# Model Checking Concurrent Collections Extended Report

Joshua H. Davis
University of Maryland, College Park
College Park, MD, USA
jhdavis@umd.edu

Stephen F. Siegel
University of Delaware
Newark, DE, USA
siegel@udel.edu

## Abstract

Concurrent collections are data structures such as sets, lists, and queues that can be safely accessed by multiple concurrently executing threads. Numerous algorithms have been introduced and developed over decades. They use a variety of synchronization mechanisms, including locks, condition variables, and atomic operations such as compare-and-set. The algorithms are notoriously difficult to get right, and various computer-aided verification approaches have been proposed to help. In this paper we introduce an approach based on small scope explicit-state model checking with a number of optimizations to ameliorate state-explosion, realized in a tool, the Collect verifier. In an extensive case study, we have applied Collect to 20 implementations from a widely-used text, verifying many of the algorithms within small scope and finding some previously undiscovered defects in others.

## 1 Introduction

Concurrent data structures offer developers a familiar interface for efficiently managing data in multithreaded programs, but are difficult to implement correctly. An implementation must behave correctly for any number of threads, each of which can perform any sequence of operations with any arguments. For any such scenario, there are an enormous number of possible thread interleavings.

Many of these data structures are *collections*, which typically provide methods to *add* and *remove* items, and sometimes to check whether the structure *contains* an item. Even in the sequential case, the space of collections is complex, incorporating hash sets, lists, stacks, queues, priority queues, and many other kinds. For each interface, a variety of implementation strategies exist, e.g., cuckoo hashing vs. linear probing, various kinds of heaps for implementing priority queues, and so on.

Concurrency adds enormous complexity, as threads must carefully synchronize to avoid erroneous behavior. Over the years, developers have explored a number of different synchronization strategies. Examples include coarse-grained locking, fine-grained locking, lazy synchronization, lock-free (or nonblocking) synchronization, and optimistic synchronization [11, 12, 26]. These strategies offer different performance tradeoffs as well as different correctness challenges. Various correctness criteria, such as *linearizability* [15], have also been introduced. Over decades, a large literature has grown around the "art" of concurrent collection specification, design, and analysis.

As the difficulties of developing correct concurrent collections became clear, researchers began to explore various computer-aided verification approaches. These approaches differ in the range of structures they can handle, the input language they accept, the degree of automation, completeness, and other factors. They draw upon techniques from model checking, proof systems, static analysis, and type theory, among others. We summarize some of this work in §6.

In this project, we are exploring the use of small scope, stateful model checking. Our goal is to develop a set of techniques and an easy-to-use tool that are applicable to a wide range of collection kinds, synchronization primitives, and strategies. The tool, the Collect verifier [6], is free and open-source. We have also been engaged in an ongoing extensive case study of all concurrent data structures in a widely-used text [14]. The results of this study are also available on the Collect web site.

The contributions of this paper are:

- §2 a precise and easy-to-understand classification of correctness specifications for concurrent collections, decomposed by collection kind, synchronization protocol, and consistency property, and incorporating "stuck" executions;
- §3 a set of small scope model-checking techniques for efficiently verifying that a concurrent collection meets its specification;
- §4 an automated command-line tool, Collect, that implements those techniques and gives the user precise control over dimensions of the small scope and other options; and
- §5 a case study applying that tool to all 20 set, queue, and priority queue implementations in [14], and which reveals several previously undiscovered defects.

## 2 Specification

We adopt the following terminology. An *interface I* consists of a list of methods, each with a name, list of input types, and output type (or *void*). An *event* for *I* comprises: (1) a method name in *I*, (2) an integer thread ID, (3) a bit indicating either *invocation* or *response*, and (4) either the list of argument

values (for an invocation) or the return value (if any, for a response).

For any $n \geq 1$, an $n$-threaded *trace* (or *history*) for $I$ is a finite sequence $\alpha$ of events for $I$ in which all thread IDs are in the range $0..n-1$, and the following holds. For any $i \in 0..n-1$, let $\alpha|_i$ be the projection of $\alpha$ onto the events in thread $i$. Then $\alpha|_i$ has the form $e_1 e_2 \cdots e_r$, where for each odd $i \in 1..r$ there is some method $f$ such that $e_i$ is an invocation of $f$ and, if $i+1 \leq r$, $e_{i+1}$ is a response from $f$. We say $e_{i+1}$ is the *matching response* to $e_i$. Note $\alpha|_i$ may be empty, or may end with an invocation with no matching response.

A concurrent object $c$ for $I$ provides an implementation of all methods in $I$. We consider executions consisting of a fixed set of $n$ threads, each of which calls methods on $c$. Each invocation or response occurs at a discrete moment in time. Recording this sequence of events results in an $n$-threaded trace for $I$. Correctness of $c$ is specified by a predicate on such traces; given any trace, such a predicate determines whether the trace is correct or incorrect.

Let $\alpha$ be an $n$-threaded trace and $i \in 0..n-1$. If $\alpha|_i$ is nonempty and ends with an invocation, we say *thread $i$ becomes stuck in $\alpha$*. We say that $\alpha$ is a *stuck* trace if there is some thread which becomes stuck in $\alpha$. If $\alpha$ is not stuck, it is a *complete* trace. We will see that in some cases, getting stuck is considered correct behavior.

We next turn to the question of how to specify correctness predicates on traces. Such a specification will consist of two parts: (1) an *oracle*, which specifies correct sequential traces, and (2) a *consistency property* which specifies how a concurrent trace should relate to sequential ones. We describe these in turn.

### 2.1 Oracles

An *oracle* $o$ for interface $I$ is a predicate on sequential (single-threaded) traces on $I$. In practice, $o$ might be implemented as a simple sequential program with an interface similar to $I$, except that nondeterministic methods take an extra argument specifying the desired return value. All calls should return, i.e., the oracle itself should never become stuck. An additional method

```
bool isAccepting(bool stuck);
```

tells whether $o$ is in an accepting state for a complete trace (by passing *false* for `stuck`) or for a stuck trace (by passing *true* for `stuck`). To check whether a sequential trace $\alpha$ is correct, one calls the operations of $\alpha$ on $o$, checks that the return values match those specified in $\alpha$, and then calls `isAccepting` to see whether $o$ accepted the trace.

In this paper, we consider three standard collection interfaces—*set*, *queue*, and *priority queue*—and describe oracles for each. Each of these collections has a well-known interface and expected behavior. For example, a set with element type $T$ supports the three methods

```
bool add(T item);  bool remove(T item);
bool contains(T item);.
```

The state of the set consists of a (mathematical) set $S$ and these methods have their standard meanings; `add` and `remove` also return *true* iff the call resulted in a change to $S$.

