Testing for Linearizability

Gavin Lowe

Department of Computer Science, University of Oxford. `gavin.lowe@cs.ox.ac.uk`

SUMMARY

Linearizability is a well-established correctness condition for concurrent datatypes. Informally, a concurrent datatype is linearizable if operation calls appear to have an effect, one at a time, in an order that is consistent with a sequential (specification) datatype, with each operation taking effect between the point at which it is called and when it returns. We present a testing framework for linearizabilty. The basic idea is to generate histories of the datatype randomly, and to test whether each is linearizable. We consider five algorithms —one existing, and four new— for testing whether a history of a concurrent datatype implementation is linearizable. Four of these are generic: they will work with any concurrent datatype for which there is a corresponding sequential specification datatype. The fifth considers specifically a history of a concurrent queue. We also combine algorithms in competition parallel in various ways. We perform an experimental evaluation of the different algorithms. We illustrate that the framework is very effective in finding bugs, and discuss the pragmatics of using the framework. Copyright © 0000 John Wiley & Sons, Ltd.

1. INTRODUCTION

Linearizability [14] is a well-established and accepted correctness condition for concurrent datatypes. Informally, a concurrent datatype C is linearizable with respect to a sequential (specification) datatype S if every history c of C (i.e. a sequence of operation calls and returns) is linearizable to a history s of S: c can be reordered to produce s, but this reordering respects the order of non-overlapping operation calls. Put another way, the operations of c seem to take effect, one at a time, in the order described by s, but each operation takes effect between the point at which it is called and when it returns. We give the formal definition below.

In this paper we show how to test whether a given concurrent datatype implementation is linearizable with respect to a given sequential datatype. This question was previously considered by Wing and Gong [29]. Their approach is to record a history of a concurrent object, and then to search for a corresponding sequential history. However, as we shall show in Section 6, their algorithm can often perform poorly, failing to terminate in a reasonable amount of time. In this paper we present four new algorithms for determining linearizability of a history. One is an adaptation of Wing and Gong's algorithm; the other three are rather different. The Wing and Gong algorithm and three of our new algorithms are generic: they can be applied to a history of an arbitrary datatype for which there is a corresponding deterministic sequential specification datatype. The final new algorithm applies specifically to histories of a queue.

We have found the testing framework very useful in practice. In the work of [21], and in preparing our masters course on concurrent datatypes, we have used the framework to find a number of bugs: both silly bugs, but also a number of subtle bugs involving particular interleavings between threads, which can take hours to spot without guidance. The framework is easy to use: creating a new testing program takes a few minutes; adapting it to a different datatype with the same interface is

trivial. Our students have also found the framework easy to use. Normally, bugs (if they exist) are discovered within 20 seconds, often less than a second. The framework can also detect deadlocks (simply by the execution being seen not to terminate).

We see testing as complementary to formal verification (we discuss verification-based approaches to linearizability below). Testing is far simpler and quicker to carry out, and needs no special expertise. On the other hand, verification gives stronger guarantees, but is distinctly non-trivial. Current tools fall short of being fully automatic. Further, they can analyse only rather small instances; in particular, we are not aware of any application to a hash table that uses resizing (resizing is often very difficult to get right). We believe that some of the existing verification-based approaches to linearizability would benefit from the algorithms considered in this paper. If one is going to perform formal verification, it is worth carrying out testing first, to be reasonably confident that the implementation is correct.

In the next section we present the formal definition of linearizability. In Sections 3 and 4 we present the four generic linearizability testers. In Section 5 we present our queue-oriented linearizability tester. In Section 6 we carry out experiments to compare the testers. In Section 7 we describe the testing framework and how to use it. In Section 8 we consider the use of the testers in finding errors: we discuss pragmatics for designing testing programs, discuss one bug found using the framework in detail, briefly describe some other bugs found, and give experimental results concerning the speed with which bugs are found. We sum up in Section 9. All the code related to this paper is available from `http://www.cs.ox.ac.uk/people/gavin.lowe/ LinearizabiltyTesting/`.

Related Work. There has been surprisingly little previous work about testing for linearizability, beyond the work by Wing and Gong [29]. Kingsbury [16] describes a linearizability tester, oriented towards histories recorded from concurrent databases; however, his algorithms are several orders of magnitude slower than the ones in this paper. Pradel and Gross [22] generate tests on concurrent datatypes, in which threads call operations on a shared datatype; they detect exceptions or deadlocks that cannot be exhibited by any linearized execution of the test, but do not detect other errors where results diverge from linearized executions. In [23], they test a class against its superclass, and detect behaviours that are not allowed by the superclass; thus the approach is limited to classes where there is a superclass for which the subclass is intended to be substitutable. If every behaviour of the subclass is allowed by the superclass, and the superclass is linearizable, then so is the subclass; however, if the subclass has linearizable behaviours that are not allowed by the superclass, the test will fail. They report that it takes an average of 108 minutes to find an output-diverging error, considerably slower than our approach.

Valgrind [25] is a general-purpose debugging and profiling program. In particular, Helgrind is a component that detects data races in programs that use the POSIX pthreads threading primitives. This is somewhat different from our approach: some datatypes contain "benign" data races that do not constitute errors; our approach identify only races that lead to errors.

By contrast, there have been more verification-based approaches to linearizability. Burckhardt et al. [4] check for linearizability using a model checker, but using a technique that has much in common with our own. They pick a small number of threads (e.g. three) to perform a small number (e.g. three) of random operations. They then use the CHESS model checker to generate all behaviours caused by interleaving these operations, and test whether each corresponds to a sequential execution of the same operations. Their approach is effective in finding bugs, but is several orders of magnitude slower than ours, taking up to hundreds of minutes to find each bug.

Vechev et al. [28] study linearizabilty using the SPIN model checker, using two different approaches. One approach uses bounded-length runs, at the end of which it is checked whether the run was linearizable, by considering all relevant re-orderings of the operations. The other approach requires linearization points to be identified by the user. The approach suffers from state-space explosion issues: for a concurrent set based on a linked list, applicability is limited to two threads and two keys, even when linearization points are provided. Liu et al. [19] study linearizability in the context of refinement checking, using the model checker PAT.

Gibbons and Korach [11] show that testing a single history for linearizability is NP-complete in general. However, they show that in the case of a history of a *register*, with a *fixed* number of processors, there is an $O(n \log n)$ algorithm; however, their approach does not seem to extend to other datatypes such as queues and stacks.

Alur et al. [1] show that verifying linearizability of a finite-state implementation with boundedly many threads is EXPSPACE. Bouajjani et al. show that with unboundedly many threads, verifying linearizability is undecidable in general [2], but decidable for certain datatypes such as stacks and queues [3].

Černý et al. [5] show that linearizability is decidable for a class of linked-list programs that invoke a fixed number of operations in parallel. Their approach shows that a program is a linearizable version of its own sequential form (rather than of a more abstract specification). In practice, their approach is limited to a pair of operations in parallel, because of the state space explosion.

Vafeiadis [27] uses abstract interpretation to verify linearizability, by considering candidate linearization points. The technique works well on some examples, but does not always succeed, and works less well on examples with more complex abstractions. Colvin et al. [6] and Derrick et al. [8] prove linearizability by verifying a simulation against a suitable specification, supported by a theorem prover. These approaches give stronger guarantees than our own, but require much more effort on the part of the verifier.

2. LINEARIZABILITY

We now give a formal definition of linearizability [14]. We use the following *events* to describe histories:

- The event t.call.$op(args)$ represents thread t calling operation op with arguments $args$;
- The event t.ret.res represents an operation of thread t returning result res.

A *history* is a sequence of such events. A history h is *well-formed* if for each thread t, the restriction of h to events of t, denoted $h \mid t$, alternates between call and ret events; we consider only well-formed histories. A history is *complete* if every call event has a matching ret event. A *legal sequential history* is a history of the sequential specification object. Two histories, h_1 and h_2 are equivalent if for each thread t, $h_1 \mid t = h_2 \mid t$. An *operation call* is a pair of corresponding call and ret events. A history h defines a partial order $<_h$ over operation calls: $op_1 <_h op_2$ if in h, op_1 returns before op_2 is called.

Definition 1
A history h of the concurrent object is *linearizable* if it can be extended to a history h' by appending zero or more ret events, and there is a legal sequential history s such that

- the restriction of h' to completed operations is equivalent to s; and
- $<_h \subseteq <_s$, i.e. if op_1 precedes op_2 in h, the same is true in s.

A concurrent object is linearizable if each of its histories is linearizable.

Our testing framework considers only complete histories. The following lemma justifies this, under the assumption of deadlock-freedom (i.e. whenever there is an out-standing method call, eventually a method call returns [13]; if the concurrent datatype is not deadlock-free, the tests will probably detect this).

