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 [18, 28]. 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 [26].

The C/C++ standard committee extended the C and C++ languages with support for low-level atomic operations in the C/C++11 standard [2, 3, 8] 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 [24] 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 [30], CPPMEM [6], and Relacy [33]. 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) [22].

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-

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 [10], Paraglider [32], and VYRD [20] leverage linearizability to test concurrent data structures. **Unfortunately, efficient implementations of many common data structures, e.g., RCU [18], MS Queue [28], 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 [30, 33] 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.

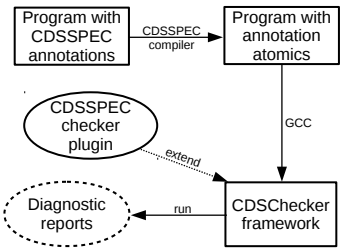

Figure 1: CDSSPEC system overview

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. C/C++ MEMORY MODEL

We next briefly summarize the key aspects of the C/C++ memory model. The memory model describes a set of atomic operations and the corresponding allowed behaviors of programs that utilize them. A more detailed formal treatment of the memory model [6] and a more detailed informal description [2, 3] are available in the literature. Any operation on an atomic object will have one of six *memory orders*, each of which falls into one or more of the following categories.

relaxed: `memory_order_relaxed` – weakest ordering

release: `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a store-release may form release/consume or release/acquire synchronization

consume:[3] `memory_order_consume` – a load-consume may form release/consume synchronization

acquire: `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a load-acquire may form release/acquire synchronization

seq-cst: `memory_order_seq_cst` – strongest ordering

2.1 Relations

The C/C++ memory model expresses program behavior in the form of binary relations or orderings. The following subsections will briefly summarize the relevant relations. Much of this discussion resembles the preferred model from the formalization in [6].

Sequenced-Before: The order of program operations within a single thread of execution establishes an intra-thread *sequenced-before* (*sb*) relation. Note that while some operations in C/C++ provide no intra-thread ordering—the equality operator (==), for example—we ignore this detail and assume that *sb* totally orders all operations in a thread.

Reads-From: The *reads-from* (*rf*) relation consists of store-load pairs (X, Y) such that Y reads its value from the effect of X—or $X \xrightarrow{rf} Y$. In the C/C++ memory model, this relation is non-trivial, as a given load operation may read from one of many potential stores in the program execution.

Synchronizes-With: The *synchronizes-with* (*sw*) relation captures synchronization that occurs when certain atomic operations interact across two threads. For instance, release/acquire synchronization occurs between a pair of atomic operations on the same object: a store-release X and a load-acquire Y. If Y reads from X, then X synchronizes with Y—or $X \xrightarrow{sw} Y$.

Happens-Before: In CDSSPEC, we avoid consume operations, and so the *happens-before* (*hb*) relation is simply the transitive closure of *sb* and *sw*.

The *hb* relation restricts the stores that loads can read from. For example, if we have two stores X and Y and a load Z to the same memory location and $X \xrightarrow{hb} Y \xrightarrow{hb} Z$, then Z cannot read from X.

Sequential Consistency: All seq-cst operations in a program execution form a total ordering (*sc*) so that, for instance, a seq-cst load may not read from a seq-cst store prior to the most recent store (to the same location) in the *sc* ordering, nor from any store that happens before that store. The *sc* order must be consistent with *hb*.

Modification Order: Each atomic object in a program execution has an associated *modification order* (*mo*)—a total order of all stores to that object—which informally represents a memory-coherent ordering in which those stores may be observed by the rest of the program. In general the modification orders for all objects cannot be combined to form a consistent total ordering.

3. MOTIVATING EXAMPLE

We will use the read-copy update (RCU) [18] implementation in Figure 2 as a running example. RCU data structures are useful for usage scenarios with large numbers of reads and relatively few updates. RCU based data structures scale to large numbers of readers because read operations do not contain any low-level store operations and thus do not generate coherence traffic to invalidate cache lines. Lines 3 through 33 contain the code for the RCU implementation, and Lines 35 through 44 show a test case for the implementation.

The RCU implementation maintains a shared pointer (Line 9) `node` to reference the data shared by readers and updaters. A reader loads the shared pointer (Line 18) to read the shared data, while an updater allocates a new struct (Line 24), loads the shared pointer (Line 25), and then attempts a compare and swap (CAS) operation to update the shared pointer. This process repeats until the CAS is successful.

This implementation uses low-level atomic operations provided by C/C++ to implement the RCU mechanism. For example, in Line 29, the `update` method uses a CAS operation to update the `node`. The memory order parameters attached to the atomic operations ensure that memory accesses that were performed before the `update` operation must occur before memory accesses that were performed after the `read` operation.

3.1 Consistency Model

In the example code, two threads, Thread 1 and Thread 2, each update one of the RCU objects `x` and `y`, respectively, and then access the object updated by the other thread. Under the C/C++ memory model, this example admits an execution in which both Thread 1 and Thread 2 read the initial value 0.

Figure 3 shows a possible order for this execution in which the model checker processes statements in the order shown in the execution labeled Original Trace[4], which is an interleaving of the in-

[3]Consume is not broadly supported by compilers due to challenges associate with preserving data dependencies and is unlikely to provide significant performance gains on x86 hardware. We take the same approach as many compilers and treat consumes as acquires.