A *queue* supports two methods:

```
void enqueue(T item);  T dequeue();.
```

Its state is a finite sequence of $T$ and the methods have the usual FIFO semantics.

A *priority queue* supports two methods

```
void enqueue(T item, int score);
T removeMin();.
```

Its state is a finite multiset of pairs $(x, s)$ where $x \in T$ and $s$ is a nonnegative integer *score* (or *priority*). Method `enqueue` adds a pair to this multiset, while `removeMin` removes an entry $(x, s)$ with minimal $s$ and returns $x$. Note that `removeMin` is in general nondeterministic as there can be multiple entries with the same score.

These descriptions almost suffice to specify three oracles, but further information is required to complete them. For example, in a concurrent queue, what should happen when a thread invokes `dequeue` when the queue is empty? In some implementations, this call will return a special value (e.g., *null*); in others, the call is expected to block until some other thread enqueues data. The latter is an example of a *synchronization* protocol, and these protocols are reflected in variants of the queue oracle.

We consider three common synchronization protocols and describe the resulting oracle in detail in the case of a queue in Fig. 1. Each automaton accepts precisely the correct sequential traces, ignoring the arguments and return values.

An implementation in which every method invocation is expected to return, regardless of actions of other threads, is *nonblocking*. Note in Fig. 1(a) that the nonblocking queue oracle does not accept any stuck trace; each invocation must be immediately followed by the matching response.

A *bounded* queue has a specified capacity; a thread blocks if it tries to dequeue when the queue is empty or enqueue when the queue is full. Fig. 1(b) shows the behavior of bounded queue oracle with capacity 2. This is an example where ending in a stuck state is the correct behavior of the oracle.

A *synchronous* queue has an even tighter synchronization restriction: each enqueue call must synchronize with a dequeue call (and vice-versa). In other words, these method calls must overlap. The oracle for a synchronous queue (Fig. 1(c)) accepts complete traces that begin with `enqueue` and in which `enqueue` and `dequeue` alternate. Stuck executions are accepted if, after executing an equal number of `enqueue` and `dequeue` calls, one of these methods is invoked but does not respond. We will explain below how every correct stuck concurrent execution of a synchronous queue relates to correct executions of this oracle.

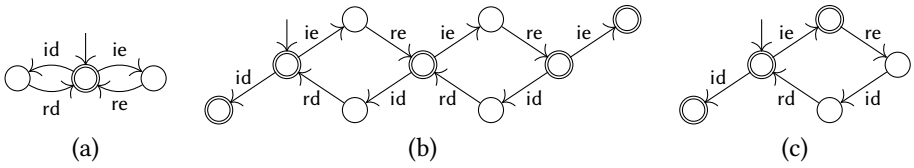


**Figure 1.** Synchronization specifications for queue oracles. ie=invoke enqueue, re=response enqueue, id=invoke dequeue, rd=response dequeue. (a) nonblocking. (b) bounded with capacity 2. (c) synchronous.

In theory these various synchronization protocols could apply to sets or priority queues, but in our case study we encountered only nonblocking sets and priority queues. In summary, we have encountered 5 distinct oracles: nonblocking set, nonblocking queue, nonblocking priority queue, bounded queue, and synchronous queue.

### 2.2 Consistency Properties

The consistency property specifies how any concurrent trace must relate to one or more oracle traces. In this paper, we consider three such properties: *sequential consistency* [18], *linearizability* [15], and *quiescent consistency* [7, 12]. Intuitively, each property is based on some partial order on events, which must be preserved when permuting the events of the concurrent trace. If one of these permutations is accepted by the oracle, the concurrent trace is correct. We now describe these properties precisely.

Let $I$ be an interface, $n \geq 1$, and $\alpha$ an $n$-threaded trace for $I$. An *operation* in $\alpha$ is a pair $e_1e_2$ where $e_1$ is an invocation and $e_2$ is the matching response, or the single event $e_1$ if $e_1$ has no matching response in $\alpha$. For operations $x$ and $y$, write $x < y$ if $x$ has a response event and that event occurs before the invocation of $y$ in $\alpha$.

Let $x_1, \ldots, x_r$ be the operations of $\alpha$, with the invocations occurring in the same order as they do in $\alpha$. Let $\pi$ be a permutation of $1..r$. Define $\pi^*(\alpha)$ to be the event sequence formed from the concatenation

$$x_{\pi^{-1}(1)}x_{\pi^{-1}(2)} \cdots x_{\pi^{-1}(r)}.$$

Note the $i$-th operation of $\alpha$ becomes the $\pi(i)$-th operation of $\pi^*(\alpha)$. Let $\bar{\pi}(\alpha)$ be the same as $\pi^*(\alpha)$ except that all thread IDs are changed to 0. We are interested in the case where $\bar{\pi}(\alpha)$ is a sequential trace accepted by an oracle. Note, however, it may not be a trace if $\alpha$ is stuck, and it can not be a trace if two or more threads become stuck, since a sequential trace can have at most one stuck call.

We say $\pi$ *preserves program order in* $\alpha$ if whenever $1 \leq i < j \leq r$ and $x_i$ and $x_j$ are operations from the same thread, $\pi(i) < \pi(j)$.

We say $\pi$ *preserves linearization in* $\alpha$ if, whenever $1 \leq i < j \leq r$ and $x_i < x_j$, $\pi(i) < \pi(j)$. Note this implies $\pi$ preserves program order.

Let $k \in 0..r$. We say $\alpha$ *is quiescent at* $k$ if for all $i \in 1..k$ and $j \in k+1..r$, $x_i < x_j$. Equivalently, in the state just after the first $2k$ events of $\alpha$, no thread has a pending method call. (Note any trace is quiescent at 0.) We say $\pi$ *preserves quiescence* in $\alpha$ if for any such $k$, whenever $0 < j \leq k$, we have $\pi(j) \leq \pi(k)$. As $\pi$ is bijective, this implies that whenever $k < j \leq r$, $\pi(k) < \pi(j)$. In short, $\pi$ will not move an event across a quiescent boundary. Observe that if $\pi$ preserves linearization then $\pi$ preserves quiescence.

The appropriate notion of correctness for stuck traces is subtle. We have already observed that if $\alpha$ has two or more stuck threads, it cannot be permuted to a sequential trace. This issue is analyzed in [4] (in the case of linearizability), which concludes that the proper generalization to stuck executions requires focusing on one stuck thread at a time, ignoring the final stuck call of all other stuck threads. The intuition is that "all of the pending operations in the stuck history need to have a justification for being stuck." We adopt this approach to specifying correct stuck executions.