Lemma 2
If a concurrent object is deadlock-free and every *complete* history is linearizable, then every history is linearizable.

For a complete history, the definition of linearizability is simpler.

Definition 3
A *complete* history h is *linearizable* if there is a legal sequential history s such that h is equivalent to s, and $<_h \subseteq <_s$.

Input: A complete history h and a sequential specification object S with an undo operation.
Returns: true iff h is linearizable.

```
1  def isLinearizable(h, S) : Boolean = {
2    if (h is empty) return true
3    else{
4      for each minimal operation op {
5        // try linearizing op first
6        let res be the result of op in h
7        run op on S
8        if (S gives result res && isLinearizable(h − op, S)) return true
9        else undo op on S
10     }
11     return false
12   }
13 }
```

Figure 1. The Wing & Gong linearizability algorithm.

3. THE WING & GONG ALGORITHM

Wing and Gong presented an algorithm for linearizability testing in [29]. We sketch the algorithm here, and refer the reader to [29] for full details. We assume that worker threads have performed operations on the concurrent datatype, and the history of call and return events recorded; the algorithm tests whether that history is linearizable.

The algorithm is given in Figure 1. It assumes a sequential specification object S that supports undo-ing of previous operations. We say that an operation op is *minimal* in a given history if no return event of another operation is before the call of op; this means that op could be linearized first. We write $h - op$ for the result of removing the call and return events of operation op from h.

The algorithm considers each operation op that could be linearized first. If the result recorded from op is consistent with S, it tries linearizing the remainder of the history with respect to the subsequent state of S. If either test fails, the algorithm backtracks, undoing op on S. Thus the algorithm considers each sequential history s consistent with the concurrent history h (i.e. such that $<_h \subseteq <_s$), and tests whether it is a valid sequential execution, i.e. whether the operations return the same results in the sequential and concurrent histories.

The implementation of the algorithm represents the history using a doubly-linked list of events, where each invocation event includes a reference to the corresponding return event. The recursion of Figure 1 is replaced by an iteration, storing each operation op in a stack to support backtracking. If backtracking is necessary, an operation is popped off the stack, reinserted in the history, and undone on S.

We extended the algorithm to give debugging information to the user in the case that the history is found not to be linearizable. If it is found that a history is not linearizable, the algorithm prints the maximum linearizable prefix, and the following event, necessarily a return event, and the alternative values that could have been returned at this point.

3.1. Wing & Gong Graph Search Algorithm

An inefficiency of the Wing & Gong Algorithm is that it fails to recognise when it encounters a configuration equivalent to one it has seen earlier, i.e. where the same operations have been linearized and the sequential object is in the same state. That is, it performs a *tree search*, as opposed to a *graph search*. We refer to it as the *Wing & Gong Tree Search Algorithm* from now on.

Returning to a previously seen configuration can happen quite often. It is common for two operations op_1 and op_2 to commute: for example, on a map, a pair of reads will commute, as will a

pair of operations on different keys. The configuration reached after linearizing op_1 then op_2 will be equivalent to after linearizing op_2 then op_1. However, if the Wing & Gong Tree Search Algorithm fails to find a linearization after the first ordering, it will repeat the search over the equivalent search space reached after the second ordering. Clearly, the search space can grow exponentially in the number of such concurrent commuting operations. Indeed, as we will discuss in Section 6, this means that the Wing & Gong Tree Search Algorithm does not prove practical: it often fails to terminate in a reasonable amount of time.

The *Wing & Gong Graph Search Algorithm* overcomes this inefficiency, by using the standard technique of remembering which configurations have been seen earlier, using a hash set: here, a configuration comprises the state of the sequential object and the set of operations linearized so far.

In order for this to work, configurations must not share the sequential specification object. Thus using a single undoable sequential object is not sufficient. Instead, we require the sequential object to be *immutable*, and for operations on it to return the resulting sequential object in addition to the natural result of the operation; for example, a dequeue operation on a queue would return the value dequeued and a new immutable object representing the new state of the queue.

The implementation uses some optimisations. Profiling of a prototype revealed that a large proportion of the run time was spent testing sequential objects for equality (within the hash set). The implementation therefore memoizes the results of previous equality tests. Sequential objects are arranged in a union-find structure corresponding to previous successful equality tests. Further, each equivalence class in the union-find structure memoizes those sequential objects that have previously been found unequal to this equivalence class.

In addition, previously performed operations on sequential objects are memoized. This avoids repeating an operation that has previously been performed on a sequential object known to be equal. The biggest advantage of this optimisation is in combination with the previous: since it avoids creating a new sequential object, it means that the results of memoized equality tests can be used.

4. JUST-IN-TIME LINEARIZATION ALGORITHMS

In this section we present two algorithms based on similar techniques: one tree search algorithm, and one graph search algorithm. We call the algorithms *Just-in-Time Linearization Algorithms*, since they linearize operations as late as possible.

In order to formalise the algorithms, we give an automaton-based definition of linearizability. We build an automaton, the *specification automaton*, to capture all linearizable histories[*]. Each state of the automaton is a *configuration*, namely a tuple $(s, calls, rets)$, where

- s is a state of the sequential specification object;
- $calls$ is a set of events of the form $t.\mathsf{call}.o(a)$, representing that thread t has called method $o(a)$, but this has not yet been linearized;
- $rets$ is a set of events of the form $t.\mathsf{ret}.r$ indicating that the last operation of thread t has been linearized, producing result r, but this operation has not yet returned.

The specification automaton has transitions between states as in Figure 2, corresponding to a thread t calling an operation $o(a)$, the operation being linearized, and the operation returning, respectively. We write $s \xrightarrow{o(a):r}_S s'$ to indicate that when operation $o(a)$ is carried out on state s of the sequential object, it produces result r and moves into state s'. The initial state of the specification automaton is $(s_0, \{\}, \{\})$, where s_0 is the initial state of the sequential specification object. Note that each reachable configuration contains at most one event of each thread t in $calls$ or $rets$.

Each path through the automaton represents a possible linearizable *execution* of a concurrent object. The call and ret events represent operation calls and returns, while the lin events represent the corresponding linearization points. The restriction to call and ret events therefore represents a linearizable history.

---

[*]The specification automaton is similar to the automaton of [6].

$$\frac{calls \text{ and } rets \text{ contains no call of } t}{(s,\ calls,\ rets) \xrightarrow{t.\mathsf{call}.o(a)} (s,\ calls \cup \{t.\mathsf{call}.o(a)\},\ rets)} \text{ call}$$

$$\frac{t.\mathsf{call}.o(a) \in calls \qquad s \xrightarrow{o(a):r}_S s'}{(s,\ calls,\ rets) \xrightarrow{t.\mathsf{lin}.r} (s',\ calls - \{t.\mathsf{call}.o(a)\},\ rets \cup \{t.\mathsf{ret}.r\})} \text{ lin}$$

$$\frac{t.\mathsf{ret}.r \in rets}{(s,\ calls,\ rets) \xrightarrow{t.\mathsf{ret}.r} (s,\ calls,\ rets - \{t.\mathsf{ret}.r\})} \text{ ret}$$

Figure 2. Transition rules for the specification automaton.

Given an observed history of the concurrent object, we want to test whether it corresponds to a path of the specification automaton. However, in order to make this more efficient, we trim some transitions of the specification automaton, in what can be thought of as a form of partial-order reduction.

Observe that in the specification automaton, we can re-order certain transitions, since lin events are not disabled by call events or ret events of other threads.

Lemma 4

- If $conf \xrightarrow{t.\mathsf{lin}.r} \xrightarrow{t'.\mathsf{call}.o'(a')} conf'$, then $conf \xrightarrow{t'.\mathsf{call}.o'(a')} \xrightarrow{t.\mathsf{lin}.r} conf'$.
- If $conf \xrightarrow{t.\mathsf{lin}.r} \xrightarrow{t'.\mathsf{ret}.r'} conf'$ and $t \neq t'$, then $conf \xrightarrow{t'.\mathsf{ret}.r'} \xrightarrow{t.\mathsf{lin}.r} conf'$.

Thus in any execution, we can push all lin events through call events or ret events of other threads, as far as possible. We call the resulting execution a *just-in-time linearization execution*, because all the operation calls are linearized as late in the execution as possible.

Definition 5
A *just-in-time linearization execution* is an execution where every lin event is followed by another lin event or by a ret event of the same thread.

Lemma 6
If a recorded history has a linearization, then it has a just-in-time linearization.

