The Anchor Verifier for Blocking and Non-blocking Concurrent Software

CORMAC FLANAGAN, University of California, Santa Cruz, USA
STEPHEN N. FREUND, Williams College, USA

Verifying the correctness of concurrent software with subtle synchronization is notoriously challenging. We present the Anchor verifier, which is based on a new formalism for specifying synchronization disciplines that describes both (1) what memory accesses are permitted, and (2) how each permitted access commutes with concurrent operations of other threads (to facilitate reduction proofs). Anchor supports the verification of both lock-based blocking and cas-based non-blocking algorithms. Experiments on a variety concurrent data structures and algorithms show that Anchor significantly reduces the burden of concurrent verification.

CCS Concepts: • **Theory of computation → Program verification**; **Program specifications**; • **Software and its engineering → Concurrent programming languages**; **Formal software verification**.

Additional Key Words and Phrases: concurrent program verification, reduction, synchronization

ACM Reference Format:
Cormac Flanagan and Stephen N. Freund. 2020. The Anchor Verifier for Blocking and Non-blocking Concurrent Software. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 156 (November 2020), 29 pages. https://doi.org/10.1145/3428224

1 INTRODUCTION

Concurrent systems use a wide variety of synchronization disciplines that often include locks, read-only data structures, thread-local exclusive access, different access permissions for reads and writes, atomic compare-and-swap, monotonically increasing updates, and synchronization disciplines that dependent on the values of flags or other data structures. Reasoning about the correctness of algorithms based on such subtle synchronization is tricky and error-prone, particularly in the presence of an unbounded number of preemptive threads.

Much prior work has produced tools for verifying concurrent software [Brookes 2007; Chajed et al. 2018; Cohen et al. 2009; Elmas 2010; Feng 2009; Flanagan et al. 2008, 2002, 2005; Freund and Qadeer 2004; Gu et al. 2018; Hawblitzel et al. 2015; Jung et al. 2015; Leino et al. 2009; Lorch et al. 2020a,a; O'Hearn 2004; Xu et al. 2016; Yi et al. 2012]. Using such tools in practice, however, remains daunting. Some are limited in the types of synchronization they can reason about, while others present programming models far from typical programming languages or require much time and effort to use. Diagnosing verification failures and addressing them is also challenging.

This paper presents the Anchor concurrent software verifier that is designed to address two central challenges in concurrent software verification:

(1) How to clearly and concisely *specify* the sophisticated synchronization mechanisms used in concurrent software.

Authors' addresses: Cormac Flanagan, University of California, Santa Cruz, Santa Cruz, CA, USA; Stephen N. Freund, Williams College, Williamstown, MA, USA.

2475-1421/2020/11-ART156
https://doi.org/10.1145/3428224

(2) How to use those synchronization specifications to *verify* that software behaves correctly.

These specification and verification challenges are interdependent. At one extreme, standard model checking avoids the need for synchronization specifications but has trouble scaling to large numbers of threads. Conversely, scalable verification necessitates precise and expressive specifications.

Synchronization Specifications. Anchor's verification methodology is based on Lipton's theory of reduction [Lipton 1975], which employs a commuting argument to limit where thread interference may occur. Anchor's synchronization specifications facilitate reduction in that they describe both

(1) when each thread is permitted to access (read or write) each shared memory location, and
(2) how each permitted access *commutes* with potential concurrent accesses of other threads.

As an example, consider a field f protected by the lock of the enclosing object. If a thread accesses f without holding that lock, it is an error (denoted E in our specification language). If a thread holds that lock, then an access to f commutes across preceding and succeeding operations of other threads because other threads cannot simultaneously access that lock-protected field. Thus, a lock-protected access is considered a *both-mover* (denoted B). Anchor's synchronization specification for f is:

```
class C {
  int f   moves_as holds(this) ? B : E;
}
```

where the **moves_as** clause specifies the synchronization discipline for f, and the ternary expression "·? · :·" captures the notion that whether an access is permitted, and its commuting properties, depends on whether the current thread holds the protecting lock this.

Synchronization mechanisms are often designed around a rich variety of conditions, including which locks are held, whether an access is a read or a write, whether the enclosing object is thread local, the identity of the accessing thread, and the values of other variables or other aspects of program state. Such conditions are readily captured by Anchor's **moves_as** synchronization specifications. Anchor also supports verification of cas-based non-blocking algorithms.

One key benefit of Anchor's synchronization specifications is that they enable Anchor to reject problematic synchronization designs even before the first line of implementation code is written. To illustrate this point, consider the following declaration indicating that field g can be accessed by any thread at any time, and that all accesses are both-movers.

```
  int g   moves_as B;
```

This declaration is erroneous, as a write to g could conflict with concurrent writes of other threads. Anchor detects this error after inspecting only the declaration of g, and without needing to inspect implementation code using g.

P/C Equivalence. Anchor programs include **yield** annotations documenting where thread interference is observable. Anchor uses the synchronization specifications to verify that these annotations are placed correctly. As such, Anchor ensures that concurrent programs exhibit the same behavior under both

- a *preemptive* scheduler that context switches at any point, and
- a *cooperative* scheduler that context switches only at **yield** annotations.

We call this guarantee *preemptive/cooperative (P/C) equivalence*. It enables subsequent higher-level (formal or informal) reasoning to be based on the more intuitive cooperative scheduler, even though the code runs on standard, preemptive hardware. In addition, code without **yield**s is guaranteed to be atomic, so P/C equivalence is an extension of atomicity that supports more complex synchronization with intentional thread interference [Yi et al. 2012].

Anchor Verifier. We have implemented the Anchor verifier based on the specification and correctness properties described above. A program verified by Anchor

- satisfies its synchronization specification,
- is free of data race conditions (and hence exhibits sequentially-consistent behavior),
- is P/C equivalent,
- satisfies its method specifications, and
- does not go wrong due to assertion failures.

We have evaluated Anchor on a programs ranging from textbook data structures [Herlihy and Shavit 2008] to the FastTrack race detection algorithm [Flanagan and Freund 2010] and a specialized queue found in the LibLFDS library [LibLFDS 2019]. Our experience shows that Anchor is able to verify programs using complex synchronization disciplines without an excessive specification or proof burden. For example, verifying functional correctness for the FastTrack algorithm required adding about 150 lines of specification to 220 lines of implementation code. Prior work verifying those functional requirements in the CIVL verifier for concurrent programs [Hawblitzel et al. 2015] necessitated adding over 500 lines of specification to 250 lines of CIVL code [Flanagan et al. 2018; Wilcox et al. 2018]. Similarly, verifying functional correctness for a lock-free concurrent queue required writing code and specifications totaling to less than 10% of the code, specifications, and proofs required to verify that queue in Armada [Lorch et al. 2020a].

Our correctness argument for Anchor's core analysis is formalized as a stack of three operational semantics, related by appropriate notions of simulation, for an idealized subset of Java. The first *standard semantics* formalizes the execution behavior of programs. The second *instrumented semantics* embodies the core correctness guarantees of our approach and checks that the synchronization discipline is respected and that each yield-free code fragment is reducible. The third *thread-modular semantics* executes one thread of the instrumented semantics in isolation, using the synchronization specifications to model possible behaviors of interleaved threads at yield points. This semantics is the core of our analysis and, by being thread-modular, enables us to reason about an arbitrary number of threads. Specifically, Anchor analyzes the behavior of code under the thread-modular semantics via a translation to Boogie [Barnett et al. 2005]. If Boogie shows the thread-modular semantics does not go wrong, then the original program satisfies the correctness properties above.

Contributions. In summary, the contributions of this paper include the following:

- A concise, expressive notation for synchronization specifications that describes (1) which memory accesses are permitted and (2) how they commute with concurrent operations. (Sections 2 and 4)
- A set of self-consistency checks to detect errors in synchronization specifications early in the development process, even before the first line of concurrent code is written. (Section 5)
- A verification technique for concurrent software that uses synchronization specifications to perform reduction and thread-modular reasoning. (Sections 6 and 7)
- A correctness argument based on three related operational semantics. (Sections 6 and 7)
- The Anchor verifier, an implementation of our technique for a Java-like language, as well as a compiler that generates executable code directly from verified Anchor source. (Section 8)
- An evaluation of Anchor showing its effectiveness on a variety of algorithms. (Section 9)

2 BACKGROUND AND EXAMPLES

Review: Lipton's Theory of Reduction. To verify P/C-equivalence, Anchor utilizes Lipton's theory of reduction [Lipton 1975]. That theory is based on classifying how operations of one thread t commute with concurrent operations of another thread u.

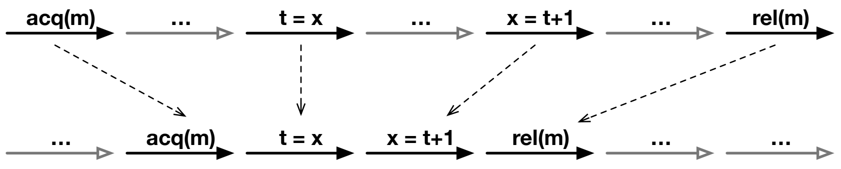


Fig. 1. Commuting operations in Lipton's Theory of Reduction.

- A t-operation is a *right-mover* (denoted R) if it commutes "to the right" of any subsequent u-op, in that performing the steps in the opposite order (i.e. u followed by t) does not change the resulting state. For example, a lock acquire by t is a right-mover because any subsequent u-op cannot modify that lock.
- Conversely, a t-operation is a *left-mover* (denoted L) if it commutes "to the left" of a preceding u-operation. For example, a lock release by t is a left-mover because any preceding u-op cannot modify that lock.
- An operation is a *both-mover* (B) if it is both a left- and a right-mover, and it is a *non-mover* (N) if neither a left- nor a right-mover. For example, a race-free memory access is a both-mover because there are no concurrent, conflicting accesses, but an access to a race-prone variable is a non-mover since there may be concurrent writes.

Consider a sequence of steps performed by a particular thread that consists of zero or more right-movers (R); at most one non-mover (N); and zero or more left-movers (L). Such a sequence $\mathsf{R}^*; [\mathsf{N}]; \mathsf{L}^*$ is said to be *reducible*; any interleaved steps of other threads can be "commuted out" to yield an execution in which the steps run in a cooperative fashion without interleaved steps from other threads. For example, a lock acquire followed by race-free memory accesses and then a lock release is reducible. Figure 1 illustrates how such a sequence of steps interleaved with steps of other threads may be commuted within a trace to produce a sequence with the same behavior, but without interleaved steps. ANCHOR uses this theory to show that each method is P/C equivalent by verifying that all execution paths consist of reducible sequences $\mathsf{R}^*; [\mathsf{N}]; \mathsf{L}^*$ separated by yields.

Synchronization specifications. To leverage Lipton's theory, ANCHOR relies on a synchronization specification that describes (1) when each thread can access each memory location and (2) how each access commutes with steps from other threads. A basic synchronization specification is one of the standard mover classifications (R/L/B/N) or the special error mover E to indicate that an access is not permitted. Commutativity can also be conditioned on a boolean expression *expr*.

$$\begin{array}{rcl} \mathit{Mover} & ::= & \mathsf{B} \mid \mathsf{L} \mid \mathsf{R} \mid \mathsf{N} \mid \mathsf{E} \\ \mathit{SyncSpec} & ::= & \mathit{Mover} \\ & \mid & \mathit{expr}\ ?\ \mathit{SyncSpec}\ :\ \mathit{SyncSpec} \end{array}$$

Here, *expr* may be any boolean expression from the source language, and it may refer to the following additional special variables and predicates.

`tid`:	the thread identifier of the accessing thread.
`isRead()`:	whether the access is a read.
`newValue`:	the value being written, if the access is a write.
`holds(`*obj*`)`:	whether the lock for the object denoted by the expression *obj* is held.
`isLocal(`*obj*`)`:	whether *obj* is accessible by only the current thread.

The remainder of this section illustrates synchronization specifications and our verification technique via a series of examples.

```
  class Node {
    int item  moves_as (isRead() || isLocal(this)) ? B : E;
    Node next moves_as (isRead() || isLocal(this)) ? B : E;

    Node(int item, Node next) {
  B   this.item = item;  this.next = next;
    }
  }
```

```
class LockBasedStack {
  Node head moves_as holds(this) ? B : E;

  modifies this;
  ensures this.head.item == item;
  ensures this.head.next == old(head.next);
  public int push(int item) {
R   acquire(this);
B   Node node = new Node(item, this.head);
B   this.head = node;
L   release(this);
  }


  modifies this;
  ensures $result   == old(this.head.item);
  ensures this.head == old(this.head.next);
  public int pop() {
R   acquire(this);
B   while (this.head == null) {
L     release(this);
      yield;
R     acquire(this);
    }
B   int value = this.head.item;
B   this.head = this.head.next;
L   release(this);
    return value;
  }
}
```

```
class LockFreeStack {
  volatile Node head moves_as N;

  modifies this;
  ensures this.head.item == item;
  ensures this.head.next == old(head.next);
  public void push(int item) {
    while (true) {
N     Node n = this.head;
      yield;
B     Node nu = new Node(item, next);
B/N   if (cas(this.head, n, nu)) { // if cas fails
        break;                     //   then B
      }                            //   else N
    }
  }

  modifies this;
  ensures $result   == old(this.head.item);
  ensures this.head == old(this.head.next);
  public int pop() {
    while (true) {
N     Node top = this.head;
      yield;
      if (top != null) {
B       Node n = top.next;
B/N     if (cas(this.head, top, n)) { // if cas fails
          return top.item;            //   then B
        }                             //   else N
      }
    }
  }
}
```

Fig. 2. LockBasedStack and LockFreeStack examples.

