Proving Linearizability Using Partial Orders

Artem Khyzha[1] Mike Dodds[2]
Alexey Gotsman[1] Matthew Parkinson[3]

[1] - IMDEA Software Institute, Madrid
[2] - University of York, UK
[3] - MSR Cambridge, UK

Concurrent libraries

- Encapsulate efficient concurrent data structures:
  - Java: java.util.concurrent
  - C++: Intel Threading Building Blocks
  - C#: System.Collections.Concurrent
- Implement stacks, queues, skip lists, hash tables etc
- Hard to prove correct

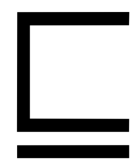
Concurrent data structure

- Proven to observationally refine the sequential spec.

- History = a well-formed sequence of operation invokations and responses

- $\forall$client, $H_C \in$ client(impl). $\exists H_A \in$ client(spec). $H_C \sim H_A$

data structure implementation



data structure specification

Naïve Queue

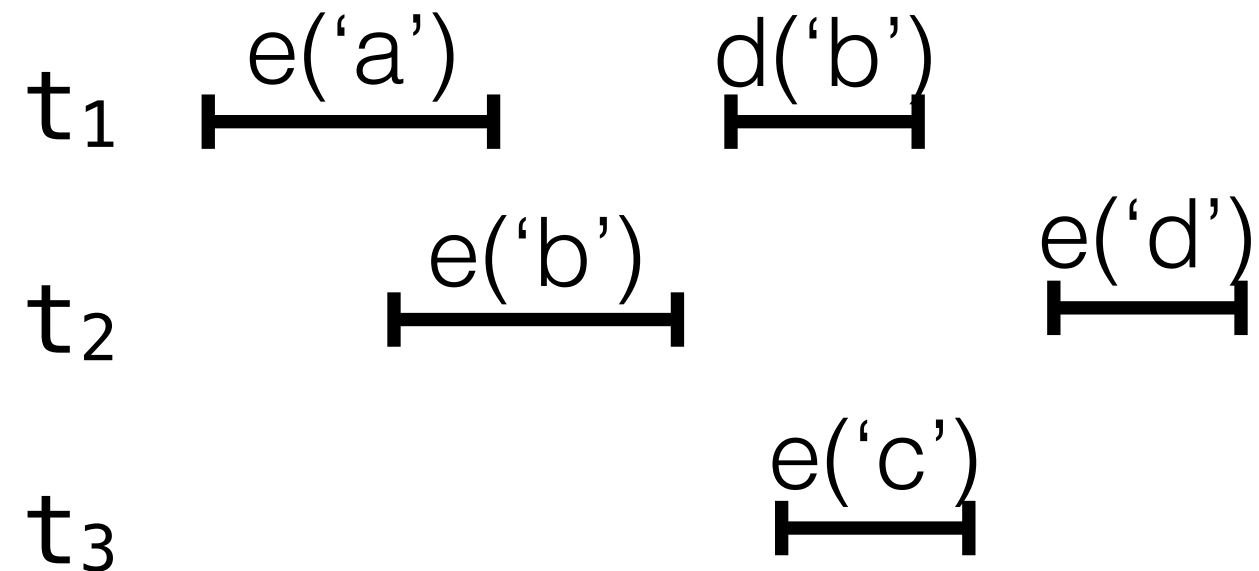
```
struct Node {
  Node *next; int val;
} *Tail;

void enqueue(Val v) {
  lock();
  try {
    Node e = new Node*;
    e.val = v;
    Tail.next = e;
    Tail = e;
  } finally {
    unlock();
  }
```

Atomic queue ADT

```
Sequence S;

void enqueue(int v) {
  atomic { S = S :: v; }
}
```

Linearizability

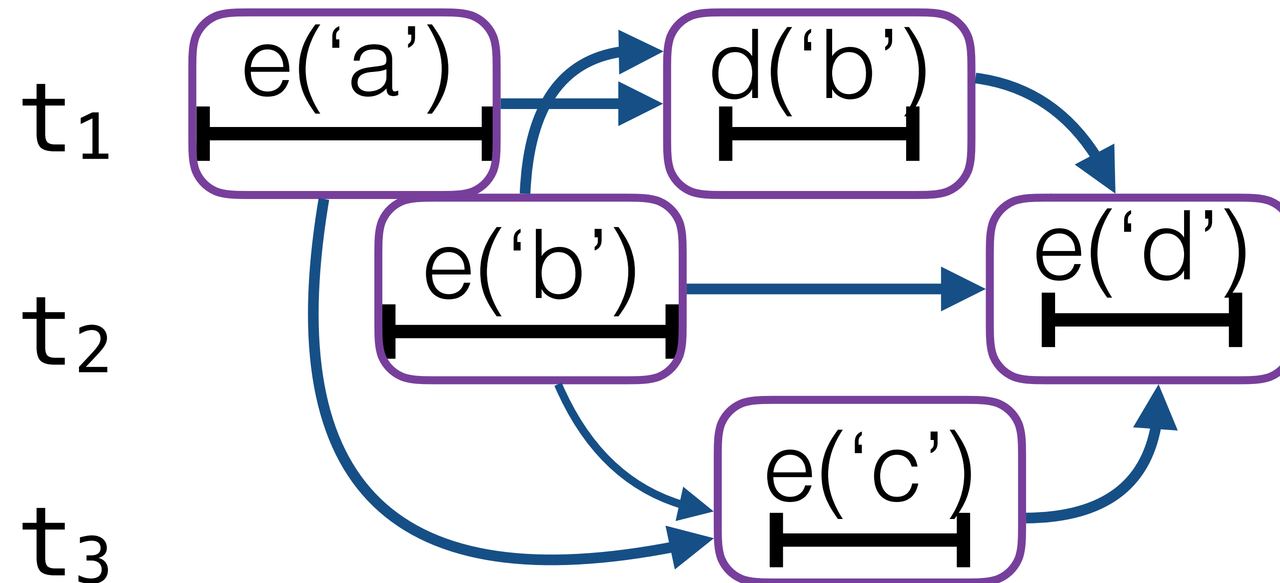
for every concrete history



find matching behaviour of the seq. spec.

Linearizability

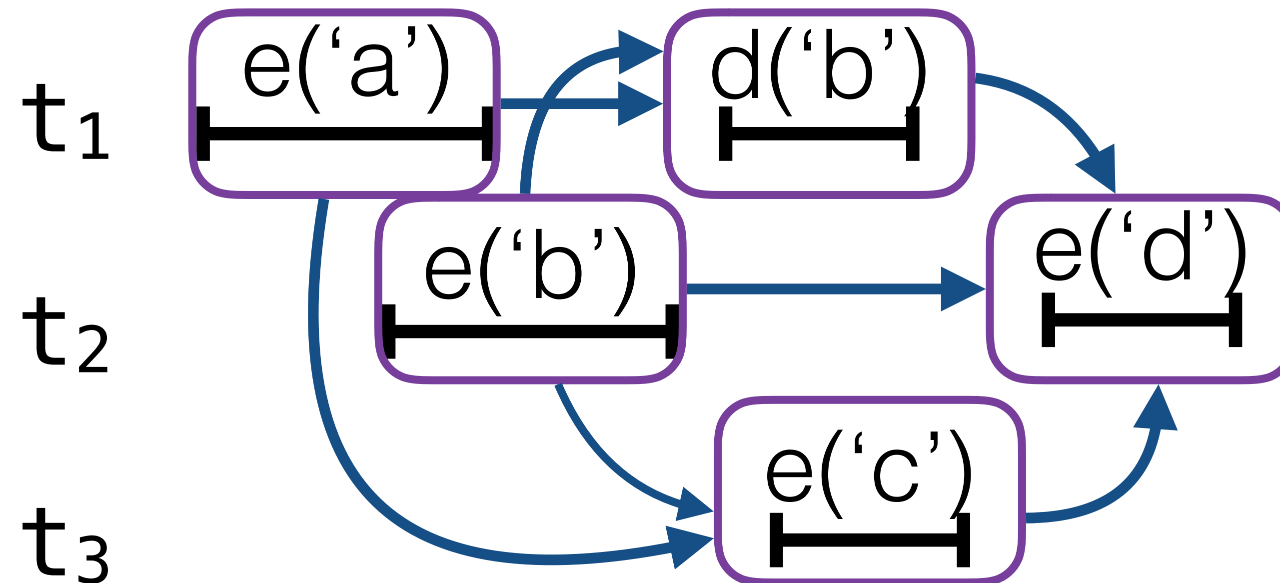
for every concrete history



find matching behaviour of the seq. spec.

