# POSTER: Testing Concurrency on the JVM with Lincheck

Nikita Koval
JetBrains, IST Austria
nikita.koval@jetbrains.com

Maria Sokolova
JetBrains, IST Austria
maria.sokolova@jetbrains.com

Alexander Fedorov
JetBrains
aleksandr.fyodorov@jetbrains.com

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

Dmitry Tsitelov
Devexperts
cit@devexperts.com

## 1 Introduction

Concurrent programming can be notoriously complex and error-prone. Programming bugs can arise from a variety of sources, such as operation re-reordering, or incomplete understanding of the memory model. A variety of formal and model checking methods have been developed to address this fundamental difficulty. While technically interesting, existing academic methods are still hard to apply to the large codebases typical of industrial deployments, which limits their practical impact.

This paper presents `Lincheck` [1] — a new practical tool for testing concurrent algorithms implemented in JVM-based languages, such as Java, Kotlin, or Scala. Roughly, `Lincheck` takes the list of operations on the data structure to be tested, generates a series of concurrent scenarios, executes them in either stress testing or model checking mode, and checks whether there exists some sequential execution which can explain the results. We use `Lincheck` to test the concurrent algorithms in the Kotlin Coroutines library [2] and to check a set of student assignments. In addition, we used it to find several known and unknown bugs in popular libraries, such as the race between removing and adding an element to the head of the Java's `ConcurrentLinkedDeque`.

## 2 Lincheck by Example

Listing 1 represents an incorrect implementation of counter (lines 1–4) and a sample `Lincheck` test for it (lines 6–12) in Kotlin. Here, we use both stress testing and model checking execution modes. (lines 6–7). The initial state of the examined data structure is specified by constructor; here we simply create a new counter (line 9). Line 10 specifies the only `inc` operation on our counter, which should be marked with `@Operation` annotation. Finally, we run the analysis by invoking `LinChecker.check` function on the testing class (line 11).

```
1  class Counter {
2    var value = 0
3    fun inc(delta: Int) = value += delta //returns new
4  }
5
6  @StressCTest
7  @ModelCheckingCTest
8  class CounterTest {
9    val c = Counter() // init state
10   @Operation fun inc(delta: Int) = c.inc(delta)
11   @Test run() = LinChecker.check(this::class)
12 }
```

**Listing 1.** Counter Test Example

Since the counter above is incorrect, the corresponding test fails with an error similar to the one from Listing 2. While `Lincheck` always provides a failing scenario with the wrong results (if found), the model checking mode also provides a trace to reproduce the error.

```
| inc(1): 1 | inc(2): 2 |
Execution trace:
|           | inc(2)                          |
|           |   R:0  at Counter.inc(Counter.kt:3) |
| inc(1): 1 |                                 |
|           |   W(2) at Counter.inc(Counter.kt:3) |
|           |   RESULT: 2                     |
```

**Listing 2.** Example of Invalid Execution

## 3 Scenarios Generation and Testing

**Scenarios.** Users specify the number of scenarios to be examined, as well as the numbers of threads and operations in them; after that, `Lincheck` generates them in a random way. It worth noting that we support some popular constraints like single producer or consumer for queues, for which the corresponding operations appear only in one thread. Note, that the `inc(..)` operation in Listing 1 takes an integer parameter `delta` which specifies the increment amount. At the same time, Listing 2 provides a specific scenario, with some input values to the `inc(..)` invocations; they are also generated using a random function on the specified range (it can be configured for each parameter).

**Stress Testing.** This mode was inspired by the JCStress harness for testing Java Memory Model on predefined


*PPoPP '20, February 22–26, 2020, San Diego, CA, USA*

ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374503

scenarios [3]. Essentially, it fairly starts real threads, synchronizes them, and executes the operations, repeating the run the specified number of times in hope to hit an interleaving that produces incorrect results. In Lincheck, we also added random busy-wait cycles between the operations and inside them to increase the number of different interleavings. We find this mode very useful to check algorithms for bugs introduced by low-level effects, such as a missed `volatile` modifier.

