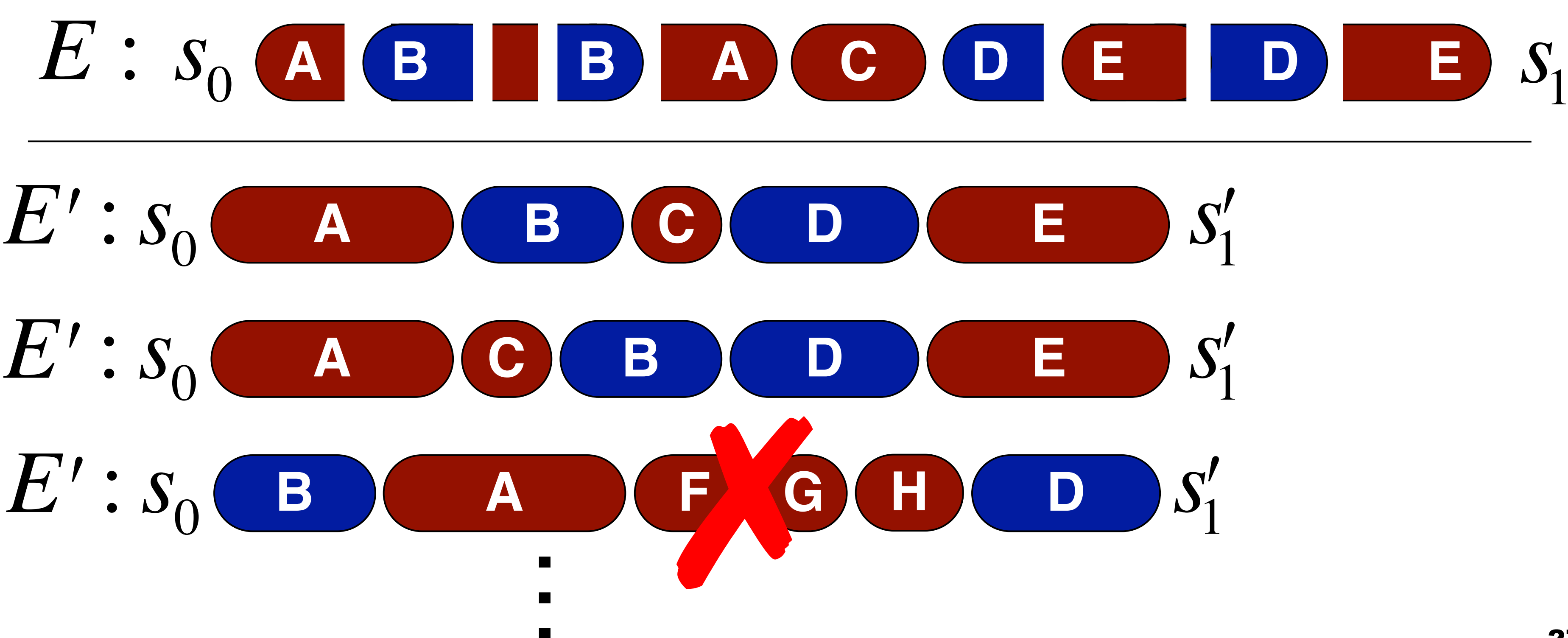
Infeasible to try all serial executions.

Can we **restrict** this search?



