UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
CDSSPEC: Testing Concurrent Data Structures Under the C/C++11 Memory Model

Permalink
https://escholarship.org/uc/item/93s3701k

Author
Ou, Peizhao

Publication Date
2014

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,  
IRVINE

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

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE
in Electrical Engineering and Computer Science

by

Peizhao Ou

Thesis Committee:
Professor Brian Demsky, Chair
Professor Rainer Doemer
Professor Harry Xu

2014
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT OF THE THESIS</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.0.1 Previous Work on Linearizability</td>
<td>2</td>
</tr>
<tr>
<td>1.0.2 New Challenges from the C/C++11 Memory Model</td>
<td>4</td>
</tr>
<tr>
<td>1.0.3 Specification Language and Tool Support</td>
<td>5</td>
</tr>
<tr>
<td>1.0.4 Contributions</td>
<td>6</td>
</tr>
<tr>
<td>2 Motivating Example</td>
<td>8</td>
</tr>
<tr>
<td>2.0.5 Trace Reordering</td>
<td>9</td>
</tr>
<tr>
<td>2.0.6 Synchronization Properties</td>
<td>11</td>
</tr>
<tr>
<td>3 Specification Language Design</td>
<td>12</td>
</tr>
<tr>
<td>3.0.7 Overview</td>
<td>14</td>
</tr>
<tr>
<td>3.0.8 Language Constructs</td>
<td>15</td>
</tr>
<tr>
<td>3.0.9 Example</td>
<td>18</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>22</td>
</tr>
<tr>
<td>4.0.10 Model Checking Framework</td>
<td>23</td>
</tr>
<tr>
<td>4.0.11 Specification Compiler</td>
<td>24</td>
</tr>
<tr>
<td>4.0.12 Dynamic Checking</td>
<td>25</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>27</td>
</tr>
<tr>
<td>5.0.13 Expressiveness</td>
<td>28</td>
</tr>
<tr>
<td>5.0.14 Performance</td>
<td>30</td>
</tr>
<tr>
<td>5.0.15 Finding Bugs</td>
<td>30</td>
</tr>
<tr>
<td>6 Related Work</td>
<td>35</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>Bibliography</td>
<td>38</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CDSSpec System Overview</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>C++11 Spin Lock Example</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>History of a Possible Execution of Lock</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Grammar for CDSSpec Specification Language</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Annotated lock specification example</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Benchmark Results</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Bug Injection Detection Results</td>
<td>30</td>
</tr>
</tbody>
</table>
I would like to thank all those who offered help and made contribution to the improvement of this work. I particularly would like to thank the members of my thesis committee, Professor Rainer Doemer, Professor Harry Xu and Professor Brian Demsky. Most importantly, I would like to thank my advisor, Professor Brian Demsky, who has been exceptionally supportive and an indispensable part of this project. His guidance have promoted my progress through every step of the way, and I have learned a lot working with him, including not only the necessary body of knowledge but also the insights and important research skills that will help me do better research in the future.

This project was partly supported by a Google Research Award and the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786.
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. We have evaluated CDSSpec by annotating and checking several concurrent data structures.
Chapter 1

Introduction

Careful 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 [15, 18, 27]. 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 [15].

Traditionally, developers had to target their implementations of such data structures to a specific compiler and processor, using intimate knowledge of the platform and typically coding parts in assembly. 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 allow developers to write portable implementations of concurrent data structures. To support the relaxations typically performed by compilers and processors, the C/C++11 memory model provides weaker semantics than sequential consistency [24] and as a result,
correctly using these operations is extremely 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++ model including CDSChecker [29], CPPMEM [6], and Relacy [32]. These tools explore behaviors that are allowed under the C/C++ memory model. Using these tools for testing can be challenging as different interleavings or reorderings often legitimately produce different behaviors, and it can be burdensome to write code to check the output of a test case under a given interleaving or reordering.

1.0.1 Previous Work on Linearizability

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? One 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 [22].

We view concurrent data structures as concurrent objects that have internal state and methods that can be simultaneously called by multiple threads to read or update the state. Each method call has two related events, a method invocation followed by a method response. An execution history is a total order of method invocations and responses.

