Checking Concurrent Data Structures Under the C/C++11 Memory Model

Peizhao Ou and Brian Demsky
University of California, Irvine, USA
{peizhaoo,bdemsky}@uci.edu

Abstract

Concurrent data structures often provide better performance on multi-core processors but are significantly more difficult to design and test than their sequential counterparts. The C/C++11 standard introduced a weak memory model with support for low-level atomic operations such as compare and swap (CAS). While low-level atomic operations can significantly improve the performance of concurrent data structures, they introduce non-intuitive behaviors that can increase the difficulty of developing code.

In this paper, we develop a correctness model for concurrent data structures that make use of atomic operations. Based on this correctness model, we present CDSSPEC, a specification checker for concurrent data structures under the C/C++11 memory model. We have evaluated CDSSPEC on 10 concurrent data structures, among which CDSSPEC detected 3 known bugs and 93% of the injected bugs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programing; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Relaxed Memory Models; Concurrent Data Structure Specifications

1. Introduction

Concurrent data structures can improve scalability by supporting multiple simultaneous operations, reducing cache coherence traffic, and reducing the time taken by individual data structure operations. Researchers have developed many concurrent data structures with these goals [24, 35, 38]. Concurrent data structures often use sophisticated techniques including low-level atomic instructions (e.g., compare and swap), careful reasoning about the order of memory operations, and fine-grained locking. Developing correct concurrent data structures requires subtle reasoning about the data structure and the memory model, and there are numerous examples of incorrect concurrent data structures [40, 41].

PPoPP '17, February 04 - 08, 2017, Austin, TX, USA

ACM 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018749

1.1 Concurrent Data Structure Specifications

Researchers have developed several techniques for specifying correctness properties of concurrent data structures written for the sequential consistency memory model. While these techniques cannot handle the behaviors exhibited by data structure implementations under relaxed memory models, they provide insight into intuitive approaches for specifying concurrent data structure behavior.

A common approach for specifying the correctness of concurrent data structures is in terms of sequential executions of either the concurrent data structure or a simplified sequential version. The problem then becomes how do we map a concurrent execution to a sequential execution? A common criterion is *linearizability* — linearizability states that a concurrent operation can be viewed as taking effect at some time between its invocation and its response [32].

An *equivalent sequential data structure* is a sequential version of a concurrent data structure that can be used to express correctness properties by relating executions of the concurrent data structure to executions of the equivalent sequential data structure. The equivalent sequential data structure is often simpler, and in many cases, one can simply use existing well-tested implementations from libraries.

A *history* is a total order of the method invocations and responses in an execution. A *sequential history* is one where all invocations are immediately followed by the corresponding responses. A concurrent object is linearizable if for all executions: (1) the invocations and responses can be reordered to yield a sequential history under the constraint that an invocation cannot be reordered before the preceding responses and (2) the concurrent execution yields the same behavior as this sequential history.

A weaker variation of linearization is *sequential consistency*[1]. Sequential consistency only requires that there exists a sequential history that is consistent with the *program order* (intra-thread order). This order does not need to be consistent with the order that operations were actually issued in.

Unfortunately, efficient implementations of many common data structures, e.g., RCU [24], MS Queue [38], etc., for the C/C++ memory model are neither linearizable [29] nor sequentially consistent!

[1] Note that the term sequential consistency in the literature is applied to both the consistency model that data structures expose to clients as well as the guarantees that the memory system provides for loads and stores.

1.2 New Challenges from the C/C++ Memory Model

The C/C++ memory model [1, 2] brings the following two challenges that prevent the application of previous approaches for specifying the correctness of concurrent data structures:

Relaxed Executions Break Existing Data Structure Consistency Models: C/C++ data structures often expose clients to weaker (non-SC) behaviors to gain performance. A common guarantee is to provide happens-before synchronization between operations that perform updates and the operations that read those updates. These data structures often do not guarantee that different threads observe updates in the same order. For example, even when one uses the relatively strong `release` and `acquire` (i.e., `std::memory_order_release` and `std::memory_order_acquire`) memory orderings in C++, it is possible for two different threads to observe two stores happening in different orders. Thus, many data structures legitimately admit executions for which there are no sequential histories that preserve program order.

Like many other relaxed memory models, the C/C++ memory model does not define a total order over all memory operations. Instead of a total order, the C/C++ memory model is formulated as a graph of memory operations with edges representing several partial orders. This complicates the application of traditional approaches to correctness, e.g., linearization cannot be applied. Specifically, approaches that relate the behaviors of concurrent data structures to analogous sequential data structures break down due to the absence of a total ordering of the memory operations. While many dynamic tools [40, 50] for exploring code behavior under relaxed models do as a practical matter print out an execution in some order, this order insufficient as relaxed memory models generally make it possible for a data structure operation (1) to see the effects of operations that appear later in any such order (e.g., a load can read from a store that appears later in the order) and (2) to see the effects of older, stale updates.

Constraining Reorderings (Specifying Synchronization Properties): Synchronization[2] in C/C++ orders memory operations to different locations. Concurrent data structures often establish synchronization to avoid exposing their users to highly non-intuitive behaviors that may break client code. The C/C++ memory model formalizes synchronization in terms of the *happens-before* relation. The happens-before relationship is a partial order over memory accesses. If memory access x happens before memory access y, it means that the effects of x must be ordered before the effects of y.

To illustrate the meaning of synchronization in this context, consider a concurrent queue that does not properly establish synchronization between corresponding pairs of enqueue and dequeue operations. Figure 1 shows a problematic execution of such a queue: (1) thread A initializes the fields of a new object X; (2) thread A enqueues the reference to X; (3) thread B dequeues the reference to X; (4) thread B reads the fields of X through the dequeued reference. In (4), thread B could fail to see the initializing writes from (1). This surprising behavior could occur if the compiler or CPU reorders the initializing writes to be executed after the enqueue operation, i.e., reordering (1) after (2).

Thread A	Thread B
`X->field1=1; // (1)`	`Obj *r1=q->deq(); // (3)`
...	if (r1!=NULL)
`q->enq(X); // (2)`	`int r2=r1->field1; // (4)`

Figure 1. The problematic execution of a concurrent queue in which the load in (4) does not see the initialization in (1) (i.e., `r2!=1`) when the dequeue operation in (3) dequeues from the enqueue operation in (2)

1.3 Specification Language and Tool Support

This paper presents CDSSPEC, a specification checking tool that is designed to be used in conjunction with model checking tools. We have implemented it as a plugin for the CDSCHECKER model checker. After implementing a concurrent data structure, developers annotate their code with a CDSSPEC specification. To test their implementation, developers compile the data structure with the CDSSPEC specification compiler to extract the specification and generate code that is instrumented with specification checks. Then, developers compile the instrumented program with a standard C/C++ compiler. Finally, developers run the binary under the CDSSPEC checker. CDSSPEC then exhaustively explores the behaviors of the unit test and generates diagnostic reports for executions that violate the specification.

Researchers have developed tools for exploring the behavior of code under the C/C++ memory model including CDSCHECKER [40], CPPMEM [13], and Relacy [50], and also under a revised C/C++ memory model [11]. While these tools can be used to explore executions, they can be challenging to use for testing as they don't provide support beyond assertions for specifying data structure behavior.

1.4 Contributions

This paper makes the following contributions:

- **Correctness Model:** It presents a non-deterministic correctness model that captures the relaxed behaviors of concurrent data structures as a set of non-deterministic behaviors for a sequential data structure.
- **Specification Language:** It introduces a specification language that enables developers to write specifications of concurrent data structures developed for relaxed memory models in a simple fashion that captures key correctness properties. Our specification language is the first to our knowledge that supports concurrent data structures that use C/C++ atomic operations.

[2] Synchronization here is not mutual exclusion, but rather a lower-level property that captures which stores must be visible to a thread. Effectively, it constrains which reorderings can be performed by a processor or compiler.

- **A Tool for Checking C/C++ Data Structures Against Specifications:** CDSSPEC is the first tool to our knowledge that can check concurrent data structures that exhibit relaxed behaviors against specifications that are specified in terms of intuitive sequential executions.
- **Evaluation:** It shows that the CDSSPEC specification language can express key correctness properties for a set of real-world concurrent data structures, that CDSSPEC can detect bugs, and that CDSSPEC tool can unit test real world data structures with reasonable performance. Specifically, CDSSPEC detected 3 known bugs, 93% of the injected bugs, and one overly strong memory order parameter claimed to be necessary in a paper.