[4]Note that the C/C++ memory model does not include any notion of a trace. We merely use the term trace to refer to the order in

```
#define mo_acquire memory_order_acquire
#define mo_acq_rel memory_order_acq_rel
struct Node {
   int data ;
   int version;
};

class RCU {
   atomic<Node*> node;
   public:
   RCU() {
      Node *n = new Node;
      n->data= 0;
      n->version = 0;
      atomic_init(&node, n);
   }
   void read(int *data, int *version) {
      Node *res = node.load(mo_acquire);
      *data = res->data;
      *version = res->version;
   }
   void update(int data) {
      bool succ = false;
      Node *newNode = new Node;
      Node *prev = node.load(mo_acquire);
      do {
         newNode->data = data;
         newNode->version = prev->version + 1;
         succ = node.compare_exchange_strong(prev,
            newNode, mo_acq_rel, mo_acquire);
      } while (!succ);
   }
};

RCU x, y; // Define two RCU objects
int r1, r2, v1, v2;
void thrd_1() { // Thread 1
   x.update(1);
   y.read(&r1, &v1); // r1 == 0
}
void thrd_2() { // Thread 2
   y.update(1);
   x.read(&r2, &v2); // r2 == 0
}
```

Figure 2: C++11 read-copy update example

vocation and response events of API methods indicated by "_i"and "_r", respectively.

This trace is rather counter-intuitive because it orders Thread 1's `update` event before Thread 2's `read` event, yet Thread 2's `read` event does not observe the updated value. However, under the C/C++11 memory model, this is allowed since a load is allowed to read a value that is written by an old store.

It is important to note that it is impossible to produce a sequential history for this example that preserves program order and produces the observed behavior. This means that traditional notions of correctness that relate concurrent executions to sequential executions that were developed for the SC memory model cannot directly be applied to the C/C++11 memory model.

This example leads to the following conclusion — if we want to define the behavior of concurrent executions by relating them to equivalent sequential histories, then we must give up on preserving program order.

We note that for this example, if we give up on program order, we produce the sequential history in Figure 3 labeled Reordered Trace. The data structure operations in this sequential history have behavior that is consistent with the observed behaviors in the example execution.

3.2 Synchronization Properties

A correct RCU data structure ensures the property that the invocation event of an `update` operation happens-before the response event of the subsequent `read` operation. Informally, all operations that appear before the `update` operation must appear to occur before the operations that appear after the `read` operation. While these semantics follow naturally for SC, for relaxed memory models we must ensure that neither the compiler nor the processor performs reorderings that violate these semantics. Therefore, when a reader reads the fields of the `node`, data races are eliminated since the `read` operation synchronizes with the `update` operation.

In the C/C++11 memory model, the transitive closure of *sequence-before* and *synchronizes-with* relations form the *happens-before* relation in the absence of operations with the `consume` memory order. Since every time an `update` operation finishes, a CAS operation to `node` with release semantics is executed, and the subsequent `read` loads the `node` reference with acquire semantics, the `read` operation synchronizes with the `update` operation, providing the necessary synchronization properties. Some RCU implementations may use consume semantics instead of acquire semantics, however we do not use consume for the following two reasons: (1) most existing C/C++ compilers do not support consume semantics and thus acquire does not involve extra overhead and (2) using consume semantics provides a more complex interface to clients that may require them to explicitly declare dependences throughout the code.

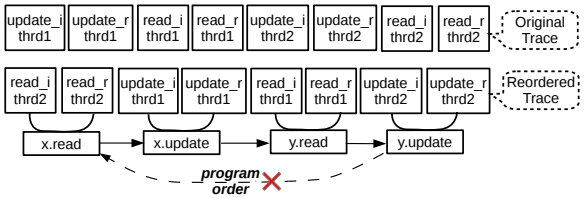

Figure 3: History of a possible execution of two RCU objects

4. SPECIFICATION LANGUAGE DESIGN

We begin by overviewing CDSSPEC's basic approach. The CDSSPEC specification language specifies the correctness properties for concurrent data structures by establishing a correspondence with an equivalent sequential data structure, which requires: (1) defining the equivalent sequential data structure; (2) establishing an equivalent sequential history; (3) relating the behavior of the concurrent execution to the sequential history; and (4) specifying the synchronization properties between method invocations. The specification language has the following key components:

1. Equivalent Sequential Data Structure: This component defines a sequential data structure against which the concurrent data structure will be checked.

2. Defining the Equivalent Sequential History: For real-world data structures, generally there exist two types of API methods, *primitive* API methods and *aggregate* API methods. Take concurrent hashtables 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 calls primitive API methods internally. From our experience working with our benchmark set, it is generally possible to generate a sequential history of primitive API method invocations for real-world data structures as they generally serialize on a single memory location — while it is possible to observe partially completed aggregate API method invocations. Therefore, our specifications will focus on the correctness of primitive API methods.

Borrowing from VYRD [20] the concept of commit points, we

which CDSSPEC prints the execution.

allow developers to specify *ordering points* — the specific atomic operations between method invocation and response events that are used for ordering method calls. Ordering points are a generalization of the commit points as they also use memory operations that do not commit the data structure operation to order method calls. Developers then specify ordering points to generate the equivalent sequential history.

