About lin-check

lin-check is a tool for Java that checks linearizability on concurrent data structures.

GitHub Repository	https://github.com/Devexperts/lin-check
Public Maven repo	-
License	MPL 2.0 (since Lincheck v3.0)
Contact	dxlab@devexperts.com

README.md (source from lin-check)

Lin-Check

NOTE: Check https://github.com/Kotlin/kotlinx-lincheck for the most recent version with extended Kotlin support.

LICENSE CHANGE: Starting from version **3.0** the product is distributed under the **MPL 2.0** license.

Lin-Check is a framework for testing concurrent data structures for correctness. In order to use the framework, operations to be executed concurrently should be specified with the necessary information for an execution scenario generation. With the help of this specification, **Lin-Check** generates different scenarios, executes them in concurrent environment several times and then checks that the execution results are correct (usually, linearizable, but different relaxed contracts can be used as well).

Table of contents

Test structure

The first thing we need to do is to define operations to be executed concurrently. They are specified as `public` methods with an `@Operation` annotation in the test class. If an operation has parameters, generators for them have to be specified. The second step is to set an initial state in the empty constructor. After the operations and the initial state are specified, **Lin-Check** uses them for test scenarios generations and runs them.

Initial state

In order to specify the initial state, the empty argument constructor is used. It is guaranteed that before every test invocation a new test class instance is created.

Operations and groups

As described above, each operation is specified via `@Operation` annotation.

```
@Operation
public Integer poll() { return q.poll(); }
```

Calling at most once

If an operation should be called at most once during the test execution, you can set `@Operation(runOnce = true)` option and this operation appears at most one time in the generated scenario.

Exception as a result

If an operation can throw an exception and this is a normal result (e.g. `remove` method in `Queue` implementation throws `NoSuchElementException` if the queue is empty), it can be handled as a result if `@Operation(handleExceptionsAsResult = ...)` options are specified. See the example below where `NoSuchElementException` is processed as a normal result.

```
@Operation(handleExceptionsAsResult = NoSuchElementException.class)
public int remove() { return queue.remove(); }
```

Operation groups

In order to support single producer/consumer patterns and similar ones, each operation could be included in an operation group. Then the operation group could have some restrictions, such as non-parallel execution.

In order to specify an operation group, `@OpGroupConfig` annotation should be added to the test class with the specified group name and its configuration:

- **nonParallel** - if set all operations from this group will be invoked from one thread.

Here is an example with single-producer multiple-consumer queue test:

```
@OpGroupConfig(name = "producer", nonParallel = true)
public class SPMCQueueTest {
  private SPMCQueue<Integer> q = new SPMCQueue<>();

  @Operation(group = "producer")
  public void offer(Integer x) { q.offer(x); }

  @Operation
  public Integer poll() { return q.poll(); }
}
```

> *A generator for `x` parameter is omitted and the default is used. See Default generators paragraph for details.*

Parameter generators

If an operation has parameters then generators should be specified for each of them. There are several ways to specify a parameter generator: explicitly on parameter via `@Param(gen = ..., conf = ...)` annotation, using named generator via `@Param(name = ...)` annotation, or using the default generator implicitly.

For setting a generator explicitly, `@Param` annotation with the specified class generator (`@Param(gen = ...)`) and string configuration (`@Param(conf = ...)`) should be used. The provided generator class should be a `ParameterGenerator` implementation and can be implemented by user. From the box **Lin-Check** supports random parameter generators for almost all primitives and strings. Note that only one generator class is used for both primitive and its wrapper, but boxing/unboxing does not happen. See `com.devexperts.dxlab.lincheck.paramgen` for details.

It is also possible to use once configured generators for several parameters. This requires adding this `@Param` annotation to the test class instead of the parameter specifying it's name (`@Param(name = ...)`). Then it is possible to use this generator among all operations using `@Param` annotation with the provided name only. It is also possible to bind parameter and generator names, see Binding parameter and generator names for details.

If the parameter generator is not specified **Lin-Check** tries to use the default one, binding supported primitive types with the existent generators and using the default configurations for them.

Binding parameter and generator names

Java 8 came with the feature (JEP 188) to store parameter names to class files. If test class is compiled this way then they are used as the name of the already specified parameter generators.

For example, the two following code blocks are equivalent.

```
@Operation
public Integer get(int key) { return map.get(key); }
```

```
@Operation
public Integer get(@Param(name = "key") int key) {
  return map.get(key);
}
```

Unfortunately, this feature is disabled in **javac** compiler by default. Use `-parameters` option to enable it. In **Maven** you can use the following plugin configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
    </configuration>
</plugin>
```

> *However, some IDEs (such as IntelliJ IDEA) do not understand build system configuration as well as possible and running a test from these IDEs will not work. In order to solve this issue you can add `-parameters` option for **javac** compiler in your IDE configuration.*

Run test

In order to run a test, `LinChecker.check(...)` method should be executed with the provided test class as a parameter. Then **Lin-Check** looks at execution strategies to be used, which can be provided using annotations or options (see Configuration via options for details), and runs a test with each of provided strategies. If an error is found, an `AssertionError` is thrown and the detailed error information is printed to the standard output. It is recommended to use **JUnit** or similar testing library to run `LinChecker.check(...)` method.