2. Example

Figure 2 presents a simple blocking queue that we will use to motivate our correctness model. In this example, enqueuers enqueue natural numbers, and they compete to insert a new node at the tail of the list by performing a CAS (i.e., `compare_exchange_strong`) operation on the `next` field of the `tail`. If one enqueuer succeeds, all other competing enqueuers will spin waiting for it to update the `tail`. Similarly, dequeuers compete to update the `head` to the next node it points to with a CAS operation if the `next` field of the `head` node is not `NULL`. A dequeuer returns the natural number it dequeues or `-1` if it fails. For illustration purposes, dequeuers do not recycle nodes.

```
1  class Queue {
2  public: atomic<Node*> tail, head;
3  Queue() { tail = head = new Node(); }
4  void enq(int val) {
5    Node *n = new Node(val);
6    while (1) {
7      Node *t = tail.load(acquire);
8      Node *old = NULL;
9      if (t->next.CAS(old, n, release)) {
10       tail.store(n, release);
11       return;
12     }
13   }
14 }
15 int deq() {
16   while (1) {
17     Node *h = head.load(acquire);
18     Node *n = h->next.load(acquire);
19     if (n == NULL) return -1;
20     if (head.CAS(h, n, release))
21       return n->data;
22   }
23 }};
```

Figure 2. Simple blocking queue example

The queue uses release/acquire synchronization (provided by the C/C++11 memory model) that practitioners are likely to use for this scenario. This ensures that `enq` and `deq` method calls[3] are first in-first out (FIFO) ordered, and that `deq` method calls dequeue fully initialized nodes. Under the sequential consistency memory model, the SC counterpart of this queue is linearizable. The CAS operation on the `next` field in line 9 would be the linearization point[4] for the `enq` method if it succeeds, while the load of the `next` field in line 18 from the last iteration of the loop would be the linearization point for the `deq` method.

Figure 3 presents an example execution that violates linearizability under release consistency [5]. In this execution, the loads of the `next` field (line 18) in the `deq` method from threads `T1` and `T2` both read from old values, and thus we get an execution that has no equivalent sequential history. Figure 4(a)[5] shows a sequential history for this problematic execution, in which *hb* represents the *happens-before* relation. Note that in this sequential history, the method call `x.deq` returns the value `-1`, deviating from the behavior of the corresponding sequential execution.

There are three approaches to maintaining the simplicity of the traditional approach to specifying the behavior of concurrent data structures by analogy to sequential executions:

- **Strengthen the Atomics:** We can pay extra runtime overhead and change all of the atomic operations to use the memory order `memory_order_seq_cst` and use linearizability. Figure 4(b) shows a sequential history in which the stronger *sc* order forces the `deq` method call on queue `x` to return `1`. Developers often avoid this solution because of the extra runtime overhead.
- **Constrain the Valid Usage Patterns:** Alternatively, we can constrain the situations under which the queue can be used. If we require that there is always a *happens-before* relationship between conflicting queue operations (i.e., `enq` and `deq` calls on the same queue), then the problematic execution is no longer a legal usage of the queue; hence, we only obtain executions that are linearizable. Figure 4(c) shows a sequential history of such executions.

  Although this approach by itself may excessively limit the usage scenarios of a data structure, it provides the useful insight that a data structure design can encompass specifying *admissibility* conditions, conditions under which the data structure has well-defined behavior. This approach has been used in the context of defining memory models — the C/C++ memory model states that programs with data races have undefined behavior and guarantees sequential consistency for race-free programs that correctly make use of `memory_order_seq_cst` atomics and mutexes.
- **Weaken the Specification:** It turns out that the problematic execution for the queue example occurs only when `deq` returns that the queue is empty (`-1`). We can weaken the sequential specification for the `deq` method to allow it to spuriously return `-1`. As a result, the sequential history shown in Figure 4(d) becomes legal. A similar solution is taken by the C/C++ memory model for the `trylock` method.

[3] Hereafter, the phrase "method call" or "call" refers to the invocation of a method in an execution.

[4] Under linearizability, a linearization point is the point in the execution at which the method call can be viewed as taking effect.

[5] The notations `x.enq(1)` means enqueuing `1` to queue `x`, and `x.deq(-1)` means the `deq` call on queue `x` returns `-1`.

T1	T2
`x.enq(1)`	`y.enq(1)`
`r1=y.deq()`	`r2=x.deq()`

Figure 3. A non-linearizable execution in which queues `x` and `y` are initially empty, and `r1=r2=-1`

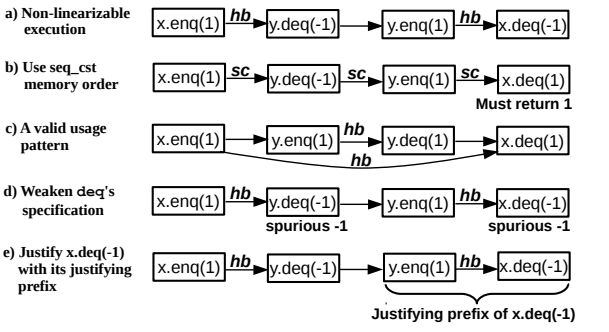

Figure 4. Sequential histories of the non-linearizable execution (shown in Figure 3) and some potential solutions

2.1 Constraining Non-determinism in the Specification

Although weakening the specification by itself works, it is not a perfect solution. One drawback is that the specification can permit uncontrolled non-determinism (i.e., the specification is too loose) so that it misses detecting even trivial correctness properties. For example, we may want to forbid the single-thread execution in which a `deq` method call following an `enq` method call returns `-1`. However, the proposed weakened specification permits such executions. It is worth noting that linearizability and sequential consistency for data structures can use non-deterministic specifications, but they also suffer from this drawback.

Upon closer examination of the problematic execution, we find a fundamental difference between the execution shown in Figure 4(d) and the execution where a `deq` call (after an `enq` call) returns `-1` in a single-thread execution. In the latter case, the program has established the *happens-before* relationship between the `enq` call and the subsequent `deq` call. Hence, under the C/C++11 memory model, the `deq` call will see the effects of the `enq` call. However, in Figure 4(d), the `enq` call on queue `x` does not *happen-before* the `deq` call on queue `x` (i.e., no *hb* between the two method calls), and thus it is acceptable for the `deq` call to return `-1`.

Therefore, CDSSPEC combines non-deterministic specifications for methods with a mechanism for justifying non-deterministic behaviors. Under the C/C++ memory model, a memory operation is required to observe the effects of the memory operations that *happen before* it. Thus, by the nature of the *hb* relation, for C/C++11 data structures, the effects of the subsequence of operations that are ordered before an operation o by the *hb* relation (or in some cases the *sc* relation) are required to be visible to o; and we call this subsequence (ending with o) a *justifying prefix* of o. The behavior of an operation in a justifying prefix justifies the operation's non-deterministic selection of behaviors in the sequential history. The sequential history shown in Figure 4(e) becomes acceptable since for the `deq` call on queue `x`, only the `enq` call on queue `y` is ordered before it in its justifying prefix (and the `enq` call on queue `y` has no effects on queue `x`), and hence the `deq` call on queue `x` is allowed to return -1.

2.2 Handling Concurrent Operations and Multiple Justifying Prefixes

The problem is unfortunately still more complicated. To illustrate this, consider one of the simplest data structures — a C/C++11 atomic register accessed by relaxed operations. A register is an object that encapsulates a value that can be read by a `read` call and updated by a `write` call. The C/C++ memory model allows a read a to return not only (1) the value written by any write $b \xrightarrow{hb} a$ such that $\neg\exists c.\texttt{iswrite}(c), b \xrightarrow{hb} c \xrightarrow{hb} a$, but also (2) the value written by any concurrent write. The solutions we have explored so far cannot specify such properties for the following reasons: for a `read` call, 1) there can be multiple `write` calls that *happen-before* the `read` call that it can read from (and a justifying prefix may not order those write calls appropriately); and 2) it can read from any `write` call that happens concurrently with it.