Linearizability

for every concrete history

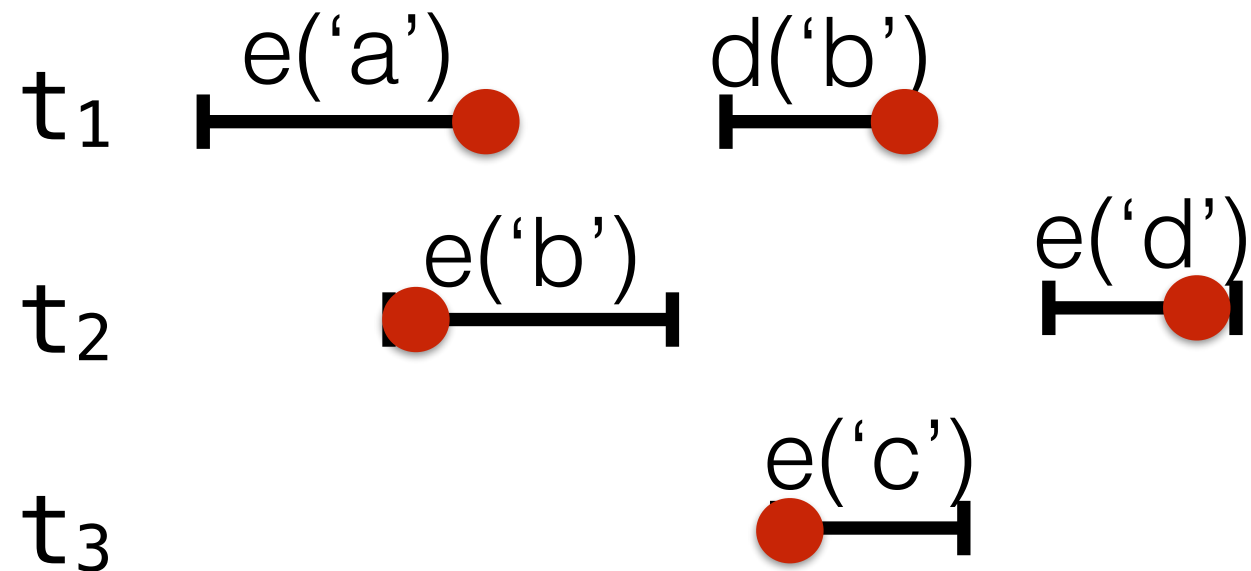


find a linearization

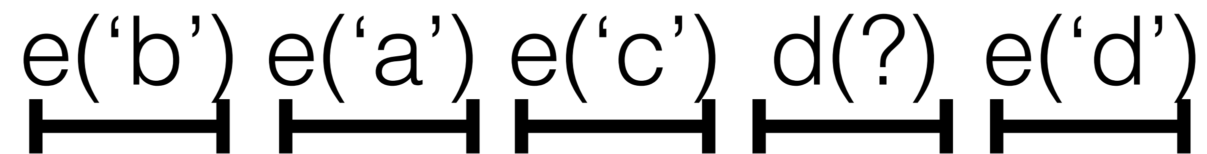
1. e('a') e('b') d('b') e('c') e('d')
2. e('a') e('b') e('c') d('b') e('d')
3. e('b') e('a') d('b') e('c') e('d')
4. e('b') e('a') e('c') d('b') e('d')

Linearization points

for every concrete history



find a linearization



Standard **proof method**: linearization points

Queue impl.

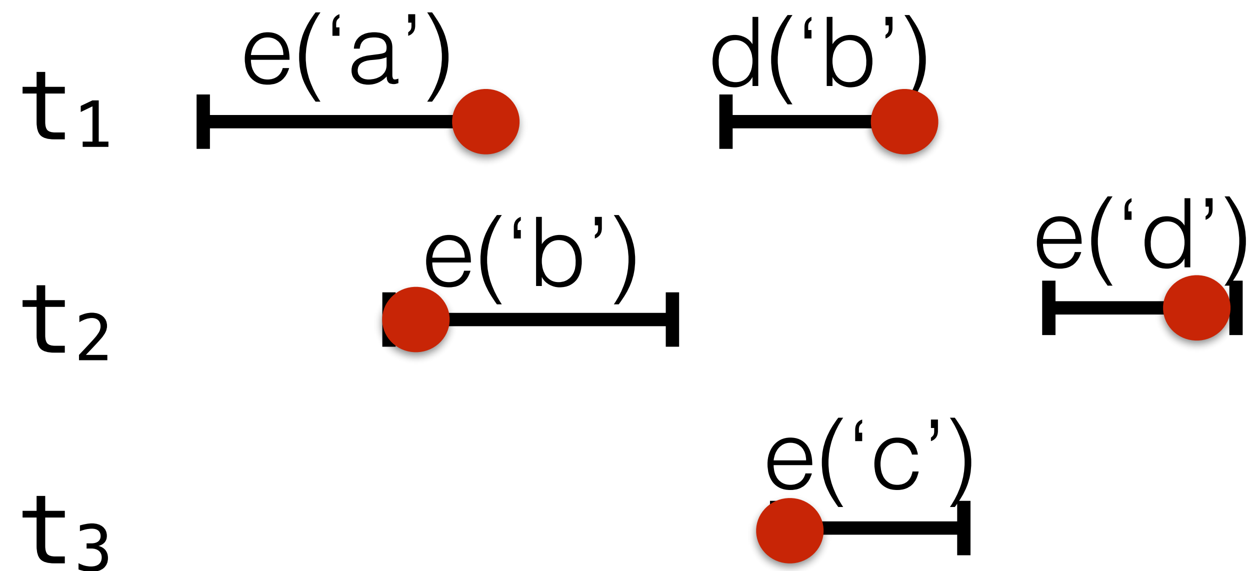
```
struct Node {
  Node *next; int val;
} *Top;

void enqueue(Val v) {
  lock();
  try {
    Node e = new Node*;
    e.val = v;
    tail.next = e;
    tail = e;
  } finally {
    unlock();
  }
```

Atomic queue ADT

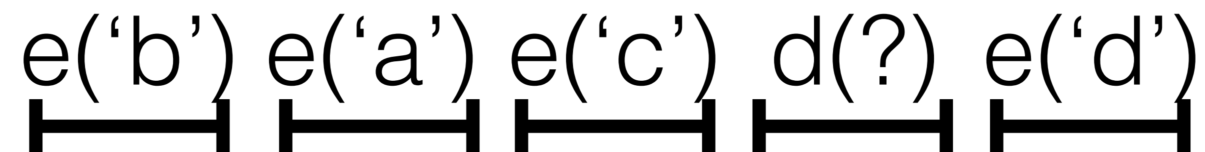
```
Sequence S;

void enqueue(int v) {
  atomic { S = S :: v; }
}
```

