Program Analysis – Lecture 13
Analyzing Concurrent Programs
(Part 1)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
What does the following code print?

```javascript
function d(x) {
    return new Promise(resolve => {
        setTimeout (() => {
            resolve(x * 2);
        }, 2000);
    });
}
d(5). then ((r) => {
    console.log(r);
});
```

5 10 undefined Something else
What does the following code print?

```javascript
function d(x) {
    return new Promise(resolve => {
        setTimeout (() => {
            resolve(x * 2);
        }, 2000);
    });
}
d(5). then((r) => {
    console.log(r);
});
```

(but only after waiting for two seconds)

5 10 undefined Something else
Warm-up Quiz

What does the following code print?

```javascript
function d(x) {
    return new Promise(resolve => {
        setTimeout (() => {
            resolve(x * 2);
        }, 2000);
    });
}
d(5). then ((r) => {
    console.log(r);
});
```

Promise: Represents a result that is not yet complete

5 10 undefined Something else
What does the following code print?

```javascript
function d(x) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(x * 2);
        }, 2000);
    });
}
d(5).then((r) => {
    console.log(r);
});
```

Wait 2,000 milliseconds and then return the promise’s result

5 10 undefined Something else

https://ilias3.uni-stuttgart.de/vote/KN2I
What does the following code print?

```javascript
function d(x) {
    return new Promise(resolve => {
        setTimeout (() => {
            resolve(x * 2);
        }, 2000);
    });
}
d(5). then ((r) => {
    console.log(r);
});
```

Wait for the promise to resolve and then use its result

5  10  undefined  Something else

https://ilias3.uni-stuttgart.de/vote/KN2I
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
Why Bother with Concurrency?

- The free lunch provided by Moore’s law is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, multi-core processors became mainstream
  - Need concurrent programs to make full use of the hardware

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are "embarrassingly parallel"
Why Bother with Concurrency?

- The free lunch provided by Moore's law is over.
- CPU clock speeds stopped to increase around 2005.
- Instead, multi-core processors became mainstream.
- Need concurrent programs to make full use of the hardware.

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests.
  - Computations done on huge data are often "embarrassingly parallel."
Why Bother with Concurrency?

- The free lunch provided by Moore’s law is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, multi-core processors became mainstream
  - Need concurrent programs to make full use of the hardware

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are "embarrassingly parallel"
Concurrency Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in actor concurrency model, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via locks and barriers
Concurrenty Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in *actor concurrency model*, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via *locks* and *barriers*
Example

```java
int a = 0, b = 0;
boolean r = false, t = false;
a = 1;
r = true;
t = r;
b = a;
```

What does this program mean?
→ Behavior depends on thread interleaving
\[ a = b = 0 ; \quad r = t = \text{false} \]

\[ \sqrt{a = 1 \quad r = \text{true}} \quad t = r \quad b = a \]

\[ \begin{align*}
1 & \quad a = 1 \\
2 & \quad a = 1 \\
3 & \quad a = 1 \\
4 & \quad t = r \\
5 & \quad t = r \\
6 & \quad t = r
\end{align*} \]

\[ a = 1 \quad r = \text{true} \quad t = r \quad b = a \quad r = \text{true} \quad a = 1 \quad r = \text{true} \quad b = a \quad r = \text{true} \quad b = a \quad r = \text{true} \]

\[ \begin{align*}
1 & \quad t = \text{false} \\
2 & \quad t = \text{false} \\
3 & \quad t = \text{false} \\
4 & \quad t = \text{false} \\
5 & \quad t = \text{false} \\
6 & \quad t = \text{false}
\end{align*} \]

\[ b = 1 \quad b = 1 \quad b = 0 \quad b = 1 \quad b = 1 \quad b = 1 \]

\[ t = \text{true} \quad \text{implies} \quad b = 1 \]
Assumption made here:
Programs execute under sequential consistency

- Program order is preserved: Each thread’s instructions execute in the specified order
- Shared memory behaves like a global array: Reads and writes are done immediately

- We assume sequential consistency for the rest of the lecture
- Many real-world platforms have more complex semantics (”memory models”)

Sequential Consistency
What Can Go Wrong?