Specifically, for each $i \in 1..n$ for which thread $i$ is stuck in $\alpha$, let $\alpha[i]$ be the trace obtained by removing from $\alpha$ the last event of thread $j$, for all $j \neq i$ such that thread $j$ is stuck in $\alpha$. Hence $\alpha[i]$ is a trace in which thread $i$ is the one and only thread that becomes stuck.

Now we can define the three consistency properties: *sequentially consistent* (**SC**), *linearizable* (**L**), and *quiescent consistent* (**QC**):

**Definition 2.1.** Let $c$ be a concurrent object with oracle $o$ and $\alpha$ an $n$-threaded trace of $c$. We say $\alpha$ is **SC** (resp. **L**, **QC**) if the following hold:

1. if $\alpha$ is complete then there is some permutation $\pi$ that preserves program order (resp. linearization, quiescence) in $\alpha$ such that $\bar{\pi}(\alpha)$ is accepted by $o$, and
2. if $\alpha$ is stuck then for each $i \in 0..n-1$ for which thread $i$ is stuck in $\alpha$, there is some permutation $\pi$ that preserves program order (resp. linearization, quiescence) in $\alpha[i]$ such that $\bar{\pi}(\alpha[i])$ is accepted by $o$.

We say $c$ is **SC** (resp. **L**, **QC**) if every trace of $c$ is **SC** (resp. **L**, **QC**).

From the observations above, we have $\mathbf{L} \Rightarrow \mathbf{SC}$ and $\mathbf{L} \Rightarrow \mathbf{QC}$. In general, **SC** and **QC** are not comparable [14, §3.5.1].

**Example 2.2.** Consider the complete 4-threaded trace $\alpha$ of a synchronous queue:

0:ie 1:ie 2:id 0:re 3:id 1:re 2:rd 3:rd.

The number before the ':' is the thread ID and the event abbreviations are as in Fig. 1. There is a permutation $\pi$ preserving linearization such that $\pi^*(\alpha)$ is

0:ie 0:re 2:id 2:rd 1:ie 1:re 3:id 3:rd.

Note the response in thread 0 occurs before the invocation of thread 3 in $\alpha$, so this ordering must be preserved by $\pi$. Assuming the value dequeued by thread 2 is the value enqueued by thread 0, and the value dequeued by 3 is that enqueued by 1, $\bar{\pi}(\alpha)$ is accepted by the synchronous queue oracle, so $\alpha$ is linearizable.

## 3 Verification Approach

In this section, we describe the techniques we have used to verify or refute the proposition that a concurrent collection satisfies one of the consistency properties described above. For simplicity, we assume the element type $T$ is the set of nonnegative integers.

The basic technique is stateful model checking [5] within a bounded scope. Specifically, the user specifies concrete bounds on all parameters of the verification task, such as the number of threads, the number of method calls made by each thread, and the values of the arguments. These bounds determine a finite transition system, the reachable states of which are explicitly enumerated in a depth-first search order. The states are saved (in a hash set) and the search backtracks when it encounters a state seen before.

If a violation is found, the search stack encodes a counterexample in the form of an execution prefix ending in a violating state. This can be presented to the user to help isolate the defect. If no violation is found, the parameters can be gradually increased and the process repeated. While this process does not produce an iron-clad guarantee of correctness, decades of experience support the *small scope hypothesis*, which posits that almost all defects manifest within small bounds [16].

We use the CIVL Model Checker [25]. The main input language for this tool is CIVL-C, which is essentially an extension of standard C with additional primitives and libraries supporting concurrency, assertions, assumptions, resizable arrays, and other constructs useful for verification. CIVL also employs *partial order reduction* [23] to avoid exploring different interleavings that are guaranteed to result in the same final state, soundly reducing the number of states that need to be explored. The state is also transformed into a canonical form, using techniques such as heap and process canonicalization [24], to further reduce state explosion. The model checker also has an option to perform a preemption-bounded search [20], an unsound reduction which nevertheless has proved effective at finding defects in systems with large state spaces.

A few additional CIVL features were used in this project. One option enables analysis of programs that use automated garbage collection. Such programs are not required to `free` objects allocated by `malloc`. (When this option is off, memory leaks are reported if an allocated object becomes unreachable.) Another feature is the ability to find a violation with minimal trace length (option `-min`). After a violation is found, it can be further explored with `civl replay`. This was useful in determining the root cause of defects discovered in our case study.

We now look at some of the specific techniques used to apply model checking to concurrent collection verification.

***Schedules.*** The verification problem is decomposed into independent tasks, each corresponding to a different *schedule*. An $n$-threaded schedule ($n \geq 1$) specifies a nonempty sequence of *steps* for each thread. Each step specifies a method call, including the arguments. For example, the following is a 3-threaded schedule for a set:

```
thread 0: add(0), remove(1)
thread 1: add(1)
thread 2: contains(1).
```

Given a schedule, a complete program is constructed linking the concurrent collection implementation, oracle, schedule, and a driver. The driver creates $n$ worker threads, each of which executes the operations specified by the schedule and terminates. The driver records the results of each method call in fields in the step. If all workers become stuck, due to deadlock or a non-progress cycle, there is a mechanism to force all threads to terminate while setting a special "stuck" flag. The driver then checks the correctness of the (stuck or complete) execution by enumerating permutations and checking against the oracle, according to the protocol described in §2. An assertion violation is issued if the property is violated. The model checker is applied to this program to explore all possible executions (arising from interleavings or other sources of nondeterminism). If the model checker reports that the assertions are never violated, every execution of that schedule is correct.

***Checking* SC.** Each schedule step contains an integer *result* field for a thread to store the value returned by the step's method call. These fields are initialized with a special negative value that is overwritten when the call returns. If there is an entry which is not overwritten, the thread became stuck. A 0 is written to this field if a method returning void returns.

Once all workers terminate, the completed steps of the schedule are arranged in a sequence $\alpha$ preserving thread order in a canonical way: all steps of worker 0, followed by all steps of worker 1, etc. If worker $i$ has $r_i$ steps, and $r = \sum_{i=0}^{n-1} r_i$, then there are

$$\frac{r!}{r_0! \cdots r_{n-1}!}$$

permutations $\pi$ of $1..r$ that preserve thread order. The driver enumerates these permutations. For each $\pi$, it forms the sequential schedule $\bar{\pi}(\alpha)$ and executes the oracle to see if

it accepts this sequence. As soon as it finds an acceptable one, the program terminates. If it exhausts all $\pi$ without satisfying the oracle, an assertion violation is generated and a diagnostic message is printed describing the schedule.