LockBasedStack. The lock-based stack implementation in Figure 2 exemplifies the annotations used to verify P/C equivalence and functional correctness:

- **moves_as** annotations capture synchronizations specifications for fields.
- **yield** annotations indicate where interference may occur in method bodies.
- **requires**, **modifies**, and **ensures** annotations specify method behavior.

We introduce additional annotations below for capturing how variables may be modified by other threads at **yield**s (**noABA** and **yields_as**) and for stating object and loop invariants (**invariant**).

The code in Figure 2 maintains the stack as a linked list of `Node` objects, where reads of the `item` and `next` fields are always permitted and are both movers (B), but writes to these fields are permitted only when the node is still local to its allocating thread. Thus, once nodes are shared between threads, writes are forbidden and are error movers (E).

The `LockBasedStack` class contains a `head` node that is protected by its enclosing lock `this`, and that is implicitly initialized with `null`. The `push()` method allocates a new `Node` and then stores it in `head`. To respect the synchronization discipline, the `push()` method acquires the lock on the stack object prior to accessing `head`. The `push()` method is atomic (reducible) since it consists of a lock acquire (R), a lock-protected read of `head` (B), initialization of a thread-local node (B), a lock-protected write to `head` (B), and a lock release (L), thus matching the pattern $\mathsf{R}^*; [\mathsf{N}]; \mathsf{L}^*$. We include these inferred movers to the left of the code. For field accesses, the synchronization discipline is evaluated at the current program point to determine its commutativity. For example, the write to `this.head` occurs when the lock for `this` is held, and evaluating the synchronization specification for `head` in this context yields B. Note that the `Node` constructor is inlined at its call sites during verification and thus does not require a specification.

The `pop()` method similarly acquires the stack's lock, but then busy waits until the stack has at least one element in it. While waiting, it releases and re-acquires the lock. Since other threads may change `head` between the release and acquire, we include a **yield** at that point. Since **yield** annotations are part of a program's specification, they do not affect the run-time behavior of the code. Any call to `pop()` performs the following sequence of heap operations:

$$\underbrace{\texttt{acq(this)}}_{\mathsf{R}};\ \left[\ \underbrace{\texttt{this.head == null}}_{\mathsf{B}};\ \underbrace{\texttt{rel(this)}}_{\mathsf{L}};\ \texttt{yield};\ \underbrace{\texttt{acq(this)}}_{\mathsf{R}};\ \right]^*\ \underbrace{\texttt{this.head != null}}_{\mathsf{B}};\ \underbrace{\texttt{...}}_{\mathsf{B}};\ \underbrace{\texttt{rel(this)}}_{\mathsf{L}};$$

where the elided steps read and update `this.head`. This sequence $\mathsf{R}; [\mathsf{B}; \mathsf{L}; \texttt{yield}; \mathsf{R};]^*; \mathsf{B}; \mathsf{B}; \mathsf{L}$ consists of reducible sequences $\mathsf{R}^*; [\mathsf{N}]; \mathsf{L}^*$ separated by yields and is thus P/C equivalent. Moreover, the reducible sequences are all no-ops, except for the last, which has the effect of atomically removing a node from the stack, as required by `pop()`'s specification. In that specification, the special variable `$result` refers to the method's return value, and terms of the form `old(`e`)`, such as `old(this.head.item)`, refer to the value of e at the beginning of the yield-free region captured by the specification. Note that `old(this.head.item)` may have a different value than it had at the start of the method call if interference occurs at a documented yield point.

`LockFreeStack`. The alternative `LockFreeStack` implementation in Figure 2 uses optimistic concurrency control instead of a lock. Any thread can read or write the field `head` at any time, All accesses to it are thus non-movers N because of the potential for conflicting concurrent writes. Anchor requires `head` to be declared `volatile` to ensure sequential consistency.[1]

The method `push()` reads `this.head` into `n`, allocates a new node `nu`, and then tries to atomically change `this.head` from `n` to `nu` with an atomic compare-and-swap (`cas`) operation. If the `cas` operation fails, the code retries until it succeeds. The succeeding `cas` operation is a non-mover (N), since it writes to head. Our analysis permits `cas` to fail non-deterministically so we treat failing `cas` operations as both-movers (B). The B/N annotation in the figure captures the `cas` operation's failing/succeeding commutativities. Any call to `push()` performs the following sequence of heap operations:

$$\left[\ \underbrace{\texttt{n=this.head}}_{\mathsf{N}};\ \texttt{yield};\ \underbrace{\texttt{nu=...}}_{\mathsf{B}};\ \underbrace{\texttt{failed-cas}}_{\mathsf{B}};\ \right]^*\ \underbrace{\texttt{n=this.head}}_{\mathsf{N}};\ \texttt{yield};\ \underbrace{\texttt{nu=...}}_{\mathsf{B}};\ \underbrace{\texttt{successful-cas}}_{\mathsf{N}};$$

[1] In general, any field exhibiting N, R, or L commuting behavior may be prone to data races (unordered conflicting accesses). Requiring those fields to be volatile ensures sequential consistency, which simplifies Anchor's formal development and implementation. It also provides a strong guarantee to the programmer even on platforms with relaxed memory models.

```
    class AtomicInt {
      noABA volatile int n  moves_as isRead() ? N
                                        : (newValue == this.n + 1 ? N : E)
                            yields_as newValue >= this.n;

      modifies this;
      ensures this.n == old(this.n) + 1;
      public int inc() {
        while (true) {
N/R       int x = this.n;                        // if cas fails then N else R
B/N       if (cas(this.n,x,x+1)) { return n+1; } // if cas fails then B else N
          yield;
        }
      }
    }
```

Fig. 3. AtomicInt example.

The sequence [N; yield; B; B]*; N; yield; B; N consists of reducible sequences separated by yields and is thus P/C equivalent. As in the previous example, the reducible sequences are all no-ops, except for the last, which has the heap effect of atomically adding the new node to the stack, as required by push()'s specification. The pop() method is similar.

Note that LockFreeStack's methods satisfy the *exact same* specifications as LockBasedStack's, despite using optimistic concurrency control instead of locks.

AtomicInt Example. As another illustration of Anchor's cas-based reasoning, consider class AtomicInt in Figure 3. The synchronization specification for volatile field n indicates that the field can be read and written at any time (N) and that writes must always increment n (expressed via the condition newValue == this.n + 1. That field is labeled **noABA** to indicate that it exhibits *ABA freedom* [Michael 2004]. This means that if a thread reads a value A from n, and then later reads the same value A again, then no other thread could have changed n from A to a different value B, and then back to A again in between those two reads. This property helps prove absence of interference between threads. Anchor verifies that any field declared **noABA** exhibits ABA freedom.

The last loop iteration in inc() consists of a read of n into x, followed by a successful cas changing this.n from x to x+1. Since n is ABA free, no other thread could have written to n between the read and the successful cas. As such, Anchor considers the read operation to be a right-mover, provided it is followed by a successful cas [Wang and Stoller 2005]. On all earlier iterations, the read of this.n is followed by a failed cas, which is a both-mover. Thus, any call to inc() performs the following sequence of heap operations:

$$\left[\ \underbrace{\texttt{x = this.n;}}_{\texttt{N}}\ \underbrace{\texttt{failed-cas;}}_{\texttt{B}}\ \texttt{yield;}\ \right]^{*}\ \underbrace{\texttt{x = this.n;}}_{\texttt{R}}\ \underbrace{\texttt{successful-cas;}}_{\texttt{N}}$$

The sequence [N; B; yield]*R; N consists of reducible sequences separated by yields and is P/C equivalent. Moreover, the reducible sequences are all no-ops, except for the last, which has the heap effect of atomically incrementing this.n, as required by inc()'s specification.

This cas-based reasoning is critical for verifying the correctness of many non-blocking algorithms. It involves verifying that **noABA** fields are free from ABA problems and treating reads from them differently, depending on whether or not each read is followed by a successful cas operation. To the best of our knowledge Anchor is the first SMT-based verifier that automates this kind of reasoning.

Anchor Error Messages. Anchor reports understandable and actionable error messages for incorrect synchronization disciplines, unintended thread interference, and violations of assertions and invariants. For example, adding the following method to LockBasedStack in Figure 2 produces an error report containing the trace on the right below. It contains the commutativity of each synchronization and memory access, and how the commutativity for each access was computed.

```
public void buggy() {             |     call this.push(10)
  this.push(10);                  | [R]   acquire(this);
  // must acquire lock here       |       ...
  assert this.head.item == 10;    | [L]   release(this);
}                                 | [E] tmp9 = this.head; [!isLocal(this) && !holds(this) => E]
```

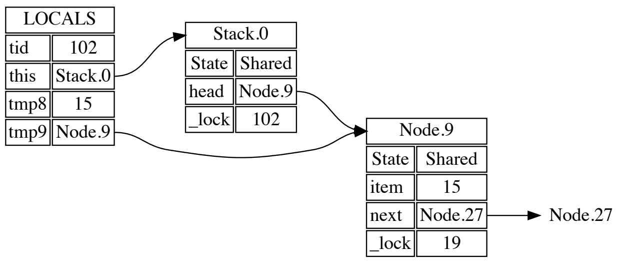
Such traces often immediately identify a synchronization specification violation, as illustrated here by the error commutativity E for the access to this.head without holding the lock. After inserting the necessary synchronization, Anchor reports the following reduction error trace.

```
public void buggy() {             |     call this.push(10)
  this.push(10);                  | [R]   acquire(this);
  // yield necessary here         |       node = new Node();
  acquire(this);                  | [B]   tmp = this.head;  [!isLocal(this) && holds(this) => B]
  assert this.head.item == 10;    | [B]   node.item = 10;   [isLocal(node) => B]
  release(this);                  | [B]   node.next = tmp;  [isLocal(node) => B]
}                                 | [B]   this.head = node; [!isLocal(this) && holds(this) => B]
                                  | [L]   release(this);
                                  | [R] acquire(this);   // R cannot follow L in reducible trace
```

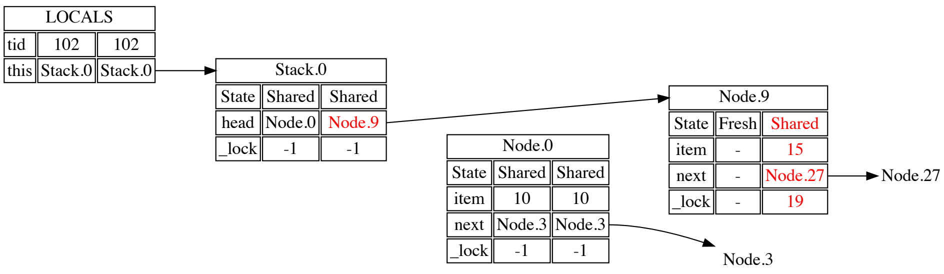
Adding the missing **yield** fixes the reduction error but results in the Anchor reporting that the assertion may fail, again providing a failing trace:

```
public void buggy() {             |     call this.push(10)
  this.push(10);                  | [R]   acquire(this);
  yield; // added                 | [B]   ... as before ...
  acquire(this);                  | [L]   release(this);
  assert this.head.item == 10;    | === yield;
  release(this);                  | [R] acquire(this);
}                                 | [B] tmp9 = this.head; [!isLocal(this) && holds(this) => B]
                                  | [B] tmp8 = tmp9.item; [!isLocal(tmp2) && isRead() => B]
                                  |     assert tmp8 == 10;
```

As in some other tools [Le Goues et al. 2011], Anchor provides a way to inspect the program state via memory snapshots for each line of the trace. The snapshot immediately before the assert is:



The record for the object Node.9 includes its item and next fields, whether it is thread-local or shared, and the tid of the thread holding its lock (if any). Some of these values are not relevant (*e.g.*, the lock holder), but we do see that the head node stores 15 instead of the expected value 10. Inspecting the snapshot for the earlier **yield** indicates that this is where the change occurred.



This snapshot shows the pre-**yield** state and any changes that occurred during the **yield**, specifically that `this.head` was changed from `Node.0` to `Node.9`. Snapshots like these, coupled with the detailed traces, have proven effective for debugging errors both in specifications and in code.