Thus, we define a method call m_c to be a *concurrent method call* of m if $\neg(m \xrightarrow{r} m_c \vee m_c \xrightarrow{r} m)$, where $\xrightarrow{r}$ is an ordering relation for individual atomic operations that corresponds to the ordering guarantee the implementation relies on (discussed in more detail in Section 3.1). In this example, we use the *happens-before* relation as $\xrightarrow{r}$. We can now handle registers by having a non-deterministic specification that allows a `read` call to read from a write that is not the most recent. More importantly, however, we constrain the non-determinism in the specification by extending our approach to allow accessing concurrent method calls. As a result, in the atomic register example, the non-deterministic behavior that a `read` returns the value written by a `write` that it *happens-before* is disallowed.

Thus, one can view our approach as follows: our specification specifies admissibility conditions under which a data structure has well-defined behaviors; then for all admissible executions, there exists an equivalent sequential execution on all sequential histories (that respect $\xrightarrow{r}$) with respect to a non-deterministic specification; and for each method call m with non-deterministic behaviors, m's behaviors have to be enabled by some equivalent sequential execution on one of its justifying prefixes or a concurrent method call.

2.3 Key Correctness Properties of the Queue Example

We show how our approach can capture the key correctness properties of the queue in Figure 2. Using a sequential FIFO queue as the sequential data structure, we have two options for writing a specification for the queue:

1. Deterministic Specification with Admissibility Constraints: Admissibility specifies whether a given execution

satisfies a data structure's assumptions and thus the data structure's specification should hold for the given execution. We specify admissibility with a set of *admissibility rules* that specify the conditions under which two method calls are required to be ordered relative to each other. Thus, we can simply state that a failed `deq` call should be ordered relative to other `enq` calls on the same queue, and then we have a deterministic specification with respect to the sequential FIFO.

2. Non-deterministic Specification: Another option is a non-deterministic specification (i.e., `deq` calls can spuriously fail) that applies unconditionally. We need to specify the following key correctness properties:

I. Correct Synchronization: In the example, a successful `deq` should return a fully initialized item. The sequential FIFO queue ensures that when a `deq` call d dequeues an item from a corresponding `enq` call e, d must synchronize with e; otherwise, there would exist a sequential history in which d can be ordered before e, which would violate the sequential FIFO's semantics.

II. `enq` and `deq` calls take effect in a FIFO order: We can weaken the specification for the sequential FIFO queue by allowing `deq` calls to return empty (`-1`). As a result, the weakened sequential queue ensures that `enq` and successful `deq` calls take effect in FIFO order.

III. Constrain `deq`'s non-determinism: In this case, we can state that a `deq` call d can only return empty when the corresponding `deq` call in the execution of the sequential FIFO on one of d's justifying prefixes also returns empty.

3. Correctness Model

In this section, we describe the approach in more detail and discuss the composability properties of our approach. Readers who are not interested in formalisms may wish to skip ahead to Section 4. Readers who are unfamiliar with formalisms but would like to learn may find the following references helpful: (1) readers interested in execution-oriented approaches can read Maranget *et al*. on ARM and POWER [36] and then read Mark Batty's thesis [9] as a bridge to C/C++11; (2) readers interested in an axiomatic approach can read Algave *et al*. [7] and the work of Batty *et al*. on overhauling SC; and (3) readers coming from a Java viewpoint can read Shipilëv's blog [43, 44].

3.1 Formalizing the Approach

We begin with a concurrent execution $E = (M, \xrightarrow{r})$ of a concurrent object O_c. The set M is the set of method calls in the execution E. The relation $\xrightarrow{r}$ is an acyclic, transitive ordering relation that orders atomic operations. For a C/C++11 data structure, this is in general either *hb* or *sc* and corresponds to the ordering guarantee the implementation relies on. Since methods can be composed of many atomic operations, the ordering relation $\xrightarrow{r}$ is not directly applicable to method calls. To extend the relation $\xrightarrow{r}$ to order the set of method calls M, we provide *ordering point* annotations that the developer uses to select the specific atomic operation instances that order method calls.

In order to map a concurrent execution to an execution on an equivalent sequential data structure, we first define a non-deterministic, sequentialized version of the concurrent object O_c as O_s. We then define the set of concurrent methods calls for a method call $m \in M$ in execution $E = (M, \xrightarrow{r})$ as $concurrent(m) = \{m_c \mid m_c \in M \wedge m_c \neq m \wedge \neg(m_c \xrightarrow{r} m \vee m \xrightarrow{r} m_c)\}$.

We next define the set of executions for which a concurrent object's behavior is well defined. Given a concurrent object O_c, we define an admissibility function $\circ$. For method calls $m_1, m_2 \in M$, if the admissibility condition $m_1 \circ m_2$ returns `true`, then m_1 and m_2 are not required to be ordered by $\xrightarrow{r}$. Consider the queue example. To write a deterministic specification for it, we can require an `enq` call e and a `deq` call d on the same queue that returns empty (`-1`) to be ordered by $\xrightarrow{r}$ (i.e., $e \circ d$ returns false).

Thus, the specification S for a concurrent object O_c is a tuple — $S = (O_s, \circ)$. We next define the notion of an *admissible* execution with respect to S below. Recall that an admissible execution is an execution that meets the requirements of the concurrent object's design and thus for which the concurrent object's behavior is well-specified.

Definition 1 (Admissibility)**.** *An execution $E = (M, \xrightarrow{r})$ of a concurrent object O_c is admissible for a specification $S = (O_s, \circ)$ iff: $\forall m_1, m_2 \in M, m_1 \neq m_2 \Rightarrow m_1 \xrightarrow{r} m_2 \vee m_2 \xrightarrow{r} m_1 \vee m_1 \circ m_2$.*

We next develop the framework for specifying the behaviors of admissible executions. As is the case for linearizability, we first define the corresponding sequential histories for an execution E.

Definition 2 (Valid Sequential History)**.** *Given an execution $E = (M, \xrightarrow{r})$ on a concurrent object O_c and a corresponding total order of method calls H, we say that H is a valid sequential history of E iff:*

1. *E and H contain the same set of method calls, and*
2. *$\forall m_1, m_2 \in M. m_1 \xrightarrow{r} m_2 \Rightarrow m_1$ must appear before m_2 in the total order H (i.e., $m_1 \xrightarrow{H} m_2$).*

Our specifications are non-deterministic — they may admit several behaviors for a given method call. To further constrain the specifications, we introduce the notion of a justifying subhistory, a portion of the execution E that justifies a method call exhibiting a given non-deterministic behavior.

Definition 3 (Justifying Subhistory)**.** *Given an execution $E = (M, \xrightarrow{r})$ and a method call $m \in M$, we say that an ordered list of method calls J taken from M is a justifying subhistory of m iff:*

1. *$m \in J$, and*
2. *$\forall m' \in M, m' \xrightarrow{r} m \Rightarrow m' \in J$, and*
3. *$\forall m' \in M, m' \neq m \wedge \neg(m' \xrightarrow{r} m) \Rightarrow m' \notin J$, and*
4. *$\forall m_1, m_2 \in J, m_1 \xrightarrow{r} m_2 \Rightarrow m_1$appears before m_2 in J.*

Definition 4 (Justified Behaviors)**.** *We say that the behavior of a method call m in an execution E is justified for a specification S iff: 1) There exists at least one justifying subhistory J for m, the behavior of every method call $m' \in J$ where $m' \neq m$ is justified, and m in J returns the same value as the corresponding method call in E; or 2) the return value of m is determined by the $concurrent(m)$ set.*

Note that the process of determining the return value of m by the $concurrent(m)$ set can be viewed as a non-deterministic function that takes the $concurrent(m)$ set as input and non-deterministically outputs a value. Consider the C/C++11 atomic register (accessed by relaxed operations) as an example. According to Definition 4, a `read` operation r is justified if r returns a value stored by either: (1) the most recent `write` operation in one of r's justifying histories; or (2) one of the `write` operations in the $concurrent(r)$ set. Now that we have defined what it means for given method call's behavior to be justified, we can define the correctness criteria for a concurrent object O_c.

Definition 5 (Non-deterministic Linearizability for a Valid Sequential History)**.** *We say that a concurrent object O_c is non-deterministic linearizable on a valid sequential history H for a specification S iff each method call in H (1) is justified and (2) returns a value that is the same as the corresponding method call in E or as specified in its non-deterministic specification.*

Take the execution of the blocking queue in Figure 3 as an example. Consider one of the valid sequential histories of this execution, denoted as H: `x.enq(1)` $\rightarrow$ `y.enq(1)` $\rightarrow$ `r1=x.deq()` $\rightarrow$ `r2=y.deq()`, where `r1` and `r2` are both `-1`. First, both `deq` calls are justified since there does not exist any `enq` call in any of their justifying subhistories (so they are allowed to return `-1`). Secondly, both `deq` calls return the value `-1`, which is allowed by the non-deterministic specification (spuriously returning `-1`). Thus, according to Definition 5, both objects `x` and `y` are non-deterministic linearizable on H for the non-deterministic specification. Next, we extend the notion of non-deterministic linearizability to executions and concurrent objects.