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 can in many cases just leverage existing well tested implementations in the STL library.

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 structure.

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.

For a linearizable object, a *linearization point* for a method call is the point at which the method call appears to take effect. For lock-based data structures, a linearization point generally can be any point in the critical section. For lock-free data structures, the linearization point depends on the details of the algorithm and may not be apparent until later in the execution.

Researchers have developed several tools including Line-Up [10], Paraglider [31], and VYRD [20] that leverage notions of linearizability to test concurrent data structures. Essentially the idea is for each execution trace of the concurrent algorithm, find a sequence of linearization points that is both (1) consistent with the concurrent execution trace and (2) for which a sequential execution in the same order generates the same behavior. These tools generally take one of two approaches to this problem: they either require a specification for the linearization points or they simply search for a suitable order.
Approaches based on linearization implicitly assume the sequential consistency memory model — they assume that given a set of actions that comprise an execution in which some actions have been declared to be linearization points (or invocation/return events), the trace totally orders the actions.

### 1.0.2 New Challenges from the C/C++11 Memory Model

**No Total Order:** Like many other relaxed memory models, the C/C++11 memory model does not include a notion of a total order over all memory operations, thus thwarting the application of traditional approaches to correctness, e.g., linearization cannot be directly 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 [29, 32] 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). The C/C++ memory model instead is formulated as a graph of memory operations with several partial orders defined in this graph.

While there are serious difficulties in directly applying existing correctness models that relate concurrent data structure executions to sequential executions, we wish to maintain the intuitive appeal and simplicity of describing the behavior of concurrent data structures in terms of sequential executions.

**Specifying Synchronization Properties:** Synchronization\(^1\) in C/C++ provides an or-

---

\(^1\)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.
dering between memory operations to different locations. Concurrent data structures must establish synchronization or they potentially expose their users to highly non-intuitive behavior. For example, consider the case of a concurrent queue that does not establish synchronization between enqueue and dequeue operations. If one thread initializes the fields of an object and then enqueues a reference to that object in such a queue and a second thread dequeues the reference and uses it to read the fields of that object, the second thread could fail to see the initializing writes. 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++11 memory model formalizes synchronization in terms of a happens-before relation. The C/C++11 happens-before relationship is a partial order between memory accesses. If memory access \( x \) happens before memory access \( y \), it means all the effects of \( x \) must be ordered before the effects of \( y \).

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. For example, for a correctly synchronized queue, we want an enqueue to happen before the corresponding dequeue. This avoids the synchronization problems discussed earlier in this section.

1.0.3 Specification Language and Tool Support

Figure 1.1 presents an overview of the CDSSpec system. After implementing a concurrent data structure, developers then 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 all possible behaviors of the specific unit test and generates diagnostic reports for any executions that violate the specification.

1.0.4 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 captures the key correctness properties. Our specification language is the first to our knowledge that supports concurrent data structures that use C/C++11 atomic operations.

- **A Technique to Produce an Equivalent Sequential History Without Traces:** It presents an approach to order the memory operations for the C/C++11 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 intra-thread ordering such
that all loads read from some previous store in the order.

• **Synchronization Properties:** It presents constructs for specifying the happens be-
fore relations that a data structure should establish and tool support for checking these
properties and exposing synchronization related bugs.

• **Evaluation:** It shows that the CDSSpec specification language can express key cor-
rectness 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.
Chapter 2

Motivating Example

We will use the spinlock in Figure 2.1 as a running example. We begin by using it to explore the challenges of specifying correctness properties of concurrent data structures under the C/C++11 memory model. Lines 1 through 13 contain the code for the lock implementation, and Lines 14 through 27 show an example use of the lock. This lock provides mutual exclusion for a protected region beginning with lock and ending with unlock. The example uses an atomic shared boolean variable flag to indicate whether the lock has already been acquired. The lock method sets the flag to true to acquire the lock, and the unlock method resets the flag to release the lock. The implementation also includes a try_lock method which tries to acquire the lock once.

