Linearizability Testing Manual

Gavin Lowe

April 8, 2016

This manual describes how to use the linearizability testing framework, described in [Low16]. The framework is available from `http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/`.

We assume here that the reader is familiar with the notion of linearizability [HW90]. We also assume familiarity with Scala [OSV08].

The framework can be used to test whether a concurrent datatype is linearizable. The basic idea is to run a number of worker threads upon the datatype, recording the history of invocation and return events, and then to test whether that history is linearizable.

1 Defining linearizability testers

Several algorithms for testing whether a history is linearizable are described in [Low16]. The following sections describe how to use these algorithms within the framework. The normal algorithm to use on a concurrent queue is the queue-oriented algorithm (Section 1.4). The normal algorithm for other datatypes is either the JIT Graph Search Algorithm (Section 1.1) or the JIT-JIT Competition Tester 1.3): the latter is faster, but requires a little more programming to use.

Most of the algorithms make use of a corresponding deterministic sequential specification datatype S. The tester tests whether the concurrent datatype is linearizable with respect to S.

1.1 Graph search linearizability testers

Some of the algorithms perform a graph search through an appropriate search space, i.e. they store all nodes of that search space that have been previously encountered, and so avoid repeating work. These algorithms make use of an *immutable* sequential specification datatype S.

```
1  object JITGraphTester{
2    type C = LockFreeQueue[Int]; type S = scala.collection.immutable.Queue[Int]
3    def seqEnqueue(x: Int)(q: S): (Unit, S) = ((), q.enqueue(x))
4    def seqDequeue(q: S): (Option[Int], S) =
5      if(q.isEmpty) (None, q) else{ val (v,q1) = q.dequeue; (Some(v), q1) }
6    def worker(me: Int, theLog: GenericThreadLog[S, C]) = {
7      for(i <- 0 until 1000)
8        if(Random.nextFloat <= 0.3){
9          val x = Random.nextInt(20)
10         theLog.log(_.enqueue(x), "enqueue("+x+")", seqEnqueue(x)) }
11       else theLog.log(_.dequeue, "dequeue", seqDequeue)
12   }
13   def main(args: Array[String]) = {
14     for(i <- 0 until 1000){
15       val concQueue = new LockFreeQueue[Int] // The shared concurrent queue
16       val seqQueue = Queue[Int]() // The sequential specification queue
17       val tester =
18         LinearizabilityTester.JITGraph[S, C](seqQueue, concQueue, 6, worker, 1000)
19       assert(tester() > 0)
20     }
21   }
22 }
```

Figure 1: An example testing program using the JIT Graph tester.

To illustrate how the framework can be used, Figure 1 gives a stripped-down testing program applied to a queue (a fuller version could be used with multiple concurrent queue implementations, and would replace the numerical constants by variables, specifiable via the command line).

The test program works on a concurrent datatype of some type C; here we use a lock-free queue containing Ints, based on [HS12, Section 10.5]. A dequeue operation returns a value of type Option[Int]: either None to indicate an empty queue, or a value of the form Some(x) to indicate a successful dequeue of x.

The test program also requires a corresponding, immutable, deterministic sequential specification datatype S. Further, the specification datatype must support equality tests that correspond to semantic equality, and have a compatible hashCode [OSV08, Chapter 28]. Here we use an immutable queue from the Scala API[1]. (In practice, most such sequential datatypes are from the Scala API.)

For each operation op : A on the concurrent datatype, we need a corre-

[1] `www.scala-lang.org/api/current/#scala.collection.immutable.Queue`

sponding function seqOp : S => (A, S) on the sequential datatype, which returns the same value as the concurrent operation, paired with the new value of the sequential datatype. These are normally simple wrappers around API code (see lines 3–5).

The main part of the test program is the definition of a worker function that performs and logs operations on the concurrent datatype. The worker function takes as parameters the identity of the worker, and a log object theLog of type GenericThreadLog[S, C]. Here, the worker performs 1000 operations; each is (with probability 0.3) an enqueue of a random value, or (with probability 0.7) a dequeue. Each operation is performed and logged via a call to theLog.log, taking three parameters:

- the operation to be performed on the concurrent datatype;
- a string describing the operation; this is used in debugging output in the case that a non-linearizable history is found; it is also used internally; semantically different operations should have different strings; and
- the corresponding operation on the sequential datatype.

The call to theLog.log logs the invocation of the concurrent operation, performs the operation, and logs the result returned.