Abbreviations. Anchor supports abbreviations for common synchronization idioms encountered in practice. We introduce those abbreviations here before showing one more sophisticated example.

```
       thread_local  ≡  isLocal(this) ? B : E
       guarded_by l  ≡  holds(l) ? B : E
 write_guarded_by l  ≡  isRead() ? (holds(l) ? B : N) : (holds(l) ? N : E)
           readonly  ≡  isRead() ? B : E
          immutable  ≡  isRead() ? R : E
```

For thread-local fields, the allocating thread initially has exclusive access (B). If the object ever becomes shared between threads, subsequent accesses are errors (E). The `guarded_by` abbreviation captures lock-based exclusive access, as for `LockBasedStack`'s `head` field. The `write_guarded_by` abbreviation captures a more subtle discipline where a lock must be held for writes but not necessarily for reads. Reads while holding the lock are both-movers (B) since there cannot be concurrent writes; reads without holding the lock are non-movers (N) since there may be concurrent writes; and lock-protected writes are non-movers (N) since there may be concurrent reads.

The `readonly` abbreviation is used when a field's value is fixed from the moment the field becomes accessible to multiple threads. Reads of `readonly` fields are both-movers. For immutable fields, writes are forbidden (E), but reads are okay and are right-movers due to the absence of subsequent writes. Reads of immutable fields are not left-movers, however, as the field could have changed before becoming immutable.

FastTrack `VarState` Synchronization Discipline. We conclude this section with the class `VarState` in Figure 4 that shows a more involved synchronization discipline used in the FastTrack dynamic data race detector [Flanagan and Freund 2010; Wilcox et al. 2018].

The key idea is that the synchronization discipline depends on whether the field `readEpoch` is set to the special constant value `SHARED`. For example, the field `readEpoch` itself is initially `write_guarded_by this` (line 141) but becomes `immutable` once it is set to the constant value `SHARED`. The declaration on lines 143–146 has the form

`int[`**moves_as** *SyncDiscipline1*`] vc` **moves_as** *SyncDiscipline2*`;`

to indicate that the field `vc` is protected by *SyncDiscipline2* and the array element `vc[index]` is protected by *SyncDiscipline1* (which may mention `index`). According to this declaration, the array entry `vc[index]` is initially `guarded_by this` (line 143). However, once `readEpoch` becomes `SHARED`, the synchronization discipline for `vc[index]` changes. That memory location can then be read by any thread holding the lock `this` *or* by the thread with thread identifier `tid == index`

```
140     class VarState {
141       volatile int readEpoch  moves_as (this.readEpoch != SHARED) ? write_guarded_by this
142                                                                : immutable;
143       int[moves_as (this.readEpoch != SHARED) ? guarded_by this
144                                              : isRead() ? (holds(this) || tid == index ? B:E)
145                                                         : (holds(this) && tid == index ? B:E)]
146         vc moves_as (this.readEpoch != SHARED) ? guarded_by this : write_guarded_by this;
147       ...
148     }
149
150     class ThreadState {
151       // method spec elided for simplicity
152       public boolean read(VarState sx) {
153     B   int e = ...;  // e is never SHARED
154
155         // fast path:
156     N/R int r = sx.readEpoch;    // if sx.readEpoch != SHARED then N else R
157         if (r == e) return true;
158         if (r == SHARED &&
159       N     sx.vc[tid] == e) return true;
160         yield;
161         // slow path:
162     R   acquire(sx);
163     N   ...
164     L   release(sx);
165       }
166     }
```

Fig. 4. FastTrack VarState synchronization specification.

(line 144). Writes to vc[index] require both holding the lock *and* having tid == index (line 145). Under this discipline, all permitted accesses to vc[index] are both-movers.

These complex specifications enable Anchor to verify the reducibility of several core FastTrack methods, including the read method in Figure 4, which elides locking in the fast-path code to maximize performance. If sx.readEpoch == e, the path is reducible since it only initializes e (B) and reads sx.readEpoch (N) before returning. If sx.readEpoch is not SHARED, the fast-path code is reducible since it only initializes e (B) and reads sx.readEpoch (N) before reaching the **yield**. If sx.readEpoch is SHARED, the code reads sx.readEpoch (now R because it is immutable), reads vc (N because it is write_guarded_by a lock that is not held, line 146), and reads vc[tid] (B because tid is the index being accessed, also line 146). We summarize the two accesses on line 146 as **N** in the figure, since code consisting of a non-mover and a both-mover has the same commutativity properties as just a non-mover. In all three cases, the fast-path code is reducible. The slow-path code after the **yield** is also reducible, which enables Anchor to prove that all executions of this method are P/C equivalent and satisfy its specification.

3 RELATED WORK

A large body of prior work has addressed the important problem of concurrent software verification. We focus primarily on related work employing reduction-based techniques.

Reduction. Lipton [Lipton 1975] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. Doeppner [Jr. 1977], Back [Back 1989], and Lamport and Schneider [Lamport and Schneider 1989] extended this work to proofs of general safety properties, Misra [Misra 2001] to programs built with monitors [Hoare 1974] communicating via procedure calls, and Cohen and Lamport [Cohen and Lamport 1998] to proofs of liveness properties. Farzan and Vandikas [Farzan and Vandikas 2020] recently extended these ideas to encompass a search for a sound reduction strategy.

Calvin-R. ANCHOR and Calvin-R [Freund and Qadeer 2004] both verify concurrent software using reduction and thread-modular reasoning. ANCHOR improves on Calvin-R in terms of expressiveness and usability: it permits different synchronization specifications for reads and writes; its synchronization specifications explicitly describe the mover status of each access, enabling additional self consistency checks; it supports explicit yields to document thread interference; it verifies P/C equivalence; and it provides better error messages with traces and heap snapshots upon verification failure. Unlike Calvin-R, ANCHOR supports method calls via inlining, thus avoiding the burden and complexity of specifying all methods. Calvin-R would not be able to verify most of our benchmarks.

CIVL. ANCHOR is inspired by CIVL [Hawblitzel et al. 2015] (and its predecessor QED [Elmas 2010]), and by the difficulties we encountered while verifying complex algorithms in CIVL. CIVL verifies software through a series of layers of repeated abstraction and reduction, but its expressiveness introduces significant user-visible complexity. Indeed, its acronym “Concurrent Intermediate Verification Language” suggests that it is not targeted towards user-level verification. Moreover, the CIVL source language is not executable. In contrast, ANCHOR supports standard imperative object-oriented code enriched with specifications in order to achieve both expressiveness and usability. Section 9 compares verifying the FastTrack data race detector in both ANCHOR and CIVL.

Armada. Like ANCHOR, Armada [Lorch et al. 2020a] focuses on SMT-based verification via a combination of reduction and thread-modular reasoning, as well as additional strategies for, *e.g.*, data abstraction. A key difference between Armada and ANCHOR is our emphasis on formally specifying synchronization disciplines, checking them for consistency independently from the code, and only then checking that code conforms to those specifications. In contrast, Armada checks commuting properties for all pairs of operations in the code itself. We believe our approach offers some benefits including better documentation, more modular development methodology (outlined in Section 9), more understandable errors messages, and potentially better scalability. In addition, Armada does not include built-in support for `cas`. One avenue for future work is to leverage our expressive synchronization specifications in tools like Armada.

Permission-Based Reasoning. A number of projects use permissions or ownership, in the style of separation logic, to reason about what memory locations are accessible to various threads. Viper [Müller et al. 2016] is a toolchain for verifying sequential and concurrent programs with mutable state, based on permissions or ownership. It supports fractional permissions, allowing multiple threads to simultaneously read a shared memory location, but does not appear to support more sophisticated synchronization disciplines, such as cas-based synchronization. VerCors [Blom et al. 2017] is a tool for static verification of parallel programs using permission-based separation logic that uses Viper as a back-end for their Java and OpenCL verifiers.

Coq-based Techniques. CSpec [Chajed et al. 2018] is a Coq library for verifying concurrent systems modeled in Coq [Coq 2019] using movers and reduction. It is very expressive, particularly because additional proof techniques can be added as additional Coq code, but that flexibility comes at the price of needing to write significant Coq code for both specifications and proofs. CCAL [Gu et al. 2018] provides a compositional semantic model in Coq for composing multithreaded layers of software and for verifying the correctness of software components. It focuses on rely-guarantee reasoning [Jones 1983], which uses a ‘rely’ predicate to modify shared state at *every* point in the analyzed code. In contrast, reduction soundly restricts such modifications to only yield points.

Another Coq-based framework for verifying concurrent software is Iris [Jung et al. 2018], which uses higher-order separation logic to verify correctness of higher-order imperative programs.

Model Checking. Much work is focused on concurrent software model checking [Chamillard et al. 1996; Musuvathi et al. 2008; Yahav 2001], which is hard to apply to concurrent components with an unbounded number of threads. Partial-order methods [Godefroid 1997; Godefroid and Wolper 1991; Peled 1994] have been used to limit state-space explosion while checking concurrent programs.

4 THE ANCHORJAVA CONCURRENT LANGUAGE: SYNTAX AND SEMANTICS

We formalize Anchor's verification methodology using the idealized concurrent language AnchorJava shown in Figure 5. This formalism captures the most novel aspects of our verification technique, including our use of synchronization specifications, reduction, and thread-modular reasoning. Since our approach to enforcing functional method specifications is an adaptation of existing techniques [Freund and Qadeer 2004; Hawblitzel et al. 2015] we omit it from our development to avoid additional complexity and outline that aspect of Anchor in Section 8.

AnchorJava classes include fields and methods. Method declarations $mn(\overline{x})$ `{` s; `return` z `}` include a unique method name mn, formal parameters $\overline{x}$, and a body s followed by a return statement. We omit static types and local variable declarations, which are orthogonal to our development. Statements are mostly in A-normal form [Flanagan et al. 1993; Sabry and Felleisen 1992].

States Σ consist of a heap H and a collection of threads T, where each thread (s, σ) combines a statement s with its thread local environment σ. Expressions e are left abstract, and we use $\sigma(e)$ to denote the evaluation of e with respect to a given thread local environment σ. The heap H maps *locations* to values, where each location $\rho.f$ combines an object *address* ρ with a field name f. The heap also maps each object address ρ to the thread identifier (or Tid) of the thread holding the object's lock, or $\perp$ if the lock is not held. We use $\mathcal{E}$ to range over evaluation contexts and say that a thread $(\mathcal{E}[\texttt{wrong}], \sigma)$ has *gone wrong*. Rather than including a special statement `assert` e, we encode it as `if` (e) `skip wrong`.

The state evaluation relation $\Sigma \rightarrow_t \Sigma'$ performs a step of thread t via an auxiliary relation $(s \cdot \sigma) \cdot H \rightarrow_t (s' \cdot \sigma') \cdot H'$ for threads. Those rules are mostly straightforward. Updates to functions are written as, for example, $\sigma[z := \texttt{true}]$, which extends σ to map z to `true`.

The compare-and-swap operation $z = \texttt{cas}(y.f, x, w)$ compares $y.f$ to x, and if they match, sets $y.f$ to w and z to `true` via [Std CAS+]. Alternatively, the `cas` operation can fail non-deterministically and set z to `false` via [Std CAS-]. If f is declared with the option modifier `noABA`, then locations $\rho.f$ must be *ABA free*. This means they never change from a value A to B and then back to A again. For such locations, we assume there is some ordering $\sqsubseteq_{\rho.f}$ that is respected by writes to $\rho.f$ and the rules [Std CAS+] and [Std Write] check that this ordering is respected.

We define two different evaluation relations (or schedulers) for AnchorJava programs, as shown in Figure 5. The relation $\Sigma \rightarrow \Sigma'$ models *preemptive* scheduling; it performs a step of an arbitrary thread t. The relation $\Sigma \mapsto \Sigma'$ models *cooperative* scheduling where once one thread t is running, it keeps running (without interleaved steps of other threads) until it reaches a yield point. We say that thread t with state (s, σ) is *yielding* if it is at a yield statement ($s = \mathcal{E}[\texttt{yield}]$), has gone wrong ($s = \mathcal{E}[\texttt{wrong}]$), or has terminated ($s = \texttt{skip}$). Thus, the cooperative relation $\Sigma \mapsto \Sigma'$ only performs a step of thread t if all other threads $u \neq t$ are yielding. In particular, context switches only happen at *yielding states* in which all threads are yielding.