Linearization points

for every concrete history



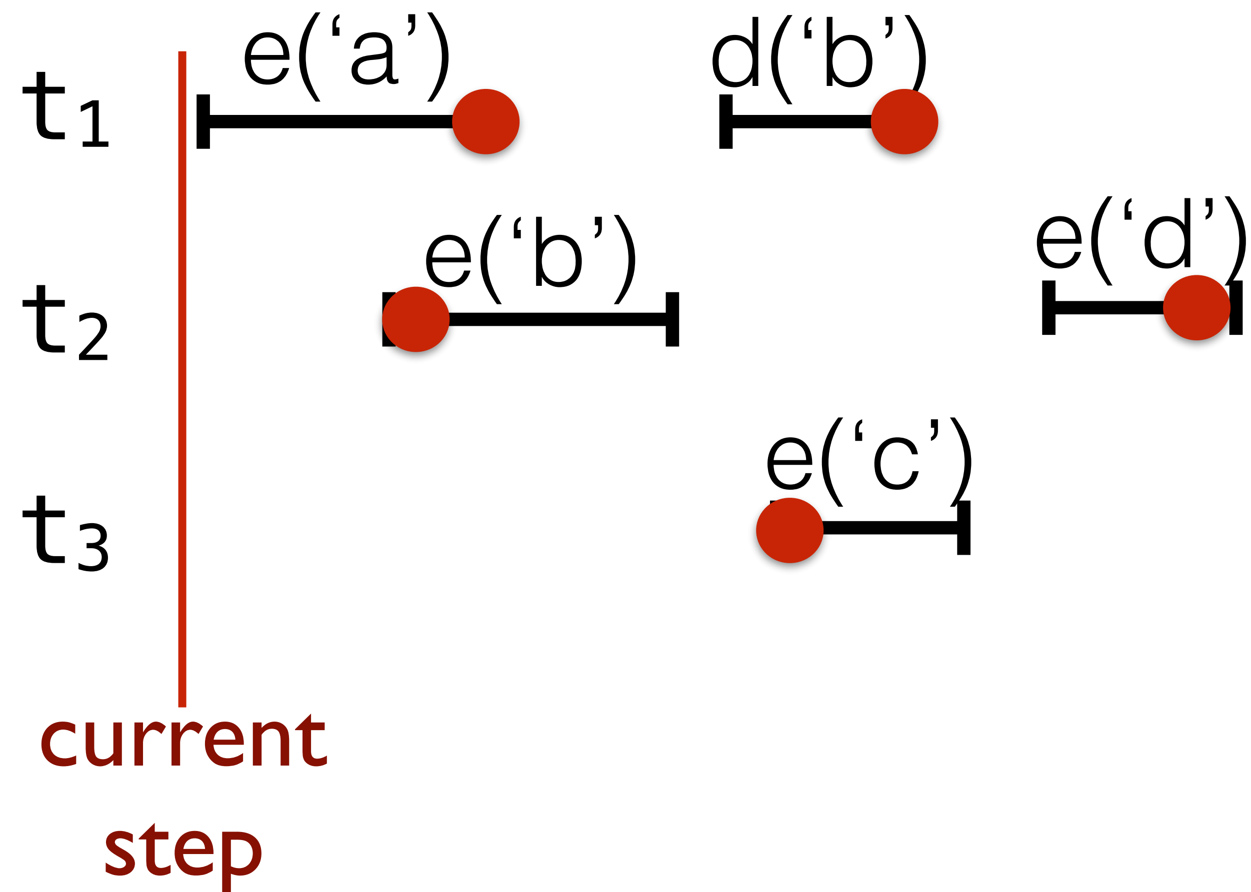
find a linearization



Standard **proof method**: linearization points

Forward simulation

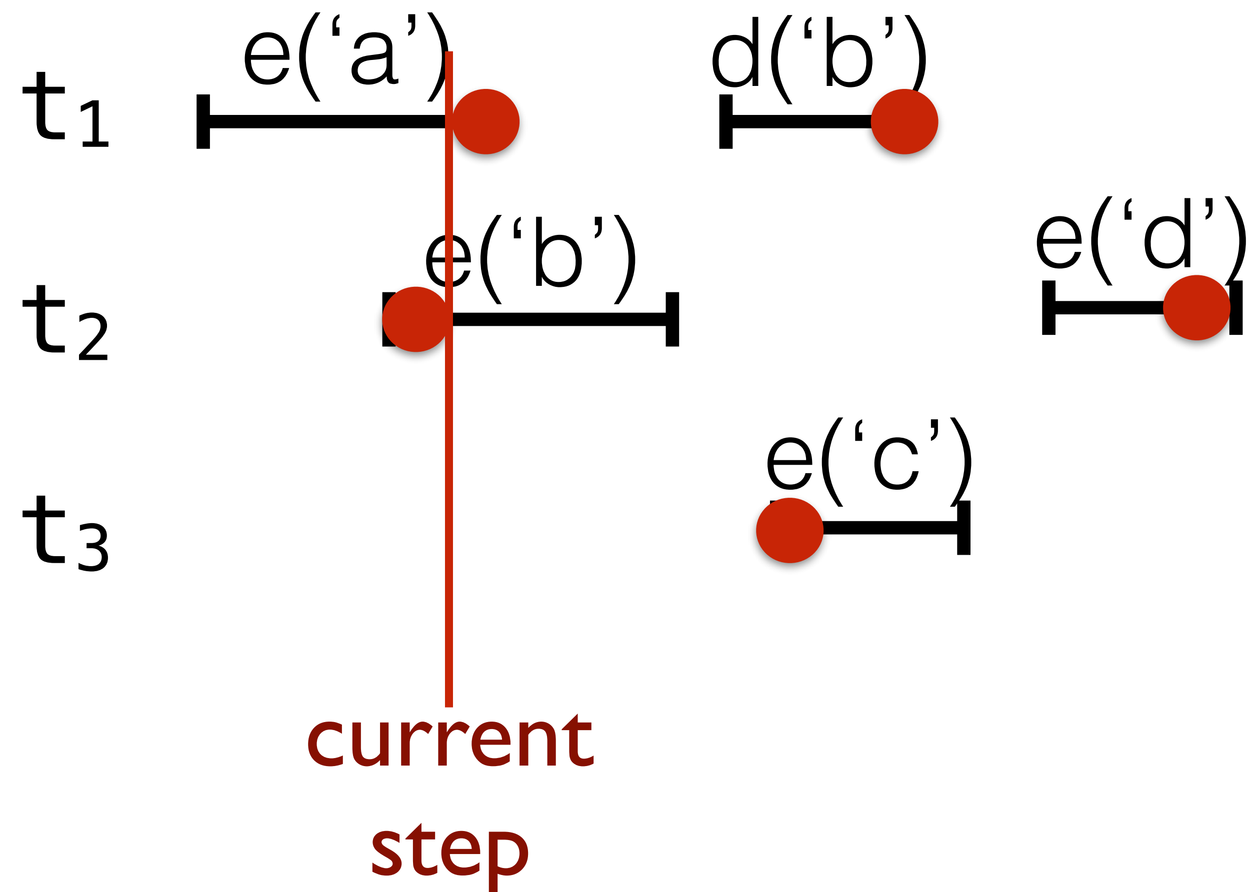
for every concrete history



build a linearization: `<empty sequence>`

Forward simulation

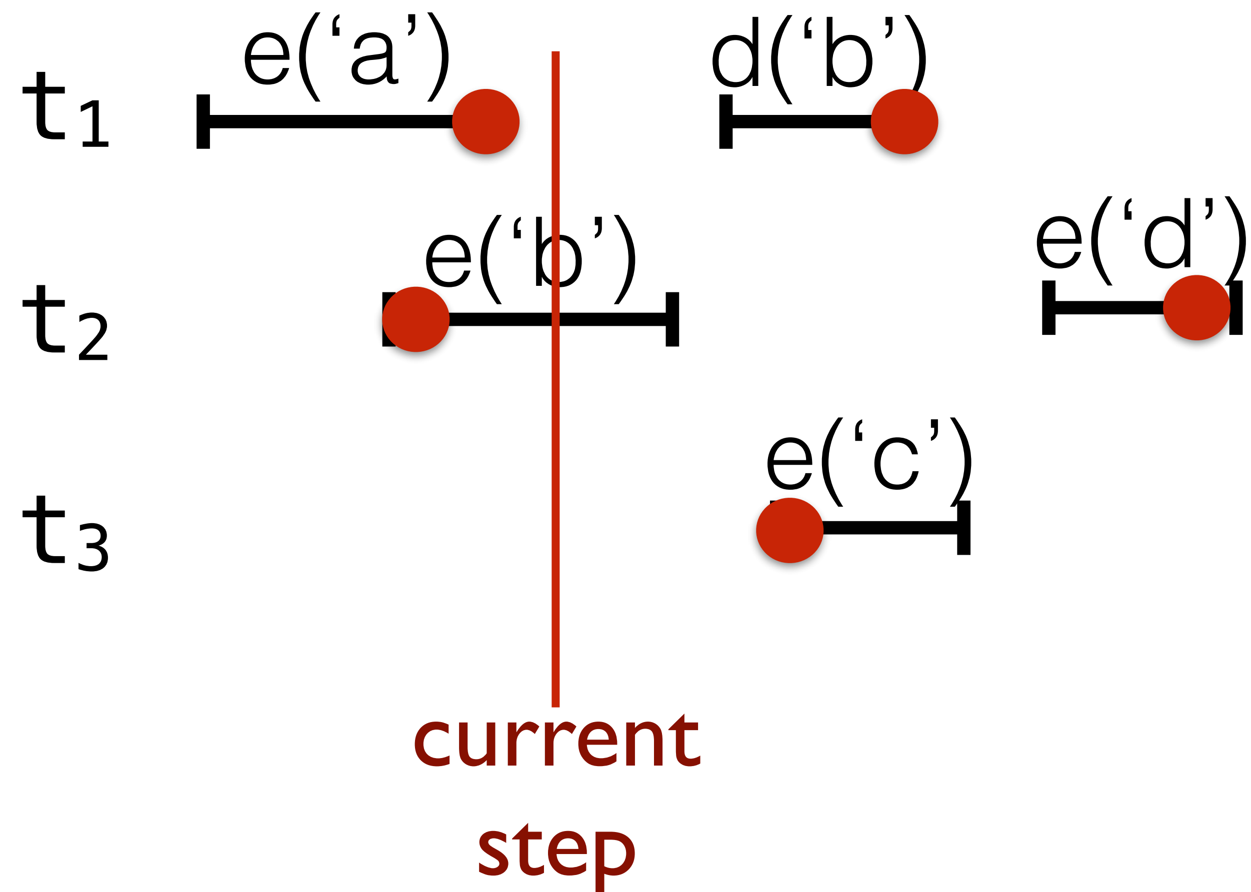
for every concrete history



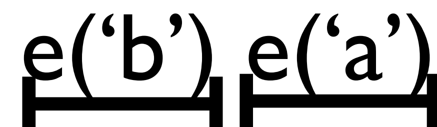
build a linearization: e('b')

