Lincheck: Testing Concurrent Data Structures in Java

Nikita Koval, Hydra 2019
This is joint work with Dmitry Tsitelov, Maria Sokolova, Roman Elizarov, and Anton Evdokimov
Speaker: Nikita Koval

- Graduated @ ITMO University
- Previously worked as developer and research engineer @ Devexperts
- Teaching concurrent programming course @ ITMO University
- Researcher @ JetBrains
- PhD student @ IST Austria

@nkoval_
Writing concurrent code is pain
Writing concurrent code is pain

... testing it is not much easier!
var \ i = 0

\ i.\ inc() \quad \ i.\ inc()
```javascript
var i = 0

i.inc() // 0
// 1
// 1
```

```javascript
i.inc() // 1
// 0
```
var \ i \ = \ 0

\ i.\text{inc}() \ // \ 0 \ \ | \ \ \ \ i.\text{inc}() \ // \ 0
We do not expect this!
Sequential model

Sequential specification on operations

Concurrent model

Linearizability (usually)
Execution *is linearizable* $\iff \exists$ equivalent *sequential* execution wrt *happens-before* order (a bit harder)
Execution *is linearizable* $\iff \exists$ equivalent sequential execution wrt *happens-before* order (a bit harder)

```
val q = MSQueue<Int>()

q.add(1) q.poll(): 1
q.poll(): 2 q.add(2)
```
Execution *is linearizable* $\Leftrightarrow \exists$ equivalent *sequential* execution wrt *happens-before* order (a bit harder)

```scala
val q = MSQueue[Int]()
q.add(1)
q.poll(): 1
q.add(2)
q.poll(): 2
```
This counter is not linearizable
How to check whether my data structure is linearizable?
How to check whether my data structure is linearizable?

Formal proofs
How to check whether my data structure is linearizable?

Formal proofs

Model checking
How to check whether my data structure is linearizable?

Formal proofs

Model checking

Testing
How to check whether my data structure is linearizable?

Formal proofs

Model checking

Testing
How does the ideal test look?
How does the ideal test look?

class MSQueueTest {
    val q = MSQueue<Int>()
}

Initial state
How does the ideal test look?

class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) = q.add(element)

    @Operation fun poll() = q.poll()
}

Operations on the data structure
class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) =
        q.add(element)

    @Operation fun poll() = q.poll()
}

Operation parameters can be non-fixed!
class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) = q.add(element)

    @Operation fun poll() = q.poll()

    @Test fun runTest() = LinChecker.check(QueueTest::class)
}
How does the ideal test look?

```kotlin
class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) =
        q.add(element)

    @Operation fun poll() = q.poll()

    @Test fun runTest() =
        LinChecker.check(QueueTest::class)
}

Do we have such instrument?
```
How does the ideal test look?

class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) = 
        q.add(element)

    @Operation fun poll() = q.poll()

    @Test fun runTest() = 
        LinChecker.check(QueueTest::class)
}

Do we have such instrument?

YEEES!
Lin-Check Overview

Lincheck = Linearizability Checker (supports not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck
Lin-Check Overview

Lincheck = Linearizability Checker (supports not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck

1. Generates a random scenario
2. Executes it a lot of times
3. Verifies the results
Lin-Check Overview

Lincheck = Linearizability Checker (supports not only linearizability)
https://github.com/Kotlin/kotlinx-lincheck

1. Generates a random scenario
2. Executes it a lot of times
3. Verifies the results
Invalid Execution Example

Init part:
[poll(): null, add(9)]

Parallel part:
| poll(): null | add(4) |
| add(3)       | add(6) |
| poll(): 4    | poll(): 3 |

Post part:
[add(1), poll(): 6]
How to check results for correctness?

Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results
How to check results for correctness?

Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results

2 threads x 15 operations \Rightarrow \text{OutOfMemoryError}
How to check results for correctness?

Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results

Smarter solution: Labeled Transition System (LTS)
LTS (Labeled Transition System)

Initial state

inc(): 0  inc(): 1  inc(): 2

dec(): 1  dec(): 2  dec(): 3

Operation with result

LTS is infinite
LTS (Labeled Transition System)
LTS-based verification

\[\text{val } q = \text{MSQueue}\langle\text{Int}\rangle()\]

\[\begin{array}{l}
q\text{.add}(4) & q\text{.poll}(): 4 \\
q\text{.poll}(): 9 & q\text{.add}(9)
\end{array}\]
LTS-based verification

```java
val q = MSQueue<Int>()

q.add(4)
q.poll(): 9
q.poll(): 4
q.add(9)
```
LTS-based verification

Result is different
LTS-based verification

val q = MSQueue[Int]()

q.add(4)
q.poll(): 4
q.poll(): 9
q.add(9)
q.poll(): 4
LTS-based verification

\[
\text{val } q = \text{MSQueue}<$\text{Int}>()
\]

\[
\begin{align*}
q.\text{add}(4) & \\
q.\text{poll}() & : 4 \\
q.\text{poll}() & : 9 \\
q.\text{add}(9) & \\
\end{align*}
\]
LTS-based verification

\[\text{val } q = \text{MSQueue}\langle\text{Int}\rangle()\]

\[q.\text{add(4)}\]
\[q.\text{poll()}: 4\]

\[q.\text{poll()}: 9\]
\[q.\text{add(9)}\]
LTS-based verification

\[
\text{val } q = \text{MSQueue}\langle\text{Int}\rangle()
\]

\[
q\cdot\text{add}(4) \quad q\cdot\text{poll}(): 4
\]

\[
q\cdot\text{add}(9) \quad q\cdot\text{poll}(): 9 \quad q\cdot\text{add}(9)
\]

A path is found ⇒ correct
Lazy LTS creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation
Lazy LTS creation

- We build LTS lazily, like on the previous slides
- We use sequential implementation
Lazy LTS creation

- We build LTS lazily, like on the previous slides
- We use sequential implementation
- Equivalence via `equals/hashCode` implementations

```kotlin
class MSQueueTest {
    val q = MSQueue<Int>()
    // Operations here
    override fun equals(other: Any?) = ...  
    override fun hashCode() = ...
}
```
Lazy LTS creation

- We build LTS lazily, like on the previous slides
- We use sequential implementation
- Equivalence via equals/hashcode implementations

```kotlin
class MSQueueTest: VerifierState() {
    val q = MSQueue<Int>()

    // Operations here

    override fun generateState() = q
}
```
What if our data structure is blocking by design?
val c = Channel<Int>()

_\underline{\text{c.send}(4)} \quad \underline{\text{c.receive}()} // 4_

send waits for receive and vice versa
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
  val elem = c.receive()
  process(elem)
}
```

```scala
val c = Channel()
```
**Producer 1**

\[
\text{val elem = ...} \\
\text{c.send(elem)}
\]

**Producer 2**

\[
\text{val elem = ...} \\
\text{c.send(elem)}
\]

**Consumer**

\[
\text{while(true) { } } \\
\text{val elem = c.receive()} \\
\text{process(elem)} \\
}\]

\[
\text{val c = Channel()}
\]

Has to wait for send
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
  val elem = c.receive()
  process(elem)
}
```

```scala
val c = Channel()
```
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
  val elem = c.receive()
  process(elem)
}
```

val `c = Channel()`
Producer 1

\[
\text{val elem = ...} \\
\text{c.send(elem)}
\]

Producer 2

\[
\text{val elem = ...} \\
\text{c.send(elem)}
\]

Consumer

\[
\text{while(true)} \{ \\
\text{val elem = c.receive()} \\
\text{process(elem)} \\
\}
\]

\[
\text{val c = Channel()}
\]
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
  val elem = c.receive()
  process(elem)
}
```

```scala
val c = Channel()
```
Producer 1

```scala
val elem = ... 
c.send(elem)
```

Producer 2

```scala
val elem = ... 
c.send(elem)
```

Consumer

```scala
while(true) {
  val elem = c.receive()
  process(elem)
}
```

```scala
val c = Channel()
```
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
val elem = c.receive()
process(elem)
}
```

```scala
val c = Channel()
```

Has to wait for receive
Producer 1

val elem = ...
2  c.send(elem)

Producer 2

zzz
4  c.send(elem)

Consumer

while(true) {
1  val elem = c.receive()
3  process(elem)
}

val c = Channel()
Producer 1

```scala
val elem = ...
c.send(elem)
```

Producer 2

```scala
val elem = ...
c.send(elem)
```

Consumer

```scala
while(true) {
val elem = c.receive()
process(elem)
}
```

```scala
val c = Channel()
```

Has to wait for receive
```-scala
val c = Channel<Int>(

c.send(4) // 4

c.receive() // 4
)
```

Non-linearizable because of suspension
val c = Channel<Int>()

c.send(4)
c.receive(): // 4
  register as a waiter
  suspend

c.send(4)
  return element
val c = Channel<Int>()

c.send(4)

c.receive(): // 4
  register as a waiter
  suspend

return element

} request

} follow-up

[1] “Nonblocking Concurrent Data Structures with Condition Synchronization” by Scherer, W.N. and Scott, M.L.
val c = Channel<Int>()
c.receive\textsuperscript{REQ}(): tik
c.send(4)
c.receive\textsuperscript{FUP}(tik): 4

Unique ticket, $\in \mathbb{N}$
Dual Data Structures

```scala
val c = Channel<Int>()
c.receive[^REQ()](): tik
c.send(4)
c.receive[^FUP](tik): 4
```

Follow-ups should be invoked after the corresponding requests
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,>
Dual Data Structures

```kotlin
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

suspended
LTS for Dual Data Structures

```scala
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```
LTS for Dual Data Structures

\[ \text{val } c = \text{Channel}\langle\text{Int}\rangle() \]
\[ c.\text{receive}(0): <s, 1> \]
\[ c.\text{send}(0, 4) \]
\[ c.\text{receive}(1): <4, _> \]
Val c = Channel<Int>()
c.receive(0): <s, 1>
c.send(0, 4)
c.receive(1): <4, _>

LTS for Dual Data Structures
LTS for Dual Data Structures

```
val c = Channel<Int>()
c.receive(0): <s, 1>
c.send(0, 4)
c.receive(1): <4, _>
```

```
rt={1}
```
LTS for Dual Data Structures

\[
\text{val } c = \text{Channel}\langle\text{Int}\rangle() \\
c.\text{receive}(0): <s, 1> \\
c.\text{receive}(1): <4, _> \\
c.\text{send}(0, 4) \\
r =\{1\} \\
c.\text{receive}(1): <4, _>
\]

Looks similar
LTS for Dual Data Structures

val c = Channel<Int>()

c.send(0, 4): <s, 1>

val c = Channel<Int>()
c.send(0, 4): <s, 1>
c.send(0, 4): <s, 2>
c.receive(0): <4, _>
c.send(1, 4)
c.send(0, 4): <s, 1>
c.send(0, 4): <s, 2>
c.receive(0): <4, _>
c.send(1, 4)
The only difference is in tickets.
LTS for Dual Data Structures

The only difference is in tickets.

Let’s forbid such duplicate transitions.
LTS for Dual Data Structures

val c = Channel<Int>()
c.send(0, 4): <s, 1>
c.send(0, 4): <s, 2>
c.receive(0): <4, _>
c.send(1, 4)

rf={2→1}
Verifier for Dual Data Structures

val c = Channel<Int>()

<table>
<thead>
<tr>
<th>c.receive(): 4</th>
<th>c.send(4): s+Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.receive(): s</td>
<td></td>
</tr>
</tbody>
</table>
Verifier for Dual Data Structures

```
val c = Channel<Int>()

val c = Channel<Int>()
c.receive(): 4  c.send(4): $+Unit
c.receive(): $  c.send(4): $ + Unit
```
Verifier for Dual Data Structures

```scala
val c = Channel[Int]()