3. Specifying Correct Behaviors: Developers use the CDSSPEC specification to provide a set of *side effects* and *assertions* to describe the desired behavior of each API method. Side effects capture how a method updates the equivalent sequential data structure. Assertions include both preconditions and postconditions which specify what conditions each method must satisfy before and after the method call, respectively. The specification of the side effects and assertions may make use of the states of the equivalent sequential data structure, meaning that these components 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.

4. Synchronization: The synchronization specification describes the desired happens before relations between API method invocations. CDSSPEC specifications allow developers to specify the properties at the abstraction of methods. This makes specifications cleaner, more understandable, and less error-prone because the properties do not rely on low-level implementation details. Moreover, CDSSPEC specifications allow developers to attach conditions to methods when specifying synchronization properties so that a method call might synchronize with another only under a specific condition. For example, consider a spin lock with a `try_lock()` method, we need to specify that only a successful `try_lock()` method invocation must synchronize with the previous `unlock()` method invocation.

Figure 4 presents the grammar for the CDSSPEC specification language. The grammar defines three types of specification annotations: *structure annotations*, *method annotations*, and *ordering point annotations*. In the grammar, *code* means legal C/C++ source code; and *label* means a legal C/C++ variable name. Annotations are embedded in C/C++ comments. This format does not affect the semantics of the original program and allows for the same source to be used by both a standard C/C++ compiler to generate production code and for the CDSSPEC specification compiler to extract the CDSSPEC specification from the comments. We discuss these constructs in more detail throughout the remainder of this section.

structureSpec	→	`"@Structure_Define"` *structureDefine* (*happensBefore*)?
structureDefine	→	(`"@DeclareStruct:"` *code*)* `"@DeclareVar:"` *code* `"@InitVar:"` *code* (`"@DefineFunc:"` *code*)*
happensBefore	→	`"@Happens_Before:"` (*invocation* `"->"` *invocation*)$^+$
invocation	→	*label* (`"("` *label*`")"`)?
methodSpec	→	`"@Method:"` *label* `"@Ordering_Point_Set:"` *label* (`"|"` *label*)* (`"@HB_Condition:"` *label* `"::"` *code*)* (`"@ID:"` *code*)? (`"@PreCondition:"` *code*)? (`"@SideEffect:"` *code*)? (`"@PostCondition:"` *code*)?
potentialOP	→	`"@Potential_Ordering_Point:"` *code* `"@Label:"` *label*
opCheck	→	`"@Ordering_Point_Check:"` *code* `"@Potential_Ordering_Point_Label:"` *label* `"@Label:"` *label*
opLabelCheck	→	`"@Ordering_Point_Label_Check:"` *code* `"@Label:"` *label*
opClear	→	`"@Ordering_Point_Clear:"` *code* `"@Label:"` *label*
code	→	`<Legal C/C++ code>`

Figure 4: Grammar for CDSSPEC specification language

4.1 Example

To make the CDSSPEC specification language more concrete, Figure 5 presents the annotated RCU example. We use ordering points to order the method invocations to construct the equivalent sequential history for the concurrent execution. We first specify the ordering points for the `read` and `update` methods to define the equivalent sequential history. For the `read` method, the load access of the `node` field in Line 20 is the only operation that can be an ordering point. For the `update` method, there exists more than one atomic operation. However, they only serve as an ordering point when it successfully uses the CAS operation to update the `node` field. Thus, we specify in Line 39 that the CAS operation should be the ordering point for the update operation when `succ` is `true`. We associate the methods with the two ordering points in Lines 15 and 27.

Next, we specify the equivalent sequential data structure for this RCU implementation. Line 3 declares two integer fields `_data` and `_version` as the internal states of the equivalent sequential RCU, and Line 4 initializes both variables to `0`.

We then specify the correct behaviors of the equivalent sequential history by defining the side effects and assertions for the `read` and `update` methods. Side effects specify how to perform the corresponding operation on the equivalent sequential data structure. For example, Line 29 specifies the side effects of the `update` to the sequential states. Assertions specify properties that should be true of the concurrent data structure execution. For example, Line 17 specifies that the `read` method should satisfy the postcondition that the returned `data` and `version` fields should have consistent values as the internal states of the equivalent sequential data structure.

Our implementation of the RCU data structure is designed to establish synchronization between the `update` method invocation and the later `read` or `update` method invocation. This is important, for example, if a client thread were to update an array index and use the RCU data structure to communicate the updated index to a second client thread. Without synchronization, the second client thread may not see the update to the array index. Besides, the synchronization between two `update` calls ensures that a later `read` will see the most updated values. In order to specify the synchronization, we associate `read` and `update` methods with method call identifiers (Lines 16 and 28), which for this example is the value of the `this` pointer. Together these annotations ensure that every API method call on the same RCU object will have the same method call ID. Line 5 then specifies the synchronization properties for the RCU — any `update` method invocation should synchronize with all later `read` and `update` method invocations that have the same method call identifier as that `update` method. This guarantees that `update` calls should synchronize with the `read` and `update` calls of the same RCU object.