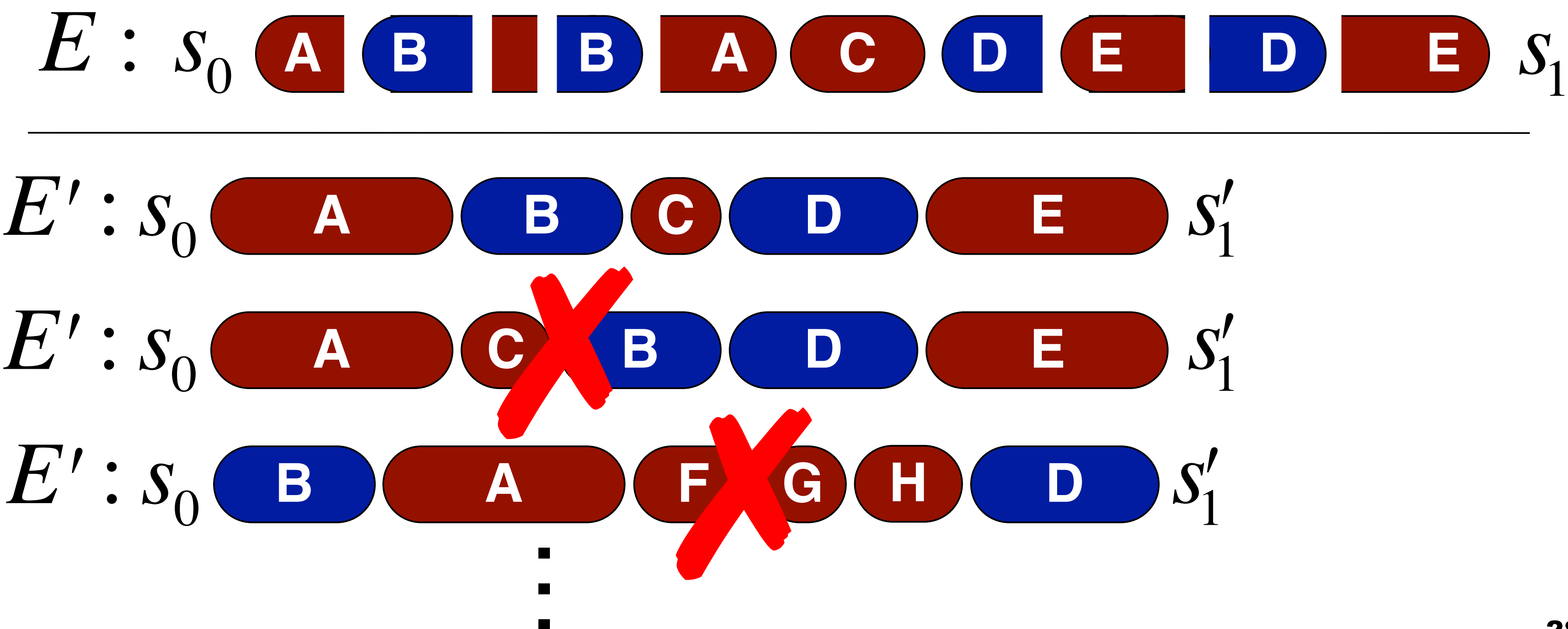
Testing Semantic Atomicity

1. The final states of E, E' satisfy $\Psi(s_1, s_1')$.
2. E and E' execute the same atomic blocks.



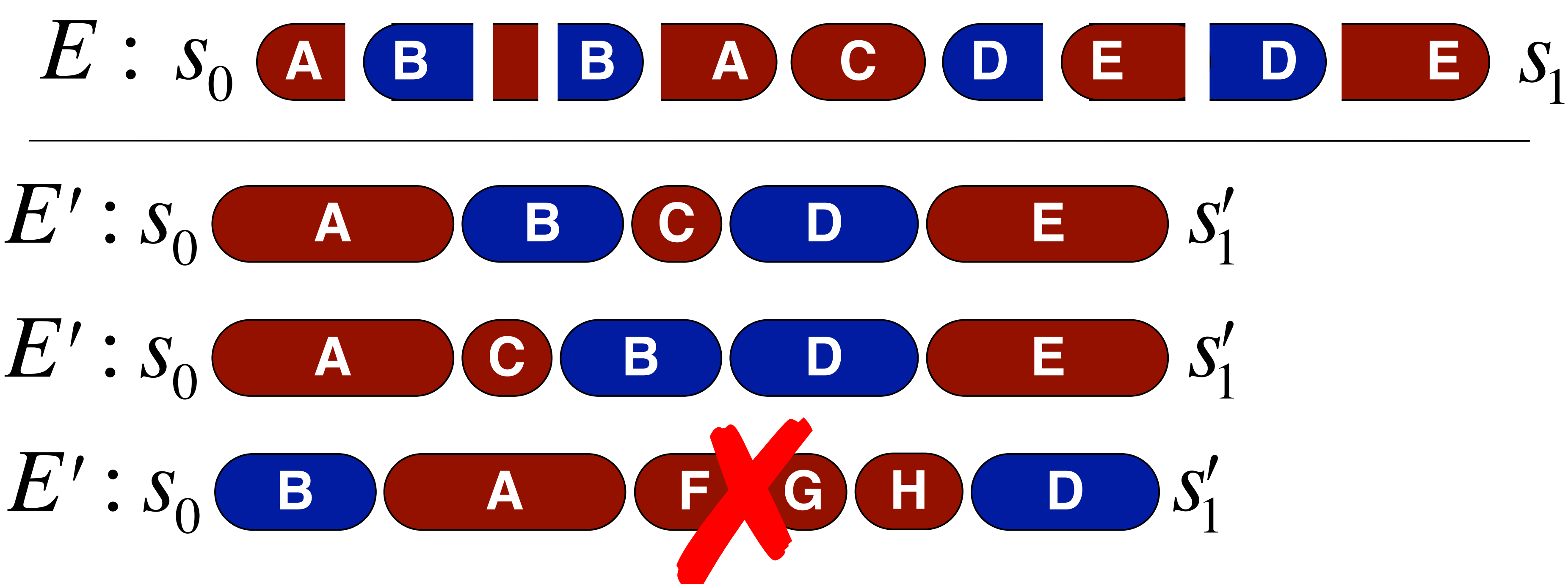
Testing Semantic Atomicity

1. The final states of E, E' satisfy $\Psi(s_1, s_1')$.
2. E and E' execute the same atomic blocks.