This implementation uses low-level atomic operations to implement the lock. In Line 3, the try_lock method uses a CAS operation to set the flag, and it returns whether the operation succeeds. The lock method simply repeatedly calls try_lock in a loop until it succeeds. In Line 12, the unlock method uses an atomic store operation to reset the flag. The memory order semantics attached to the atomic operations is used to impose the necessary synchronization between threads (Line 4 and Line 12).
```c++
1 atomic<bool> flag(false); // Shared variables
2 bool try_lock() {
3     bool succ = flag.compare_exchange_strong(false,  
4         true, memory_order_acquire,  
5         memory_order_relaxed);  
6     return succ;
7 }
8 void lock() {  
9     while(!try_lock());
10 }
11 void unlock() {  
12     flag.store(false, memory_order_release);
13 }
14 int x = y = 0;
15 void thrd_1() { // Thread 1
16     lock();
17     x = 1;
18     y = 1;
19     unlock();
20 }
21 void thrd_2() { // Thread 2
22     if (try_lock()) {
23         int r1 = y;
24         int r2 = x;
25         unlock();
26     }
27 }
```

Figure 2.1: C++11 Spin Lock Example

2.0.5 Trace Reordering

Consider an execution in which two threads, Thread 1 and Thread 2, update and read the shared variables `x` and `y` by calling the methods `thrd_1` and `thrd_2`, respectively. The `thrd_1` method acquires the lock, stores the value 1 in `x` and `y`, and then releases the lock. The `thrd_2` method tries to acquire the lock by calling `try_lock`, and if it succeeds, it reads `y` and `x` and then releases the lock. Figure 2.2 shows one possible trace\(^1\), which is an interleaving of the invocation and response events in the lock method indicated by "i" and "r", respectively. The trace is rather counter-intuitive because `try_lock` fails even though it finishes before

\(^1\)We use the term trace loosely, to refer to the order in which CDSSPEC prints the execution.
Figure 2.2: History of a Possible Execution of Lock

the first lock invocation in the trace\(^2\). However, under the C/C++11 memory model, this is allowed since a load is allowed to read a value that is written by the later store. In this trace, the CAS operation in the try_lock reads the true value written by the later CAS operation in the lock call. This means that traditional notions of correctness that relate concurrent executions to sequential executions that were developed for the SC memory model are non-trivial to apply to the C/C++11 memory model. For instance, linearizability would imply that the concurrent execution’s behavior should match the Reordered Trace 1 from Figure 2.2. But the Reordered Trace 1 is not correctly reordered because the try_lock method would not fail if it logically happens before the lock. The Reordered Trace 2, however, is consistent with the observed behavior of the concurrent execution.

\(^2\)The trylock in our example is not intended to match the spurious failure semantics of C/C++11’s mutex. Instead, it serves to illustrate how we address the challenges that arise from a relaxed memory model.
2.0.6 Synchronization Properties

A correct lock data structure ensures the property that the invocation event of an `unlock` synchronizes with the response event of the subsequent `lock`. More concretely, all operations that are ordered after a lock operation must not be reordered to occur before the lock acquisition and all operations that are ordered before an unlock operation must not be reordered to occur after the unlock operation (roach-motel ordering semantics guarantee that operations protected by a lock cannot be reordered out of the critical section). 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, if the `try_lock` in Thread 2 succeeds, the program must only observe either \( r_1=r_2=0 \) or \( r_1=r_2=1 \), but it should not observe \( r_1=1 \land r_2=0 \) or \( r_1=0 \land r_2=1 \). According to the C/C++ memory model, the transitive closure of `sequence-before` and `synchronize-with` forms the `happens-before` relation in the absence of operations with the consume memory ordering. Since every time an `unlock` finishes, a store to `flag` with release semantics is executed, and the subsequent `lock` must read the `flag` with acquire semantics when it executes a successful CAS operation, the store operation synchronizes with the CAS operation, guaranteeing the necessary synchronization properties.
Chapter 3