4.2 Defining the Equivalent Sequential History

```
1  class RCU {
2     /** @Structure_Define:
3        @DeclareVar:  int _data, _version;
4        @InitVar:  _data = 0; _version = 0;
5     @Happens_Before: Update->Read  Update->Update*/
6     atomic<Node*> node;
7     public:
8     RCU() {
9        Node *n = new Node;
10       n->data = 0;
11       n->version = 0;
12       atomic_init(&node, n);
13    }
14    /** @Interface: Read
15    @Ordering_Point_Set: Read_Point
16    @ID: this
17    @PostCondition:
18       _data == *data && _version == *version */
19    void read(int *data, int *version) {
20       Node *res = node.load(mo_acquire);
21       /** @Ordering_Point_Label_Check: true
22       @Label: Read_Point */
23       *data = res->data;
24       *version = res->version;
25    }
26    /** @Interface: Update
27    @Ordering_Point_Set: Update_Point
28    @ID: this
29    @SideEffect: _data = data; _version++; */
30    void update(int data) {
31       bool succ = false;
32       Node *newNode = new Node;
33       Node *prev = node.load(mo_acquire);
34       do {
35          newNode->data = data;
36          newNode->version = prev->version + 1;
37          succ = node.compare_exchange_strong(prev,
38             newNode, mo_acq_rel, mo_acquire);
39          /** @Ordering_Point_Label_Check: succ
40          @Label: Update_Point */
41       } while (!succ);
42    }
43 };
```

Figure 5: Annotated RCU specification example

While defining the equivalent sequential history for concurrent executions is well studied in the context of the SC memory model, optimizations that developers typically use in the context of weaker memory models create the following new challenges when we generate the equivalent sequential history using ordering points.

- **Absence of a Meaningful Trace or Total Order:** For the SC memory model, an execution can be represented by a simple interleaving of all the memory operations, where each load operation reads from the last store operation to the same location in the trace. However, under a relaxed memory model like C/C++11, the interleaving does not uniquely define an execution because a given load operation can potentially read from many different store operations in the interleaving. Therefore, we have to rely on the intrinsic relations between ordering points such as *reads from* and *modification order* to order method calls.
- **Lack of Program Order Preserving Sequential History:** Moreover, as discussed in Section 3.1, in general it is not possible to arrange an execution in any totally ordered sequential history that preserves program order.

  A key insight is that many concurrent data structures' API methods have a commit point, which is a single memory operation that makes the update visible to other threads and that also serves as an ordering point. When two data structure operations have a dependence, it is often the case that their respective commit points are both conflicting accesses to the same memory location. In this case, the modification order provided by C/C++ is sufficient to order these operations since modification order is guaranteed to be consistent with the happens before relation (and therefore also the sequenced before relation).

  For cases where the method calls are independent, such as a `put(X, 1)` followed by a `get(Y)` in a hashtable where `X` and `Y` are different keys, the lack of an ordering is not a problem since those methods commute.

Ordering Points Annotations: In many 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 a real ordering point. Therefore, the CDSSPEC specification separates the definition of ordering points as follows:

1. `Potential_Ordering_Point` annotation: The labeling of ordering points that identifies the location of a potential ordering point.
2. `Ordering_Point_Check` annotation: The checking of ordering points that checks at a later point whether a potential ordering point was really an ordering point.

These two constructs together identify ordering points. 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 `Potential_Ordering_Point` annotation with a condition `ConditionA` and a label `LabelA`, and then use the label `LabelA` in an `Ordering_Point_Check` annotation at a later point.

The `Ordering_Point_Label_Check` annotation combines the `Potential_Ordering_Point` and the `Ordering_Point_Check` annotations, and makes specifications simpler for the use case where the ordering point is known immediately. For example, in Line 21 of Figure 5, we use the `Ordering_Point_Label_Check` annotation to identify the ordering point for `read` because we know the load operation in Line 20 is an ordering point at the time it is executed.

Some data structure operations may require multiple ordering points. For example, consider a transaction implementation that first attempts to lock all of the involved objects (dropping the locks and retrying if it fails to acquire a lock), performs the updates, and then releases the locks. To order such transactions in a relaxed memory model, we must consider all of the locks it acquires rather than just the last lock. Thus, we allow a method invocation to have more than one ordering point, and the additional ordering points serve to order the operation with respect to multiple different memory locations. For the transaction example, it may be necessary to retry the acquisition of locks. To support this scenario, the `Ordering_Point_Clear` annotation removes all previous ordering points when it satisfies a specific condition.

Moreover, when an API method calls another API method, they can share ordering points. In that case, CDSSPEC requires that at that ordering point, the concurrent data structure should satisfy the precondition and postcondition of both API methods.

4.3 Checking the IO Behavior of Methods

With the specified ordering points, CDSSPEC is able to generate the equivalent sequential history. Developers then need to define the equivalent sequential data structure. For example, in Line 3 and 4 of the annotated RCU example, we use the `Structure_define` annotation to define the equivalent sequential RCU by specifying the internal states as two integers, `_version` and `_data`. In the `Structure_define` annotation, developers can also specify definitions for customized structs

and methods for convenience. We design these annotations in such a way that developers can write specifications in C/C++ code such that they do not have to learn a new specification language.

After defining the internal states and methods of the equivalent data structure, developers use the `SideEffect` annotation to define the corresponding sequential API methods, which should contain the action to be performed on the equivalent sequential data structure. For example, in Line 29 of the annotated RCU example, we use `SideEffect` to specify that when we execute the `update` method on the equivalent sequential RCU, we should update the internal states of `_version` and `_data` accordingly. When the `SideEffect` annotation is omitted for a specific API method, it means that no side effects will happen on the sequential data structure when that method is called. Take the annotated RCU as an example, the `read` has no side effects on the equivalent sequential RCU.