Proof: Given a linearization, the transformations in Lemma 4 can be applied repeatedly, to move later any lin event that is not followed by another lin event or by a ret event of the same thread. This produces a just-in-time linearization. □

For example, consider the execution

$$\langle t_1.\mathsf{call}.\mathsf{enqueue}(1),\ t_2.\mathsf{call}.\mathsf{enqueue}(2),\ t_1.\mathsf{ret}.(),\ t_2.\mathsf{ret}.()\rangle$$

of a queue. There are five possible linearizations of this history. However, there are only two just-in-time linearizations, corresponding to the two orders in which the operations can take effect:

$$\langle t_1.\mathsf{call}.\mathsf{enqueue}(1),\ t_2.\mathsf{call}.\mathsf{enqueue}(2),\ t_1.\mathsf{lin}.(),\ t_1.\mathsf{ret}.(),\ t_2.\mathsf{lin}.(),\ t_2.\mathsf{ret}.()\rangle,$$
$$\langle t_1.\mathsf{call}.\mathsf{enqueue}(1),\ t_2.\mathsf{call}.\mathsf{enqueue}(2),\ t_2.\mathsf{lin}.(),\ t_1.\mathsf{lin}.(),\ t_1.\mathsf{ret}.(),\ t_2.\mathsf{ret}.()\rangle.$$

The reduction in linearizations is greater when more calls are concurrent.

We trim the specification automaton so as to leave only just-in-time linearization histories: after a lin event, allow only another lin event or a ret event corresponding to the last lin event. We call this reduced automaton the *just-in-time linearization (JITL) automaton*. This reduction is what makes the subsequent algorithms faster.

6

Our Just-in-Time Linearization Algorithms then, in effect, explore the JITL automaton, trying to find an execution that corresponds to the recorded history. However, we do not explicitly build the automaton, but explore it on the fly. More concretely, each algorithm traverses the history, maintaining the current configuration.

- If the algorithm encounters a call event, it adds it to *calls* and continues.
- If it encounters a return event $t.ret.r$, and that event is in *rets* (so this operation has already been linearized), then it removes that event from *rets* and continues.
- If it encounters a return event that is not in *rets*, it considers every sequence $t_1, \ldots, t_k$ of other threads that have events in *calls*. For each such sequence:
  - The algorithm tries to perform the operations of $t_1, \ldots, t_k$ and t, in order, on the sequential object s, comparing the results returned by s with the corresponding results in h;
  - If all match, it removes the events of $t_1, \ldots, t_k$ and t from *calls*, and adds corresponding events for $t_1, \ldots, t_k$ to *rets* (i.e. linearizing the operations of $t_1, \ldots, t_k, t$, and returning the operation of t); it then continues from the next event in the history;
  - If any of the returned events do not match, or the continuation from the next event fails, then the algorithm backtracks, and considers the next sequence of operations to linearize.

  If every such sequence fails, the history isn't linearizable from the current point, so the algorithm backtracks.

There are two variants of the algorithm, corresponding to a tree search and a graph search.

The tree search algorithm uses an undoable sequential specification object. The backtracking is supported using a stack that records which operations to undo when backtracking, and which other sequences of operations to try linearizing.

The graph search algorithm uses an immutable sequential object, as with the Wing & Gong Graph Search Algorithm. The backtracking is supported using a stack that record which configurations to backtrack to, and which other sequences of operations to try linearizing. A hash table is used to record configurations previously reached, to avoid repeating parts of the search. In addition, the algorithm employs the two optimizations from the Wing & Gong Graph Search Algorithm, memoizing the results of equality tests between sequential objects, and the results of operations on the sequential objects.

We have also experimented with a breadth-first search, graph search version of the same technique. However, this proved less efficient in practice.

We now perform a partial complexity analysis of the algorithms. Suppose we have a history h of length N, performed by p threads. There are up to p transitions from each configuration, so in the worst case the algorithm explores $O(p^N)$ configurations. However, as we shall see in Section 6, the algorithms perform much better than this in practice. Further, we can obtain a better bound in some cases.

Consider the set of all configurations the graph search algorithm can reach after n events of the history. In each such configuration, some subset L of the pending operations will have been linearized; there are at most 2^p such subsets. Suppose we can find a bound $B_{p,n}$ on the number of states that the sequential specification object can be in for each such choice of L. In this case, the algorithm visits at most $2^p.B_{p,n}$ configurations after n events, giving a total for a history of length N of $\sum_{n=0}^{N} 2^p.B_{p,n}$.

For example, if the specification object is a register, then the value of the register will be the last value written by one of the p threads, or possibly (if no call has returned) the initial value of the register, giving a bound of $B_{p,n} = p + 1$. Hence, for a history of length N, the algorithm visits at most $(N + 1).2^p.(p + 1)$ configurations; thus the algorithm is linear in the length of the history; it is exponential in the number of threads, but, in practice, the number of threads is typically small. Likewise, for a map with K keys, a similar argument gives a bound of $(N + 1).2^p.K^{p+1}$, again linear in the length of the history (albeit with a potentially large constant factor).

However, for other datatypes we obtain much less good bounds. For example, with a queue, if all p threads enqueue distinct values concurrently, they can be linearized in $p!$ different orders; if this

is repeated, after m enqueue operations (so $n = 2m$ events) there can be (at least) $p!^{m/p}$ different states for the queue. Thus the algorithm is exponential in the worst case.

5. QUEUE-ORIENTED LINEARIZABILITY ALGORITHM

In this section we concentrate on concurrent queues: we consider the problem of testing whether a complete history h of a concurrent queue is linearizable. Our basic approach is to pair matching enqueues and dequeues, partially inspired by the work of Henzinger et al. [12].

Our approach concentrates on *operations* rather than their individual call and return events. We write $enq(x)$ for an enqueue operation of x, $deq(x)$ for a dequeue operation that returns x, and $deq(\text{EMPTY})$ for an unsuccessful dequeue operation, i.e. that is called on an empty queue.

Recall that each history h defines a partial order $<_h$ over its operations: $op_1 <_h op_2$ if op_1 returns before op_2 is called. Recall also that an operation op is *minimal* in h if there is no operation op' such that $op' <_h op$; note that the first operation to be linearized must be minimal. We say that a dequeue operation d is *deq-minimal* if there is no other dequeue operation d' such that $d' <_h d$; note that the first dequeue to be linearized must be deq-minimal. We say that a dequeue operation d *matches* an enqueue operation e if the value dequeued by d is the same as the value enqueued by e. Our basic approach will be to pair a deq-minimal dequeue d with a matching minimal enqueue e; we will show that, in most cases, h is linearizable if and only if the history obtained from removing d and e, and performing an additional small transformation is linearizable.

We start with some easier cases.

Lemma 7
If complete history h contains only enqueue operations, then it is linearizable.

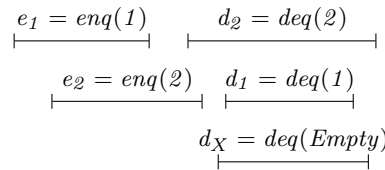A minimal unsuccessful dequeue can be linearized first.

Lemma 8
Suppose h has a minimal unsuccessful dequeue operation d (i.e. d returns EMPTY). Then h is linearizable iff $h' = h - d$ is linearizable.

Proof: ($\Leftarrow$) Suppose h' is linearizable to s'. Then it is clear that h is linearizable to $s = \langle d \rangle ^\frown s'$ (cf. Definition 3): this is clearly legal, since d can happen initially and does not change the state; and the ordering of operations in h and s are consistent because d is minimal.

($\Rightarrow$) Suppose h is linearizable to s. Then h' is linearizable to $s' = s - d$: it is clear that s' is legal since d doesn't change the state; and if $op_1 <_{h'} op_2$ then $op_1 <_h op_2$ so $op_1 <_s op_2$ and so $op_1 <_{s'} op_2$. $\qquad\square$

When neither of the above lemmas apply, we seek to pair a minimal enqueue e with a matching deq-minimal dequeue d; we call such a pair of operations a *minimal matching pair*. We will remove such a pair (e, d) from the history, and then test whether the remainder of the history is linearizable. For example, consider the history h illustrated by the timeline below.

We can start by removing the minimal matching pair (e_1, d_1); then remove the minimal matching pair (e_2, d_2); and finally remove the unsuccessful dequeue d_X (which is now minimal). This corresponds to the linearization $s = \langle e_1, e_2, d_1, d_2, d_X \rangle$; note that as $e_2 <_h d_1$ we need $e_2 <_s d_1$.

However, this is not enough, as illustrated below.

8
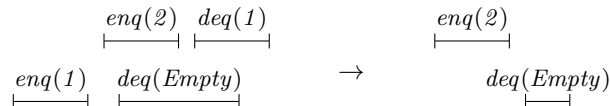
The history on the left is not linearizable; but simply removing the $enq(1)$ and $deq(1)$ operations gives a linearizable history. Instead, in addition to removing the $enq(1)$ and $deq(1)$, we delay the call of the $deq(Empty)$ operation, as illustrated on the right, which gives an unlinearizable history.