Specification Language Design

Concurrent data structures are well known to be more challenging to reason about and to test than their sequential counterparts, and the reorderings allowed by the C/C++ relaxed memory model introduce a new level of complexity beyond simple concurrency. Sequential data structures can be tested based on the input/output behavior of calls to their API, and linearizability is a common criteria for mapping a concurrent execution to an equivalent sequential execution. However, in the C/C++11 setting, low-level atomic operations allow counterintuitive reorderings, making such a mapping non-trivial. Therefore, the design of the CDSSpec specification language must address the following aspects in its design:

Absence of a Meaningful 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. Thus relaxed memory models like C/C++11 often introduce a reads-from relationship to map loads to the stores that they read from.
Method Preconditions and Postconditions: A traditional technique for specifying sequential data structures is to write preconditions and postconditions for API methods based on the internal state of the data structure. Applying this approach to concurrent data structures can be complicated and impractical because the internal states may be simultaneously changed by another thread during a data structure operation. Linearizability provides a nicer abstraction for specifying concurrent data structures by mapping concurrent method calls to an equivalent sequential sequence of method calls. Adapting linearizability to C/C++11 data structures brings two challenges: (1) how do we specify the corresponding sequential data structure’s behavior and (2) how do we specify the points at which the concurrent method calls appear to happen instantly? The second challenge is more subtle in C/C++11 as there is no intrinsic order for ordering linearization points. Borrowing the convention used by VYRD [20], we define the points at which concurrent API calls appear to take effect as commit points rather than linearization points because the order of those points is not necessarily equal to the ordering in which they appear in some arbitrary trace order generated by the model checker.

Synchronization: Under a relaxed memory model, memory accesses can happen in counterintuitive ways, thus making synchronization as important an issue as the specification of preconditions and postconditions.

Fundamentally, synchronization properties capture composability properties of data structure operations. Consider a concurrent queue storing pointers to objects. The precondition and postcondition of the specification only checks that the pointers that it enqueues and dequeues are consistent. However, we must also ensure that when dequeue returns a pointer, the dequeuing thread sees all of the updates to the object that were performed by the enqueuing thread before it made the enqueue call. Synchronization properties can be expressed in terms of method calls that must synchronize with each other.

Commit point ordering: The specification identifies certain atomic operations between
the invocation and response event of a method call as commit points. As we discussed above, in order to generate a sequential history for the execution trace, we must order the commit points. For example, in Figure 2.1, the execution trace contains a failed \texttt{try.lock} call followed by a \texttt{lock} call and an \texttt{unlock} call. To obtain a consistent trace, recall that we need an ordering in which the \texttt{lock} call occurs first, the failed \texttt{try.lock} call next, and finally the \texttt{unlock} call. Ordering the commit points requires additional information from the execution.

### 3.0.7 Overview

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) identifying the commit points for each concurrent method invocation; (3) defining preconditions and postconditions for each API method; and (4) specifying the synchronization properties between method invocations. The CDSSpec 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. **Side Effects and Assertions**: Developers use the CDSSpec specification to provide a set of \textit{side effects} and \textit{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 state of the equivalent sequential data structure, meaning that these components can access the states and the parameters and return values of
the method calls 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.

3. Commit Points: Each method has a set of commit points, which are the specific atomic operations at which the method invocation may appear to take effect. The CDSSpec checking infrastructure uses annotations to identify the commit points.

Some internal methods may be called by multiple API methods. In this case, the same atomic operation can be identified as a potential commit point for multiple API methods, and each API method later has a check to verify whether it was a real commit point. Therefore, the CDSSpec specification separates the labeling and the checking of commit points.

4. Synchronization: The synchronization specification describes the desired happens before relations between the methods of data structures. 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 instance, an unlock must synchronize with a later successful try_lock but not with a failed try_lock.

3.0.8 Language Constructs

Figure 3.1 presents the grammar for the CDSSpec specification language. The grammar defines three types of specification annotations: structure annotations, method annotations, and commit point annotations. 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 next discuss the constructs in more detail below:

**Structure Annotations:** The `Structure_define` annotation defines the equivalent sequential data structure (both its state and methods) and the concurrent data structure's synchronization properties. Developers can also include definitions for customized structs and methods. Developers can specify the equivalent sequential data structure in C/C++ such that they do not have to learn a new specification language. In the grammar in Figure 3.1, `code` means legal C/C++ source code.