The linearizability tester is constructed at line 18 and run at line 19; here we use the JIT Graph Search Algorithm. The function to construct the tester takes as arguments: the sequential datatype; the concurrent datatype; the number p of worker threads to run; the definition of a worker thread; and the number of operations performed by each worker. The test for linearizability itself is performed at line 19, repeated to consider 1000 histories. The linearizability tester runs p workers concurrently, logging the operation calls on the concurrent datatype. It then tests whether the resulting history is linearizable, returning a positive result if so, or giving debugging output if not.

The Wing & Gong Graph Search Algorithm can be used very similarly. The only change is in the construction of the linearizability tester:

```
val tester =
  LinearizabilityTester.WGGraph[S, C](seqQueue, concQueue, 6, worker, 1000)
```

The parameters are as for JITGraph. In practice, the JIT Graph Search Algorithm tends to be faster.

1.2 Tree search linearizability testers

Two of the linearizability algorithms perform a tree search, that is, they do not store which states have been encountered previously, and so might repeat

```
1  object JITTreeTester{
2    type C = LockFreeQueue[Int]; type US = UndoableQueue[Int]
3    def worker(me: Int, theLog: GenericThreadLog[US, C]) = {
4      for(i <− 0 until 1000)
5        if(Random.nextFloat <= 0.3){
6          val x = Random.nextInt(20)
7          theLog.log(_.enqueue(x), "enqueue("+x+")", _.enqueue(x))
8        }
9        else theLog.log(_.dequeue, "dequeue ", _.dequeue)
10   }
11   def main(args: Array[String]) = {
12     for(i <− 0 until 1000){
13       val concQueue = new LockFreeQueue[Int]
14       val seqQueue = new UndoableQueue[Int]()
15       val tester =
16         LinearizabilityTester.JITTree[US, C](seqQueue, concQueue, 6, worker, 1000)
17       assert(tester() > 0)
18     }
19   }
20 }
```

Figure 2: An example testing program using the JIT Tree Search tester.

work.

These algorithms use an undoable sequential specification datatype. That is, the specification datatype has to extend the trait scala.collection.mutable.Undoable, and so provide an undo method that undoes the last operation that has not already been undone.

These algorithms are faster than the algorithms of the previous section on most histories; however, sometimes they hit bad cases, where they fail to terminate within a reasonable amount of time. In practice, they are normally used in competition parallel with a graph search algorithm; see next section.

A linearizability tester based on the JIT Tree Search algorithm is in Figure 2. Here we take the specification datatype to be an UndoableQueue (the definition is included with the distribution). Most of the rest of the definition is as in the previous section.

Likewise, a linearizability tester based on the Wing & Gong Tree Search Algorithm can be used in the same way, with the tester constructed using the function LinearizabilityTester.WGTree. In practice, the JIT Tree Search Algorithm is normally faster than the Wing & Gong Algorithm.

```
1  object JITJITCompTester{
2    type C = LockFreeQueue[Int]; type S = scala.collection.immutable.Queue[Int]
3    type US = scala.collection.immutable.Queue[Int]
4    def seqEnqueue(x: Int)(q: S): (Unit, S) = ((), q.enqueue(x))
5    def seqDequeue(q: S): (Option[Int], S) =
6      if(q.isEmpty) (None, q) else{ val (v,q1) = q.dequeue; (Some(v), q1)
7    def compWorker(me: Int, theLog: CompetitionThreadLog[S,US,C]) = {
8      for(i <- 0 until 1000)
9        if(Random.nextFloat <= 0.3){
10         val x = Random.nextInt(20)
11         theLog.log(_.enqueue(x), "enqueue "+x, seqEnqueue(x), _.enqueue(x))
12       }
13       else theLog.log(_.dequeue, "dequeue ", seqDequeue, _.dequeue)
14   }
15   def main(args: Array[String]) = {
16     for(i <- 0 until 1000){
17       val concQueue = new LockFreeQueue[Int] // The shared concurrent queue
18       // The immutable and undoable sequential specification queues
19       val imSeqQueue = Queue[Int](); val uSeqQueue = new UndoableQueue[Int]()
20       val tester = CompetitionTester.JITJIT(
21         compWorker, 6, 1000, concQueue, imSeqQueue, uSeqQueue)
22       assert(tester() > 0)
23     }
24   }
25 }
```

Figure 3: An example testing program using competition parallel composition of the JIT Graph and Tree Search algorithms.

1.3 Competition testers

Often the fastest linearizability testers are based on the competition parallel composition of a graph search and a tree search algorithm. Both algorithms are run on the history; when one terminates, the other is interrupted. Often the tree search algorithm is faster; but when the tree search algorithm hits a bad case, the graph search algorithm still normally terminates within a reasonable amount of time.

