# CDSSpec: Testing Concurrent Data Structures Under the C/C++11 Memory Model

Peizhao Ou and Brian Demsky

## ABSTRACT

Concurrent data structures often provide better performance on multi-core platforms, but are significantly more difficult to design and verify than their sequential counterparts. The C/C++11 standard introduced a weak language memory model supporting low-level atomic operations such as compare and swap (CAS). While these atomic operations can significantly improve the performance of concurrent data structures, programming at this level introduces non-intuitive behaviors that significantly increase the difficulty of developing code.

In this paper, we present CDSSPEC, a specification language checker that allows developers to write simple specifications for low-level concurrent data structures that make use of C/C++11 atomics and check the correctness of concurrent data structures against these specifications. CDSSPEC is designed to be used in conjunction with model checking tools and we have implemented it as a plugin to CDSCHECKER. We have evaluated CDSSPEC by annotating and checking several concurrent data structures.

## 1. INTRODUCTION

Concurrent data structure design can improve scalability by supporting multiple simultaneous operations, reducing memory coherence traffic, and reducing the time taken by an individual data structure operation. Researchers have developed many concurrent data structure designs with these goals [6, 11]. Concurrent data structures often use sophisticated techniques including low-level atomic instructions (e.g., compare and swap), careful reasoning about the order of loads and stores, and fine-grained locking. For example, while the standard Java hash table implementation can limit scalability to a handful of cores, more sophisticated concurrent hash tables can scale to many hundreds of cores [10].

The C/C++ standard committee extended the C and C++ languages with support for low-level atomic operations in the C/C++11 standard [1, 2, 4] to enable developers to write portable implementations of concurrent data structures. To support the relaxations typically performed by compilers and processors, the C/C++ memory model provides weaker semantics than sequential consistency [9] and as a result, correctly using these operations is challenging. Developers must not only reason about potential interleavings, but also about how the processor and compiler might reorder memory operations. Even experts make subtle errors when reasoning about such memory models.

Researchers have developed tools for exploring the behavior of code under the C/C++ memory model including CDSCHECKER [12], CPPMEM [3], and Relacy [14]. These tools explore behaviors that are allowed under the C/C++ memory model. While these tools can certainly be useful for exploring executions, they can be challenging to use for testing as they don't provide support (other than assertions) for specifying the behavior of data structures. Using assertions can be challenging as different interleavings or reorderings legitimately produce different behaviors, and it can be very difficult to code assertions to check the output of a test case for an arbitrary (unknown) execution.

This paper presents CDSSPEC, a specification language and 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.

### 1.1 Background on Specifying the Correctness of Concurrent Data Structures

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

One approach for specifying the correctness of concurrent data structures is in terms of equivalent 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 an equivalent sequential execution? A common criterion is *linearizability* — linearizability simply states that a concurrent operation can be viewed as taking effect at some time between its invocation and its return (or response) [8].

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 original concurrent data structure with 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 the STL library.

An execution *history* is a total order of method invocations and responses. A *sequential history* is one where all invocations are followed by the corresponding responses immediately. A concurrent execution is correct if its behavior is consistent with its equivalent sequential history replayed on the equivalent sequential data struc-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

ture. A concurrent object is linearizable if for all executions:

1. Each method call appears to take effect instantaneously at some point between its invocation and response.
2. The invocations and responses can be reordered to yield a sequential history under the rule that an invocation cannot be reordered before the preceding responses.
3. The concurrent execution yields the same behavior as the 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* (the intra-thread order). This ordering does not need to be consistent with the order that the operations were actually issued in.

Line-Up [5], Paraglider [13], and VYRD [7] leverage linearizability to test concurrent data structures. **Unfortunately, efficient implementations of many common data structures, e.g., RCU [6], MS Queue [11], etc., for the C/C++ memory model are neither linearizable nor sequentially consistent! Thus previous tools cannot check such data structures under the C/C++ memory model.**

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

The C/C++ memory model brings the following two key challenges that prevent the application of previous approaches to specifying the concurrent data structures to this setting:

1. **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 implement updates and the operations that read those updates. These data structures often do not guarantee that different threads observe updates in the same order — in other words the data structures may expose clients to weaker consistency models than sequential consistency. For example, even when one uses the relatively strong `acquire` and `release` memory orderings in C++, it is possible for two different threads to observe two stores happening in different orders, i.e., executions can fail the IRIW test. 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 include a total order over all memory operations, thus even further complicating the application of traditional approaches to correctness, e.g., linearization cannot be applied. In particular the 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 of the dynamic tools [12, 14] for exploring the behavior of code under relaxed models do as a practical matter print out an execution in some order, this order is to some degree arbitrary as relaxed memory models generally make it possible for a data structure operation to see the effects of operations that appear later in any such an order (e.g., a load can read from a store that appears later in the order). Instead of a total order, the C/C++ memory model is formulated as a graph of memory operations with several partial orders defined in this graph.
2. **Constraining Reorderings (Specifying Synchronization Properties):** Synchronization[2] in C/C++ provides an ordering between memory operations to different locations. Concurrent data structures must establish synchronization or they potentially expose their users to highly non-intuitive behavior that is likely to break client code. For example, consider the case of a concurrent queue that does not establish synchronization between enqueue and dequeue operations. Consider the following sequence of operations: (1) thread A initializes the fields of a new object X; (2) thread A enqueues the reference to X in such a queue (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 could reorder the initializing writes to be executed after the enqueue operation. If the fields are non-atomic, such loads are considered data races and violate the data race free requirement of the C/C++ language standard and thus the program has no semantics.

   The C/C++ memory model formalizes synchronization in terms of a *happens-before* relation. The C/C++ 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 $y$.

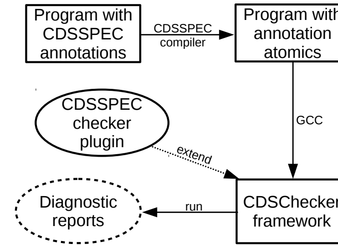


**Figure 1: CDSSPEC system overview**

## 1.3 Specification Language and Tool Support

Figure 1 presents an overview of the CDSSPEC system. 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 a program 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 specific unit test and generates diagnostic reports for any executions that violate the specification.

## 1.4 Contributions

This paper makes the following contributions:

- **Specification Language:** It introduces a specification language that enables developers to write specifications of concurrent data structures developed for a relaxed memory model in a simple fashion that capture the key correctness properties. Our specification language is the first to our knowledge that supports concurrent data structures that use C/C++ atomic operations.
- **A Technique to Relate Concurrent Executions to Sequential Executions:** It presents an approach to order the memory oper-

[1]It is important to note that the term sequential consistency in the literature is applied to both the consistency model that data structures expose clients to as well as the guarantees that the underlying memory system provides for load and store operations.

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

ations for the C/C++ model, which lacks a definition of a trace and for which one generally cannot even construct a total order of atomic operations that is consistent with the program order. The generated sequential execution by necessity does not always maintain program order.

- **Synchronization Properties:** It presents (a) constructs for specifying the happens before relations that a data structure should establish, and (b) tool support for checking these properties and exposing synchronization related bugs.
- **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 our tool can detect bugs, and that our tool can unit test real world data structures with reasonable performance.

# 2. FORMALIZATION OF CORRECTNESS MODEL

Unlike the SC memory model, finding an appropriate correctness model for concurrent data structures under the C/C++11 memory model is challenging. For example, linearizability no longer fits C/C++ by the fact that the C/C++ memory model allows atomic loads to read from atomic stores that appear later in the trace and that it is in general impossible to produce a sequential history that preserves program order for the C/C++ memory model.

Consider the following example:

```
Thrd 1:            Thrd 2:
x = 1;             y = 1;
r1 = y;            r2 = x;
```

Suppose each operation in this example is the only operation of each method call, and shared variables `x` and `y` are both initilly `0`. Each store operation has `release` semantics and each load operation has `acquire` semantics. For the execution where both `r1` and `r2` obtain the old value `0`, we encounter a challenge of generating a sequential history. Since neither load operation reads from the corresponding store, they should be ordered before their corresponding store operation. On the other hand, both stores happen before the other load (intra-thread ordering), making it impossible to topologically sort the operations to generate a consistent sequential history.

Intuitively however, we can weaken some constraints, e.g. the *happens-before* edges in some cases, to generate a reordering of ordering points such that they yield a sequential history that is consistent with the specification. We formalize our approach as follow. The goal is to come up with a correctness model that is weaker than linearizability and SC and yet is composable.

First of all, we represent a trace as an *execution graph*, where each node represents each API method call with a set of ordering points, and edges between nodes are derived from the *happens-before* edges and the *modification order* edges between ordering points. We define *opo* as the *ordering point order* relation between ordering point. Given two operations $X$ and $Y$ that are both ordering points, the *modification order* edges are as follow:

1. **Modification Order (write-write):** $X \xrightarrow{mo} Y \Rightarrow X \xrightarrow{opo} Y$.
2. **Modification Order (read-write):** $A \xrightarrow{mo} Y \ \wedge\ A \xrightarrow{rf} X \Rightarrow X \xrightarrow{opo} Y$.
3. **Modification Order (write-read):** $X \xrightarrow{mo} B \ \wedge\ B \xrightarrow{rf} Y \Rightarrow X \xrightarrow{opo} Y$.
4. **Modification Order (read-read):** $A \xrightarrow{mo} B \ \wedge\ A \xrightarrow{rf} X \ \wedge\ B \xrightarrow{rf} Y \Rightarrow X \xrightarrow{opo} Y$.