***Checking* L.** The process is similar to the one above, but more information must be recorded to determine when one call returned before another was invoked. In our first approach to this problem, we introduced a shared integer variable *time*, which is initially 0 and is incremented after each invocation or response event. Each step has fields for the start and end times of the call which are filled in when the method of that step is invoked or returns.

While this approach is sound, it records more information than necessary. Different interleavings may result in final schedules that differ only in their start/stop times, but represent traces that are equivalent under the linearizability relation. This can lead to extreme, and extremely unnecessary, state explosion.

In our revision, time is modeled abstractly, as follows: there are shared variables `status`, a boolean initially *false*, and `abstract_time`, an integer initially 0. Each worker in the concurrent execution uses the following atomic methods to get its start/stop times:

```
get_start_time {
  if (status) {status=false; abstract_time++;}
  return abstract_time;
}

get_stop_time {
  status=true;
  return abstract_time;
}
```

In the concurrent execution, if call $a$ returns (a "stop") before call $b$ begins (a "start"), there must occur a start after a stop (with no intervening start or stop events) at some point after $a$ returns but at or before $b$ begins. Therefore the abstract start time for $b$ will be at least one greater than the abstract stop time of $a$. Hence this abstraction suffices to define the partial order on the set of events that is needed to check **L**. (In effect, this time abstraction chooses a canonical representation of an *interval order* [17, Sec. 6.6].)

Once the concurrent execution terminates, the driver proceeds as in the **SC** case, iterating over all permutations that preserve thread order. This time, however, it filters out any permutation which moves an invocation with abstract start time $k$ before a response with abstract stop time $j < k$.

***Checking* QC.** To check all executions of a schedule are **QC**, a barrier is inserted arbitrarily into the schedule. The workers execute their pre-barrier steps, then join up, and perform their post-barrier steps. Hence the barrier becomes a quiescent point in the concurrent execution. The driver then searches for a permutation of the pre-barrier steps that is accepted by the oracle, and then a permutation of the post-barrier steps that is accepted. For the second phase, the oracle starts in the state it ended in after the first phase. An assertion violation is issued if no satisfactory pair of permutations is found.

The barrier is chosen by iterating over threads $i$. For each $i$, an integer $b_i$ is chosen nondeterministically from $0..r_i$; the barrier for thread $i$ occurs just after its $b_i$-th step. Note there are $\prod_{i=0}^{n-1}(r_i + 1)$ choices for the barrier.

The model checker explores all possible nondeterministic choices for the barrier in addition to those for the concurrent execution. If it completes without finding an assertion violation, all executions of the schedule are **QC**.

```
struct Lock {$proc owner; int count;};

/*@ depends_on \access(l); */
static $atomic_f void
Lock_acquire_aux(Lock l) {
  $when(l->owner == $proc_null)
    l->owner = $self;
}

/*@ depends_on \nothing; */
static $atomic_f void
lock_increment(Lock l) { l->count++; }
```

**Figure 2.** CIVL model of Java's `ReentrantLock`, excerpt.

***Utility classes.*** Concurrent collection implementations use various mechanisms for synchronization, including locks (and variants such as fair locks and reentrant locks), condition variables, and atomic operations (including atomic getters, setters, and compare-and-swap), and spin loops. In particular, most of the algorithms in [14] are written in Java, and make extensive use of Java concurrency interfaces and classes, including `Lock`, `Condition`, `AtomicInteger`, `AtomicBoolean`, `AtomicReference`, and `AtomicMarkableReference`. We have implemented these in CIVL-C, as well as other utility classes such as `ArrayList` and `Bin`.

In implementing the concurrent utilities, we took care to limit state explosion via sound partial order reduction. We consider the `ReentrantLock` to illustrate the techniques used.

The header file `Lock.h` declares an opaque `Lock` type (a pointer to an incomplete structure) together with functions to create, destroy, acquire, and release a `Lock`. An excerpt of `ReentrantLock.cvl`, which implements that interface, is shown in Fig. 2. The `owner` field uses CIVL's `$proc` type to record the identity of the thread that currently owns the lock; this field is `$proc_null` when the lock is free. The function to acquire the lock uses a guarded command `$when` which blocks unless the condition is *true*.

The acquire operation is decomposed using 3 auxiliary atomic functions. This is to exploit a CIVL feature which

allows an expert developer to specify facts about the dependency relation that the model checker cannot deduce on its own. The model checker's partial order reduction scheme uses this information to reduce the number of states explored. For example, the function `am_owner` must commute with any transition from another thread: if this thread is the owner, then no other thread can change that fact until this thread releases the lock; if this thread is not the owner, then again no other thread can make it the owner. Informing the model checker of this fact allows it to explore only a single transition from a state in which a call to `am_owner` is enabled. In contrast, `Lock_acquire_aux` is not necessarily independent of actions from a thread that can access the object pointed to by `l`.

***Detection and recovery from stuck executions.*** The verification approach we have described requires that a program detect when its worker threads become stuck, and then release the workers so that the main thread may resume execution. For example, a concurrent bounded queue using locks and condition variables for synchronization will become stuck if every worker invokes `dequeue` when the queue is empty. When this happens, the main thread should respond by searching for matching stuck sequential traces and conclude the concurrent execution is correct. However, while CIVL, like most model checkers, has the ability to detect and report deadlocks or cycles in the state space, it does not provide any way for the program under analysis to detect and respond to this condition.

Our solution is to extend each concurrency mechanism with an ability to detect when a stuck state is reached, set a special flag indicating this is the case, and then release the workers. For example, condition variables provide a method `await` which causes a thread to release a lock and sleep until notified by a `signal` or `signalAll` call from another thread. In our implementation of condition variables, `await` will also return when a deadlock occurs because all threads are either terminated or in an `await` call. After `await` returns the worker may call a method `isStuck()` to determine whether `await` returned due to deadlock. This feature is implemented by keeping track of the status of each thread: whether it has terminated, whether it is blocked in an `await` call, and if so, the condition variable on which it is waiting. All of this is implemented in CIVL-C code; no changes to the model checker were required.

We developed a similar mechanism, the `NPDetector` (non-progress detector), to detect when all workers reach a state in which they will loop forever without ever changing the shared state. This requires that the changes to the shared state within the loop are instrumented with a call to a function `signal` defined in this interface. The tops and bottoms of the loop are also instrumented with certain function calls. Internally, the `NPDetector` keeps track of when each thread has made it through a complete iteration without changing the shared state. Again, these routines are each implemented in a few lines of CIVL-C code. For now, a user of the `NPDetector` needs to manually insert these calls into each suspect loop; in the future, we hope to automate this process.