We use method names and conditional guard expressions to specify the actual synchronization that must be established by the concurrent data structure. In order to flexibly express the synchronization between methods, we allow method calls to have identifiers (or IDs) and happens-before conditional guard expressions, which only rely on the method arguments and the method’s return value. The ID is important because it allows developers to impose synchronization between specific method invocations. For example, for a concurrent queue, an `enqueue` call only synchronizes with a `dequeue` call when the enqueued element and the dequeued element are the same. The happens-before conditional guard expression is also important since synchronization can be conditional in some cases (e.g., `try_lock` only establishes happens-before if it is successful). The semantics of `method1(HB_condition1) -> method2(HB_condition2)` is that all instances of calls to `method1` that satisfy the conditional guard expression `HB_condition1` happen-before the next instance of a call to `method2` that satisfies the conditional guard expression `HB_condition2` such that both instances shared the same ID. A `methodCluster` is a cluster of conditional method labels that share the same synchronization properties.

**Method Annotations:** This construct specifies a method’s commit points, synchronization, preconditions, postconditions, and side effects. Developers define a set of commit point
labels for each method that identify the method’s commit points. The HB\_Condition component contains the happens-before conditional guard expression for the method, and the ID is a C/C++ expression that computes a unique ID for the call. Multiple conditional guard expressions may be defined, and the conditional guard expression can access the method instance’s parameter values and return value. If the unique ID is omitted, a default value is used. The PreCondition annotation specifies a condition to be checked before the method appears to happen, and the SideEffect annotation specifies the corresponding action to be performed on the equivalent sequential data structure. Similarly, PostSideEffect and PostCondition are the action to be performed and condition to be checked after the method appears to happen.

**Commit Point Annotations:** In many cases, it is not possible to determine whether a given atomic operation is a commit point until later in the execution. Thus, we separate the specification of commit points into two parts: (1) the Potential\_Commit\_Point annotation identifies the location of a potential commit point, and (2) the Commit\_Point\_Check annotation checks at a later point whether a potential commit point was really a commit point.

These two constructs together identify commit points. For example, assume that $A$ is an atomic operation that is marked as a potential commit point with the label LabelA under the condition ConditionA. The developer would write a Potential\_Commit\_Point annotation with the label LabelA and then use the label LabelA in a Commit\_Point\_Check annotation at a later point.

The Commit\_Point\_Label\_Check annotation combines the Potential\_Commit\_Point and the Commit\_Point\_Check annotations, and makes specifications simpler for the use case that the commit point is known immediately. For example, a CAS operation may be a commit point whenever it succeeds, and we know instantly whether it succeeds. In this case, we can use the Commit\_Point\_Label\_Check annotation to both define a commit point and to perform
the check.

Some data structures 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 release the locks. To order such transactions in a relaxed memory model, we must consider all of the locks it acquires and not just the last lock. Thus, we allow a method invocation to have more than one commit point, and the additional commit point 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 Commit_Point_Clear annotation removes all previous commit points when it satisfies a specific condition.

### 3.0.9 Example

To make the CDSSpec specification language more concrete, Figure 3.2 presents the annotated lock example. The example begins with a Structure_Define annotation. Line 2 declares the boolean variable locked as the internal state of the equivalent sequential lock, indicating whether the lock is acquired or not. Line 3 initializes the locked to be false, indicating the lock is initially unlocked.

Next, we define the side effects and assertions for all of the lock’s API methods. Each method’s specification appears immediately before the method’s definition. For example, Line 6 labels the method try_lock as Try_Lock, and associates two commit points, Lock_Fail_Point and Lock_Succ_Point, with the method. Line 9 declares the happens-before condition Trylock_Succ to be true when the method try_lock returns the value true. Line 10 associates try_lock with the DEFAULT method call identifier. Note that the method specifications for all three methods have DEFAULT method call ID, which means that any successful try_lock method invocation should synchronize with all unlock method invo-
Figure 3.1: Grammar for CDSSpec Specification Language

cations ordered before the `try_lock` invocation. Line 12 specifies that the `try_lock` method should satisfy the precondition that it returns the value `true` or `false` according to whether the `locked` variable is `false` or `true`. Line 13 defines the side effects of the `try_lock` method on the equivalent sequential data structure — if the concurrent execution returns `true`, its sequential counterpart should switch the `locked` flag to `true`. The `lock` and `unlock` method
specifications are similar to \texttt{try\_lock}'s specification. For example, Line 29 specifies that before setting the \texttt{locked} variable to \texttt{true}, the precondition that the \texttt{locked} variable is \texttt{false} should be satisfied.