```
@StressCTest // stress execution strategy is used
public class MyConcurrentTest {
  <empty constructor and operations>

  @Test
  public void runTest() {
    LinChecker.check(MyConcurrentTest.class);
  }
}
```

> *It is possible to add several `@..CTest` annotations with different execution strategies or configurations and all of them should be processed.*

Execution strategies

The section above describes how to specify the operations and the initial state, whereas this section is about executing the test. Using the provided operations **Lin-Check** generates several random scenarios and then executes them using the specified execution strategy. At this moment stress strategy is implemented only, but some managed strategies will be added soon as well.

Stress strategy

The first implemented in **Lin-Check** execution strategy is stress testing strategy. This strategy uses the same idea as `JCStress` tool - it executes the generated scenario in parallel a lot of times in hope to hit on an interleaving which produces incorrect results. This strategy is pretty useful for finding bugs related to low-level effects (like a forgotten volatile modifier), but, unfortunately, does not guarantee any coverage. It is also recommended to use not only Intel processors with this strategy because its internal memory model is quite strong and cannot produce a lot of behaviors which are possible with ARM, for example.

In order to use this strategy, just `@StressCTest` annotation should be added to the test class or `StressOptions` should be used if the test uses options to run (see Configuration via options for details). Both of them are configured with the following options:

- **iterations** - number of different scenarios to be executed;
- **invocationsPerIteration** - number of invocations for each scenario;
- **threads** - number of threads to be used in a concurrent execution;
- **actorsPerThread** - number of operations to be executed in each thread;
- **actorsBefore** - number of operations to be executed before the concurrent part, sets up a random initial state;
- **actorsAfter** - number of operations to be executed after the concurrent part, helps to verify that a data structure is still correct;
- **verifier** - verifier for an expected correctness contract (see Correctness contracts for details).

Correctness contracts

Once the generated scenario is executed using the specified strategy, it is needed to verify the operation results for correctness. By default **Lin-Check** checks the result for linearizability, which is de-facto a standard type of correctness. However, there are also verifiers for some relaxed contracts, which should be set via `@..CTest(verifier = ..Verifier.class)` option.

Linearizability

Linearizability is a de-facto standard correctness contract for thread-safe algorithms. It means that an execution is equivalent to some operations sequence which produces the same results and does not avoid happens-before order. The `LinearizabilityVerifier` is used by default to check for this correctness type.

The verifier lazily constructs a transition graph, where states are test instances and edges are operations. Then it tries to find a path which does not violate the happens-before order and produces same results on operations. In order not to have state duplicates, it is better to implements `equals(..)` and `hashCode()` methods.

Serializability

Serializability is one of the base contracts, which ensures that an execution is equivalent to one that invokes operations in any serial order. The `SerializabilityVerifier` is used for this contract.

Alike linearizability verification, it also constructs a transition graph and expects `equals(..)` and `hashCode()` methods overrides.

Quiescent consistency

Quiescent consistency is a stronger guarantee than serializability but still relaxed comparing to linearizability. It ensures that an execution is equivalent to some operations sequence which produces the same results and does not reorder operation between quiescent points. Quiescent point is a cut where all operations before the cut are happens-before all operations after it. In order to check for this consistency, use `QuiescentConsistencyVerifier` and mark all quiescent consistent operations with `@QuiescentConsistent` annotation, all other operations are automatically linearizable.

Alike linearizability verification, it also constructs a transition graph and expects `equals(..)` and `hashCode()` methods overrides.

```
@StressCTest(verifier = QuiescentConsistencyVerifier.class)
public class QuiescentQueueTest {
  private QuiescentQueue<Integer> q = new QuiescentQueue<>();

  // Only this operation is quiescent consistent
  @QuiescentConsistent
  public Integer poll() {
    return q.poll();
  }