## 4 Collect: the Concurrent Collection Verifier

The techniques described in §3 have been implemented in a new open-source, freely available tool, the Collect verifier [6]. Collect is written in Java and extends the CIVL model checker. To use Collect, the user writes a concurrent implementation, in CIVL-C, of one of the standard collection interfaces, currently one of `Set.h`, `Queue.h`, or `PQueue.h`. As mentioned above, this implementation may use other provided utility modules. The user then invokes Collect through its command line interface, specifying the kind of collection and property to be checked, the range of schedules to generate, and other details described below.

Collect then generates a set of schedules. For each schedule, it forms a whole program comprising the user's code, the schedule, driver, and oracle, and invokes CIVL's verification engine. CIVL is invoked through its Java API, so the whole process takes place within one instance of the Java Virtual Machine. Collect is multithreaded: the user specifies the number of Java threads, and the schedules are distributed to the threads using the manager-worker pattern.

***Basic options.*** The command line option `-kind=X`, where `X` is one of `set`, `queue`, or `pqueue`, specifies the collection kind. The option `-spec=Y`, where `Y` is one of `nonblocking`, `bounded`, or `synchronous` specifies the synchronization protocol, which together with the kind determines the oracle to be used. The consistency property to be checked is specified by `-prop=Z`, where `Z` is one of `sc`, `linear`, or `quiescent`.

***Specifying the schedule scope.*** There are many ways to bound the schedule space. We describe the case for sets. The user specifies

1. an upper and lower bound on the number of threads,
2. an upper bound $M$ on the values that the set can hold (the minimum value is always 0),
3. an upper and lower bound on the total number of steps in a schedule.

Each step is specified by one of three kinds (*add*, *remove*, *contains*) and has an argument in $[0, M-1]$. Hence the total number of schedules in these bounds is finite. Similar bounds are used for queue and priority queues.

***Thread symmetry.*** Collect provides additional options to control the set of schedules generated. One is *thread symmetry* (`-threadSym`). Most of the implementations we consider are thread-symmetric as the algorithms never access a

thread ID. Consider schedules with $n$ threads and let $\Sigma$ denote the group of permutations of the $n$ thread IDs. There is a natural group action of $\Sigma$ on schedules and on concurrent traces. For $\sigma \in \Sigma$, and schedule $s$, the set of traces resulting from schedule $\sigma(s)$ must be the result of applying $\sigma$ to each trace of $s$. In particular, all executions of $s$ are **SC** (resp. **L** or **QC**) iff all executions of $\sigma(s)$ are. Hence to verify or refute one of these properties, it suffices to pick one representative from each equivalence class of schedules. This technique has been used before to verify **L**[30].

***Unsound reductions.*** Other options reduce the number of schedules generated. Unlike thread symmetry, these may cause Collect to miss a violation. Options include `-addsDominate`, which skips schedules that have more *remove*s than *add*s, and `-genericVals`. The latter is for queues and priority queues. A reasonable assumption of these implementations is that they are agnostic to the values of the items enqueued. Hence all schedules use the values $0, 1, \ldots$., in that order, for their *add* operations. For priority queues, `-distinctPriorities` keeps only schedules in which the scores of the items added are distinct.

***Pre-adds.*** Some defects may be found on short paths starting from a state in which the collection is not empty. Therefore Collect has an option to perform a specified number of *pre-adds*, add operations executed sequentially before the threads are created. The option `-npreAdds=a..b` specifies the range of values to use for the number of pre-adds.

***Termination.*** CIVL can detect both deadlocks and the presence of cycles in the reachable state space—both indicating a nonterminating execution. However a cycle may not necessarily represent an actual execution if it violates weak fairness, i.e., if there is a thread that is constantly enabled but never executes on the cycle. We added an option `-fair` that tells CIVL to ignore unfair cycles. We also implemented a utility class `FairReentrantLock`, which implements `Lock.h`, to model a Java `ReentrantLock` with fairness set to *true*.

***Hash functions.*** Collect provides two options for modeling hash functions. *Nondeterministic hashing* constructs the hash function on-the-fly by assigning and caching a nondeterministically chosen integer to each input. This option takes two parameters, a *domain bound* $m$ and *range bound* $N$. The hash function constructed accepts any nonnegative integer, reduces it modulo $m$, and returns an integer in $0..N-1$. Note there are $N^m$ such functions. The other option is to use the identity function.

# 5 Case Study

We have chosen the implementations of [14] for a case study for several reasons. First, this text provides a wealth of different kinds of concurrent collections and uses a large variety of synchronization primitives and strategies. The Java code shown in the text is almost complete, and there is also companion code available from the book web site.[1] The work is mature: a first edition appeared in 2008 [12], a revised first edition which incorporated many corrections in 2012 [13], and a second expanded edition in 2020 [14]. The text is influential, widely read, and used in courses throughout the world. The second edition acknowledges 75 people who contributed corrections. Any remaining defects have survived a high level of scrutiny and must be subtle.

Our analysis covers all data structures from the chapters on Lists, Queues, Hash Sets, and Priority Queues (chapters 9, 10, 13, and 15, respectively), for a total of 20 Java classes. Lists and hash sets implement the *set* interface, while queues and priority queues implement *queue* and *priority queue*, respectively, as described in §2.1. The names of these classes appear in Table 2, along with several variations described below.

For list-based sets, the text makes the simplifying assumption that hash functions are injective, while the hash-based sets (except `LockFreeHashSet`) make no such assumption and are designed to handle hash collisions.

## 5.1 Methods

We translated the book's Java classes to CIVL-C manually. While mostly rote, there were some challenges. The first issue is that CIVL-C is not an object-oriented language. So for each class, such as `Node`, we create a C `struct` and define the `Node` type to be a pointer to that `struct`. A method call such as `u.foo(a1,...)` is transformed to `Node_foo(u, a1,...)`. Fortunately, the runtime class of all method calls in the Java code is statically determinable. The Java classes also use generic types (e.g., `Queue<T>`), a feature not supported by CIVL-C. As explained in §3, we chose to fix $T = int$ for this study. The Java code uses `try S1... finally S2` blocks to ensure `S2` is executed before each `return` statement in `S1`; we manually inserted `S2` before each such return statement. Finally, for implementations which accept stuck executions, some insertions were required as described in §3. With these techniques, the CIVL-C code looks very close to the original Java code; Fig. 3 is a typical example.

## 5.2 Experimental Setup

We ran a large number of experiments applying Collect to these CIVL-C codes. Each experiment involved a particular command line *configuration*, i.e., choices for the options described in §4. Five of these configurations are detailed in Table 1. The table shows each configuration's range of pre-add count, thread count, and step count explored.

