The ANCHOR Verifier for Blocking and Non-Blocking Concurrent Software

ANONYMOUS AUTHOR(S)

Verifying the correctness of performance critical concurrent software with subtle synchronization is notoriously challenging, and many existing verifiers are either inadequately expressive or excessively burdensome to use. We present the ANCHOR verifier, which is based on a new, expressive formalism for specifying synchronization disciplines that naturally matches how programmers should think about reduction-based correctness arguments. ANCHOR proves concurrent components are race-free; do not go wrong; respect their synchronization disciplines; and are preemptive/cooperative equivalent, meaning that thread interference only happens at explicit `yield` annotations. Experiments on textbook concurrent data structures and the FASTTRACK analysis demonstrate that ANCHOR significantly reduces the burden of concurrent verification.

1 INTRODUCTION

Building maximally performant concurrent systems necessitates using a wide variety of synchronization disciplines, ranging from locks, read-only data structures, and thread-local exclusive access to sophisticated non-blocking algorithms based on atomic compare-and-swap. Additionally, synchronization disciplines may vary over time, depending on the values of flags or other data structures, may provide different access permissions to reads and writes, and may restrict permissible writes to, for example, enforce monotonically increasing updates. 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. 2015a; Jung et al. 2015; Leino et al. 2009; Lorch et al. 2020,?; O'Hearn 2004; Xu et al. 2016; Yi et al. 2012]. However, using such tools remains a daunting task in practice. Some tools are limited in the types of synchronization they can effectively reason about, while others present programming models far from typical programming languages or require much time and effort to write a verifiable specification or proof script. Diagnosing verification failures and addressing them is also difficult if the verifier is not carefully designed to provide meaningful feedback.

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 kinds of often-sophisticated synchronization mechanisms used in concurrent software.
(2) How to *verify* that each thread in a system behaves correctly, under the assumption that interleaved operations of other threads obey the documented synchronization discipline.

Clearly, 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, intuitive, and expressive synchronization specifications.

ANCHOR utilizes Lipton's theory of reduction [Lipton 1975], which employs a commuting argument to verify that thread interference can be safely assumed to occur only at `yield` annotations. ANCHOR's reduction analysis is enabled by synchronization specifications documenting the commutivity of each heap access as one of Left-moving, Right-moving, Both-moving, Non-moving,

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

or Error (for prohibited accesses). For example, a lock acquire operation right-commutes across a subsequent operation of a different thread; a race-free memory access both right- and left-commutes across preceding operations of other threads; and accessing a lock protected memory location without holding the lock is an Error. The commutivity of a memory access may depend on a variety of conditions, including which locks are held, whether the location is thread-local, whether the access is a read or a write, whether a particular flag is set to true, and so on. To support these dependencies Anchor supports conditional synchronization disciplines of the form

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

Anchor's synchronization specifications are more expressive than prior tools (*e.g.* [Flanagan et al. 2008, 2002; Freund and Qadeer 2004; Hawblitzel et al. 2015a,b; Yi et al. 2012]) in that they support conditional synchronization disciplines and they explicitly specify how each access commutes with concurrent operations by other threads. This expressiveness enables Anchor to

- check synchronization specifications for self-consistency properties even before the first line of implementation code is written;
- verify more complex algorithms;
- provide a better intuitive model of which code fragments are reducible; and
- provide better diagnostic error messages to the programmer.

The Anchor guarantees that thread interference is only observable at `yield` annotations. 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 can be seen as an extension of atomicity supporting complex synchronization with intentional thread interference [Yi et al. 2012].

Anchor supports verification of `cas`-based non-blocking algorithms, such as the `cas`-based atomic counter shown in Figure 2. For that counter, Anchor can infer that a read from `n` followed by a successful `cas` is reducible, because knowing that `n` is free of ABA problems [Michael 2004] rules out the possibility of intervening concurrent writes. This `cas` support complicates our formal development and implementation but is critical for verifying high-performance non-blocking algorithms that rely on ABA-freedom arguments for correctness.

Our correctness argument for Anchor is formalized as a stack of three operational semantics for an idealized subset of Java. The first is a *standard semantics* formalizing the execution behavior programs. The second is an *instrumented semantics* extending the standard semantics to check that the synchronization discipline is respected and that each yield-free code fragment is reducible. The final is a *thread-modular semantics* that executes one thread of the instrumented semantics in isolation, using the synchronization specifications to model possible behaviors of interleaved threats at yield points, which enables Anchor to scale to an unbounded number of threads. These three semantics are related by appropriate notions of simulation.

The Anchor implementation 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

- does not go wrong under the standard preemptive semantics,
- is free of data race conditions and hence exhibits sequentially-consistent behavior,

- satisfies it synchronization specification,
- satisfies its method specifications, and
- is P/C equivalent.

We have evaluated ANCHOR on a variety of examples ranging from textbook data structures and algorithms [Herlihy and Shavit 2008] to the FASTTRACK race detection algorithm [Flanagan and Freund 2010; Wilcox et al. 2018]. Our experience shows that ANCHOR is readily able to verify programs using subtle and complex synchronization disciplines without an excessive specification or proof burden. For example, verifying P/C equivalence for the complex FASTTRACK algorithm required adding about 50 lines of specification to 220 lines of implementation code. Verifying functional correctness required adding about 150 lines of specification to those 220 lines. Prior work verifying those functional requirements in CIVL necessitated adding over 500 lines of specification to 250 lines of CIVL code [Flanagan et al. 2017; Wilcox et al. 2018].

In summary, the contributions of this paper include:

- A concise and expressive notation for synchronization specifications that describes how memory accesses 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 that uses synchronization specifications to perform reduction and thread-modular reasoning to verify concurrent code with sophisticated synchronization and an unbounded number of threads. (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 demonstrating its effectiveness on a variety of concurrent algorithms. (Section 9)

2 BACKGROUND AND EXAMPLES

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

- 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 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 other threads. For example, a lock acquire followed by a race-free access and a lock release is reducible. ANCHOR

```
1  class SyncSpecs {
2    volatile int x   moves_as N;
3    volatile int y   moves_as B;  // <-- invalid
4    int z            moves_as holds(this) ? B : E;
5
6    public void double() {
7  R   acquire(this);
8  B   int t = this.z;
9  B   this.z = 2 * t;
10 L   release(this);
11   }
12 }
```

Fig. 1. ANCHOR synchronization specification examples.

uses this theory to show that each method is P/C equivalent by verifying that all execution paths consist of reducible sequences separated by yields.

Synchronization specifications. To leverage Lipton's theory, ANCHOR relies on a synchronization specification that describes when each thread can access each memory location and how each access commutes with steps from other threads. In addition to the four movers listed above, we use the error-mover (E) to denote accesses are not permitted. Thus, our language of movers, and their partial ordering based on decreasing commutivity.

$$m \in \mathit{Mover} ::= \mathsf{B} \mid \mathsf{L} \mid \mathsf{R} \mid \mathsf{N} \mid \mathsf{E}$$
$$\mathsf{B} \sqsubset \{\mathsf{L}, \mathsf{R}\} \sqsubset \mathsf{N} \sqsubset \mathsf{E}$$

These movers are the basic building blocks of our conditional synchronization disciplines:

$$\begin{array}{rcl} \mathit{SyncDiscipline} & ::= & m \\ & \mid & \mathit{expr}\ ?\ \mathit{SyncDiscipline}\ :\ \mathit{SyncDiscipline} \end{array}$$

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

- `this`: The object being accessed.
- `tid`: The unique 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(`l`)`: Whether lock for object l is held, where l is a path expression (*i.e.* a local variable optionally followed by field or array accesses, such as `x`, `x.f`, `a[i]`, or `y.f.g`).
- `isLocal(`l`)`: Whether l is local to the current thread.
- `isShared(`l`)`: Whether l is potentially shared.