Intuitively, if method $A$ has an information flow to method $B$, we want method $B$ to see the effects before method $A$. In C/C++11, on the other hand, we want method $A$ to have `release` semantics while method $B$ to have `acquire` semantics so that they establish the happens-before relationship. For example, for a concurrent queue, we want a dequeuer synchronizes with the corresponding enqueuer so that when the dequeuer obtains a reference to an object, it can read the fully initialized value of that object. To some degree this can also avoid data races. When it comes to C/C++11 data structures, the ordering points of method calls should have release/acquire semantics on stores and loads.

In order to relax the contraints on the original execution graph, we will have to disregard some happens-before edges. To make our model intuitive, we want to keep the happens-before edges from stores to stores and from load operations to store operations because that can ensure information is only flowing from earlier store operations. Besides, we also want to keep the happens-before edges between operations on the same memory location since otherwise the generated history will become very counter-intuitive. However, such a model does not work in C/C++ in general. Consider the following example:

Consider the following example:

```
Thrd 1:     Thrd 2:     Thrd 3:     Thrd 4:
x = 1;      y = 2;      r1 = x;     r3 = y;
y = 1;      x = 2;      r2 = x;     r4 = y;
```

We encounter a challenge for the execution where the following holds: `r1==2&&r2==1&&r3==1&&r4==2`. For any total order that is consistent with the happens-before relation, that total order will be inconsistent with the modification order in either variable `x` or `y`. For example, if `y = 1` is ordered before `y = 2`, then `x = 1` is ordered before `x = 2` while `r1 = x` is ordered before `r2 = x`. As a result, we cannot possibly move up the second load operation `r2 = x` across `r1 = x` to generate a consistent sequential history. To solve this, one option is that allow loads to move up any operation including load and store operation on the same location. However, it is extremely conuter-intuitive for a later load operation to read an older value. By analysis, the problem here is that the store operations with different values to the same memory location are not ordered. This is actually a really rare case because it could be possible that only one store operation takes effect and all other store operations are useless. We believe that such case of blind stores from different threads is not the general pattern for concurrent data structures, and we believe that store operations are ordered. In terms of ordering points, we believe that in real-world data structures, store operations of ordering points are ordered by happens-before relation. **[[argue why ordered stores are reasonable to concurrent data structures]]**

We next define an action called *tranform* that can be performed on the graph as follow:

$\forall X, Y, X \in \mathit{OrderingPoints} \wedge Y \in \mathit{OrderingPoints} \wedge address(X) \neq address(Y) \wedge X \xrightarrow{hb} Y \wedge Y \in \mathit{LoadOps} \Rightarrow \forall Z, Z \in \mathit{orderingPoints} \wedge Z \xrightarrow{hb} X, Z \xrightarrow{hb} Y \wedge \neg X \xrightarrow{hb} Y$.

Under these analysis and assumptions, we clearly define our correctness model as follow:

**[[prove that our correctness model is composable]]**

# 3. REFERENCES

```
1:  function InferParams(testcases, initialParams)
2:      inputParams := initialParams
3:      if inputParams is empty then
4:          inputParams := the weakest parameters
5:      end if
6:      for all test case t in testcases do
7:          candidates := inputParams
8:          results := { }
9:          while candidates is not empty do
10:             Candidate c := pop from candidates
11:             run CDSChecker with c and check SC
12:             if ∃ SC violation v then
13:                 StrengthenParam(v, c, candidates)
14:             else
15:                 results += WeakenOrderParams(c)
16:             end if
17:         end while
18:         inputParams := results
19:     end for
20:     return results
21: end function
22: procedure StrengthenParam(v, c, candidates)
23:     while ∃ a fix f for violation v do
24:         possible_repairs := strengthen c with fix f
25:         candidates += possible_repairs
26:     end while
27: end procedure
```

**Figure 2: Algorithm for inferring order parameters**


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

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

[3] 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.

[4] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[5] 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.

[6] 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*, 2011.

[7] 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.

[8] 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.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[10] 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`.

[11] 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.

[12] 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.

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

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