A linearizability tester using the competition parallel composition of the JIT Graph and Tree Search algorithms is in Figure 3. It requires both an immutable sequential specification datatype S, and an undoable one US. The log function takes as arguments the corresponding operations on each of the specification datatypes.

```
1  object QueueTester{
2    type C = LockFreeQueue[Int]
3    def worker(me: Int, theLog: QueueThreadLog[Int, C]) = {
4      for(i <− 0 until 1000)
5        if(Random.nextFloat <= 0.3){
6          val x = Random.nextInt(20); theLog.logEnqueue(x, _.enqueue(x))
7        }
8        else theLog.logDequeue(_.dequeue)
9    }
10   def main(args: Array[String]) = {
11     for(i <− 0 until 1000){
12       val concQueue = new LockFreeQueue[Int] // The shared concurrent queue
13       val tester = QueueLinTester[Int, C](concQueue, 6, worker, 1000)
14       assert(tester() > 0)
15     }
16   }
17 }
```

Figure 4: An example testing program using the queue-oriented tester.

Other competition parallel testers can be obtained using CompetitionTester.JITWG, CompetitionTester.WGJIT or CompetitionTester.WGWG: in each case, the first-named algorithm is the tree search algorithm.

1.4 The queue-oriented tester

The testers seen so far have been generic, in the sense that they can be used with any concurrent datatype for which there is an appropriate sequential specification datatype. By contrast, the queue linearizability tester can be used only with concurrent queues, specifically where the dequeue operation returns a value of type Option[A], as above (if the concurrent queue does not provide this interface, it is normally possible to use wrapping code to adapt it).

Figure 4 gives an example using this tester. Each worker takes a log of type QueueThreadLog[A, C] where A is the type of data stored in the queue, and C is the type of the concurrent queue. Enqueues are logged using the logEnqueue function, which takes as parameters the value being enqueued and the operation on the concurrent queue. Dequeues are logged using the logDequeue function, which takes as parameters the operation on the concurrent queue.

2 Pragmatics

As with any testing framework, some thought in designing tests can make it more likely that bugs (should they exist) are found. For example, consider a hash table; we identify three classes of bugs.

- Bugs based upon concurrent operations on the same key: to find such bugs, we want to maximise the frequency of such concurrent operations, by choosing a small key space, just one or two keys.
- Bugs concerning resizing: by contrast with the previous case, we need to choose a larger key space, large enough that runs will contain resizes.
- Bugs concerning hash collisions between different keys: in order to find such bugs, we can define a type of keys with a bad hash function, for example one that produces only one or two different results.

More generally, if the implementation performs comparisons upon some underlying datatype, such as the type of keys or hashes, then choosing a small value for that type makes it more likely to find that bug. (On the other hand, if there are no comparisons on an underlying type, as is the case with the type of data in most map implementations, choosing a larger value for the type can make it easier to understand any debugging trace: it can be harder to interpret the trace if the same value appears in two different circumstances.)

Now consider a queue.

- Some bugs may manifest themselves only when the queue is empty. To find these, we should run tests where the queue will frequently be empty, which will be the case when the probability of doing an enqueue is less than about 0.5.
- For a *bounded* queue, some bugs may manifest themselves only when the queue is full. To find such bugs, we should run tests where the queue will frequently be full, which will be the case if we choose a fairly small bound on the size of the queue, and choose the probability of doing an enqueue to be more than about 0.5. (By contrast, when testing an unbounded queue, it seems unlikely that bugs will manifest themselves only when the queue becomes particularly large.)

Similar observation apply to a stack.

Our experiments (on an eight-core machine) suggest that the linearizability testers find bugs most quickly when there are five or six workers. Further, they seem to find bugs more quickly when run on a "noisy" machine, with

other applications running at the same time. This is because the other applications will mean that the workers have to compete for CPU time, and so will be frequently descheduled; this erratic scheduling makes certain bugs manifest themselves more frequently. Running two copies of the linearizability tester simultaneously can be effective, since each provides noise for the other.

Some of the linearizability testers can be slow on long histories. Our experiments suggest that taking histories to contain about 8000 operations in total often gives the greatest throughput (in terms of operations per second). But it can be better to start with shorter histories, and then to experiment with different lengths.

For a datatype like a stack, if two values are pushed concurrently, they may be linearized in any order. However, that order will be revealed only when one of them is popped. This causes the size of the search space to grow exponentially in the number of values stored in the stack. It is therefore a good idea to keep the size of the stack small, by arranging for most operations to be pops. Similar comments apply for queues, except with the queue-oriented algorithm, which is much more robust.

References

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Low16] Gavin Lowe. Testing for linearizability. Submitted for publication, 2016.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.