Figure 1 illustrates these synchronization specifications. The `moves_as` specification on the volatile field `x` on line 3 indicates that `x` can be accessed by any thread at any time, and that all accesses are thus non-movers (N), as they may not commute with concurrent accesses. ANCHOR requires any fields exhibiting N, R, or L commutivity must be declared `volatile` to ensure race freedom and sequentially-consistent behavior.

```
13 class AtomicInt {
14   noABA volatile int n  moves_as isRead() ? N
15                                          : (newValue == this.n + 1 ? N : E)
16                         yields_as newValue >= this.n;
17
18   ensures this.n == old(this.n) + 1;
19   public int inc() {
20     while (true) {
21 N/R   int x = this.n;
22 B/N   if (cas(this.n,x,x+1)) { return n+1; }
23       yield;
24     }
25   }
26 }
```

Fig. 2. `AtomicInt` example.

In contrast, the specification on the field `y` claims that unrestricted accesses to `y` are all both-movers (B). This specification is not valid since a write to `y` cannot commute with a concurrent write. ANCHOR detects such errors even before any code using `y` is written.

Field `z` is only accessible by a thread holding the lock of the enclosing object, specified using the `holds(this)` predicate. Such lock-protected accesses are both-movers (B) as there are no concurrent accesses. Accesses without holding the lock are errors (E).

These specifications document the synchronization discipline and also facilitate reduction arguments. For example, method `double()` on line 6 is atomic because it consists of a lock acquire (R), a race-free read (B), a race-free write (B), and a lock release (L). Figure 1 includes these movers to the left of the code for clarity.

ANCHOR supports abbreviations for common synchronization idioms including the following:

$$
\begin{aligned}
\texttt{immutable} &\equiv \texttt{isRead()}\ ?\ \texttt{R} : \texttt{E} \\
\texttt{readonly} &\equiv \texttt{isRead()}\ ?\ \texttt{B} : \texttt{E} \\
\texttt{thread\_local} &\equiv \texttt{isLocal(this)}\ ?\ \texttt{B} : \texttt{E} \\
\texttt{guarded\_by}\ l &\equiv \texttt{holds(}l\texttt{)}\ ?\ \texttt{B} : \texttt{E} \\
\texttt{write\_guarded\_by}\ l &\equiv \texttt{isRead()}\ ?\ (\texttt{holds(}l\texttt{)}\ ?\ \texttt{B} : \texttt{N}) \\
&\qquad\qquad : (\texttt{holds(}l\texttt{)}\ ?\ \texttt{N} : \texttt{E})
\end{aligned}
$$

Writes to immutable fields 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 an immutable field could have changed before reaching its immutable state. The `readonly` abbreviation is for cases when a field's value is fixed from the moment the field becomes accessible. Reads of readonly fields are thus both-movers. For thread-local fields of an object, the allocating thread initially has exclusive access (B). If the object ever becomes shared between threads, subsequent accesses are not allowed (E).

The `guarded_by` abbreviation captures lock-based exclusive access, as for the field `z` on line 4. 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 writes are non-movers (N) since there may be concurrent reads.

`AtomicInt` Example. As an illustration of Anchor's `cas`-based reasoning, consider class `AtomicInt` in Figure 2.The synchronization specification for volatile field `n`

```
moves_as isRead() ? N
                  : (newValue == this.n + 1) ? N : E
```

indicates that the field can be read by any thread at any time (`N`) and that writes (also `N`) must increment `n`. Consequently, the field `n` is ABA-free [Michael 2004], meaning that the field is never changed from a value A to B and then back to A again.

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`. We permit `cas` to fail non-deterministically, meaning that we can treat failed `cas` operations as both-movers. 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 $[\texttt{N};\texttt{B};\texttt{yield}]^{*}\texttt{R};\texttt{N}$ consists of reducible sequences separated by `yield`s 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 concurrent algorithms, including various non-blocking algorithms. It involves verifying ABA-free fields and treating reads from them differently depending on whether they or not they are 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.

`LockBasedStack` and `LockFreeStack` Examples. The `LockBasedStack` class in Figure 3 contains a `head` node with initial value `null`. The `push()` method allocates a new `Node`, initializes its `item` and `next` fields via the constructor, 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 `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.

The alternative `LockFreeStack` implementation in Figure 3 uses a `cas` operation for optimistic concurrency control. Other threads may change `head` between the read and `cas` operation, necessitating the inclusion of a `yield` between those operations. Unlike the `inc` example in Figure 2, a successful `cas` operation does not rule out the possibility that `head` changed after the read, since another thread could call `push` and then `pop`, resulting in `head` being changed and then restored to the original value. Consequently, `head` cannot be declared ABA-free.

We discuss method specifications for `push()` and `pop()` below. Interestingly, despite using different synchronization disciplines, both stack implementations satisfy the same method specifications.

Anchor Error Messages. A key feature of Anchor is its ability to provide understandable and actionable error messages. To this end, Anchor embeds auxiliary information in the generated Boogie code for mapping Boogie errors and counterexamples back to meaningful messages in the original code. We illustrate the types of messages Anchor reports for synchronization and reduction errors below. We have simplified and reformatted some parts of these examples for

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

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

class LockBasedStack {
  Node head moves_as
            isLocal(this) ? B : guarded_by this;

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

  public int pop() {
    while (true) {
R     acquire(this);
B     if (this.head != null) break;
L     release(this);
      yield;
    }
B   int value = this.head.item;
B   this.head = this.head.next;
L   release(this);
    return value;
  }
}
```

```
class LockFreeStack {
  volatile Node head moves_as
                     isLocal(this) ? B : N;

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

  public int pop() {
    while (true) {
N     Node top = this.head;
      if (top != null) {
B       Node next = top.next;
        yield;
B/N     if (cas(this.head, top, next)) {
          return top.item;
        }
      }
      yield;
    }
  }
}
```

Fig. 3. `LockBasedStack` and `LockFreeStack` examples.

clarity and brevity, but they embody the key aspects how ANCHOR reports errors. We first consider the following synchronization error trace reported when the following method is added to the `LockBasedStack` class from Figure 3.

```
public void buggy() {
  this.push(10);
  // must acquire lock here
  assert this.head.item == 10;
}
```

```
    call this.push(10)
[R]   acquire(this);
      ...
[L]   release(this);
[E] tmp1 = this.head; [!isLocal(this) && !holds(this) => E]
```

The trace contains the commutivity of each synchronization and memory access, as well as how that commutivity was computed. Such traces are often sufficient to immediately identify a synchronization specification violation, as indicated by the error commutivity `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() {
  this.push(10);
  // yield necessary here
  acquire(this);
  assert this.head.item == 10;
  release(this);
}
```

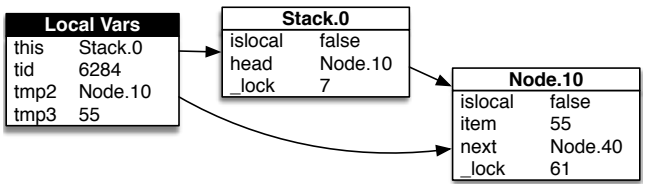
```
    call this.push(10)
[R]   acquire(this);
      node = new Node();
[B]   tmp = this.head;  [!isLocal(this) && holds(this) => B]
[B]   node.item = 10;   [isLocal(node) => B]
[B]   node.next = tmp;  [isLocal(node) => B]
[B]   this.head = node; [!isLocal(this) && holds(this) => B]
[L]   release(this);
[R] acquire(this);   // R cannot follow L in reducible trace
```

As before, the reduction error message allows us to readily identify either a missing `yield` or a situation in which the synchronization specification fails to ensure the necessary commuting properties. Once the missing `yield` is inserted, ANCHOR reports that the assertion may fail, again providing a failing trace:

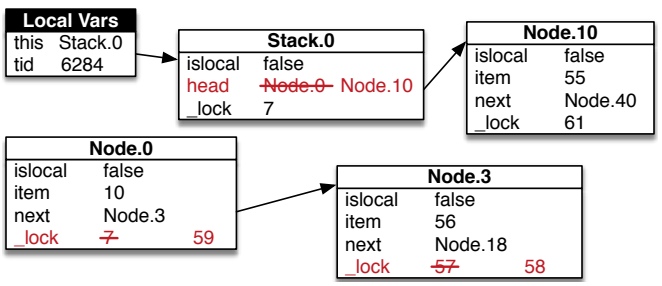
```
public void buggy() {
  this.push(10);
  yield; // added
  acquire(this);
  assert this.head.item == 10;
  release(this);
}
```

```
    call this.push(10)
[R]   acquire(this);
[B]   ... as before ...
[L]   release(this);
=== yield;
[R] acquire(this);
[B] tmp2 = this.head; [!isLocal(this) && holds(this) => B]
[B] tmp3 = tmp2.item; [!isLocal(tmp2) && isRead() => B]
    assert tmp3 == 10;
```

As in some other verification tools [Le Goues et al. 2011], ANCHOR provides a way to visually 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.10` includes its `item` and `next` fields, whether it is thread-local, and the `tid` of the thread holding its lock (if any). From this snapshot, we see that the `head` node stores 55 instead of the expected value 10. Inspecting the snapshot for the earlier `yield` indicates that this is where the change occurred.



```
147 class VarState {
148   volatile int readEpoch  moves_as (this.readEpoch != SHARED) ? write_guarded_by this
149                                                               : immutable;
150   int[moves_as (this.readEpoch != SHARED) ? guarded_by this
151                                           : isRead() ? (holds(this) || tid == index ? B:E)
152                                                      : (holds(this) && tid == index ? B:E)]
153     vc moves_as (this.readEpoch != SHARED) ? guarded_by this : write_guarded_by this;
154 }
```

Fig. 4. FASTTRACK `VarState` synchronization specification.