<table>
<thead>
<tr>
<th>c.receive(): 4</th>
<th>c.send(4): $+Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.receive(): $</td>
<td></td>
</tr>
</tbody>
</table>
```

Results are different
Verifier for Dual Data Structures

```scala
val c = Channel<Int>()

c.receive(): 4

val channel = Channel<Int>()
c.send(4): 4+Unit

c.receive(): 5

c.receive(): 5

suspended, ticket 1
```
Verifier for Dual Data Structures

val c = Channel<Int>()

c.receive(): 4

suspended, ticket 1

resumed

rt={1}

c.send(4): $+Unit

c.receive(): s
Verifier for Dual Data Structures

\[
\text{val } c = \text{Channel<Int}() \\
\text{c.receive(): 4} \\
\text{c.receive(): } s \\
\text{c.send(4): } s + \text{Unit} \\
\text{suspended, ticket 1 resumed}
\]
Verifier for Dual Data Structures

\begin{align*}
\text{val } c & \quad = \quad \text{Channel}\langle\text{Int}\rangle() \\
\text{c.receive}(0) & \quad : \quad \langle s, 1 \rangle \\
\text{c.send}(0, 4) & \quad : \quad \langle s, 1 \rangle \\
\text{c.receive}(0) & \quad : \quad 4 \\
\text{rt} & \quad = \{1\} \\
\text{c.send}(1, 4) & \\
\text{c.receive}(0) & \quad : \quad \langle s, 2 \rangle \\
\end{align*}
Lazy Dual Data Structures LTS creation

```
val c = Channel<Int>()
c.send(0, 4): <s, 1>
c.receive(0): <4, _>
c.send(0, 4): <s, 2>
c.send(1, 4)
rf={2→1}
c.receive(0): <4, _>
```
Lazy Dual Data Structures LTS creation

Let's define as an externally observable state as before

```kotlin
val c = Channel<Int>()
c.receive(0): <4, _>
c.send(0, 4)
c.receive(1): <4, _>
c.send(1, 4)
rf={2→1}
```

Let's define as an externally observable state as before
Lazy Dual Data Structures LTS creation

Let's define as an externally observable state as before

They should be different!

```scala
val c = Channel[Int]()
c.receive(0): <s, 1>
c.send(0, 4): <s, 1>
c.send(0, 4): <s, 2>
c.send(1, 4): {2 → 1}
```
Lazy Dual Data Structures LTS creation

\( st = \text{list of suspended operations} \)

\( rt = \text{set of resumed operations} \)
States are equal iff $\exists f: \mathbb{N} \rightarrow \mathbb{N}$ that

1. externally observable states
2. $st$-s wrt $rf$ on tickets (as lists)
3. $rt$-s wrt $rf$ on tickets (as sets) are equal
Lazy Dual Data Structures LTS creation

States are equal iff \( \exists f: \mathbb{N} \rightarrow \mathbb{N} \) that
1. externally observable states
2. \( st\)-s wrt \( rf \) on tickets (as lists)
3. \( rt\)-s wrt \( rf \) on tickets (as sets)

are equal

maintained by Lin-Check
Channel Test Example

class RendezvousChannelTest: LinCheckState() {
    val c = Channel()

    @Operation suspend fun send(x: Int) = c.send(x)
    @Operation suspend fun receive(): Int = c.receive()

    override fun generateState() = Unit
}

class BufferedChannelTest: LinCheckState() {
    val c = Channel()

    @Operation suspend fun send(x: Int) = c.send(x)
    @Operation suspend fun receive(): Int = c.receive()

    override fun generateState(): Any {
        val state = ArrayList<Int>()
        var x: Int?
        while (true) {
            x = c.poll()
            if (x == null) break
            state += x
        }
        return state
    }
}
Uncovered topics

- Verifiers for several relaxed contracts
- How to run scenarios in the most “dangerous” way
- API
Future plans

- Smart running strategies
- Supporting randomized relaxed contracts
Questions?

https://github.com/Kotlin/kotlinx-lincheck