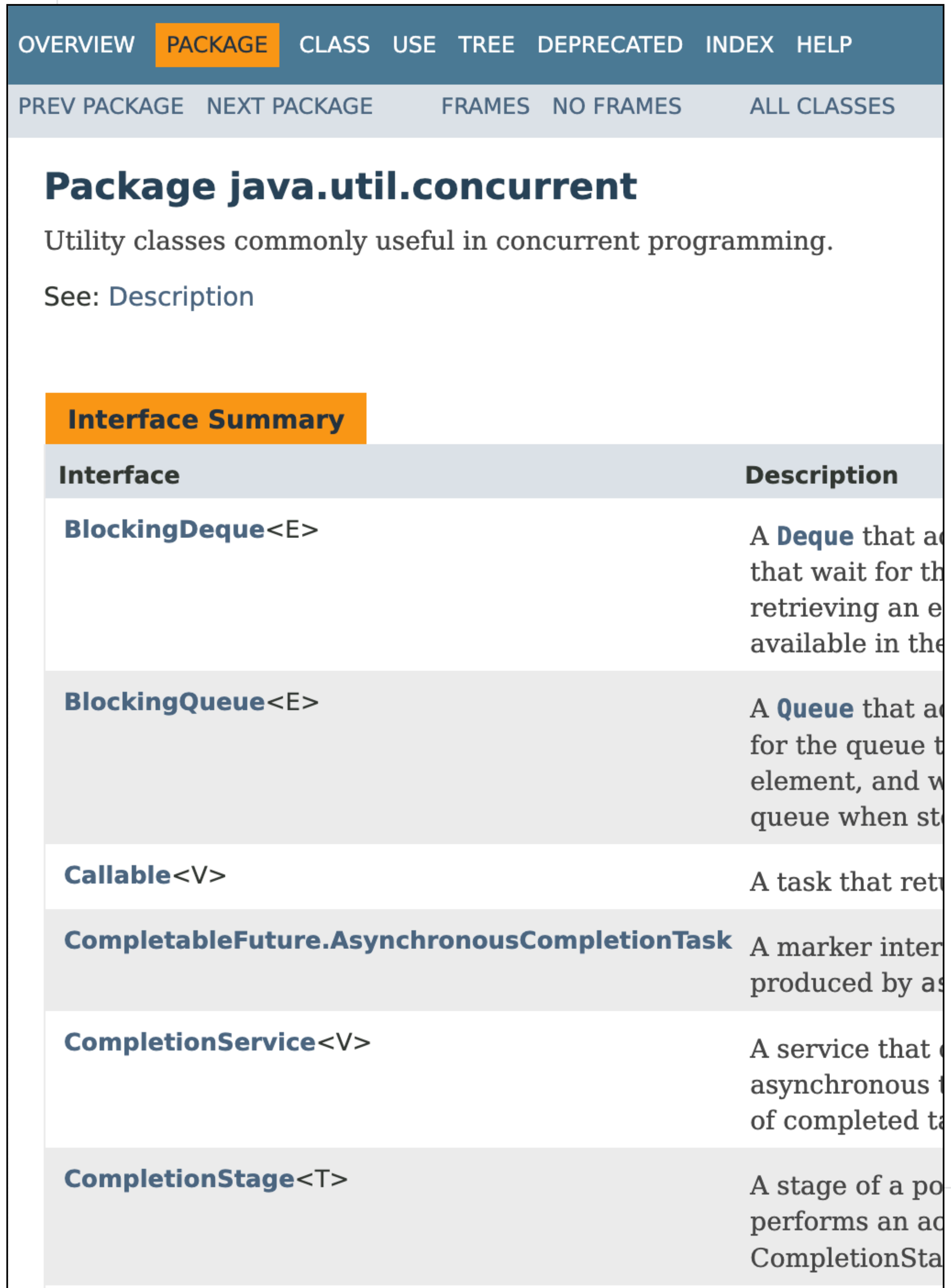
Scenario-Based Proofs for Concurrent Objects

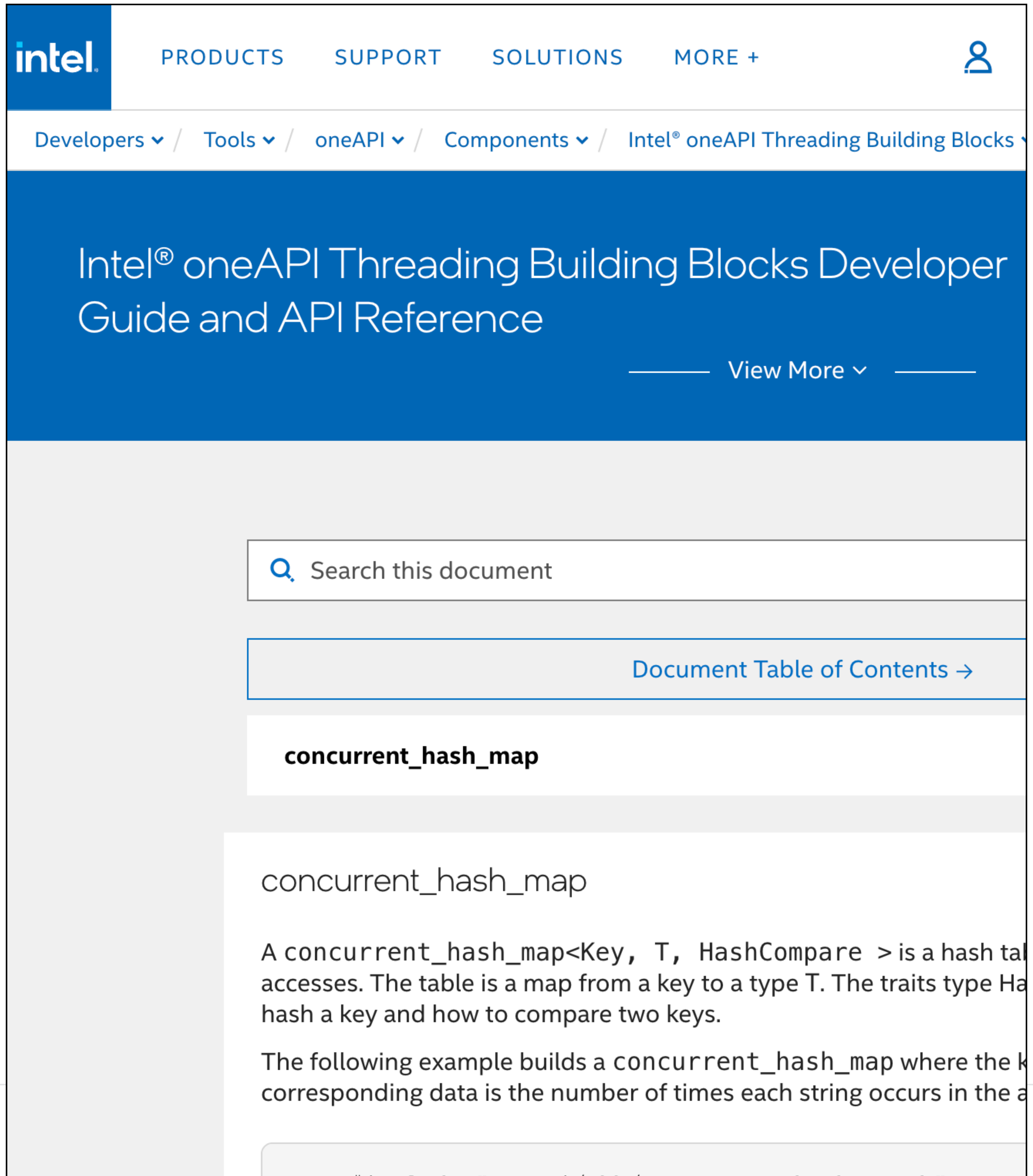
Constantin Enea
LIX - CNRS - École Polytechnique
France

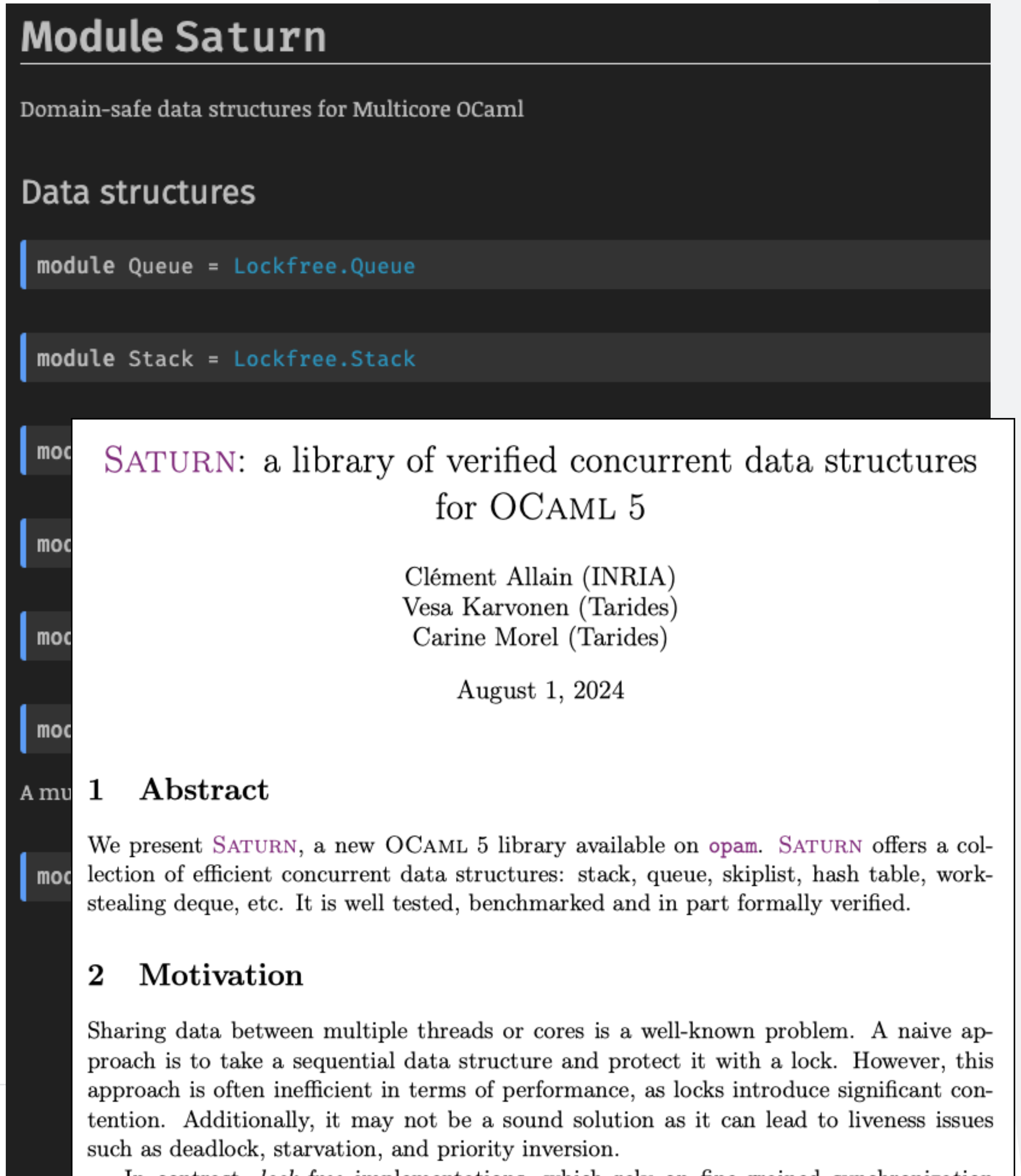
Eric Koskinen
Stevens Institute of Technology
United States

Proceedings of the ACM on Programming Languages (OOPSLA)
October 24, 2024.

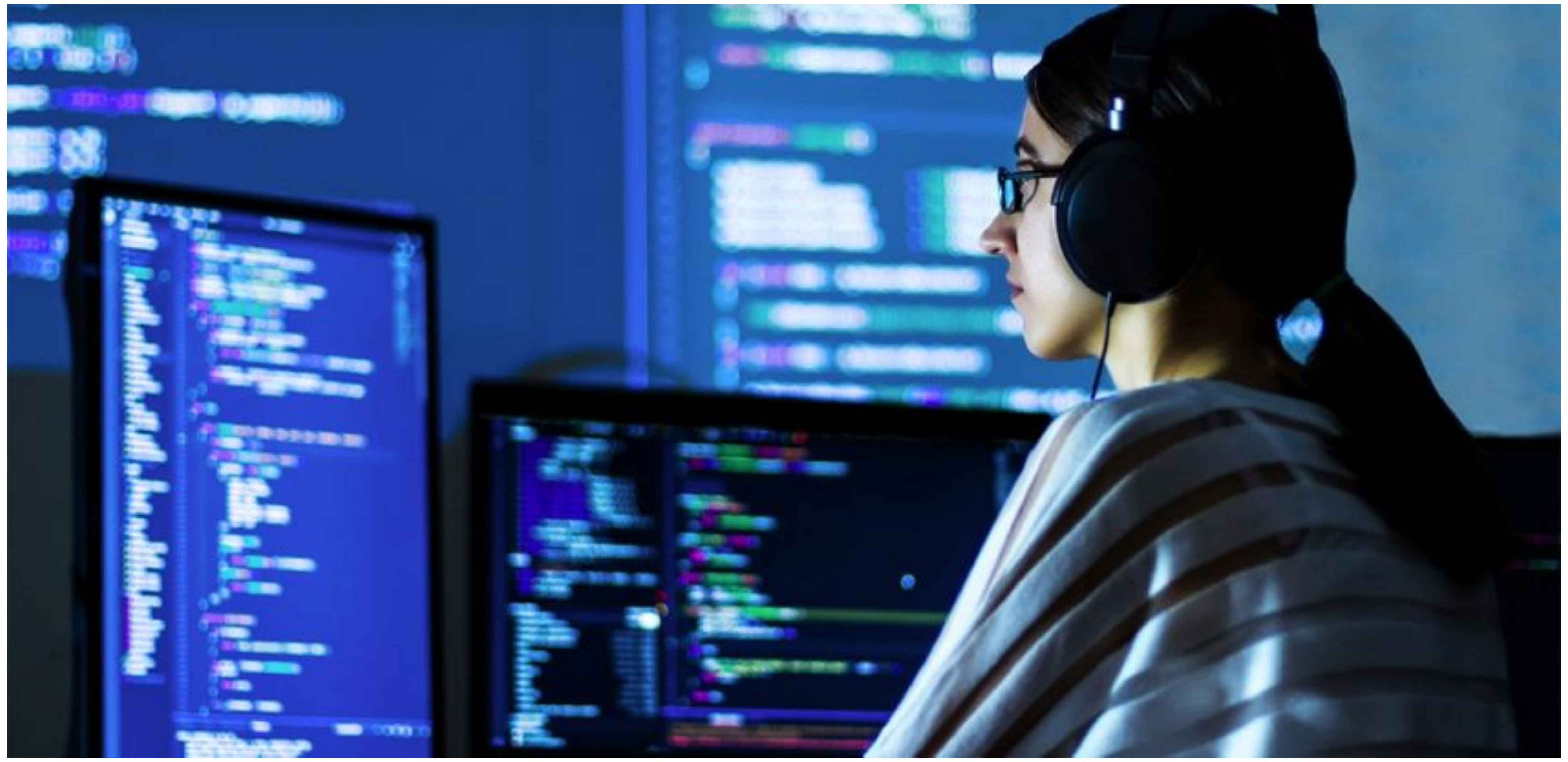
Concurrent Objects

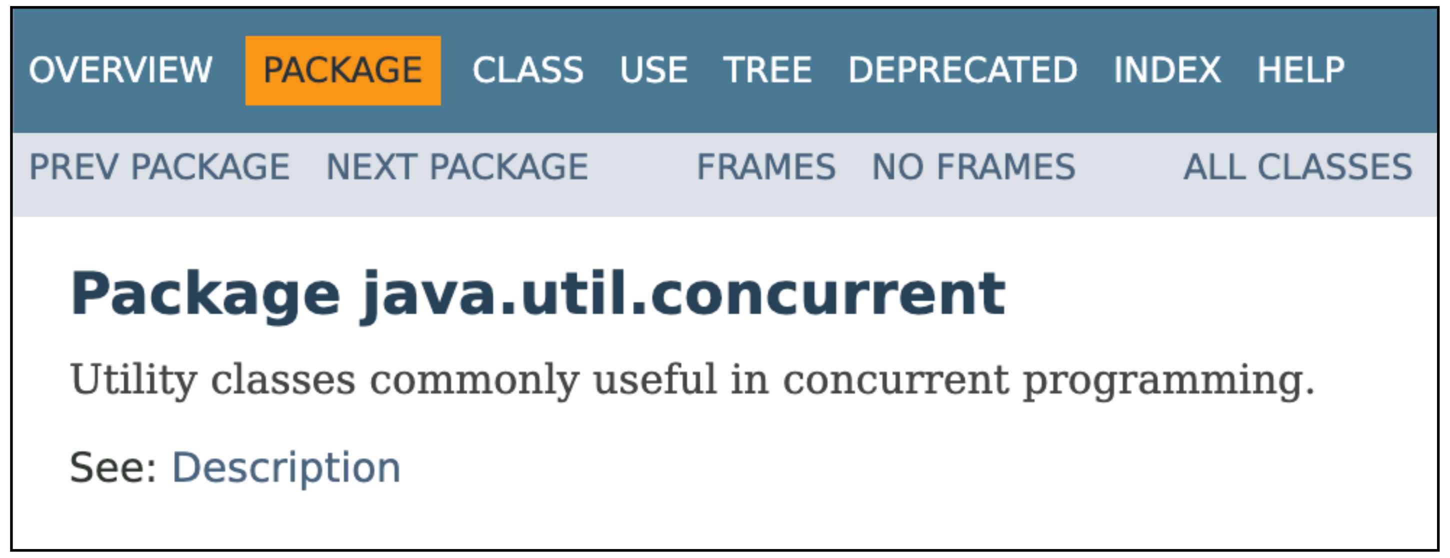


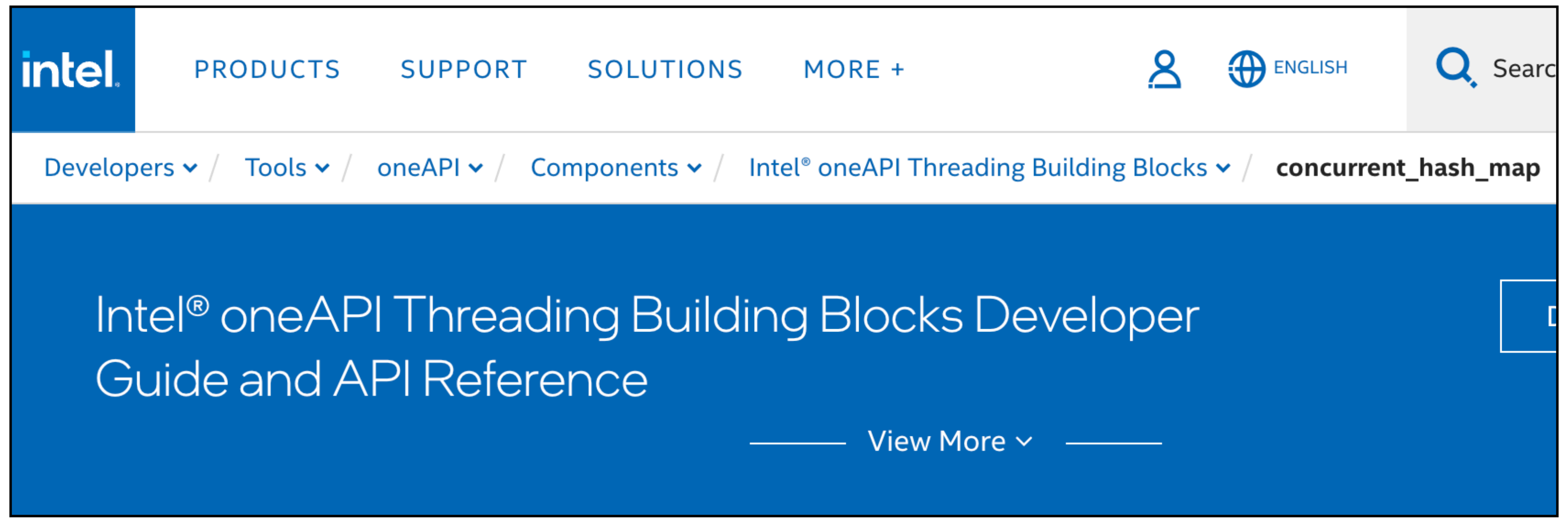


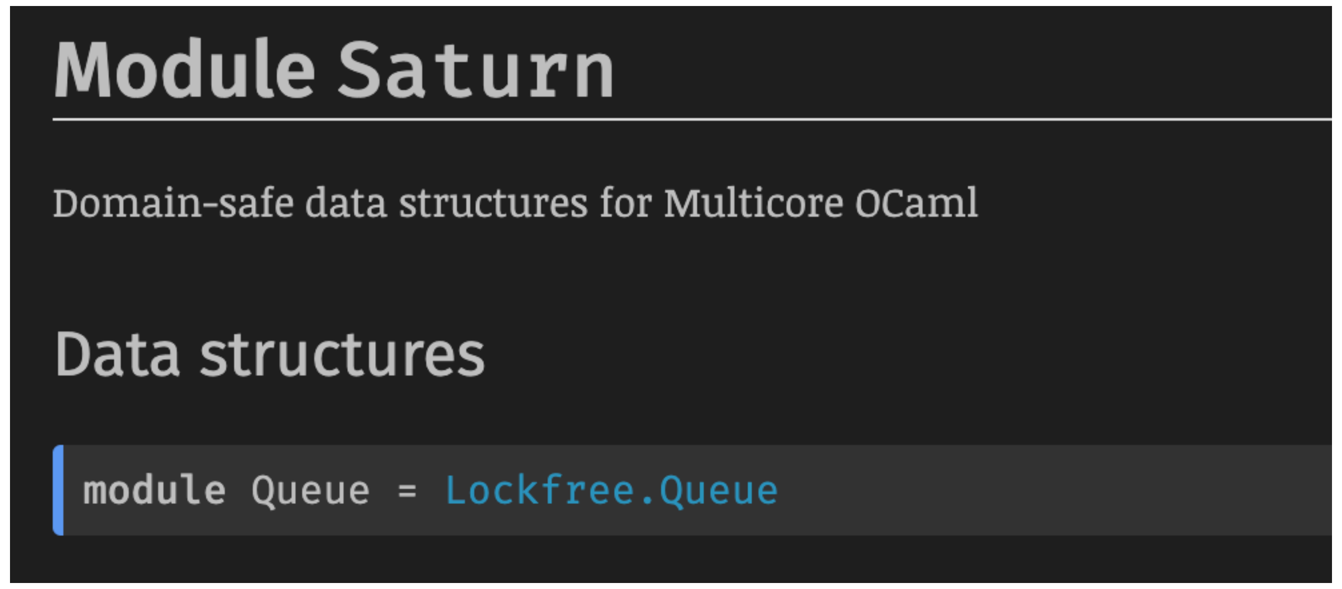


Concurrent Objects









Canonical Concurrent Objects

Michael/Scott Queue	Maged Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. PODC 1996.
SLS Queue	William Scherer III, Doug Lea, and Michael L. Scott. "Scalable synchronous queues." PPoPP 2006.
Treiber's Stack	R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
Hendler *et al.* Elim. Stack	Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. SPAA 2004.
Harris *et al.* RDCSS	Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. DISC 2002.
Herlihy/Wing Queue	Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programing Languages and Systems 1990.

Canonical 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-

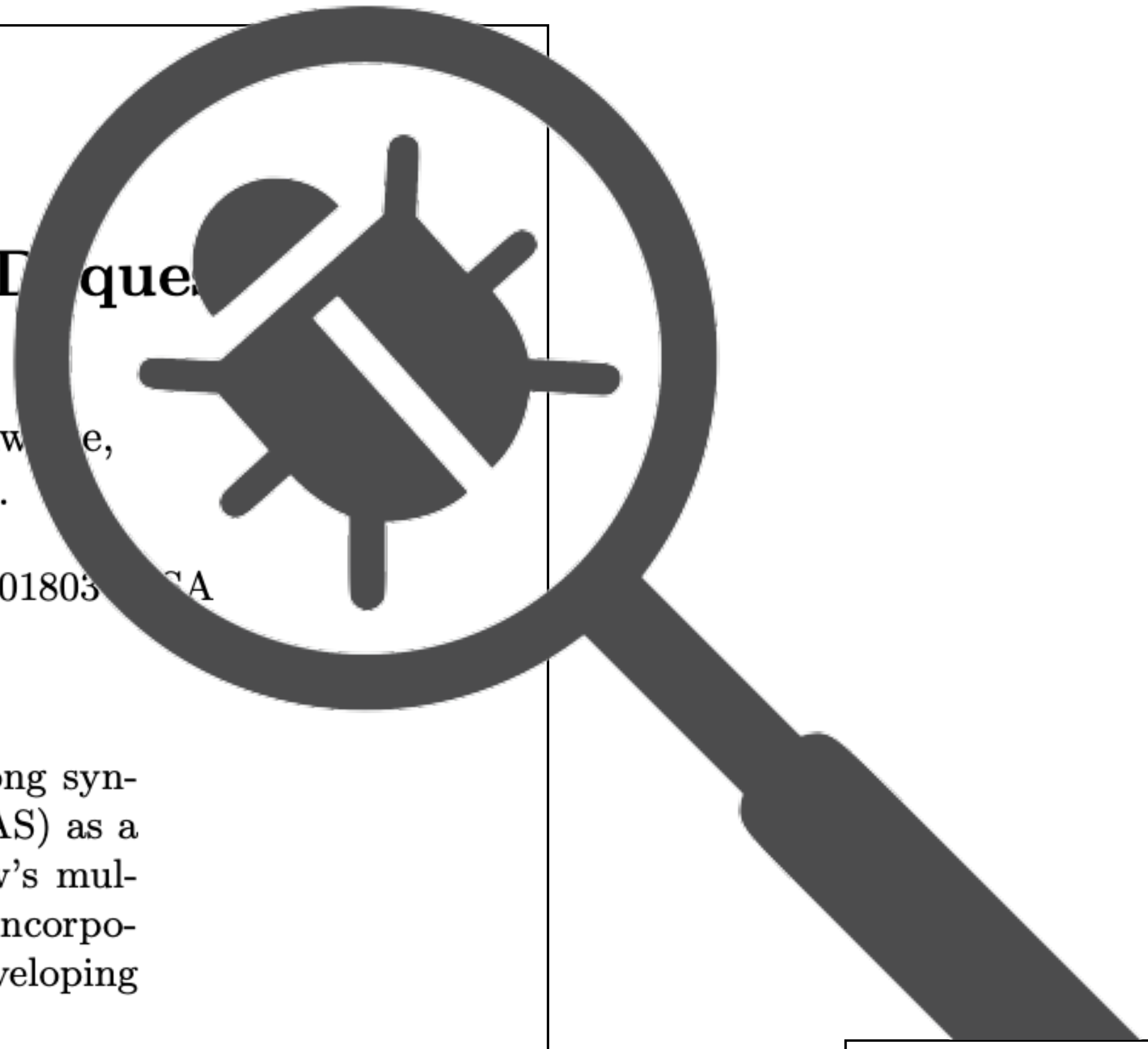
Canonical Concurrent Objects

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 SA

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

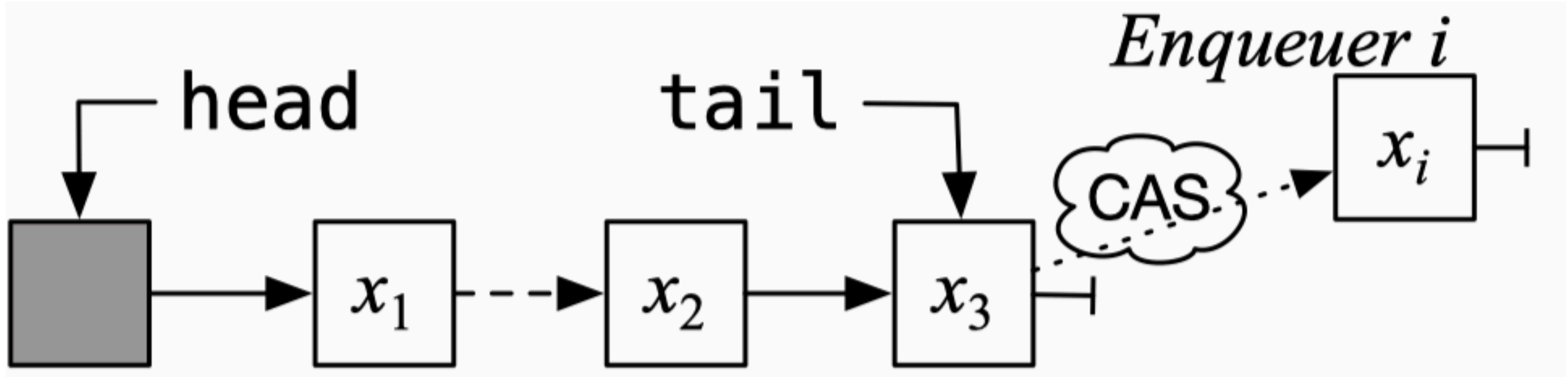
Despite years of research, the design of efficient nonblocking algorithms remains difficult. A key reason is that current shared-memory multiprocessor architectures support only single-location synchronisation primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). Recently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

Canonical Concurrent Objects

How do authors argue for correctness? 🤔




```
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;
} } } }
```

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.

```
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;
} } } }
```

> *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.

```
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;
} } } }
```

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 $\tau_{\mathtt{enq}}$.

```
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;
} } } }
```

> *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.

```
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;
} } } }
```

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.

```
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;
} } } }
```

> *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.

```
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;
} } } }
```

> *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.

```
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;
} } } }
```

> *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. …l.
5. …ext to point to its new node.
6. …) threads fail their CASes on

Quotient Expression

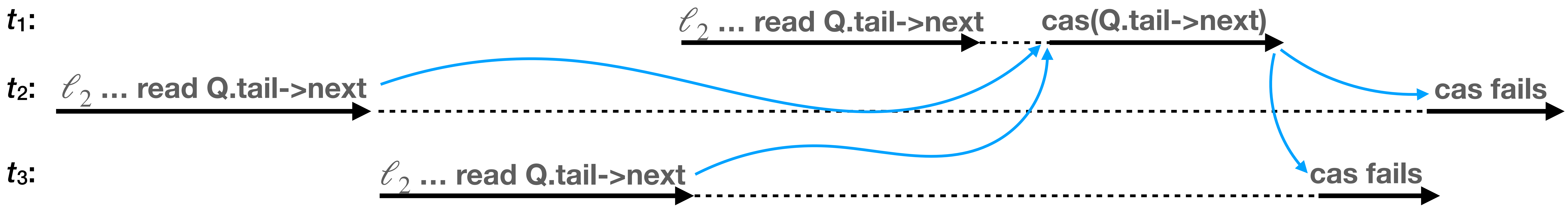
$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

```
1  int enq(int v){ loop {
2   node_t *node=...;
3   node->val=v;
4   tail=Q.tail;
5   next=tail->next;
6   if (Q.tail==tail) {
7    if (next==null) {
8     if (CAS(&tail->next,
9           next,node))
10          ret 1;
11 } } } }
```

> *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.*



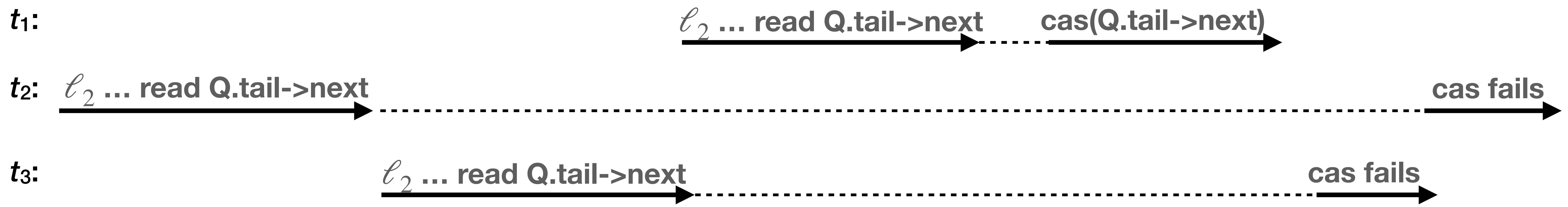
Quotient Expression

$$(\ell_2 \cdots \ell_8)^N$$
$$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$$
$$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$$

Canonical interpretation for 3 threads.

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS | SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

> *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.*



Quotient Expression

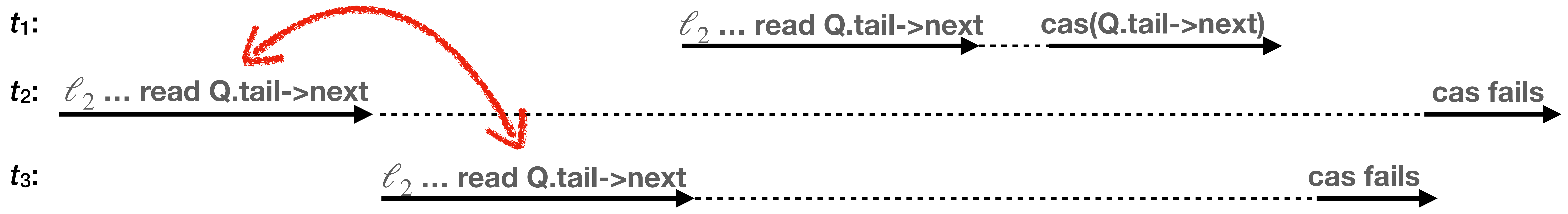
$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Canonical interpretation for 3 threads.

> *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.*

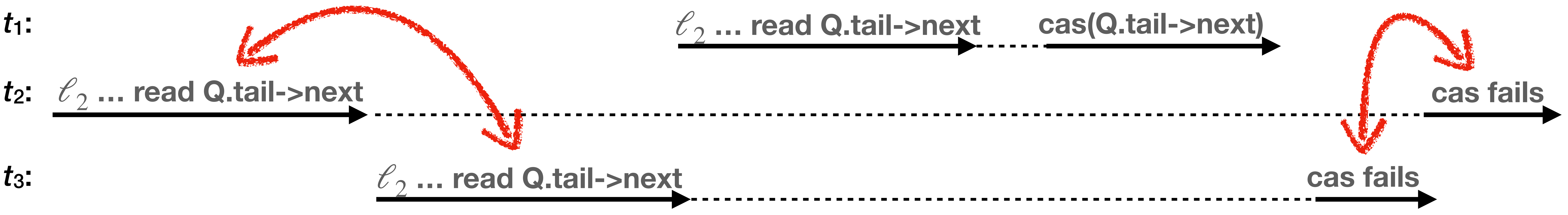


Quotient Expression

$$(\ell_2 \cdots \ell_8)^N \cdot (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true}) \cdot (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$$

Canonical interpretation for 3 threads.

> *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.*



Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

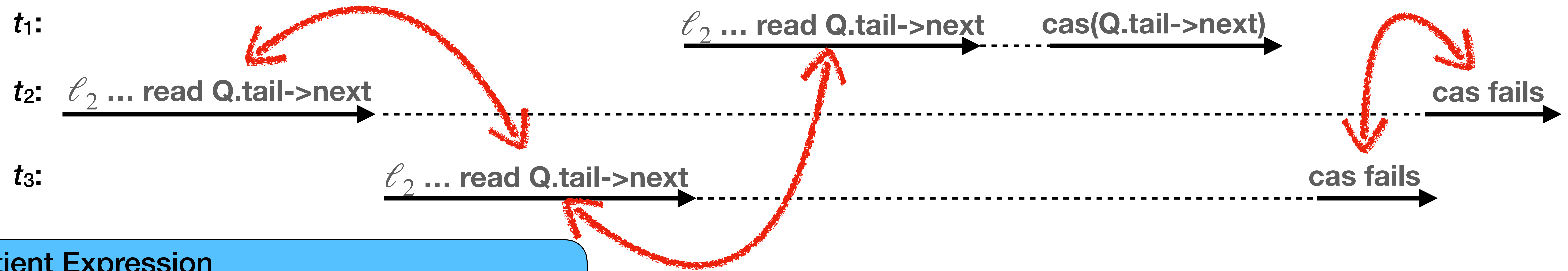
$\cdot\ (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\ (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Canonical interpretation for 3 threads.

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS | SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

> *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.*



Canonical interpretation for 3 threads.

Quotient Expression

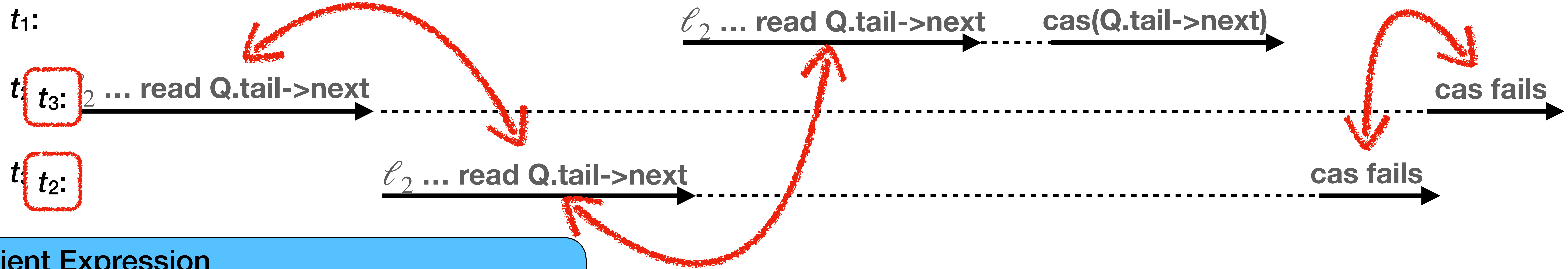
$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS | SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

> *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.*



Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

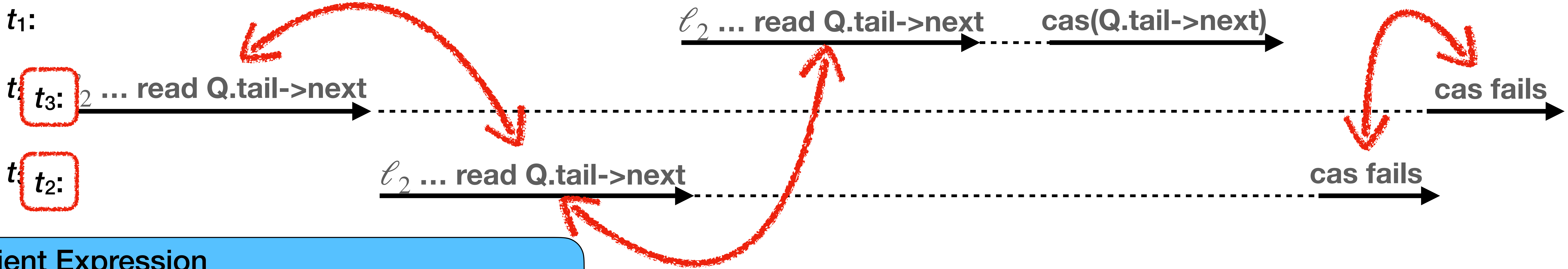
$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Canonical interpretation for 3 threads.

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS | SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

> *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.*



Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)/true})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Canonical interpretation for 3 threads.

Trace equivalence up to commutativity/relabel

$$\tau \equiv_{\bowtie} \tau'$$

Enqueue Succeed Layer

$(\ell_2 \cdots \ell_8)^N$

$\cdot\ (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)})$

$\cdot\ (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

Herlihy/Wing Queue

Enqueue Succeed Layer

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot$ cas(Q.tail->next)$)$

$\cdot\, (\ell_8 \cdot$ cas()/false $\hookleftarrow \ell_2)^N$

$+$

Dequeue Succeed Layer

$(\ell_2^D \cdots \ell_{10}^D)^N$

$\cdot\, (\ell_2^D \cdots \ell_{10}^D \cdot$ cas(Q.head)/t$)$

$\cdot\, (\ell_8^D \cdot$ cas()/false $\hookleftarrow \ell_2^D)^N$

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS

SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

Enqueue Succeed Layer

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)})$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N$

$+$

Dequeue Succeed Layer

$(\ell_2^D \cdots \ell_{10}^D)^N$

$\cdot\, (\ell_2^D \cdots \ell_{10}^D \cdot \text{cas(Q.head)/t})$

$\cdot\, (\ell_8^D \cdot \text{cas()/false} \hookleftarrow \ell_2^D)^N$

$+$

Advancer Succeed Layer

$(\ell_2^A \cdots \ell_6^A)^N$

$\cdot\, (\ell_2^A \cdots \ell_6^A \cdot \text{cas(Q.tail)/t}$

$\cdot\, (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2^A)^N$

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS

SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

$$\Bigg(\underbrace{(\ell_2 \cdots \ell_8)^N \cdot (\ell_2 \cdots \ell_8 \cdot \text{cas(Q.tail->next)}) \cdot (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2)^N}_{\textit{Enqueue Succeed Layer}} + \underbrace{(\ell_2^D \cdots \ell_{10}^D)^N \cdot (\ell_2^D \cdots \ell_{10}^D \cdot \text{cas(Q.head)/t}) \cdot (\ell_8^D \cdot \text{cas()/false} \hookleftarrow \ell_2^D)^N}_{\textit{Dequeue Succeed Layer}} + \underbrace{(\ell_2^A \cdots \ell_6^A)^N \cdot (\ell_2^A \cdots \ell_6^A \cdot \text{cas(Q.tail)/t} \cdot (\ell_8 \cdot \text{cas()/false} \hookleftarrow \ell_2^A)^N}_{\textit{Advancer Succeed Layer}} \Bigg)^*$$


Idea

Idea

- Concurrent object proof methodology based on representative interleavings.

Idea

- Concurrent object proof methodology based on representative interleavings.
- Formal version of concurrent object authors' "scenarios."

- Concurrent object proof methodology based on representative interleavings.
- Formal version of concurrent object authors' "scenarios."
- **Technique:** For an object, find a core set of such representatives—(a "quotient") described by a quotient expression (as seen on previous slide).

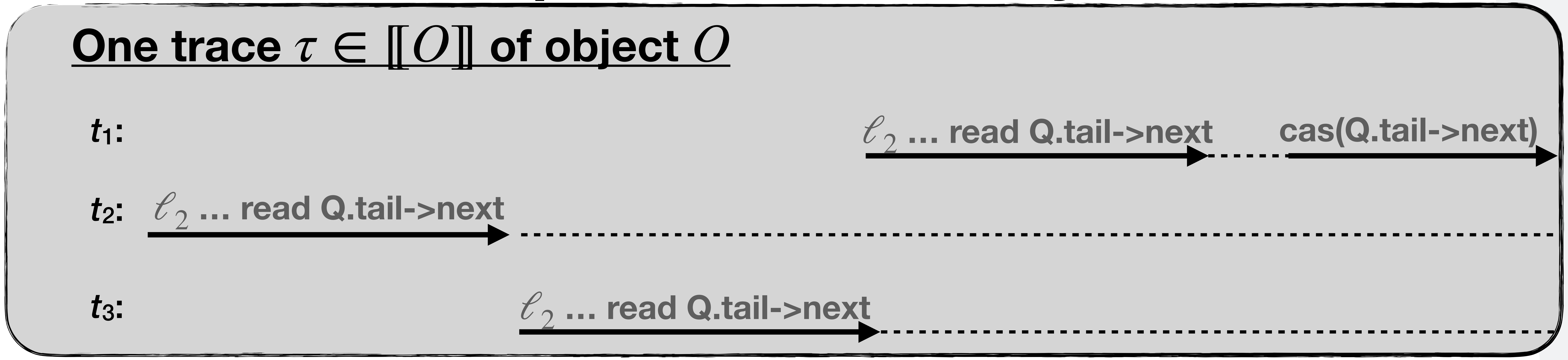
- Concurrent object proof methodology based on representative interleavings.
- Formal version of concurrent object authors' "scenarios."
- **Technique:** For an object, find a core set of such representatives—(a "quotient") described by a quotient expression (as seen on previous slide).
- Each representative interleaving equivalent to infinitely many others.

- Concurrent object proof methodology based on representative interleavings.
- Formal version of concurrent object authors' "scenarios."
- **Technique:** For an object, find a core set of such representatives—(a "quotient") described by a quotient expression (as seen on previous slide).
- Each representative interleaving equivalent to infinitely many others.
- **Benefit:** Easier to work with quotient, e.g., linearizability.

Defining an Object's Quotient

- **Trace equivalence relation up to commutativity**

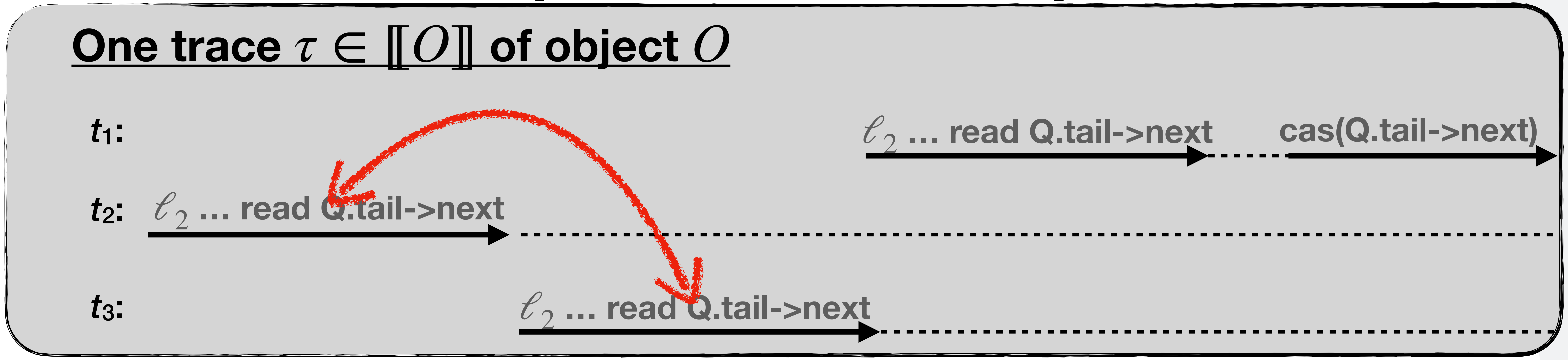
Single-swap:
$(\tau \equiv_1 \tau')$



Defining an Object's Quotient

- **Trace equivalence relation up to commutativity**

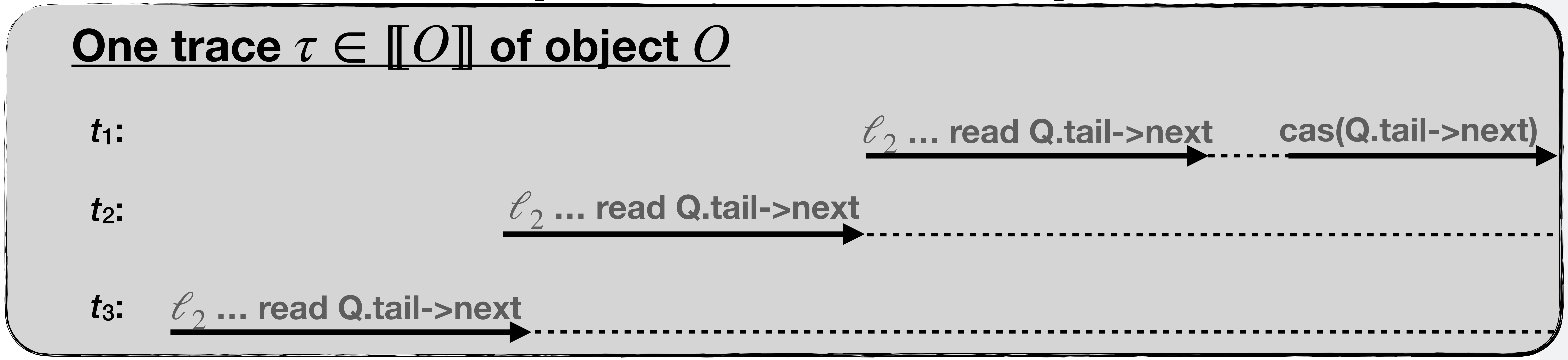
Single-swap:
$(\tau \equiv_1 \tau')$



Defining an Object's Quotient

- **Trace equivalence relation up to commutativity**

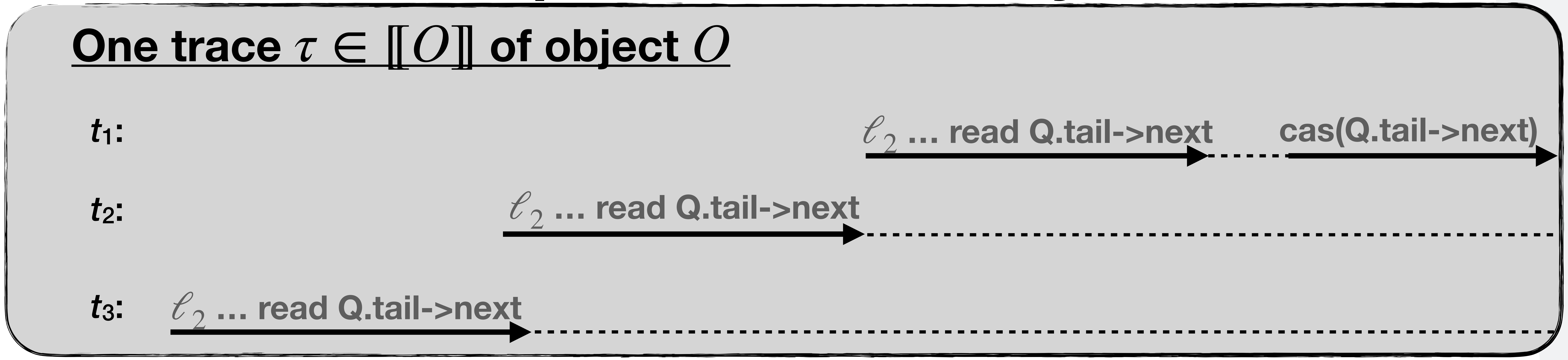
Single-swap:
$(\tau \equiv_1 \tau')$



Defining an Object's Quotient

- **Trace equivalence relation up to commutativity**

Single-swap:
$(\tau \equiv_1 \tau')$



Definition: Trace ***equivalence up to commutativity*** denoted $\tau \equiv_{\bowtie} \tau'$ is the least reflexive-transitive relation that includes all such "single-swaps" $\tau \equiv_1 \tau'$.

Defining an Object's Quotient

Defining an Object's Quotient

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

Defining an Object's Quotient

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

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

Defining an Object's Quotient

<u>Definition</u>: ***Commutativity quotient*** of a concurrent object is a (sub)set of the object's traces $\llparenthesis O \rrparenthesis \subset [\![O]\!]$ such that:

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

Defining an Object's Quotient

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

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

Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot$ cas(Q.tail->next)/true$)$

$\cdot\, (\ell_8 \cdot$ cas()/false $\hookleftarrow \ell_2)^N$

Defining an Object's Quotient

Definition: ***Commutativity quotient*** of a concurrent object is a (sub)set of the object's traces $\llparenthesis O \rrparenthesis \subset [\![O]\!]$ such that:

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

Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot$ cas(Q.tail->next)/true)

$\cdot\, (\ell_8 \cdot$ cas()/false $\hookleftarrow \ell_2)^N$

Context-free grammar

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

Defining an Object's Quotient

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

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

Quotient Expression

$(\ell_2 \cdots \ell_8)^N$

$\cdot\, (\ell_2 \cdots \ell_8 \cdot$ cas(Q.tail->next)/true)

$\cdot\, (\ell_8 \cdot$ cas()/false $\hookleftarrow \ell_2)^N$

Context-free grammar

Interpretation: one canonical trace for every n

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

Challenges & Contributions

- Quotients, semantically.
- Quotient expressions.
- Automata.
- Verifying concurrent objects.
- Automated generation.

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

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


q_1
Q.tail=Q.head
∧ Q.tail->next=null
Enqueue
Succeed Layer
q_3
Q.tail=Q.head
∧ Q.tail->next≠null
q_2
Q.tail≠Q.head
∧ Q.tail->next=null
Enqueue
Succeed Layer
q_4
Q.tail≠Q.head
∧ Q.tail->next≠null

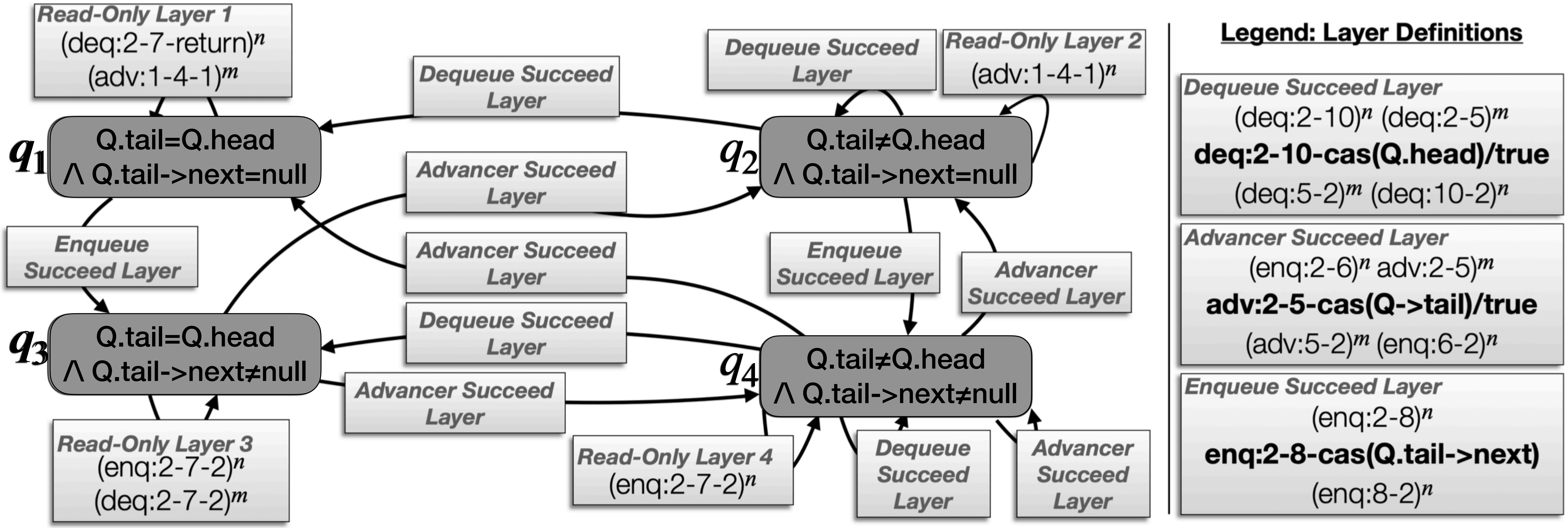
Enqueue Succeed Layer

$(\text{enq:2-8})^n$

enq:2-8-cas(Q.tail->next)

$(\text{enq:8-2})^n$

Quotient Automaton



Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

Herlihy/Wing Queue

Summary: Michael-Scott Queue

Summary: Michael-Scott Queue

- Many CAS operations

Summary: Michael-Scott Queue

- Many CAS operations
- Linearization points through helping (advancing the tail)

Summary: Michael-Scott Queue

- Many CAS operations
- Linearization points through helping (advancing the tail)
- Quotient:

Summary: Michael-Scott Queue

- Many CAS operations
- Linearization points through helping (advancing the tail)
- Quotient:
  - Proof organized like authors' arguments

Summary: Michael-Scott Queue

- Many CAS operations
- Linearization points through helping (advancing the tail)
- Quotient:
  - Proof organized like authors' arguments
  - Automaton shows when layers are enabled.

Summary: Michael-Scott Queue

- Many CAS operations
- Linearization points through helping (advancing the tail)
- Quotient:
  - Proof organized like authors' arguments
  - Automaton shows when layers are enabled.
  - Linearization points are explicit in the quotient: one per transition.

Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

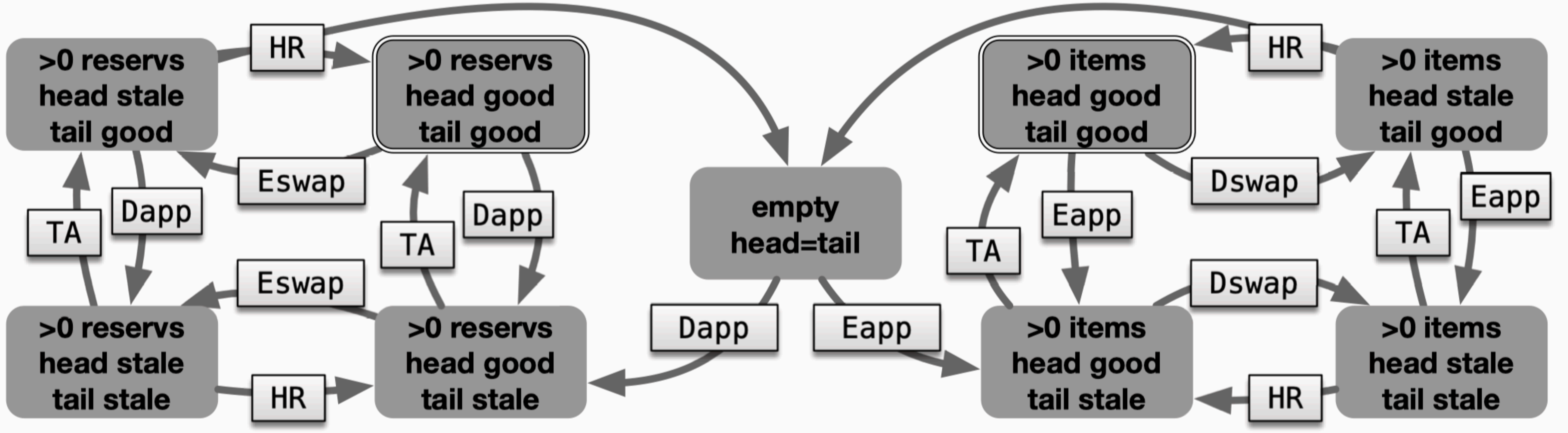
Herlihy/Wing Queue

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS

SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

When the queue is a list of **reservations**
(deq appends resv at tail, enq removes resv at head)

When the queue is a list of **items**
(enq appends items at tail, deq removes items at head)



Tail advance (TA)
DE:cas_1/t with (3 fail paths)*
DE:$cas_{3'}$/t with (3 fail paths)*

Enq append item node (Eapp)
E:cas_3/t with (1 fail path)*

Head reap (HR)
DE:$cas_{3''}$/t with (9 fail paths)*
DE:cas_6/t with (9 fail paths)*
DE:cas_7/t with (9 fail paths)*

Deq append reservation (Dapp)
D:cas_3/t with (1 fail path)*

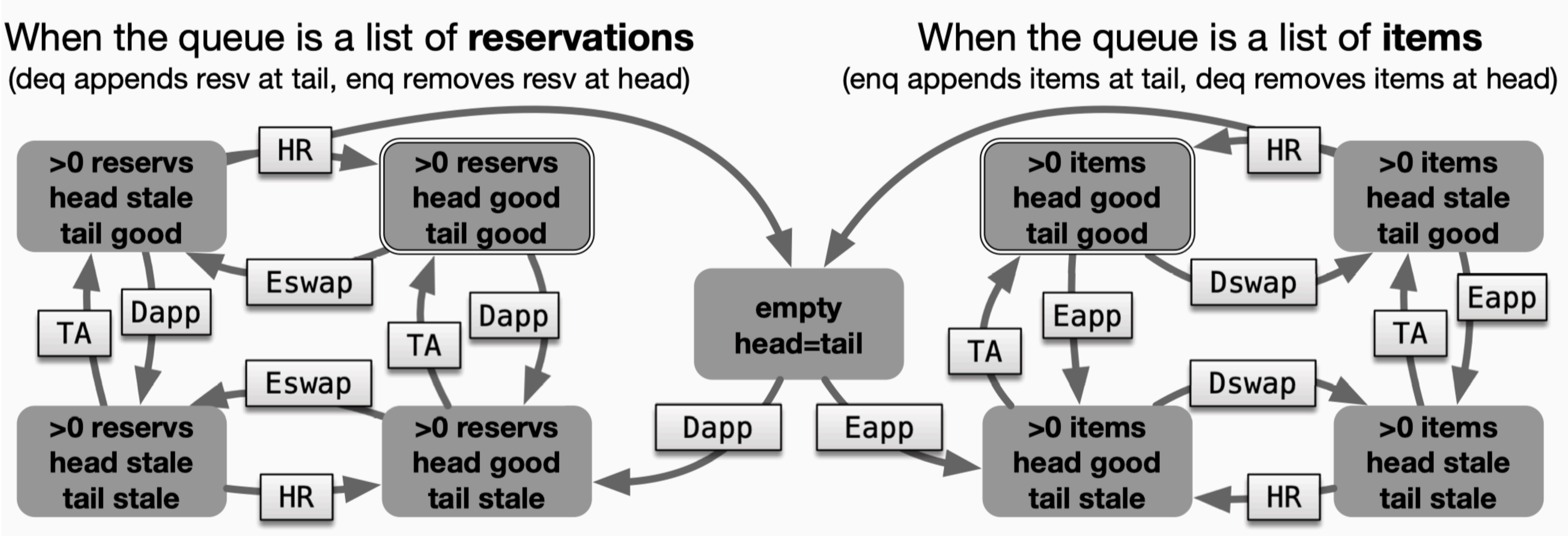
Enq swap res for item (Eswap)
DE:cas_5/t with (2 fail paths)*

Deq swap item for null (Dswap)
DE:cas_5/t with (2 fail paths)*

Michael/Scott Queue | Treiber's Stack | Harris *et al.* RDCSS

SLS Queue | Hendler *et al.* Elim. Stack | Herlihy/Wing Queue

[Regarding enqueue,] the reservation linearization point for this code path occurs at line [...] when we successfully insert our offering into the queue – Scherer III et al. [2006]



DE:cas_1/t with (3 fail paths)*

DE:$cas_{3'}$/t with (3 fail paths)*

Enq append item node (Eapp)

E:cas_3/t with (1 fail path)*

Head reap (HR)

DE:$cas_{3''}$/t with (9 fail paths)*

DE:cas_6/t with (9 fail paths)*

DE:cas_7/t with (9 fail paths)*

Deq append reservation (Dapp)

D:cas_3/t with (1 fail path)*

Enq swap res for item (Eswap)

DE:cas_5/t with (2 fail paths)*

Deq swap item for null (Dswap)

DE:cas_5/t with (2 fail paths)*

Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

Herlihy/Wing Queue

Summary: SLS Queue

Summary: SLS Queue

- Operations involve multiple CAS steps

Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping

Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping
- Operations are ***synchronous***.

Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping
- Operations are ***synchronous***.
- Quotient:

Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping
- Operations are ***synchronous***.
- Quotient:
  - Proof organized like authors' arguments

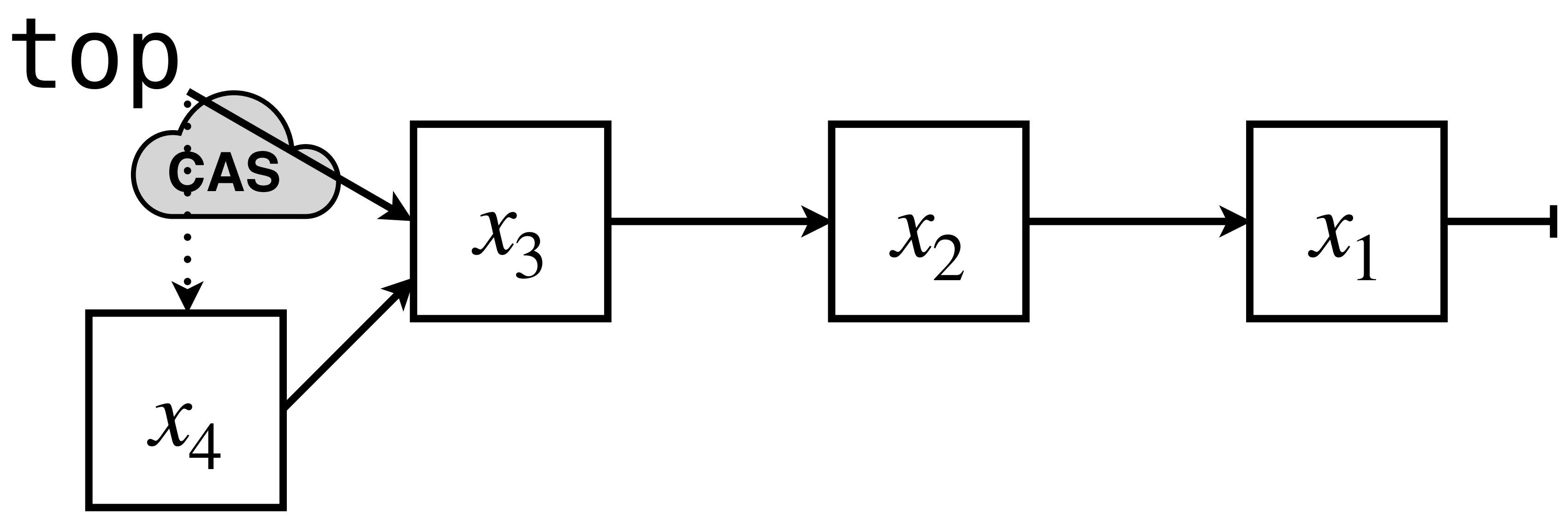
Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping
- Operations are ***synchronous***.
- Quotient:
  - Proof organized like authors' arguments
  - Automaton shows enabled layers.

Summary: SLS Queue

- Operations involve multiple CAS steps
- Linearization points through helping
- Operations are ***synchronous***.
- Quotient:
  - Proof organized like authors' arguments
  - Automaton shows enabled layers.
  - Linearization points are explicit.

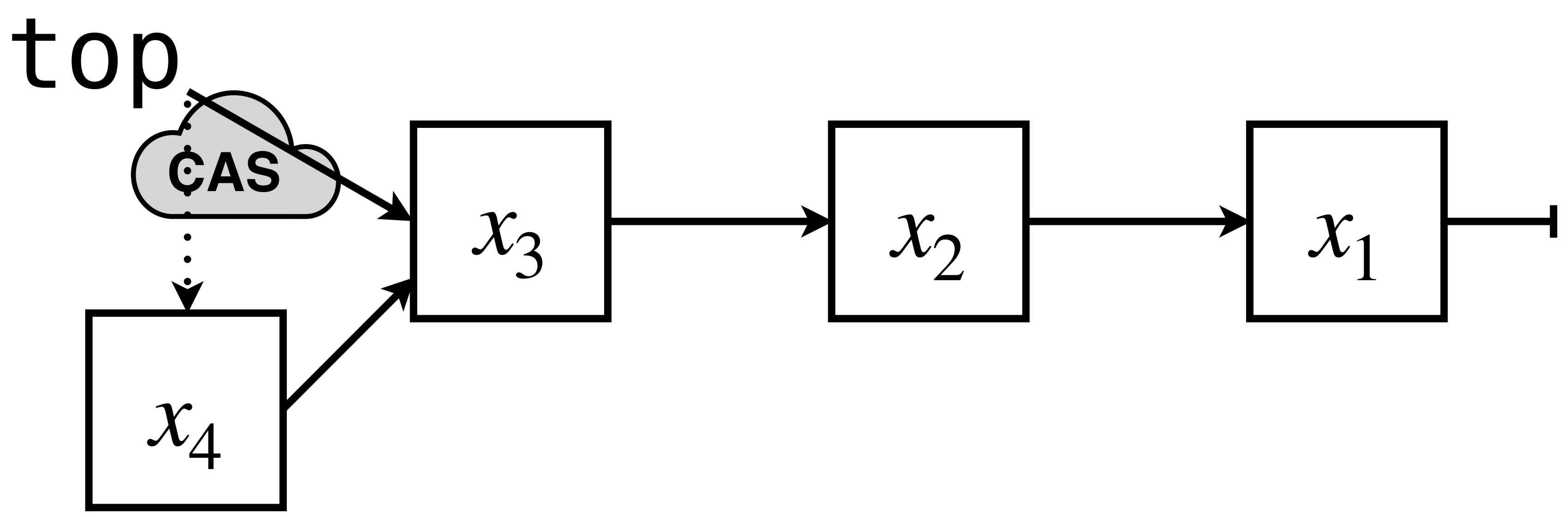
Treiber's Stack



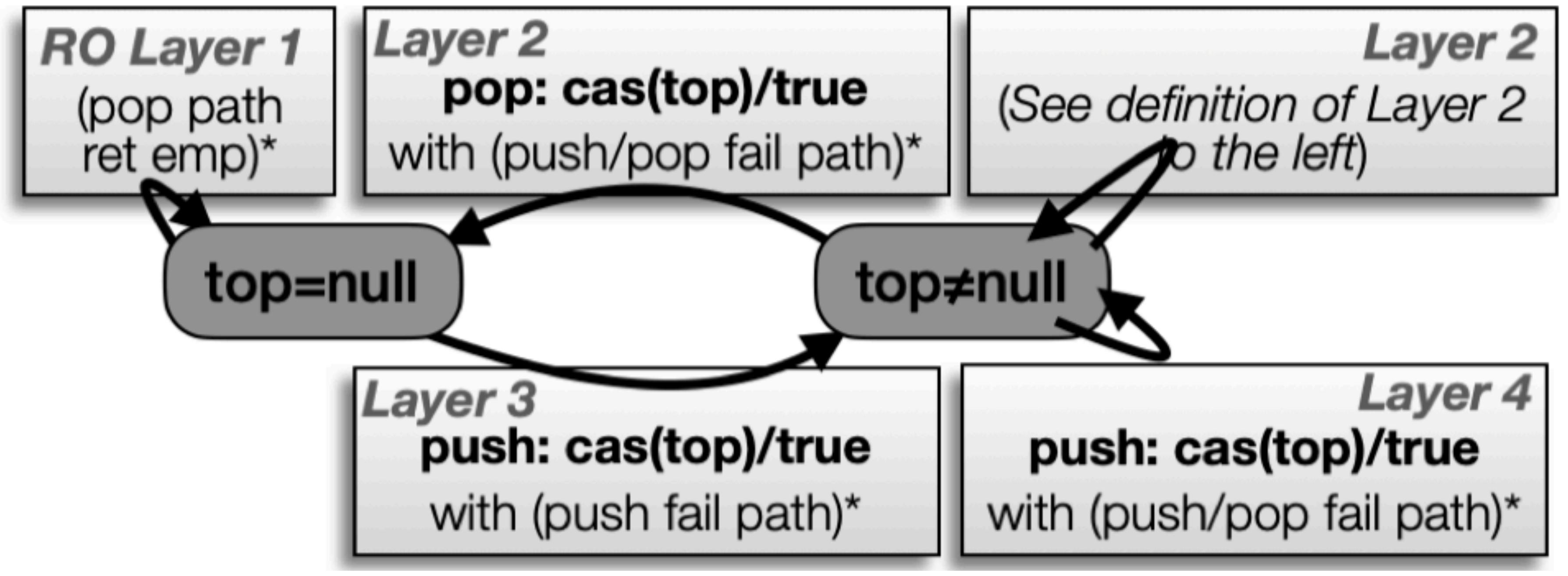
Michael/Scott Queue | **Treiber's Stack** | Harris *et al.* RDCSS

SLS Queue | **Hendler *et al.* Elim. Stack** | Herlihy/Wing Queue

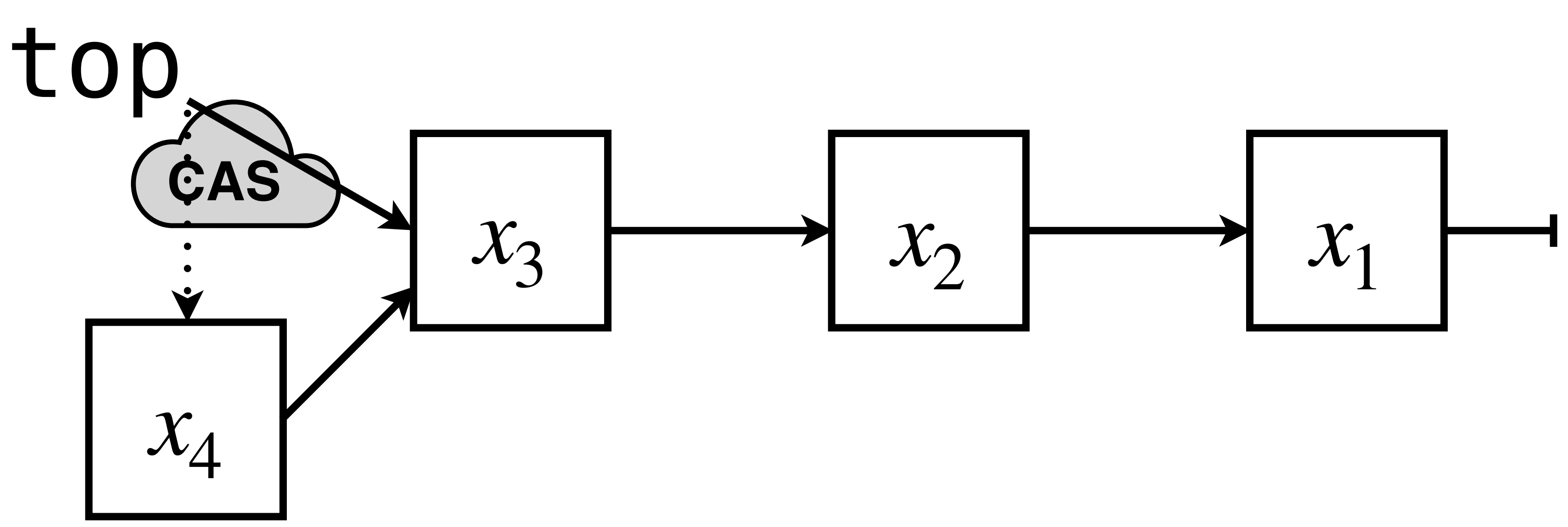
Treiber's Stack



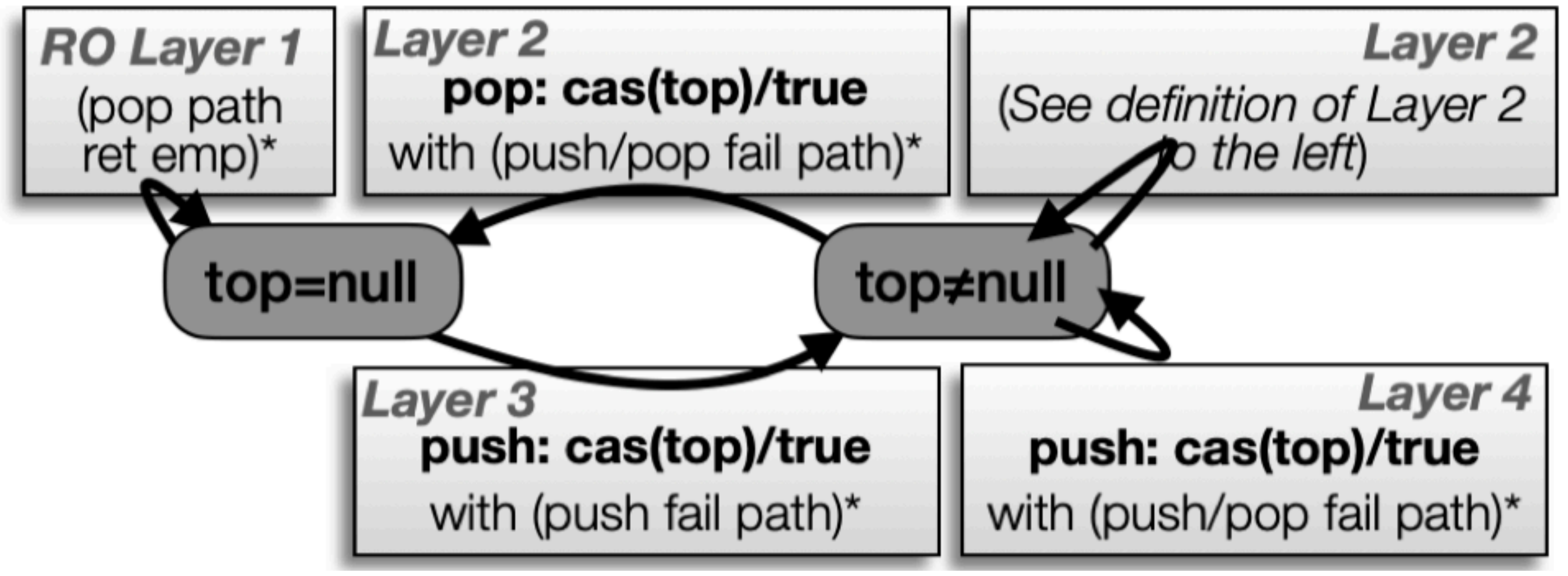
Quotient for Treiber's Stack



Treiber's Stack



Quotient for Treiber's Stack



Elimination Stack extension [Hendler et al. 2004]

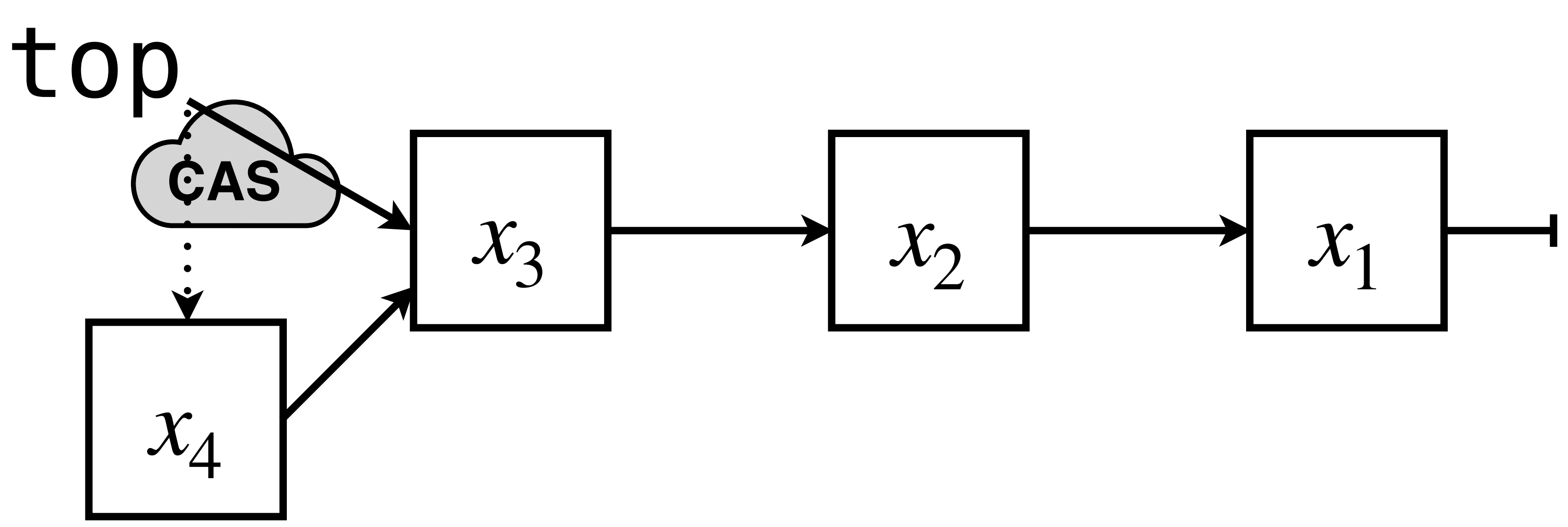
`location[tid]`

		(Push, tid2, 42)		(Pop, tid4, _)	

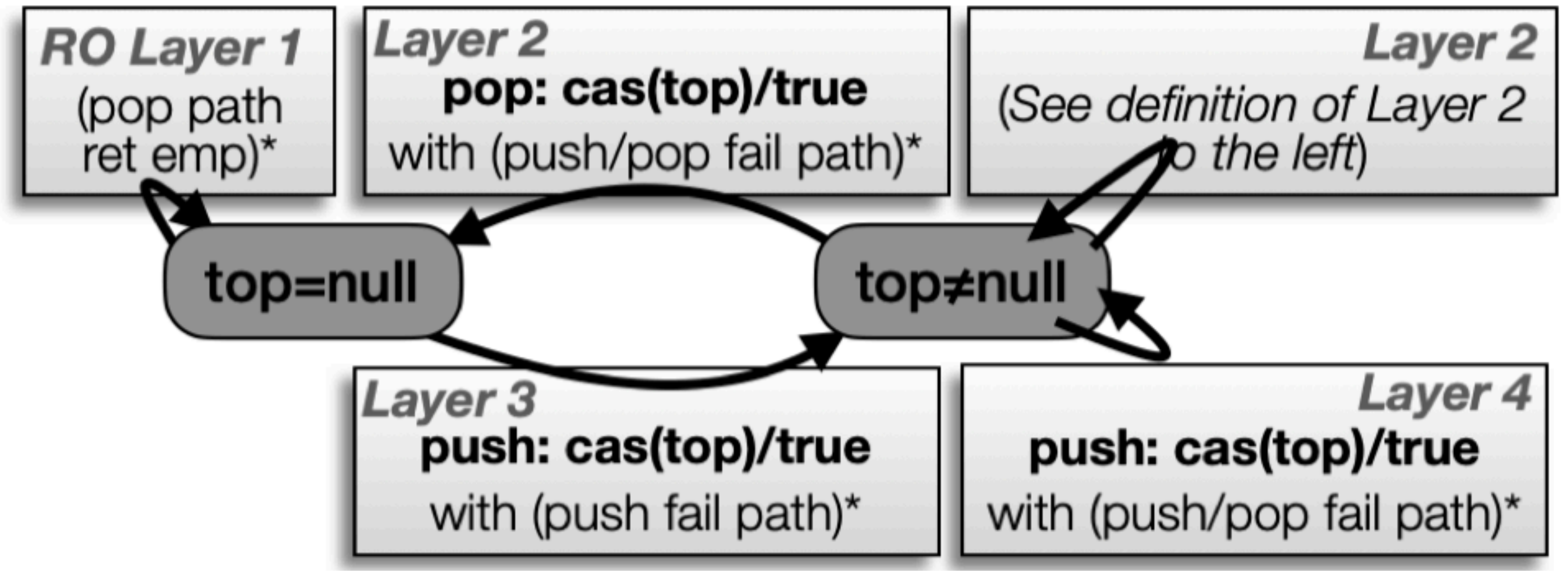
`collision[ ]`

	tid4				

Treiber's Stack



Quotient for Treiber's Stack



Elimination Stack extension [Hendler et al. 2004]

`location[tid]`

		(Push, tid2, 42)		(Pop, tid4, _)	

`collision[ ]`

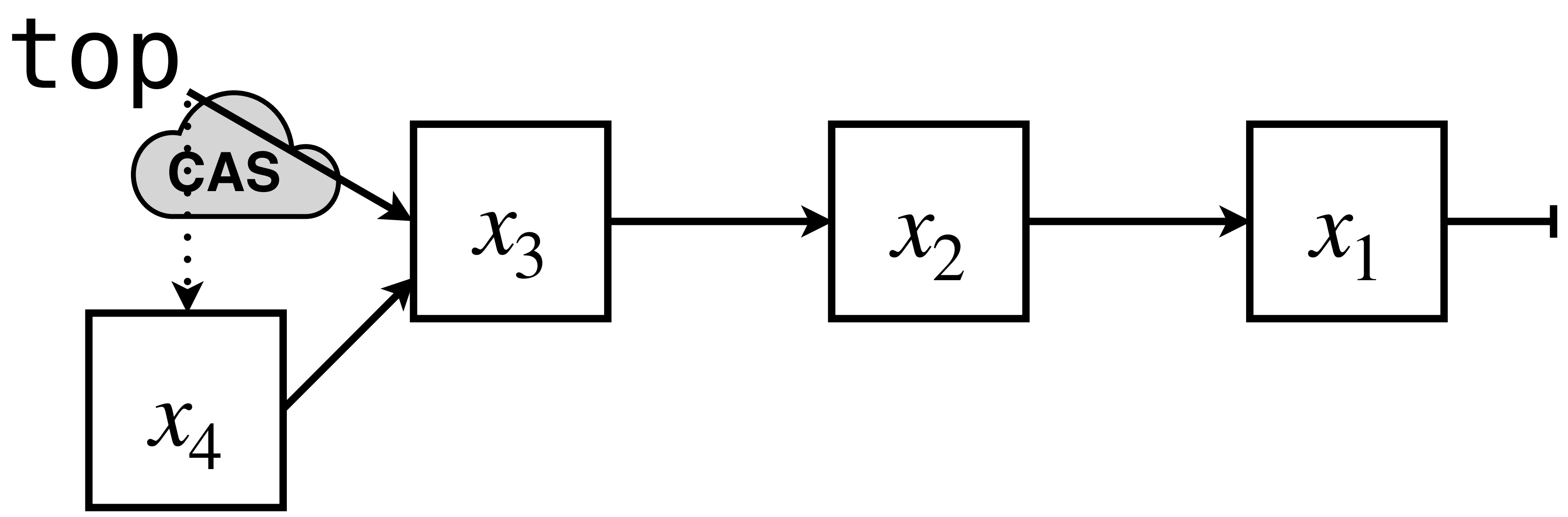
	tid4				

[A] colliding operation op is active if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is passive if op fails in the CAS of line S10 or S19. [underlines added] – Hendler et al. [2004]

Michael/Scott Queue | **Treiber's Stack** | Harris *et al.* RDCSS

SLS Queue | **Hendler *et al.* Elim. Stack** | Herlihy/Wing Queue

Treiber's Stack



Elimination Stack extension [Hendler et al. 2004]

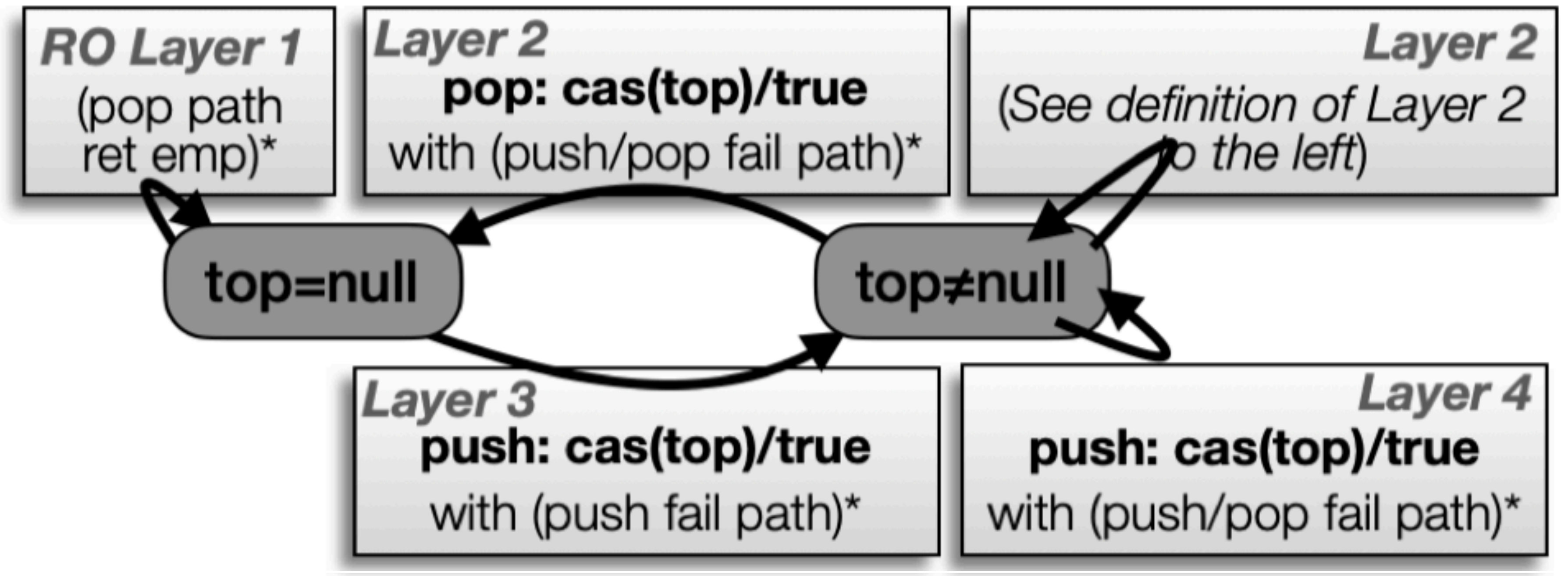
location[tid]