The \texttt{Structure\_Define} annotation in Line 4 specifies the synchronization property for the lock — any \texttt{unlock} method invocation should synchronize with the next successful \texttt{lock} method invocation. Invocations of the \texttt{try\_lock} method can succeed or fail when they try to acquire the lock, and a synchronization requirement should only be imposed on successful \texttt{try\_lock} method invocations. The annotation \texttt{@HB\_Condition} for \texttt{try\_lock} defines \texttt{Trylock\_Succ} to be true when \texttt{try\_lock} returns true. Since every \texttt{lock} must include a successful \texttt{try\_lock}, we only need to specify the synchronization property between an \texttt{unlock} invocation and the next successful \texttt{try\_lock} invocation.

The specification identifies three commit points in the source code. For the \texttt{lock} and \texttt{try\_lock} methods, the \texttt{compare\_exchange\_strong} operation in Line 15 is the only operation that can be a commit point. Line 18 labels the commit point for a successful lock operation when the CAS operation succeeds. Similarly, Line 21 defines the commit point for a failed \texttt{try\_lock} operation. Note that the commit point \texttt{Lock\_Succ\_Point} is shared between the \texttt{try\_lock} and \texttt{lock} method because the \texttt{lock} method calls the \texttt{try\_lock} method to acquire the lock. For the \texttt{unlock} method, Line 41 identifies a commit point with a single annotation as this operation always commits the enclosing \texttt{unlock} method invocation.

When an API method calls another API method, they can share commit points. For example, since \texttt{lock} calls \texttt{try\_lock} until it acquires the lock, we will have a sequence of failed \texttt{try\_lock} followed by a successful \texttt{try\_lock} in the \texttt{lock} method call, and the commit point in Line 18 is the commit point for the successful \texttt{try\_lock} and the outer \texttt{lock} call. \texttt{CDSSpec} requires that at that commit point, the concurrent data structure should satisfy the precondition and postcondition of both the inner and outer API methods.
/** @Structure_Define:
  @DeclareVar: bool locked;
  @InitVar: locked = false;
  @Happens_Before: Unlock→Try_Lock(Trylock_Succ)*/

atomic <bool> flag(false); // Shared variables

/** @Method: Try_Lock
  @Commit_Point_Set:
  Lock_Fail_Point | Lock_Succ_Point
  @HB_Condition: Trylock_Succ :: __RET__
  @ID: DEFAULT
  @PreCondition:
  (! locked && __RET__ ) || ( locked && ! __RET__ )
  @SideEffect: if ( __RET__ ) locked = true ; */

bool try_lock () {
  bool succ = flag.compare_exchange_strong(false,
    true, memory_order_acquire,
    memory_order_relaxed);
  /** @Commit_Point_Label_Check: succ
    @Label: Lock_Succ_Point
    @End */
  /** @Commit_Point_Label_Check: ! succ
    @Label: Lock_Fail_Point
    @End */
  return succ;
}

/** @Method: Lock
  @Commit_Point_Set: Lock_Succ_Point
  @ID: DEFAULT
  @PreCondition: ! locked
  @SideEffect: locked = true ; */

void lock () {
  while (! try_lock () ) ;
}

/** @Method: Unlock
  @Commit_Point_Set: Unlock_Point
  @ID: DEFAULT
  @PreCondition: locked
  @SideEffect: locked = false ; */

void unlock () {
  flag.store(false, memory_order_release);
  /** @Commit_Point_Label_Check: true
    @Label: Unlock_Point */
}