Definition 6 (Non-deterministic Linearizability for an Execution)**.** *We say that a concurrent object O_c is non-deterministic linearizable on an execution E for a specification S iff O_c is non-deterministic linearizable on all valid sequential histories of E for S.*

Definition 7 (Non-deterministic Linearizability for a Concurrent Object O_c)**.** *We say that a concurrent object O_c is non-deterministic linearizable for a specification S iff O_c is non-deterministic linearizable on all admissible executions of O_c for S.*

3.2 Composability

We next discuss the composability properties of this approach. In other words, if each object in a system is non-deterministic linearizable for its specification, then the system composed of these objects is non-deterministic linearizable for the composition of the each object's specification. This is generally a desirable property in modular design since one can reason about the correctness of each data structure in isolation. Next, we define how to compose specifications and executions.

Definition 8 (Specification Composability)**.** *The composition $S_A \oplus S_B$ of two specifications $S_A = (O_{s_A}, \circ_A)$ and $S_B = (O_{s_B}, \circ_B)$ applies O_{s_A} to operations on O_{c_A} and applies O_{s_B} to operations on O_{c_B}, and the composition of the admissibility functions $\circ_A \oplus \circ_B$ returns the value:*

1. *$m_1 \circ_A m_2$, if m_1 and m_2 are both called on O_{c_A}, or*
2. *$m_1 \circ_B m_2$, if m_1 and m_2 are both called on O_{c_B}, or*
3. *true.*

Definition 9 (Execution Composability)**.** *An execution E on the composition of two concurrent objects O_{c_A} and O_{c_B} has two partial executions $E_A = (M_A, \xrightarrow{r_A})$ and $E_B = (M_B, \xrightarrow{r_B})$. The ordering relation $\xrightarrow{r}$ for E can be expressed as the union of three disjoint sets $\xrightarrow{r} = \xrightarrow{r_A} \cup \xrightarrow{r_B} \cup \xrightarrow{r_{AB}}$, where $\xrightarrow{r_A}$ is the ordering relation between methods calls in M_A, $\xrightarrow{r_B}$ is the ordering relation between methods calls in M_B, and $\xrightarrow{r_{AB}}$ is the ordering relation between method calls in M_A and M_B. We say that such compositions are valid if $\xrightarrow{r_A} \cup \xrightarrow{r_B} \cup \xrightarrow{r_{AB}}$ is acyclic.*

It is useful to note that $\xrightarrow{hb} \cup \xrightarrow{sc}$ in C/C++11 is always acyclic, and $\xrightarrow{r_A} \cup \xrightarrow{r_B} \cup \xrightarrow{r_{AB}}$ is acyclic if $\xrightarrow{r_A}$ and $\xrightarrow{r_B}$ are a subset of $\xrightarrow{hb} \cup \xrightarrow{sc}$. This requirement places minor limitations on the ordering relation; for example, it makes using modification order (C/C++'s per memory location order) problematic. Given these definitions, we next prove the following composability theorem.

Theorem 1 (Composability)**.** *Suppose the concurrent object O_{c_A} is non-deterministic linearizable for a specification S_A; the concurrent object O_{c_B} is non-deterministic linearizable for a specification S_B; and the composition of any two admissible executions from O_{c_A} and O_{c_B} gives an acyclic ordering relation $\xrightarrow{r}$. Then the composition of O_{c_A} and O_{c_B} is non-deterministic linearizable for $S_A \oplus S_B$.*

Proof. Proof by contradiction. Suppose $E = (M, \xrightarrow{r})$ is an admissible execution that violates $S_A \oplus S_B$. Then, there must exist at least one of the object calls in E whose behavior violates its specification.

Without loss of generality, assume the object was O_{c_A}. Since E was admissible, by the definition of $S_A \oplus S_B$, $E_A = (M_A, \xrightarrow{r_A})$, where $\xrightarrow{r_A}$ is the part of $\xrightarrow{r}$ that orders methods calls to M_A, is also admissible. Now, since E violates the specification, there exist two possible cases:

1) There exists at least one sequential history H of E that violates the specification, meaning that there exists at least a method call $m_A \in M_A$ that violates the specification. Now consider H_A, H projected onto the object O_{c_A}. H_A is a valid sequential history of E_A since $\xrightarrow{r_A}$ is a subset of $\xrightarrow{r}$. Also,

since $S_A \oplus S_B$ applies O_{s_A} on O_{c_A}, m_A in the execution of H_A also violates O_{s_A}, meaning that H_A violates S_A.

2) There exists a method call m_A that was not justified. This means that all of its justifying subhistories J and its concurrent method calls violate the specification. Since $\xrightarrow{r_A}$ is a subset of $\xrightarrow{r}$, the concurrent set $concurrent(m_A)$ in E is the union of the concurrent set in E_A and a set of method calls on O_{c_B}. Since only the concurrent method calls on O_{c_A} matter in E for m_A, the $concurrent(m_A)$ sets have the same effect in E and E_A, meaning that $concurrent(m_A)$ also cannot justify m_A's behavior in E_A. Then consider J'_A, the set of all possible justifying subhistories of m_A in E_A; and J_A, the set of subhistories comprised of all of J projected onto O_{c_A}. For any justifying subhistory $j \in J'_A \Rightarrow j \in J$ because in the relation $\xrightarrow{r}$, $\xrightarrow{r_B} \cup \xrightarrow{r_{AB}}$ does not constrain the ordering of method calls on O_{c_A}. Thus, J'_A is a subset of J_A, meaning that all of m_A's justifying subhistories violate the specification.

Thus, O_{c_A} must violate its specification, and we have a contradiction. □

A key restriction is that developers must be sure that their application only generates admissible executions. Otherwise, the program may observe surprising behaviors as it generates executions outside of the correctness model.

Application to Relaxed Atomics One limitation of our approach is that it rules out using *modification-order* relation as $\xrightarrow{r}$. It is still however applicable to many data structures that intentionally use relaxed atomics to achieve performance. For example, our model generally fits data structures that extensively use relaxed atomics but provide synchronization for committing operations, e.g., the ticket lock in our benchmark. Our model can even be useful in more extreme cases. Consider a counter implemented exclusively with relaxed atomics that has two operations — `read` and `increment`. We can write a very weak specification with a sequential counter in which the `increment` or `read` call may non-deterministically return older values. If, however, after a period of time, the program reaches a synchronization point (e.g., thread join), the specification would guarantee that a `read` call must be consistent with the number of `increment` calls before the synchronization.

4. Specification Language Design

The CDSSPEC specification language captures the correctness of concurrent data structures by establishing a correspondence with an equivalent sequential data structure. Figure 5 presents the core grammar of the CDSSPEC specification language. It defines three types of specification annotations: *structure annotations*, *method annotations*, and *ordering point annotations*, discussed in Section 4.1, 4.2 and 4.3, respectively. Annotations are embedded in C/C++ comments. This format does not affect the semantics of the original program and allows the same source to be used by both a standard C/C++ compiler to generate production code and the CDSSPEC specification compiler to extract the

Structure	→	(*admissibility*)* *stateDefine*
stateDefine	→	`"@DeclareState:"` (*code*)?
		(`"@Initial:"` *code*)?
		(`"@Copy:"` *code*)?
		(`"@Clear:"` *code*)?
admissibility	→	`"@Admit:"` *label* `"<->"` *label* `"("` *cond* `")"`
Method	→	(`"@PreCondition:"` *code*)?
		(`"@JustifyingPrecondition:"` *code*)?
		(`"@SideEffect:"` *code*)?
		(`"@JustifyingPostcondition:"` *code*)?
		(`"@PostCondition:"` *code*)?
OrderingPoint	→	`"@OPDefine:"` *cond* |
		`"@PotentialOP("` *label* `"):"` *cond* |
		`"@OPCheck("` *label* `"):"` *cond* |
		`"@OPClear:"` *cond* |
		`"@OPClearDefine:"` *cond*
code	→	`<Legal C/C++ code>`
cond	→	`<A legal C/C++ boolean expression>`
label	→	`<A legal C/C++ variable name>`