Common source of errors: **Data races**

- Two accesses to the same shared memory location
- At least one access is a write
- Ordering of accesses is non-deterministic
Example

```java
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```
Example

// bank account
int balance = 10;

Thread 1

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

Thread 2

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Shared memory location

Read

Write
Example

// bank account
int balance = 10;

Thread 1  Thread 2

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

3 races
Example

// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Quiz: What values can balance have after executing this code?

https://ilias3.uni-stuttgart.de/vote/KN2I
Example

// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Possible outcomes:
balance may be 3, 8, and 15

But: Only 8 is correct
Avoiding Data Races

Use **locks** to ensure that **accesses to shared memory do not interfere**

```c
int balance = 10;

acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);

Thread 1
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);

Thread 2
```
Avoiding Data Races

Use locks to ensure that accesses to shared memory do not interfere

```
int balance = 10;

Thread 1
acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);
```

```
Thread 2
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);
```

Same lock ⇒ Mutually exclusive critical sections
Avoiding Data Races

Use **locks** to ensure that **accesses to shared memory do not interfere**

```java
int balance = 10;

synchronized (L) {
    int tmp1 = balance;
    balance = tmp1 + 5;
}

synchronized (L) {
    int tmp2 = balance;
    balance = tmp2 - 7;
}
```

(Java syntax)
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
Basic idea: Look for "unprotected" accesses to shared memory

Assumption: All accesses to a shared memory location \( v \) should happen while holding the same lock \( L \)

\[ \rightarrow \text{Consistent locking discipline} \]

Dynamic analysis that monitors all lock acquisitions, lock releases, and accesses of shared memory locations
Lockset Algorithm (Simple Form)

- Let $\text{locksHeld}(t)$ be the set of locks held by thread $t$

- For each shared memory location $v$, initialize $C(v)$ to the set of all locks

- On each access to $v$ by thread $t$
  - Set $C(v) := C(v) \cap \text{locksHeld}(t)$
  - If $C(v) = \emptyset$, issue a warning
Lockset Algorithm (Simple Form)

- Let $\text{locksHeld}(t)$ be the set of locks held by thread $t$.
- For each shared memory location $v$, initialize $C(v)$ to the set of all locks.
- On each access to $v$ by thread $t$:
  - Set $C(v) := C(v) \cap \text{locksHeld}(t)$.
  - If $C(v) = \emptyset$, issue a warning.

Lockset refinement
### Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>locksHeld</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = 10</td>
<td>{}</td>
</tr>
<tr>
<td>acquire (L1)</td>
<td>{}</td>
</tr>
<tr>
<td>tmp1 = balance</td>
<td>{}</td>
</tr>
<tr>
<td>balance = tmp1 + a</td>
<td>{}</td>
</tr>
<tr>
<td>release (L1)</td>
<td>{}</td>
</tr>
<tr>
<td>tmp2 = balance</td>
<td>{}</td>
</tr>
<tr>
<td>acquire (L2)</td>
<td>{}</td>
</tr>
<tr>
<td>balance = tmp2 - b</td>
<td>{}</td>
</tr>
<tr>
<td>release (L2)</td>
<td>{}</td>
</tr>
</tbody>
</table>

### C(balance)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

---

**warning**: data race
Simple Lockset is Too Strict

Simple lockset algorithm produces false positives for

- variables initialized without locks held
- read-shared data read without locks held
- read-write locking mechanisms
  (producer-consumer style)
Refining the Lockset Algorithm

- Keep state of each shared memory location
- Issue warnings only in the Shared-modified state
Summary: Eraser

- **Dynamic analysis** to detect data races
- Assumes **consistent locking discipline**
- **Limitations**
  - May report false positives when locks are acquired inconsistently but correctly
  - May miss data races because it does not consider all possible interleavings
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
Thread Safety

- Popular way to encapsulate the challenges of concurrent programming: Thread-safe classes
- Class ensures correct synchronization
- Clients can use instances as if they were alone
- Rest of program can treat implementation of thread-safe class as a blackbox
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”

“operations ... 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”
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”

“operations ... 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”
Example from JDK

```java
StringBuffer b = new StringBuffer()
  b.append("a")
  b.append("b")
  b.append("c")
  b.append("b")
```

Thread 1

Thread 2
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a")
b.append("b")
b.append("a")
b.append("b")
b.append("c")

Thread 1

Thread 2
```

Quiz: What can be the content of \( b \) if `StringBuffer` is thread-safe?
Example from JDK

```java
StringBuffer b = new StringBuffer()
b.append("a")
b.append("b")
b.append("c")
```