In general, when we remove a matching pair (e, d), for any dequeue d' concurrent to d, and for any operation e' (necessarily an enqueue) such that $e' <_h d$, we move the call of d' later so that $e' < d'$; if there are multiple such d', we maintain the order of their call events. The transformation is illustrated below.
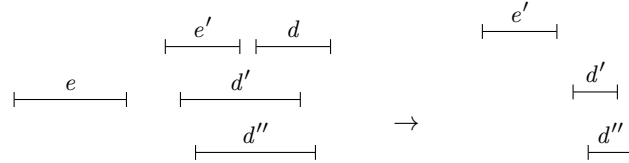
We will linearize d between e' and d', so $e' <_s d <_s d'$. We write $h -^D d$ for the above transformation. Note that this transformation preserves $<$-relationships.

The following pair of lemmas show that considering all such minimal matching pairs, and transforming the history as above, is sufficient to decide whether or not the history is linearizable.

Lemma 9

Let h be a complete linearizable history. Suppose that

1. h has no minimal unsuccessful dequeue;
2. h has at least one successful dequeue.

Then there is a minimal matching pair (e, d) such that $h - e -^D d$ is linearizable.

Proof: Suppose h is linearizable to s. Let e and d be the first enqueue and dequeue in s. Necessarily, e and d form a minimal matching pair in h. Let $h' = h - e -^D d$ and $s' = s - e - d$. We show that h' is linearizable to s'.

- s' is legal since s was, and no unsuccessful dequeue precedes d in s.
- Suppose $op_1 <_{h'} op_2$. If $op_1 <_h op_2$ then $op_1 <_s op_2$ so $op_1 <_{s'} op_2$. Otherwise, the relationship $op_1 <_{h'} op_2$ must have been introduced by the "$-^D d$" transformation, so $op_1 <_h d$ and op_2 is a dequeue; but then $op_1 <_s d <_s op_2$ and so $op_1 <_{s'} op_2$.

□

Lemma 10

Suppose history h is such that

1. h has no minimal unsuccessful dequeue;
2. There is a minimal matching pair (e, d) such that $h' = h - e -^D d$ is linearizable.
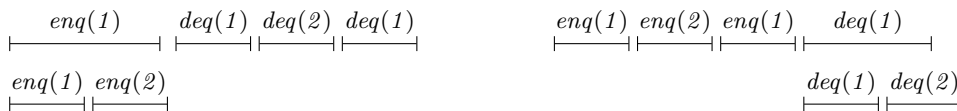
Then h is linearizable.

Proof: Suppose h' is linearizable to s'. Let s_1 be the minimal prefix of s' that contains each operation e' (necessarily an enqueue) such that $e' <_h d$, and let s_2 be the remainder of s' so that $s' = s_1 \frown s_2$. Hence $s_1 = \langle \rangle$ or last $s_1 <_h d$; note that in the latter case, last $s_1 <_{h'} d'$ for every dequeue d', by the construction of h', so last $s_1 <_{s'} d'$; i.e. s_1 contains no dequeue operation.

Let $s = \langle e \rangle \frown s_1 \frown \langle d \rangle \frown s_2$. We show that h is linearizable to s.

- Clearly s is legal, since e and d correspond, s_1 contains no dequeue, and s' is legal.
- For any operations other than e and d, if $op <_h op'$ then $op <_{h'} op'$, so $op <_{s'} op'$, so $op <_s op'$.
- If $e <_h op$ then clearly $e <_s op$.
- The case $op <_h e$ can't happen since e is minimal.
- If $op <_h d$ then op is in s_1 by definition, so $op <_s d$.
- If $d <_h op$ then either $s_1 = \langle \rangle$ or last $s_1 <_h d <_h op$; in the latter case, last $s_1 <_{h'} op$ so last $s_1 <_{s'} op$; hence op is in s_2 so $d <_s op$.

□

The above lemmas do not immediately provide us with an efficient algorithm. In particular, Lemma 9 seems to require us to consider *all* minimal matching pairs in order to show that a history is not linearizable. In Lemma 12, below, we show that in certain, very common, circumstances, it is enough to consider just a single such pair (e, d); in these circumstances, h is linearizable if and only if $h - e -^D d$ is linearizable.

In Lemma 12, we will pick e so that no enqueue of the same value returns before e, and we will pick d so that no dequeue of the same value returns before d. We say that a minimal matching pair is *earliest-ending* when this holds. The examples below justify this.

$$
\begin{array}{ll}
enq(1) & deq(1)\ deq(2)\ deq(1) \\
enq(1)\ enq(2) &
\end{array}
\qquad
\begin{array}{ll}
enq(1)\ enq(2)\ enq(1) & deq(1) \\
& deq(1)\ deq(2)
\end{array}
$$

In the left-hand example, a linearization is obtained only if the first $deq(1)$ is paired with the lower, earlier-ending, $enq(1)$. Likewise, in the right-hand example, the first $enq(1)$ must be paired with the lower, earlier-ending, $deq(1)$.

Further, we will place a restriction on unsuccessful dequeues that are concurrent with the earliest-ending deq-minimal dequeue d we consider. The example below shows why such a restriction is necessary.

$$
\begin{array}{ll}
e = enq(0) & deq(0) \\
& enq(0)\quad enq(1)\quad d = deq(0) \\
& deq(Empty)
\end{array}
$$
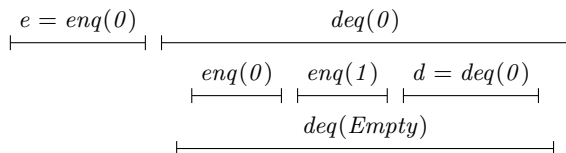
Here, the history h is linearizable, by pairing e with the top $deq(0)$. However, we cannot create a linearization by pairing e with the lower, earlier-ending $deq(0)$, d: the transformed history $h - e -^D d$, which has the start of the $deq(Empty)$ shifted to after the $enq(1)$, is not linearizable.

The following definition captures the class of unsuccessful dequeues that we need to disallow in Lemma 12.

Definition 11
We say that a history h has a *conflicting unsuccessful dequeue d_X for d* if d is an earliest-ending deq-minimal dequeue, and there exist an enqueue e' and a successful dequeue d' such that $e' <_h d$, $e' \not\prec_h d'$, and $e' \not\prec_h d_X$.

The timeline below illustrates such a history.

$$
\begin{array}{ll}
e' & d \\
d' & \\
d_X = deq(Empty) &
\end{array}
$$
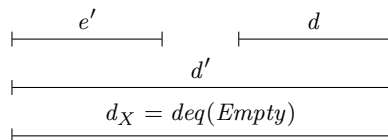
Lemma 12
Let h be a complete history. Suppose that

1. h has no minimal unsuccessful dequeue;
2. there is an earliest-ending minimal matching pair (e, d);
3. there is no conflicting unsuccessful dequeue for d.

Then h is linearizable iff $h' = h - e -^D d$ is linearizable.

Proof: ($\Leftarrow$)   This follows immediately from Lemma 10.

($\Rightarrow$)   Suppose h is linearizable to s, and suppose e and d use value x. We transform s, to produce a new linearization of h, and such that e and d are the first enqueue and dequeue of x, respectively.

We illustrate the transformations by considering the example history

$$s \quad = \quad \langle enq(0),\ enq(1),\ e = enq(0),\ deq(0),\ deq(1),\ enq(2),\ d = deq(0)\rangle.$$

The transformations are as follows.

1. If there is an enqueue e' and a dequeue d' such that $e' <_h d$ but $d' <_s e'$, then shift d' to after e'; if there are multiple such d' then maintain their order. Let s_1 be the resulting history. Applied to the example, assuming $enq(2) <_h d$, this produces

$$s_1 \quad = \quad \langle enq(0),\ enq(1),\ e = enq(0),\ enq(2),\ deq(0),\ deq(1),\ d = deq(0)\rangle.$$

   We show that each such dequeue that is shifted is successful. Suppose for a contradiction that an unsuccessful dequeue d_X is shifted. So, for some e', $d_X <_s e' <_h d$, and so $e' \not<_h d_X$. By condition 1, no unsuccessful dequeue is minimal, so the first operation in s is an enqueue e_0. For legality, there must be a matching successful dequeue d_0 before d_X in s, so $e_0 <_s d_0 <_s d_X <_s e'$. Hence $e' \not<_h d_0$. This contradicts condition 3 (with $d' = d_0$ in Definition 11).
   We now show that s_1 is a linearization of h.
   - Note that s_1 is still legal, since we have simply delayed successful dequeues, without reordering.
   - If $d' <_h op$ for some shifted d', then $e' <_h op$ (since $e' <_h d$, the call of d precedes the return of d' in h, and $d' <_h op$), so we still have $d' <_{s_1} op$ after this transformation.
   - Clearly all other required orderings are preserved.