Figure 5. The core grammar for the CDSSPEC specification language

CDSSPEC specification from the comments. In the following discussion, we use Figure 6 to illustrate how to apply CDSSPEC to write a non-deterministic specification for the blocking queue example from Figure 2.

```
1  /** @DeclareState: IntList *q; */
2
3  /** @SideEffect: STATE(q)->push_back(val); */
4  void enq(int val) {
5    Node *n = new Node(val);
6    while (1) {
7      Node *t = tail.load(acquire);
8      Node *old = NULL;
9      if (t->next.CAS(old, n, release)) {
10       /** @OPDefine: true */
11       tail.store(n, release);
12       return;
13     }
14   }
15 }
16 /** @SideEffect:
17 S_RET=STATE(q)->empty()?-1:STATE(q)->front();
18 if(S_RET!=-1&&C_RET!=-1) STATE(q)->pop_front();
19 @PostCondition:
20     return C_RET==-1?true:C_RET==S_RET;
21 @JustifyingPostcondition: if (C_RET==-1)
22     return S_RET==-1;*/
23 int deq() {
24   while (1) {
25     Node *h = head.load(acquire);
26     Node *n = h->next.load(acquire);
27     /** @OPClearDefine: true */
28     if (n == NULL) return -1;
29     if (head.CAS(h, n, release))
30       return n->data;
31   }
32 }
```

Figure 6. Annotated blocking queue specification example

4.1 Structure Annotations

Each data structure needs a structure annotation to specify:

Admissibility: Developers specify the admissibility of executions by using the `Admit` annotation to write a set of admissibility rules. In the absence of an admissibility rule, a pair of methods is not required to be ordered with respect

to each other. The construct takes in two method names and then takes a guard expression that specifies whether two method calls are required to be ordered with respect to each other. The admissibility guard expression can access the return value and arguments of a method call. Since Figure 6 presents a non-deterministic specification that applies to all executions, we do not need to write any admissibility rules.

Suppose we instead wanted to write a stronger, deterministic specification for the blocking queue that applies conditionally, we would write an admissibility rule: "`@Admit:deq<->enq(M1->C_RET==-1)`", where the key word `M1` refers to the first method call (i.e., the `deq` call) in the rule, and `C_RET` refers to the return value of the concurrent method call. This means whenever a `deq` call returns empty, it must be ordered relative to other `enq` calls in order for the deterministic specification to apply. Note that admissibility criteria are evaluated on pairs of concrete method calls that are not ordered relative to each other when the CDSSPEC checker checks sequential histories.

Equivalent sequential data structure: In the structure annotation, the developer can specify the internal state of the equivalent sequential data structure using a `DeclareState` annotation. The developer specifies the internal state by writing a list of variable declarations. CDSSPEC includes several useful pre-defined types — an ordered list, a set, and a hashmap so that developers can use them to specify the key properties of the data structure more conveniently. For example, we just need one line to declare an ordered list to represent a sequential FIFO in line 1 of Figure 6.

The `Initial`, `Copy`, and `Clear` annotations specify how the CDSSPEC checker should initialize, copy, and destroy the state structure when it checks an execution, respectively. They are usually empty since the CDSSPEC specification has useful defaults for the pre-defined types. For example, an ordered list is initially empty by default.

4.2 Ordering Point Annotations

In real-world data structures, API methods can generally be classified into two types, *primitive* API methods and *aggregate* API methods. Consider a concurrent hashtable as an example. It provides primitive API methods such as `put` and `get` that implement the core functionality of the data structure, and it also has aggregate API methods such as `putAll` that call primitive API methods internally. In our experience working with our benchmark set, it is generally possible to generate sequential histories and justifying subhistories for primitive API method calls for real-world data structures as they generally serialize on a single memory location; however, it is possible to observe partially completed aggregate API method calls, which unfortunately breaks the correctness criteria of linearizability and sequential consistency for data structures. Considering this issue, we design our specification language to primarily target specifying the correctness of primitive API methods of concurrent data structures that have linearizable or sequentially consistent counterparts[6].

As discussed in Section 3, the ordering relation $\xrightarrow{r}$ over method calls is the key to establishing valid sequential histories and justifying subhistories to check concurrent executions against the equivalent sequential data structure. Recall that while the *hb* or *sc* relations form the order relation $\xrightarrow{r}$ for atomic operations, data structure operations are often comprised of many atomic operations. Thus, it is important to specify which atomic operations should order method calls relative to each other.

Borrowing from VYRD's [27] concept of commit points, we allow developers to specify *ordering points* — the specific atomic operations between method invocation and response events that order method calls. Ordering points are a generalization of commit points as there can be more than one ordering point and they can include memory operations that do not commit the data structure operation. In the example, line 10 specifies that a successful CAS operation on the `next` field is the ordering point for an `enq` call, while line 27 specifies that the load operation from the last loop iteration as the ordering point of a `deq` call. Thus, when a `deq` call d returns an item enqueued by an `enq` call e, the ordering points of d and e establish a *hb* relation, which is consistent with the ordering relation between d and e (i.e., $e \xrightarrow{r} d$).

Ordering points are evaluated in each method call when the CDSSPEC checker constructs sequential histories. Thus, in the CDSSPEC specification language, we use ordering point annotations to inform the CDSSPEC checker which atomic operation invocations are ordering points for API method calls. Each ordering point annotation has a condition (that can access any variables visible at the point of the annotation) and takes effect when its condition is satisfied. We next present the individual annotations shown in Figure 5:

OPDefine: In most cases, we immediately know whether a memory operation is an ordering point. We then use the `OPDefine` annotation to specify that the atomic operation that immediately precedes the annotation is an ordering point. For example, in Figure 6, the successful CAS operation on the `next` field (line 9) is the ordering point for an `enq` call. Thus, we specify an `OPDefine` annotation in line 10 with a condition `true`. When an `enq` call is executed to line 10, the successful CAS operation, which immediately precedes the annotation, becomes the ordering point.

OPClear & **OPClearDefine**: In some cases, the same line of code in a method will be executed multiple times, e.g., an atomic operation in a loop. It is often the case that an atomic operation from the last iteration serves as the ordering point. To support this common programming idiom, we can use an `OPClear` annotation, which removes all previously observed ordering points in the current method call. Then we can

[6] Since the CDSSPEC checker allows developers to combine and check both CDSCHECKER's assertions and the CDSSPEC specifications at the same time, developers can still use assertions to check aggregate methods.

use an `OPDefine` annotation (discussed above) to define the atomic operation from the last iteration as the ordering point. To make it even more convenient to use, we introduce the syntactic sugar `OPClearDefine` to combine the effect of an `OPClear` annotation followed by an `OPDefine` annotation.

For example, in line 27 of Figure 6, we use the `OPClearDefine` annotation to specify that only the load operation on the `next` field from the last iteration of the `while` loop is an ordering point since only the last iteration is effective (i.e., returns either an item or `-1`).

PotentialOP & **OPCheck**: In some cases, it is not possible to determine whether a given atomic operation is an ordering point until later in the execution. For example, some internal methods may be called by multiple API methods. In this case, the same atomic operation can be identified as a potential ordering point for multiple API methods, and each API method later has a checking annotation to verify whether it was the actual ordering point. For example, assume that A is an atomic operation that is potentially an ordering point under some specific condition. The developer would then write a `PotentialOP` annotation with the label `LabelA` to label it as a potential ordering point, and then use the label `LabelA` in an `OPCheck` annotation at a later point.

4.3 Method Annotations

Once we establish a sequential history for an execution or a justifying subhistory for a method call, we use this component to relate the behavior of the concurrent execution to a sequential execution on the sequential history or justifying subhistory in a design-by-contract (DbC) fashion — with a set of *side effects* and *assertions* that describe the desired behavior of each API method call. Note that the method annotations can access the internal variables and methods of the equivalent sequential data structure. Additionally, they can reference the values available at the method invocation and response, i.e., the parameter values and the return value. For each API method, we require a method annotation and provide three types of annotations for this purpose:

SideEffect: After defining the internal state of the equivalent data structure, we use the `SideEffect` annotation to define the corresponding action to be performed on the equivalent sequential data structure. The `SideEffect` annotation corresponds to the sequential executions on both valid sequential histories and justifying subhistories. When this annotation is omitted for a specific API method, the method call will have no side effects on the sequential data structure.