		(Push, tid2, 42)		(Pop, tid4, _)	

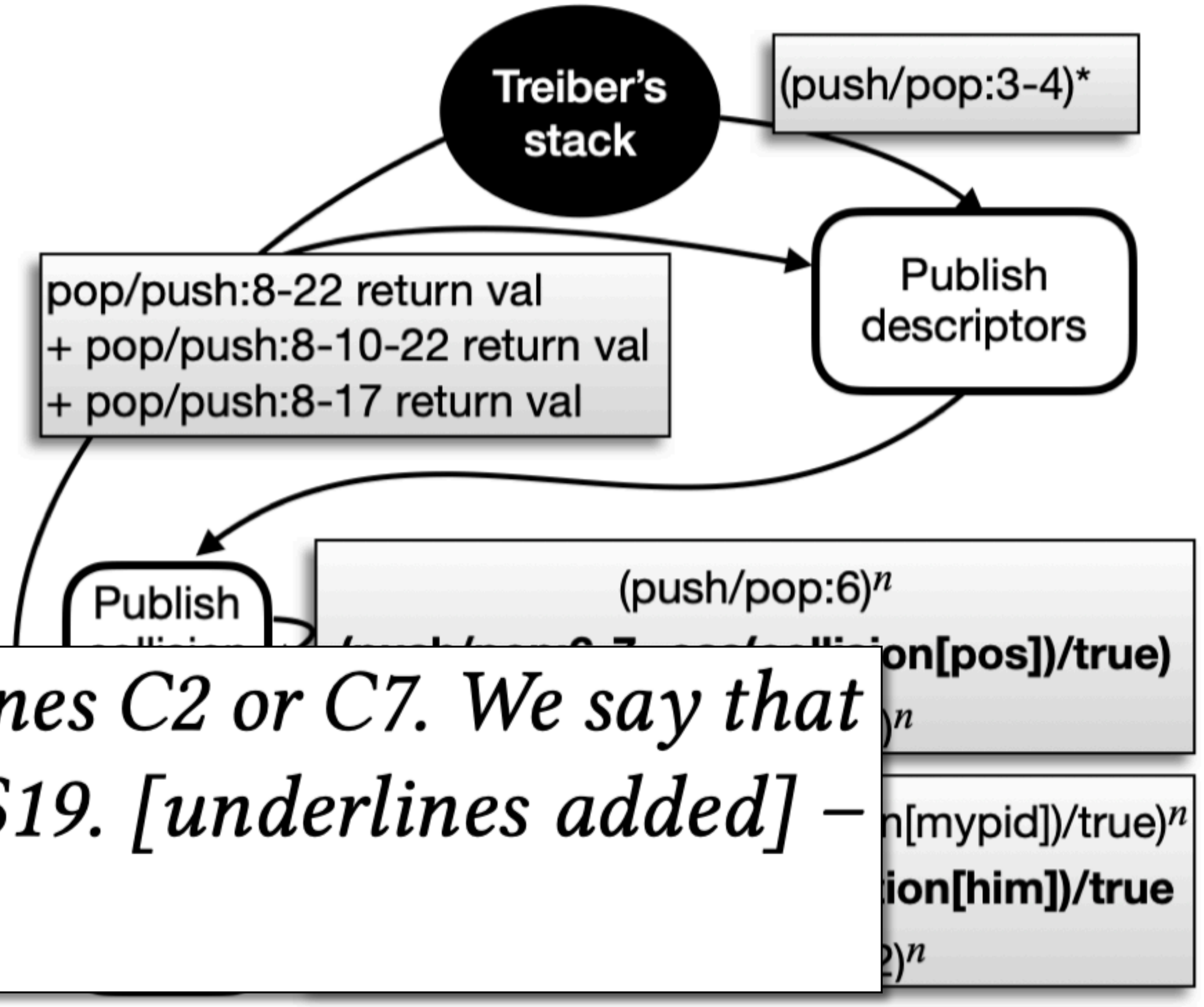
collision[ ]

	tid4				

Quotient for Treiber's Stack



Quotient for the Elimination Stack



> *[A] colliding operation op is* <u>*active*</u> *if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is* <u>*passive*</u> *if op fails in the CAS of line S10 or S19. [underlines added]* – Hendler et al. [2004]

Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

Herlihy/Wing Queue

Summary: Stacks

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.
- Linearization points for two operations at one CAS operation (elimination)

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.
- Linearization points for two operations at one CAS operation (elimination)
- Quotient:

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.
- Linearization points for two operations at one CAS operation (elimination)
- Quotient:
    - Proof organized like authors' arguments

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.
- Linearization points for two operations at one CAS operation (elimination)
- Quotient:
  - Proof organized like authors' arguments
  - Linearization points explicit.

Summary: Stacks

- One ADT (Treiber) used as a ***submodule***.
- Linearization points for two operations at one CAS operation (elimination)
- Quotient:
  - Proof organized like authors' arguments
  - Linearization points explicit.
  - Captures "active" versus "passive" concepts (in the automaton layers).

- **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: $(deqF^{*} \cdot (enqI)^{+} \cdot enqW^{*} \cdot deqT^{*})^{*}$

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

Some enqueuers increments back

- **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

- **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

- **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[...] slot in a doubly-nested loop.
- Quotient expression: $(deqF^* \cdot (enqI)^+ \cdot enqW^* \cdot deqT^*)^*$

dequeue scans that need to restart

Some enqueuers increments back

(Maybe) some enq's writes a slot

dequeue scans that succeed

> *Enq execution occurs in two steps, which may be interleaved with steps of other concurrent operations: an array slot is reserved by atomically incrementing back, and the new item is stored in items.* – Sec 4.1 of Herlihy and Wing [1990]

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

dequeue scans that need to restart

Some enqueuers increments back

(Maybe) some enq's writes a slot

dequeue scans that succeed

Michael/Scott Queue

Treiber's Stack

Harris *et al.* RDCSS

SLS Queue

Hendler *et al.* Elim. Stack

Herlihy/Wing Queue

Summary: Herlihy-Wing Queue

Summary: Herlihy-Wing Queue

- Future-dependent linearization points

Summary: Herlihy-Wing Queue

- Future-dependent linearization points
- Linearization points cannot be associated with fixed statements.

Summary: Herlihy-Wing Queue

- Future-dependent linearization points
- Linearization points cannot be associated with fixed statements.
- Quotient:

Summary: Herlihy-Wing Queue

- Future-dependent linearization points
- Linearization points cannot be associated with fixed statements.
- Quotient:
  - Proof organized like authors' arguments

Summary: Herlihy-Wing Queue

- Future-dependent linearization points
- Linearization points cannot be associated with fixed statements.
- Quotient:
  - Proof organized like authors' arguments
  - Quotient expressed through regular expressions.

Summary: Herlihy-Wing Queue

- Future-dependent linearization points
- Linearization points cannot be associated with fixed statements.
- Quotient:
    - Proof organized like authors' arguments
    - Quotient expressed through regular expressions.
    - Linearization points ***become fixed*** in the quotient expression.

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 $q \xrightarrow{\lambda} q'$.

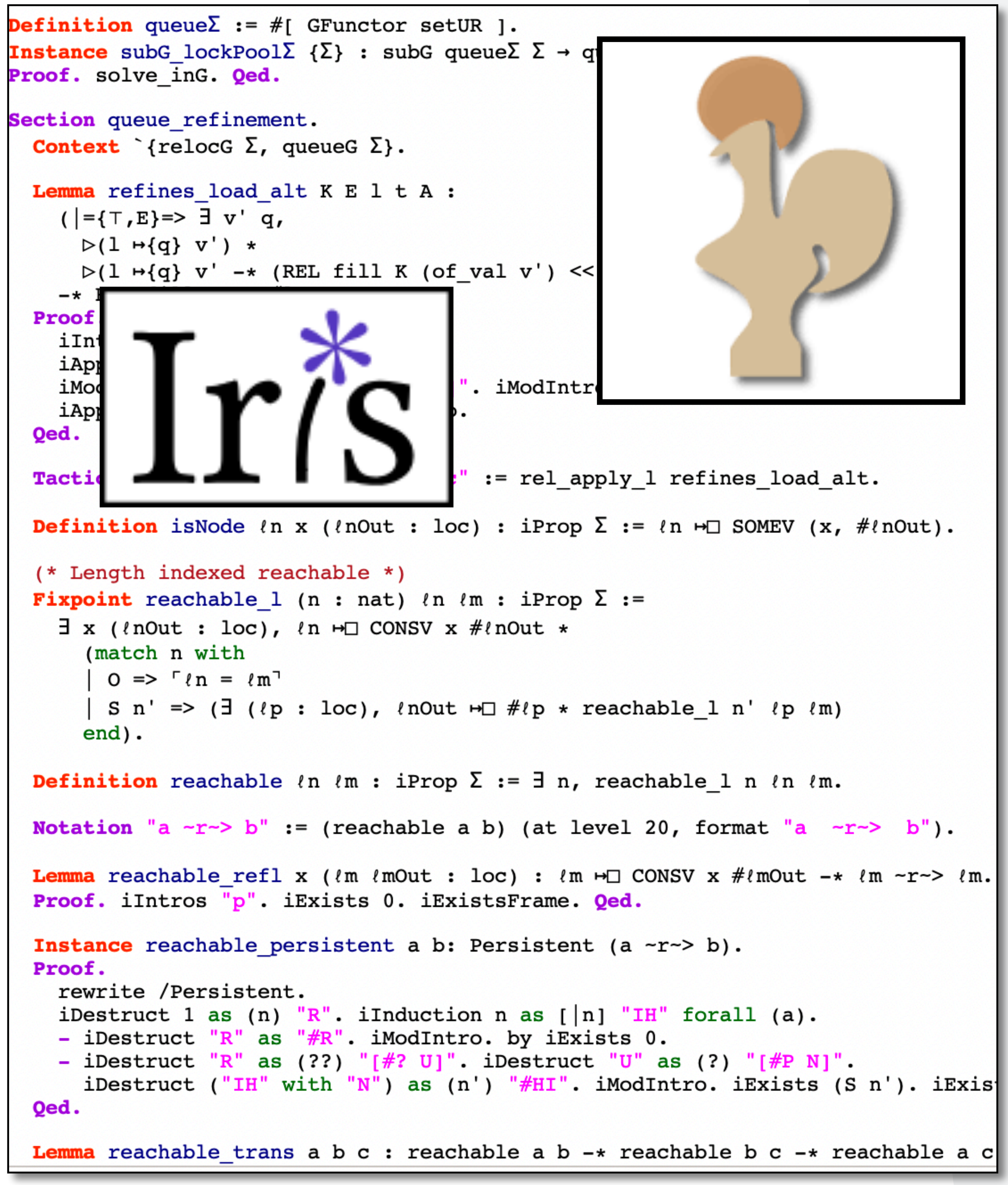
Generating Quotient Automata

- 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

Related Works on Linearizability & Reduction

- Owicki and Gries [1976]
- Rely/Guarantee [Jones 1983]
- Concurrent Separation Logic [Bronat et al. 2005; Brookes 2004; O'Hearn 2004; Parkinson et al. 2007]
- Views [Dinsdale-Young et al. 2013], TaDa [da Rocha Pinto et al. 2014]
- Numerous other works [Dragoi et al. 2013; Jung et al. 2018, 2020; Krishna et al. 2018; Ley-Wild and Nanevski 2013; Nanevski et al. 2019; Raad et al. 2015; Sergey et al. 2015; Turon et al. 2013; Vafeiadis 2008, 2009]
- Reductions [Lipton 1975, Elmas et al 2009], Civl [Hawblitzel et al. 2015; Kragl and Qadeer 2018; Kragel et al. 2018].
- Many others …



Civl

A verifier for concurrent programs

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…**

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

Thank you!