5 SYNCHRONIZATION SPECIFICATIONS

In source code, synchronization disciplines are documented with **moves_as** clauses, as in

```
int f    moves_as   SyncSpec;
```

Syntax:

$$\begin{array}{rl}
s \in \mathit{Stmt} ::= & x = y.f \mid y.f = x \\
\mid & z = \mathsf{cas}(y.f, x, w) \\
\mid & x = e \mid x = \mathsf{new}\ c \\
\mid & \mathsf{acq}(x) \mid \mathsf{rel}(x) \mid \mathsf{yield} \\
\mid & z = y.mn(\overline{x}) \mid \mathsf{while}\ e\ s \\
\mid & \mathsf{if}\ e\ s\ s \mid s; s \mid \mathsf{skip} \mid \mathsf{wrong}
\end{array}$$

$$\begin{array}{l}
meth \in \mathit{Method} ::= mn(\overline{x})\ \{\ s; \mathsf{return}\ z\ \} \\
\mathit{field} \in \mathit{Field} ::= \mathsf{noABA}_{opt}\ f \\
D \in \mathit{Defn} ::= \mathsf{class}\ c\ \{\ \overline{\mathit{field}}\ \overline{meth}\ \}
\end{array}$$

$w, x, y, z \in \mathit{Var}$ $\quad e \in \mathit{Expr}$ $\quad f, g \in \mathit{FieldName}$ $\quad mn \in \mathit{MethodName}$ $\quad c \in \mathit{ClassName}$

Standard State:

$$\begin{array}{rcll}
H & \in & \mathit{Heap} & = (\mathit{Location} \rightarrow \mathit{Value}) \\
 & & & \cup\, (\mathit{Address} \rightarrow \mathit{Tid}_\perp) \\
k, l & \in & \mathit{Location} & = \rho.f \\
\rho & \in & \mathit{Address} & \\
v & \in & \mathit{Value} & ::= \rho \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \ldots \\
\sigma & \in & \mathit{Env} & = \mathit{Var} \rightarrow \mathit{Value} \\
T & \in & \mathit{Threads} & = \mathit{Tid} \rightarrow (\mathit{Stmt} \times \mathit{Env}) \\
\Sigma & \in & \mathit{State} & = T \cdot H \\
t, u & \in & \mathit{Tid} & \\
\mathcal{E} & \in & \mathit{EvalCtxt} & = [\bullet] \mid \mathcal{E}; s
\end{array}$$

$$\boxed{(s \cdot \sigma) \cdot H \rightarrow_t (s' \cdot \sigma') \cdot H'}$$

[Std Acq/Rel]
$$\frac{\sigma(x) = \rho \qquad H(\rho) = \perp}{(\mathsf{acq}(x) \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma) \cdot H[\rho := t]} \qquad \frac{\sigma(x) = \rho \qquad H(\rho) = t}{(\mathsf{rel}(x) \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma) \cdot H[\rho := \perp]}$$

[Std Yield]
$$\frac{}{(\mathsf{yield} \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma) \cdot H}$$

[Std New]
$$\frac{\rho \text{ is fresh}}{(x = \mathsf{new}\ c \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma[x := \rho]) \cdot H}$$

[Std Read]
$$\frac{\sigma(y) = \rho}{(x = y.f \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma[x := H(\rho.f)]) \cdot H}$$

[Std Write]
$$\frac{\sigma(y) = \rho \qquad \sigma(x) = v \qquad \text{if } f \text{ is declared as } \mathsf{noABA} \text{ then } H(\rho.f) \sqsubseteq_{\rho.f} v}{(y.f = x \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma) \cdot H[\rho.f := v]}$$

[Std CAS+]
$$\frac{\sigma(y) = \rho \qquad H(\rho.f) = \sigma(x) \qquad \sigma(w) = v \qquad \text{if } f \text{ is declared as } \mathsf{noABA} \text{ then } H(\rho.f) \sqsubseteq_{\rho.f} v}{(z = \mathsf{cas}(y.f, x, w) \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma[z := \mathsf{true}]) \cdot H[\rho.f := v]}$$

[Std CAS-]
$$\frac{}{(z = \mathsf{cas}(y.f, x, w) \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma[z := \mathsf{false}]) \cdot H}$$

[Std Expr/Skip]
$$\frac{}{(x = e \cdot \sigma) \cdot H \rightarrow_t (\mathsf{skip} \cdot \sigma[x := \sigma(t, e)]) \cdot H} \qquad \frac{}{(\mathsf{skip}; s \cdot \sigma) \cdot H \rightarrow_t (s \cdot \sigma) \cdot H}$$

[Std If-True/If-False]
$$\frac{\sigma(e) = \mathsf{true}}{(\mathsf{if}\ e\ s_1\ s_2 \cdot \sigma) \cdot H \rightarrow_t (s_1 \cdot \sigma) \cdot H} \qquad \frac{\sigma(e) = \mathsf{false}}{(\mathsf{if}\ e\ s_1\ s_2 \cdot \sigma) \cdot H \rightarrow_t (s_2 \cdot \sigma) \cdot H}$$

[Std While]
$$\frac{}{(\mathsf{while}\ e\ s \cdot \sigma) \cdot H \rightarrow_t (\mathsf{if}\ e\ (s;\ \mathsf{while}\ e\ s)\ \mathsf{skip} \cdot \sigma) \cdot H}$$

[Std Call]
$$\frac{\begin{array}{c} mn(\overline{x'})\ \{\ s; \mathsf{return}\ z'\ \} \in \bar{D} \\ \forall w \in FV(s) \setminus \{\overline{x'}, \mathsf{this}, z'\}.\ \theta(w) \text{ is fresh} \qquad \theta(\overline{x'}) = \overline{x} \qquad \theta(\mathsf{this}) = y \qquad \theta(z') = z \end{array}}{(z = y.mn(\overline{x}) \cdot \sigma) \cdot H \rightarrow_t (\theta(s) \cdot \sigma) \cdot H}$$

$$\boxed{\Sigma \rightarrow_t \Sigma'} \qquad \boxed{\Sigma \rightarrow \Sigma'} \qquad \boxed{\Sigma \mapsto \Sigma'}$$

[Std Thread]
$$\frac{(s \cdot \sigma) \cdot H \rightarrow_t (s' \cdot \sigma') \cdot H'}{T[t := (\mathcal{E}[s], \sigma)] \cdot H \rightarrow_t T[t := (\mathcal{E}[s'], \sigma')] \cdot H'}$$

[Std Preemptive]
$$\frac{\Sigma \rightarrow_t \Sigma'}{\Sigma \rightarrow \Sigma'}$$

[Std Cooperative]
$$\frac{\Sigma \rightarrow_t \Sigma' \qquad \forall u \neq t.\ T_u \text{ is yielding}}{\Sigma \mapsto \Sigma'}$$

Fig. 5. AnchorJava syntax and standard semantics.

where *SyncSpec* can provide different specifications for reads and writes, and can depend on the state of locks or other data in the heap.

We capture the meaning of such synchronization specifications in our formal development as two functions $R_{\rho.f}$ and $W_{\rho.f}$ drawn from the sets *ReadMover* and *WriteMover*, respectively. These two functions separately define read and write access permissions for a memory location $\rho.f$:

$$\begin{array}{llcll} R_{\rho.f} & \in & \mathit{ReadMover} & = & \mathit{Tid} \times \mathit{Heap} & \rightarrow \mathit{Mover} \\ W_{\rho.f} & \in & \mathit{WriteMover} & = & \mathit{Tid} \times \mathit{Heap} \times \mathit{Value} & \rightarrow \mathit{Mover} \\ & & \mathit{Mover} & ::= & \mathsf{B} \mid \mathsf{L} \mid \mathsf{R} \mid \mathsf{N} \mid \mathsf{E} & \end{array}$$

Thread t can read $\rho.f$ in heap H provided that $R_{\rho.f}(t,H) \neq \mathsf{E}$. In addition, $R_{\rho.f}(t,H)$ specifies the appropriate mover for each permitted read access. For writes, the synchronization discipline can restrict the value written (*e.g.*, to enforce that a variable is monotonically increasing). Thus, the function $W_{\rho.f}$ for $\rho.f$'s write synchronization discipline also takes the value being written.

Movers are partially ordered as follows, based on their decreasing commutativity:

$$\mathsf{B} \sqsubseteq \{\mathsf{L},\mathsf{R}\} \sqsubseteq \mathsf{N} \sqsubseteq \mathsf{E}$$

Detecting Synchronization Errors Early. These synchronization specifications help identify specification errors early, even before the first line of code is written.

Suppose a read of $\rho.f$ by thread t in heap H is a right-mover, i.e. $R_{\rho.f}(t,H) \sqsubseteq \mathsf{R}$. Then the synchronization discipline must prohibit a subsequent write to $\rho.f$ by any other thread u, i.e. $W_{\rho.f}(u,H,\_) = \mathsf{E}$, because the read would not right-commute over such a write. The *validity* condition [Valid Read-R] below enforces this restriction. As an example, it prohibits the erroneous synchronization specification for field g in Section 1, where $R_{\rho.g}(t,H) = W_{\rho.gb}(u,H,v) = \mathsf{B}$.

Similarly, the condition [Valid Write-R] ensures that a right-mover write of v to $\rho.f$ is not followed by a conflicting read or write by another thread in the post state $H[\rho.f := v]$. The final two conditions address left-mover reads and writes and prohibit conflicting accesses in the pre-state.

Definition 5.1 (Validity). *Specification* R, W *is valid if for all* $\rho.f$, t, H, v, *and* u *where* $u \neq t$:

[Valid Read-R] $\quad R_{\rho.f}(t,H) \sqsubseteq \mathsf{R} \;\Rightarrow\; W_{\rho.f}(u,H,\_) = \mathsf{E}$

[Valid Write-R] $\quad W_{\rho.f}(t,H,v) \sqsubseteq \mathsf{R} \;\Rightarrow\; W_{\rho.f}(u,H[\rho.f := v],\_) = \mathsf{E} \;\wedge\; R_{\rho.f}(u,H[\rho.f := v]) = \mathsf{E}$

[Valid Read-L] $\quad R_{\rho.f}(t,H) \sqsubseteq \mathsf{L} \;\Rightarrow\; W_{\rho.f}(u,H[\rho.f := \_],H(\rho.f)) = \mathsf{E}$

[Valid Write-L] $\quad W_{\rho.f}(t,H,v) \sqsubseteq \mathsf{L} \;\Rightarrow\; W_{\rho.f}(u,H[\rho.f := \_],H(\rho.f)) = \mathsf{E} \;\wedge\; R_{\rho.f}(u,H) = \mathsf{E}$

Additionally, suppose thread t is permitted to write v to $\rho.f$ in heap H, i.e. $W_{\rho.f}(t,H,v) \neq \mathsf{E}$. This permission should be *stable* on an interleaved write $\rho'.g = v'$ of another location by another thread u such that $W_{\rho'.g}(u,H,v') \neq \mathsf{E}$. That is, the interleaved write should not change $W_{\rho.f}(t,H,v)$. We refer to this correctness requirement on specifications as *strict stability*. Read permissions may similarly be required to be invariant under interleaved writes, giving us the following requirement.

Definition 5.2 (Strict Stability). *Specification* R, W *is strictly stable if for all* ρ, ρ', t, H, v, v', *and* u *where* $W_{\rho'.g}(u,H,v') \neq \mathsf{E}$ *and* $u \neq t$:

$$\begin{array}{rl} & W_{\rho.f}(t,H,v) = W_{\rho.f}(t,H[\rho'.g := v'],v) \\ \wedge & R_{\rho.f}(t,H) = R_{\rho.f}(t,H[\rho'.g := v']) \end{array}$$

This strict stability condition works well for most programs. To support more subtle synchronization disciplines, Anchor enforces a more relaxed notion of stability that is weaker than Definition 5.2 yet still sufficiently strong to support reduction-based verification. That more complex definition appears in the Supplementary Appendix.

Instrumented State:

$$
\begin{array}{lcllcl}
p & \in & \mathit{Phase} & ::= & \textsc{Pre} \mid \textsc{Post} \mid \textsc{Err} \\
P & \in & \mathit{PhaseMap} & = & \mathit{Tid} \to \mathit{Phase} \\
C & \in & \mathit{CasSet} & = & \mathcal{P}(\mathit{Location} \times \mathit{Tid} \times \mathit{Value}) \\
\Pi & \in & \mathit{InstState} & = & T \cdot H \cdot C \cdot P
\end{array}
$$

$$\boxed{(s,\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (s',\sigma')\cdot H'\cdot C'\cdot p'}$$

[Inst Acq]
$$\frac{\sigma(x)=\rho \qquad H(\rho)=\bot}{(\mathtt{acq}(x)\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma)\cdot H[\rho:=t]\cdot C\cdot(p \triangleright \mathsf{R})}$$

[Inst Rel]
$$\frac{\sigma(x)=\rho \qquad H(\rho)=t}{(\mathtt{rel}(x)\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma)\cdot H[\rho:=\bot]\cdot C\cdot(p \triangleright \mathsf{L})}$$

[Inst Yield]
$$\frac{}{(\mathtt{yield}\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma)\cdot H\cdot C\cdot \textsc{Pre}}$$

[Inst Read]
$$\frac{\sigma(y)=\rho \qquad m=\big(\mathit{CASable}_{\rho.f}(t,H,C)\ ?\ \mathsf{R}\ :\ R_{\rho.f}(t,H)\big)}{(x=y.f\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma[x:=H(\rho.f)])\cdot H\cdot C\cdot(p\triangleright m)}$$

[Inst Write]
$$\frac{\begin{array}{c}\sigma(y)=\rho \qquad \sigma(x)=v \qquad \text{if } f \text{ is declared as noABA then } H(\rho.f)\sqsubseteq_{\rho.f} v \\ \mathit{ValidWrite}_{\rho.f}(C,v) \qquad m=W_{\rho.f}(t,H,v)\end{array}}{(y.f=x\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma)\cdot H[\rho.f:=v]\cdot C\cdot(p\triangleright m)}$$