Configuration $C$ uses the non-deterministic model of the Java `hashCode` function, with domain bound 3 and range

[1] At this time, the code in the text [14] appears to be more up-to-date than the companion code. Unless stated otherwise, we use the text, and use the companion code only when necessary to fill in gaps.

```
protected boolean relocate(int i, int hi) {
  int hj = 0;
  int j = 1 - i;
  for (int round=0; round<LIMIT; round++) {
    List<T> iSet = table[i][hi];
    T y = iSet.get(0);
    switch (i) {
    case 0: hj=hash1(y)%capacity; break;
    case 1: hj=hash0(y)%capacity; break;
    }
    acquire(y);
    // ...
  }}
```

↓

```
static bool relocate(Set s, int i, int hi) {
  int hj = 0;
  int j = 1 - i;
  for (int round=0; round<LIMIT; round++) {
    ArrayList iSet = s->table[i][hi];
    T y = ArrayList_get(iSet, 0);
    switch (i) {
    case 0: hj=hash1(s,y)%s->capacity; break;
    case 1: hj=hash0(s,y)%s->capacity; break;
    }
    acquire(s, y);
    // ...
  }}
```

**Figure 3.** Excerpt of method relocate from [14, Fig. 13.27] PhasedCuckooHashSet (parent class of StripedCuckooHashSet) and CIVL translation below.

bound 2. For $\mathcal{E}$, we use a preemption bound of 2 in order to make the execution time of these experiments tractable. No preemption bound is imposed on other configurations. The number of schedules for various types of collections varies due to the differences in their interfaces and the use of different options for different collection kinds. Specifically, we apply -threadSym to all cases. For queues and priority queues we enable -genericVals. For priority queues only we enable -addsDominate and -distinctPriorities.

The "SQueue" case in Table 1 applies to the synchronous queues we examine. Because the add methods in those collections block until a remove call is made, pre-adds do not apply. We set the number of pre-adds to zero and increment the number of steps by one for each configuration for these synchronous queues.

### 5.3 Results

Table 2 lists the results in seconds for all experiments run. §5.4 describes the violations in detail. Runs automatically stop early if a violation is found. A "–" indicates "does not apply" or was not attempted. These results were collected on a 16-core Intel Xeon W-2145 system with 256GB RAM.

Because $\mathcal{C}$ differs from $\mathcal{B}$ only in the use of nondeterministic hashing, we do not apply $\mathcal{C}$ to data structures that do not use hashCode. All Lists assume an injective hashCode, which the nondeterministic model does not provide. The implementation of LockFreeHashSet also assumes an injective hashCode, although the text does not state this. The reachable state spaces of StripedCuckooHashSet and RefinableCuckooHashSet are unexpectedly infinite in $\mathcal{C}$ due to a previously undiscovered defect that will be discussed in §5.4 item 6. Finally, we do not run $\mathcal{E}$ on Sets or Lists due to the extremely large number of schedules to check.

### 5.4 Violations Found

We find eleven violations: three expected and eight new. As described in §3, we use civl verify -min to find a violation of minimal length and civl replay with printfs to review violations in detail. We indicate after each heading whether the violation was expected and the smallest configuration able to find the violation, as well as a brief description. We omit the SimpleTree violation from this list as it will be explained in detail below.

1. **Original** LockFreeList **sequential inconsistency (expected, $\mathcal{B}$).** We find remove contains a known **SC** violation [27]. We find no violations in the corrected version from [13].
2. SynchronousDualQueue **sequential inconsistency (unexpected, $\mathcal{B}$).** Method dequeue can incorrectly return without blocking when the queue is empty. We provide a simple patch which corrects this.
3. RefinableHashSet **null pointer dereference (unexpected, $\mathcal{B}$).** Method resize can set a reference to uninitialized memory shortly before add dereferences it. RefinableCuckooHashSet suffers the same defect.
4. LockFreeHashSet **sequential inconsistency (unexpected, $\mathcal{B}$).** This violation occurs in the remove function and allows two threads to indicate they have removed the same item, similar to violation 1 above.
5. StripedCuckooHashSet **null pointer dereference (unexpected, $\mathcal{B}$).** A thread in resize can set references to uninitialized memory immediately before another thread in relocate tries to read them.
6. StripedCuckooHashSet **infinite state (unexpected, $\mathcal{C}$).** With certain hashCode functions, this class can infinitely recurse on resize. RefinableCuckooHashSet suffers the same defect.
7. FineGrainedHeap **non-termination (unexpected, $\mathcal{A}$).** Two threads both in add can enter a cycle in state space. Using FairReentrantLock and weak fairness in thread scheduling eliminates the violation.
8. FineGrainedHeap **deadlock (unexpected, $\mathcal{B}$).** A deadlock can occur, on a priority queue with one item, between an adding thread and a removing thread, which both try to acquire a lock held by the other.
9. **Original** SkipQueue **non-termination (expected, $\mathcal{A}$).** In [12, Fig. 15.5], the findAndMarkMin method does not advance its current node reference in linked list traversal when it encounters a marked node.
10. SkipQueue **sequential inconsistency (expected, $\mathcal{E}$).** We verify SkipQueue is **QC** as expected [14, Sec.

| Config. | Pre-adds | Threads | Steps | hashCode Model | Preempt. Bound | List/Set Scheds. | Queue Scheds. | SQueue Scheds. | PQueue Scheds. |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{A}$ | 0 | 1..2 | 1..2 | identity | $\infty$ | 63 | 9 | 9 | 7 |
| $\mathcal{B}$ | 0..1 | 1..2 | 1..2 | identity | $\infty$ | 270 | 18 | 25 | 25 |
| $\mathcal{C}$ | 0..1 | 1..2 | 1..2 | non-det. | $\infty$ | 270 | 18 | 25 | 25 |
| $\mathcal{D}$ | 0..1 | 1..3 | 1..3 | identity | $\infty$ | 8108 | 58 | 83 | 156 |
| $\mathcal{E}$ | 0..1 | 1..3 | 1..4 | identity | 2 | 322930 | 166 | 223 | 1096 |

**Table 1.** The experimental configurations passed to COLLECT, including the number of pre-adds, threads, and steps. We also denote the technique used to model the hashCode function, the preemption bound, and the total number of schedules generated given these bounds for the different collection kinds.