That snapshot shows the pre-yield state and any changes that occurred during the `yield`. Some changes are irrelevant, such as for lock holders, but we can also see that `this.head` was changed to object `Node.10` instead of `Node.0` during the `yield`. Snapshots like these, coupled with the detailed traces, have proven effective for debugging errors both in specifications and in code.

FASTTRACK `VarState` Synchronization Discipline. We conclude this section with the class `VarState` in Figure 4 that shows a more involved synchronization discipline derived from the FASTTRACK dynamic data race detector [Flanagan and Freund 2010; Wilcox et al. 2018]. The field `readEpoch` is initially `write_guarded_by this`, as shown on line 147, but becomes `immutable` once it is set to the special value `SHARED`. The array declaration on lines 149–152 has the form

`int[moves_as` *SyncDiscipline*`] vc moves_as` *SyncDiscipline*`';`

to indicate that the field `vc` is protected by *SyncDiscipline*′ and the array element `vc[index]` is protected by *SyncDiscipline* (which may mention `index`). According to this declaration, the array entry `vc[index]` is initially `guarded_by this` (line 149). 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` (line 150). Writes to `vc[index]` require both holding the lock *and* having `tid == index` (line 151). Under this discipline, all permitted accesses to `vc[index]` are both-movers.

These complex synchronization specifications are necessary to verify the reducibility of the fast-path code from the core FASTTRACK algorithm, shown below, which elides locking to maximize performance.

```
     public boolean read(VarState sx) {
B      int e = ...;  // e cannot be SHARED

       // fast path:
N/R    int r = sx.readEpoch;
       if (r == e) return true;
       if (r == SHARED &&
N          sx.vc[tid] == e) return true;
       yield;
       // slow path:
R      acquire(sx);
N      ...
L      release(sx);
     }
```

If `sx.readEpoch` is not `SHARED`, the fast-path code is clearly reducible since it only reads `sx.readEpoch` (`N`). 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 152), and reads `vc[tid]` (`B` because `tid` is the index being accessed). Thus, this sequence is also reducible.

3 RELATED WORK

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 a 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 can support method calls via inlining, thus avoiding the burden and complexity of specifying all methods. Most of our benchmarks could not be verified with Calvin-R.

Anchor is inspired by CIVL [Hawblitzel et al. 2015a] (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.

Like Anchor, Armada [Lorch et al. 2020] also focuses on SMT-based verification of concurrent programs via a combination of reduction and thread-modular reasoning. Armada also supports additional strategies such as data abstraction, which facilitates simpler (higher-level) method specifications. A key difference between Armada and Anchor is our emphasis on formally specifying synchronization disciplines, checking them for consistency and appropriate commuting properties 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 several benefits, including a better documentation, more modular development methodology (outlined in Section 9), more understandable errors messages, and potentially better scalability. In addition, Armada does not include the built-in support for `cas` that Anchor does, including verification of ABA-free fields and automatically upgrading any read followed by a successful matching `cas` to be a right-mover. One avenue for future work is to leverage our expressive synchronization specifications in tools like Armada that support data abstraction.

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 multi-threaded layers of software and for verifying the correctness of software components. It focuses on rely-guarantee reasoning rather than 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.

Azedah and Vandikas [Farzan and Vandikas 2019] recently extended these ideas to encompass a search for a sound reduction strategy, thus enabling automatic safety proofs of complex systems for which non-reduction proofs are not feasible. Anchor currently assumes that reducible code blocks are explicitly delimited with `yield` statements, but incorporating a search for optimal yields is an interesting topic for future work.

A large body of prior work has addressed the important problem if the current software verification. 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 also been used to limit state-space explosion while model checking concurrent programs. Type-based techniques [Abadi et al. 2006; Agarwal and Stoller 2004; Boyapati and Rinard 2001; Grossman 2003] have proven useful for systems with simpler synchronization disciplines than those studied here. Dynamic concurrency analyses [Christiaens and Bosschere 2001; Flanagan and Freund 2010; Nishiyama 2004; Pozniansky and Schuster 2007; Savage et al. 1997; Schonberg 1989; von Praun and Gross 2001; Wilcox et al. 2018; Yu et al. 2005,?] are also very useful in practice but they do not provide ANCHOR's strong correctness guarantees. The notion of atomicity is a useful and concise method specification [Flanagan and Freund 2004], but it is often not applicable to concurrent software with intentional thread interference. P/C equivalence is a natural extension of atomicity that uses yields to document intentional interference between threads [Adya et al. 2002; Amadio and Zilio 2004; Boudol 2007; Cerný et al. 2017; Yi et al. 2015].

4 THE ANCHORJAVA CONCURRENT LANGUAGE: SYNTAX AND SEMANTICS

We formalize ANCHOR's verification methodology for the idealized concurrent language ANCHORJAVA shown in Figure 5. Each class definition includes field and method declarations. Each method declaration $mn(\overline{x})$ { s; `return` z } includes a unique method name mn, formal parameters $\overline{x}$, and a body s followed by a return statement. (We discuss method specifications in Section 8 below.) 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; ?].

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 include 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-set 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-]. We identify certain locations $\rho.f$ as being *ABA-free*, which 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 ($s =$ `skip`). Thus, the 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.

Syntax:

$$
\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & x = y.f \mid y.f = x \\
& \mid & z = \mathsf{cas}(y.f, x, w) \\
& \mid & x = e \mid x = \mathsf{new}\ c \\
& \mid & \mathsf{acq}(x) \mid \mathsf{rel}(x) \mid \mathtt{yield} \\
& \mid & z = y.mn(\overline{x}) \mid \mathtt{while}\ e\ s \\
& \mid & \mathtt{if}\ e\ s\ s \mid s; s \mid \mathtt{skip} \mid \mathtt{wrong} \\
\mathit{meth} \in \mathit{Method} & ::= & mn(\overline{x})\ \{\ s; \mathtt{return}\ z\ \} \\
D \in \quad \mathit{Defn} & ::= & \mathtt{class}\ c\ \{\ \overline{f}\ \overline{\mathit{meth}}\ \}
\end{array}
$$

$w, x, y, z \in \mathit{Var} \qquad f \in \mathit{FieldName}$

$e \in \mathit{Expr} \qquad mn \in \mathit{MethodName}$

$c \in \mathit{ClassName}$

Standard State:

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

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

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

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

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

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

[Std Write]
$$\frac{\sigma(y) = \rho \qquad \sigma(x) = v \qquad \text{if } \rho.f \text{ is ABA-free then } H(\rho.f) \sqsubseteq_{\rho.f} v}{(y.f = x \cdot \sigma) \cdot H \to_t (\mathtt{skip} \cdot \sigma) \cdot H[\rho.f := v]}$$

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

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

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

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

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

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

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

$\boxed{\Sigma \to \Sigma'}$ [Std Preemptive]
$$\frac{\Sigma \to_t \Sigma'}{\Sigma \to \Sigma'}$$

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

Fig. 5. AnchorJava syntax and standard semantics.