[Inst CAS+]
$$\frac{\begin{array}{c}\sigma(y)=\rho \qquad H(\rho.f)=\sigma(x) \qquad \sigma(w)=v \qquad \text{if } f \text{ is declared as noABA then } H(\rho.f)\sqsubseteq_{\rho.f} v \\ \mathit{ValidWrite}_{\rho.f}(C,v) \qquad m=W_{\rho.f}(t,H,v) \qquad p\triangleright m\neq \textsc{Err}\end{array}}{(z=\mathtt{cas}(y.f,x,w)\cdot\sigma)\cdot H\cdot (C\cup\{(\rho.f,t,\sigma(x))\})\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma[z:=\mathtt{true}])\cdot H[\rho.f:=v]\cdot C\cdot(p\triangleright m)}$$

[Inst CAS+ Wrong]
$$\frac{\sigma(y)=\rho \qquad \sigma(w)=v \qquad m=W_{\rho.f}(t,H[\rho.f:=\sigma(x)],v) \qquad p\triangleright m=\textsc{Err}}{(z=\mathtt{cas}(y.f,x,w)\cdot\sigma)\cdot H\cdot (C\cup\{(\rho.f,t,\sigma(x))\})\cdot p \Rightarrow_t (z=\mathtt{cas}(y.f,x,w)\cdot\sigma)\cdot H\cdot C\cdot\textsc{Err}}$$

[Inst CAS-]
$$\frac{}{(z=\mathtt{cas}(y.f,x,w)\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (\mathtt{skip}\cdot\sigma[z:=\mathtt{false}])\cdot H\cdot C\cdot p}$$

[Inst Std]
$$\frac{(s\cdot\sigma)\cdot H \rightarrow_t (s'\cdot\sigma')\cdot H \quad s\in\{x=\mathtt{new}\ c, x=e, \mathtt{if}\ e\ s_1\ s_2, \mathtt{while}\ e\ s_1, z=y.mn(\overline{x}), \mathtt{skip}; s_1\}}{(s\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (s'\cdot\sigma')\cdot H\cdot C\cdot p}$$

$$\boxed{\Pi \Rightarrow_t \Pi'}$$

[Inst Thread]
$$\frac{p\neq\textsc{Err} \qquad (s\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (s'\cdot\sigma')\cdot H'\cdot C'\cdot p' \qquad p'\neq\textsc{Err}}{T[t:=(\mathcal{E}[s],\sigma)]\cdot H\cdot C\cdot P[t:=p] \Rightarrow_t T[t:=(\mathcal{E}[s'],\sigma')]\cdot H'\cdot C'\cdot P[t:=p']}$$

[Inst Thread Wrong]
$$\frac{p\neq\textsc{Err} \qquad (s\cdot\sigma)\cdot H\cdot C\cdot p \Rightarrow_t (s'\cdot\sigma')\cdot H'\cdot C'\cdot\textsc{Err}}{T[t:=(\mathcal{E}[s],\sigma)]\cdot H\cdot C\cdot P[t:=p] \Rightarrow_t T[t:=(\mathcal{E}[s],\sigma)]\cdot H\cdot C\cdot P[t:=\textsc{Err}]}$$

$$\boxed{\Pi \Rightarrow \Pi'} \qquad\qquad \boxed{\Pi \Mapsto \Pi'}$$

[Inst Preemptive]
$$\frac{\Pi \Rightarrow_t \Pi'}{\Pi \Rightarrow \Pi'}$$

[Inst Cooperative]
$$\frac{\Pi \Rightarrow_t \Pi' \qquad \forall u\neq t.\ u \text{ is yielding in } \Pi}{\Pi \Mapsto \Pi'}$$

Fig. 6. AnchorJava instrumented semantics.

6 CHECKING REDUCTION

6.1 Instrumented Semantics

We now extend the standard semantics to enforce that (1) all accesses satisfy the synchronization specification and (2) the execution is P/C-equivalent. The resulting instrumented semantics is shown in Figure 6, which uses shading to highlight the major changes over the standard semantics.

Recall that a reducible code sequence has the form $\mathsf{R}^*; [\mathsf{N}]; \mathsf{L}^*$. During the initial *pre-commit* phase, only right-mover operations are permitted. Once the first non-mover or left-mover operation is encountered, that is the *commit point* of the reducible sequence, and subsequently only left-mover operations are permitted. Intuitively, all reads and writes in a reducible sequence can be assumed to occur atomically at the commit point.

The instrumented semantics records, for each thread t, a *phase* $p \in \mathit{Phase}$ indicating whether t is in the Pre or Post commit phase of a reducible sequence. The operation

$$\cdot \triangleright \cdot : \mathit{Phase} \times \mathit{Mover} \to \mathit{Phase}$$

computes the updated phase $p \triangleright m$ for t when it performs a heap access with mover m in phase p. Essentially, in the Pre-commit phase, mover steps B and R are permitted; L and N transition a thread from Pre to Post (this is called the *commit point*); and in the Post-commit phase only B and L steps are permitted. The special phase Err indicates a reduction error occurred.

$\triangleright$	B	R	L	N	E
Pre	Pre	Pre	Post	Post	Err
Post	Post	Err	Post	Err	Err
Err	Err	Err	Err	Err	Err

An instrumented state $\Pi = T \cdot H \cdot C \cdot P$ extends a standard state $\Sigma = T \cdot H$ with a phase map $P : \mathit{Tid} \to \mathit{Phase}$ and also with a *CasSet* C (described below). Thread t is *wrong* in $T \cdot H \cdot C \cdot P$ if either $T_t = (\mathcal{E}[\texttt{wrong}], \sigma)$ or $P_t = \text{Err}$.

The instrumented relation $\Pi \Rightarrow_t \Pi'$ performs a step of thread t via an auxiliary relation $(s \cdot \sigma) \cdot H \cdot C \cdot p \Rightarrow_t (s' \cdot \sigma') \cdot H' \cdot C' \cdot p'$ for threads. (See [Inst Thread].) If $p' = \text{Err}$, the thread and heap state are left unchanged via [Inst Thread Wrong] to facilitate our correctness proofs. The relations $\Pi \Rightarrow \Pi'$ and $\Pi \mapsto \Pi'$ define steps of the instrumented semantics under preemptive and cooperative schedulers, respectively. A thread t is *yielding* in Π if it is at a `yield` statement, has terminated, or has gone wrong.

The rule [Inst Acq] extends [Std Acq] to update the phase from p to $p \triangleright \mathsf{R}$, reflecting that an acquire is a right-mover. Rule [Inst Yield] resets the phase to Pre to initiate a new reducible sequence.

As illustrated by `inc()` in Figure 3, the commutativity of a read depends critically on whether a future `cas` operation is successful. For this reason, the instrumented semantics maintains a CasSet $C \in \mathcal{P}(\mathit{Location} \times \mathit{Tid} \times \mathit{Value})$, where $(\rho.f, t, v) \in C$ means that, in the future, thread t will perform a successful `cas` in which $\rho.f$ is compared to v. In this situation, thread t reading v from an ABA-free location $\rho.f$ is a right-mover provided the read does not violate the synchronization discipline, i.e. $R_{\rho.f}(t, H) \neq \mathsf{E}$. We summarize these conditions via the following predicate, which rule [Inst Read] uses to upgrade the commutativity m of a read from $R_{\rho.f}(t, H)$ to R where appropriate:

$$\mathit{CASable}_{\rho.f}(t, H, C) \overset{\text{def}}{=} f \text{ is declared as } \texttt{noABA} \ \wedge \ (\rho.f, t, H(\rho.f)) \in C_{\rho.f} \ \wedge \ R_{\rho.f}(t, H) \neq \mathsf{E}$$

Rule [Inst Write] extends [Std Write] to update the thread's phase to be $p \triangleright m$ where $m = W_{\rho.f}(t, H, v)$. For ABA-free locations $\rho.f$, the rule requires that the ordering $\sqsubseteq_{\rho.f}$ is respected. In particular, $H(\rho.f) \sqsubseteq_{\rho.f} v$ and for all values v' appearing in C because they are read by a later successful cas, $v \sqsubseteq_{\rho.f} v'$. We formalize the second requirement via the following predicate:

$$\mathit{ValidWrite}_{\rho.f}(C, v) \overset{\text{def}}{=} f \text{ is declared as } \texttt{noABA} \Rightarrow \forall(\rho.f, \_, v') \in C.\ v \sqsubseteq_{\rho.f} v'$$

Rule [Inst CAS+] similarly extends [Std CAS+] with a *ValidWrite* check. It also updates the CasSet C in the pre-state to reflect this successful cas on $\rho.f$, and it updates the phase p to $p \triangleright m$ in the post-state. Thus, a successful read-cas sequence

```
x = y.f; z = cas(y.f, x, w)
```

is reducible as follows, where we assume $R_{\rho.f}(\_, \_) = W_{\rho.f}(\_, \_, \_) = \mathsf{N}$ and $\sigma = [y \mapsto \rho, w \mapsto 2]$:

$$\begin{array}{lllll}
 & T & H & C & P \\
 & [t \mapsto (\,(\texttt{x = y.f; z = cas(y.f,x,w)}), \sigma\,)] & \cdot\ [\rho.f \mapsto 1] & \cdot\ \{(\rho.f, t, 1)\} & \cdot\ [t \mapsto \textsc{Pre}] \\
\Rightarrow_t^* & [t \mapsto (\,\texttt{z = cas(y.f,x,w)}, \sigma[x := 1]\,)] & \cdot\ [\rho.f \mapsto 1] & \cdot\ \{(\rho.f, t, 1)\} & \cdot\ [t \mapsto \textsc{Pre}] \\
\Rightarrow_t & [t \mapsto (\,\texttt{skip}, \sigma[x := 1, z := \texttt{true}]\,)] & \cdot\ [\rho.f \mapsto 2] & \cdot\ \emptyset & \cdot\ [t \mapsto \textsc{Post}]
\end{array}$$

We reach the second state from the first via [Inst Read] followed by [Inst Std] to eliminate the resulting skip. The third state follows from the second via [Inst CAS+].

Rule [Inst CAS+ Wrong] addresses the case were $p \triangleright m = \textsc{Err}$. Two subtleties are worth noting. First, in order to establish the desired commutativity properties for our correctness argument below, this rule needs to left-commute across concurrent writes to $\rho.f$. Therefore we compute the commutativity $m = W_{\rho.f}(t, H[\rho.f := \sigma(x)], v)$ in the heap $H[\rho.f := \sigma(x)]$. This heap is identical to H if the cas could have succeeded. Second, rule [Inst CAS+ Wrong] must update the CasSet C in the pre-state to be $C \cup \{(\rho.f, t, v)\}$. If we were to omit this update to C, then Anchor could not verify the read-cas code snippet above. Specifically, the read would be a non-mover if C were empty, transitioning the thread to the Post state. The cas operation would then execute via [Inst CAS+ Wrong] due to the reduction error as $\textsc{Post} \triangleright N = \textsc{Err}$, as illustrated below.

$$\begin{array}{lllll}
 & T & H & C & P \\
 & [t \mapsto (\,(\texttt{x = y.f; z = cas(y.f, x, w)}), \sigma\,)] & \cdot\ [\rho.f \mapsto 1] & \cdot\ \emptyset & \cdot\ [t \mapsto \textsc{Pre}] \\
\Rightarrow_t^* & [t \mapsto (\,\texttt{z = cas(y.f, x, w)}, \sigma[x := 1]\,)] & \cdot\ [\rho.f \mapsto 1] & \cdot\ \emptyset & \cdot\ [t \mapsto \textsc{Post}] \\
\Rightarrow_t & [t \mapsto (\,\texttt{z = cas(y.f, x, w)}, \sigma[x := 1]\,)] & \cdot\ [\rho.f \mapsto 1] & \cdot\ \emptyset & \cdot\ [t \mapsto \textsc{Err}]
\end{array}$$

6.2 Correspondence to the Standard Semantics

The instrumented semantics ($\Rightarrow$) behaves the same as a standard semantics ($\rightarrow$), except it may go wrong as a result of detecting a violation of the synchronization specifications or reducibility. In proving this correspondence, we assume that specification R, W is valid and stable; P_0 denotes the initial phase map $\lambda t.\ \textsc{Pre}$; and that the initial program state $\Sigma_0 = T_0 \cdot H_0$ is yielding and *nonblocking*, meaning that any reducible code sequence reaching its commit point must terminate.

Note that the initial CasSet C_0 must be appropriately chosen to reflect the successful cas operations that will be performed during the run. A CasSet C is *valid* for a heap H if C contains only values for future cas operations that are properly ordered with respect to the current values in H:

$$C \textit{ is valid for } H \quad \text{iff} \quad \forall \rho.f, v.\ f \text{ is declared as } \texttt{noABA} \wedge (\rho.f, \_, v) \in C \Rightarrow H(\rho.f) \sqsubseteq_{\rho.f} v$$

Theorem 1. If $T_0 \cdot H_0 \rightarrow^* T \cdot H$ then there exists a valid C_0 for H_0 such that either:

(1) there exists P such that $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \Rightarrow^* T \cdot H \cdot \emptyset \cdot P$, or
(2) $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \Rightarrow^* \Pi$ where Π is wrong.