In the example, lines 3 and 17-18 in Figure 6 specify that the `enq` and `deq` methods for the equivalent sequential data structure (i.e., the ordered list) should push an item to the back of the list and pop an item from the front of the list when the ordered list is not empty, respectively. The notation `STATE(q)` refers to the ordered list `q`, and the keyword `S_RET` refers to the return value of the sequential `deq` call.

PreCondition & **PostCondition**: In CDSSPEC, we use the `PreCondition` and `PostCondition` annotations to specify conditions to be checked before and after the execution of an API method call in the sequential execution on a valid sequential history. For example, lines 19-20 in Figure 6 state that after we execute a `deq` call in an execution on a sequential history, we should check that the corresponding concurrent `deq` call returns either empty (`-1`) or the same item as the front of the ordered list. The keyword `C_RET` here refers to the return value of the concurrent `deq` call.

JustifyingPrecondition & **JustifyingPostcondition**: For an API method call m, we use the `JustifyingPrecondition` and `JustifyingPostcondition` annotations to constrain m's non-deterministic behaviors by specifying the condition to be checked before and after we execute m in the sequential execution on m's justifying subhistories. In this annotation, developers can use the primitive `CONCURRENT`, which represents the set of concurrent method calls (of m), to write many types of non-deterministic specifications.

For example, lines 21-22 in Figure 6 state that when a `deq` call spuriously returns empty (`-1`), there must exist an execution on one of its justifying subhistories such that the `deq` call in that execution also returns empty.

Nested API Method Call When an API method calls another API method, CDSSPEC only treats the outermost method call as an API method call and any other API method calls as internal calls. Thus, only the precondition, postcondition, justifying precondition, and justifying postcondition of the outermost API method call must be satisfied.

5. Implementation

Many concurrent data structures have an unbounded space of executions, meaning that we cannot verify correctness by checking individual executions. However, many bugs can be exposed by using the specifications to check executions generated by model checking unit tests. We have implemented the CDSSPEC checker as the combination of a specification compiler and a plugin for the CDSCHECKER framework. CDSSPEC can exhaustively explore (with a few limitations) behaviors for unit tests and provide diagnostic reports for executions that violate the specifications.

5.1 Specification Compiler

The specification compiler translates an annotated C/C++ program into an instrumented C/C++ program that will generate execution traces containing the dynamic information needed to construct the sequential history (and justifying subhistories) and check the specification assertions. We next describe the key annotation actions and methods that the CDSSPEC compiler inserts into the instrumented program.

Ordering Points: Ordering point annotation actions are inserted where they appear with a corresponding conditional guard expression.

Method Boundary: To identify a method's boundaries, CDSSPEC inserts *method_begin* and *method_end* annotation actions at the beginning and end of API methods. Since checking occurs after CDSCHECKER has completed

an execution, we store the dynamic information of API method calls (i.e., the arguments and return value) that the CDSSPEC checker can access.

Side Effects, Assertions and Admissibility Checks: These are all compiled into methods which can be called by the CDSSPEC checker. The side effect and assertion methods can access the equivalent sequential data structure's state. The admissibility methods take two method call instances as arguments and return whether they are required to be ordered with respect to each other.

5.2 Dynamic Checking

At this point, we have an execution trace with the necessary annotations to construct a sequential history (and thus the justifying subhistory of each method call) and to check the execution's correctness. Before constructing the sequential history, the CDSSPEC plugin first collects the necessary information for each method call, which is the boundaries for the method call, the ordering point annotations, and the dynamic information (i.e., the arguments and return value).

Extracting the Ordering Relation $\xrightarrow{r}$: The CDSSPEC checker uses the *hb* and *sc* ordering of the ordering points to establish $\xrightarrow{r}$. Therefore, given two atomic operations X and Y that are both ordering points of two method calls A and B: $X \xrightarrow{hb} Y \vee X \xrightarrow{sc} Y \Rightarrow A \xrightarrow{r} B$.

Checking Admissibility: After establishing the ordering relation $\xrightarrow{r}$ for an execution, the CDSSPEC checker checks that for any two method calls, they are either ordered by $\xrightarrow{r}$ or that the admissibility does not require them to be ordered relative to each other. If there exists an execution that does not satisfy this criterion, our tool does not check the correctness properties and prints a warning.

Generating and Checking Sequential Histories: Since the C/C++11 memory model requires that the union of the *hb* and *sc* relation is acyclic, when each API method call has no more than one ordering point, which is the case for all of our benchmarks, the CDSSPEC checker guarantees that $\xrightarrow{r}$ is acyclic.

The CDSSPEC checker then topologically sorts $\xrightarrow{r}$ to generate a sequential history and checks that all method calls in the sequential history return the same value as in the concurrent execution. When multiple histories satisfy $\xrightarrow{r}$, by default we generate and check all possible histories. Since the set of all possible topological sortings can be large in some cases, we also provide the option of randomly generating and checking a user-customized number of sequential histories.

Checking Constrained Non-deterministic Behaviors: When a method call m has non-deterministic behaviors, the CDSSPEC checker then topologically sorts $\xrightarrow{r}$ to generate m's justifying subhistories, and ensure that there must exist at least one justifying subhistory against which the sequential execution satisfies the justifying condition.

Satisfaction Cycles One limitation of CDSCHECKER is that it may not generate executions with satisfaction cycles (i.e., out-of-thin-air executions) [10, 12, 15]. In the context of CDSSPEC, we believe that this limitation is reasonable in that we can still find many data structure bugs. Moreover, as others note [10], it is unclear whether satisfaction cycles are observable in practice, and thus bugs involving satisfaction cycles may be unlikely to affect deployed systems.

6. Evaluation

We have evaluated CDSSPEC on a contention-free lock, a port of the Linux kernel's reader-writer spinlock, two types of concurrent queues, a work stealing deque [34], and the Michael Scott queue from the CDSCHECKER benchmark suite; several data structures which were originally ported for AUTOMO [41] — an RCU implementation, a sequential lock, a ticket lock [42]; and a concurrent hashtable ported from Doug Lea's Java ConcurrentHashMap [35]. We report execution times on an Intel Xeon E3-1246 v3 machine. We have made both the CDSSPEC checker framework and benchmarks publicly available at `http://plrg.eecs.uci.edu/cdsspec`.

6.1 Expressiveness

We first report our experiences writing specifications. Due to space constraints, we describe the specifications in detail for a select set of benchmarks.

Concurrent hashtable: The concurrent hashtable was ported from the Java ConcurrentHashMap. The implementation uses an array of atomic variables to store the key/-value slots, which are divided into multiple concurrent segments protected by different locks. This implementation uses `seq_cst` atomics to establish strong orderings between the `get` and `put` methods on the same key. Thus, we can simply map it to a sequential deterministic hashtable.

In this implementation, the `put` method immediately acquires a lock. If a `get` method finds an entry in the first search, it performs a `seq_cst` load on the value, which will form a *sc* edge with the corresponding `seq_cst` update on the value in the `put` method; otherwise, the `get` method will acquire the lock and search the list again. As a result, a `get` method call will be ordered with a `put` method call on either the write/read of the value or the release/acquire of the lock. Thus, we annotate these operations as the ordering points for the `get` and `put` methods.

Chase-Lev Deque: This is a bug-fixed version of a published C11 adaptation of the Chase-Lev deque [34]. This implementation establishes *hb* ordering between all successful `steal` method calls and allows `steal` method calls to execute concurrently with the `take` and `push` method calls. Since `take` and `push` methods should be called from the same thread, we specify an admissibility condition that `take` and `push` calls should be ordered with respect to each other. Our specification allows `steal` and `take` method calls to spuriously return empty. We maintain the state as an ordered list. A `push` call pushes back its value to the list, and a `steal` or `take` call pops the first or last item from the list (when the list is not empty) and checks whether that item is consistent

with their return values. In order to make the specification tighter, for the `take` method, we impose a justifying condition that ensures that when it fails and the maintained list is not empty, there must exist concurrent `steal` method calls that take each element in the list.

If a successful `steal` method call reads a value from the deque, it should be ordered after the `push` method call that wrote that value to the deque. Therefore, we specified the store and load operations to the array in the `steal` and `push` methods as their ordering points. Since `take` and `push` calls are intended to be from the same thread, we specified the last operation in the `take` method as its ordering point.