With the well-defined equivalent sequential data structure, developers then relate the generated equivalent sequential history to the equivalent sequential data structure. In CDSSPEC, we allow developers to accomplish this by using the `PreCondition` and `PostCondition` annotations to specify the conditions to be checked before and after the API method appears to happen. For example, Line 18 in the annotated RCU example means that when `read` appears to happen, it should return the same value as the current internal variables of the equivalent sequential RCU. Note that these two annotations contain legitimate C/C++ expressions that only access the method call parameters, return value and the internal states of the equivalent sequential data structure.

4.4 Checking Synchronization

Under a relaxed memory model, compilers and processors can reorder memory operations and thus the execution can exhibit counter-intuitive behaviors. The C/C++11 memory model provides developers with memory ordering that establish synchronization, e.g., `acquire`, `release`, `seq_cst`. Synchronization serves to control which reorderings are allowed — however, restricting reorderings comes at a runtime cost so developers must balance complexity against runtime overhead. Checking that data structures establish proper synchronization is important to ensure that the data structures can be effectively composed with client code.

We generalize the notion of happens before to methods as follows. Method call c_1 happens-before method call c_2 if the invocation event of c_1 happens before the response event of c_2. Note that by this definition two method calls can both happen before each other — an example of this is the `barrier` synchronization construct. With this notion, for example, for a correctly synchronized queue, we want an enqueue to happen before the corresponding dequeue, which avoids the synchronization problems discussed earlier in Section 1.2.

In order to flexibly express the synchronization between methods, we associate API methods with method call identifiers (or IDs) and happens-before conditional guard expressions. The method call ID is a C/C++ expression that computes a unique ID for the call, and if it is omitted, a default value is used. For example, in our RCU example in Figure 5, both the `update` and `read` methods have the `this` pointer of the corresponding RCU object. The `HB_Condition` component associates one happens-before conditional guard expression with a unique label. For one method, multiple conditional guard expressions are allowed to be defined, and the conditional guard expression can only access the method instance's argument values and return value.

After specifying the method call IDs and the `HB_Condition` labels, developers can specify the synchronization as "`method1(HB_condition1) -> method2(HB_condition2)`". When the `HB_condition` is omitted, it defaults to `true`. The semantics of this expression is that all instances of calls to `method1` that satisfy the conditional guard expression `HB_condition1` should happen-before all later instances (as determined by ordering points) of calls to `method2` that satisfy the conditional guard expression `HB_condition2` such that both instances shared the same ID. The ID and happens-before conditional guard expression are important because they allow developers to impose synchronization only between specific method invocations under specific conditions. For example, in Figure 5, Line 5 specifies two synchronization rules, which together mean that the `update` should only establish synchronization with later `read` and `update` from the same RCU object under all circumstances.

5. IMPLEMENTATION

The goal of the CDSSPEC specification language is to enable developers to write specifications against which concurrent data structures can be tested. We can ensure a concurrent data structure is correct with respect to an equivalent sequential data structure if for each execution of the concurrent data structure, the equivalent sequential history for the equivalent sequential data structure yields the same results.

The execution space for many concurrent data structures is unbounded, meaning that in practice we cannot verify correctness by checking individual executions. However, the specifications can be used for unit testing. In practice, many bugs can be exposed by model checking unit tests for concurrent data structures. We have implemented the CDSSPEC checker as a unit testing tool built upon the CDSCHECKER framework. CDSSPEC can exhaustively explore all behaviors for unit tests and provide developers with diagnostic reports for executions that violate their specification.

5.1 Model Checker Framework

The CDSSPEC checker takes as input a complete execution from the CDSCHECKER model checker. The CDSCHECKER framework operates at the abstraction level of individual atomic operations and thus has neither information about method calls nor which atomic operations serve as ordering points. Thus, we extend the framework by adding *annotation* operations to CDSCHECKER's traces, which record the necessary information to check the specifications but have no effect on other operations. The CDSSPEC compiler inserts code to generate the annotation actions to communicate to the CDSSPEC checker plugin the critical events for checking the CDSSPEC specification. These annotation actions then appear in CDSCHECKER's list of atomic operations and make it convenient for CDSSPEC to construct a sequential history from the execution because for any given method call, its invocation event, its ordering points, and its response event are sequentially ordered in the list.

5.2 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 check the specification assertions. We next describe the type of annotation actions that the CDSSPEC compiler inserts into the instrumented program.

Ordering Points: Ordering points have a conditional guard expression and a label. Potential ordering point annotation actions are inserted immediately after the atomic operation that serves as the potential ordering point. Ordering point check annotation actions are inserted where they appear.

Method Boundary: To identify a method's boundaries, CDSSPEC inserts *method_begin* and *method_end* annotations at the beginning and end of methods.

Sequential States and Methods: Since checking occurs after CDSCHECKER has completed an execution, the annotation actions stores the values of any variables in the concurrent data structure that the annotations reference.