| | Collection Name | Kind | SP | CP | $\mathcal{A}$ | $\mathcal{B}$ | $\mathcal{C}$ | $\mathcal{D}$ | $\mathcal{E}$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | CoarseList | L | N | **L** | 17 | 47 | – | 1613 | – |
| 2 | FineList | L | N | **L** | 17 | 47 | – | 1766 | – |
| 3 | OptimisticList | L | N | **L** | 18 | 49 | – | 2136 | – |
| 4 | LazyList | L | N | **L** | 17 | 49 | – | 1988 | – |
| 5 | LockFreeList | L | N | **L** | 18 | 51 | – | 1856 | – |
| | LockFreeListOriginal[3] | L | N | **L** | 18 | **48** | – | **486** | – |
| 6 | BoundedQueue | Q | B | **L** | 8 | 10 | – | 115 | 123 |
| 7 | UnboundedQueue | Q | N | **L** | 6 | 9 | – | 32 | 157 |
| 8 | LockFreeQueue | Q | N | **L** | 6 | 9 | – | 61 | 171 |
| 9 | SynchronousQueue | Q | S | **L** | 6 | 11 | – | 44 | 246 |
| 10 | SynchronousDualQueue | Q | S | **L** | 8 | 14 | – | **116** | 182 |
| | SynchronousDualQueuePatched[6] | Q | S | **L** | 8 | 14 | – | 1450 | 187 |
| 11 | CoarseHashSet | H | N | **L** | 19 | 50 | 55 | 1572 | – |
| 12 | StripedHashSet | H | N | **L** | 19 | 52 | 59 | 1732 | – |
| 13 | RefinableHashSet | H | N | **L** | 21 | **50** | **54** | **572** | – |
| 14 | LockFreeHashSet | H | N | **L** | 21 | **52** | **8** | **547** | – |
| | LockFreeHashSetPatched[2] | S | N | **L** | 19 | 56 | **8** | **1054** | – |
| 15 | StripedCuckooHashSet | S | N | **L** | 20 | 53 | $\infty$ | **896** | – |
| 16 | RefinableCuckooHashSet | S | N | **L** | 22 | 59 | $\infty$ | **1093** | – |
| 17 | SimpleLinear | P | N | **QC** | 6 | 10 | – | 39 | 447 |
| 18 | SimpleTree | P | N | **QC** | 6 | 12 | – | **44** | **204** |
| 19 | FineGrainedHeap | P | N | **L** | **6** | **11** | – | **22** | **27** |
| | FineGrainedHeapFair[1] | P | N | **L** | 7 | **11** | – | **24** | **27** |
| 20 | SkipQueueOriginal[3] | P | N | **SC** | 9 | 26 | – | **228** | **254** |
| | SkipQueueQC[4] | P | N | **QC** | 10 | 26 | – | 263 | 8582 |
| | SkipQueueSC[5] | P | N | **SC** | 9 | 23 | – | 178 | **2730** |

**Table 2.** Results of the experiments. The kind (List, Queue, Set, Priority Queue), synchronization protocol (non-blocking, bounded, synchronous), and consistency property (**L**, **QC**) are listed for each collection. Each cell in the columns marked with a configuration name indicates the time in seconds to verify the data structure for the given bound or find a violation. Experiments finding a violation are indicated by red typeface. [1]: Enables the use of fair locks. [2]: Includes our fix of the double remove bug. [3]: The earlier version of the data structure from the first edition. [4]: Configured to check **QC**. [5]: Configured to check **SC**. [6]: Includes our fix of the non-blocking remove bug.

15.5], and detect a simpler **SC** violation than that provided by the book, with 3 threads executing 4 steps.

***Extended synopsis for*** `SimpleTree` ***violation.*** In `SimpleTree` we identify an unexpected **QC** violation with $\mathcal{D}$. The `SimpleTree` is a complete binary tree storing items with score $i$ in the $i$th leaf node from the left, which are each `Bin`s of items with the same score. Each non-leaf node holds a count of the number of elements contained in the `Bin`s of its left branch. It is possible for a `removeMin` call to return *null* despite the priority queue containing one item, in a manner that violates **QC**. Consider an empty tree of height 2. An `add(0,1)` results in the state shown in Fig. 4(a). Now suppose a thread $t_1$ executes two `removeMin` calls while a thread $t_2$ calls `add(1,0)`.

Thread $t_2$ places 1 in the leftmost Bin and begins moving up the tree. It increments the counter of the parent node, resulting in Fig. 4(b). But before $t_2$ updates the root node, $t_1$ calls its first removeMin, which removes item 1 and completes, resulting in Fig. 4(c). Then $t_1$ call its second removeMin, and as the root counter is 0, it proceeds down the right edges and concludes the tree is empty, returning *null*. Finally, $t_2$ increments the root node and returns.

This violates **QC**, as there is a quiescent period after the initial add(0,1), and no sequential arrangement of the second add and the two removeMins can match the above outcome. There is no correct way a removeMin call can return *null* unless the structure is empty, which would require the prior removeMin call to return item 0 of score 1, which was added before the quiescent period.

## 6 Related Work

This is not the first study attempting to verify linearizability of algorithms from [12]. Earlier work includes [27], which uses an automata-based approach and the SPIN model checker. That approach targets list-based sets only, and is applied to 2-thread schedules in which each thread calls one method. It found some known violations of linearizability. Our study goes further in that it applies to a variety of collection kinds and to larger schedules; in addition to finding some known violations, we have found several previously unknown ones.

Several earlier projects have investigated the use of model checking techniques to verify linearizability, notably [4]. That paper introduced the generalization of linearizability to stuck executions, an idea we adopted in this project. It also developed a tool, based on the CHESS dynamic model checker, which generates and executes schedules, and applied this to find defects in .NET classes. Due to state explosion, in those experiments, 100 random schedules were generated for 3 threads, each with 3 steps. The approach is also limited to deterministic collections. We have tried to improve upon this work by ameliorating state explosion through a number of optimizations (including the use of stateful, rather than stateless, model checking), by finding creative ways to bound the schedule (and other parameter) space, by allowing nondeterministic oracles, and checking other consistency properties beyond linearizability. [22] uses a stateless model checking approach focused on C++ programs that use relaxed memory operations, a feature we do not support; a few of the examples are concurrent queues. Other work involving model checking includes [28, 30].

There are other automated approaches to verifying **L**. For example, [1, 2] introduced a thread modular abstraction and symbolic representation of the state which enables, in theory, verification for unbounded numbers of threads and inputs. This has been applied successfully to structures based on singly-linked lists. In [19], the thread modular approach is extended to verify implementations that manage memory manually (malloc/free) rather than relying on a garbage collector. This work also applies to singly-linked lists, and requires the user to specify linearization points.