Linux Reader-Writer Lock: A reader-writer lock allows either multiple concurrent readers or one exclusive writer to hold a lock. We abstract it with a boolean variable `writer_lock` to store whether the write lock is held and an integer variable `reader_cnt` to store the number of threads that currently hold the read lock.

Initially, we did not allow `write_trylock` to spuriously fail, and CDSSPEC checker reports a specification violation. We then analyzed the code and found that `write_trylock` has a transient side effect (i.e., subtract and then restore a bias if it fails), thus causing racing `write_trylock` calls to fail while the sequential specification would force it to succeed. We then modified the specification of `write_trylock` to allow spurious failures so that our correctness model fits this data structure. This shows CDSSPEC can help developers iteratively refine the specifications of data structures.

Ticket Lock: This is a synchronization mechanism that controls which thread can enter a critical section. The data structure maintains two atomic variables — `nowServing` and `curTicket`. The `lock` method first grabs a ticket by atomically executing a `fetch_add` operation on `curTicket` and then spins until `nowServing` is equal to its ticket number, and the `unlock` method increases `nowServing`.

It is noteworthy that the `fetch_add` operation (read-modify-write) in the `lock` method has `relaxed` memory order, and hence the `lock` methods do not establish synchronization at the `fetch_add` operation on the `curTicket` variable, making us unable to use the accesses on `curTicket` to establish the ordering relation $\xrightarrow{r}$. However, the data structure actually establishes proper synchronization elsewhere — on the update/read of the `nowServing` variable. Thus, we can still write a CDSSPEC specification for the ticket lock.

6.2 Ease of Use

While existing model-checking tools support various relaxed memory models, they either do not support the C/C++11 memory model or only provide simple user-specified assertions (e.g., `assert()`) [4, 6, 40]. In this sense, CDSSPEC is complementary to these tools. Conceptually, CDSSPEC can be extended to support other relaxed memory models (such as TSO, PSO, etc.) and used along with these model checking tools as long as we can define an equivalent ordering relation to the *hb* and *sc* relations in other relaxed memory models.

On average, using CDSSPEC to specify our benchmarks required us to write 11.5 lines of specifications for each benchmark, indicating the CDSSPEC is easy to use. We summarize why each CDSSPEC component is easy to write: (1) specifying sequential data structures is easy and straightforward, and developers can often just use an off-the-shelf implementation from a library. Even the sequential specification of those with non-deterministic behaviors are easy to understand. For example, the non-blocking Michael & Scott queue has the same justifying condition for the `dequeue` method as our simple blocking queue. (2) When developers specify ordering points, they only need to know what operations order method calls without needing to specify the subtle reasoning about the corner cases involving interleavings and reorderings introduced by relaxed memory models. In fact, all of the ordering points that we specify in our benchmarks are consistent with the commit points for their linearizable counterparts under the SC memory model. In a total of 27 API methods, we only specified 33 ordering points (1.22 per method on average), and each ordering point only takes 1 line of specification. (3) For admissibility rules, the fact that we express them at the abstraction of methods makes them easy to both reason about and write. Overall, we only used 7 lines to specify admissibility rules in a total 1,253 lines of code (omitting blank lines and comments).

Specification Errors While it is possible for developers to write incorrect CDSSPEC specifications against which the CDSSPEC checker reports no violations, in our experience specification errors typically cause the CDSSPEC checker to report violations (and thus help with refining specifications) for the following reasons:

- Admissibility: Admissibility in CDSSPEC specifications defaults to allowing all possible executions. This ensures that specification writers have to explicitly identify executions in which the specification should not hold, and by default the CDSSPEC checker will assume the specification should be checked on all executions.
- Assertions & Side Effects: The CDSSPEC checker generates and checks all possible sequential histories, so in practice, unless developers do not specify any assertions or specify very trivial assertions, the probability of writing incorrect assertions and side effects for which all histories pass is low. Our experience in refining the Linux Reader-Writer Lock's specification discussed in Section 6.1 conforms this.
- Ordering points: Ordering points can be viewed as an efficient way to construct sequential histories and justifying subhistories. Viewed in this context, mistakes specifying ordering points typically result in spurious errors being reported by the CDSSPEC checker.

Benchmark	# Executions	# Feasible	Total Time (s)
Chase-Lev Deque	893	158	0.10
SPSC Queue	18	15	0.01
RCU	47	18	0.01
Lockfree Hashtable	6	6	0.01
MCS Lock	21,126	13,786	3.00
MPMC Queue	2,911	1,274	4.83
M&S Queue	296	150	0.03
Linux RW Lock	69,386	1,822	13.71
Seqlock	89	36	0.01
Ticket Lock	1,790	978	0.17

Figure 7. Benchmark results

6.3 Performance

Figure 7 presents performance results for CDSSPEC on our benchmark set. We list the number of the total executions that CDSCHECKER has explored, the number of the feasible executions that we checked the specification for, and the time the benchmark took to finish. All of our benchmarks finished within 14 seconds, and 9 out of 10 finished within 5 seconds, and most took less than 1 second to finish.

6.4 Finding Bugs

We also examine the effectiveness of CDSSPEC for detecting known bugs, injected bugs, and potentially overly strong memory order parameters with these benchmarks.

6.4.1 Known Bugs

M&S queue: In this benchmark, two bugs were found by AUTOMO [41], one in the `enqueue` method and the other in the `dequeue` method. Both bugs use weaker than necessary memory order parameters so that the queue behaves incorrectly. We ran the buggy implementation with our specification, and the CDSSPEC checker exposed both bugs by showing specification violations in which a dequeue either incorrectly returns empty or violates the FIFO order. Note that neither bug was found by CDSChecker, and this shows that by writing specifications, we detect more real bugs.

Chase-Lev deq: CDSCHECKER found a bug (using a weaker than necessary ordering parameter) in this benchmark, in which a load in the `steal` method can read from an uninitialized memory location when a steal and push concurrently execute and the push resizes the queue [40]. In order to show that our specification can also expose the same bug, we turn off the CDSCHECKER uninitialized load report by initializing the new array created by the resize operation. We then ran the buggy implementation under our specification, and CDSSPEC checker reported the bug when a `steal` method call returned the wrong item.

6.4.2 Injected Bugs

To further evaluate CDSSPEC, we injected bugs in our benchmarks by weakening the ordering parameters of atomic operations to the next weaker parameters, i.e., changing `seq_cst` to `acq_rel`, `acq_rel` to `release/acquire`, and `acquire/release` to `relaxed`. We weakened one operation per each trial and covered all of the atomic operations that our tests exercise. While this injection strategy may not reproduce all types of errors that developers make, it does simulate errors caused by misunderstanding the complicated semantics of relaxed memory models. More importantly, these errors not only exist in real-world data structures (e.g., all the known bugs mentioned in Section 6.4.1 and the overly strong parameters mentioned in Section 6.4.3 involve using incorrect/inappropriate parameters) but also are exceedingly difficult to reason and find even by experts (e.g., the Chase-Lev deque C11 implementation).

Figure 8 shows the results of the injection detection experiment, which introduces three types of bugs: (1) the column *Built-in* represents specification-independent bugs detected by CDSCHECKER built-in check (i.e., data races and uninitialized loads); and (2) the column *Admissibility* represents a failure to satisfy the admissibility condition once all executions pass the built-in check; and (3) the column *Assertion* represents a specification violation once all executions pass the built-in check and the admissibility condition. The detection rate is the number of injections for which we detected a bug divided by the total number of injections. Strictly speaking, a data structure implementation that allows inadmissible executions may still exhibit correct behaviors under certain usage scenarios; however, it may very well violate the original design intention of the data structure. For example, an MPMC queue without proper synchronization works correctly when only used in a single thread, but this is by no means what such a data structure is designed for.

Benchmark	# Injection	# Built-in	# Admissibility	# Assertion	# Rate
Chase-Lev Deque	7	3	0	4	100%
SPSC Queue	2	0	0	2	100%
RCU	3	3	0	0	100%
Lockfree Hashtable	4	2	0	2	100%
MCS Lock	8	4	0	4	100%
MPMC Queue	8	0	4	0	50%
M&S Queue	10	3	0	7	100%
Linux RW Lock	8	0	0	8	100%
Seqlock	5	0	0	5	100%
Ticket Lock	2	0	0	2	100%
Total	57	15	4	34	93%

Figure 8. Bug injection detection results

Although the CDSSPEC checker is a unit testing tool and limited to small-scale tests to explore common usage scenarios, CDSSPEC was able to find 100% of injections for most data structures (9/10) and to find 93% of the injections on average. For our 57 injections, 15 of them were detected by checks in CDSCHECKER and 38 additional injections were detected by CDSSPEC. This shows that by writing specifications, we detect significantly more fault injections.

MPMC Queue: This is an array-based implementation with read/write counters. Strictly speaking, it is buggy; but the bug requires a load to read from a store from a previous counter epoch (and thus the bug may be acceptable in practice). Several atomic operations have stronger orders than strictly necessary, which only serves to make triggering the bug more difficult (with the condition that the bug requires $>$100,000 threads and a counter rollover). Our test cases are small enough not to trigger a 16-bit counter rollover.

6.4.3 Overly Strong Parameters

In the injection detection experiment, we found that in the Chase-Lev deq C11 implementation, weakening one `seq_cst` CAS operation on the `Top` variable to `relaxed` triggers no specification violation. We then carefully reviewed the code and believe this is unnecessarily strong. We contacted the paper's authors, and they confirmed that the stronger parameter is not necessary. This shows that CDSSPEC specification can help developers (even experts) avoid overly strong memory ordering parameters and as a result can potentially increase performance.

Limitation of Unit Tests The CDSSPEC checker framework relies on developers providing unit tests. While certain bugs cannot be easily caught by unit tests, in our experience, writing simple unit tests to exercise each corner case (e.g., dequeuing from an empty queue, triggering resize, and racing for the last element) is an effective way to find bugs. To detect both the existing and injected bugs, we only used test cases with 3 or fewer threads, and each thread contained fewer than 2 API method calls on average (5 at most). For example, we detected an existing bug in the Chase-Lev deq implementation by using only 2 threads — a main thread that pushes 3 items and takes 2 items, and a worker thread that tries to steal two items. This shows that simple unit tests can help detect tricky bugs involving the complicated semantics of relaxed memory models.

7. Related Work

Researchers have proposed and designed specifications and approaches to find bugs in concurrent data structures based on linearization. Early work [51] proposed using linearizability to test and verify concurrent objects. Line-up [17] builds on the Chess [39] model checker to automatically check deterministic linearizability. Paraglider [49] supports checking with and without linearization points using SPIN [33]. VYRD [27] is conceptually similar to CDSSPEC— developers specify commit points for concurrent code. These approaches assume the SC memory model and a trace that provides an ordering for method invocation and response events. Our work extends the notion of equivalence to sequential executions to the relaxed memory models used by real systems.

Amit *et al*. [8] present a static analysis based on TVLA for verifying linearizability of concurrent linked data structures. Hemed *et al*. [30] have generalized linearizability to formally describe concurrency-aware objects. Researchers have designed techniques that can automatically prove linearizability [25, 46, 47]. Thread quantification can also verify data structure linearizability [14]. Colvin *et al*. formally verified a list-based set [22]. While these approaches provide stronger guarantees than CDSSPEC, they target the SC memory model and were typically used to check simpler data structures.

Researchers have proposed specification languages for concurrent data structures. Refinement mapping [3] provides the theoretical basis for designing and using specifications. Commit atomicity [28] can verify atomicity properties. Concurrit [26] is a domain-specific language that allows writing scripts to specify thread schedules to reproduce bugs and is useful when developers already have some knowledge about a bug. NDetermin [19] infers non-deterministic sequential specifications to model the behaviors of parallel code.

Batty *et al*. [10] present an approach to specifying the behavior of C/C++11 code in terms of abstractions that also support composability. Their specifications utilize relaxed atomic operations to specify the behaviors of C/C++11 code and thus may not be much easier to reason about than the original implementation. Tassarotti *et al*. [45] verify an RCU implementation using a logic for weak memory models. They capture the correctness of the values returned by reads in terms of an evolution of states, and they require that a read return a value that is at least as new as the specified state.

Bounded relaxation of data structure specifications has been proposed as an approach that allows increasing the performance of concurrent data structures [31]. Unfortunately, this approach does not model the release-acquire synchronization behavior of C/C++11 as there is in general no fixed bound for how far traces for data structure like our example queue can diverge from the traditional specification.

GAMBIT [23] uses a prioritized search technique that combines stateless model checking and heuristic-guided fuzzing to unit test code under the SC memory model. RELAXED [21] explores SC executions to identify executions with races and then re-executes the program under the PSO or TSO memory model to test whether the relaxations expose bugs. CheckFence [16] is a tool for verifying data structures against relaxed memory models and takes a SAT-based approach instead of the stateless model checking approach.

Researchers have developed verification techniques for code that admits only SC executions under the TSO and PSO memory models [18, 20]. Researchers have developed tools to infer memory orderings for C/C++ code that ensure that all executions are equivalent to SC executions [37, 41]. Vafeiadis *et al*. [48] have extended the concurrent separation logic to reason C/C++11 programs.

8. Conclusion

CDSSPEC makes it easier to unit test concurrent data structures written for the C/C++11 memory model. It extends and modifies classic approaches to defining the desired behaviors of concurrent data structures with respect to sequential versions of the same data structure to apply to the C/C++ memory model. Our evaluation shows that the approach can be used to specify and test correctness properties for a wide range of data structures.

References

[1] ISO/IEC 14882:2011, Information technology – programming languages – C++.

[2] ISO/IEC 9899:2011, Information technology – programming languages – C.

[3] M. Abadi and L. Lamport. The existence of refinement mapping. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[4] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015.

[5] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, 1996.

[6] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 2013.

[7] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[8] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.

[9] M. Batty. The C11 and C++11 concurrency model. *Ph.D. thesis, University of Cambridge*, 2014.

[10] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*, 2013.

[11] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the Symposium on Principles of Programming Languages*, 2016.

[12] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *Proceedings of the 2015 European Symposium on Programming*, 2015.

[13] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*, 2011.

[14] J. Berdine, T. Lev-Ami, R. Manivich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008.

[15] H. Boehm and B. Demsky. Outlawing Ghosts: Avoiding out-of-thin-air results. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2014.

[16] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[17] 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*, 2010.

[18] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008.

[19] J. Burnim, T. Elmas, G. Necula, and K. Sen. NDetermin: Inferring nondeterministic sequential specifications for parallelism correctness. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[20] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

[21] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[22] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification*, 2006.

[23] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[24] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[25] C. Drăgoi, A. Gupta, and T. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 2013.

[26] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[27] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: Verifying concurrent programs by runtime refinement-violation detection. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[28] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, 2004.

[29] A. Haas, C. M. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, 2012.

[30] N. Hemed, N. Rinetzky, and V. Vafeiadis. Modular verification of concurrency-aware linearizability. In *Proceedings of the 29th International Symposium on Distributed Computing*, 2015.

[31] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[32] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[33] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2003.

[34] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[35] D. Lea. util.concurrent.ConcurrentHashMap in java.util.concurrent the Java Concurrency Package. `http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`.

[36] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, 2012.

[37] Y. Meshman, N. Rinetzky, and E. Yahav. Pattern-based synthesis of synchronization for the C++ memory model. In *Formal Methods in Computer-Aided Design*, 2015.

[38] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[39] M. Musuvathi, S. Qadeer, P. A. Nainar, T. Ball, G. Basler, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008.

[40] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2013.

[41] P. Ou and B. Demsky. AutoMO: Automatic inference of memory order parameters for C/C++11. In *Proceeding of the 30th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[42] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

[43] A. Shipilëv. `https://shipilev.net/`. Oct. 2016.

[44] A. Shipilëv. Java memory model pragmatics. `https://shipilev.net/blog/2014/jmm-pragmatics/`. Sep. 2016.

[45] J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[46] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proceedings of the 2009 Conference on Verification, Model Checking, and Abstract Interpretation*, 2009.

[47] V. Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22th International Conference on Computer Aided Verification*, 2010.

[48] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2013.

[49] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *International SPIN Workshop on Model Checking Software*, 2009.

[50] D. Vyukov. Relacy race detector. `http://relacy.sourceforge.net/`, 2011 Oct.

[51] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing - Special issue on parallel I/O systems*, 17(1-2):164–182, Jan./Feb. 1993.