Side Effects and Assertions: Side effects and assertions perform their checks after an execution. The side effects and assertions are compiled into methods and the equivalent sequential data structure's states are accessible to these methods. With this encapsulation, the CDSSPEC checker simply calls these functions to implement the side effects and assertions.

Synchronization Checks: The CDSSPEC checker performs synchronization checks in two parts: compiling the rules and runtime data collection. First, the CDSSPEC compiler numbers all methods and happens-before checks uniquely. For example, the rule "`Update->Read`" can be represented as (1, 0, 2, 0), which means instances of method 1 that satisfy condition 0 should *synchronize with* instances method 2 that satisfy condition 0. In this case, condition 0 means `true`. Then, the CDSSPEC compiler generates code that communicates the synchronization rules by passing an array of integer pairs. Runtime collection is then implemented by performing the condition check at each method invocation or response and then passing the method number and happens before condition if the check is satisfied.

5.3 Dynamic Checking

At this point, we have an execution trace with the necessary annotations to construct a sequential history and to check the execution's correctness. However, before constructing the sequential history, the CDSSPEC plugin first collects the necessary information for each method call, which is the *method_begin* annotation, the ordering point annotations, the happens-before checks, and the *method_end* annotations. Since all of the operations in the trace have thread identifiers it is straightforward to extract the operations between the *method_begin* and *method_end* annotations.

Reorder Method Calls: As discussed above, determining the ordering of the ordering points is non-trivial under the C/C++ memory model. This can be complicated 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.

However, we can still leverage the reads-from relation and the modification-order relation to order the ordering points that appear in typical data structures. CDSSPEC uses the following rules to generate an ordering-point ordering *opo* relation on ordering points. Given two operations X and Y that are both ordering points:

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

Generating the Reordering: The CDSSPEC checker first builds an execution graph where the nodes are method calls and the edges represent the *opo* ordering of the ordering points of the methods that correspond to the source and destination nodes. Assuming the absence of cycles in the execution graph, the *opo* ordering is used to generate the sequential history. The CDSSPEC checker topologically sorts the graph to generate the equivalent sequential execution.

When CDSSPEC fails to order two ordering points, the operations often commute. Thus, if multiple histories satisfy the constraints of *opo*, by default we generally randomly select one. However, when those operations do not commute, we provide developers with different options: (1) they can add additional ordering points to order the two nodes or (2) they can run CDSSPEC in either of the following modes: (a) *loosely exhaustive* mode — CDSSPEC explores all possible histories and only requires that there exists some history that passes the checks or (b) *strictly exhaustive* mode — CDSSPEC explores all possible histories and requires all histories pass the checks.

Synchronization Checks: Synchronization properties are specified using the IDs and conditions of method calls, and we have that information available after CDSSPEC constructs the sequential history and checks the preconditions and postconditions. For two specific method calls c_1 and c_2, we can ensure c_1 synchronizes with c_2 by ensuring the annotation `c1_begin` happens-before the annotation `c2_end` because any operations sequenced-before `c1_begin` should happen-before any operations sequenced-after `c2_end` according to the C/C++11 memory model.

6. EVALUATION

We have implemented CDSSPEC. Our evaluation focuses on the following questions: (1) How expressive is CDSSPEC for specifying the correctness properties of real-world concurrent data structures? (2) How easy is it to use CDSSPEC? (3) What is the performance of CDSSPEC? (4) How effective was CDSSPEC in finding bugs?

In order to evaluate CDSSPEC, we have gathered a contention free lock, two types of concurrent queues, and a work stealing deque [25]. As C/C++11 is relatively new there are no C/C++11 implementations for many concurrent data structures, thus we ported several data structures. The Linux kernel's reader-writer spinlock and the Michael Scott queue were originally ported for the CDSCHECKER benchmark suite. We also ported an RCU implementation and Cliff Click's hashtable from its Java implementation [15]. We report execution times on an Intel Core i7 3770.

6.1 Expressiveness

In this section, we evaluate the expressiveness of CDSSPEC by reporting our experiences writing specifications for a range of concurrent data structures.

Lockfree hashtable: We ported Cliff Click's hashtable, which supports simultaneous lookups and updates by multiple threads as well as concurrent table resizing. The implementation uses an array of atomic variables to store the key/value slots, and uses acquire/release synchronization to establish the synchronization between hashtable accesses.

Hashtable updates consist of two CAS operations — one to claim the key slot and one to update the value. When a `put` method invocation successfully updates both the key and value, the update is visible to other threads. Thus, both CAS operations are ordering points for the `put` method, and we annotate both of them as potential ordering points. The `get` method is ordered after an invocation of the `put` only if it sees both the key and value updates. Thus we annotate an ordering point for the key read only if the key is null. We also annotate an ordering point for the value read if it reads the value slot. The test driver has two threads both of which update and read the value for the same key.

RCU: As discussed in the example, this is a synchronization mechanism used in the Linux kernel that allows concurrent reads and updates. We ran this benchmark with four threads, two update the data structure and two read the data structure.

Chase-Lev Deque: This is a bug-fixed version of a published C11 adaptation of the Chase-Lev deque [25]. It maintains a top and bottom index to a shared array of references. In terms of synchronization, when pushing an item into the sequential deque, we attach a unique ID tag to that element. When stealing or taking an item, we use that tag as the ID of the method call. Thus, we have (push, steal) or (push, take) pairs that have the same method call ID. In our test driver, one thread pushes 3 items and takes 2 items while the other one steals 1 item.