3. Non-overlapping atomic blocks appear in the same order in E and E' .



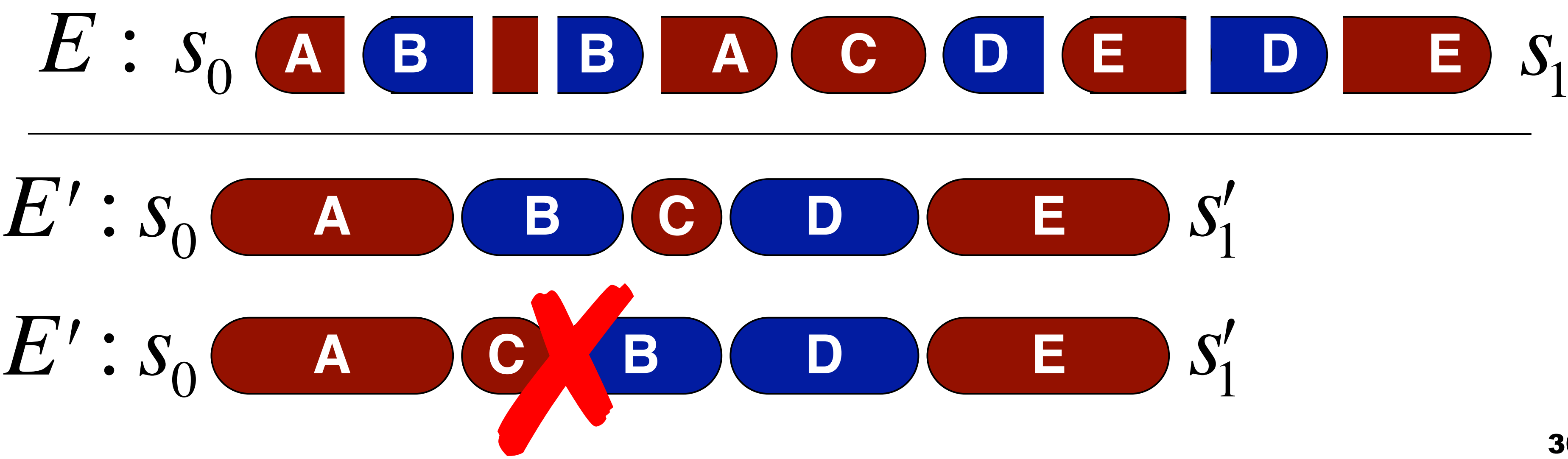
Semantic Serializability

- **Def:** Interleaved run E is **semantically serializable** iff exists a serial run E' s.t.:
  1. The final states of E, E' satisfy $\Psi(s_1, s_1')$.
  2. E and E' execute the same atomic blocks.



