Program Analysis
Analyzing Concurrent Programs (Part 3)
Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2020/2021
Outline

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

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
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")
```

Thread 1

Thread 2
Example from JDK

```java
StringBuffer b = new StringBuffer()

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

Thread 1

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

IndexOutOfBoundsException

Confirmed as bug: Issue #7100996
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")

Thread 1
b.insert(1, b)

Thread 2
b.deleteCharAt(1)
```
Generating Concurrent Tests

Example:

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

Sequential prefix: Create and set up CUT instance

- Thread 1
  - `b.insert(1, b)`
- Thread 2
  - `b.deleteCharAt(1)`
Generating Concurrent Tests

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

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

```
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")
```

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");
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

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

Randomly select a method

```java
b.insert(/* int */, /* CharSequence */)```

Creating Suffixes

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)
```

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)
```

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)
```
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)
```
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 a Test

3. Prefix + two suffixes = test

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

Spawn new thread for each suffix

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

Thread 1

Thread 2
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
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

All linearizations checked

Thread safety violation

No
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

Exception or deadlock?

No

Yes

Execute linearization

Same failure?

No

All linearizations checked

Thread safety violation
The Oracle

- Execute concurrently

  - Exception or deadlock?
    - No
      - Yes
      - Execute linearization
        - Same failure?
          - No
            - Thread safety violation
          - Yes
    - Yes
      - All linearizations checked

The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

No

Thread safety violation

All linearizations checked

No

Yes
The Oracle

1. Execute concurrently

2. Exception or deadlock?
   - Yes: Execute linearization
     - Same failure?
       - Yes: Thread safety violation
       - No: All linearizations checked
   - No: Yes

3. Yes: Thread safety violation
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

All linearizations checked

No

Same failure?

Yes

Thread safety violation

No
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?
    - No
      - Thread safety violation
    - Yes
      - Execute linearization
        - Same failure?
          - No
            - All linearizations checked
          - Yes
            - All linearizations checked

- Yes
  - All linearizations checked

35 - 8
The Oracle

- Execute concurrently
- Exception or deadlock?
  - No
  - Yes
    - Execute linearization
      - Same failure?
        - No
          - Thread safety violation
        - Yes
          - Yes
            - Yes
              - Yes

- All linearizations checked
Example

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

Thread 1

Thread 2

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

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)
b.deleteCharAt(1)
```

Thread 1

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

Thread 2

```java
StringBuffer b = ..
b.insert(1, b)
```

⚠️
Example

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

Thread 1

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

Thread 2

Thread safety violation

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

StringBuffer b = ..
b.append("abc")
b.deleteCharAt(1)
b.insert(1, b)
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)
  - Coverage-guided approach (ICSE’17): Worst-case time reduced to several minutes