2. Let e_0 be the first enqueue of x in s_1. If e_0 is not e, then swap e and e_0. Let s_2 be the resulting history. Applied to the example, this produces

$$s_2 \quad = \quad \langle e = enq(0),\ enq(1),\ e_0 = enq(0),\ enq(2),\ deq(0),\ deq(1),\ d = deq(0)\rangle.$$

   We show that s_2 is a linearization of h.
   - Clearly s_2 is still legal.
   - If $e_0 <_h op$ then $e <_h op$ since e returns before e_0 in h, by condition 2; so $e <_{s_1} op$ and so $e_0 <_{s_2} op$ after this swap.
   - The case $op <_h e$ can't happen since e is minimal.
   - Clearly all other required orderings are preserved.

   We will need the following observation below: suppose $e_0 <_h d$; then $e <_h d$ since e returns before e_0; then for every dequeue d', $e <_{s_1} d'$ by transformation 1; hence $e_0 <_{s_2} d'$. That is:

$$\text{if} \quad e_0 <_h d \quad \text{then} \quad \text{for every dequeue } d',\ e_0 <_{s_2} d'.$$

3. Let d_0 be the first dequeue of x in s_2. If d_0 is not d, then swap d and d_0. Let s_3 be the resulting history. Applied to the example, this produces

$$s_3 \quad = \quad \langle e = enq(0),\ enq(1),\ enq(0),\ enq(2),\ d = deq(0),\ deq(1),\ d_0 = deq(0)\rangle.$$

   We show that s_3 is a linearization of h.
   - Clearly s_3 is still legal.
   - Suppose $op <_h d$; then necessarily op is an enqueue. We consider three cases.
     - Case $op = e$. Then $e <_{s_2} d_0$ since e is the first enqueue of x in s_2, and s_2 is legal; hence $e <_{s_3} d$.
     - Case $op = e_0$. Then by the observation at the end of transformation 2, $e_0 <_{s_2} d_0$; hence $e_0 <_{s_3} d$.
     - Case op is some enqueue e', other than e and e_0. Then $e' <_{s_1} d_0$ by transformation 1; hence $e' <_{s_2} d_0$; and hence $e' <_{s_3} d$.
   - If $d_0 <_h op$ then $d <_h op$ since d returns before d_0, by condition 2; so $d <_{s_2} op$, and $d_0 <_{s_3} op$.

11

Input: A complete history h.
Returns: true iff h is linearizable.

```
1   def isLinearizable(h) : Boolean = {
2     if (h contains only enqueue operations)
3       return true // cf. Lemma 7
4     else if(h has a minimal unsuccessful dequeue operation d)
5       return isLinearizable(h − d) // cf. Lemma 8
6     else if(h has an earliest-ending minimal matching pair (e, d)
7               such that there is no conflicting unsuccessful dequeue)
8       return isLinearizable(h − e −ᴰ d) // cf. Lemma 12
9     else{
10      for each minimal matching pair (e, d){
11        if (isLinearizable(h − e −ᴰ d)) return true // cf. Lemma 10
12      }
13      return false // cf. Lemma 9
14    }
15  }
```

Figure 3. The linearizability algorithm for a history of a queue.

- Clearly all other required orderings are preserved.

Let $s' = s_3 - \{e, d\}$. We show that s' is a linearization of h'.

- s' is legal since s_3 is legal, and e and d correspond since they are the first enqueue and dequeue of x in s_3.
- Suppose $op_1 <_{h'} op_2$, so $op_1, op_2 \neq e, d$.
  - If $op_1 <_h op_2$ then $op_1 <_{s_3} op_2$ (by the above), so $op_1 <_{s'} op_2$.
  - Otherwise, the relationship $op_1 <_{h'} op_2$ must have been introduced by the "$-^D d$" transformation, so op_1 is an enqueue, $op_1 <_h d$ and op_2 is a dequeue. We consider two cases.
    * If $op_1 = e_0$ then by the observation at the end of transformation 2, $e_0 <_{s_2} op_2$; hence $e_0 <_{s_3} op_2$ since $op_2 \neq d$; and hence $e_0 <_{s'} op_2$.
    * Otherwise, by transformation 1, $op_1 <_{s_1} op_2$; hence $op_1 <_{s_2} op_2$, and $op_1 <_{s_3} op_2$ since $op_2 \neq d$; and hence $op_1 <_{s'} op_2$.

□

The above series of lemmas justifies the algorithm in Figure 3. In most cases, we need consider only a single earliest-ending minimal matching pair (line 8). In other cases, we potentially need to consider all-such (lines 10–13); however, in practice, it is very rare to have to consider more than one.

We now describe our implementation. We implement the history as a doubly-linked list. Each node of the list also contains a reference to the next node corresponding to the same thread. Further, each node contains a sequence number in the history (which allows the $<_h$ relation to be calculated efficiently). In addition, to support efficient searching, we store, for each thread, references to the nodes corresponding to the calls of the thread's first operation and the thread's first dequeue operation.

We replace the recursion in Figure 3 by an iteration, using a stack to support backtracking. If there is more than one minimal matching pair (e, d) at line 10, we push onto the stack each pair other than the first, together with a clone of the history linked list. If the call at line 11 fails, we backtrack by popping the next minimal matching pair and the clone of the history from the stack, and continuing.

Informal experiments suggest that backtracking is rarely necessary: it happens about once in every 9,000 runs (with 800 operations per run); when backtracking is necessary, it doubles the

number of iterations, at worst. Thus the number of iterations of the algorithm is effectively linear in the number of operations.

Since we need to test whether each minimal enqueue or deq-minimal dequeue is earliest-ending, each iteration of the algorithm could take time linear in the length of the history. It is possible to reduce this to amortized constant time, by, for each enqueue e, caching a list of other enqueues of the same value that return during the duration of e, and similarly for dequeues. However, informal experiments suggest that this makes the implementation slower in practice.

6. EXPERIMENTS

We now describe experiments to compare the algorithms described in the previous sections.

In addition, we have considered various pairs of algorithms composed together using *competition parallel*: the two algorithms ran in separate threads, but the first to complete interrupted the other. As our experiments will show, a tree search and a graph search algorithm in competition parallel can complement one another well. We use names such as "WG-JIT-Competition" for the competition parallel composition of the Wing & Gong Tree Search and the JIT Graph Search Algorithms, i.e. with the tree search algorithm denoted first.

The experiments were carried out on an eight-core machine (two Intel Xeon E5620 CPUs, without hyperthreading; 32K L1d cache, 256K L2 cache and 12288K L3 cache) with 12GB of RAM (six 2GB modules), and used the Java HotSpot 64-Bit Server VM (build 24.45-b08, mixed mode) with the Java SE Runtime Environment (build 1.7.0_45-b18), and Scala compiler version 2.10.1.

In each *run* of each experiment, several workers, normally four, each performed a number of operations upon a shared datatype, and the resulting history tested for linearizability. An *observation* constitutes a number of runs, for which the total time was recorded. Each observation was intended to be close to a typical use case of the linearizability tester. In order to ensure independence of observations, each was performed as a separate invocation of the Java Virtual Machine.

Our statistical analysis follows the methodology of Georges et al. [10]. For each choice of experimental parameters, multiple observations were made, and the mean and 95% confidence intervals calculated. This was repeated until either the size of the confidence interval was within 2% of the mean, or 50 observations had been performed, whichever came first. The graphs that follow display the means and 95% confidence intervals (except the latter are omitted when small).

In order to deal with cases where the algorithms performed badly, we limited executions to 1.3 million iterations for the Wing & Gong Graph Search Algorithm, three million iterations for the Just-in-Time Linearization Graph Search Algorithm, and ten billion iterations for the other algorithms[†].

It proved infeasible to profile the two tree search algorithms in a meaningful way. In most observations, they were faster than the graph search algorithms. But other observations took much longer, failing to terminate within a reasonable time. In an informal experiment, we timed 50 *single* runs of each tree search algorithm on a map, with 2^8 operations per worker, but limiting the algorithm to consider at most ten billion configurations. The Wing & Gong Algorithm terminated within 30ms on 43 of the 50 runs (normally within 10ms); however, on five of the runs it failed to terminate within the configuration limit (taking about 25 minutes in each case). The results for the Just-in-Time Linearization Algorithm were similar, except only one of the 50 failed to terminate within the limit (taking about 14 minutes).