Proof. By induction on the length of $\rightarrow^*$ and case analysis. □

Consequently, if the standard semantics can go wrong then so does the instrumented semantics.

Thread-Modular State:

$\Theta \quad \in \quad \mathit{ThreadModularState} = (s \cdot \sigma) \cdot H \cdot C \cdot p$

$\boxed{\Theta \ \Rrightarrow_t \Theta'}$

$$[\text{TM STEP}] \quad \frac{(s \cdot \sigma) \cdot H \cdot C \cdot p \Rightarrow_t (s' \cdot \sigma') \cdot H' \cdot C' \cdot p' \quad p' \neq \text{ERR}}{(\mathcal{E}[s] \cdot \sigma) \cdot H \cdot C \cdot p \ \Rrightarrow_t (\mathcal{E}[s'] \cdot \sigma') \cdot H' \cdot C' \cdot p'}$$

$$[\text{TM STEP WRONG}] \quad \frac{(s \cdot \sigma) \cdot H \cdot C \cdot p \Rightarrow_t (s' \cdot \sigma') \cdot H' \cdot C' \cdot \text{ERR}}{(\mathcal{E}[s] \cdot \sigma) \cdot H \cdot C \cdot p \ \Rrightarrow_t (\mathcal{E}[s] \cdot \sigma) \cdot H \cdot C \cdot \text{ERR}}$$

$$[\text{TM YIELD}] \quad \frac{\mathit{Yield}(t, H, H')}{(\mathcal{E}[\texttt{yield}] \cdot \sigma) \cdot H \cdot C \cdot p \ \Rrightarrow_t (\mathcal{E}[\texttt{yield}] \cdot \sigma) \cdot H' \cdot C \cdot p}$$

$$\text{where } \mathit{Yield}(t, H, H') \quad = \quad \forall x.\ R_x(t, H) \sqsubseteq \mathsf{R} \Rightarrow H(x) = H'(x)$$
$$\wedge \quad \forall l.\ H(l) = t \Leftrightarrow H'(l) = t$$

Fig. 7. AnchorJava thread-modular semantics.

Corollary 1. If $T_0 \cdot H_0$ goes wrong under $\rightarrow$ then there exists a valid C_0 for H_0 such that $T_0 \cdot H_0 \cdot C_0 \cdot P_0$ goes wrong under $\Rightarrow$.

Finally, a cooperative run of the instrumented semantics is cooperative under the standard semantics.

Theorem 2. If $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \Mapsto^* T \cdot H \cdot C \cdot P$ for some C_0, T, H, C, and P, then $T_0 \cdot H_0 \mapsto^* T \cdot H$.

Proof. By induction on the length of $\Mapsto^*$ and case analysis. □

6.3 Reducibility of the Instrumented Semantics

We next show that the instrumented semantics is P/C-equivalent, *i.e.* that any preemptive run ($\Rightarrow$) is reducible to a cooperative run ($\Mapsto$), under the assumption that cooperative runs do not go wrong.

Theorem 3 (Reduction). If $\Pi = T_0 \cdot H_0 \cdot C_0 \cdot P_0$ and Π_0 does not go wrong under $\Mapsto$, then:

(1) Π_0 does not go wrong under $\Rightarrow$.
(2) If $\Pi_0 \Rightarrow^* \Pi$ where Π is yielding, then $\Pi_0 \Mapsto^* \Pi$.

Proof. See Supplementary Appendix. □

7 THREAD-MODULAR SEMANTICS

Our final semantics is a *thread-modular* semantics (Figure 7) that provides the foundation for our verifier, as it avoids explicitly reasoning about all interleavings of an unbounded number of threads. Instead, it analyzes one thread at a time in a cooperative fashion, using the synchronization specification to model possible behaviors of other threads at yield points.

When analyzing some thread t, the thread-modular semantics maintains a state $\Theta = (s \cdot \sigma) \cdot H \cdot C \cdot p$ consisting of the thread state $s \cdot \sigma$ for t, the heap H, the CasSet C, and the phase p of that thread. The relation $\Theta \ \Rrightarrow_t \Theta'$ performs an arbitrary step possible in state Θ. This relation models steps of t precisely by delegating to the instrumented semantics via rule [TM Step] or [TM Step Wrong]. If $s = \mathcal{E}[\texttt{yield}]$, then rule [TM Yield] models possible behaviors of other threads under the given synchronization specification. In particular, the relation $\mathit{Yield}(t, H, H')$ captures heap mutations that threads other than t may perform in H. A variable x can only change in H' if thread t does not have right-mover read permission, and a lock l is held by t in H' iff it was held by t in H.

Note that, at each `yield`, [TM Yield] can fire multiple times before [TM Step] transitions to a non-yielding state, thus modeling non-deterministic scheduling.

The following simulation mapping $\Pi|t$ maps the instrumented state Π to a corresponded thread-modular state Θ by projecting away threads other than t:

$$(T \cdot H \cdot C \cdot P)|t = T(t) \cdot H \cdot \{\, (t, \_, \_) \in C\} \cdot P(t)$$

For any instrumented state Π_0, running $\Pi_0|t$ under $\Rrightarrow_t$ simulates (over-approximates) the behavior of thread t under $\Mapsto$ and so detects if t would go wrong.

Theorem 4 (Simulation). Suppose $\Pi_0 \Rightarrow^* \Pi$. Then $(\Pi_0|t) \Rrightarrow_t (\Pi|t)$.

Proof. By induction on the length of $\Pi_0 \Rightarrow^* \Pi$ and case analysis. □

Corollary 2. If $\Pi_0 \Rightarrow^* \Pi$ where thread t has gone wrong in Π, then $\Pi_0|t$ goes wrong under $\Rrightarrow_t$.

We now state and prove our central correctness result relating the behavior of the original program $\Sigma_0 = T_0 \cdot H_0$ and the corresponding thread-modular programs.

Theorem 5 (Correctness). Given an initial program $T_0 \cdot H_0$, suppose that for all t and all initial CasSets C_0, the thread-modular program $(T_0 \cdot H_0 \cdot C_0 \cdot P_0)|t$ does not go wrong under $\Rrightarrow_t$. Then $T_0 \cdot H_0$ does not go wrong under $\rightarrow$ and is P/C-equivalent.

Proof. For any C_0, let $\Pi_0 = T_0 \cdot H_0 \cdot C_0 \cdot P_0$. By Theorem 4, Π_0 does not go wrong under $\Mapsto$. By Theorem 3(1), Π_0 does not go wrong under $\Rightarrow$. Since this holds for any C_0, by Corollary 1, Σ_0 does not go wrong under $\rightarrow$. Next, consider any standard, preemptive run $T_0 \cdot H_0 \rightarrow^* T \cdot H$ where $T \cdot H$ is yielding. By Theorem 1, $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \Rightarrow^* T \cdot H \cdot \emptyset \cdot P$ for some C_0 and P. By Theorem 3(2), $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \Mapsto^* T \cdot H \cdot \emptyset \cdot P$. By Theorem 2, $T_0 \cdot H_0 \mapsto^* T \cdot H$.

□

8 ANCHOR VERIFIER

We have implemented the Anchor verifier based on these ideas. The input language for Anchor is a subset of Java extended with specifications. Anchor does not yet support some Java features, such as inheritance, subtyping, generics, and the full set primitive types. Anchor also includes a compiler to generate executable Java code directly from Anchor source code. That compiler utilizes the standard Java concurrency library [Lea 2019] to implement locks and `cas` operations.

Given a target program, Anchor verifies that the synchronization specification is valid, stable, and respected by the code; that the code is race-free and P/C equivalent; that methods conform to their functional specifications (if provided); and that no assertions fail. To check these properties, Anchor generates a Boogie program embodying the necessary validity/stability checks and the execution of each `public` method under the thread-modular semantics. Anchor then checks that program with the Boogie verifier [Barnett et al. 2005]. Any errors are mapped back to the Anchor source and reported along with details extracted from Boogie's counter-examples.

Modeling Heap Changes at Yield Points. The thread-modular semantics models the effect of a **yield** as zero or more heap updates via the *Yield* relation. That relation is defined in terms of each memory location's read access permission: the value of any memory location for which the current thread has right-mover read access is preserved. This rule works for the vast majority of cases, but it may admit changes to `x.f` that are not actually feasible if updates to `x.f` are restricted in some particular ways. For example, consider again `AtomicInt` from Figure 3 and a client of that class:

```
public void incTwice(AtomicInt counter) {
  int orig = counter.inc();
  yield;
  assert counter.inc() > orig;
}
```

Ignoring the **yields_as** annotation on field n in Figure 3 for the moment, Anchor would be unable to prove that the assertion in incTwice always succeeds even though it does. Specifically, since the current thread does not have right-mover read access to counter.n at the **yield**, Anchor must assume counter.n could have any value after the **yield**.

This motivates our introduction of optional **yields_as** annotations to better approximate heap updates. These annotations relate a field's post-yield value to its pre-yield value. For our AtomicInt class, we include a **yields_as** annotation guaranteeing the value of n may only increase at a **yield**. Anchor verifies that **yields_as** annotations subsume all writes permissible by the synchronization specification and that the resulting Yield relation is reflexively and transitively closed.

The related **noABA** annotation also restricts how a field may be changed at a yield point. Anchor verifies that such a field exhibits ABA freedom, namely that that the field is never changed from a value A to B and then back to A again, using the **yields_as** annotation. For example, the **yields_as** annotation on n above indicates that if n is currently A and another thread updates it to a new and distinct value B, it must be that $B > A$. No thread could change n back to A without violating the field's **yields_as** specification since $A \not\geq B$.

Method Specifications. Anchor's technique for verifying public method specifications is inspired by earlier work on Calvin-R [Freund and Qadeer 2004] and CIVL [Hawblitzel et al. 2015]. Space limitations prevent full discussion, but we illustrate the core ideas with several small examples.

A public method containing no **yield**s always has atomic behavior. That behavior can be specified with standard **requires**, **modifies**, and **ensures** annotations, as illustrated in Figure 2 for the LockBasedStack.push() method. A public method containing **yield**s may also exhibit atomic behavior if it never executes more than one yield-free code region with visible side effects. The LockBasedStack.pop() method in Figure 2 has this property.

The behavior of public methods with non-atomic behavior can be specified via a sequence of blocks containing **modifies** and **ensures** clauses. For example, the incTwice method above can be specified as:

```
{ modifies counter; ensures counter.n == old(counter.n) + 1; }
yield;
{ modifies counter; ensures counter.n == old(counter.n) + 1; }
```

We include the **yield** keyword in the specification to reinforce that interference may occur at those intermediate points, and that the heap state at those points may be exposed to other threads.

The add method for a sorted linked list implemented with hand-over-hand locking is similarly non-atomic and be specified as a sequence of three atomic blocks that refer to an additional specification variable ptr. The first initializes ptr to the head of the list; the second (which may be iterated any number of times) moves ptr one node further down the list, performing the appropriate synchronization, until the insertion point is found; and the third inserts a new node at that location.

Anchor verifies that method bodies conform to their specifications a simulation check that matches the execution of a yield-free region to a corresponding step taken in the specification.

Thread-local Analysis. Anchor uses a simple strategy to identify thread-local objects. A newly-allocated object is thread-local until a reference to it is stored in another object, at which point it is considered shared. Thus, shared objects never refer to thread-local data. The same applies to arrays. Also, Anchor assumes that the receiver and all objects passed as parameters to public methods are shared, and that the receiver of a call to a public constructor is considered local. One area for future work is to integrate a more robust analysis for ownership [Clarke et al. 1998].

Method Inlining. Anchor verifies each public method modularly but handles all embedded method calls via inlining, which simplifies our formal development and implementation when compared to existing techniques such as Calvin-R [Freund and Qadeer 2004], Armada [Lorch et al. 2020a], and CIVL [Hawblitzel et al. 2015]. In particular, inlining enables us to avoid forcing the programmer to write (and verify) full specifications of helper methods when using Anchor. Since the concurrent components we currently target are typically relatively small, we do not expect this to become a limiting factor.

Object Invariants. Anchor also supports object invariants. For example, to express that `Node`s in a linked list must be ordered by their `items`, we can add the following invariant to our `Node` class from Figure 2.

```
invariant this.next != null  ==>  this.item <= this.next.item;
```

Anchor assumes all object invariants hold at the start of each `public` method and immediately after each **yield** operation, and it verifies that the invariants hold at the end of each `public` method and immediately before each **yield** operation.

Post **Phase Termination and Termination Metrics.** Our correctness argument requires that code is non-blocking, *i.e.* that any reducible sequence reaching its commit point must terminate. As such, Anchor verifies that any thread entering the Post phase of a reducible block reaches a yield point, and it rejects any program that, when in the Post phase, block indefinitely (*e.g.*, due to an `acquire`) or loop indefinitely. To prove loops terminate, the programmer may supply a termination metric, as in, *e.g.*, Dafny [Leino 2010], via a **decreases** annotation that provides a non-negative integer expression that decreases in value on each loop iteration. (Standard loop invariants are also supported.)