5 SYNCHRONIZATION SPECIFICATIONS

As shown in Section 2, the synchronization discipline for a field access $\rho.f$ can depend on the `tid` of the accessing thread and on the state of other data or locks in the heap. As such, we formalize the read synchronization discipline[1] for each location $\rho.f$ as a function $R_{\rho.f}$:

$$R_{\rho.f} \quad \in \quad \mathit{ReadMover} \quad = \quad \mathit{Tid} \times \mathit{Heap} \rightarrow \mathit{Mover}$$

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 or that the new value is always a freshly-allocated object). Thus, the function $W_{\rho.f}$ for $\rho.f$'s write synchronization discipline also takes the value being written:

$$W_{\rho.f} \quad \in \quad \mathit{WriteMover} \quad = \quad \mathit{Tid} \times \mathit{Heap} \times \mathit{Value} \rightarrow \mathit{Mover}$$

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. The erroneous synchronization specification on line 3 of Figure 1, where $R_{\rho.y}(t, H) = W_{\rho.y}(u, H, v) = \mathsf{B}$ is prohibited by this rule.

Similarly, the second 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, respectively, and prohibit conflicting accesses in the pre-state.

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

$$\begin{array}{lrcl}
\text{[VALID READ-R]} & R_{\rho.f}(t, H) \sqsubseteq \mathsf{R} & \Rightarrow & W_{\rho.f}(u, H, _) = \mathsf{E} \\
\text{[VALID WRITE-R]} & W_{\rho.f}(t, H, v) \sqsubseteq \mathsf{R} & \Rightarrow & W_{\rho.f}(u, H[\rho.f := v], _) = \mathsf{E} \ \wedge \ R_{\rho.f}(u, H[\rho.f := v]) = \mathsf{E} \\
\text{[VALID READ-L]} & R_{\rho.f}(t, H) \sqsubseteq \mathsf{L} & \Rightarrow & W_{\rho.f}(u, H[\rho.f := _], H(l)) = \mathsf{E} \\
\text{[VALID WRITE-L]} & W_{\rho.f}(t, H, v) \sqsubseteq \mathsf{L} & \Rightarrow & W_{\rho.f}(u, H[\rho.f := _], H(l)) = \mathsf{E} \ \wedge \ R_{\rho.f}(u, H) = \mathsf{E}
\end{array}$$

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 $u \neq t$:

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

This strict stability condition works well for most programs. To support more subtle synchronization disciplines, ANCHOR actually 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.

[1] For technical convenience, we formalize the read and write synchronization disciplines separately.

Instrumented State:

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

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

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

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

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

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

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

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

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

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

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

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

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

[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 check 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.

The instrumented semantics 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 block. The operation

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

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 $P : \mathit{Tid} \rightarrow \mathit{Phase}$ 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.

The new rule [INST ACQ] extends [STD ACQ] to update the phase from p to $p \triangleright \mathsf{R}$, reflecting that the acquire is a right-mover. Rule [INST YIELD] resets the phase to PRE, thus initiating a separate new reducible sequence.

As illustrated by `inc()` in Figure 2, the commutivity 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:

$$\mathit{CASable}_{\rho.f}(t, H, C) \stackrel{\text{def}}{=} \rho.f \text{ is ABA-free } \wedge\ (\rho.f, t, H(\rho.f)) \in C_{\rho.f}\ \wedge\ R_{\rho.f}(t, H) \neq \mathsf{E}$$

Rule [INST READ] uses this predicate to upgrade the commutivity m of a read from $R_{\rho.f}(t, H)$ to R where appropriate.

Rule [INST WRITE] extends [STD WRITE] to update the thread's phase to $p \triangleright m$ 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:

$$\mathit{ValidWrite}_{\rho.f}(C, v) \stackrel{\text{def}}{=} \rho.f \text{ is ABA-free } \Rightarrow\ \forall(\rho.f, \_, v') \in C.\ v \sqsubseteq_{\rho.f} v'$$

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 $\sigma = [y \mapsto \rho, w \mapsto 2]$:

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

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 commutivity properties for our correctness argument below, this rule needs to left-commute across concurrent writes to $\rho.f$. Therefore we compute the commutivity $m = W_{\rho.f}(t, H[\rho.f := \sigma(x)], v)$ 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` sequence 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.

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

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 needs to 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 properly ordered with respect to the current values in H:

$$C \textit{ is valid for } H \quad \text{iff} \quad \forall \rho.f, v.\ \rho.f \text{ is ABA-free} \ \wedge\ \forall (\rho.f, \_, v) \in C \ \Rightarrow\ H(\rho.f) \sqsubseteq_{\rho.f} v$$

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.

Consequently, if the standard semantics can go wrong then so does the instrumented 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, any cooperative run of the instrumented semantics is also cooperative under the standard semantics.

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

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:

Thread-Modular State:

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

$\boxed{\Theta \Mapsto_t \Theta'}$

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

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

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

where $\mathit{Yield}(t, H, H') = \forall x.\ H(x) \neq H'(x) \Rightarrow (\exists u \neq t.\ W_x(u, H, H'(x)) \neq \mathsf{E})$
$\wedge\ \forall l.\ H(l) = t \Leftrightarrow H'(l) = t$

Fig. 7. ANCHORJAVA thread-modular semantics.

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

7 THREAD-MODULAR SEMANTICS

Our final semantics is a *thread-modular* semantics that provides the foundation for our verification methodology, 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 $\Theta \Mapsto_t \Theta'$ 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 some thread $u \neq t$ has permission to modify it in H, 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:

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

Given $\Pi_0 = T_0 \cdot H_0 \cdot C_0 \cdot P_0$, running $\Pi_0|t$ under $\Mapsto_t$ simulates (over-approximates) the behavior of thread t under $\Mapsto$ and so detects if t would go wrong.

Theorem 4 (Simulation). Suppose $\Pi_0 \Rightarrow^* \Pi$ where thread t has gone wrong in Π. Then $\Pi_0|t$ goes wrong under $\Mapsto_t$.

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 thread-modular program $\Pi_0|t$, where $\Pi_0 = T_0 \cdot H_0 \cdot C_0 \cdot P_0$.

Theorem 5 (Correctness). Suppose that for all t, $\Pi_0|t$ does not go wrong under $\Mapsto_t$. Then $T_0 \cdot H_0$ does not go wrong under $\longrightarrow$ and is P/C-equivalent.

Proof. By Theorem 4, Π_0 does not go wrong under $\models\!\!\Rightarrow$. By Theorem 3(1), Π_0 does not go wrong under $\Rightarrow$. 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), $T_0 \cdot H_0 \cdot C_0 \cdot P_0 \models\!\!\Rightarrow^* T \cdot H \cdot \emptyset \cdot P$. 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 and subtyping, but is sufficient for writing many concurrent algorithms. 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`.

Given a target program, Anchor verifies that no assertions fail; that the synchronization specification is valid, stable, and respected by the code; and that the code is race-free and P/C equivalent. If the code contains method specifications, those specifications are also verified. 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. That program is checked with the Boogie verifier [Barnett et al. 2005], and any errors are mapped back to the Anchor source code and reported along with details extracted from Boogie's counter-examples, as shown in Section 2.

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. 2020], and CIVL [Hawblitzel et al. 2015a]. The specification burden of using Anchor is also lower than it would otherwise be, and since concurrent components are typically relatively small, we do not expect this to become a limiting factor.

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 the receiver and all objects passed as parameters to public methods are shared. However, the receiver of a call to public constructor is considered to be local as we expect clients to call those methods only on newly-allocated thread-local objects. One area for future work is to integrate a more robust analysis for ownership [Clarke et al. 1998].

Modeling Field Updates by Other Threads. Anchor includes an additional annotation to aid in modeling state changes at `yield`s. The thread-modular semantics models the effect of a `yield` as zero or more heap updates via the *Yield* relation. That relation, in turn, is defined in terms of each memory location's write access permission. Given the difficulty of computing the transitive closure of *Yield*, we instead over-approximate it by characterizing all possible changes to each location `x.f` in one of two ways.

The first (default) approximation is that if the current thread has right-mover read access to `x.f` at the `yield`, then the value of `x.f` does not change. If the current thread does not have right-mover read access, `x.f` may have any value after the `yield`. This rule subsumes all possible writes permissible by the synchronization specification, and it is reflexively and transitively closed (since our validity/stability requirements ensure other threads cannot modify `x.f` or take away the current thread's right-mover permission).

This rule works for the vast majority of examples we studied, but it may admit changes to `x.f` that are not actually feasible if the possible updates to `x.f` performed by other threads are restricted in some way. For example, consider again `AtomicInt` from Figure 2 and a client of that class:

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

  ensures this.n == old(this.n) + 1;
  public int inc() { ... }
}
...
public void incTwice(AtomicInt counter) {
  int first = counter.inc();
  yield;
  int second = counter.inc();
  assert second > first;
}
```

Ignoring the `yields_as` annotation for the moment, ANCHOR would be unable to prove that the assertion `assert second > first` 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 `yields_as` annotations to approximate heap updates. These annotations explicitly 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 are reflexively/transitively closed.

The related `noABA` annotation further restricts how a field may be changed. It ensures that the field exhibits ABA freedom, namely that that the field is never changed from a value A to B and then back to A again. ANCHOR verifies a field's `noABA` annotation using that `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 $A \not> B$.

POST **Phase Termination and Termination Metrics.** Our correctness argument requires that any reducible code 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, may fail at run time (*e.g.*, due to a null pointer error), 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 [**?**], 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.)

Object Invariants. ANCHOR also supports object invariants. For example, to express that `Nodes` in a linked list must be ordered by their `items`, we can add the following invariant to our `Node` class from Figure 3.

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

```
class LockBasedStack {
  modifies this;
  ensures this.head.next == old(this.head) && this.head.item == item;
  public int push(int item) { ... }

  modifies this;
  ensures old(this.head) != null;
  ensures $result == old(this.head.item) && this.head == old(this.head.next);
  public int pop() { ... }
}
```

Fig. 8. ANCHOR method specifications for `LockBasedStack`.

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.

Method Specifications. ANCHOR additionally supports specifications for `public` methods using via an approach inspired by earlier work on Calvin-R [Freund and Qadeer 2004] and CIVL [Hawblitzel et al. 2015a]. Space limitations prevent a full discussion of ANCHOR's method specifications, but we illustrate the core ideas on 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 8 for the `Stack.push()` method from Figure 3, and ANCHOR verifies that the method body conforms to that specification.

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 `pop` method for `LockBasedStack` has this property, since all but that last `yield`-free region in any execution of that method is side-effect free. This atomic behavior is captured by `pop()`'s specification in Figure 8. The special variable `$result` refers to the method's return value, and terms of the form `old(`e`)`, such as `old(this.head)`, refer to the value of e at the beginning of the `yield`-free region captured by the specification. Note that `old(this.head)` may have a different value than it had at the start of the method call if interference occurs at the documented `yield` point. `LockFreeStack`'s method satisfies the *exact same* specifications as `LockBasedStack`'s, despite using optimistic concurrency control instead of locks.

The behavior of a public method 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 state for 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 in ANCHOR 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. One area for future work is to extend ANCHOR with data abstraction or a layering

Program	Synchronization Idiom	Size (LOC)	Sync. Specs	Yields	`requires`/ Invariants	ANCHOR Time (s)
Stacks (w/ `push`, `pop`, and `init`)						
Coarse	Lock-based (as in Figure 3)	31	3	0	0	3.20
Lock-free	CAS-based (as in Figure 3) [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
FastTrack	[Wilcox et al. 2018]	264	6	2	22	21.89

Table 1. Concurrency examples verified by ANCHOR.

mechanism, as in CIVL [Hawblitzel et al. 2015a], to hide yields in a method that only affect the concrete internal representation of a data structure but not the public abstraction of that structure.

9 EVALUATION

We evaluated ANCHOR on a variety of concurrent collection classes and the FASTTRACK dynamic data race detection algorithm [Flanagan and Freund 2010; Wilcox et al. 2018].

Collection Classes. We begin by verifying synchronization specifications and P/C equivalence on concurrent collection implementations. 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 via an ANCHOR command-line option for assuming no heap changes occur during `yields`.
(3) Verify the code in a multi-threaded setting, in which the heap may change at `yields`.

We found the first two steps quite valuable, enabling us to first debug synchronization specifications without writing any code and then identify and fix as many programming/reduction errors as possible before reasoning about concurrent heap changes.

Table 1 lists the number of non-empty, non-comment lines of code in each component, as well as counts of: fields with synchronization specifications, `yield` annotations, and `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).

Our collection classes are adaptations of Java implementations presented in *The Art of Multiprocessor Programming* [Herlihy and Shavit 2008]. (Table 1 refers to the origins of the techniques employed in the more sophisticated versions.) We include the lock-based and lock-free [Treiber 1986] stack implementations from Figure 3 as well as five linked list implementations that store integers in increasing order with no duplicates. Most of the `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 `yields` indicate that other threads may change the list during traversals.

Optimistic and Lazy Lists adopt optimistic concurrency control mechanisms in which initial traversals performed without locking are followed by a validation step to ensure the absence of conflicts. The Lazy variant also supports a lock-free contains operations. 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 Coarse and Lock-Free Queues follow the same pattern in the number of `yields` required and the verification time. The Bounded Queue uses separate `enqLock` and `deqLock` locks for adding and removing items, enabling those operations to run in parallel. This Bounded Queue has a maximum `capacity` and a current `size` field, which requires a sophisticated synchronization specification. That field is essentially write-guarded by `this`, but it can only be incremented when the `enqLock` is held and `size` is smaller than `capacity`. Similarly, it can only be decremented when the `deqLock` is held and `size` is greater than 0.

```
int size moves_as isRead()
          ? write_guarded_by this
          : (holds(this.enqLock) && newValue==this.size+1 && this.size < this.capacity) ||
             holds(this.deqLock) && newValue==this.size-1 && 0 < this.size) ? write_guarded_by this
                                                                            : E
    yields_as (0 <= this.size && this.size <= this.capacity)
      ==> ((0 <= newValue && newValue <= this.capacity) &&
           (holds(this.enqLock) ==> newValue <= this.size) &&
           (holds(this.deqLock) ==> newValue >= this.size));
```

This specification necessitates a `yields_as` annotation to capture how `size` may change at a yield: if the current value is in the expected range 0..`capacity`, then it remains in that range. Also, if `enqLock` is held, `size` can only get smaller (due to concurrent dequeue operations), and vice-versa for `deqLock`. Anchor time is high due, we believe, to the need to solve verification conditions involving modular arithmetic. Time is also highly variable for this particular program due to the randomized nature of the underlying SMT solver.

Another sophisticated synchronization discipline readily captured by Anchor is the `table` array maintained by the Striped HashSet. That HashSet uses the objects stored in its `locks` array to guard the linked lists stored in the `table` array. More specifically, `table[i]` is guarded by `locks[i % NUM]`, where `NUM` is the length of the `locks` array. 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 is held but can only be modified when *all* locks are held:

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

FastTrack Dynamic Data Race Detector. As an additional case study, we re-implemented the core FastTrack dynamic data race detection algorithm in Anchor. We chose this algorithm because it is a complex, self-contained concurrency component that has been precisely specified and previously verified [Wilcox et al. 2018] in the CIVL verifier [Hawblitzel et al. 2015a]. The algorithm

uses a variety of synchronization idioms in its six main entry points 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. Verifying P/C equivalence required adding about 50 lines of synchronization specifications and other ANCHOR annotations. ANCHOR allowed for the precise and concise specification of all synchronization idioms used in the code. We refer the reader to the Supplementary Appendix for our FASTTRACK implementation and synchronization specification.

Earlier work used CIVL to verify that the FASTTRACK algorithm correctly implements a complete, formal specification of the FASTTRACK analysis rules. However, that correctness guarantee comes at a cost. The core algorithm requires about 250 lines of code in CIVL, but the CIVL specification consists of an additional roughly 530 additional lines [Flanagan et al. 2017]. In other words, the specification was roughly twice the size of the code. Those specifications are quite involved and reflect a significant amount of time and effort to develop.

To capture similar functional correctness guarantees for our ANCHOR version required adding an additional 100 lines of specification (primarily method specifications and loop invariants) to the 50 lines used for verifying P/C equivalence, making the total specification be 150 lines on top of the original 220 lines of code. In this case, the specification was only about 70% the size of the code but expressed the same key correctness requirements as the larger CIVL version.

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 these specifications. More verbose mechanisms are needed to do that in CIVL. One downside of ANCHOR's higher-level approach to synchronization specifications is that it generates more complex verification conditions that lead to longer Boogie verification times (presently about 2.5 minutes vs. 15 seconds for the CIVL version). However, we have yet not conducted performance tuning. We expect optimizing the generated VCs and introducing specialized triggers for quantified formulas to improve performance.

We also 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 code that's executed is exactly the code that is verified.

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. This paper extends movers to play a primary role in synchronization specification, in addition to their traditional role in reduction-based verification.

In ANCHOR, 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, concise, and relatively intuitive. They enable a modular methodology for interleaved development and verification (Section 9), and they have enabled verification of several rather sophisticated concurrent algorithms with moderate effort.

REFERENCES

Martín Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *TOPLAS* 28, 2 (2006), 207–255.

Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference*.

Rahul Agarwal and Scott D. Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *VMCAI*. 149–160.

Roberto M. Amadio and Silvano Dal Zilio. 2004. Resource Control for Synchronous Cooperative Threads. In *CONCUR*.

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.

Gérard Boudol. 2007. Fair Cooperative Multithreading. In *CONCUR*. 272–286.

Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free Java programs. In *OOPSLA*. 56–69.

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

Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2017. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal Methods in System Design* 50, 2-3 (2017), 97–139.

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.

Mark Christiaens and Koenraad De Bosschere. 2001. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*. 761–770.

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/

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. 2019. Reductions for Safety Proofs (Extended Version). arXiv:cs.PL/1910.14619

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

Cormac Flanagan and Stephen N. Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL*. 256–267.

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

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).

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. 2017. VerifiedFT CIVL Implementation. https://github.com/boogie-org/boogie/blob/civl/Test/civl/verified-ft.bpl

Cormac Flanagan and Shaz Qadeer. 2003. Types for atomicity. In *TLDI*. 1–12.

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.

Dan Grossman. 2003. Type-Safe Multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*. 13–25.

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. 2015a. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*. 449–465.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015b. *Automated and Modular Refinement Reasoning for Concurrent Programs*. Technical Report MSR-TR-2015-8. Microsoft Research.

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.

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

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 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, 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.

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. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *PLDI*.

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.

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

Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Virtual Machine Research and Technology Symposium*. 127–138.

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.

Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *TOCS* 15, 4 (1997), 391–411.

Edith Schonberg. 1989. On-The-Fly Detection of Access Anomalies. In *PLDI*. 285–297.

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

Christoph von Praun and Thomas Gross. 2001. Object Race Detection. In *OOPSLA*. 70–82.

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.

Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. 2015. Cooperative types for controlling thread interference in Java. *Sci. Comput. Program.* 112 (2015), 227–260.

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*. 221–234.

A GENERIC REDUCTION THEOREM

We start by presenting a generic reduction theorem that extends Theorem 1 of [Flanagan and Qadeer 2003]. The key extension is that we now consider two operations to commute even if swapping them could introduce new errors (but never hide old errors), as any errors from a preemptive execution will still also be present in the corresponding cooperative execution obtained via repeated swapping.

Given a set of states *State*, with error states $E \subseteq State$, and transition relations $\rightarrow_i, \rightarrow_j \subseteq State \times State$, we say that $\rightarrow_i$ *commutes with* $\rightarrow_j$ *with respect to* E if whenever $s_1 \rightarrow_i s_2 \rightarrow_j s_3$ then either:

(1) $\exists e \in E.\ s_1 \rightarrow_j e$, or
(2) $\exists s_4 \in State, e \in E.\ s_1 \rightarrow_j s_4 \rightarrow_i e$, or
(3) $\exists s_4 \in State.\ s_1 \rightarrow_j s_4 \rightarrow_i s_3$.

Thus, swapping $\rightarrow_i$ and $\rightarrow_j$ transitions either does not affect the final state s_3 or introduces a new error state e.

For $S \subseteq State$, we define

$$\begin{aligned} S/\rightarrow_i &= (\rightarrow_i) \cap (S \times State) \\ \rightarrow_i \backslash S &= (\rightarrow_i) \cap (State \times S) \end{aligned}$$

Theorem 6 (General Reduction)**.** For all i, let R_i, L_i, and E_i be sets of states and $\rightharpoonup_i$ be a transition relation. Suppose, for all i,

(1) R_i, L_i, and E_i are pairwise disjoint.
(2) $(L_i/\rightharpoonup_i \backslash R_i)$ is false.

and for all $j \neq i$,

(3) $\rightharpoonup_i$ and $\rightharpoonup_j$ are disjoint.
(4) $(\rightharpoonup_i \backslash R_i)$ commutes with $\rightharpoonup_j$ with respect to E.
(5) $\rightharpoonup_j$ commutes with $(L_i/\rightharpoonup_i)$ with respect to E.
(6) if $p \rightharpoonup_i q$, then $R_j(p) \Leftrightarrow R_j(q)$, $L_j(p) \Leftrightarrow L_j(q)$, $E_j(p) \Leftrightarrow E_j(q)$.
(7) $\forall q \in L_i.\ q \rightharpoonup_i^* q' \in N_i$, *i.e.*, q is non-blocking.

Let

$$\begin{aligned} N_i &= \neg(R_i \vee L_i) \\ N &= \forall i.\ N_i \\ E &= \exists i.\ E_i \\ \rightharpoonup &= \exists i.\ \rightharpoonup_i \\ \rightharpoonup\!\!\!\!\rightharpoonup &= \exists i.\ (\forall j \neq i.\ N_j)/\rightharpoonup_i \end{aligned}$$

If $p \in N$ with $p \not\rightharpoonup\!\!\!\!\rightharpoonup^* E$, then

(1) $p \not\rightharpoonup^* E$, and
(2) if $p \rightharpoonup^* q \in N$ then $p \rightharpoonup\!\!\!\!\rightharpoonup^* q$.

The proof of this theorem follows the structure of the proof of Theorem 1 in [Flanagan and Qadeer 2003], extended to permit the reduced trace to go wrong more often than the original.

B PROOFS FOR SECTION 6

We now leverage the generic reduction theorem to prove our central reduction argument for ANCHOR.

First, we prove the $\Rightarrow$-steps right-commute (or go wrong) in the PRE phase and left-commute (or go wrong) in the POST phase.

Proving this property requires a notion like strict stability from Definition 5.2. That notion of stability is satisfied by many synchronization specifications. However, strict stability is sometimes overly restrictive. For example, it prohibits interleaved writes from increasing the access permissions of a thread. For maximal expressiveness, we instead derive a more relaxed notion of stability from the conditions necessary to prove the following commuting lemma for all pairs of possible execution steps.

Definition B.1 (Stability). *Specification* R, W is *stable* if for all l, k t, H, v, w, and u where $u \neq t$:

A.1	$W_l(t,H,v) \sqsubseteq \mathsf{R}$	$\wedge$	$W_k(u,H[l:=v],w) \sqsubseteq \mathsf{E}$	$\Rightarrow$	$W_k(u,H,w) \in \{ W_k(u,H[l:=v],w), \mathsf{E} \}$
A.2	$W_l(t,H,v) \neq \mathsf{E}$	$\wedge$	$W_k(u,H[l:=v],w) \not\sqsubseteq \mathsf{L}$	$\Rightarrow$	$W_k(u,H,w) \not\sqsubseteq \mathsf{L}$
A.3	$k \neq l \wedge W_l(t,H,v) \neq \mathsf{E}$	$\wedge$	$W_k(u,H[l:=v],w) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$W_k(u,H,w) \in \{ W_k(u,H[l:=v],w), \mathsf{E} \}$
C	$W_l(t,H,v) \sqsubseteq \mathsf{E}$	$\wedge$	$W_k(u,H,w) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$W_l(t,H[k:=w],v) \in \{ W_l(t,H,v), \mathsf{E} \}$
D	$W_l(t,H,v) \sqsubseteq \mathsf{R}$	$\wedge$	$W_k(u,H[l:=v],w) \sqsubseteq \mathsf{N}$	$\Rightarrow$	$W_l(t,H[k:=w],v) \in \{ W_l(t,H,v), \mathsf{E} \}$
E	$W_l(t,H,v) \sqsubseteq \mathsf{N}$	$\wedge$	$W_k(u,H[l:=v],w) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$W_l(t,H[k:=w],v) \in \{ W_l(t,H,v), \mathsf{E} \}$
F	$R_l(t,H) \sqsubseteq \mathsf{R}$	$\wedge$	$W_k(u,H,w) \sqsubseteq \mathsf{N}$	$\Rightarrow$	$R_l(t,H[k:=w]) \in \{ R_l(t,H), \mathsf{E} \}$
H	$R_l(t,H,v) \sqsubseteq \mathsf{E}$	$\wedge$	$W_k(u,H,w) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$R_l(t,H[k:=w]) \in \{ R_l(t,H), \mathsf{E} \}$
I	$k \neq l \wedge R_l(t,H) \sqsubseteq \mathsf{N}$	$\wedge$	$W_k(u,H,w) \sqsubseteq \mathsf{N}$	$\Rightarrow$	$R_l(t,H[k:=w]) \in \{ R_l(t,H), \mathsf{E} \}$
J	$W_l(t,H,v) \sqsubseteq \mathsf{R}$	$\wedge$	$R_k(u,H) \sqsubseteq \mathsf{E}$	$\Rightarrow$	$R_k(u,H) \in \{ R_k(u,H[l:=v]), \mathsf{E} \}$
K	$W_l(t,H,v) \sqsubseteq \mathsf{N}$	$\wedge$	$R_k(u,H[l:=v]) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$R_k(u,H) \in \{ R_k(u,H[l:=v]), \mathsf{E} \}$
L	$W_l(t,H,v) \sqsubseteq \mathsf{N}$	$\wedge$	$R_k(u,H[l:=v]) \not\sqsubseteq \mathsf{L}$	$\Rightarrow$	$R_k(u,H) \not\sqsubseteq \mathsf{L}$
M	$W_l(t,H,v) \sqsubseteq \mathsf{N}$	$\wedge$	$W_k(u,H,w_1) \sqsubseteq \mathsf{E}$	$\Rightarrow$	$W_k(u,H[k:=w_1],w_2) \in \{ W_k(u,H[l:=v][k:=w_1],w_2),$
N	$W_l(t,H,v_1) \sqsubseteq \mathsf{E}$	$\wedge$	$W_k(u,H,w) \sqsubseteq \mathsf{L}$	$\Rightarrow$	$W_l(t,H[k:=w][l:=v_1],v_2) \in \{ W_l(t,H[l:=v_1],v_2), \mathsf{E} \}$

(Shaded terms are trivially true but included for symmetry among the rules.)

Lemma 1 (Commuting)**.** Suppose R, W is a valid and stable synchronization discipline, $u \neq t$, and

$$\Pi_1 \Rightarrow_t \Pi_2 \Rightarrow_u \Pi_3$$

where $\Pi_2 = T_2 \cdot H_2 \cdot C_2 \cdot P_2$, and $P_2(t) = \text{PRE}$ or $P_2(u) = \text{POST}$. Then:

(1) $\exists \Pi_4.\ \Pi_1 \Rightarrow_u \Pi_4 \Rightarrow_t \Pi_3$, or
(2) $\exists \Pi_4, \Pi_5.\ \Pi_1 \Rightarrow_u \Pi_4 \Rightarrow_t \Pi_5$ and t is wrong in Π_5, or
(3) $\exists \Pi_4.\ \Pi_1 \Rightarrow_u \Pi_4$ and u is wrong in Π_4.

Next, we define the predicates $\mathbb{R}_i, \mathbb{L}_i, \mathbb{E}_i, \mathbb{N}_i, \mathbb{N} \subseteq \Pi$ over an instrumented state $\Pi = T \cdot P \cdot H$:

$$\begin{aligned}
\mathbb{R}_i(\Pi) &= P[i] = \text{PRE} \wedge \neg \mathbb{N}_i(\Pi) \\
\mathbb{L}_i(\Pi) &= P[i] = \text{POST} \wedge \neg \mathbb{N}_i(\Pi) \\
\mathbb{E}_i(\Pi) &= \text{thread } i \text{ is wrong in } \Pi \\
\mathbb{N}_i(\Pi) &= \text{thread } i \text{ is yielding in } \Pi \\
\mathbb{N}(\Pi) &= \forall i.\ \mathbb{N}_i(\Pi)
\end{aligned}$$

These predicates characterize whether thread i is in the pre-commit, or right-mover, part of a reducible code sequence ($\mathbb{R}_i$); in the post-commit, or left-mover, part ($\mathbb{L}_i$); gone wrong ($\mathbb{E}_i$); or is yielding ($\mathbb{N}_i$).

We can now formalize the *nonblocking* requirement that any transaction that has passed its commit point must be able to terminate, i.e. if $\Pi_0 \Rightarrow^* \Pi'$ and $\mathbb{L}_i(\Pi')$ then $\Pi' \Rightarrow_i^* \Pi''$ such that $\mathbb{N}_i(\Pi'')$.

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

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

Proof. By Theorem 6, where the preconditions of that theorem are satisfied as follows. For all $i, j \in \mathit{Tid}$ where $i \neq j$:

(1) $\mathbb{R}_i$, $\mathbb{L}_i$, and $\mathbb{E}_i$ are pairwise disjoint by construction.
(2) $\mathbb{L}_i \,/\, \Rightarrow_i \backslash \mathbb{R}_i$ is false. $\mathbb{L}_i$ says that $P(i) = \textsc{Post}$ and $T(i)$ is not yielding. $\mathbb{R}_i$ says that $P(i) = \textsc{Pre}$. However, the only transition of $P(i)$ from Post to Pre is for `yield`.
(3) $\Rightarrow_i$ and $\Rightarrow_j$ are disjoint.
(4) $\Rightarrow_i \,/\, \mathbb{R}_i$ commutes with $\Rightarrow_j$ with respect to E_i. By Lemma 1.
(5) $\Rightarrow_j$ commutes with $\mathbb{L}_i \setminus \Rightarrow_i$ with respect to E_i. By Lemma 1.
(6) $\Rightarrow_i$ does not change the phase or statements of other threads.
(7) By the non-blocking assumption.

□

C VERIFIEDFT IMPLEMENTATION IN ANCHOR

The following is the ANCHOR implementation of the core FASTTRACK algorithm, as outlined in [Wilcox et al. 2018]. It includes the specification of the synchronization discipline, and ANCHOR verifies that this code is P/C equivalent.