We can account for this erratic behaviour as follows. Informally, each tree search algorithm often gets lucky, searching a very small proportion of the search space before finding a correct linearization. This is partly because it considers operations in the order they are called, and it is very common for operations to be linearized in the same order. The graph search algorithms can get lucky in a similar way; but when both get lucky, the tree search algorithms are usually faster:

[†]The two graph search algorithms have heavier use of memory than the other algorithms, because of the use of a hash table to store configurations. The Wing & Gong Algorithm is particularly heavy on memory for long runs, because it records the set of operations linearized in each stored configurations. The lower limit on iterations for these algorithms prevented the experiments from failing if memory was exceeded.
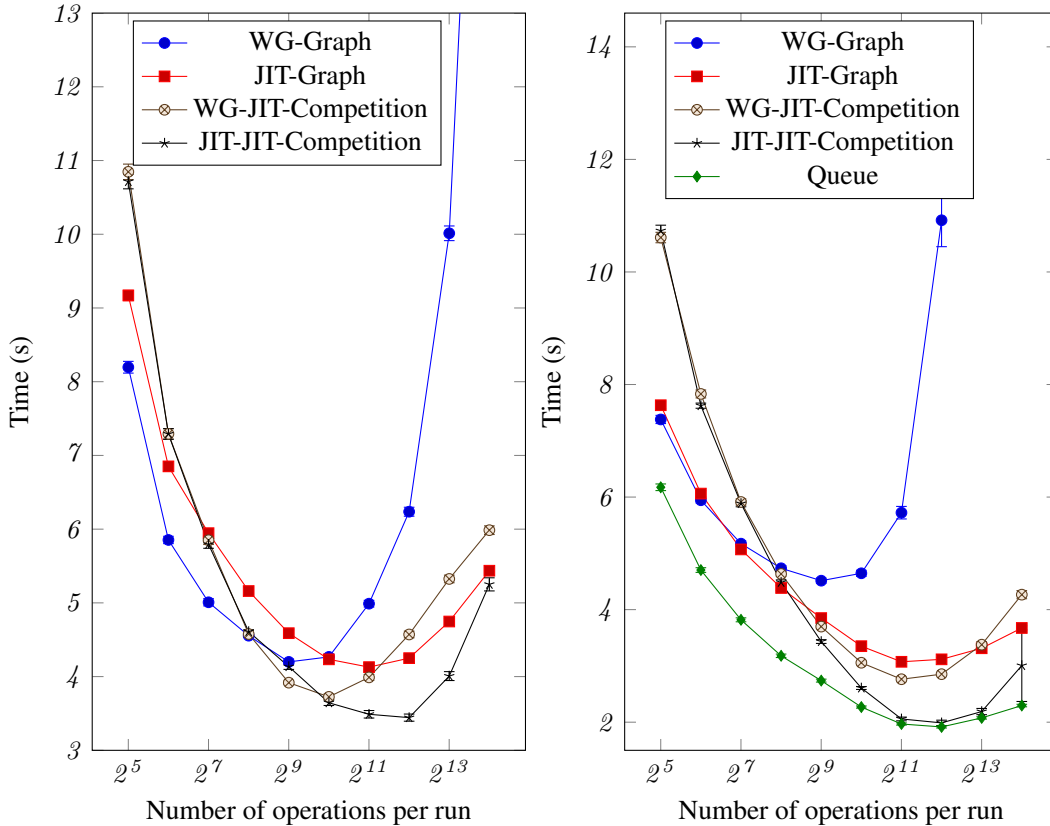
Figure 4. Comparison of linearizability testers on a map (left) and a queue (right).

the graph search algorithms have extra overheads in storing configurations, and because they use immutable specification datatypes (which often require partial cloning of the datatype). However, sometimes the search takes a wrong turn early, and, because it does not record which states it has visited earlier, the tree search has to explore a very large space before backtracking to consider the correct part of the search space; this makes it much slower than the corresponding graph search algorithm.

Figure 4 shows results for several testers. The left-hand graph gives results for a map (keys were chosen from a domain of size 20, and corresponding values from a domain of size 4; operations were split evenly between reads, updates, deletes or get-or-else-update operations); the right-hand graph gives results for a queue (30% of operations were an enqueue of a value from a domain of size 20; the remaining operations were a dequeue). The x-axis shows the number $numOps$ of operations performed by each worker in each run. In each case, the number $numRuns$ of runs per observation was chosen so that $numOps \times numRuns = 2^{18}$, i.e., each observation represents the same number of operations (given four workers, 2^{20} in total). It seem reasonable to assume that the probability of finding a bug (if one exists) is roughly proportional to the number of operations performed: then the choice of $numOps$ that minimises the run-time would seem to be optimal.

For clarity, we have omitted plots for the WG-WG and JIT-WG Competition Testers from Figure 4. These have similar behaviours to the WG-JIT and JIT-JIT Testers, respectively, up to about 2^{11} operations per run, but then tail off markedly, like the WG Graph Search Algorithm (and, in the case of tests on a queue, fail with about 2^{13} operations). We have also omitted competition testers involving the queue-oriented algorithm, since these perform no better than the queue-oriented algorithm itself.
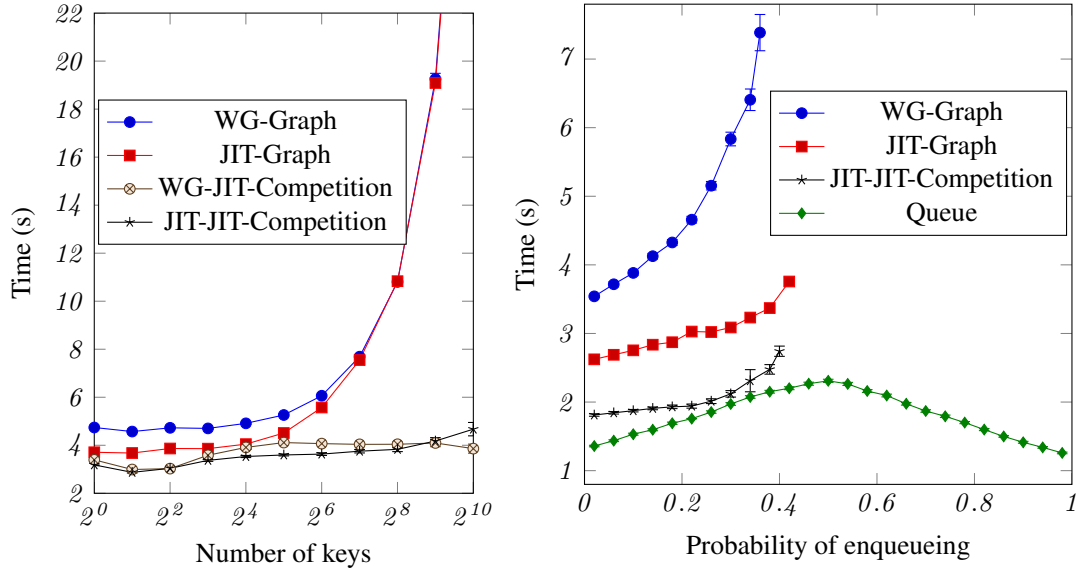
14

Figure 5. Comparison of linearizability testers on a map, with different sizes of key space (left), and for a queue with different probabilities of enqueueing.

The graphs in Figure 4 display a similar "U"-shape. The initial speed-up can be explained by the fixed overheads of each run being reduced as the number of runs decreases. The subsequent slow-down is explained by an increase in the number of configurations, and a reduction in the efficiency of datatypes (e.g. the hash table) as they become larger. Note that the minima correspond to runs that seem long enough to find bugs: it is hard to imagine cases where a bug would require a longer run to manifest itself.

For the experiments on a queue (Figure 4, right), the Queue-Oriented Algorithm outperformed each of the other algorithms. This is our preferred algorithm for linearizability testing on a queue. (We give more evidence in its favour below.)

For the experiments on a map, and among the generic algorithms for experiments on a queue, at the minima in Figure 4, the JIT-JIT-Competition Tester proved slightly faster than the others. For this competition tester, around the minima, the JIT Tree Search "won" on about 98% of runs on a queue, and about 50% of runs on a map. In most cases, the Competition Parallel benefits from the fast behaviour of the Tree Search Algorithm; when the Tree Search performs badly, the Competition Parallel falls back on the more reliable behaviour of the Graph Search Algorithm. In particular, on examples that are not linearizable, the Graph Search Algorithm normally wins, because the entire state space has to be searched. Thus these algorithms seem to combine well in competition.

Comparing just the two generic graph search algorithms, the JIT Algorithm outperforms the Wing & Gong Algorithm on a queue; the two algorithms have similar behaviour on a map, with the JIT Algorithm performing better on longer runs. We believe that this is because of the just-in-time linearization reducing the number of configurations that need to be considered.

A practical down-side of the competition testers is that the user has to provide *two* sequential specification objects, an immutable one and an undoable one. Thus the user has to spend more time programming in return for less time running the tester. Our own preference (other than with a queue) is often to stick with the JIT Graph Search Algorithm, for this reason.