Thread 1

```
b.append("a")
b.append("b")
```

Thread 2

```
b.append("c")
```

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>&quot;cab&quot;</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>&quot;ac&quot;</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>&quot;bac&quot;</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a")
b.append("b")
b.append("c")

b.append("a")
b.append("b")
```

Thread 1

Thread 2

"abc" ✓  "cab" ✓  "acb" ✓  "ac" ✗  "bac" ✗
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a")
b.append("b")
b.append("c")
```

Thread 1: "abc" ✓
Thread 2: "cab" ✓

"ac" ×
"bac" ×
Example from JDK

```java
StringBuffer b = new StringBuffer();

b.append("a")
b.append("b")
b.append("c")

Thread 1
b.append("a")
Thread 2
b.append("c")

b.append("b")

"abc" ✓  "cab" ✓  "acb" ✓  "ac"  ✓  "bac"  x
```
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a") // Thread 1
b.append("b")
b.append("c") // Thread 2

"abc" ✔  "cab" ✔  "acb" ✔  "ac" ✗  "bac" ✗
```
Testing Thread-Safe Classes

- Correctness of program relies on thread safety of specific classes
- But: What if the class is actually not thread-safe?

- ConTeGe = Concurrent Test Generator
- Creates multi-threaded unit tests
- Detects thread safety violations by comparing concurrent behavior against linearizations
Example Bug from JDK

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

```java
b.insert(1, b)
b.deleteCharAt(1)
```

Thread 1

Thread 2

```
Thread 1

Thread 2
```

```java
b.insert(1, b)  b.deleteCharAt(1)
```
Example Bug from JDK

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Thread 1
```
b.insert(1, b)
```
Thread 2
```
b.deleteCharAt(1)
```

⚠️ IndexOutOfBoundsException

Confirmed as bug: Issue #7100996
ConTeGe

Class under test (CUT) →

Generate a concurrent test →

Execute →

Thread safety oracle →

Bug
ConTeGe

Class under test (CUT)

Generate a concurrent test

Execute

Thread safety oracle

Bug
ConTeGe

Class under test (CUT)

Generate a concurrent test

Execute

Thread safety oracle

Bug
Generating Concurrent Tests

Example:

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Thread 1

Thread 2

```java
b.insert(1, b)    b.deleteCharAt(1)
```
Generating Concurrent Tests

Example:

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Sequential prefix:
Create and set up CUT instance

Thread 1
```
b.insert(1, b)
```

Thread 2
```
b.deleteCharAt(1)
```
Example:

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Thread 1

```
b.insert(1, b)
```

Thread 2

```
b.deleteCharAt(1)
```

Concurrent suffixes:
Use shared CUT instance
Test Generation Algorithm

1. Create prefix
   - Instantiate CUT
   - Call methods

2. Create suffixes for prefix
   - Call methods on shared CUT instance

3. Prefix + two suffixes = test

Selection of methods via feedback-directed test generation
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

Randomly select a constructor
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

Randomly select a constructor

StringBuffer b = new StringBuffer()
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

StringBuffer b = new StringBuffer();

After adding a call: Execute
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer ()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

After adding a call:
Execute

```java
StringBuffer b = new StringBuffer ()
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer()
```

Randomly select a method
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer()
b.append(/* String */) // Randomly select a method
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value

```java
StringBuffer b = new StringBuffer()
b.append(/* String */)
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value

```java
StringBuffer b = new StringBuffer();
b.append("abc");
```
1. Create prefix

- Instantiate CUT
- Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```
Creating a Prefix

1. Create prefix
   ■ Instantiate CUT
   ■ Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Thread 1

Thread 2
Creating Suffixes

2. Create suffixes for prefix
   - Call methods on shared CUT instance
2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```
2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(/* int */, /* CharSequence*/)
```
2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Arguments:

a) Take available object
b) Call method returning required type
c) Random value

```java
b.insert(/* int */, /* CharSequence */)
```
2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value

```java
b.insert(-5, b)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(-5, b)
```

After adding a call: Execute
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.insert(-5, b)
```

After adding a call: Execute
Creating Suffixes

2. Create suffixes for prefix
   - Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(/* int */, /* CharSequence */)```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating Suffixes

2. Create suffixes for prefix
   - Call methods on shared CUT instance

   ```java
   StringBuffer b = new StringBuffer();
   b.append("abc");
   b.insert(1, b);
   ```