Further afield, there are deductive approaches that require interaction with proof assistants or annotations to generate a proof. While requiring significantly more user effort and expertise, these provide the strongest guarantees, without bounds, and are applicable to a wide range of implementations. See [8–10, 21].

Many other consistency properties have been proposed, in addition to the three implemented in Collect. These include *eventual consistency* [29] and *quasi-linearizability* [3].

## 7 Conclusions

We have introduced a comprehensive schema for correctness specification of concurrent collections, as well as efficient model checking techniques to verify them. We implemented these techniques in a tool, Collect, using the CIVL model checker. We demonstrate the applicability of Collect in a case study analyzing 20 concurrent data structures from *The Art of Multiprocessor Programming* within various small scopes. In the process, we found eight new defects which had gone undiscovered for many years. For the implementations where no defects were found, we have at least identified significant regions of the parameter space that are defect-free and at best provided strong evidence for their correctness. This work represents, to our knowledge, the most comprehensive study to date applying computer-aided verification to concurrent collections. The project is on-going, and the results will continue to appear on a publicly-viewable dashboard to encourage community contribution [6].

Collect and our study remain limited in some respects. First, we assume a sequentially consistent memory model. If an algorithm has a data race but all sequentially consistent executions are correct, our approach may fail to detect a defect. Second, our case study examined only Java implementations, which rely on garbage collection. Implementations in languages that require manual memory management face even deeper correctness challenges. We plan to add data race detection to Collect, and to explore verification of C++ implementations in future work.

## References

[1] Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 324–338. doi:10.1007/978-3-642-36742-7_23

[2] Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2016. Automated Verification of Linearization Policies. In *Static Analysis*, Xavier Rival (Ed.). Springer, Berlin, Heidelberg, 61–83. doi:10.1007/978-3-662-53413-7_4

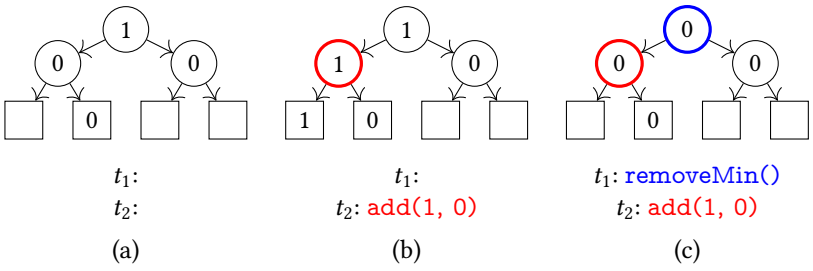


**Figure 4.** Depiction of events leading to a **QC** violation in `SimpleTree`. Coloring indicates the node last updated by each in-flight method at the time of the snapshot.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Principles of Distributed Systems*, Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–410. doi:10.1007/978-3-642-17653-1_29

[4] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 330–340. doi:10.1145/1806596.1806634

[5] Edmund M. Clarke, Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking* (2 ed.). MIT press, Cambridge, MA, USA. https://mitpress.mit.edu/books/model-checking-second-edition

[6] Collect Verifier [n. d.]. Collect: the Concurrent Collection Verifier. https://collect-verifier.org Accessed 28-Jan-2025.

[7] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 200–214. doi:10.1007/978-3-319-06410-9_15

[8] Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2, Article 19 (Sept. 2015), 43 pages. doi:10.1145/2796550

[9] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* 17 (2021), 9:1–9:59. Issue 3. doi:10.46298/LMCS-17(3:9)2021

[10] Colin S Gordon, Michael D Ernst, Dan Grossman, and Matthew J Parkinson. 2017. Verifying invariants of lock-free data structures with rely-guarantee and refinement types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 3 (2017), 1–54. doi:10.1145/3064850

[11] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems* (Pisa, Italy) *(OPODIS'05)*. Springer-Verlag, Berlin, Heidelberg, 3–16. doi:10.1007/11795490_3

[12] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann, San Francisco, CA. https://www.elsevier.com/books/the-art-of-multiprocessor-programming/herlihy/978-0-12-370591-4

[13] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming* (revised reprint ed.). Morgan Kaufmann, San Francisco, CA. https://shop.elsevier.com/books/the-art-of-multiprocessor-programming-revised-reprint/herlihy/978-0-12-397337-5

[14] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The Art of Multiprocessor Programming* (second ed.). Morgan Kaufmann, Cambridge, MA. https://www.elsevier.com/books/the-art-of-multiprocessor-programming/herlihy/978-0-12-415950-1

[15] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. doi:10.1145/78969.78972

[16] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (Sept. 2019), 66–76. doi:10.1145/3338843

[17] Mitchel T. Keller and William T. Trotter. 2023. Applied Combinatorics. https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Applied_Combinatorics_(Keller_and_Trotter)

[18] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. doi:10.1109/TC.1979.1675439

[19] Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *Proceedings of the ACM on Programming Languages* 3, Issue POPL, article No. 58 (2019), 1–31. doi:10.1145/3290371

[20] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 446–455. doi:10.1145/1250734.1250785

[21] Duc-Than Nguyen, Lennart Beringer, William Mansky, and Shengyi Wang. 2024. Compositional Verification of Concurrent C Programs with Search Structure Templates. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, New York, NY, USA, 60–74. doi:10.1145/3636501.3636940

[22] Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. *SIGPLAN Not.* 48, 10 (Oct. 2013), 131–150. doi:10.1145/2544173.2509514

[23] Doron Peled. 1998. Ten years of partial order reduction. In *Computer Aided Verification (CAV 1998) (Lecture Notes in Computer Science, Vol. 1427)*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer, Berlin, Heidelberg, 17–28. doi:10.1007/BFb0028727

[24] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. 2003. Space-Reduction Strategies for Model Checking Dynamic Software. *Electronic Notes in Theoretical Computer Science* 89, 3 (2003), 499–517. doi:10.1016/S1571-0661(05)80009-X SoftMC 2003, Workshop on Software Model Checking (Satellite Workshop of CAV '03).

[25] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, New York. doi:10.1145/2807591.2807635 Article no. 61, pages 1–12.

[26] John D. Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 214–222. doi:10.1145/224964.224988

[27] Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. Model Checking of Linearizability of Concurrent List Implementations. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 465–479. doi:10.1007/978-3-642-14295-6_41

[28] Martin Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *Model Checking Software*, Corina S. Păsăreanu (Ed.). Springer, Berlin, Heidelberg, 261–278. doi:10.1007/978-3-642-02652-2_21

[29] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. doi:10.1145/1435417.1435432

[30] Shao Jie Zhang. 2011. Scalable automatic linearizability checking. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1185–1187. doi:10.1145/1985793.1986037