Semantic Atomicity for Multithreaded Programs

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

- 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
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.
Atomicity a **non-interference property**.

- Block of code is **atomic** if it behaves *as if* executed all-at-once and without interruption.

```c
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?
Atomicity a **non-interference property**.

- Block of code is **atomic** if it behaves as if executed all-at-once and without interruption.

```c
int bal = 0;

void deposit(int a) {
    int t = bal;
    bal = t + a;
}

Thread 1:
- deposit(10)
- bal = 10

Thread 2:
- deposit(5)
- bal = 5

Atomicity specification does **not** hold.
Atomicity a **non-interference property**.

- Block of code is **atomic** if it behaves **as if** executed all-at-once and without interruption.

```c
int bal = 0;

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

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

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

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

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

In any **serial** execution:

head: null → 1 → 1

remove(1)

head: null → 1

remove(1)

head: null
Motivating Example

But in an **interleaved** execution:

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

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

![Diagram showing threaded execution of a linked list with an atomically removed value](image)
Motivating Example

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

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

Traditional atomicity requires:

∀ interleaved executions

Initial State $s_0$ → Final State $s_1$

∃ serial execution

Final State $s_1' 

S_1 == S_1'$
Motivating Example

Traditional atomicity requires:

∀ interleaved executions

∃ serial execution

Initial State $s_0$

Final State $s_1$

Final State $s_1'$

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

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

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

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

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

∀ interleaved executions

∃ serial execution

Initial State $s_0$

Final State $s_1'$

Final State $s_1'$

Final State $s_1''$

null

null
Semantic Atomicity

Thread 1:
\[
\text{@atomic } \{
q.\text{remove}(1);
\}
\]

Thread 2:
\[
\text{@atomic } \{
q.\text{remove}(1);
\}
\]

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

\[\forall \text{ interleaved executions} \]

Initial State \(s_0\)

\[\exists \text{ serial execution} \]

Final State \(s_1\)

Final State \(s_1'\)

\[\Psi(s_1, s_1') \]

null null
Semantic Atomicity Example

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

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

Atomicity predicate: q.equals(q')

∀ interleaved executions

Initial State s₀

∃ serial execution

Final State s₁

Final State s₁'

Ψ(s₁, s₁')
Atomicity vs. Determinism

- **Semantic Atomicity:**
  - Initial State $s_0$
  - $\forall$ interleaved executions
  - $\exists$ serial execution
  - Final State $s_1$
  - $\forall$ interleaved executions

- **Semantic Determinism:**
  - Initial State $s_0$
  - $\forall$ interleaved executions
  - $\forall$ interleaved executions
  - Final State $s_1$
  - $\forall$ interleaved executions
  - $\Psi(s_1, s_1')$
Semantic Atomicity Example

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

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

Atomicity predicate: bal == bal'
```

With CAS, updates to balance are atomic.

“Performance counter” of # of CAS failures.
Semantic Atomicity Example

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
Interleaved run $E$ is semantically atomic w.r.t. $\Psi$ 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 $\Psi$?
\[ \text{Interleaved run } E \text{ is semantically atomic w.r.t. } \Psi \text{ iff there exists a serial run } E' \text{ s.t.:} \]

- The final states of \( E, E' \) satisfy \( \Psi(s_1, s'_1) \).

\[ E : s_0 \quad A \quad B \quad B \quad A \quad C \quad D \quad E \quad D \quad E \quad s_1 \]

\[ E' : s_0 \quad A \quad B \quad C \quad D \quad E \quad s'_1 \quad \Psi(s_1, s'_1) \]

\[ E' : s_0 \quad A \quad C \quad B \quad D \quad E \quad s'_1 \quad \Psi(s_1, s'_1) \]

\[ E' : s_0 \quad B \quad A \quad F \quad G \quad H \quad D \quad s'_1 \quad \Psi(s_1, s'_1) \]
Infeasible to try all serial executions.

Can we restrict this search?

\[ E : s_0 \xrightarrow{A} B \xrightarrow{B} A C D E D E s_1 \]

\[ E' : s_0 \xrightarrow{A} B \xrightarrow{C} D E s_1' \]

\[ E' : s_0 \xrightarrow{A} C \xrightarrow{B} D E s_1' \]

\[ E' : s_0 \xrightarrow{B} A \xrightarrow{F} G H D s_1' \]

\[ \vdots \]
1. The final states of $E, E'$ satisfy $\Psi(s_1, s'_1)$.
2. $E$ and $E'$ execute the same atomic blocks.
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 $E'$.
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.

$E : s_0 \begin{array}{ccccccc} A & B & B & A & C & D & E \end{array} s_1$

$E' : s_0 \begin{array}{ccccccc} A & B & C & D & E \end{array} s'_1$

$E' : s_0 \begin{array}{ccccccc} A & C & B & D & E \end{array} s'_1$

$E' : s_0 \begin{array}{ccccccc} B & A & \cancel{F} & G & H & D \end{array} s'_1$
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 $E'$.

$$E : s_0 \xrightarrow{A} B \xrightarrow{B} A \xrightarrow{C} D \xrightarrow{E} D \xrightarrow{E} s_1$$

$$E' : s_0 \xrightarrow{A} B \xrightarrow{C} D \xrightarrow{E} s'_1$$

$$E' : s_0 \xrightarrow{A} C \xrightarrow{B} D \xrightarrow{E} s'_1$$
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 $\leq K$ overlapping. 

$\Rightarrow$ 
Can check semantic strict serializability by examining $\leq K!$ serial runs.
Testing Semantic Atomicity

- To test atomicity of program P:
  - Systematically/randomly generate executions $E$ with $\leq K$ 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
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 \( \leq 4 \) others.
- Check semantic strict serializability of each.
- To compare, also check conflict-serializability.
## Experimental Results I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Test Runs</th>
<th>Semantic Atomicity Violations</th>
<th>Strict Atomicity Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Runs</td>
<td>Static Blocks</td>
<td>Runs</td>
</tr>
<tr>
<td>JDK LinkedQueue</td>
<td>200</td>
<td>241</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>JDK SkipListMap</td>
<td>1400</td>
<td>487</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>JDK CwArrayList</td>
<td>600</td>
<td>222</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lock-free list</td>
<td>100</td>
<td>319</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>lazy list-based set</td>
<td>100</td>
<td>231</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Experimental Results II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Test Runs</th>
<th>Semantic Atomicity Violations</th>
<th>Strict Atomicity Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Runs</td>
<td>Static Blocks</td>
</tr>
<tr>
<td>PJ pi</td>
<td>150</td>
<td>20</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>PJ keysearch</td>
<td>200</td>
<td>904</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PJ fractal</td>
<td>250</td>
<td>73</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PJ phylogenetic</td>
<td>4400</td>
<td>603</td>
<td>27</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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