Semantic Strict Serializability

- **Def:** Interleaved E is **semantically strictly serializable** iff exists a serial run E' s.t.:
  1. The final states of E, E' satisfy $\Psi(s_1, s_1')$.
  2. E and E' execute the same atomic blocks.
  3. Non-overlapping atomic blocks appear in the same order in E and E' .



Semantic Strict Serializability

- **Def:** Interleaved E is **semantically strictly serializable** iff exists a serial run E' s.t.:
  1. The final states of E, E' satisfy $\Psi(s_1, s_1')$.
  2. E and E' execute the same atomic blocks.
  3. Non-overlapping atomic blocks appear in the same order in E and E'.

E has **N** blocks, with **≤ *K* overlapping**.

==>

Can check semantic strict serializability by examining **≤ *K!*** serial runs.

Testing Semantic Atomicity

- To test atomicity of program P:
  - Systematically/randomly generate executions E with **≤ *K* overlapping atomic blocks**.
  - For each E, report a violation if not **semantically strictly serializable**.
- **Small Scope Hypothesis:** Can find bugs with small # of overlapping atomic blocks.

Outline

- Introduction + Motivation
- Background: Atomicity
- Specifying Semantic Atomicity
- Testing Semantic Atomicity
- **Experimental Evaluation**
- Conclusion

EECS Electrical Engineering and Computer Sciences

Experimental Evaluation

BERKELEY PAR LAB

- Wrote semantic atomicity specs for several Java benchmarks.
  - Concurrent data structures and parallel apps.
- **Setup:** For each benchmark:
  - Generate 200-900 random interleaved runs, with one atomic block interrupted by ≤4 others.
  - Check semantic strict serializability of each.
  - To compare, also check conflict-serializability.

EECS
Electrical Engineering and Computer Sciences

Experimental Results I

Berkeley Par Lab

Benchmark	LoC	Test Runs	Semantic Atomicity Violations		Strict Atomicity Violations	
			Runs	Static Blocks	Runs	Static Blocks
JDK LinkedQueue	200	241	7	**2**	**≫7**	**4**
JDK SkipListMap	1400	487	6	**2**	**≫7**	**4**
JDK CwArrayList	600	222	0	0	0	0
lock-free list	100	319	57	**1**	**≫57**	**2**
lazy list-based set	100	231	0	0	**≫0**	**2**

Experimental Results II

Benchmark	LoC	Test Runs	Semantic Atomicity Violations		Strict Atomicity Violations	
			Runs	Static Blocks	Runs	Static Blocks
PJ pi	150	20	5	**1**	5	1
PJ keysearch	200	904	0	0	0	0
PJ fractal	250	73	0	0	0	0
PJ phylogenetic	4400	603	27	**1**	**≫27**	**2**

Application benchmarks from Parallel Java Library (Kaminsky 2007), use ~15000 LoC from PJ library.

JDK Atomicity Bug

```
ConcurrentLinkedQueue q;
q.add(1);  q.add(2);

Thread 1:                  Thread 2:
 @atomic {
   q.remove(1);             @atomic {
 }                            sz = q.size();
 @atomic {                  }
   q.add(3);
 }
       Atomic with respect to:
         q.equals(q') ∧ (sz == sz')
```

- **Not atomic:** `q.size()` can return `sz=3`.

Semantic Atomicity Bug II

```
parallel-for (t in trees) {
   @atomic {
      cost = compute_cost(t);
      synchronized (min_cost) {
         min_cost = min(min_cost, cost);
      }
      if (cost == min_cost) {
         min_tree = t;
      }
   }
}       Atomic with respect to:
           min_tree.equals(min_tree')
             ∧ (min_cost == min_cost')
```

Updates to `min_tree` not synchronized.

Outline

- Introduction + Motivation
- Background: Atomicity
- Specifying Semantic Atomicity
- Testing Semantic Atomicity
- Experimental Evaluation
- **Conclusion**

Conclusion

- **Semantic atomicity**.
  - Generalization for capturing high-level non-interference properties of real, complex code.
  - Testing via strict serializability.
  - Found several unknown atomicity errors.
- **Overall Goal:** Lightweight specifications for parallel correctness.
  - Easy for programmers to write.
  - With testing, effective in finding real bugs.
  - Determinism [CACM'10,ICSE'10], NDSeq [PLDI '11]

Questions?