  @Operation
  public boolean offer(Integer val) {
    return q.offer(val);
  }

  @Test
  public void test() {
    LinChecker.check(QuiescentQueueTest.class);
  }

  // equals(..) and hashCode() here
}
```

Quantitative relaxation

One more trade-off contract is suggested by T. Henzinger et al., which relaxes a data structure semantics. Instead of allowing some reorderings, they suggest to allow "illegal" by the data structure specification results, but with a penalty. This penalty is called a transition cost and then is used to count a path cost, which should be less then a relaxation factor if an execution is correct.

> *Look at this paper for details: Henzinger, Thomas A., et al. "Quantitative relaxation of concurrent data structures." ACM SIGPLAN Notices. Vol. 48. No. 1. ACM, 2013.*

In order to describe the contract of a testing data structure, the transition costs, the path cost function and the relaxation factor should be specified.

At first, all relaxed operations should be annotated with `@QuantitativeRelaxed`. The current version of **Lin-Check** counts a path cost using all relaxed operations, but grouping is going to be introduced later.

Then the special cost counter class have to be defined. This class represents a current data structure state and has the same methods as testing operations, but with an additional `Result` parameter and another return type. If an operation is not relaxed this cost counter should check that the operation result is correct and return the next state (which is a cost counter too) or `null` in case the result is incorrect. Otherwise, if a corresponding operation is relaxed (annotated with `@QuantitativeRelaxed`), the method should return a list of all possible next states with their transition cost. For this purpose, a special `CostWithNextCostCounter` class should be used. This class contains the next state and the transition cost with the predicate value, which are defined in accordance with the original paper. Thus, `List<CostWithNextCostCounter>` should be returned by these methods and an empty list should be returned in case no transitions are possible. In order to restrict the number of possible transitions, the relaxation factor should be used. It is provided via a constructor, so **Lin-Check** uses the `(int relaxationFactor)` constructor for the first instance creation.

The last thing to do is to provide the relaxation factor, the cost counter class, and the path cost function to the verifier. For this purpose, the test class should have an `@QuantitativeRelaxationVerifierConf(...)` annotation, which is then used by `QuantitativeRelaxationVerifier`. As for the path cost function, `MAX`, `PHI_INTERVAL`, and `PHI_INTERVAL_RESTRICTED_MAX` are implemented in `PathCostFunction` class in accordance with the past cost functions in the original paper.

Here is an example for k-stack with relaxed `pop()` operation and normal `push` one:

```
@StressCTest(verifier = QuantitativeRelaxationVerifier::class)
@QuantitativeRelaxationVerifierConf(
  factor = K,
  pathCostFunc = MAX,
  costCounter = KRelaxedPopStackTest.CostCounter::class
)
class KRelaxedPopStackTest {
  private val s = KRelaxedPopStack<Int>(K)

  @Operation
  fun push(x: Int) = s.push(x)

  @QuantitativeRelaxed
  @Operation
  fun pop(): Int? = s.pop()

  @Test
  fun test() = LinChecker.check(KRelaxedPopStackTest::class.java)

  // Should have '(k: Int)' constructor
  data class CostCounter @JvmOverloads constructor(
    private val k: Int,
    private val s: List<Int> = emptyList()
  ) {
    fun push(value: Int, result: Result): CostCounter {
       check(result.type == VOID)
       val sNew = ArrayList(s)
       sNew.add(0, value)
       return CostCounter(k, sNew)
    }

    fun pop(result: Result): List<CostWithNextCostCounter<CostCounter>> {
      if (result.value == null) {
        return if (s.isEmpty())
          listOf(CostWithNextCostCounter(this, 0))
        else emptyList()
      }
      return (0..(k - 1).coerceAtMost(s.size - 1))
        .filter { i -> s[i] == result.value }
        .map { i ->
          val sNew = ArrayList(s)
          sNew.removeAt(i)
          CostWithNextCostCounter(CostCounter(k, sNew), i)
        }
    }
  }
}
```

Configuration via options

Instead of using `@..CTest` annotations for specifying the execution strategy and other parameters, it is possible to use `LinChecker.check(Class<?>, Options)` method and provide options for it. Every execution strategy has its own Options class (e.g., `StressOptions` for stress strategy) which should be used for it. See an example with stress strategy:

```
public class MyConcurrentTest {
  <empty constructor and operations>

  @Test
  public void runTest() {
    Options opts = new StressOptions()
        .iterations(10)
        .threads(3)
        .logLevel(LoggingLevel.INFO);
    LinChecker.check(StressOptionsTest.class, opts);
  }
}
```

Sample

Here is a test for a not thread-safe `HashMap` with its result. It uses the default configuration and tests `put` and `get` operations only:

Test class

```
@Param(name = "key", gen = IntGen.class, conf = "1:5")
@StressCTest
public class HashMapLinearizabilityTest {
    private HashMap<Integer, Integer> map = new HashMap<>();;

    @Operation
    public Integer put(@Param(name = "key") int key, int value) {
        return map.put(key, value);
    }

    @Operation
    public Integer get(@Param(name = "key") int key) {
        return map.get(key);
    }

    @Test
    public void test() {
        LinChecker.check(HashMapLinearizabilityTest.class);
    }

    // 'map' field is included in equals and hashCode
    @Override public boolean equals(Object o) { ... }
    @Override public int hashCode() { ... }
}
```

Test output

```
= Invalid execution results: =
Execution scenario (init part):
[get(4), get(9), get(5), get(8), get(3)]
Execution scenario (parallel part):
| put(1, 4) | get(4)    |
| get(3)    | put(3, 6) |
| get(4)    | put(6, 1) |
| put(6, 4) | get(8)    |
| get(8)    | put(3, 0) |
Execution scenario (post part):
[get(3), get(6), get(4), put(9, -10), put(6, 10)]

Execution results (init part):
[null, null, null, null, null]
Execution results (parallel part):
| null | null |
| null | null |
| null | null |
| 1    | null |
| null | null |
Execution results (post part):
[0, 4, null, null, 4]
```

Contacts

If you need help, you have a question, or you need further details on how to use **Lin-Check**, you can refer to the following resources:

- dxLab research group at Devexperts
- GitHub issues

You can also use the following e-mail to contact us directly: