Scenario-Based Proofs for Concurrent Objects

Eric Koskinen • FRIDA 2024 • July 23, 2024

Joint work with Constantin Enea (É.P.)

Concurrent Objects

Module Saturn

Domain-safe data structures for Multicore OCaml

Data structures

```
module Queue = Lockfree.Queue
```

```
module Stack = Lockfree.Stack
```

```
module Work_stealing_deque = Lockfree.Work_stealing_deque
```

```
module Single_prod_single_cons_queue =
Lockfree.Single_prod_single_cons_queue
```

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See: Description

Interface Summary

Interface	Description
BlockingDeque<E>	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
CompletableFuture.AsynchronousCompletionTask	A marker interface identifying asynchronous tasks produced by async methods.
CompletionService<V>	A service that decouples the production of new asynchronous tasks from the consumption of the results

intel. PRODUCTS SUPPORT SOLUTIONS MORE + ENGLISH Search Intel.com

Developers / Tools / oneAPI / Components / Intel® oneAPI Threading Building Blocks / concurrent_hash_map

Intel® oneAPI Threading Building Blocks Developer Guide and API Reference

Download PDF

View More

Search this document

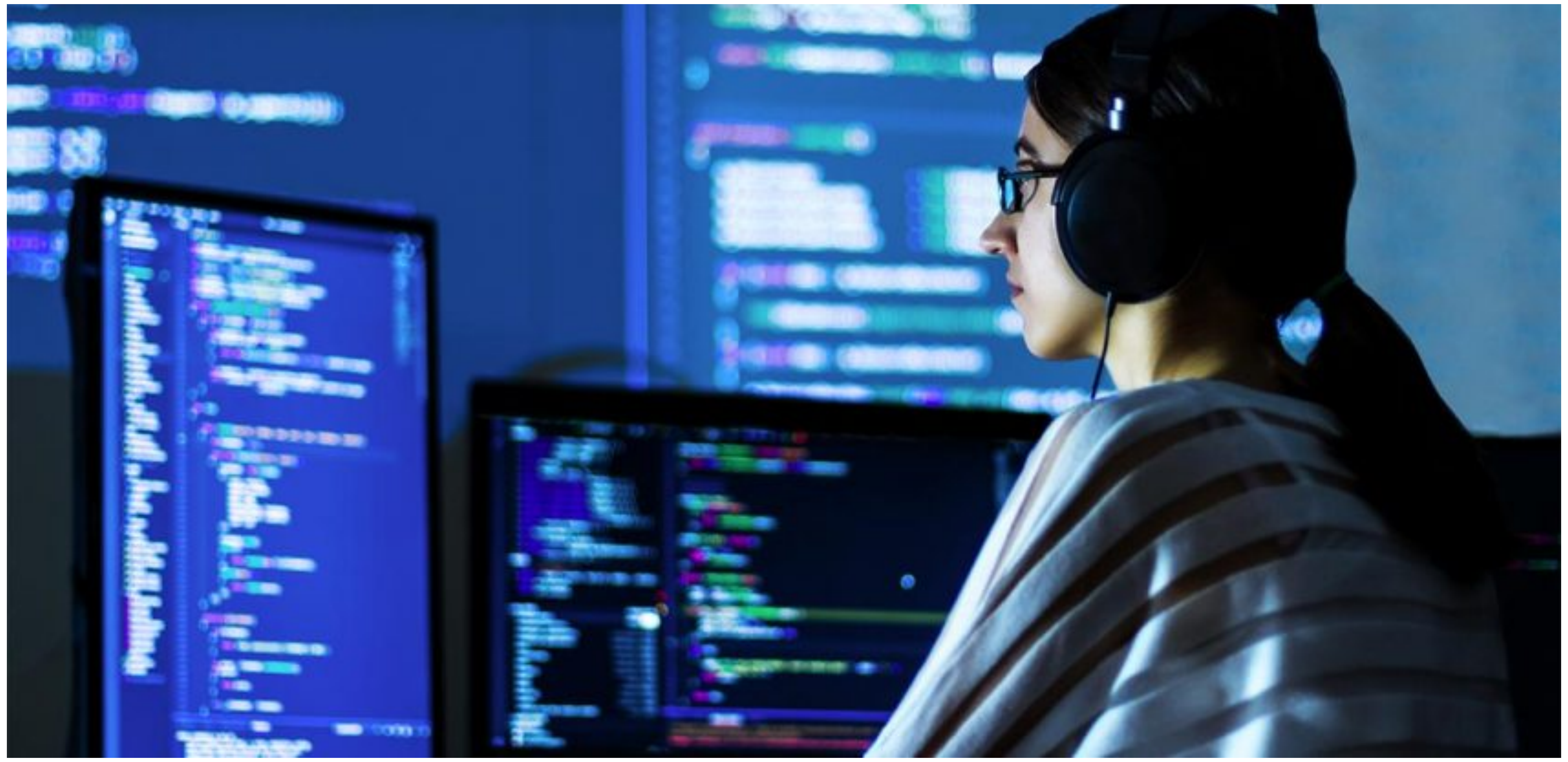
Document Table of Contents →

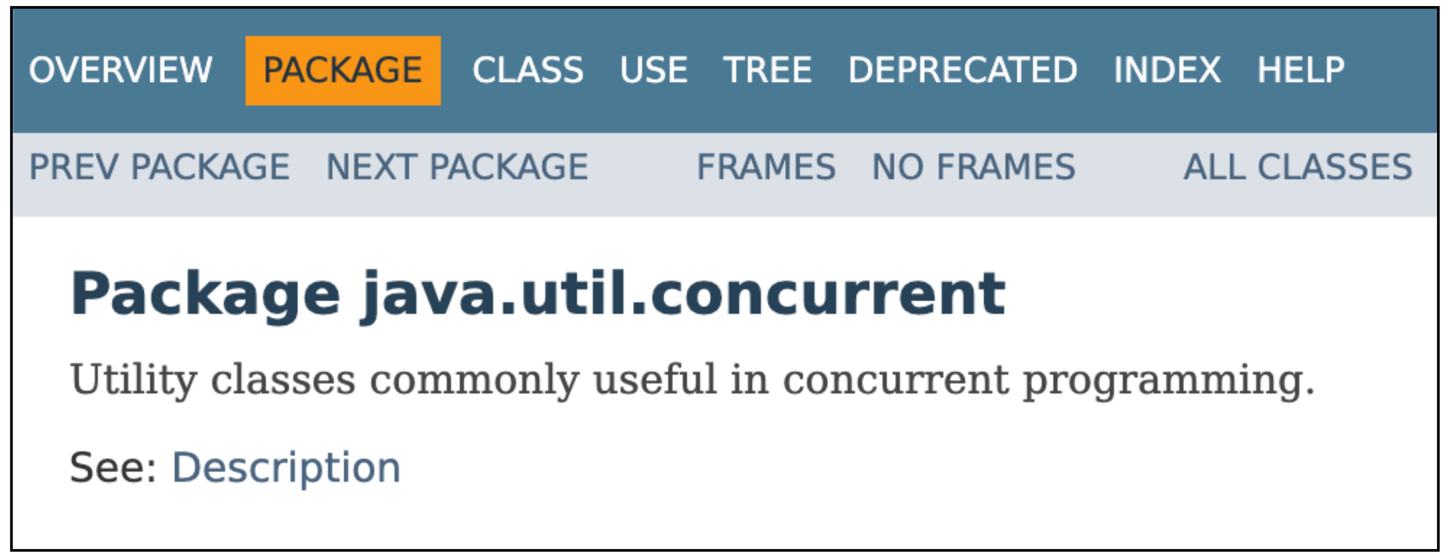
concurrent_hash_map

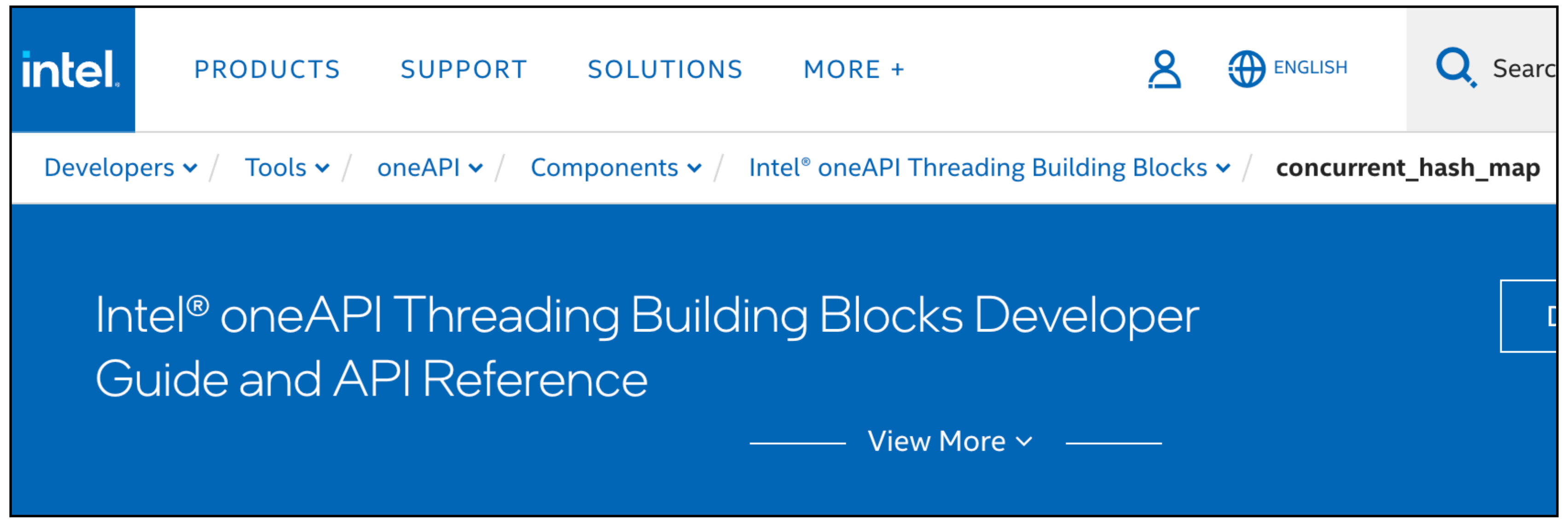
concurrent_hash_map

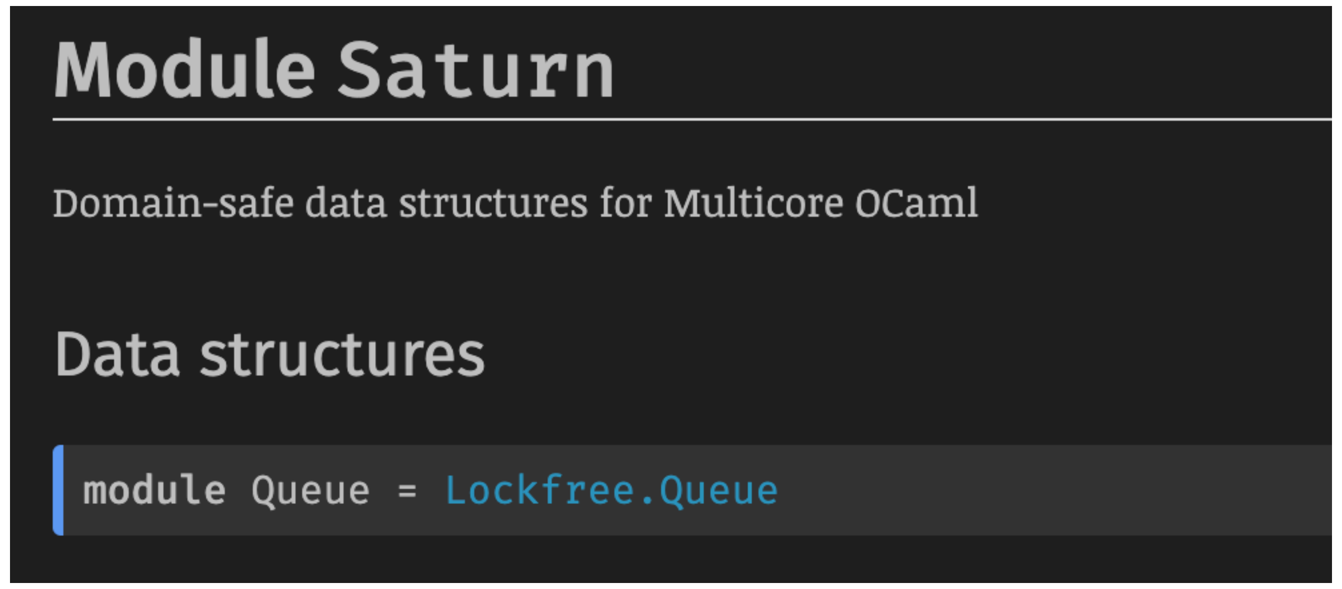
A `concurrent_hash_map<Key, T, HashCompare >` is a hash table that permits concurrent accesses. The table is a map from a key to a type T. The traits type HashCompare defines how to hash a key and how to compare two keys.

Concurrent Objects









Concurrent Objects

Even Better DCAS-Based Concurrent Deques

David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite,
Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr.

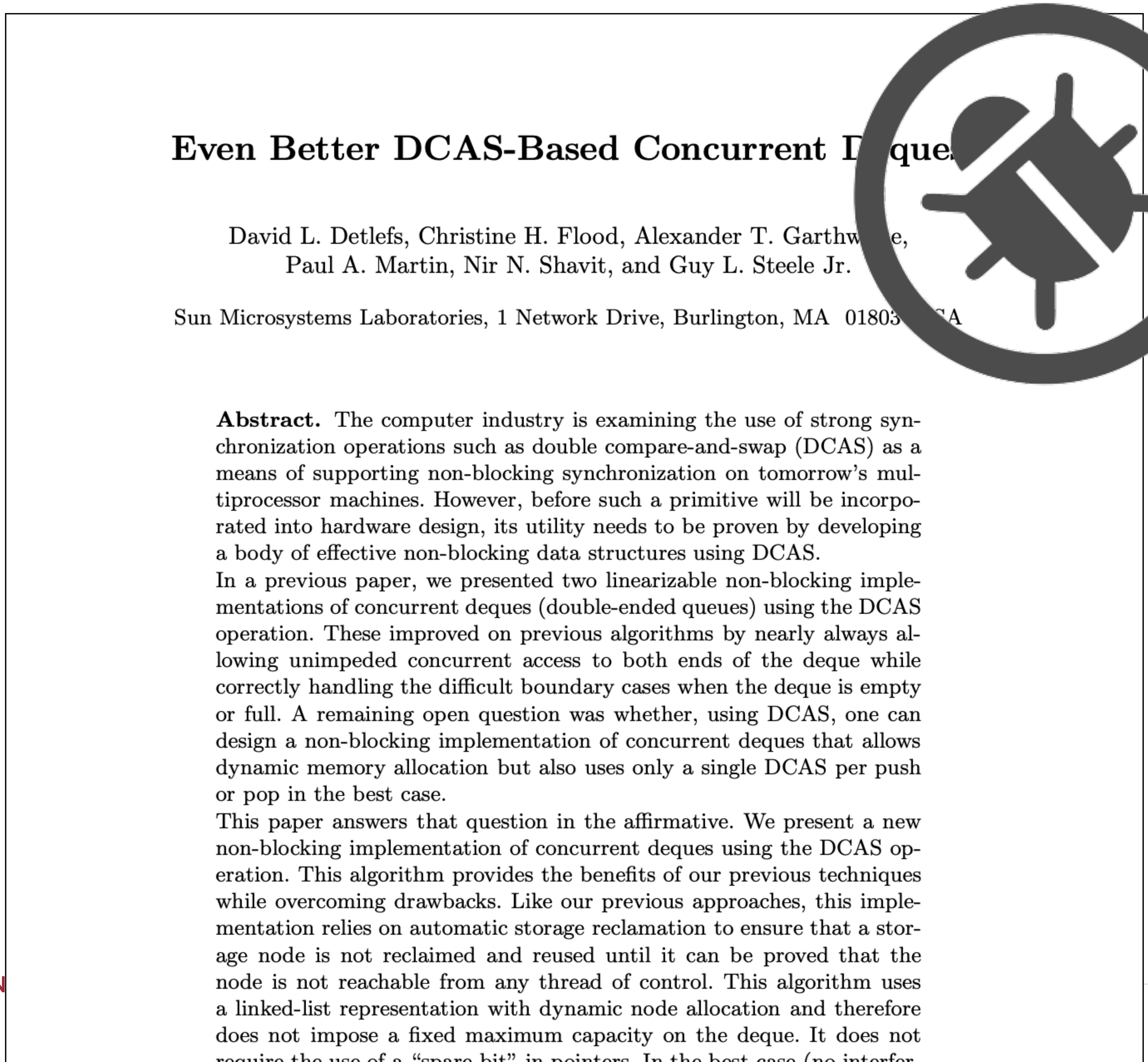
Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803 USA

Abstract. The computer industry is examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on tomorrow's multiprocessor machines. However, before such a primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.

In a previous paper, we presented two linearizable non-blocking implementations of concurrent deques (double-ended queues) using the DCAS operation. These improved on previous algorithms by nearly always allowing unimpeded concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation but also uses only a single DCAS per push or pop in the best case.

This paper answers that question in the affirmative. We present a new non-blocking implementation of concurrent deques using the DCAS operation. This algorithm provides the benefits of our previous techniques while overcoming drawbacks. Like our previous approaches, this implementation relies on automatic storage reclamation to ensure that a storage node is not reclaimed and reused until it can be proved that the node is not reachable from any thread of control. This algorithm uses a linked-list representation with dynamic node allocation and therefore does not impose a fixed maximum capacity on the deque. It does not require the use of a “spare bit” in pointers. In the best case (no interfer-

Concurrent Objects



Even Better DCAS-Based Concurrent D que

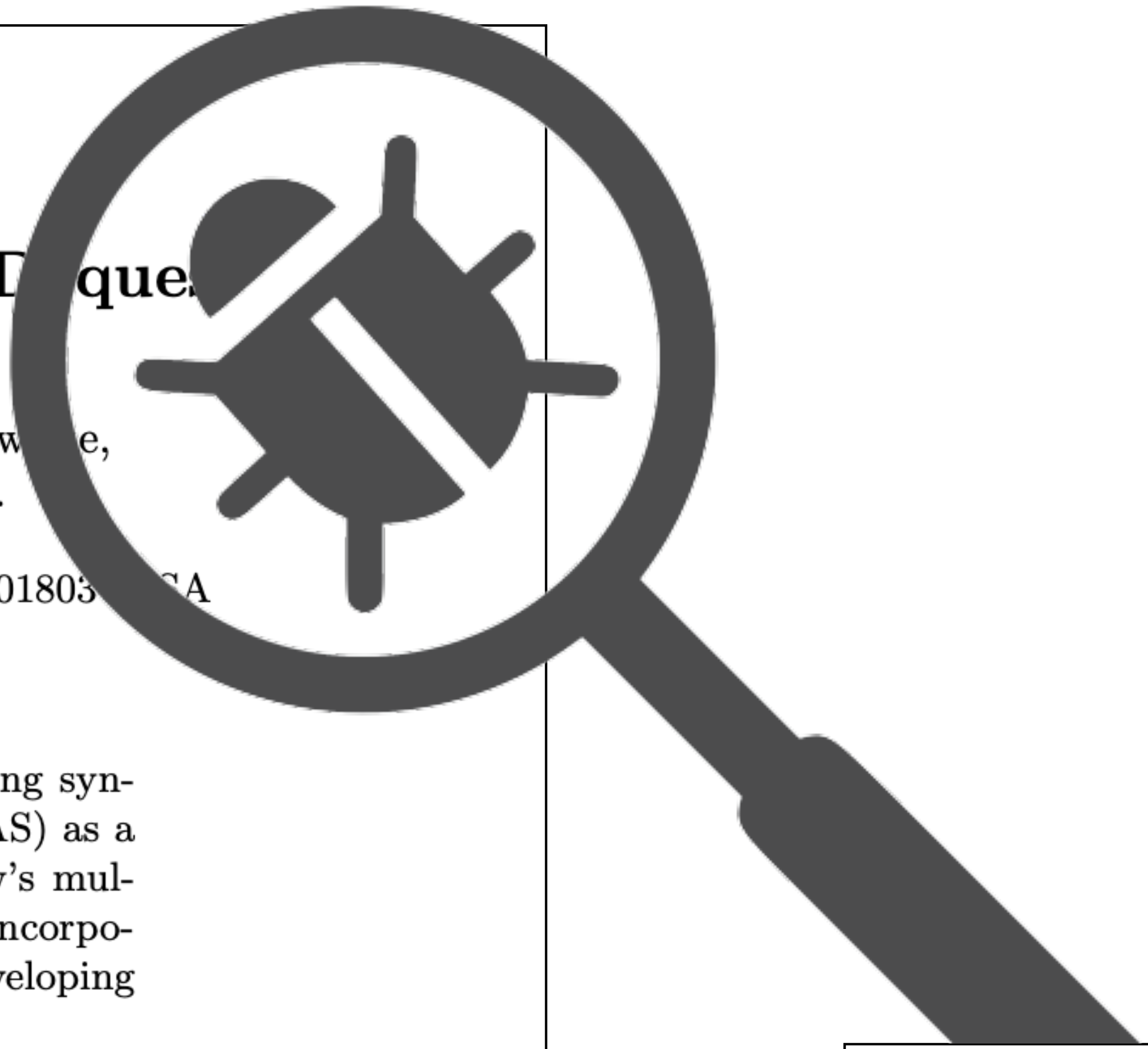
David L. Detlefs, Christine H. Flood, Alexander T. Garthw e,
Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr.

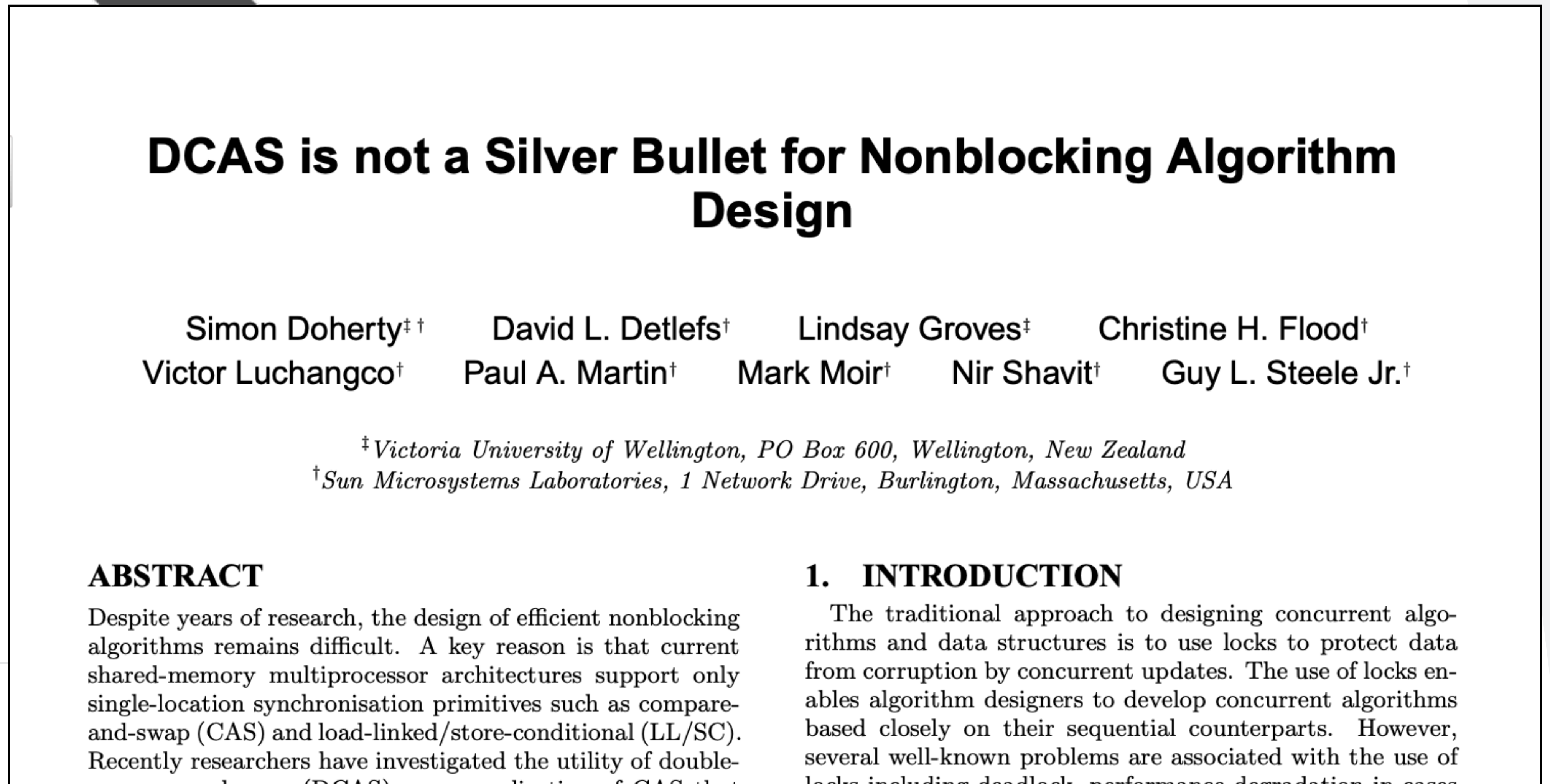
Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803 A

Abstract. The computer industry is examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on tomorrow's multiprocessor machines. However, before such a primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.

In a previous paper, we presented two linearizable non-blocking implementations of concurrent deques (double-ended queues) using the DCAS operation. These improved on previous algorithms by nearly always allowing unimpeded concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation but also uses only a single DCAS per push or pop in the best case.

This paper answers that question in the affirmative. We present a new non-blocking implementation of concurrent deques using the DCAS operation. This algorithm provides the benefits of our previous techniques while overcoming drawbacks. Like our previous approaches, this implementation relies on automatic storage reclamation to ensure that a storage node is not reclaimed and reused until it can be proved that the node is not reachable from any thread of control. This algorithm uses a linked-list representation with dynamic node allocation and therefore does not impose a fixed maximum capacity on the deque. It does not





DCAS is not a Silver Bullet for Nonblocking Algorithm Design

Simon Doherty[‡ †] David L. Detlefs[†] Lindsay Groves[‡] Christine H. Flood[†]
Victor Luchangco[†] Paul A. Martin[†] Mark Moir[†] Nir Shavit[†] Guy L. Steele Jr.[†]

[‡]*Victoria University of Wellington, PO Box 600, Wellington, New Zealand*
[†]*Sun Microsystems Laboratories, 1 Network Drive, Burlington, Massachusetts, USA*

ABSTRACT

Despite years of research, the design of efficient nonblocking algorithms remains difficult. A key reason is that current shared-memory multiprocessor architectures support only single-location synchronisation primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). Recently researchers have investigated the utility of double-

1. INTRODUCTION

The traditional approach to designing concurrent algorithms and data structures is to use locks to protect data from corruption by concurrent updates. The use of locks enables algorithm designers to develop concurrent algorithms based closely on their sequential counterparts. However, several well-known problems are associated with the use of

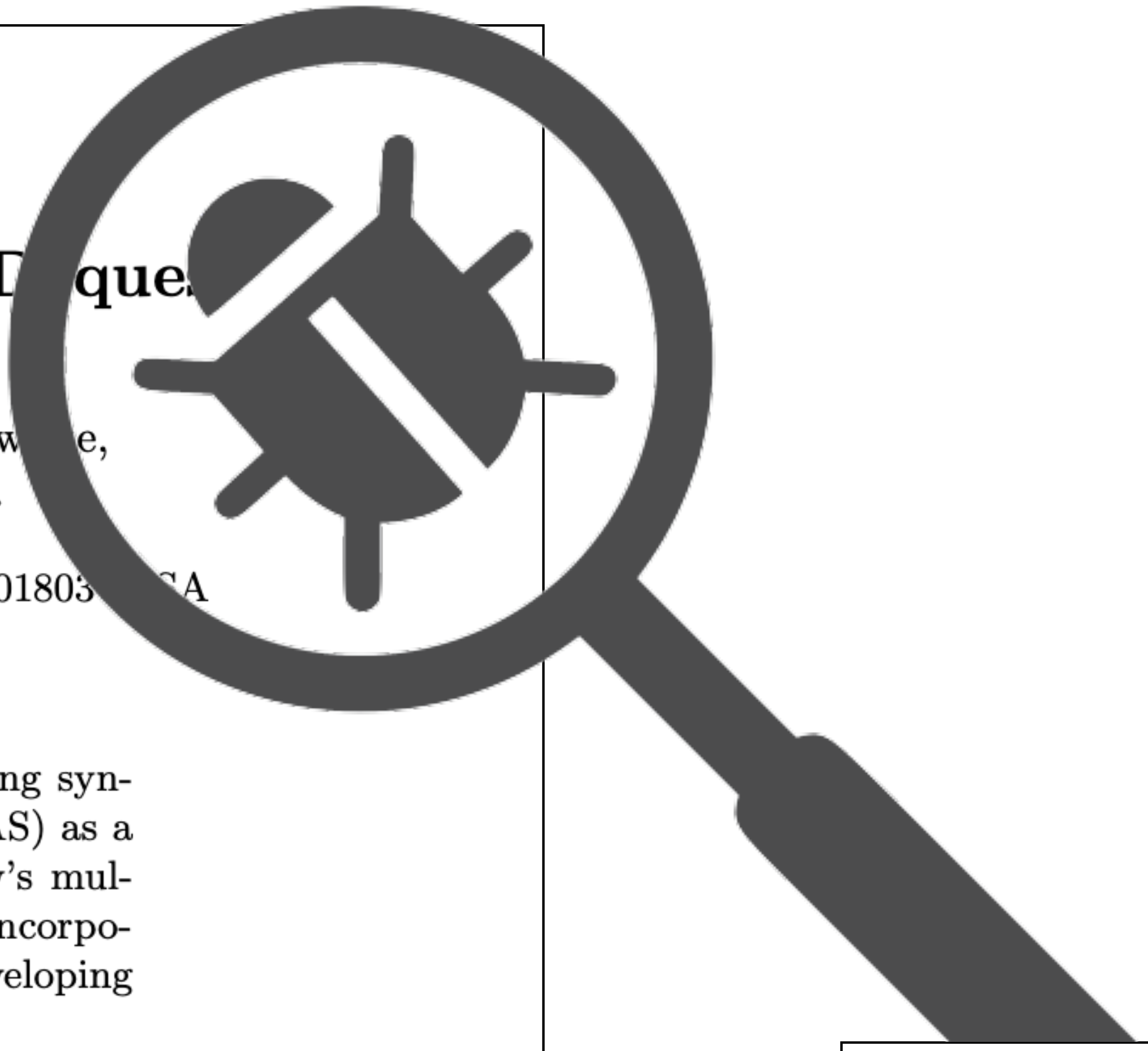
Concurrent Objects

Even Better DCAS-Based Concurrent D que

David L. Detlefs, Christine H. Flood, Alexander T. Garthw e,
Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr.

Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803 A

Abstract. The computer industry is examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on tomorrow's multiprocessor machines. However, before such a primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.
In a previous paper, we presented two linearizable non-blocking implementations of concurrent deques (double-ended queues) using the DCAS operation. These improved on previous algorithms by nearly always allowing unimpeded concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation but also uses only a single DCAS per push or pop in the best case.
This paper answers that question in the affirmative. We present a new non-blocking implementation of concurrent deques using the DCAS operation. This algorithm provides the benefits of our previous techniques while overcoming drawbacks. Like our previous approaches, this implementation relies on automatic storage reclamation to ensure that a storage node is not reclaimed and reused until it can be proved that the node is not reachable from any thread of control. This algorithm uses a linked-list representation with dynamic node allocation and therefore does not impose a fixed maximum capacity on the deque. It does not require the use of a "spare bit" in pointers. In the best case (no interfer-



Checking a Multithreaded Algorithm with $^{+}$CAL

Leslie Lamport
Microsoft Research

11 Jul 2006

To appear in DISC 2006

Abstract

A colleague told me about a multithreaded algorithm that was later reported to have a bug. I rewrote the algorithm in the $^{+}$CAL algorithm language, ran the TLC model checker on it, and found the error. Programs are not released without being tested; why should algorithms be published without being model checked?

DCAS i

Simon D
Victor Luch

[‡]*Victoria University of Wellington, PO Box 600, Wellington, New Zealand*
[†]*Sun Microsystems Laboratories, 1 Network Drive, Burlington, Massachusetts, USA*

ABSTRACT

Despite years of research, the design of efficient nonblocking algorithms remains difficult. A key reason is that current shared-memory multiprocessor architectures support only single-location synchronisation primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). Recently researchers have investigated the utility of double-

1. INTRODUCTION

The traditional approach to designing concurrent algorithms and data structures is to use locks to protect data from corruption by concurrent updates. The use of locks enables algorithm designers to develop concurrent algorithms based closely on their sequential counterparts. However, several well-known problems are associated with the use of

So we need more rigorous guarantees.

Michael-Scott Queue

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

Maged M. Michael   Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{michael,scott}@cs.rochester.edu

Abstract

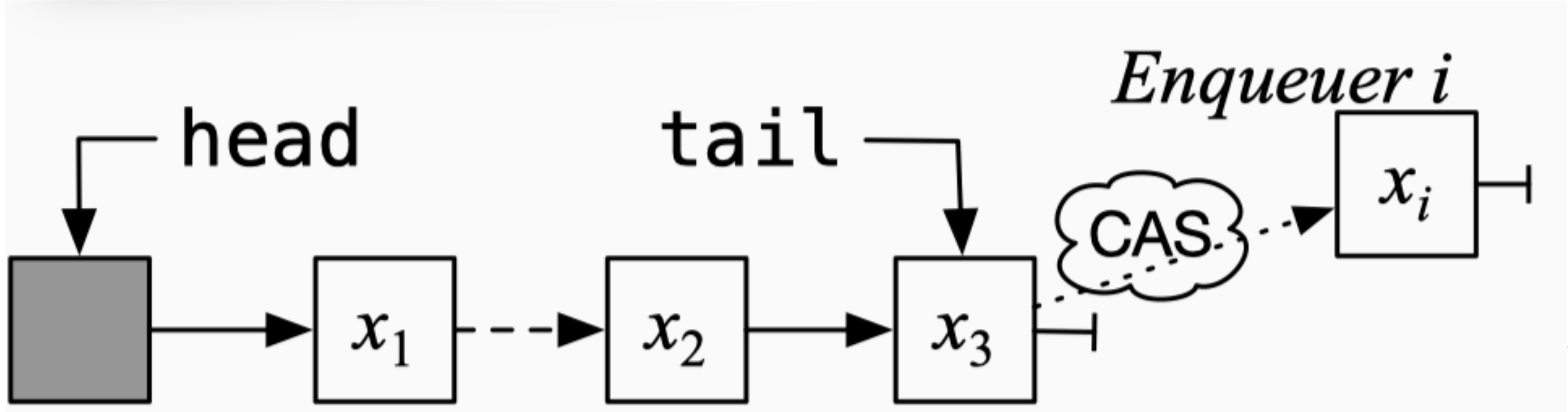
Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. Both algorithms are simple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. `compare_and_swap` or `load_linked/store_conditional`). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives (e.g. `test_and_set`). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpre-

1 Introduction

Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking algorithms are more robust in the face of these events

Michael-Scott Queue

```
int enq(int v){ loop {
 node_t *node=...;
 node->val=v;
 tail=Q.tail;
 next=tail->next;
 if (Q.tail==tail) {
  if (next==null) {
   if (CAS(&tail->next,
          next,node))
      ret 1;
} } } }
```

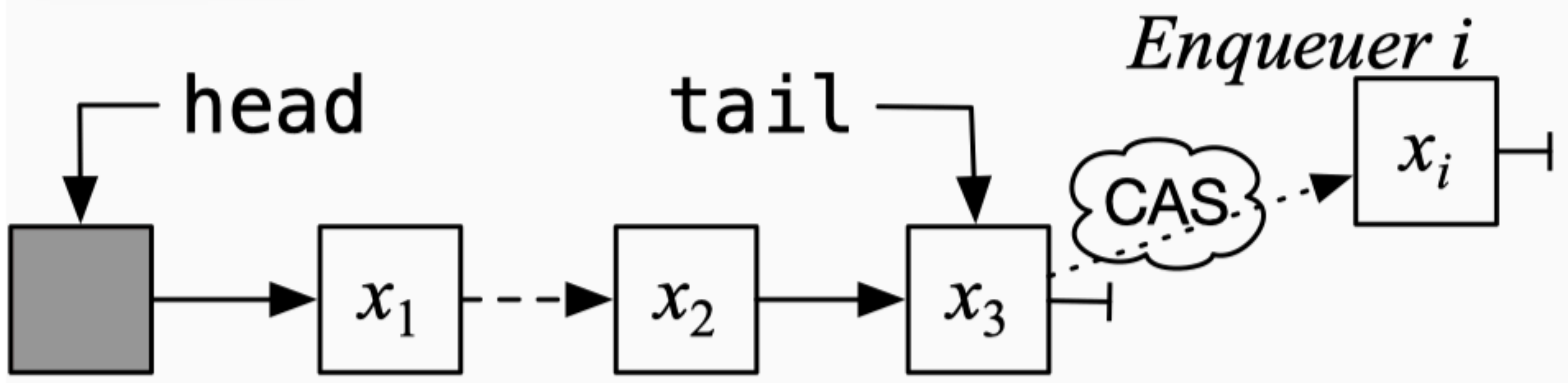


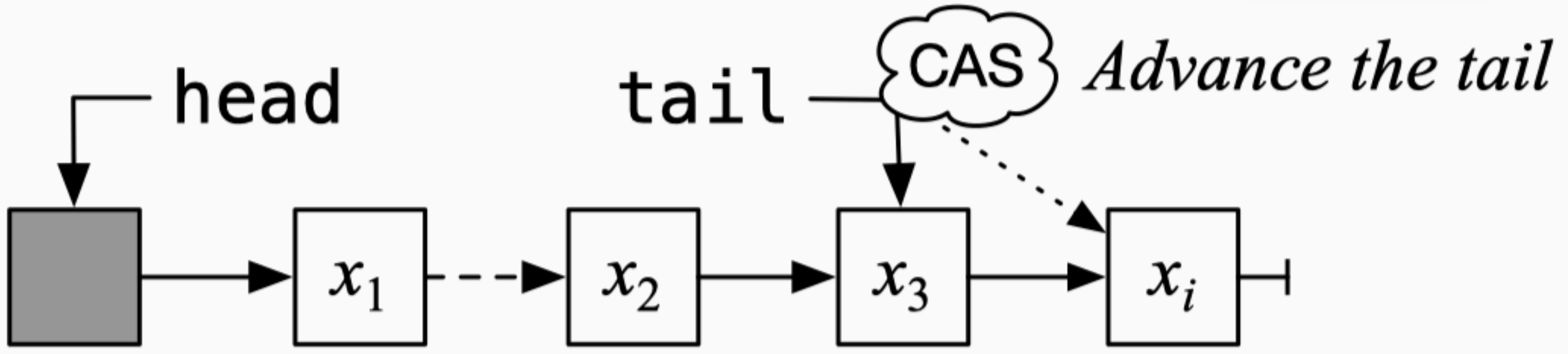
Michael-Scott Queue

```
int enq(int v){ loop {
 node_t *node=...;
 node->val=v;
 tail=Q.tail;
 next=tail->next;
 if (Q.tail==tail) {
  if (next==null) {
   if (CAS(&tail->next,
          next,node))
      ret 1;
} } } }
```

Factored out tail advancement:

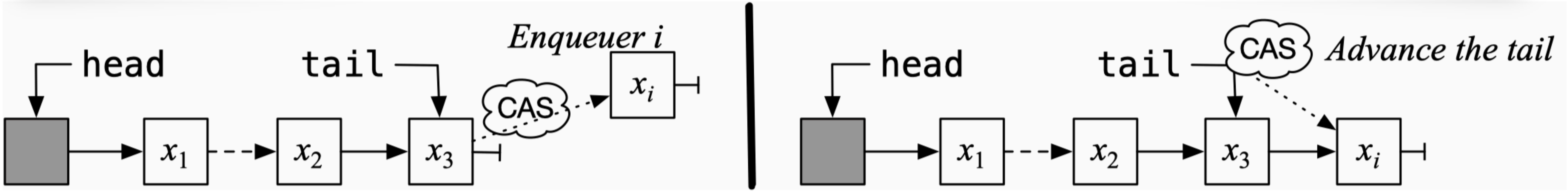
```
adv(){ loop {
 tail=Q.tail;
 next=tail->next;
 if (next!=null){
  if (CAS(&Q->tail,
     tail,next))
   ret 0;
 }
} }
```





Michael-Scott Queue

```
int deq(){ loop {
 int pval;
 head=Q.head;tail=Q.tail;
 next=head->next;
 if (Q.head==head) {
  if (head==tail) {
   if (next==null) ret 0;
  } else {
    pval=next->val;
    if (CAS(&Q->head,
            head,next))
      ret pval;
  } } } } }
```



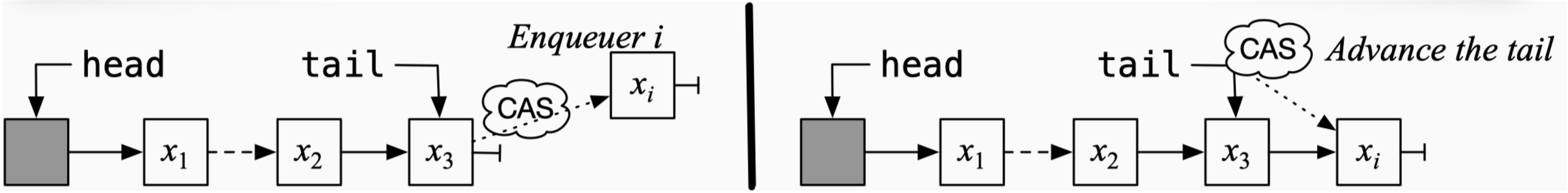
Michael-Scott Queue

```
int enq(int v){ loop {
 node_t *node=...;
 node->val=v;
 tail=Q.tail;
 next=tail->next;
 if (Q.tail==tail) {
  if (next==null) {
   if (CAS(&tail->next,
          next,node))
      ret 1;
} } } }
```

```
int deq(){ loop {
 int pval;
 head=Q.head;tail=Q.tail;
 next=head->next;
 if (Q.head==head) {
  if (head==tail) {
   if (next==null) ret 0;
  } else {
    pval=next->val;
    if (CAS(&Q->head,
             head,next))
      ret pval;
 } } } } }
```

Factored out tail advancement:

```
adv(){ loop {
 tail=Q.tail;
 next=tail->next;
 if (next!=null){
  if (CAS(&Q->tail,
     tail,next))
   ret 0;
 }
} }
```



Proving Linearizability

- Owicki/Gries
- Rely/Gaurantee
- Concurrent Separation Logic
- RGSep
- Deny-Guarantee
- Views
- Iris
- Many others …

```
Definition queueΣ := #[ GFunctor setUR ].
Instance subG_lockPoolΣ {Σ} : subG queueΣ Σ → queueG Σ.
Proof. solve_inG. Qed.

Section queue_refinement.
  Context `{relocG Σ, queueG Σ}.

  Lemma refines_load_alt K E l t A :
    (|={⊤,E}=> ∃ v' q,
      ▷(l ↦{q} v') ∗
      ▷(l ↦{q} v' -∗ (REL fill K (of_val v') << t @
    -∗ REL fill K (! #l) << t : A.
  Proof.
    iIntros "Hlog".
    iApply refines_atomic_l; auto.
    iMod "Hlog" as (v' q) "[Hl Hlog]". iModIntro.
    iApply (wp_load with "Hl"); auto.
  Qed.

  Tactic Notation "rel_load_l_atomic" := rel_apply_l refines_load_alt.

  Definition isNode ℓn x (ℓnOut : loc) : iProp Σ := ℓn ↦□ SOMEV (x, #ℓnOut).

  (* Length indexed reachable *)
  Fixpoint reachable_l (n : nat) ℓn ℓm : iProp Σ :=
    ∃ x (ℓnOut : loc), ℓn ↦□ CONSV x #ℓnOut ∗
      (match n with
       | O => ⌜ℓn = ℓm⌝
       | S n' => (∃ (ℓp : loc), ℓnOut ↦□ #ℓp ∗ reachable_l n
      end).

  Definition reachable ℓn ℓm : iProp Σ := ∃ n, reachable_l

  Notation "a ~r~> b" := (reachable a b) (at level 20, form

  Lemma reachable_refl x (ℓm ℓmOut : loc) : ℓm ↦□ CONSV x #
  Proof. iIntros "p". iExists 0. iExistsFrame. Qed.

  Instance reachable_persistent a b: Persistent (a ~r~> b).
  Proof.
    rewrite /Persistent.
    iDestruct 1 as (n) "R". iInduction n as [|n] "IH" forall (a).
    - iDestruct "R" as "#R". iModIntro. by iExists 0.
```

What did Michael/Scott say?

What did Michael/Scott say?

They describe "scenarios"

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

* Actually this prose from Herlihy/Shavit TAOMPP.

What did Michael/Scott say?

They describe "scenarios"

> *An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.*

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"
"some other thread"
"only then"
"reads tail, and finds the node that appears to be last (Lines 12–13)"
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"
"some other thread"
"only then"
"reads tail, and finds the node that appears to be last (Lines 12–13)"
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."

Important ADT states

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"
"some other thread"
"only then"
"reads tail, and finds the node that appears to be last (Lines 12–13)"
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."

Important ADT states

Concurrent threads

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"	Important ADT states
"some other thread"	Concurrent threads
"only then"	Event Order
"reads tail, and finds the node that appears to be last (Lines 12–13)"	
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."	

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"	Important ADT states
"some other thread"	Concurrent threads
"only then"	Event Order
"reads tail, and finds the node that appears to be last (Lines 12–13)"	Thread-local step sequence
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."	

What did Michael/Scott say?

They describe "scenarios"

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

"queue is nonempty," "tail is lagged"	Important ADT states
"some other thread"	Concurrent threads
"only then"	Event Order
"reads tail, and finds the node that appears to be last (Lines 12–13)"	Thread-local step sequence
"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."	Linearization points

Why can’t proofs be more “scenario” orientated?

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call $\tau_{\mathtt{enq}}$.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call $\tau_{\mathtt{enq}}$.
3. $\tau_{\mathtt{enq}}$ reads the tail and the tail's next pointer.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call $\tau_{\mathtt{enq}}$.
3. $\tau_{\mathtt{enq}}$ reads the tail and the tail's next pointer.
4. $\tau_{\mathtt{enq}}$ finds that tail's next is null.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call $\tau_{\mathtt{enq}}$.
3. $\tau_{\mathtt{enq}}$ reads the tail and the tail's next pointer.
4. $\tau_{\mathtt{enq}}$ finds that tail's next is null.
5. $\tau_{\mathtt{enq}}$ atomically updates tail's next to point to its new node.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call $\tau_{\mathtt{enq}}$.
3. $\tau_{\mathtt{enq}}$ reads the tail and the tail's next pointer.
4. $\tau_{\mathtt{enq}}$ finds that tail's next is null.
5. $\tau_{\mathtt{enq}}$ atomically updates tail's next to point to its new node.
6. The other (unboundedly many) threads fail their CASes on tail's next and restart.

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call τ_{enq}.
3. τ_{enq} reads the tail and the tail's next pointer.
4. τ_{enq} finds that tail's next is null.
5. τ_{enq} atomically updates tail's next to point to its new node.
6. The other (unboundedly many) threads fail their CASes on tail's next and restart.

$$r_{\mathsf{next}} \equiv (\tau \in T : read + \tau_{enq} : read)^* \cdot (\tau_{enq} : \mathsf{cas/succeed}) \cdot (\tau \in T : restart)^*$$

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

1. Unboundedly many threads are reading the data structure.
2. There is a distinguished thread, let's call τ_{enq}.
3. τ_{enq} reads the tail and the tail's next pointer.
4. τ_{enq} finds that tail's next is null.
5. τ_{enq} atomically updates tail's next to point to its ne
6. The other (unboundedly many) threads fail their C .

And two others:

$$r_{\mathsf{tail}} \equiv \cdots$$

$$r_{\mathsf{head}} \equiv \cdots$$

$$r_{\mathsf{next}} \equiv (\tau \in T : read + \tau_{enq} : read)^* \cdot (\tau_{enq} : \mathsf{cas/succeed}) \cdot (\tau \in T : restart)^*$$

Benefits

$$\left(r_{\text{next}} + r_{\text{tail}} + r_{\text{head}}\right)^*$$

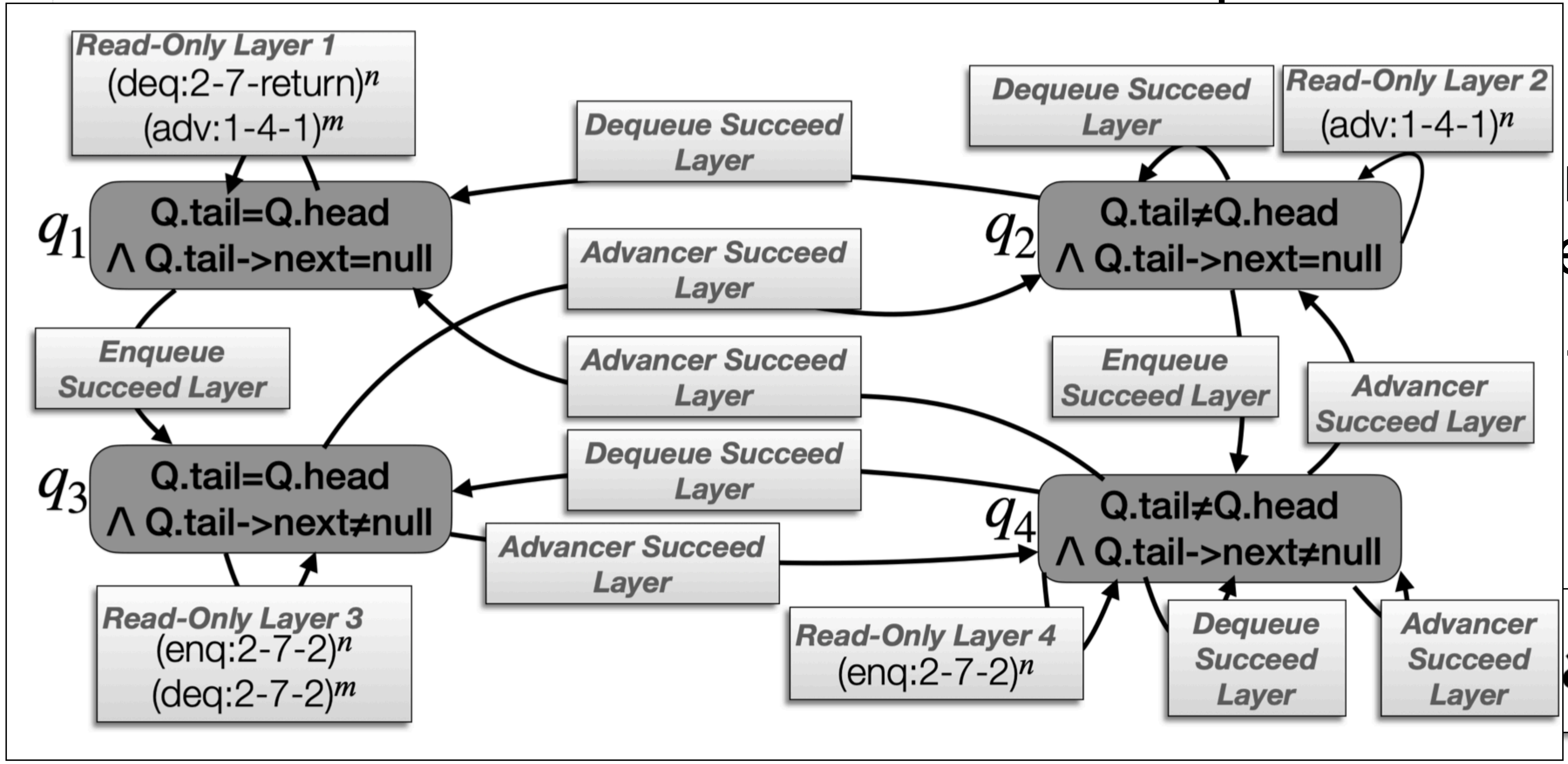
- **Concise.** MSQ's concurrent executions can be represented with these three expressions.
- There are four other expressions but they are event simpler, read-only interleavings.
- **Unbounded.** Interleavings between an unbounded number of enqueuers and dequeuers can be seen as the unbounded alternation $\left(r_{\text{next}} + r_{\text{tail}} + r_{\text{head}}\right)^*$.

$$r_{\mathsf{next}} \equiv (\tau \in T : read + \tau_{enq} : read)^* \cdot (\tau_{enq} : \mathsf{cas/succeed}) \cdot (\tau \in T : restart)^*$$

Benefits

$$\left(r_{\text{next}} + r_{\text{tail}} + r_{\text{head}}\right)^*$$

- **Concise**. MSQ's concurrent executions can be represented with these three expressions.
- There are four other expressions but they are event



n an unbounded

ers can be seen as

$+ r_{\text{tail}} + r_{\text{head}})^* \cdot$

as/succeed) $\cdot$ $(\tau \in T : \mathit{restart})^*$

Some Questions

- Why safe to only discuss seemingly limited scenarios?
- How can we describe such scenarios?
- Later: will it match the prose proofs? Automation?

A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

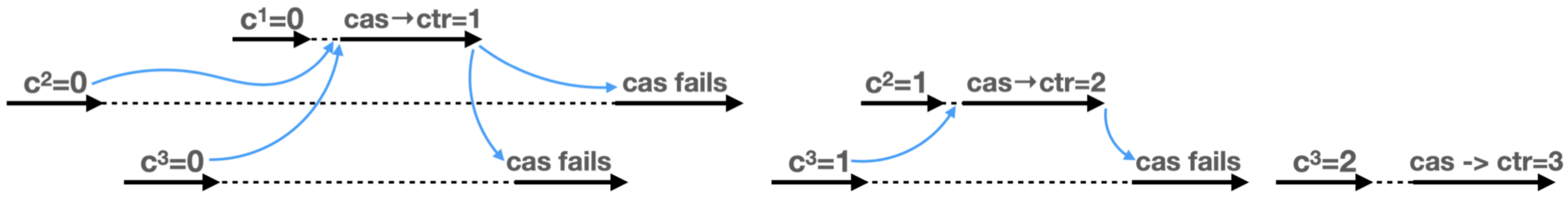
A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

A “nice” execution



A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

A "nice" execution

Three canonical phases

A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

A “nice” execution

Three canonical phases

$$(\tau \in T : c^{\tau} = 0 + c^{\tau'} = 0)^n \cdot (\tau' : \text{cas(ctr,0,1)/true}) \cdot (\tau \in T : \text{cas(ctr,0,1)/false})^n$$

A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

Equivalent to other executions:

e.g. we reorder/swap some actions within a layer

A simpler example

Concurrent Counter

```
int increment() {
    while (true) {
      int c = ctr;
      if (CAS(ctr,c,c+1))
         return c;
    }
}
```

```
int decrement() {
    while (true) {
      int c = ctr;
      if ( c == 0 )
        return 0;
      if (CAS(ctr,c,c-1))
        return c;
    }
}
```

Or to one where we rename threads:

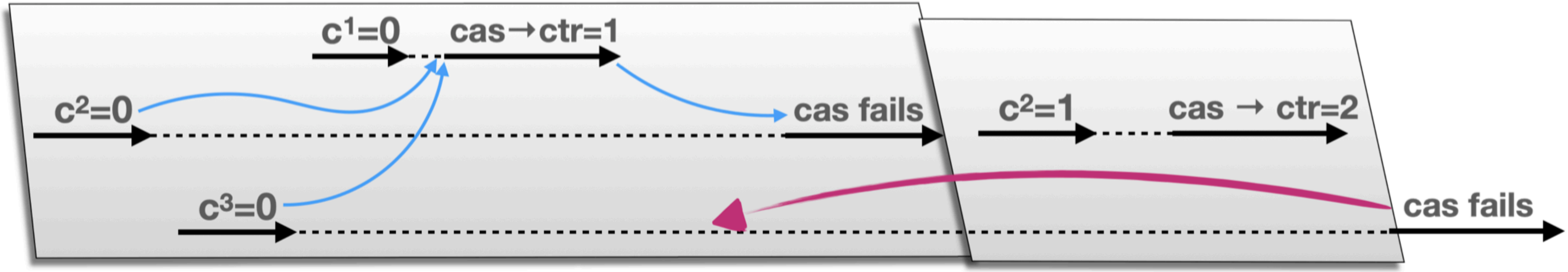
A simpler example

Concurrent Counter

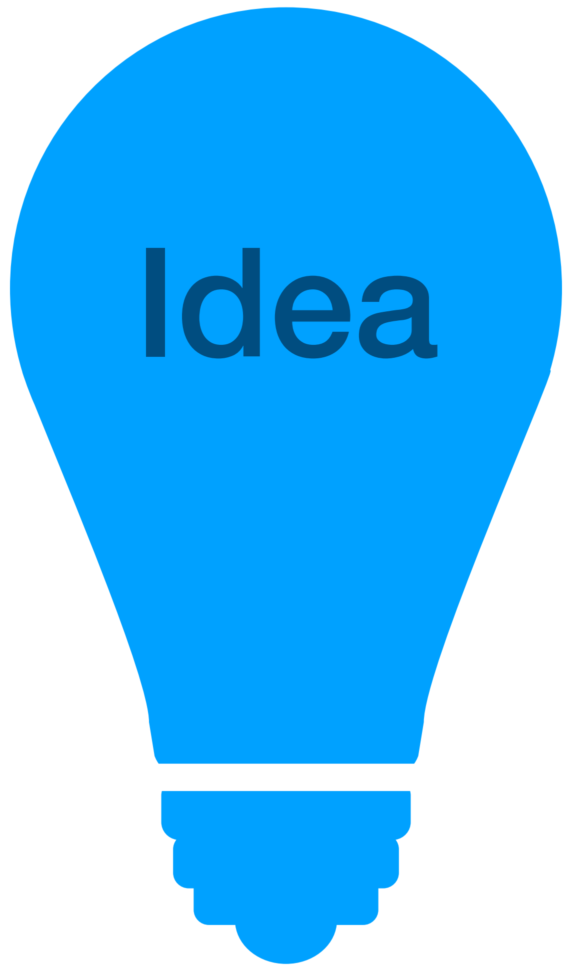
```
int increment() {
    while (true) {
        int c = ctr;
        if (CAS(ctr,c,c+1))
            return c;
    }
}
```

```
int decrement() {
    while (true) {
        int c = ctr;
        if ( c == 0 )
            return 0;
        if (CAS(ctr,c,c-1))
            return c;
    }
}
```

Yet another execution:



… with a "late" cas fail.



Notion of Representative Interleaving.

(Note: each representative interleaving could be equivalent to infinitely many others)

Find a core set—a "quotient"—of such representatives, much easier to work with, e.g., linearizability.

Object Quotient, Semantically

Object Quotient, Semantically

<u>Definition</u>: The *commutativity* ***quotient*** of a concurrent object is a (sub)set of the object's traces $\langle\lfloor O \rfloor\rangle \subset [\![O]\!]$ such that:

Object Quotient, Semantically

<u>Definition</u>: The *commutativity* ***quotient*** of a concurrent object is a (sub)set of the object's traces $\langle\lfloor O \rfloor\rangle \subset [\![O]\!]$ such that:

- Completeness:
  $\forall \tau \in [\![O]\!] \,.\, \exists \tau', \tau'' \,.\, relabel(\tau, \tau') \wedge \tau' \equiv_{\bowtie} \tau'' \wedge \tau'' \in \langle\lfloor O \rfloor\rangle$

Object Quotient, Semantically

Definition: The *commutativity* ***quotient*** of a concurrent object is a (sub)set of the object's traces $\langle\lfloor O \rfloor\rangle \subset [\![O]\!]$ such that:

- Completeness:
  $\forall \tau \in [\![O]\!] \,.\, \exists \tau', \tau'' \,.\, relabel(\tau, \tau') \wedge \tau' \equiv_{\bowtie} \tau'' \wedge \tau'' \in \langle\lfloor O \rfloor\rangle$
- Optimality:
  $\forall \tau, \tau' \in \langle\lfloor O \rfloor\rangle \,.\, \neg(\tau \equiv_{\bowtie} \tau')$

Topics

OOPSLA 2024

- **Quotients, formally.**
- **Expressing quotients.**
- **Automata.**
- **Verifying concurrent objects.**
- **Some automation.**

Scenario-Based Proofs for Concurrent Objects

CONSTANTIN ENEA, LIX - CNRS - École Polytechnique, France
ERIC KOSKINEN, Stevens Institute of Technology, USA

Concurrent objects form the foundation of many applications that exploit multicore architectures and their importance has lead to informal correctness arguments, as well as formal proof systems. Correctness arguments (as found in the distributed computing literature) give intuitive descriptions of a few canonical executions or "scenarios" often each with only a few threads, yet it remains unknown as to whether these intuitive arguments have a formal grounding and extend to arbitrary interleavings over unboundedly many threads.

We present a novel proof technique for concurrent objects, based around identifying a small set of scenarios (representative, canonical interleavings), formalized as the commutativity quotient of a concurrent object. We next give an expression language for defining abstractions of the quotient in the form of regular or context-free languages that enable simple proofs of linearizability. These quotient expressions organize unbounded interleavings into a form more amenable to reasoning and make explicit the relationship between implementation-level contention/interference and ADT-level transitions.

We evaluate our work on numerous non-trivial concurrent objects from the literature (including the Michael-Scott queue, Elimination stack, SLS reservation queue, RDCSS and Herlihy-Wing queue). We show that quotients capture the diverse features/complexities of these algorithms, can be used even when linearization points are not straight-forward, correspond to original authors' correctness arguments, and provide some new scenario-based arguments. Finally, we show that discovery of some object's quotients reduces to two-thread reasoning and give an implementation that can derive candidate quotients expressions from source code.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Logic and verification**; **Program reasoning**; • **Computing methodologies** → **Concurrent algorithms**.

Additional Key Words and Phrases: verification, linearizability, commutativity quotient, concurrent objects

ACM Reference Format:
Constantin Enea and Eric Koskinen. 2024. Scenario-Based Proofs for Concurrent Objects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 140 (April 2024), 30 pages. https://doi.org/10.1145/3649857

1 INTRODUCTION

Efficient multithreaded programs typically rely on optimized implementations of common abstract data types (ADTs) like stacks, queues, and sets, whose operations execute in parallel to maximize efficiency. Synchronization between operations must be minimized to increase throughput [Herlihy and Shavit 2008]. Yet this minimal amount of synchronization must also be adequate to ensure that operations behave as if they were executed atomically, so that client programs can rely on their (sequential) ADT specification; this de-facto correctness criterion is known as *linearizability* [Herlihy and Wing 1990]. These opposing requirements, along with the general challenge in reasoning about interleavings, make concurrent data structures a ripe source of insidious programming errors.

Algorithm designers (*e.g.,* researchers defining new concurrent objects) argue about correctness by considering some number of "scenarios", *i.e.,* interesting ways of interleaving steps of different

Authors' addresses: Constantin Enea, LIX - CNRS - École Polytechnique, Paris, France, cenea@lix.polytechnique.fr; Eric

Expressing Quotients

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

Expressing Quotients

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

Sequence of labels performed by one thread

$$[\![x := v \cdot x{+}{+}]\!]$$

Expressing Quotients

$$\mathrm{expr} = \omega \mid \omega_1^n \cdot \mathrm{expr} \cdot \omega_2^n \mid \mathrm{expr}^* \mid \mathrm{expr} + \mathrm{expr} \mid \mathrm{expr} \cdot \mathrm{expr}$$

Sequence of labels performed by one thread

$$\{(t : \mathtt{x:=v}), (t : \mathtt{x++})\} \in [\![\mathtt{x:=v} \cdot \mathtt{x++}]\!]$$

Expressing Quotients

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

for every application of this production,
n is a fresh variable not occurring in expr

$$((\texttt{c:=ctr})_{inc})^n \cdot$$
$$(\texttt{c:=ctr}) \cdot \langle\!\langle [\texttt{c=ctr}] \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \text{ret}(\texttt{c}) \cdot$$
$$\left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$$

Expressing Quotients

$$\mathsf{expr} = \omega \mid \omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n \mid \mathsf{expr}^* \mid \mathsf{expr} + \mathsf{expr} \mid \mathsf{expr} \cdot \mathsf{expr}$$

for every application of this production,
n is a fresh variable not occurring in expr

$$\left((\texttt{c:=ctr})_{inc}\right)^n \cdot$$
$$(\texttt{c:=ctr}) \cdot \langle\!\langle[\texttt{c=ctr}] \cdot \texttt{ctr:=c+1}\rangle\!\rangle \cdot \mathsf{ret}(\texttt{c}) \cdot$$
$$\left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$$

$\in$

Expressing Quotients

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

for every application of this production,
n is a fresh variable not occurring in expr

$$\left((\texttt{c:=ctr})_{inc}\right)^n \cdot (\texttt{c:=ctr}) \cdot \langle\!\langle [\texttt{c=ctr}] \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \mathsf{ret}(\texttt{c}) \cdot \left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$$

$$\in$$

$$t_2 : (c := ctr) \cdot t_3 : (c := ctr) \cdot (t_1 : (c := ctr) \cdot t_1 : \langle [c = ctr] \cdot ctr := c + 1 \rangle \cdot t_1 : \mathsf{ret}(0)) \cdot t_3 : \overline{[c = ctr]} \cdot t_2 : \overline{[c = ctr]} \cdot$$
$$t_3 : (c := ctr) \cdot t_2 : (c := ctr) \cdot t_2 : \langle [c = ctr] \cdot ctr := c + 1 \rangle \cdot t_2 : \mathsf{ret}(1) \cdot t_3 : \overline{[c = ctr]} \cdot$$
$$t_3 : (c := ctr) \cdot t_3 : \langle [c = ctr] \cdot ctr := c + 1 \rangle \cdot t_3 : \mathsf{ret}(2)$$

Expressing Quotients

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

… and other usual KAT constructors

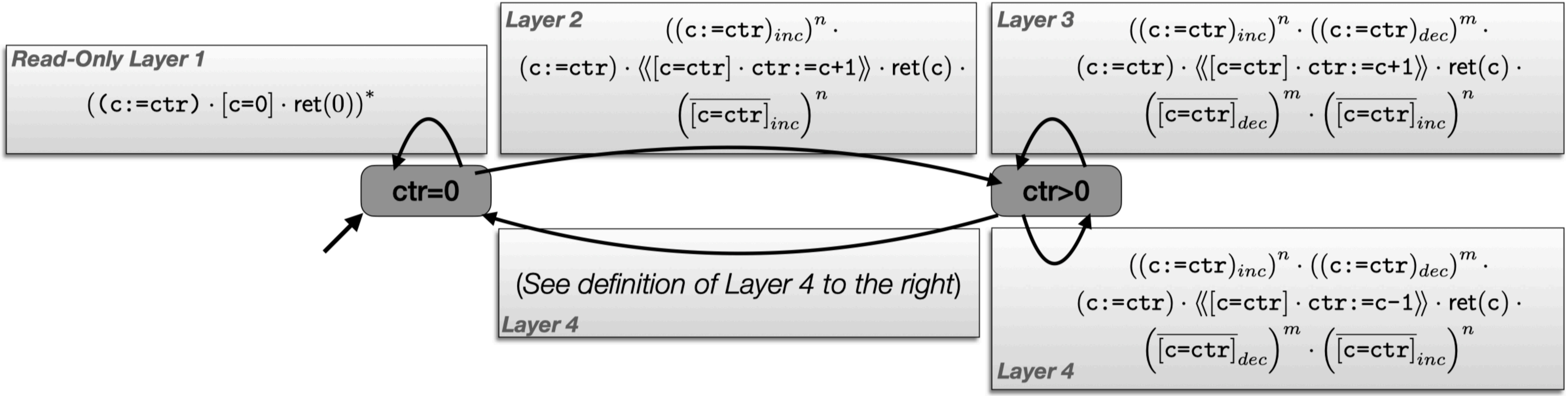
Expressing Quotients with Automata

- More refined
- Paths are sometimes infeasible

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

Expressing Quotients with Automata

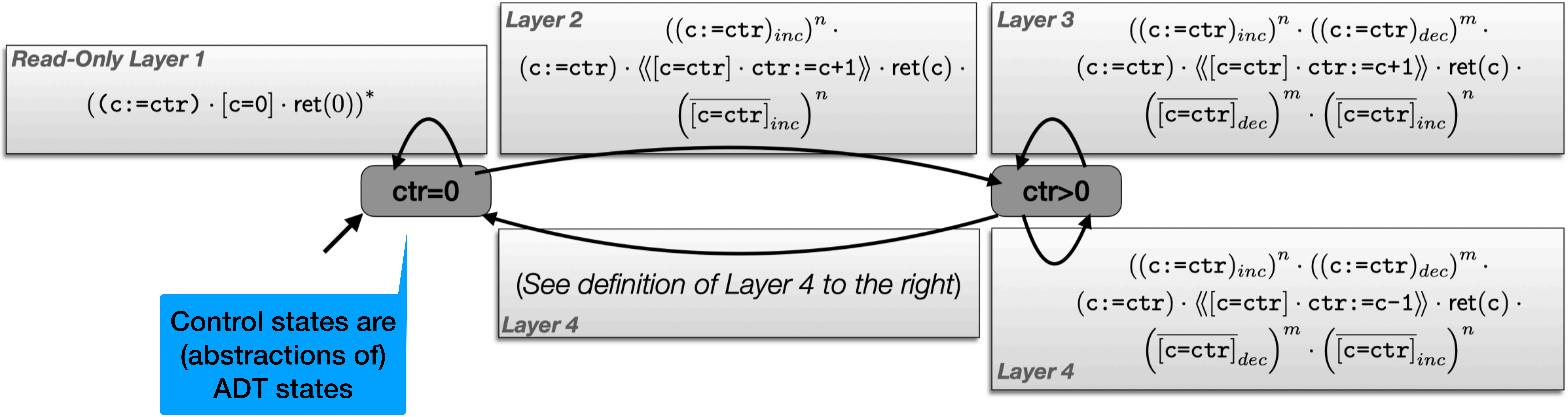
- **More refined**
- **Paths are sometimes infeasible**



$$\mathrm{expr} = \omega \mid \omega_1^n \cdot \mathrm{expr} \cdot \omega_2^n \mid \mathrm{expr}^* \mid \mathrm{expr} + \mathrm{expr} \mid \mathrm{expr} \cdot \mathrm{expr}$$

Expressing Quotients with Automata

- **More refined**
- **Paths are sometimes infeasible**

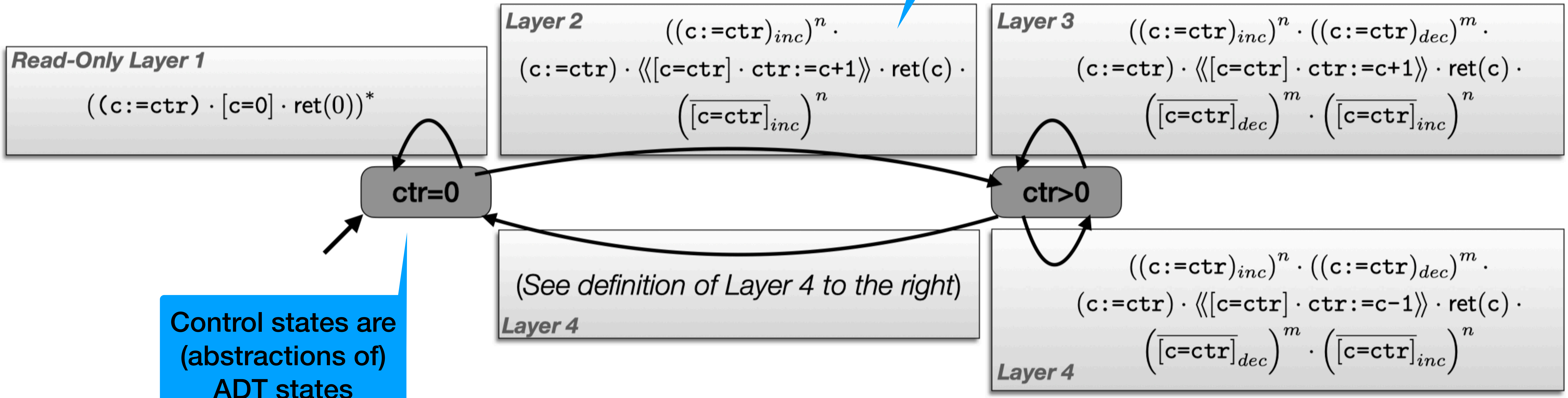


$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

Expressing Quotients with Automata

- More refined
- Paths are sometimes infeasible

Arcs are labeled with expressions



Control states are (abstractions of) ADT states

$$\text{expr} = \omega \mid \omega_1^n \cdot \text{expr} \cdot \omega_2^n \mid \text{expr}^* \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr}$$

Evaluation

Michael/Scott [1996] Queue

SLS Queue [2006]

Harris et al RDCSS [2002]

Hendler et al Elim. Stack [2004]

Herlihy/Wing [1990] Queue

Evaluation

Michael/Scott [1996] Queue

q_1 Q.tail=Q.head ∧ Q.tail->next=null

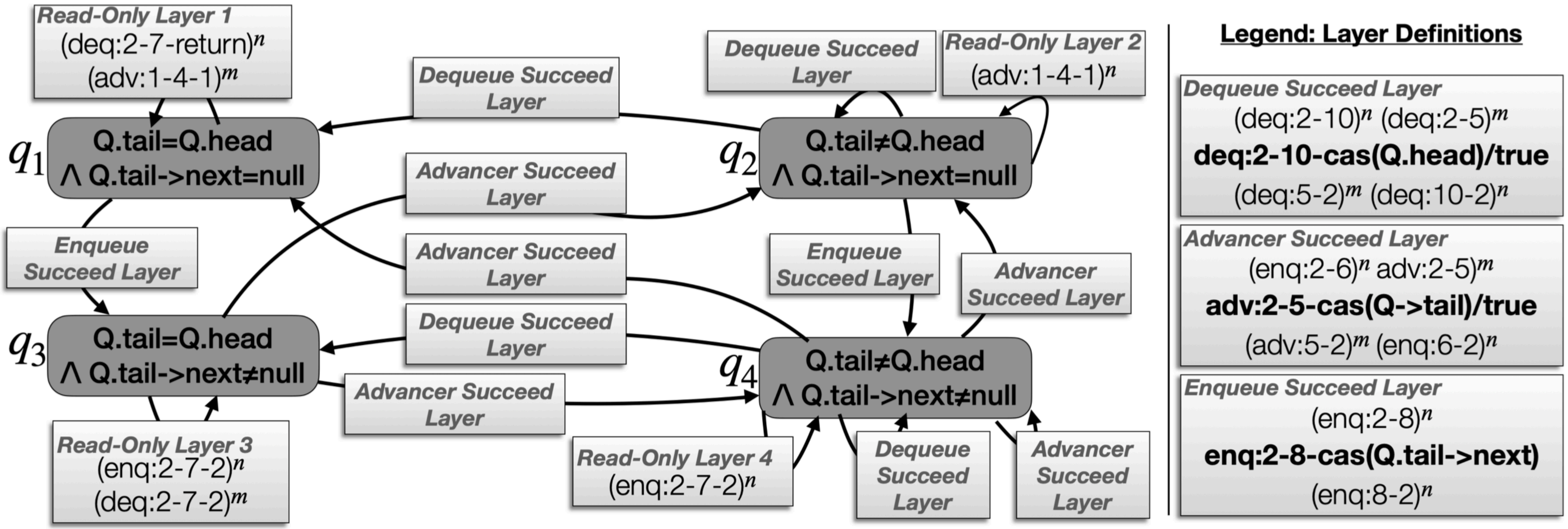
q_2 Q.tail≠Q.head ∧ Q.tail->next=null

q_3 Q.tail=Q.head ∧ Q.tail->next≠null

q_4 Q.tail≠Q.head ∧ Q.tail->next≠null

Evaluation

Michael/Scott [1996] Queue



Evaluation

Michael/Scott [1996] Queue

Proof Element	Herlihy and Shavit [2008]	Quotient Proof
ADT states	"queue is nonempty," "tail is lagged"	ADT states, e.g. (`Q.tail=Q.head` $\wedge$ `Q.tail->next` $\neq$ `null`)
Concurrent threads	"some other thread"	Superscripting $(\ldots)^n$
Event order	"only then"	Arcs in the quo automaton
Thread-local step seq.	"reads tail, and finds the node that appears to be last (Lines 12–13)"	Layer paths, e.g., enq:2-6
Linearization pts.	"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."	The successful CAS in the Dequeue Succeed Layer or Read-Only Layer 1