```
/*

 An Anchor implementation of the VerifiedFT-v2 algorithm from

  VerifiedFT: A Verified, High-Performance Precise Dynamic Race Detector
  James R. Wilcox, Cormac Flanagan, Stephen N. Freund
  PPOPP, 2018

 We refer the reader to that paper for a full discussion of the
 synchronization and algorithm.

 Notes:

  * In the original, inheritance was used to create three specialized
    versions of VectorClocks for ThreadState, VarState, LockState.  In
    Anchor, we needed to duplicate the VectorClock code in each of the
    three classes since there is no inheritance.

  * Unlike the original, we include initializers for each of the
    three main classes.

  * We use array abbreviations in Anchor, which were not included in
    the paper, eg:

    array T = int[moves_as ...]

    This introduces an array type abbreviation that can be used
    later, as in

    [T] v = new [T](10);

  * In array specs, the array object is referred to as "athis", and the
    enclosing object is "this".

  * We encode an ecoch tid@c as (tid + c * 256).  That is, the lower 8
    bits of an int is the tid and the upper 24 is the clock.  We
    assume no more than 255 threads will be used and that overflow
    does not occur.  We use the constant -1 to represent the SHARED
    special value.

*/

// Some macros to avoid repeated low-level expressions:
#define EPOCH_TID(x) (x % 256)
#define EPOCH_CLOCK(x) (x / 256)
#define EPOCH_LEQ(x,y) x <= y
#define INC(x) (x + 256)
```

```
#define VC_GET(vc, i) ((i < vc.length) ? vc[i] : (i))
#define MAX(x,y) ((x) > (y) ? (x) : (y))


class Object { }

//////

class VarState {

  // NOTES:
  // * SHARED is -1.
  // * We omitted thread-local accesses from the specs in the body of the paper
  //   for simplicity, but include them here in order to verify
  //   the VarState constructor.

  array VC_VarState =
    int[moves_as isLocal(athis) ? B
                                : (this.read != -1) ? guarded_by this
                                                    : isRead() ? (holds(this) || tid == index ? B:E)
                                                               : (holds(this) && tid == index ? B:E)]

  volatile int read  moves_as isLocal(this) ? B
                                            : (this.read != -1) ? write_guarded_by this
                                                                : (isRead() ? R : E);

  volatile int write  moves_as isLocal(this) ? B : write_guarded_by this;

  volatile [VC_VarState] vc  moves_as isLocal(this) ? B
                                                    : (this.read != -1) ? guarded_by this
                                                                        : write_guarded_by this;

  invariant isShared(this) ==> this.vc != null;

  public VarState() {
    this.read = 0;
    this.write = 0;
    this.vc = new [VC_VarState](0);
  }

  int get(int i) {
    [VC_VarState] vc = this.vc;
    return (i < vc.length) ? vc[i] : i;
  }

  void ensureCapacity(int n) {
    if (n > this.vc.length) {
      [VC_VarState] newVC = new [VC_VarState](n);

      for (int i = 0; i < n; i = i + 1) {
        newVC[i] = this.get(i);
      }
```

```
      this.vc = newVC;
    }
  }

  void set(int index, int value) {
    this.ensureCapacity(index + 1);
    this.vc[index] = value;
  }

  int size() {
    return this.vc.length;
  }

  bool leq(ThreadState st) {
    int stSize = st.size();
    int thisSize = this.size();
    int n = MAX(thisSize, stSize);
    int i = 0;
    while (i < n) {
      if (this.get(i) > st.get(i)) {
        return false;
      }
      i = i + 1;
    }
    return true;
  }
}

//////

class LockState {
  array VC_LockState = int[moves_as  guarded_by this.target]

  Object target moves_as isLocal(this) ? B : readonly;

  [VC_LockState] vc moves_as isLocal(this) ? B : guarded_by this.target;

  invariant isShared(this.target);
  invariant isShared(this) ==> this.vc != null;

  requires target != null;
  public LockState(Object target) {
    this.target = target;
    this.vc = new [VC_LockState](0);
  }

  int get(int i) {
    [VC_LockState] vc = this.vc;
    return (i < vc.length) ? vc[i] : i;
  }

  void ensureCapacity(int n) {
```

```
    [VC_LockState] vc = this.vc;

    if (n > vc.length) {
      [VC_LockState] newVC = new [VC_LockState](n);
      for (int i = 0; i < n; i = i + 1) {
        newVC[i] = this.get(i);
      }
      this.vc = newVC;
    }
  }

  void set(int index, int value) {
    this.ensureCapacity(index + 1);
    this.vc[index] = value;
  }

  int size() {
    return this.vc.length;
  }

  void copy(ThreadState st) {
    [VC_LockState] oldVCPtr = this.vc;

    this.ensureCapacity(st.size());
    for (int j = 0; j < this.size(); j = j + 1) {
      int x = st.get(j);
      this.vc[j] = x;
    }
  }
}

/////////

class ThreadState {

  array VC_ThreadState = int[moves_as isLocal(this) ? B
                                                    : this.stopped ? (isRead() ? B : E)
                                                                   : (tid == this.t ? B:E)]

  volatile bool stopped
    moves_as isLocal(this) ? B
                           : isRead() ? (tid == this.t ? B
                                                       : (this.stopped ? R : N))
                                      : (tid == this.t && !this.stopped && newValue ? N : E);

  Tid t  moves_as  isLocal(this) ? B : readonly;

  invariant 0 <= this.t && this.t < 256;

  volatile [VC_ThreadState] vc
      moves_as isLocal(this) ? B
                             : this.stopped ? isRead() ? B : E
```

```
                                          : (tid == this.t ? B : E);

invariant isShared(this.vc) && this.vc != null;


int get(int i) {
  [VC_ThreadState{this}] vc = this.vc;
  return (i < vc.length) ? vc[i] : i;
}

void set(int index, int value) {
  this.ensureCapacity(index + 1);
  this.vc[index] = value;
}

void ensureCapacity(int n) {
      [VC_ThreadState] vc = this.vc;

  if (n > vc.length) {
    [VC_ThreadState] newVC = new [VC_ThreadState](n);

    for (int i = 0; i < n; i = i + 1) {
      newVC[i] = this.get(i);
    }
    this.vc = newVC;
  }
}

int size() {
  return this.vc.length;
}

void joinThread(ThreadState st) {
  for (int i = 0; i < st.size(); i = i + 1) {
    int thisI = this.get(i);
    int stI = st.get(i);
    this.set(i, MAX(thisI, stI));
  }
}

void inc(int t) {
  this.set(t, INC(this.get(t)));
}


requires tid == this.t;
requires !this.stopped;
public bool write(VarState sx) {
  int e = this.get(this.t);
  int w = sx.write;
  if (w == e) { return true; }
  yield;
```

```
    synchronized(sx) {
      w = sx.write;
      int ww = this.get(EPOCH_TID(w));
      if (w > ww) { return false; }
      int r = sx.read;
      if (r != -1) {
        int rr = this.get(EPOCH_TID(r));
        if (r > rr) { return false; }
        sx.write = e;
      } else {
        if (!sx.leq(this)) {
          return false;
        }
        sx.write = e;
      }
    }
    return true;
  }

  requires tid == this.t;
  requires !this. stopped;
  public bool read(VarState sx) {
    int t = this.t;
    int e = this.get(t);

    int r;
    r = sx.read;
    if (r == e || r == -1 && sx.get(t) == e) { return true; }
    yield;
    synchronized(sx) {
      int w = sx.write;
      int ww = this.get(EPOCH_TID(w));
      if (w > ww) { return false; }
      r = sx.read;
      if (r != -1) {
        int rr = this.get(EPOCH_TID(r));
        if (r <= rr) {
          sx.read = e;
        } else {
          sx.set(EPOCH_TID(r), r);
          sx.set(t, e);
          sx.read = -1;
          return true;
        }
      } else {
        sx.set(t, e);
      }
    }
    return true;
  }
```

```
  requires tid == this.t;
  requires !this.stopped;
  requires holds(sm.target);
  public void acq(LockState sm) {
    [VC_ThreadState] oldThisVc = this.vc;
    for (int j = 0; j < sm.size(); j = j + 1)
    {
      int thisJ = this.get(j);
      int smJ = sm.get(j);
      this.set(j, MAX(thisJ, smJ));
    }
  }


  requires tid ==this.t;
  requires !this.stopped;
  requires holds(sm.target);
  modifies sm, sm.vc;
  public void rel(LockState sm) {
    sm.copy(this);
  }



  requires 0 <= t && t < 256;
  requires t != 0 ==> forker != null && tid == forker.t;
  public ThreadState(int t, ThreadState forker) {
    this.t = t;
    this.stopped = false;

    if (t != 0) {
      int fSize = forker.size();
      int n = MAX(t + 1, fSize);
      [VC_ThreadState] vc = new[VC_ThreadState] (n);
      for (int i = 0; i < n; i = i + 1) {
        vc[i] = i == t ? INC(forker.get(i)) : forker.get(i);
      }
      this.vc = vc;
    } else {
      [VC_ThreadState] vc = new[VC_ThreadState](1);
      vc[0] = 256;
      this.vc = vc;
    }
  }


  requires tid == this.t;
  requires !this.stopped;
  requires this.t != uTid;
  requires uTid > 0 && uTid < 256;
  public ThreadState fork(int uTid) {
    // We create the new ThreadState object here so that we
```

```
    // have threadlocal access to it.
    ThreadState u = new ThreadState(uTid, this);
    this.inc(this.t);
    return u;
  }



  requires tid == this.t;
  requires !this.stopped;
  requires u.stopped;
  public void join(ThreadState u) {
    [VC_ThreadState] oldVCPtr = this.vc;
    int n = MAX(u.size(), this.size());

    for (int i = 0; i < n; i = i + 1) {
      int uI = u.get(i);
      int thisI = this.get(i);
      int x = MAX(uI, thisI);
      this.set(i, x);
    }
    this.inc(this.t);
  }


}
```