9 EVALUATION

We evaluated Anchor on a variety of standard concurrent collection classes, the FastTrack dynamic data race detection algorithm [Flanagan and Freund 2010; Wilcox et al. 2018], and a specialized queue found in the LibLFDS library [LibLFDS 2019].

9.1 P/C Equivalence for Concurrent Collections

We begin by verifying synchronization specifications and P/C equivalence for a dozen concurrent collection implementations adapted from Java implementations presented in *The Art of Multiprocessor Programming* [Herlihy and Shavit 2008]. Table 1 lists these programs, including references to the origins of the techniques employed in the more sophisticated versions. We chose these programs because they contain synchronization idioms representative of those used in large, complex systems. Indeed, many of the more sophisticated programs we studied are not only subtle enough to have had full technical papers dedicated to them but have also served as canonical examples of how to design high performance or lock-free algorithms. In general, the programs required several minutes to several hours of programmer time to specify and verify, depending on the complexity of the synchronization. We adopted the following methodology in our work:

(1) Write the synchronization specification and verify validity and stability.
(2) Implement the code and verify it in a single-threaded setting. This step is enabled by command-line option that configures Anchor to assume no heap changes occur during **yield**s. In other words, we verify the program assuming no thread interference.
(3) Verify the code in the default setting in which Anchor assumes multiple threads may be running concurrently. In this case, the heap may change at **yield**s.

We found the first two steps quite valuable. The first enabled us debug synchronization specifications without writing any code. We found that once we understood the basic design of the code, writing the specification was fairly straightforward, and indeed precisely writing it down often helped us

Table 1. Concurrent Collections verified for P/C equivalence by ANCHOR. (Section 9.1.)

Program	Synchronization Idiom	Total Size (LOC)	Sync. Specs	Yields	`requires`/ Invariants	ANCHOR Time (s)
Stacks (w/ `push`, `pop`, and `init`)						
Coarse	Lock-based (as in Figure 2)	31	3	0	0	3.20
Lock-free	CAS-based (as in Figure 2) [Treiber 1986]	43	3	2	0	3.48
Linked Lists (w/ `add`, `remove`, `contains`, and `init`)						
Coarse	One lock for all nodes	100	4	0	26	4.24
Fine	Hand-over-hand locking [Bayer and Schkolnick 1977]	115	4	3	22	3.62
Optimistic	Fine, with lock-free search + validation [Herlihy and Shavit 2008]	157	4	14	26	4.60
Lazy	Optimistic, with lock-free contains [Heller et al. 2005]	113	4	11	18	4.68
Lock-Free	CAS-based [Harris 2001; Michael 2002]	192	4	14	26	21.46
Queues (w/ `enqueue`, `dequeue`, and `init`)						
Coarse	One lock for all nodes	45	2	0	6	4.79
Lock-Free	CAS-based [Michael and Scott 1996]	74	2	8	4	5.55
Bounded	Separate locks for enq/deq, bounded size [Lea 2019; Michael and Scott 1996]	137	2	6	16	35.49
HashSets (w/ `add`, `remove`, `contains`, and `init`)						
Coarse	One lock for all buckets	190	3	0	28	14.68
Striped	array of locks for buckets [Lea 2019]	232	3	3	39	62.09

understand why the implementation was correct. When we did make mistakes, ANCHOR's error reports enabled us to identify and refine problematic cases. Those reports typically pointed directly to the part of the synchronization specification that needed to be strengthened or weakened.

The second step enabled us to identify and fix many basic programming and reduction errors before verifying correctness in the presence of heap changes at **yield** points. As in the first step, the error reports, like those in Section 2, helped isolate errors quickly.

Table 1 lists the number of non-empty, non-comment lines of code in each program, as well as counts of fields with synchronization specifications, **yield** annotations, `requires` clauses, and loop/object invariants. The table also includes ANCHOR verification time (computed as the average of 10 runs on a MacOS 10.15 computer with 3.2 GHz Intel Core i7 processor and 64GB of memory). In addition to lock-based and lock-free stack implementations from Figure 2, we include five linked list implementations that store integers in increasing order with no duplicates. The high number of **requires** clauses and invariants needed to verify those lists are related to ordering rather than concurrency and would be needed even if verifying a sequential implementation.

The Coarse List protects all linked list nodes with a single lock, and the Fine List protects each node with its own lock. The latter utilizes hand-over-hand locking to support concurrent traversals. The **yield**s in Fine, Optimistic, and Lazy Lists indicate that other threads may change the list as one thread traverses it. The Lock-Free list foregoes locks altogether in favor of CAS operations. As one would expect, the three optimistic cases require more **yield** operations and result in proof obligations that are more difficult to dispatch, as evidenced by the higher ANCHOR run times.

The Bounded Queue has complex synchronization that uses separate `enqLock` and `deqLock` locks for adding and removing items, enabling those operations to run in parallel. The high ANCHOR verification time seems due to the theories and strategies employed by the underlying Z3 SMT solver [de Moura and Bjørner 2008] to handle modular arithmetic.

Table 2. Components verified for P/C equivalence and functional correctness.

Program		Size (LOC) Code	Specs and Proofs	Total	Verification Time (s)
Fastrack Datarace Detection Algorithm					
Anchor	Section 9.2	220	150	370	40.8
CIVL	[Wilcox et al. 2018]	261	528	789	10.0
Anchor/CIVL				**0.46**	**4.08**
Single Enqueuer/Dequeuer Lock-Free Queue					
Anchor	Section 9.3	35	27	62	13.4
Armada	[Lorch et al. 2020a]	70	694	764	2167
Anchor/Armanda				**0.08**	**0.006**

The Striped HashSet also exhibits a complex synchronization strategy. That class uses elements in a `locks` array to guard linked lists stored in a resizable `table` array. Specifically, `table[index]` is protected by `locks[i % this.locks.length]`. A resizing operation replaces the `table` with a larger one when a certain usage threshold is reached. To ensure that resizing happens atomically with respect to other concurrent operations, the `table` field can be read when any lock in the `locks` array is held but can only be modified when *all* locks in `locks` are held:

```
List[moves_as guarded_by this.locks[index % this.locks.length]] table
  moves_as isRead() ? (exists int i :: 0 <= i < this.locks.length &&  holds(this.locks[i])) ? B : E
                    : (forall int i :: 0 <= i < this.locks.length ==> holds(this.locks[i])) ? B : E;
```

As before, the need to reason about modular arithmetic leads to longer verification times.

9.2 Comparison to CIVL: The FastTrack Race Detector

We also implemented and verified the core FastTrack dynamic data race detection algorithm in Anchor. We chose this case study because it is a complex, self-contained concurrent algorithm that was previously verified [Wilcox et al. 2018] in the CIVL verifier [Hawblitzel et al. 2015]. The algorithm uses a variety of synchronization idioms to ensure lock-free handling of the most common cases, including a mix of immutable, thread-local, read-only, lock-protected, write-protected, and volatile data, as well as synchronization mechanisms that change over time.

Despite having a sophisticated synchronization discipline, the algorithm itself is straightforward and was originally described in roughly 120 lines of Java-like pseudo-code. Our Anchor implementation is about 220 lines, with the additional lines due to syntactic differences, the inclusion of constructors absent in the original, and field and method duplication necessitated by the lack of inheritance. We verified that our code correctly implements a complete, formal specification of the FastTrack analysis. As shown in Figure 2, this required adding 150 lines of specification on top of the original 220 lines of code for a total of 370 lines. Anchor verified this program in about 40 seconds. Our implementation is available in the companion artifact for this paper [Flanagan and Freund 2020].

Comparison to CIVL. We previously implemented the same algorithm in CIVL using 211 lines of code plus 528 lines of specification, for a total of 789 lines [Flanagan et al. 2018; Wilcox et al. 2018]. In other words, the total code base for Anchor was 42% the size of CIVL's.

Those larger specifications are quite involved and required a significant amount of time and effort to develop. One reason for this difference is Anchor's concise synchronization specifications, and its ability to characterize the possible effects of thread interference at yield points using those specifications. More verbose and low-level mechanisms are needed to do that in CIVL. For example, in CIVL, the commutativity of memory accesses is captured in the declarations of specialized accessor functions. For the `readEpoch` field from Figure 4, the two-line Anchor synchronization specification becomes four accessor functions in CIVL: `VarStateGetR` and `VarStateSetR` for reading and writing to the field with the lock held (`B` and `N`, respectively); `VarStateGetRNoLock` for reading without the lock held (`N`); and `VarStateGetRShared` for reading when the value is `SHARED` (`R`). An additional "refined" version for each function must be included to specify its commuting behavior. One of these accessors functions and its refinement is shown below.

```
// Accessor for VarState's readEpoch when it is SHARED.
procedure {:yields} {:layer 0} {:refines "AtomicVarStateGetRShared"}
VarStateGetRShared({:linear "tid"} tid: Tid, x: VarState) returns (e: Epoch);

// Refinement capturing commutativity and context in which it can be used.
procedure {:right} {:layer 1,20}
AtomicVarStateGetRShared({:linear "tid"} tid: Tid, x: VarState) returns (e: Epoch) {
    assert ValidTid(tid); assume sx.readEpoch[x] == SHARED; e := SHARED;
}
```

In addition to higher notational overhead, one consequence of this approach is that we cannot easily write code in which a specific memory access has different commutativities on different execution paths. This scenario occurs when reading `sx.readEpoch` on line 156 in Figure 4. The commutativity depends on whether the value read is `SHARED`, as captured in the Anchor specification for `readEpoch`. In contrast, we must write the following in CIVL to capture that subtlety:

```
if (*) {
  // read of sx.readEpoch is N
  call r := VarStateGetRNoLock(tid, sx); assume r != SHARED; ...
} else {
  // read of sx.readEpoch is R
  call r := VarStateGetRShared(tid, sx); assume r == SHARED; ...
}
```

Such encodings move the verified version further away from an executable version, and they rely on reasoning not captured by the CIVL verifier to ensure that the assumptions in the two branches cover all possible behaviors.

Anchor also uses synchronization specifications to model potential heap updates at yield points. In CIVL, on the other hand, the programmer must describe those potential heap updates manually.

While there are some clear benefits to Anchor's higher-level approach to synchronization specifications, one downside is that Anchor generates more complex verification conditions. Indeed Anchor took about 4x longer than CIVL to verify the Anchor FastTrack implementation. We have yet not conducted performance tuning, and we expect introducing specialized triggers for quantified formulas to improve performance.

Finally, we note that CIVL employs a lower-level programming language suitable for verification but not for execution or ease of use. In contrast, Anchor can compile input programs directly into Java, meaning that the executed code is exactly the code that is verified.

9.3 Comparison to Armada: Single Enqueuer/Dequeuer Lock-Free Queue

We also implemented the largest benchmark from the recently published work on Armada [Lorch et al. 2020a]. That benchmark is a non-blocking queue from LibLFDS [LibLFDS 2019] that designed for use by a single enqueuer thread and single dequeuer thread [Lamport 1983]. The original C++ code represents a queue of key-value pairs as an array of small structures. Since Anchor does not support structures we instead employ an array of `Pair` objects.

```
class Queue {

 // Concrete State
 array ElemsType = Pair[moves_as RepInv && ((tid == 0 && inTailToHead(index)) ||
                                           (tid == 1 && inHeadToTail(index))) ? B : E];

 [ElemsType] elems moves_as isLocal(this) ? B : readonly;

 volatile int tail moves_as isRead() ? tid == 0 ? B : N
                                   : tid == 0 && inTailToHead(newValue) ? N : E
                  yields_as RepInv ==> inTailToHead(newValue);

 volatile int head moves_as isRead() ? tid == 1 ? B : N
                                   : tid == 1 && (inHeadToTail(newValue) || newValue == this.tail) ? N : E
                  yields_as RepInv ==> inHeadToTail(newValue) || newValue == this.tail;

 invariant RepInv;

 // Abstract State
 ghost Seq<Pair> spec;
 invariant this.spec.length == (Len + this.tail - this.head) % Len;
 invariant (forall int i :: inHeadToTail(i) ==> this.elems[i] == this.spec[(Len + i - this.head) % Len]);

 modifies this;
 ensures this.spec == [ ];
 public Vector() {
   // spec, head, tail all zero/empty initialized
   this.elems = new [ElemsType](512);
 }

 requires tid == 0;
 modifies this;
 ensures !$result && this.spec == old(this.spec)
      || $result && this.spec == old(this.spec) ++ [x];
 public boolean enqueue(Pair x) {
N   if ((this.tail + 1) % Len != this.head) {
     yield;
B     this.elems[this.tail] = x;
N     this.tail = (this.tail + 1) % Len;
     this.spec = this.spec ++ [x];
     return true;
   } else {
     return false;
   }
 }

 requires tid == 1;
 modifies this;
 ensures $result == null && this.spec == old(this.spec)
      || old(this.spec) == [$result] ++ this.spec;
 public Pair dequeue() {
N  if (this.head != this.tail) {
     yield;
B    int result = this.elems[this.head];
N    this.head = (this.head + 1) % Len;
     this.spec = this.spec[1..SeqLen(this.spec)];
     return result;
   } else {
     return null;
   }
 }
}
```

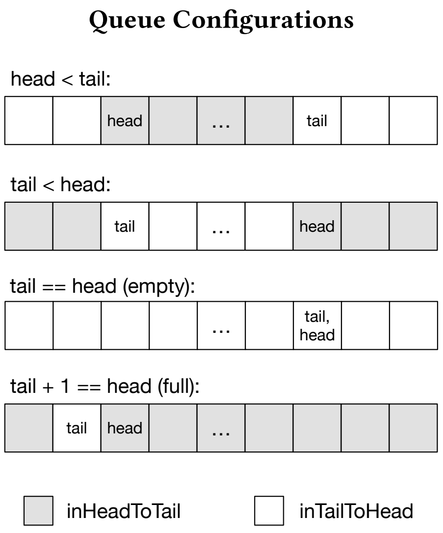


Fig. 8. A Concurrent Queue supporting a single enqueuing thread and single dequeuing thread.