Forward simulation

for every concrete history

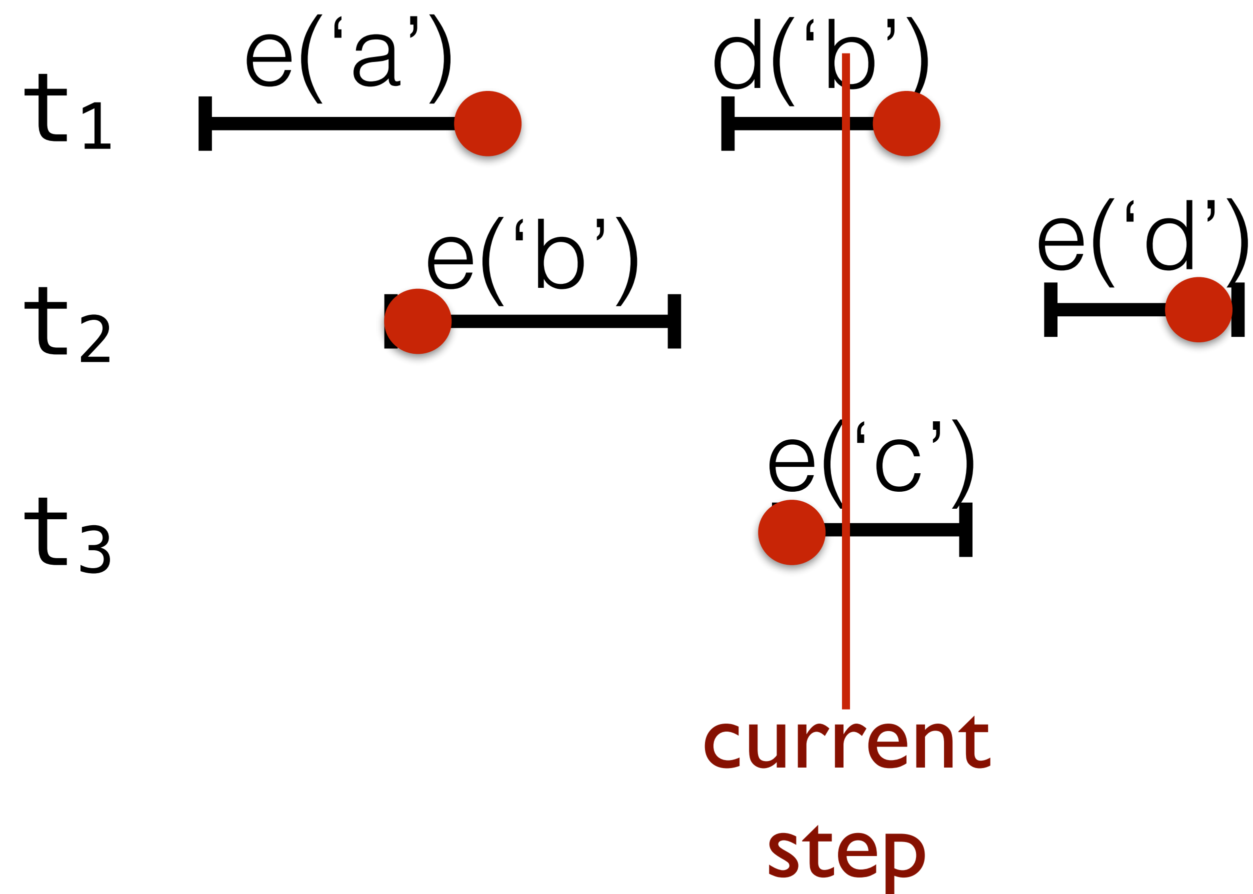


build a linearization:

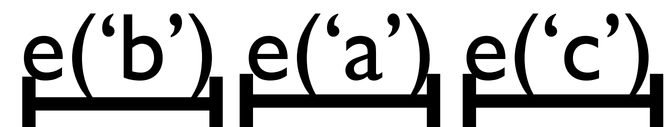


Forward simulation

for every concrete history

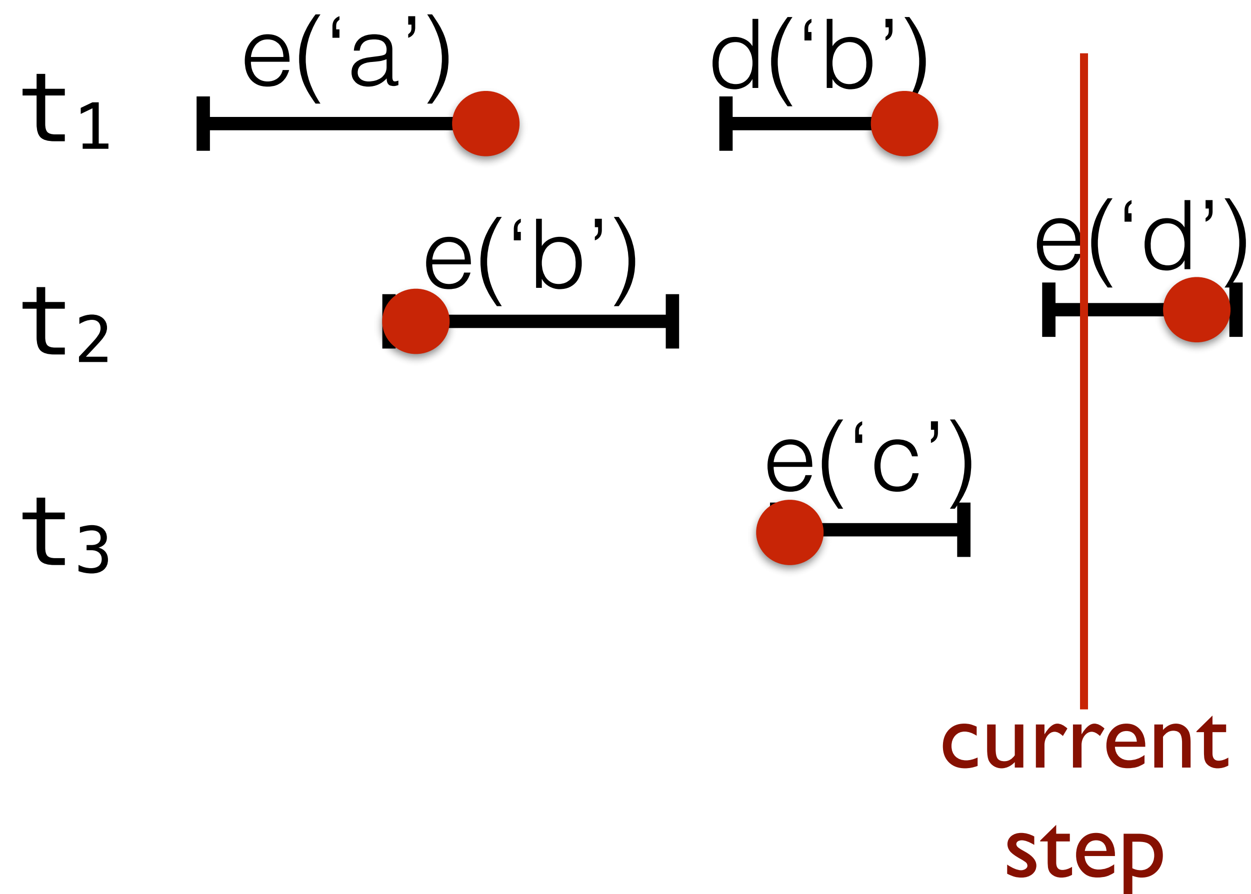


build a linearization:

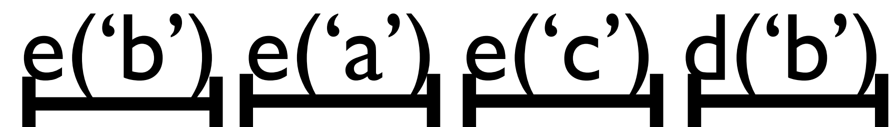


Forward simulation

for every concrete history

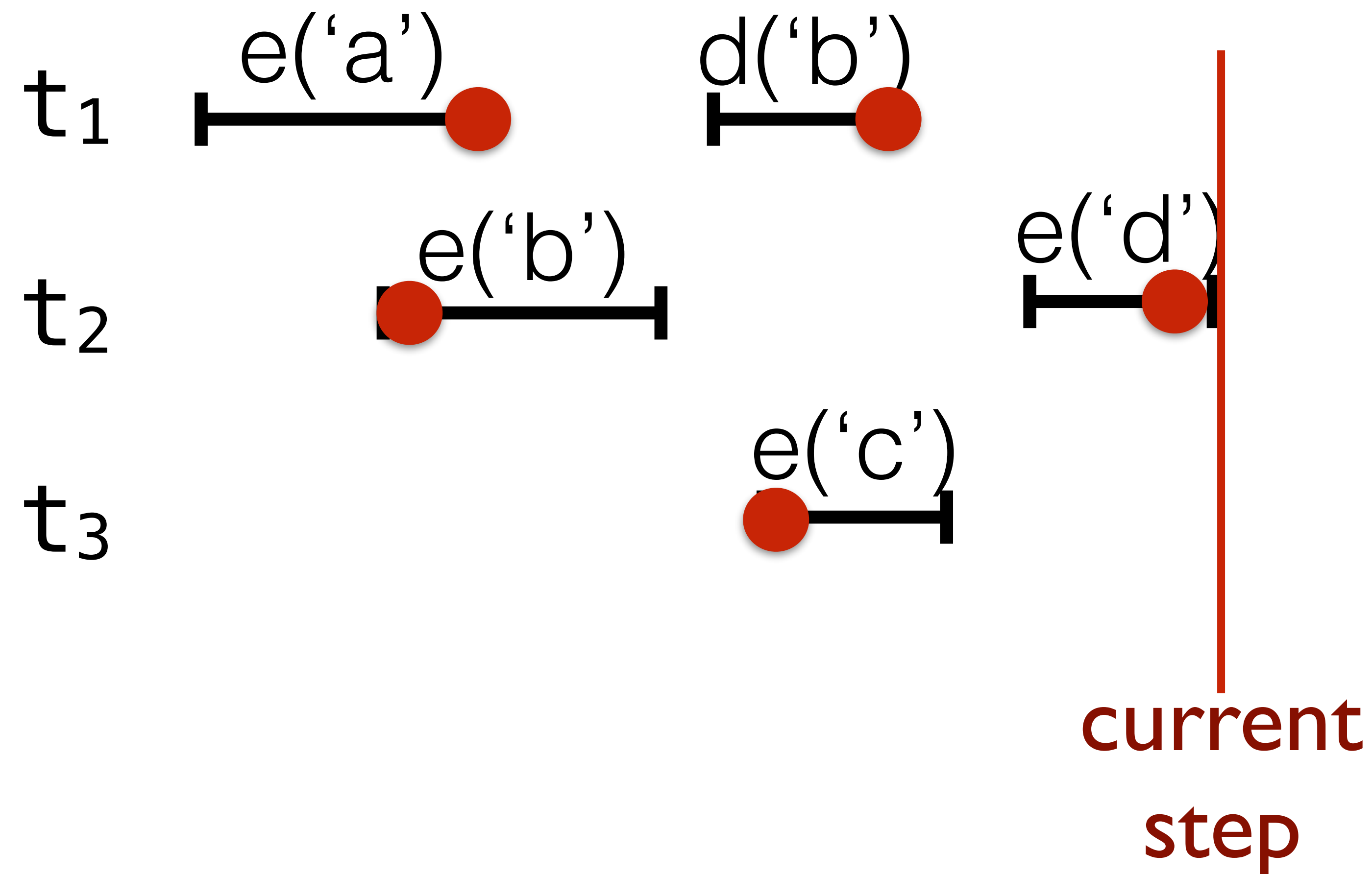


build a linearization:

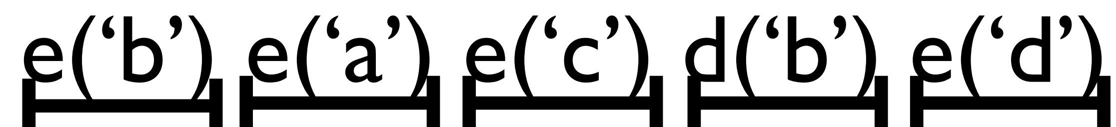


Forward simulation

for every concrete history

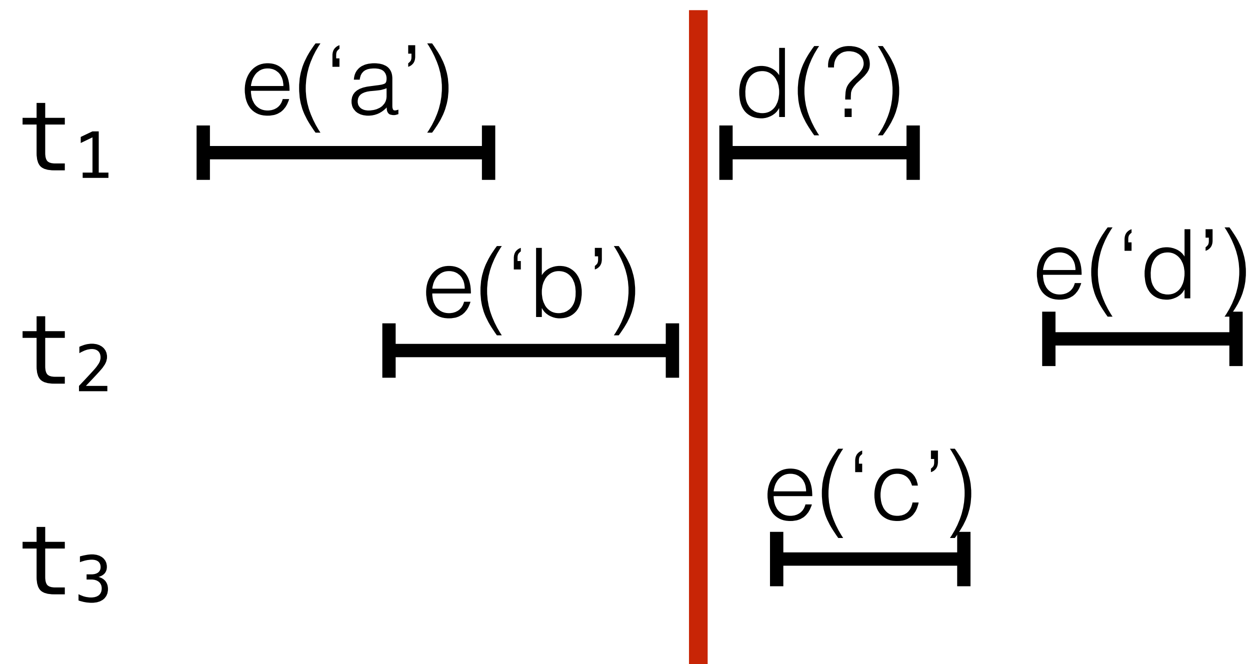


build a linearization:



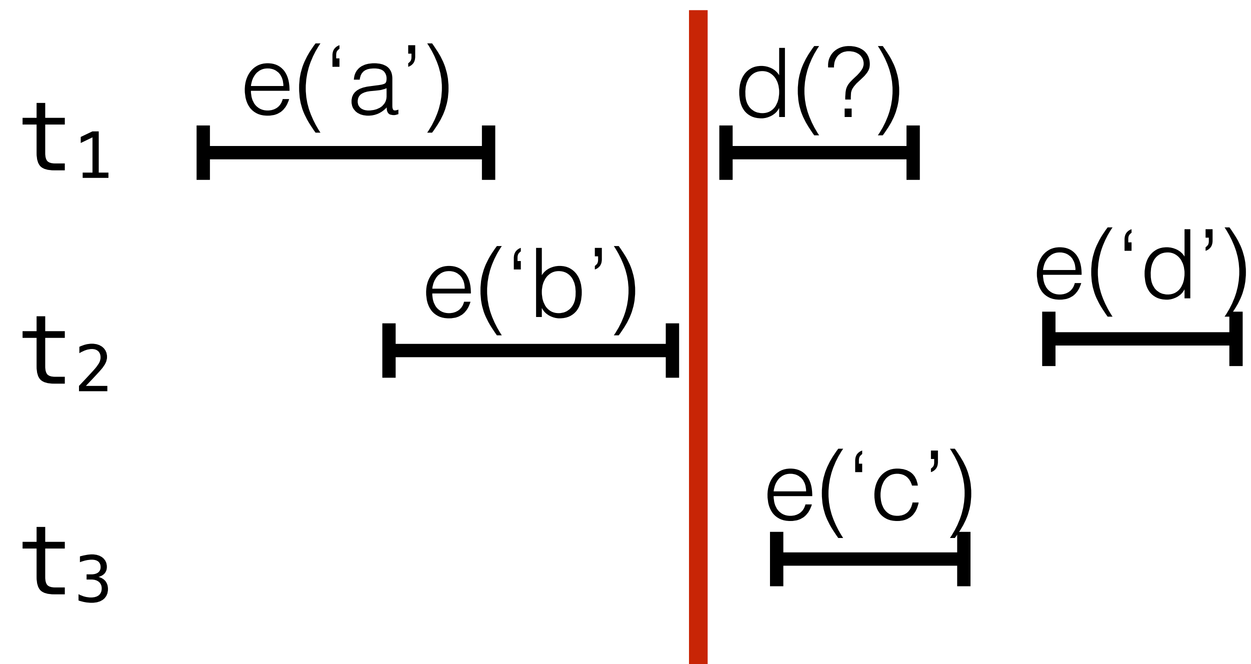
Problem

for the concrete history



- by the red moment we still may not know where the linearization points of e('a') and e('b') are