**Model Checking.** Since we developed Lincheck to test our non-blocking algorithms, most of which de-facto use the sequential consistency memory model, we added a model checking mode, which was originally inspired by the CHESS framework for C# [4], which studies all possible schedules with a bounded number of context switches. In this mode, we ignore weak memory model effects; it is recommended to use both stress testing and model checking in practice. In Lincheck, we specify the total number of interleavings to be studied, so that the number of context switches increase during the analysis. Thus, Lincheck finds an incorrect schedule with the fewest number context switches possible.

Internally, we iteratively construct a tree of all possible interleavings with progress information for each node. An example of such a tree for the scenario from Listing 1 is presented in Figure 1. Each edge in this tree represents a choice that can be done by the scheduler, while leaves represent the resulting interleavings. In order to explore interleavings evenly, we store the percentage of examined interleaving for each subtree, and use weighted random choices to choose the next switch point. Thus, the probability to start the next interleaving from the first thread is two times higher than from the second.

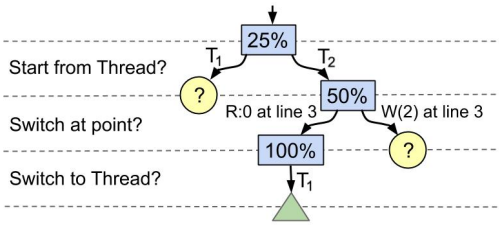
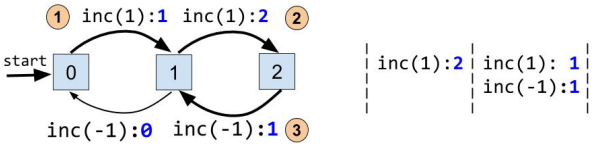



**Figure 1.** A partially built tree of possible schedules with one contexts switch for the scenario from Listing 1. Rectangles are nodes with choices, circles are unexplored nodes, triangles are interleavings.

As for implementation, we use byte-code instrumentation [5] to invoke the analysis after each shared variable access; thus, users should not modify their algorithms in order to use model checking. Moreover, we implemented several heuristics to avoid useless switches.

## 4 Results Verification

In order to check whether the results are correct, Lincheck tries to explain them though some sequential execution which does not reorder operation in threads. We define the sequential semantics via building a labeled transition system (LTS): the states represent the data structure states and the transitions are labeled with operations and the corresponding results.




The figure above illustrates the process of LTS-based results verification for the counter. Starting from the initial state, Lincheck tries to make a transition by an operation from one of the threads; if the results of the verifying step and LTS coincide, it repeats the procedure recursively. Thus, it finds sequential executions which produce the verifying results, or guarantees that there is no such one.

**LTS Construction.** Since LTS is infinite, Lincheck builds transitions lazily, invoking operations of the provided concurrent implementation in a sequential way; it is also possible to specify a simpler sequential implementation instead. Note, that several transition sequences may lead to the same state. For example, applying `inc(-1)` after the first `inc(1)` leads to the initial state. Since Lincheck does not know the formal specification of the testing data structure, it is tricky to explain why these two increments lead to the same state as the initial one. As a solution, users should define equivalency relation between states by implementing `equals()` and `hashCode()` methods.

**Other Contracts.** In addition to linearizability, we have supported the *dual data structures* formalism [6] for blocking by design operations (e.g., `remove` in blocking queues) and several relaxed contracts, such as serializability, quiescent consistency, quasi-linearizability [7], and quantitative relaxation [8].

## References


[1] Lincheck. https://github.com/Kotlin/kotlinx-lincheck.

[2] Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines.

[3] The Java Concurrency Stress tests. https://openjdk.java.net/projects/code-tools/jcstress.

[4] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, June 2007.

[5] ObjectWeb ASM. https://asm.ow2.io.

[6] W.N. Scherer III and M.L. Scott. Nonblocking concurrent objects with condition synchronization. In *Proceedings of the 18th International Symposium on Distributed Computing*, 2004.

[7] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[8] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, volume 48. ACM, 2013.