We present the full implementation in Figure 8. A `Queue` includes an `elems` array (line 204) storing `Pair` objects. (The type of `elems` is specified using the array type abbreviation `ElemsType` introduced by the **array** declaration on line 201.) The `Queue` also has `tail` and `head` indices (lines 206 and 210) that are always within the bounds of `elems`. This requirement is captured by the invariant on line 214, which uses the following abbreviations for readability:

$$\begin{array}{rcl} \mathit{Len} & \equiv & \texttt{this.elems.length} \\ \mathit{RepInv} & \equiv & \texttt{this.elems != null \&\& 0 <= this.head <} \ \mathit{Len} \ \texttt{\&\& 0 <= this.tail <} \ \mathit{Len} \end{array}$$

New items are added at `tail` and removed from `head`. Thus, all indices conceptually falling in the range from `head` and `tail` hold stored values, and all other indices are empty, although it is slightly subtle to describe this range precisely because of the way in which the queue values may wrap back around to the start of the array. The Queue Configurations in Figure 8 illustrate the various cases, which we capture in the following two predicates over array indices i:

$$\begin{array}{rcll} \mathit{inHeadToTail}(i) & \equiv & \texttt{this.head <= this.tail ?} & \texttt{this.head <=} \ i \ \texttt{< this.tail} \\ & & & \texttt{:} \ \texttt{this.head <=} \ i \ \texttt{<} \ \mathit{Len} \ \texttt{||} \ \texttt{0 <=} \ i \ \texttt{< this.tail} \\ \mathit{inTailToHead}(i) & \equiv & \texttt{this.tail < \ this.head ?} & \texttt{this.tail <=} \ i \ \texttt{< this.head} \\ & & & \texttt{:} \ \texttt{this.tail <=} \ i \ \texttt{<} \ \mathit{Len} \ \texttt{||} \ \texttt{0 <=} \ i \ \texttt{< this.head} \end{array}$$

We specify the behavior of `Queue` with the ghost field `spec` (line 217). Ghost fields are present only at verification time and not at run time, They enable us to express specifications using data abstraction. Specifically, `spec` represents the current state of the queue as sequence of elements.[2] The invariants on lines 218–219 state that `spec` matches the concrete queue representation.

Ignoring concurrency for the moment, the `enqueue` method checks that the queue is not already full before proceeding to add the new value `x` at `elems[tail]` and incrementing `tail`. Line 237 additionally appends `x` onto the abstract sequence `spec`. Since that line contains no accesses to normal, non-ghost, memory locations, it is ignored during reduction. The method specification (lines 230–231) describes behavior in terms of the update to `spec`. The `dequeue` method is similar.

`Queue` supports safe concurrent accesses, provided there is only ever one enqueuing thread and one dequeuing thread. To model that requirement we require that only Thread 0 calls `enqueue` (line 228), and that only Thread 1 calls `dequeue` (line 248). We define the synchronization discipline for `Queue` for those two threads:

- `elems`: Thread 1 can access the portion of the array storing enqueued values, and Thread 0 can access the unused portion of the array. All other accesses are errors.
- `tail`: Since only Thread 0 may write to `tail`, reads by Thread 0 are both-movers and reads by other threads are non-movers. Thread 0 can update `tail` to index `newValue` only if that value is within the currently unused portion of the array, *i.e.* $\mathit{inTailToHead}$(`newValue`). Thus, Thread 0 can update `tail` to make the queue be "more full", but not "less full".
- `head`: Similarly, Thread 1 can update `head` to a `newValue` such that $\mathit{inHeadToTail}$(`newValue`) or `newValue == this.tail`, thereby making the queue "less full".

That synchronization discipline guarantees race freedom and P/C equivalence with only two **yield**s, at lines 234 and 250. Moreover, the **yields_as** annotations for `tail` (line 208) and `head` (line 212) capture key properties about what each thread may observe at yield points. For example, the **yields_as** annotation for `tail` indicates that, assuming the representation invariant RepInv holds, other threads will only change `tail` to point to another element in the unused portion of the array. That is, when a thread reaches a **yield**, it may observe a change in `tail`, but the new value will always reflect a fuller queue than the old value. Thus, in `dequeue`, if the queue is not empty before

[2] In ANCHOR, ghost values of type `Seq<T>` are immutable sequences of `T` elements, where `[ ]` is the empty sequence and `[x]` is a single-element sequence. Sequences support concatenation (s_1 `++` s_2), length (`s.length`), and subsequence `s[start..end]` operations. Sequences may only be used in specifications, not actual code.

the **yield**, it is not empty afterwards. In addition, reading head is a both-mover and reading tail is a non-mover, resulting in line 249 having an overall commutativity of N. All accesses after the **yield** are both-movers, except for the write to head, which is a non-mover, resulting in lines 251 and 252 having commutativities of B and N, respectively. Similar reasoning guarantees that a non-full queue will never become full at the **yield** in enqueue, and that enqueue is P/C equivalent.

Implementing the Queue, designing the appropriated specification, and verifying the code took roughly one person-day of time. Anchor specifications increased code size from 35 to 62 lines, and Anchor verifies the code in 13.4 seconds. The Java code generated from the Anchor implementation has performance indistinguishable from a Java implementation written by hand.

Comparison to Armada. Armada was used to verify a similar specification for this queue. To prove a program correct in Armada, one starts with the initial program and then writes a series of programs, each simpler than the previous, until we reach a program serving as the high-level specification of the first. The programmer must then prove that each program is a refinement of the previous, using either builtin proof strategies or custom strategies provided by the programmer.

Writing the proofs for the queue took about 6 person-days of work [Lorch et al. 2020a], about 6x as long as we spent using Anchor to verify a similar correctness property. The original implementation was 70 lines of code, and the the final layer and necessary proofs comprised an additional 694 lines. This total of 764 lines is about 12x greater than the size of the code and specification for the Anchor version. Moreover, the computation time required for the Armada verifier [Lorch et al. 2020b] to verify this example on our test machine was about 160x the time required when using Anchor. Excluding verification of the builtin proof strategies distributed with Armada reduces the overall time, but only to about 150x the Anchor time.

Armada supports a number of powerful features absent in Anchor, including the ability to specify multiple layers of refinement, reason about behavior under the TSO relaxed memory model, and extend the system with additional proof techniques. Those features can be highly valuable and enable proof strategies beyond Anchor's capabilities, but they clearly come at a cost.

10 CONCLUSION

Reduction is an effective technique for concurrent program verification. Reduction proofs are fundamentally about movers that specify how heap operations commute over steps of other threads. In Anchor, movers play not just their their traditional role in reduction-based verification, but also a primary role in synchronization specifications. Indeed, conditional movers are the basis of both the programmer's conceptual model of correctness and the verifier's SMT-based reasoning, and are also used to communicate from the programmer to the verifier (via synchronization specifications) as well as from the verifier to the programmer (via error messages). This approach appears to work well in practice. Conditional mover synchronization specifications are expressive and concise. They enable a modular methodology for interleaved development and verification (Section 9), and they have enabled verification of several sophisticated concurrent algorithms with moderate effort.

ACKNOWLEDGMENTS

We thank James Wilcox, Shaz Qadeer, and the anonymous reviewers for feedback on earlier drafts of this paper, and also Jay Lorch for his assistance with Armada. This paper is based upon work supported by the National Science Foundation under Grants 1812951 and 1813133.

REFERENCES

Ralph-Johan Back. 1989. A Method for Refining Atomicity in Parallel Algorithms. In *PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages*. 199–216.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (FMCO)*. 364–387.

Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Inf.* 9 (1977), 1–21.

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *IFM (Lecture Notes in Computer Science)*, Vol. 10510. Springer, 102–110. https://link.springer.com/chapter/10.1007/978-3-319-66845-1_7

Stephen Brookes. 2007. A semantics for concurrent separation logic. *TCS* 375, 1-3 (2007), 227–270.

Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *OSDI*. 306–322.

A. T. Chamillard, Lori A. Clarke, and George S. Avrunin. 1996. *An Empirical Comparison of Static Concurrency Analysis Techniques*. Technical Report 96-084. Department of Computer Science, University of Massachusetts at Amherst.

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48–64.

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*. 23–42.

Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *CONCUR (Lecture Notes in Computer Science)*, Davide Sangiorgi and Robert de Simone (Eds.), Vol. 1466. Springer, 317–331.

Coq 2019. The Coq Proof Assistant. https://coq.inria.fr/

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

Tayfun Elmas. 2010. QED: a proof system based on reduction and abstraction for the static verification of concurrent software. In *ICSE*. 507–508.

Azadeh Farzan and Anthony Vandikas. 2020. Reductions for safety proofs. *Proc. ACM Program. Lang.* 4, POPL (2020), 13:1–13:28.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. 315–327.

Cormac Flanagan and Stephen N. Freund. 2010. FastTrack: efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010), 93–101.

Cormac Flanagan and Stephen N. Freund. 2020. Software Artifact for Article "The Anchor Verifier for Blocking and Non-Blocking Concurrent Software". https://doi.org/10.5281/zenodo.4032624

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4 (2008), 20:1–20:53.

Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2002. Thread-Modular Verification for Shared-Memory Programs. In *ESOP*. 262–277.

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. 2005. Modular verification of multithreaded programs. *TCS* 338, 1-3 (2005), 153–183.

Cormac Flanagan, Stephen N. Freund, and James R. Wilcox. 2018. VerifiedFT CIVL Implementation. https://github.com/boogie-org/boogie/blob/3b7fc31f4ef3f8efc70c812e374c01384509b7f2/Test/civl/verified-ft.bpl

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. 237–247.

Stephen N. Freund and Shaz Qadeer. 2004. Checking Concise Specifications for Multithreaded Software. *Journal of Object Technology* 3, 6 (2004), 81–101.

Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *POPL*. 174–186.

Patrice Godefroid and Pierre Wolper. 1991. A Partial Approach to Model Checking. In *LICS*. 406–415.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. 646–661.

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*. 300–314.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*. 449–465.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. 2005. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*. 3–16.

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

C. A. R. Hoare. 1974. Monitors: An Operating System Structuring Concept. *CACM* 17, 10 (1974), 549–557.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

Thomas W. Doeppner Jr. 1977. Parallel Program Correctness Through Refinement. In *POPL*. 155–169.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.

Leslie Lamport. 1983. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.* 5, 2 (1983), 190–222.

Leslie Lamport and Fred B. Schneider. 1989. *Pretending Atomicity.* Research Report 44. DEC Systems Research Center.

Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. 2011. The Boogie Verification Debugger (Tool Paper). In *Software Engineering and Formal Methods (SEFM)*. 407–414.

Doug Lea. 2019. Concurrency JSR-166. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (Dakar) (Lecture Notes in Computer Science)*, Vol. 6355. Springer, 348–370.

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*. 195–222.

LibLFDS 2019. LFDS 7.11 queue implementation. https://github.com/liblfds/liblfds7.1.1/tree/master/liblfds7.1.1/liblfds711/src/lfds711_queue_bounded_singleproducer_singleconsumer

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020a. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*. ACM, 197–210.

Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020b. Replication Package for Article "Armada: Low-Effort Verification of High-Performance Concurrent Programs". https://doi.org/10.1145/3395653

Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. 73–82.

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504.

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. 267–275.

Jayadev Misra. 2001. *A Discipline of Multiprogramming - Programming Theory for Distributed Applications.* Springer.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*.

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. 49–67.

Doron A. Peled. 1994. Combining Partial Order Reductions with On-the-fly Model-Checking. In *CAV*. 377–390.

Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-Passing Style. In *LISP and Functional Programming*. ACM, 288–298.

R. K. Treiber. 1986. *Systems Programming: Coping With Parallelism.* Technical Report RJ 5118. IBM Almaden.

Liqiang Wang and Scott D. Stoller. 2005. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*. 61–71.

James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: a verified, high-performance precise dynamic race detector. In *PPOPP*. 354–367.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *CAV*. 59–79.

Eran Yahav. 2001. Verifying Safety Properties of Concurrent Java Programs using 3-valued Logic. In *POPL*. 27–40.

Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. 2012. Cooperative types for controlling thread interference in Java. In *ISSTA*. 232–242.