Linux Reader-Writer Lock: A reader-writer lock allows either multiple concurrent readers or one exclusive writer. We can abstract it with a boolean `writer_lock` representing whether the writer lock is held and an integer `reader_cnt` representing the number of threads that are reading. We test this benchmark with a single lock that protects shared variables. We have two threads that read and write the shared variables under the protection of a read lock and a write lock.

MCS Lock: This benchmark is an implementation of the Mellor-Crummey and Scott lock [27, 1]. This lock queues waiting threads in a FIFO. Our test driver utilizes two threads that read and write shared variables with the protection of the lock.

M&S Queue: This benchmark is an adaptation of the Michael and Scott lock free queue [28] to the C/C++ memory model. We ran with two threads, one of which enqueues and the other of which dequeues an item.

SPSC Queue: This is a lock-free single-producer, single-consumer queue. We used a test driver that has two threads — one enqueues a value and the other dequeues it.

MPMC Queue: This is a multiple-producer, multiple-consumer queue. Producers call `write_prepare` to obtain a free slot, update the slot, and call `write_publish` to publish it. Consumers call `read_fetch` to obtain a valid slot, read the slot, and call `read_consume` to free it. The specification focuses on the synchronization properties which require `write_publish` to synchronize with `read_fetch` to ensure the data integrity and `read_consume` to synchronize with `write_prepare` to ensure that slots are not prematurely recycled. The test driver contains two threads, each of which enqueue and dequeue an item.

6.2 Ease of Use

In addition to expressiveness, it is also important for specification languages to be easy to use. In our experience using CDSSPEC to specify the real-world data structures in our benchmark set, we found that CDSSPEC was easy to use. CDSSPEC specifications have only three parts — equivalent sequential data structures, ordering points, and synchronization properties, and we explain the reasons as follows. (1) Specifying sequential data structures is easy and straightforward, and developers can often just use an off-the-shelf implementation from a library. (2) When developers specify ordering points, they only need to know what operations order methods without needing to specify the subtle reasoning about the corner cases involving interleavings and reorderings introduced by relaxed memory models. Take the Michael & Scott queue as an example, we can easily order enqueuers with the point when `enqueue` loads the `tail` pointer right before inserting the new node. (3) For synchronization, the fact that we allow specifying synchronization at the abstraction of methods makes it easy. For example, we only used 36 lines to specify synchronization in a total 1,033 lines of code (omitting blanks and comments).

Benchmark	# Executions	# Feasible	Total Time (s)
Chase-Lev Deque	1,365	232	0.15
SPSC Queue	19	15	0.01
RCU	1269	756	0.11
Lockfree Hashtable	30,941	25,731	11.39
MCS Lock	19,501	13,546	2.62
MPMC Queue	170,220	93,224	45.63
M&S Queue	168	114	0.05
Linux RW Lock	148,053	405	13.06

Figure 6: Benchmark results

6.3 Performance

Figure 6 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 complete. All of our benchmarks complete within one minute and most take less than 3 seconds to complete.

6.4 Finding Bugs

Benchmark	# Injection	# DR	# UL	# Correctness	# Sync	Rate
Chase-Lev Deque	10	0	2	3	2	70%
SPSC Queue	2	2	0	0	0	100%
RCU	3	0	0	1	2	100%
Lockfree Hashtable	5	0	0	0	2	40%
MCS Lock	4	0	0	0	4	100%
MPMC Queue	6	0	0	0	2	33%
M&S Queue	11	0	6	3	0	82%
Linux RW Lock	8	0	0	0	8	100%
Total	49	2	8	7	20	76%

Figure 7: Bug injection detection results

The next component of the evaluation examines the effectiveness of CDSSPEC for finding bugs.

New Bugs: In the M&S queue benchmark used in [30], the dequeue interface does not differentiate between dequeuing the integer zero and returning that no item is available, and it passed our initial specification. However, after modifying the dequeue interface to match that in the original paper, CDSSPEC is able to find a new bug that CDSCHECKER did not find. The original test driver for this benchmark performed the enqueues first to make it easy to write assertions that are valid for all executions. CDSSPEC allows specification assertions to capture the behavior of the specific execution and thus is able to discover the given bug.

Injected Bugs: To further evaluate CDSSPEC, we injected bugs in our benchmarks by weakening the ordering parameter of the atomic operations. These include changing `release`, `acquire`, `acq_rel` and `seq_cst` 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 that are caused by misunderstanding the complicated semantics of relaxed memory models.

This fault injection strategy will introduce one of two types of bugs. The first type is a specification-independent bug, which can be detected by the underlying CDSCHECKER infrastructure which includes internal data races and uninitialized loads. The second type is a specification-dependent bug, which passes the built-in checks but violates the CDSSPEC specification. These include failed assertions and synchronization violations. We classifying bugs as follows. If CDSCHECKER reports a data race or an uninitialized load, CDSSPEC reports the error and stops. If not, CDSSPEC continues to check the execution against the specification. It first checks for violations of the preconditions and postconditions and then for violations of the synchronization specification.

Figure 7 shows the results of the injection detection. The column *DR* represents data races, *UL* represents uninitialized loads,

correctness represents a failed precondition or postcondition, and *sync* represents a synchronization violation. The detection rate is the number of injections for which we detected a bug divided by the total number of injections.