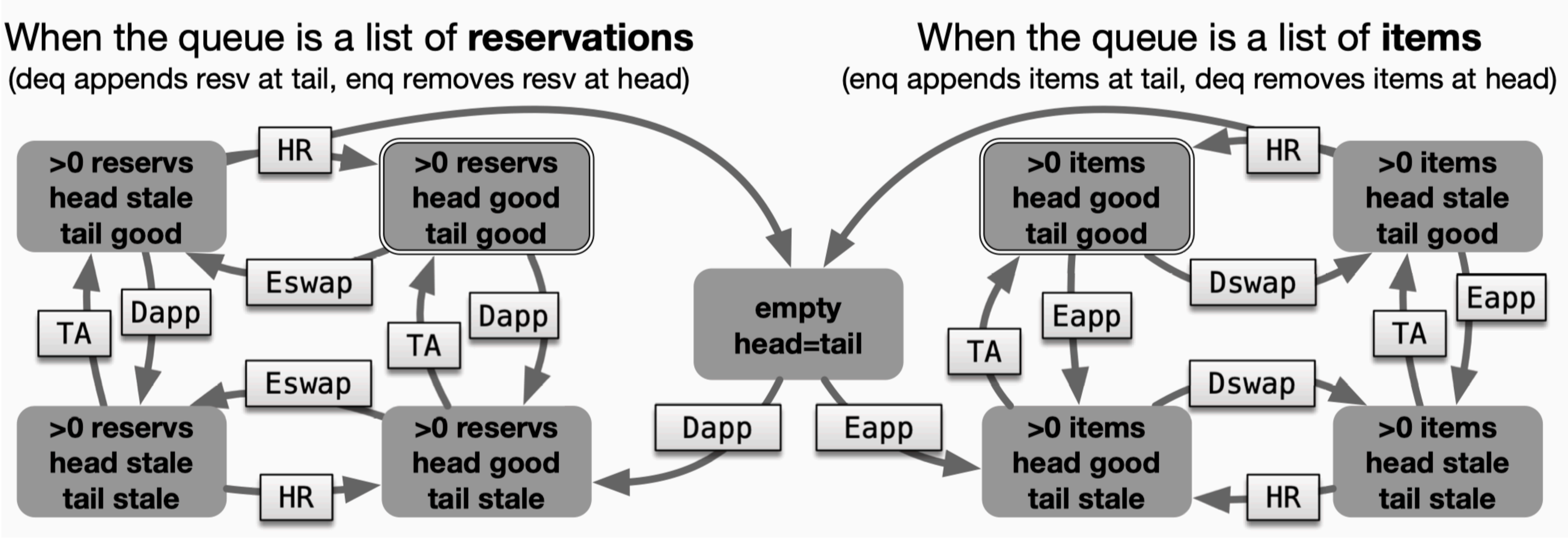
Evaluation

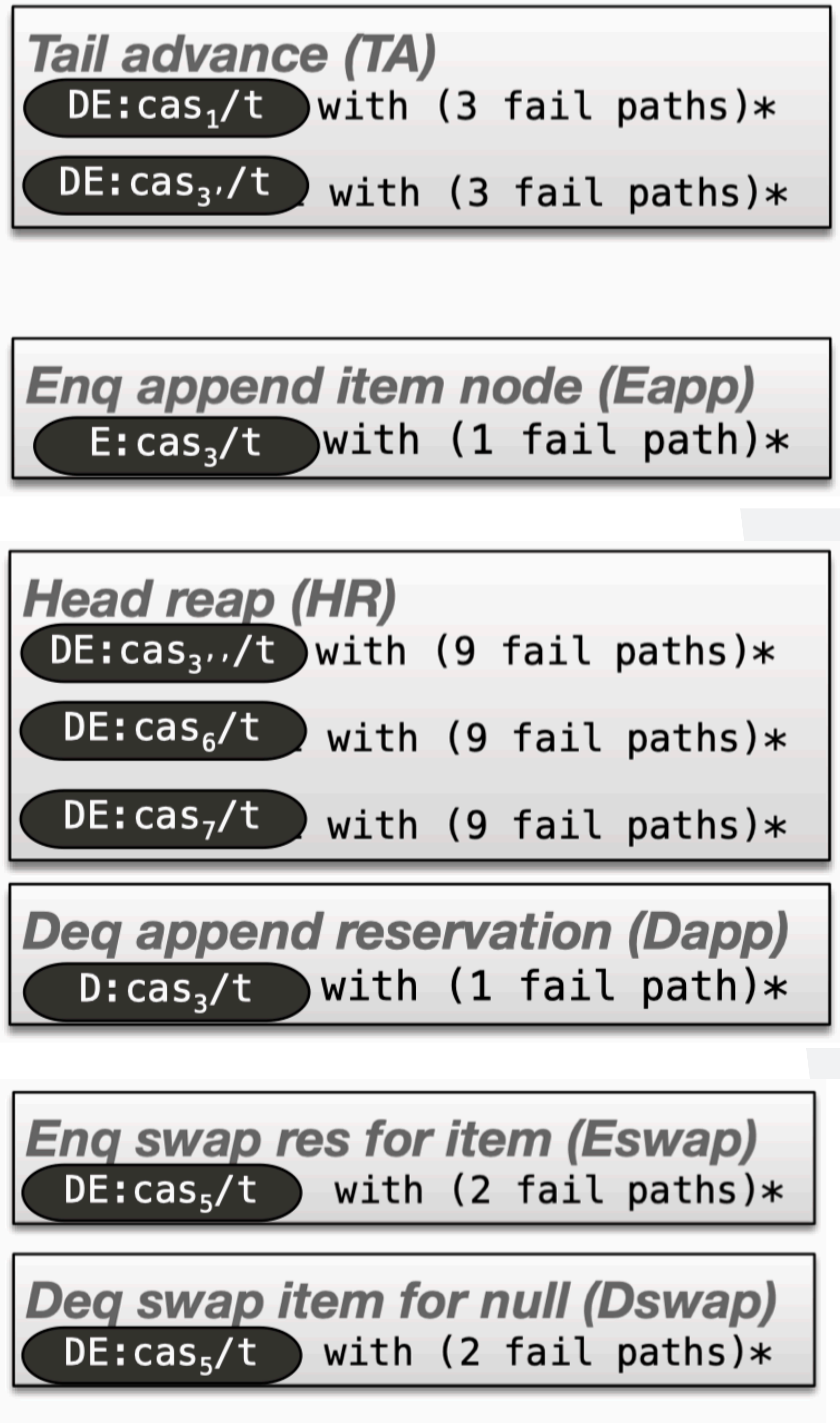
SLS Queue [2006]

- **Synchronous:** threads block on dequeue
- **Reservations:** When queue has no elements (but waiting threads) it becomes a queue of reservations.
- **Implementation has multiple writes for a single invocation.**
- **Linearizability:** LPs must account for dequeuers arriving before their corresponding enqueuer.

Evaluation

SLS Queue [2006]





Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeatedly scans the array looking for the first non-empty slot in a doubly-nested loop.

Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeatedly scans the array looking for the first non-empty slot in a doubly-nested loop.
- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeatedly scans the ar … r the first non-empty slot in a doubly-nested loop.

Some enqueuers increments back

- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeatedly scans the ar … empty slot in a doubly-nested loop.
- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

Some enqueuers increments **back**

(Maybe) some enq's writes a slot

Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeatedly scans the ar
  slot in a doubly-nested loop.
- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

Some enqueuers increments **back**

(Maybe) some enq's writes a slot

dequeue scans that succeed

Evaluation

Herlihy/Wing [1990] Queue

- **Linearizability:** Depend on the future! Not fixed.
- An array of slots for items, with a shared variable **back**
- enq atomically reads and increments back and then later stores a value at that location.
- deq repeate… a… slot in a doubly-nested loop.

dequeue scans that need to restart

Some enqueuers increments **back**

(Maybe) some enq's writes a slot

dequeue scans that succeed

- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

Evaluation

Michael/Scott [1996] Queue

- Many cas operations
- LP helping

SLS Queue [2006]

- Synchronous
- Multiple writes
- LP helping

Herlihy/Wing [1990] Queue

- Future-dependent LPs

Harris et al RDCSS [2002]

- Multiple CAS steps
- Phases

Hendler et al Elim. Stack [2004]

- Elimination
- Submodule: Treiber's stack
- LP of one happens in another (helping)

Generating Quotient Automata

Generating Quotient Automata

- **MSQ and Treiber Stack have a certain structure**
- **Enumerate the "local paths" and the "write paths"**
- **Compute automaton ADT states**: boolean combinations of weakest preconditions)
- **Compute automaton edges**: whenever q implies precondition of a write path, compute every q' and each local path that is possible due to the write path. Create layer edge $q \xrightarrow{\lambda} q'$.

Generating Quotient Automata

- **Implemented in CIL, using Ultimate Automizer**
- **Automatically generated automata for a few examples:**

Example	States $\lvert Q \rvert$	# Paths # k_l	# k_w	# Trans. $\lvert\delta\rvert$	# Layers $\lvert\Lambda(O)\rvert$	Time (s)	# Solver Queries
`evenodd.c`	2	2	2	6	3	52.2	32
`counter.c`	2	3	2	6	5	67.8	36
`descriptor.c`	4	6	2	6	6	160.2	74
`treiber.c`	2	3	2	6	5	71.4	37
`msq.c`	4	9	3	17	7	441.6	314
`listset.c`	7	6	2	59	7	603.8	494

Conclusion

- **Working with representative interleavings (the quotient) is easier than working with all interleavings.**
- **Quotient can be expressed by simple context-free expressions**
- **Applies to a variety of objects** (MSQ, SLS, HWQ, Treiber, Elim)
- **Can be automated for some; open questions…**

Open Questions

- How to automate other concurrent objects?
- How to mechanize checking completeness of a quotient
- How to generate quotient expressions more generally

Thank you!

STEVENS
INSTITUTE OF TECHNOLOGY
1870

OOPSLA 2024

Scenario-Based Proofs for Concurrent Objects

CONSTANTIN ENEA, LIX - CNRS - École Polytechnique, France
ERIC KOSKINEN, Stevens Institute of Technology, USA

Concurrent objects form the foundation of many applications that exploit multicore architectures and their importance has lead to informal correctness arguments, as well as formal proof systems. Correctness arguments (as found in the distributed computing literature) give intuitive descriptions of a few canonical executions or “scenarios” often each with only a few threads, yet it remains unknown as to whether these intuitive arguments have a formal grounding and extend to arbitrary interleavings over unboundedly many threads.

We present a novel proof technique for concurrent objects, based around identifying a small set of scenarios (representative, canonical interleavings), formalized as the commutativity quotient of a concurrent object. We next give an expression language for defining abstractions of the quotient in the form of regular or context-free languages that enable simple proofs of linearizability. These quotient expressions organize unbounded interleavings into a form more amenable to reasoning and make explicit the relationship between implementation-level contention/interference and ADT-level transitions.

We evaluate our work on numerous non-trivial concurrent objects from the literature (including the Michael-Scott queue, Elimination stack, SLS reservation queue, RDCSS and Herlihy-Wing queue). We show that quotients capture the diverse features/complexities of these algorithms, can be used even when linearization points are not straight-forward, correspond to original authors’ correctness arguments, and provide some new scenario-based arguments. Finally, we show that discovery of some object’s quotients reduces to two-thread reasoning and give an implementation that can derive candidate quotients expressions from source code.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Logic and verification**; **Program reasoning**; • **Computing methodologies** → **Concurrent algorithms**.

Additional Key Words and Phrases: verification, linearizability, commutativity quotient, concurrent objects

ACM Reference Format:
Constantin Enea and Eric Koskinen. 2024. Scenario-Based Proofs for Concurrent Objects. *Proc. ACM Program.*

Extra Slides

The ABA problem

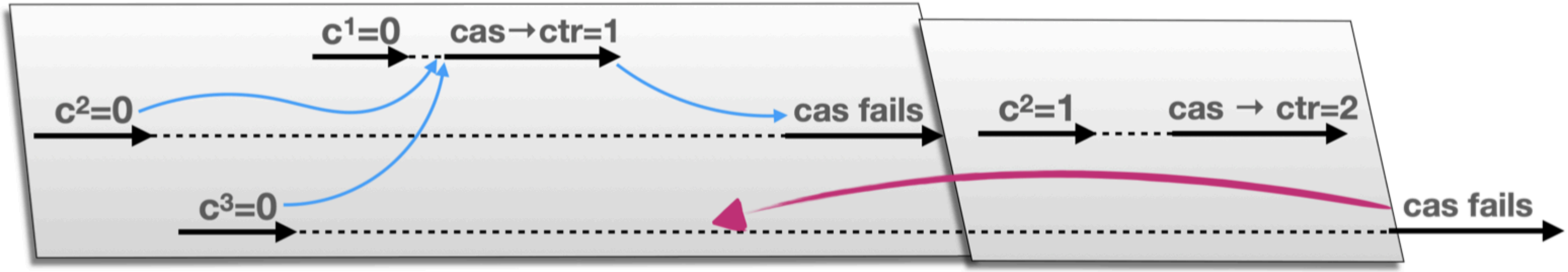


Fig. 8. An increment-only execution for which there is an equivalent representative execution (as suggested by the large wavy arrow) that is in the layer quotient.

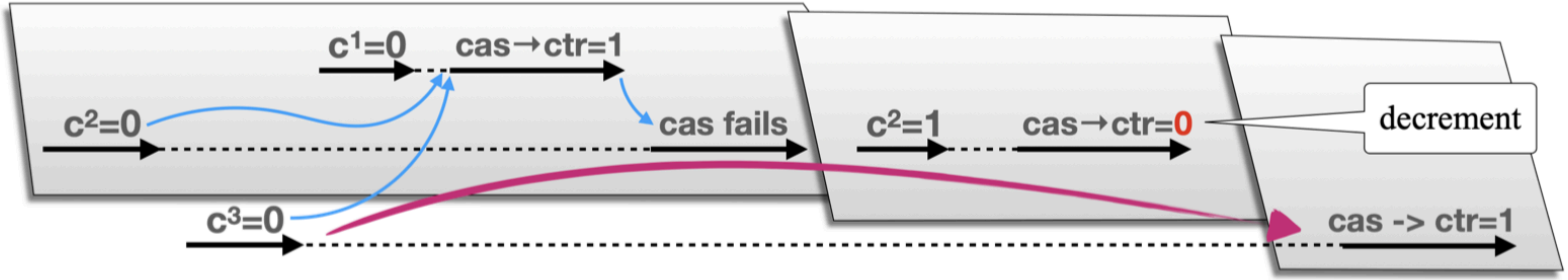


Fig. 9. An execution where the second thread executes a decrement, which is equivalent to the representative execution suggested by the wavy arrow.