   Arguments:
   a) Take available object
   b) Call method returning required type
   c) Random value
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer();
b.append("abc")
b.insert(1, b)
```
2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer();
b.append("abc")
b.insert(1, b)
```

```java
b.insert(1, b)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer();
b.append("abc")
```

```java
b.insert(1, b)  b.deleteCharAt(1)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

After adding a call:
```
Execute
```

```java
b.insert(1, b)  // b.deleteCharAt(1)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

After adding a call:
Execute
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

```java
b.insert(1, b)  b.deleteCharAt(1)
```
Creating a Test

3. Prefix + two suffixes = test
Creating a Test

3. Prefix + two suffixes = test

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

```java
b.insert(1, b)    // b.deleteCharAt(1)
```
Creating a Test

3. Prefix + two suffixes = test

StringBuffer b = new StringBuffer()
b.append("abc")

Thread 1

b.insert(1, b)  b.deleteCharAt(1)

Thread 2

Spawn new thread for each suffix
Approach

Class under test (CUT) → Generate a concurrent test

Generate a concurrent test → Execute

Execute → Thread safety oracle

Thread safety oracle → Bug
Approach

Class under test (CUT) →

Generate a concurrent test →

Execute →

Thread safety oracle → Bug
Thread Safety Oracle

Does the test execution expose a thread safety violation?

- Focus on exceptions and deadlocks
- Compare concurrent execution to linearizations
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

No

All linearizations checked

Thread safety violation
The Oracle

1. Execute concurrently
   - Exception or deadlock?
     - No
     - Yes
6. Execute linearization
5. Same failure?
   - Yes
   - No
4. All linearizations checked
3. Thread safety violation
The Oracle

Execute concurrently

Exception or deadlock?

Yes → Execute linearization

Same failure?

Yes → Yes

No → Thread safety violation

No

All linearizations checked
The Oracle

Execute concurrently

Exception or deadlock?

No

Yes

Execute linearization

Same failure?

No

Thread safety violation

All linearizations checked

Yes
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

No

All linearizations checked

Thread safety violation

No
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

All linearizations checked

Thread safety violation

No
The Oracle

1. Execute concurrently
   - Exception or deadlock?
     - No
     - Yes
       - Execute linearization
         - Same failure?
           - No
             - Thread safety violation
           - Yes
2. All linearizations checked
The Oracle

Execute concurrently

Exception or deadlock?

Yes → Execute linearization

Same failure?

Yes →

No → Thread safety violation

All linearizations checked
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)  b.deleteCharAt(1)
```
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")

b.insert(1, b) // Thread 1
b.insert(1, b) // Thread 2

b.deleteCharAt(1)
```

⚠️
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")

Thread 1
b.insert(1, b)

Thread 2
b.deleteCharAt(1)
```

StringBuffer b = ..
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")

b.insert(1, b)  // Thread 1
b.deleteCharAt(1)  // Thread 2

StringBuffer b = ..
b.append("abc")
b.insert(1, b)
```

The operations in `Thread 1` and `Thread 2` can lead to unexpected results due to thread interference. It's important to use thread-safe operations or synchronized access to shared resources when dealing with multi-threaded environments.
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)   // Thread 1
b.deleteCharAt(1)   // Thread 2
b.insert(1, b)   // Thread 1
b.deleteCharAt(1)   // Thread 2
```

Thread safety violation

```java
StringBuffer b = ..
b.append("abc")
b.insert(1, b)   // Thread 1
b.deleteCharAt(1)   // Thread 2
b.insert(1, b)   // Thread 1
b.deleteCharAt(1)   // Thread 2
```
Properties of the Oracle

Sound but incomplete *

- All reported violations are real
- Cannot guarantee thread safety

Independent of bug type

- Data races
- Atomicity violations
- Deadlocks

* with respect to incorrectness
Implementation & Results

- Implemented for Java classes
- Applied to popular thread-safe classes from JDK, Apache libraries, etc.
- Found 15 concurrency bugs, including previously unknown problems in JDK
- Takes between several seconds and several hours (worst-case: 19 hours)
  - Recent master thesis, published at ICSE’17: Worst-case time reduced to several minutes
Summary

- Concurrent programming is inevitable
- Writing correct concurrent programs is hard
- Techniques to detect concurrency bugs
  - Dynamic data race detection
  - Test generation and thread safety checking