Figure 3.2: Annotated lock specification example
Chapter 4

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. The CDSSpec checker can exhaustively explore all behaviors for unit tests and provide developers with diagnostic reports for executions that violate their specification.
4.0.10 Model Checker Framework

The CDSSpec checker takes as input a complete execution trace from the CDSChecker model checker. The C/C++11 memory model formalizes an execution as a graph of atomic operations along with the happens-before (hb), reads-from (rf), and modification-order (mo) relations for the execution. We use the notation $X \xrightarrow{hb} Y$ to indicate that operation $X$ happens-before $Y$. The reads-from relation is a set of pairs of store and load operations such that $(X, Y) \in rf$ means that $Y$ reads the value from the effect of $X$, denoted as $X \xrightarrow{rf} Y$. The modification-order relation is a total order of stores to a given memory location\(^1\). We denote $A \xrightarrow{mo} B$ to mean that the store $A$ to location $X$ is modification ordered before the store $B$ to location $X$.

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 commit 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 commit points, and its response event are sequentially ordered in the list.

\(^1\)Note that although stores to a given memory location have a total order, the total orders for different memory locations cannot in general be composed to yield a single total order.
4.0.11 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 check assertions and construct the sequential history. We next describe the type of annotation actions that the CDSSPEC compiler inserts into the instrumented program.

Commit Points: Commit points have a conditional guard expression and a label. Potential commit point annotation actions are inserted immediately after the atomic operation that serves as the potential commit point. Commit 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 state is 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 Unlock -> Try_Lock(Try_Lock_Succ) can be represented as (1, 0, 2, 3), which means instances of method 1 that satisfy condition 0 should synchronize with instances method 2 that satisfy condition 3. Then, the CDSSPEC compiler generates code that communicates the synchro-
nization 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.

4.0.12 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 commit 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 linearization order of the commit points is non-trivial under the C/C++ memory model. Determining the order is complicated by the fact that the C/C++ memory model allows atomic loads to read from atomic stores that appear later in the trace.

However, we can still leverage the reads-from relation, the happens-before relation, and the modification-order relation to order the commit points that appear in typical data structures. CDSSpec uses the following three rules to generate a commit point ordering cpo relation on commit points.

1. **Reads-From:** $X \xrightarrow{rf} Y \Rightarrow X \xrightarrow{cpo} Y$ if operations $X$ and $Y$ are both commit points.

2. **Modification Order:** $X \xrightarrow{mo} Y \Rightarrow X \xrightarrow{cpo} Y$ if operations $X$ and $Y$ are both commit points.
3. **Happens-Before**: The happens-before ($hb$) relation is the transitive closure of sequence-before ($sb$) and synchronizes with ($sw$) in the absence of operations with the consume memory ordering. $X \xrightarrow{hb} Y \Rightarrow X \xrightarrow{cpo} Y$ if operations $X$ and $Y$ are both commit points.

**Generating the Reordering**: The CDSSpec checker first builds an execution graph where the nodes are method calls and the edges represent the $cpo$ ordering of the commit points of the methods that correspond to the source and destination nodes. Assuming the absence of cycles in the execution graph, the $cpo$ ordering is used to generate the sequential history. If two methods are not ordered by the $cpo$ order, we assume that they commute and select an arbitrary ordering for the methods. The CDSSpec checker topologically sorts the graph to generate the equivalent sequential execution.

When CDSSpec fails to order commit points, the operations often commute. Thus, if multiple histories satisfy the constraints of $cpo$, we simply pick one. When those operations do not commute, we assume developers will add additional ordering points to allow CDSSpec to order the two nodes.

**Check Synchronization Properties**: Synchronization properties are specified using the IDs and conditions of method calls, and we have that information ready 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 $c_1_{\text{begin}}$ happens-before the annotation $c_2_{\text{end}}$ because any operations sequenced-before $c_1_{\text{begin}}$ should happen-before any operations sequenced-after $c_2_{\text{end}}$ according to the C/C++11 memory model.
Chapter 5

Evaluation

We have implemented CDSSpec. Our evaluation focuses on the following questions: (1) How expressive is CDSSpec for specifying correctness properties for real concurrent data structures? (2) What is the performance of CDSSpec? (3) 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.
5.0.13  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 commit points for the `put` method, and we annotate both of them as potential commit points. The `get` method commits after an invocation of the `put` only if it sees both the key and value updates. Thus we annotate a commit point for the key read only if the key is null. We also annotate a commit 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.