Figure 5 (left) shows results for experiments for testing linearizability on a map, with different sizes of key space. (Each measurement consisted of 2^7 runs, in each of which four workers each performed 2^{11} operations.) The slow down for each of the graph search algorithms can be explained by the fact that the testers often need to clone the map; the cost of this is proportional to the number of keys in the map. It is interesting that the two competition parallel testers do not have such a

slow down: the tree search algorithms do not need to clone the map, and the overall performance is dominated by their behaviour.

Figure 5 (right) studies the behaviour of the various algorithms on a queue, as a function of the probability with which each worker performed an enqueue operation. (In each run, each worker performed 2^{11} operations; each operation comprised 2^{7} runs.)

The generic algorithms each reached their iteration limit on some run with an enqueueing probability of about 0.4. If two values are enqueued concurrently, the algorithm has to consider both possible linearization orders; this is only resolved when one of the items is dequeued. Thus the number of configurations grows exponentially with the length of the queue. (With other datatypes, the linearization order of concurrent operations is normally resolved sooner.) A larger probability of enqueueing makes concurrent enqueues more common, making it more likely for the queue to become long.

However, the Queue-Oriented Algorithm performed much better. We can explain the shape of the graph as follows: with a low probability of enqueueing, most of the operations are unsuccessful dequeues, which the algorithm deals with very efficiently; with a high probability of enqueueing, the algorithm soon reaches its termination condition of the history containing just enqueue operations; however, with intermediate probabilities, it has to do more work of finding matching enqueues and dequeues. (Pragmatically, it seems more likely that bugs, if they exist, will be found with an intermediate probability.)

It is possible to run several linearizability testers in competition parallel with each other, thereby increasing the speed of testing. The graph to the right shows the results of an experiment running different numbers n of Queue-Oriented linearizability testers: in each observation, each tester performed $840/n$ runs; thus each observation represents the same number of runs. (In each run, four worker threads each performed 2^{11} operations, each operation being an enqueue with probability 0.5). The running time initially falls steeply, almost as $\frac{1}{n}$, until reaching a point where the number of runnable threads often exceeds the number of cores. At the minimum, with five testers in parallel, there is a speed-up of a factor of about three over a single tester; since each tester is running four threads for a significant proportion of the time, this is about as much of a speed-up as one might hope for.

Number of linearizability testers
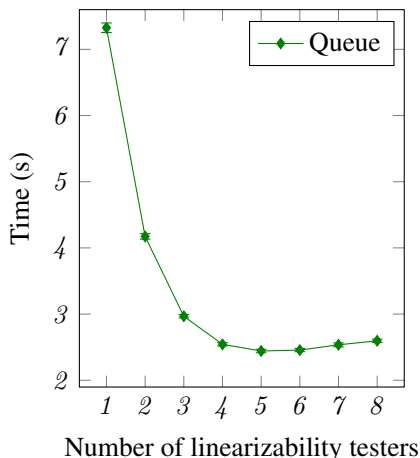
7. THE LINEARIZABILITY TESTING FRAMEWORK

In this section we outline out testing framework. The framework uses Scala; it can also be used directly with Java datatypes, or with datatypes in other languages (such as C or C++) via the Java Native Interface. It would be straightforward to adapt the techniques to other languages. The Scala API contains lots of immutable datatypes, which are useful as specification datatypes with generic graph search algorithms.

To illustrate how the framework can be used, Figure 6 gives a stripped-down testing program, for the JIT Graph Search Algorithm applied to a queue (the full version can be used with multiple concurrent queue implementations, and replaces 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 [13, 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 with equality tests: here an immutable queue from the Scala API [24]. For each operation op : A on the concurrent datatype, we need a corresponding function

16

```
1   object QueueTest{
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 200)
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    def main(args: Array[String]) =
13      for(i <− 0 until 1000){
14        val concQueue = new LockFreeQueue[Int] // The shared concurrent queue
15        val seqQueue = Queue[Int]() // The sequential specification queue
16        val tester = LinearizabilityTester.JITGraph[S, C](seqQueue, concQueue, 4, worker, 200)
17        assert(tester() > 0)
18      }
19  }
```

Figure 6. An example testing program.

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. Here, the worker performs 200 operations; each is (with probability 0.3) an enqueue of a random value, or (with probability 0.7) a dequeue. More precisely, the worker takes a log object theLog as a parameter; 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 for indexing the map of previously performed operations, described earlier; semantically different operations should have different strings; and
- the corresponding operation on the sequential datatype.

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

The linearizability tester is constructed at line 16 and run at line 17; 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 17, 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.

Other linearizability algorithms can be used in a similar way, with slightly different types for the worker; see [20].

7.1. Logging

We now discuss how the logging of call and return events is performed. One subtlety is that we do not want the logging to have an effect upon the behaviour of the datatype we are testing. This is not completely straightforward. Some datatype implementations suffer from *memory consistency bugs*, where an update performed by one thread has not propagated to another thread, either because the former's cache has not been flushed, or because the latter's cache has not been refreshed; in other words, the implementation has insufficient synchronization.

Our original logging technique used a *shared* log, where each thread performed a synchronization event in order to claim a slot in the log to use. Unfortunately, this synchronization meant that the thread's cache was both flushed and refreshed, which meant that many such memory consistency bugs would not be found: the synchronization in the logging masked the lack of synchronization in the datatype.

Instead, our approach is for each thread to have its own private log, into which it writes call and return events, each paired with a timestamp[‡]. Once all threads have finished, the thread logs are merged into timestamp order. The correctness of this approach requires the underlying timestamping mechanism to be reasonably accurate: if updates made by an operation of thread t_1 can be observed by an operation of thread t_2, then the timestamp for t_1's call event must be strictly less than the timestamp for t_2's return event. The approach seems to work correctly on our machine, but may work less well on other machines.

Our timestamping approach allows us to find memory synchronization bugs (see Section 8). However, the degree to which caches are flushed and refreshed depends upon both the implementation of the compiler and the details of the hardware, so may vary from one machine to another. Indeed, rather small changes in our logging code seemed to have an effect upon the memory behaviour and hence how frequently such bugs were found.

8. FINDING ERRORS

We now discuss some pragmatics of designing testing programs so as to find errors. We present in detail one error found using the framework, and briefly survey others, to give an indication of the framework's capabilities. We also give statistics for the time taken to find errors (the experimental set-up and statistical analysis are as in Section 6).

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 stored 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.

[‡]Obtained using the java.lang.System.nanoTime function.

```
369  1 invokes update(0, 0)
370  0 invokes delete 0
371  0 returns ()
372  0 invokes update(0, 2)
373  0 returns ()
374  0 invokes delete 0
375  0 returns ()
376  1 returns ()
378  1 invokes getOrElse(0, X)
379  -- Previous event not linearized
380  1 returns 2
381  -- Previous event not linearized
382  -- Allowed return values: 0, X
```
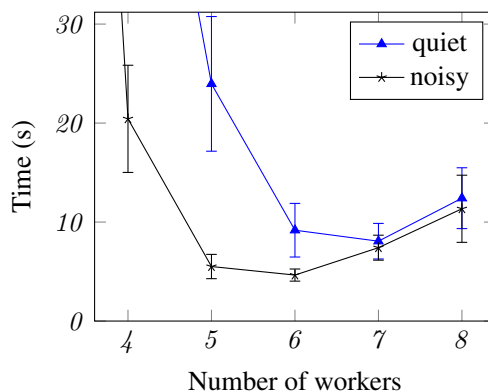

Figure 7. Part of a debugging trace (left). Time to find the error for different numbers of workers (right).

- 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 observations apply to a stack.

The most subtle bug we have found using the framework concerned our attempt to extend the hash map of Laarman et al. [17]. That map supported only a get-or-else-update operation, and didn't allow resizing. In [21], we extended the hash table to support other operations and resizing. Each entry in the table has a status field, either *Empty* indicating an empty slot, *Updating* indicating that a thread is updating the entry (which acts as a simple lock), *Stored* indicating that a value was stored, or (for our extensions) *Deleted* indicating that a value had been deleted. A thread performing an update spins while the status is *Updating*, then performs a compare-and-set (CAS) operation to set the status to *Updating*, does the update, and sets the status to *Stored*.

In the incorrect extension, a delete operation sets the status to *Deleted*; this is a bug: a CAS operation should be used, although this is not obvious. Figure 7 (left) gives part of a debugging output, using a map with two keys. The error relates to key 0. We have filtered out all references to the other key, and given a suitable suffix; the left-hand column gives line numbers from the original file. Thread 1 invoked the getOrElse(0, X) operation (line 378), which should return the value of key 0, or the value X if 0 is not in the map; this gives result 2 (line 380), whereas only the values 0 or X would allow linearization. With the help of the history, we can explain the error as follows: the update(0, 0) operation (line 369), changed the status to *Updating*, performed the update, and was then descheduled. The first delete(0) operation (line 370) then changed the status to *Deleted*. The update(0, 2) operation (line 372) changed the stored value to 2 and the status to *Stored*. Then the second delete(0) operation (line 374) changed the status to *Deleted*. Finally, the initial update(0, 0) was re-scheduled, and changed the status back to *Stored*, erroneously indicating that the value 2 is again stored against the key. This error seems to need five operations to interact in this way, so is very hard to discover by hand.