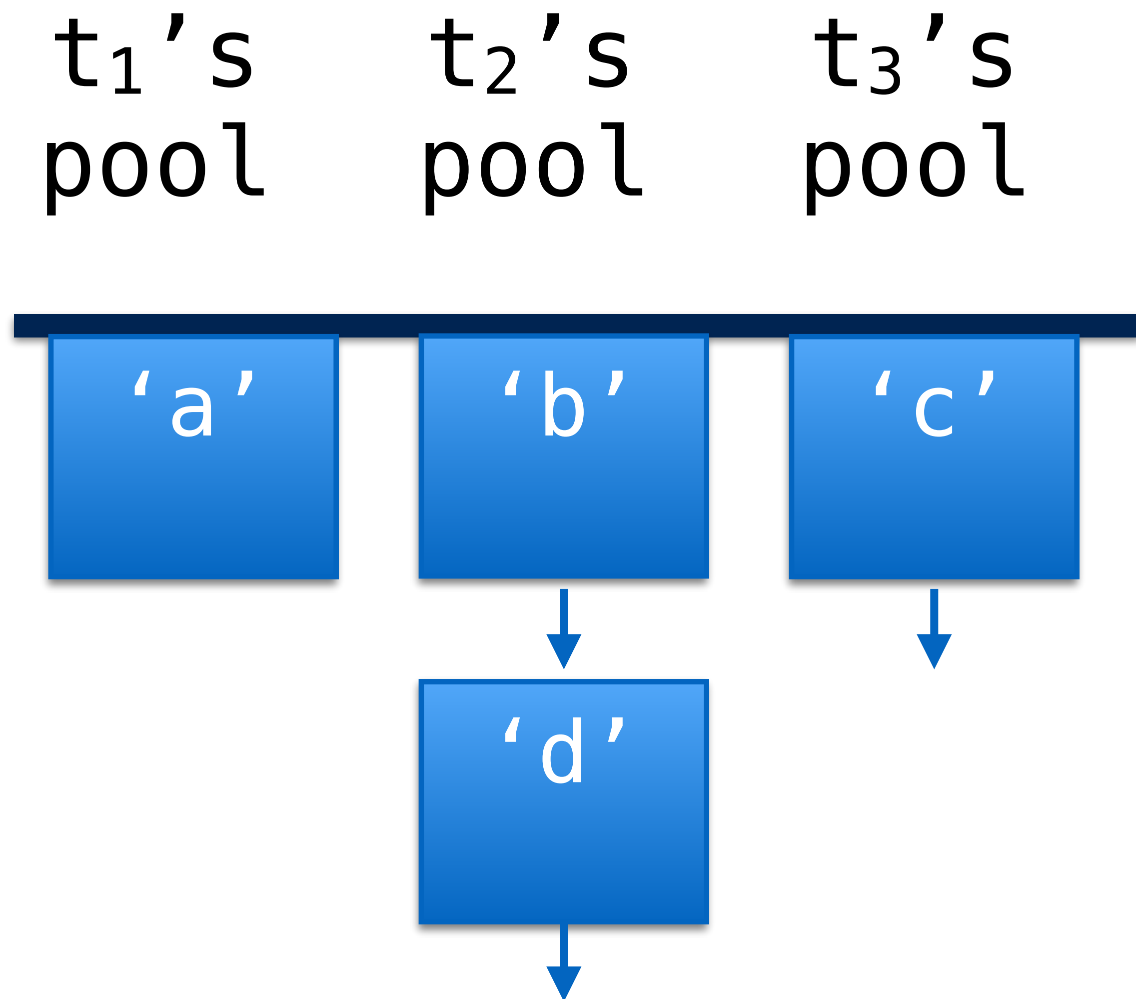
Problem



Actually happens in some algorithms:

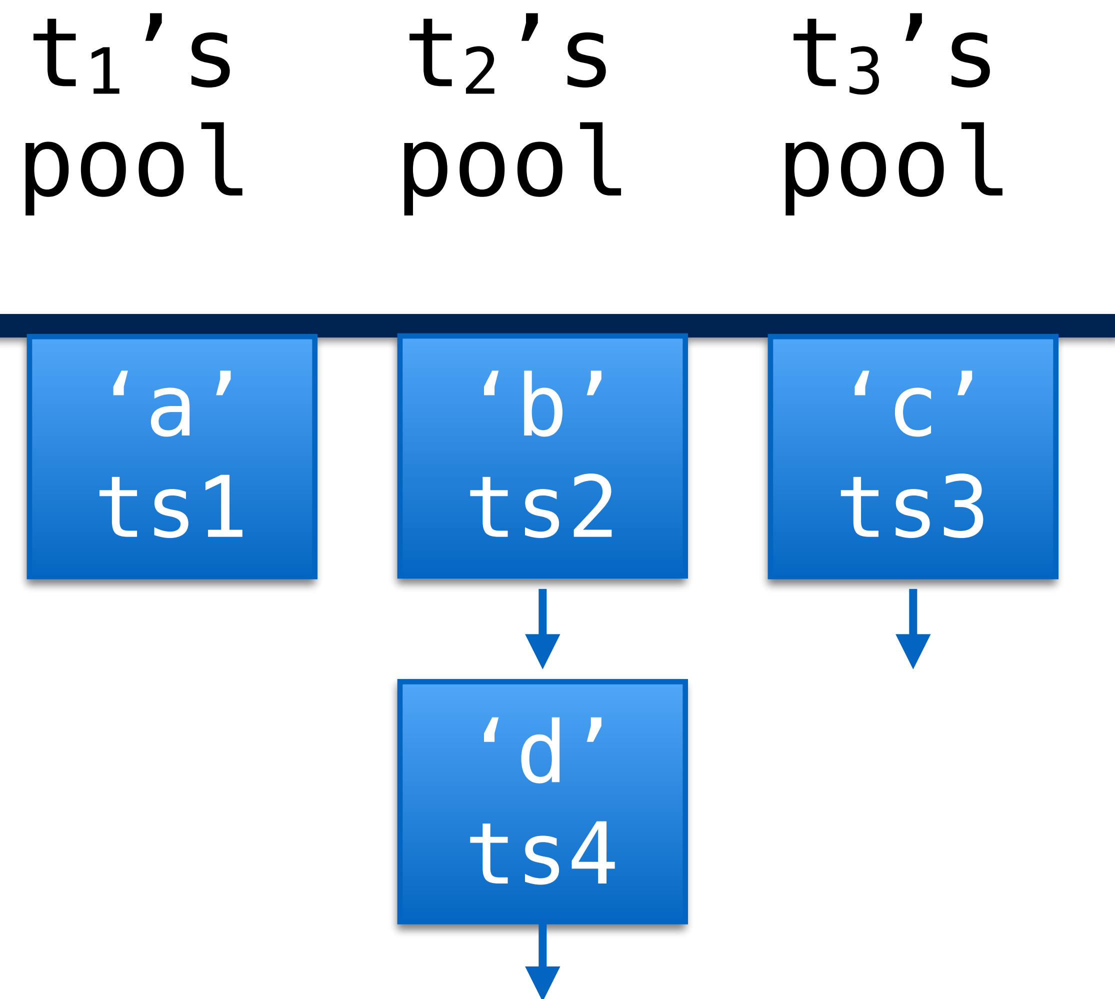
- Herlihy-Wing Queue
- The time-stamped stack (POPL'15)
- The time-stamped queue (PhD thesis of A. Haas)

The TS queue



- each thread has its own pool
- pool is an abstract sequence
- (single producer multiple consumers)

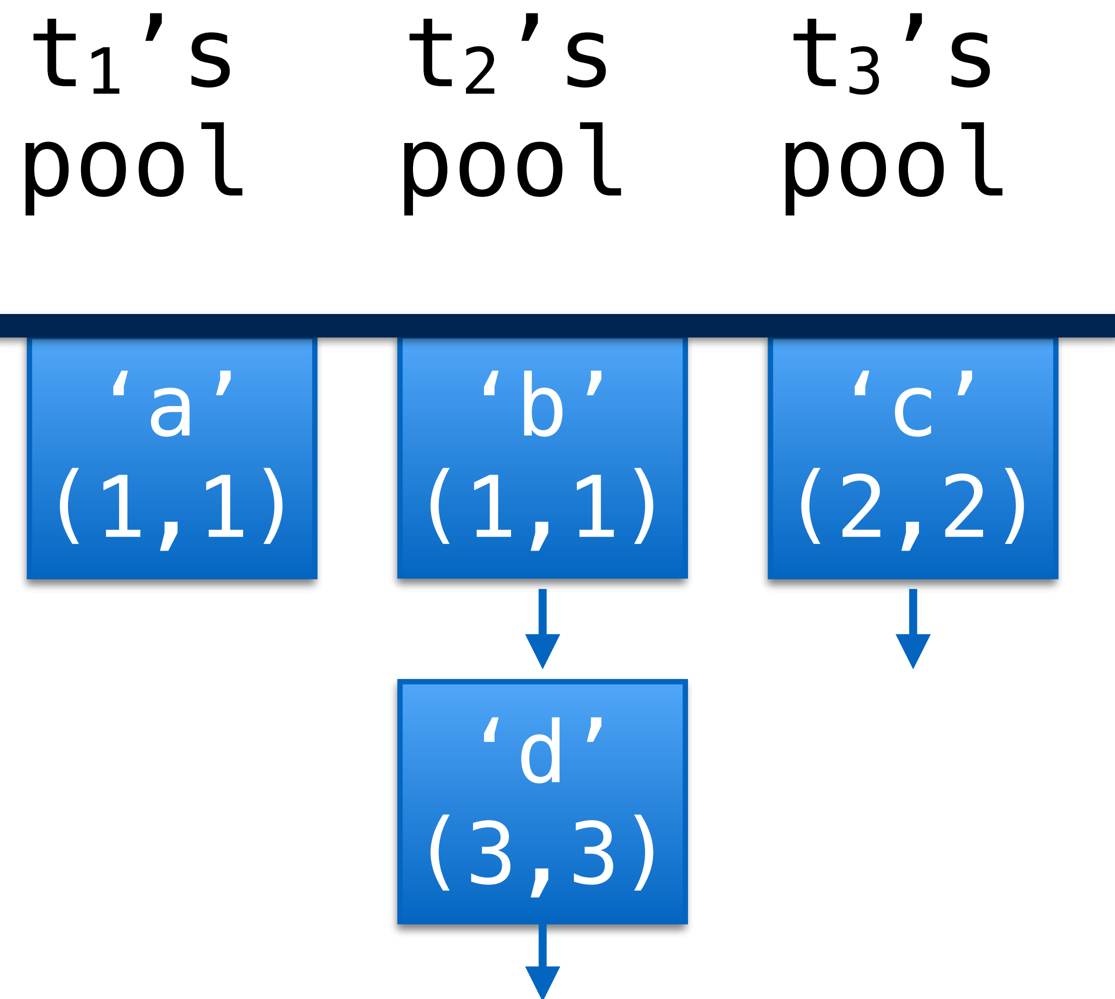
The TS queue




```
enqueue(Val v) {
   ts := newTimestamp();
   insert(this_thread, v, ts);
}
```

- (the algorithm is slightly simplified for the presentation)

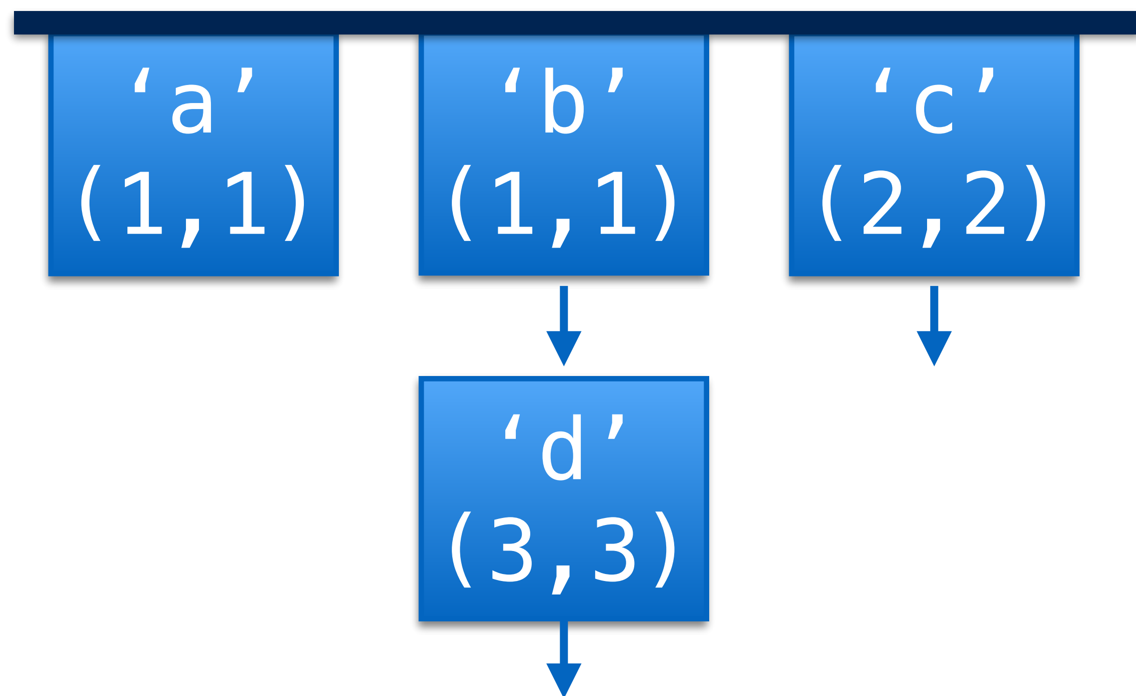
Timestamps




```
enqueue(Val v) {
  ts := newTimestamp();
  insert(this_thread, v, ts);
}
```

- timestamps = intervals
- $(a, b) < (c, d)$ iff $b < c$

The TS queue



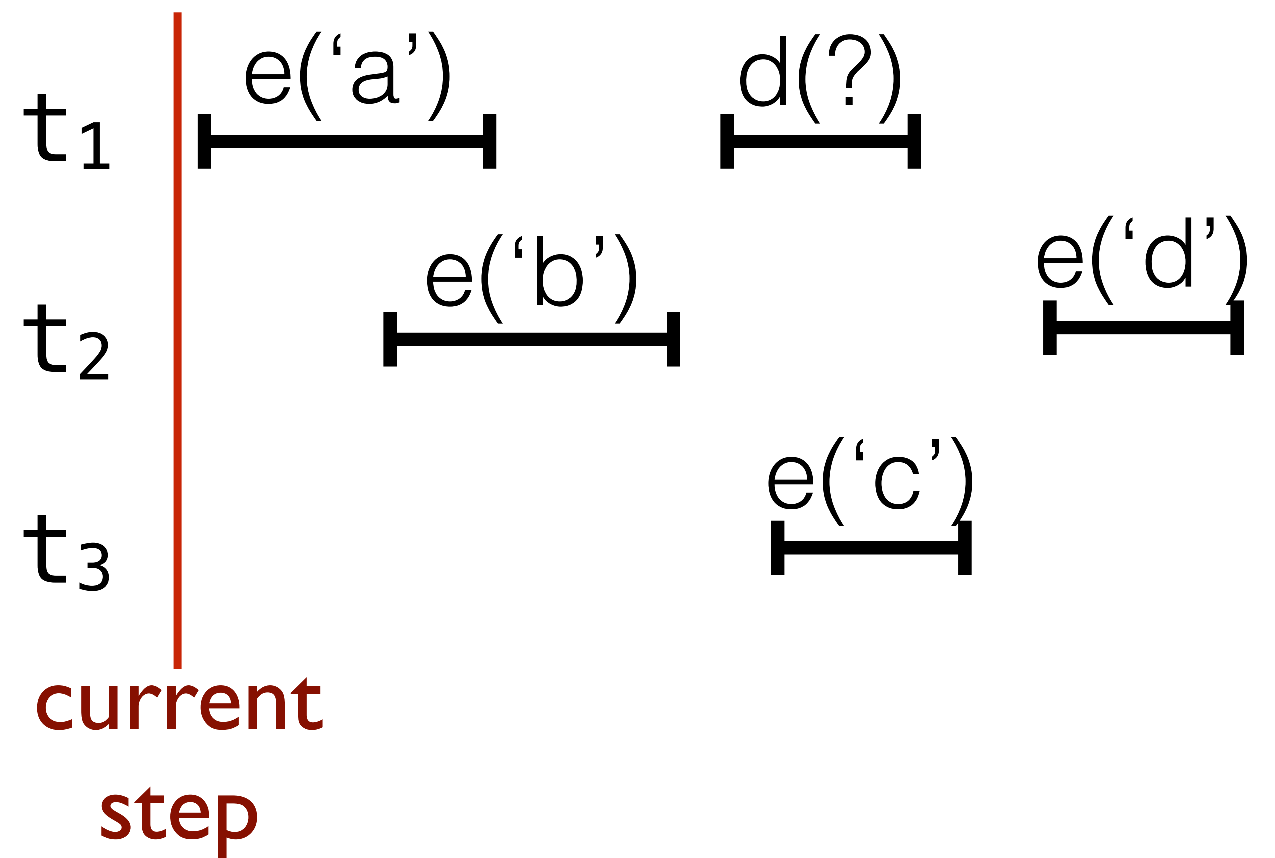

```
int counter = 1;

TS newTimestamp() {
   int l := counter;
   if CAS(counter, l, l+1)
      r := l;
   else
      r := counter-1;
   return (l, r);
}
```