Linux Reader-Writer Lock: Our initial specification for this benchmark did not allow `write_trylock` to spuriously fail. However, when we checked this benchmark against that specification, CDSSPEC checker reports a correctness violation. We then analyzed the code and found that `write_trylock` first subtracts a bias from the `lock` variable to attempt to acquire the lock, and restores that bias if the attempt to acquire the lock fails. In the scenario where two `write_trylock` are racing for the lock before the lock is released, one `write_trylock` can first decrement the lock variable, the lock can be released by the original holder, and then the second `write_trylock` can attempt to acquire the lock. Even though the history indicates that the lock is unlocked, it still holds a transient value due to the partially completed first `write_trylock` invocation. Thus, the second `write_trylock` invocation will also fail. As the second `write_trylock` serializes after both the first unsuccessful `write_trylock` and the `unlock` operation, 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 their data structures. By analyzing the CDSSPEC diagnostic report, developers can better understand any inconsistencies between the specification and the implementation.

MCS Lock: Three of the weakened operations are not detected because they cause the execution to fail to terminate (and hit a trace bound). We reviewed the code and found that weakening any of those three operations makes the lock spin forever.

M&S Queue: Our test driver does not cause an enqueue or dequeue thread to help another enqueue thread update the tail pointer, which corresponds to two of the undetected injections.

Lockfree Hashtable: Our experiment only focuses on the two primitive methods, `get` and `put` without triggering the `resize`. We were able to successfully check all executions from a test driver for the lockfree hash table that generates executions that have no program order preserving sequential histories.

MPMC Queue: The undetected injections in this benchmark are primarily due to the limitation of our test driver. One synchronization property of this benchmark is that `read_consume` should synchronize with the next `write_prepare` to ensure that a slot cannot be reused before the consumer has finished with the slot. Our test driver is unable to reach this case so those injections are not detected.

From our experiments on concurrent data structures, we can see that CDSSPEC checker can help detect incorrect memory orderings, help developers refine data structure specifications, and help determine whether strong memory orderings are really necessary. Since CDSSPEC checker is a unit testing tool, it is limited to small-scale tests to explore common usage scenarios of the data structures. As a unit testing tool, CDSSPEC was able to find 100% of injections for many data structures and to find 76% of the injections on average. For our 49 injections, 10 of them were detected by checks in CDSCHECKER, and 27 additional injections were detected by CDSSPEC. This shows that by writing specifications, we detect significantly more fault injections.

7. RELATED WORK

Researchers have proposed and designed specifications and approaches to find bugs in concurrent data structures based on linearization. Early work by Wing and Gong [34] proposed using linearizability to test and verify concurrent objects. Line-up [10] builds on the Chess [29] model checker to automatically check deterministic linearizability. It automatically generates the sequential specification by systematically enumerating all sequential behaviors. Paraglider [32] supports checking with and without linearization points based on SPIN [23]. All of these approaches assume that there exist a sequential history that is consistent with program order. Furthermore, they also 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*. [5] present a static analysis based on TVLA for verifying linearizability of concurrent linked data structures. Valeiadis [31] demonstrates a shape-value abstraction which can automatically prove linearizability. Thread quantification can also verify data structure linearizability [7]. Colvin *et al*. formally verified a list-based set [16]. While these approaches provide stronger guarantees than CDSSPEC, they were typically used to check simpler data structures and require experts to use. Moreover, they target the SC memory model.

Researchers have proposed specification languages for concurrent data structures. Refinement mapping [4] provides the theoretical basis for designing and using specifications. Commit atomicity [21] can verify atomicity properties. Concurrit [19] is a domain-specific language that allows programmers to write scripts to specify thread schedules to reproduce bugs, and is useful when programmers already have some knowledge about a bug. NDetermin [12] infers nondeterministic sequential specifications to model the behaviors of parallel code.

VYRD [20] is conceptually similar to CDSSPEC— developers specify commit points for concurrent code. The parallel code is then executed and the commit points are used to identify a sequential execution that should have the same behaviors. VYRD was designed for the SC memory model — it is unable to construct a sequential refinement for a relaxed memory model or check synchronization properties.

GAMBIT [17] uses a prioritized search technique that combines stateless model checking and heuristic-guided fuzzing to unit test code under the SC memory model. RELAXED [14] 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 [9] is a tool for verifying data structures against relaxed memory models and takes a SAT-based approach instead of the stateless model checking approach used by CDSCHECKER.

Researchers have developed verification techniques for code that admits only SC executions under the TSO and PSO memory models [13, 11]. The basic idea is to develop an execution monitor that can detect whether non-SC executions exist by examining only SC executions.

8. CONCLUSION

The CDSSPEC specification language and checking system 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 range of data structures including a lock-free hashtable, work-stealing deque, queues and locks.

9. REFERENCES

[1] http://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html. Oct. 2012.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] C. Click. A lock-free hash table. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf, May 2007.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] 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.

[27] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[28] 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.

[29] 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.

[30] 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.

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

[32] M. Vechev, E. Yahav, and G. Yorsh. Experience with model

checking linearizability. In *International SPIN Workshop on Model Checking Software*, 2009.

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

[34] 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.