Figure 7 (right) gives statistics about the time taken to find this error with different numbers of worker threads (using the JIT-JIT Competition Tester; each observation considered runs of 8400 operations split evenly between the workers, repeated until an error was found). It turns out that the time taken to discover the error is significantly affected by the state of the machine on which the tests are run. The errors are found more quickly when run on a "noisy" machine, with other applications

running at the same time (principally a linearizability tester testing another datatype, plus standard desk-top applications, using on average about two of the eight cores). Under these noisy conditions, worker threads have to compete to be scheduled, and will sometimes be descheduled for fairly significant lengths of time; such erratic scheduling makes this bug more likely to occur.

The bug is found most quickly with five or six workers on a noisy machine, and with between six and eight workers on a quiet machine (the confidence intervals overlap, so it is not possible to be confident about the precise minima). With fewer workers, the error is less frequent because of the lack of concurrency, and the lack of competition for the CPUs which makes scheduling less erratic. With more workers, the linearization testing algorithms become slightly slower, so each history takes longer to analyse.

In [21], we also extended the map of Laarman et al. to support resizing. During resizing, items in the old table are marked when they are copied into the new one (so that other threads know they must visit the new table). An incorrect version erroneously failed to use a CAS operation to mark the item. This means that the effect of a concurrent operation in the old table may not be copied across. A test, based upon a key space of size 32, with initial table capacity 4, finds the error in an average time of 8.2 ± 1.7s (using the JIT-JIT Competition Tester on a noisy machine). The size of the key space is chosen with the aim of ensuring a reasonable number of resize operations while still allowing a reasonable chance of concurrent operations on the same key.

In [26], Shann et al. proposed a concurrent array-based bounded queue. However, Colvin and Groves [7] point out an error in the design, which means that the queue is not linearizable. In fact, the same error causes the queue to sometimes deadlock. In an informal experiment, we ran the queue-oriented linearizability tester on this queue until either it seemed to deadlock or a non-linearizable history was obtained, repeating 50 times. In 29 cases, the queue deadlocked; in the other 21 cases, a non-linearizable history was found, normally within 100ms.

To demonstrate that our logging allows us to find memory consistency bugs, we carried out an experiment with workers reading and writing a shared non-volatile variable (with 60% of operations being reads, and values written from a range of size 10), and testing whether the history was linearizable. This detected the bug in an average time of just 679 ± 26ms (using the JIT Graph Search Algorithm).

Herlihy and Shavit [13, Section 14.4] describe a lock-free skiplist. This implementation assumes no hash collisions of keys. We produced an implementation which aimed to remove this assumption. Erroneously, the implementation made it possible to skip over a key that does not appear in a particular level of the skiplist. A test, based upon a bad hash function that produces just two different results, finds the bug in an average time of just 123 ± 7ms (using the JIT Graph Search Algorithm).

Finally, in [21] we implemented a sharded hash map with lock-free reads: updates to the map obtain a lock on part of the map, and then performed an update; however, reads are done without locking. An error in an early version was that an update set the status of a slot to indicate a key was in the map *before* writing a new datum into the slot; as a result, a concurrent read could read an old, previously deleted datum. A test, based upon a map with two keys, finds the bug in an average time of just 671 ± 44ms (using the JIT-JIT Competition Tester on a noisy machine).

9. CONCLUSIONS

We have presented a framework for testing linearizability of concurrent datatypes. We have adapted the Wing & Gong Tree Search Algorithm to a graph search algorithm, making it feasible to apply in practice. We have produced two generic algorithms that use the idea of just-in-time linearization as an effective form of partial-order reduction. We have produced a special-purpose algorithm for testing linearizability of queue histories. We have carried out an experimental comparison of the algorithms, and demonstrated their effectiveness in finding bugs.

The framework works well. It is quick and easy to produce test programs: our students have had no difficulty using the framework. And bugs seem to be found very quickly.

As noted in the introduction, we would like to investigate the application of the algorithms of this paper to verification, using a model checker to generate all histories of a datatype (up to some maximum length), and then using these algorithms to test those histories for linearizability.

We would like to make a few extensions to the framework. The testing framework supports only shared-memory programs. We would like to extend the framework to support distributed-memory programs. The main challenge here is logging: once a history has been obtained, the algorithms of this paper can be applied. We expect that our timestamp-based approach to logging will not work in a distributed setting, because of clock drift. One approach would be to have a dedicated logging process, to which other processes synchronously send logging messages. Alternatively, one could implement a log using a form of logical timestamps, such as Lamport timestamps [18].

For datatypes such as sets and maps, a history is linearizable if and only if the restriction to each key is linearizable. Building on the Wing & Gong Graph Search Algorithm of this paper, Horn and Kroening [15] have built a linearizability checker that tests for each key separately, giving an order-of-magnitude speed-up on longer runs.

We would like to develop more special-purpose algorithms for testing linearizability for particular datatypes. In particular, we believe it might be possible to produce algorithms for stacks and priority queues, in a style similar to our algorithm for queues in Section 5. Dodds et al. [9] give a characterization of linearizability of a history of a concurrent stack, which may prove useful.

At present, we restrict the framework to *deterministic* sequential specifications. However, some datatypes have inherently nondeterministic specifications, such as a set with an operation to return an arbitrary element. We intend to extend the framework to nondeterministic specifications. This would require a specification object that can enumerate all legal results of an operation.

Finally, there are some interesting out-standing questions about the complexity of testing linearizability of a history. As noted earlier, Gibbons and Korach [11] show that determining whether a single history is linearizable is NP-complete in general. However, for a history of a register with a *fixed* number of processors, there is a polynomial time algorithm; indeed, our JIT Graph Search Algorithm runs in linear time. This result carries across to datatypes like maps and sets. However, we are not aware of any corresponding results for datatypes such as queues and stacks: we conjecture that determining linearizability is NP-complete in these cases.

ACKNOWLEDGEMENT

REFERENCES

1. R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160, 2000.
2. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specificaitons. In *ESOP '13*. Springer, 2013.
3. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *ICALP 2015*. Springer, 2015.
4. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, pages 330–340, 2010.
5. P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV '10)*, pages 465–479, 2010.
6. R. Colvin, S. Dohery, and L. Groves. Verifying concurrent data structures by simulation. *Electronic Notes in Theoretical Computer Science*, 137:93–110, 2005.
7. R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, 2005.
8. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Transactions on Programming Languages and Systems*, 33(1):4:1–4:43, 2011.
9. M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *Proceedings of POPL '15*, 2015.
10. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Notices.*, 42(10):57–76, 2007. http://dl.acm.org/citation.cfm?id=1297033.
11. P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal of Computing*, 26(4):1208–1244, 1997.

12. T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *Proceedings of Concur*, 2013.
13. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.
14. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
15. A. Horn and D. Kroening. Faster linearizability checking via p-compositionalit. In *Proceedings of FORTE*, pages 50–65, 2015.
16. K. Kingsbury. Computational techniques in Knossos. `https://aphyr.com/posts/314-computational-techniques-in-knossos`, 2014.
17. A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, 2010.
18. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
19. Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. J. Zhang, and J. S. Dong. Verifying linearizability via optimized refinement checking. *IEEE Transactions on Software Engineering*, 39(7):1018–1039, 2013.
20. G. Lowe. *Linearizability Testing Manual*. University of Oxford. `http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/`.
21. G. Lowe. Concurrent hash maps: A comparative study. Technical report, University of Oxford, 2014.
22. M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *PLDI*, pages 521–530, 2012.
23. M. Pradel and T. R. Gross. Automatic testing of sequential and concurrent substitutability. In *International Conference on Software Engineering (ICSE)*, 2013.
24. *scala.collection.immutable.queue API documentation*. `www.scala-lang.org/api/current/#scala.collection.immutable.Queue`.
25. J. Seward, N. Nethercote, J. Weidendorfer, and the Valgrind Development Team. *Valgrind 3.3 Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd, 2008.
26. C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Seventh International Conference on Parallel and Distributed Systems*, pages 470–475, 2000.
27. V. Vafeiadis. Automatically proving linearizability. In *Proceedings of Computer Aided Verification (CAV 2010)*, volume 6174 of *LNCS*, pages 450–46. Springer, 2010.
28. M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software (SPIN'09)*, pages 261–278, 2009.
29. J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17:164–182, 1993.