**Read Copy Update:** Read-copy update (RCU) is a synchronization mechanism in the Linux kernel. It allows concurrent reads with updates in a non-blocking fashion. This benchmark maintains an atomic reference to an immutable shared object. The `read` and `write` methods linearize at the point where they successfully read and update the shared pointer, respectively. Invocations of the `write` method should synchronize with the next successful `write` or `read` method invocation. 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 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.

**Contention-Free Lock:** This benchmark is an implementation of the algorithm designed by Mellor-Crummey and Scott [26, 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 [27] 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 synchro-
nize 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.

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

Figure 5.1: Benchmark Results

5.0.14 Performance

Figure 5.1 presents performance results for CDSSpec on our benchmark set. We list the number of the total executions CDSChecker has explored, the number of the feasible executions we checked 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.

5.0.15 Finding Bugs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Injection</th>
<th># DR</th>
<th># UL</th>
<th># Correctness</th>
<th># Sync</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chase-Lev Deque</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>70%</td>
</tr>
<tr>
<td>SPSC Queue</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>ReadCopyUpdate</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>100%</td>
</tr>
<tr>
<td>Lockfree Hashtable</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>40%</td>
</tr>
<tr>
<td>MCS Lock</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>MPMC Queue</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>33%</td>
</tr>
<tr>
<td>M&amp;S Queue</td>
<td>11</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>55%</td>
</tr>
<tr>
<td>Linux RW Lock</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>100%</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>19</td>
<td>70%</td>
</tr>
</tbody>
</table>

Figure 5.2: 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 [29], the dequeue interface does not differentiate between dequeuing the integer zero and returning that no item is available, and it passed out 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 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 5.2 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, linearizability would force it to succeed. We restored linearizability by modifying the specification of write_trylock to allow spurious failures. 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:** When we first wrote the specification for this benchmark, we believed it to be linearizable. However, in our evaluation, we tried the following unit test:

```java
void threadA() {
    put(K1, V1);
    r1 = get(K2);
}

void threadB() {
    put(K2, V2);
    r2 = get(K1);
}
```

This test case begins with the value slots for both K1 and K2 equal to the value NULL before threadA and threadB start. We then observed an execution where both r1 and r2 have the value of NULL. Both get invocations fail to observe updates from the corresponding put invocation. This execution is problematic in that its execution graph contains a cycle, which fails the topological sort in ordering commit points. However, as far as we know, it only happens in this benchmark, and besides it breaks both linearizability and sequential consistency!

The Java Class Library’s ConcurrentHashMap does not allow this behavior — it is linearizable and provides more intuitive semantics. Developers who use this hashtable must be extremely cautious — the hashtable exposes some of the surprising behaviors of weak memory models to its clients. For example, the C/C++ memory model guarantees that DRF programs have SC semantics. Programs that use this hashtable do not have this guarantee!

**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 70% of the injections on average. For our 50 injections, 7 of them were detected by checks in CDSChecker, and 28 additional injections were detected by CDSSpec. This shows that by writing specifications, we detect significantly more fault injections.
Chapter 6

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 [33] proposed using linearizability to test and verify concurrent objects. Line-up [10] builds on the Chess [28] model checker to automatically check deterministic linearizability. It automatically generates the sequential specification by systematically enumerating all sequential behaviors. Paraglider [31] supports checking with and without linearization points based on SPIN [23]. All of these approaches assume the SC memory model and a trace that provides an ordering for method invocation and response events. Our work extends the notion of linearizability 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 [30] 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 CD-SChecker.

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.
Chapter 7

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 the classic linearization approach to apply to the C/C++ memory model and allows developers to define the desired behaviors of concurrent data structures with respect to sequential versions of the same data structure. 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.
Bibliography