The TS queue

t_1's pool

t_2's pool

t_3's pool

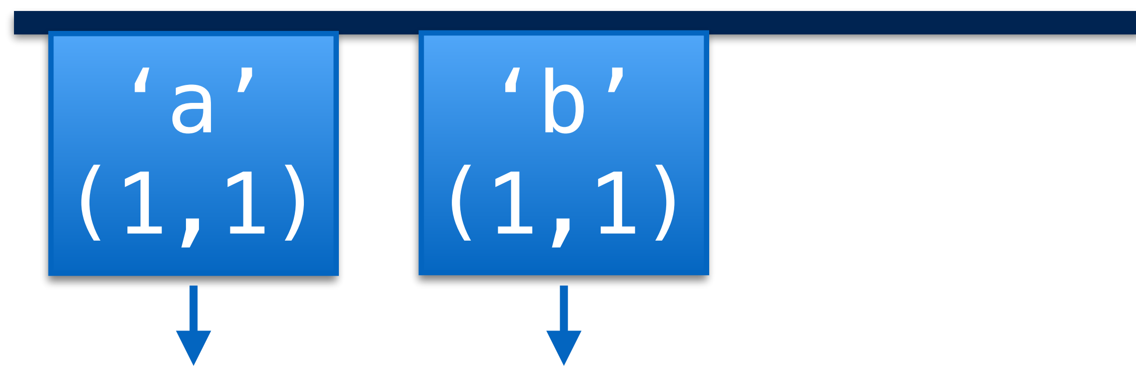


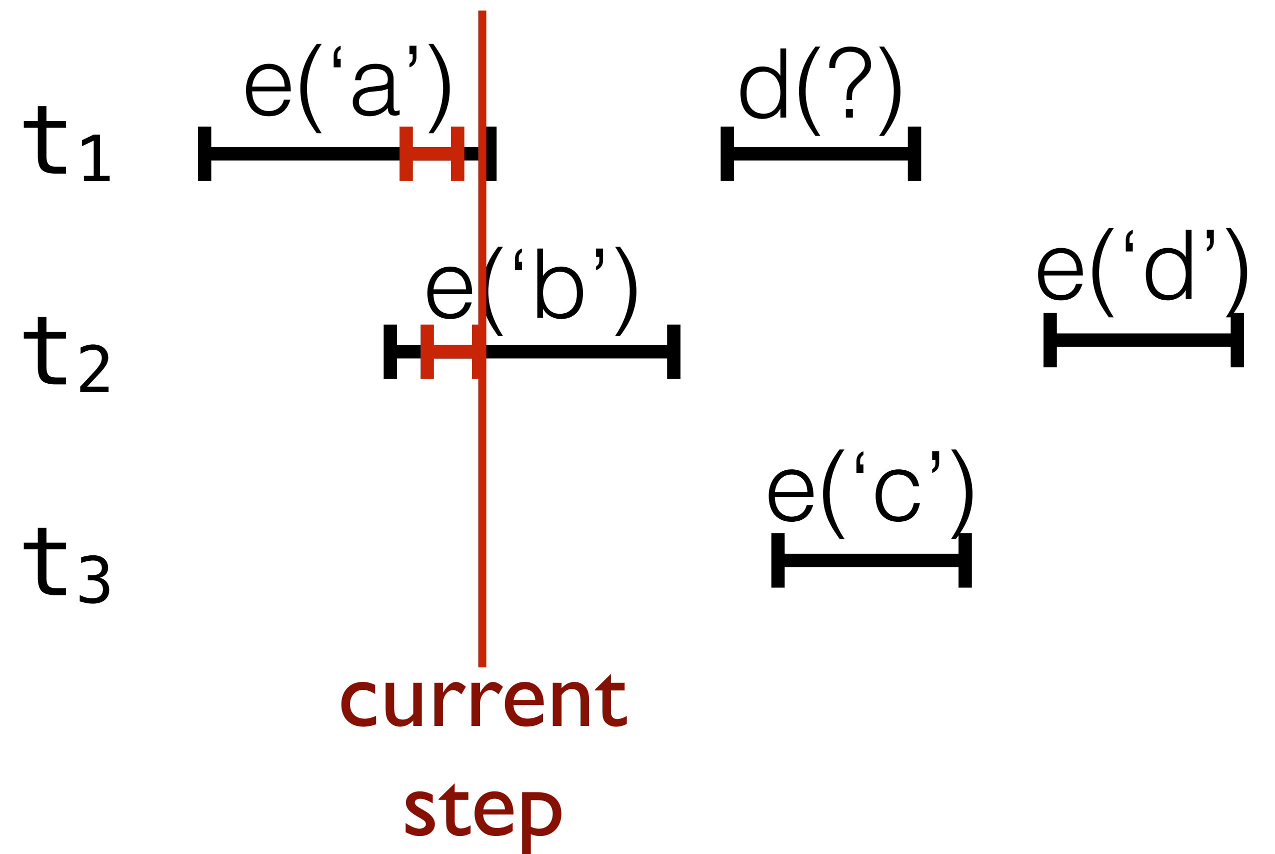
The TS queue

t_1's pool

t_2's pool

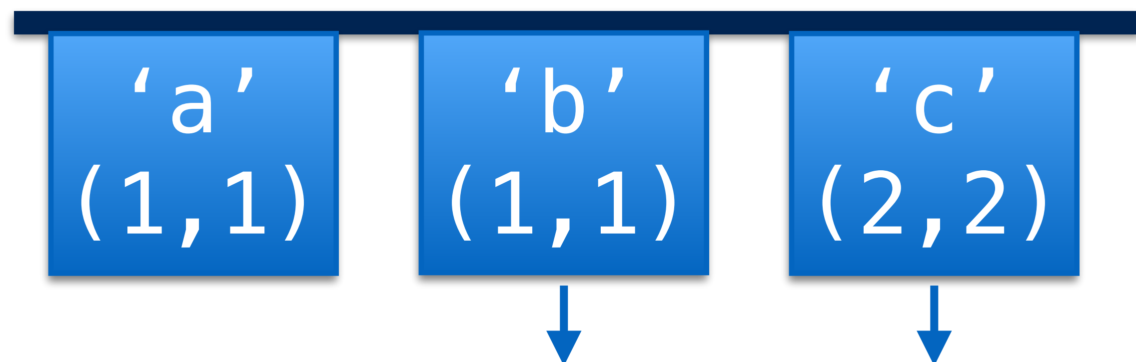
t_3's pool





The TS queue

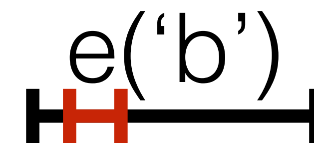
t_1's pool t_2's pool t_3's pool



t_1

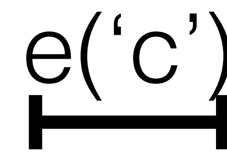
d(?)

t_2



e('d')

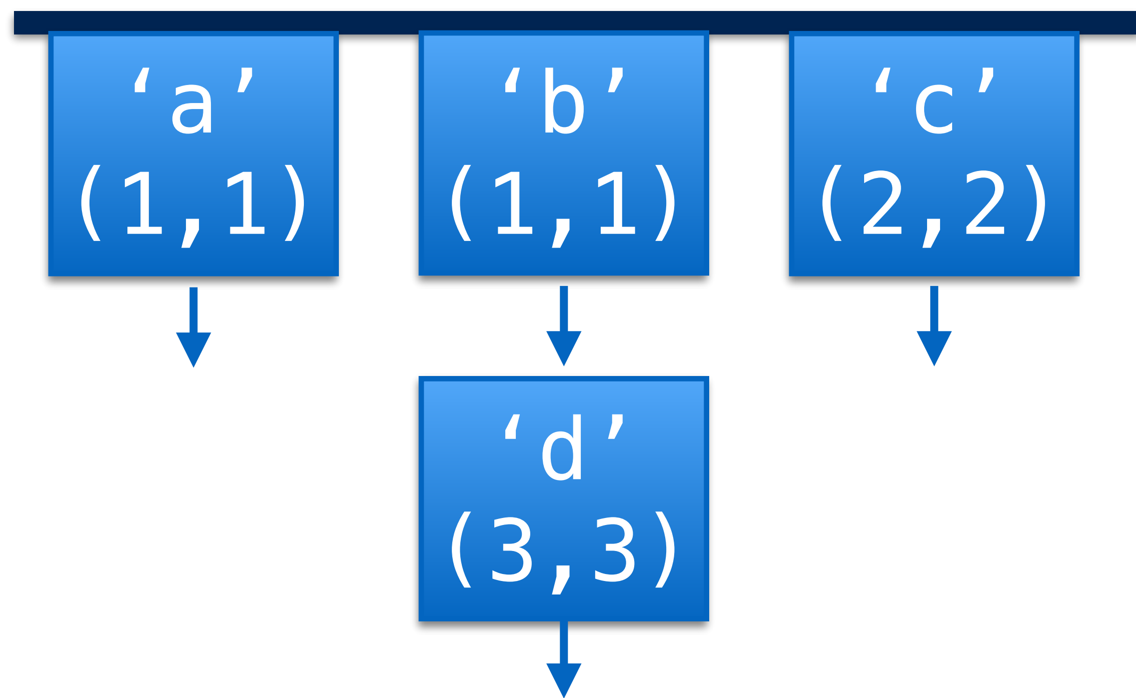
t_3



current step

The TS queue

t_1's pool t_2's pool t_3's pool



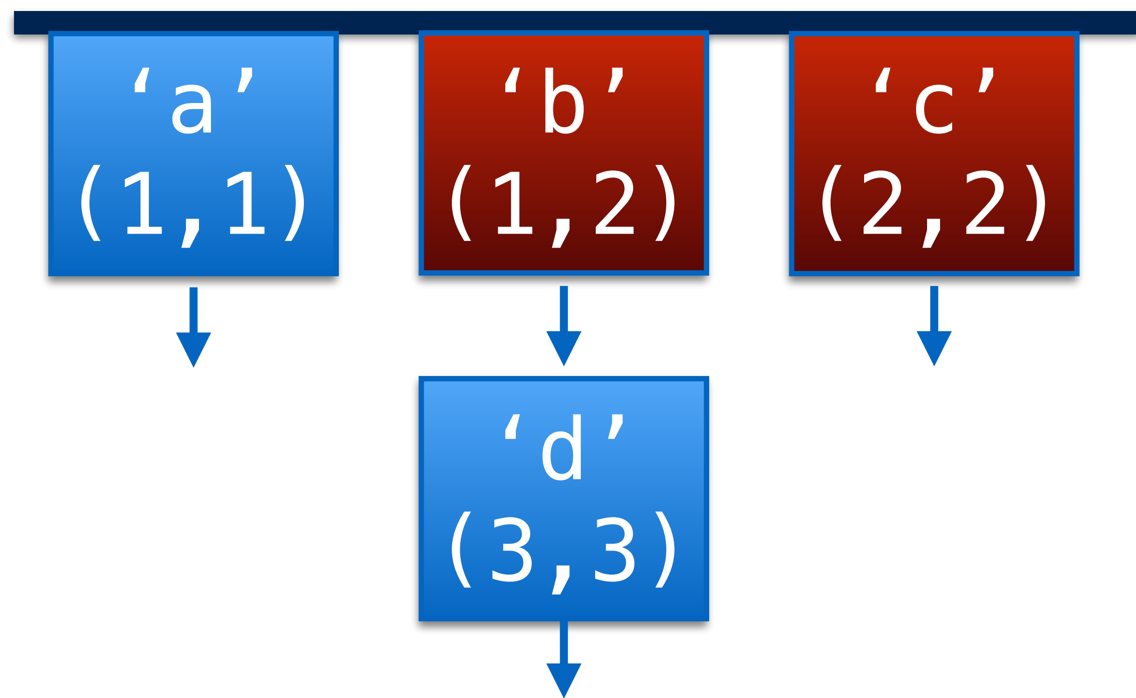
t_1 e('a') d(?)

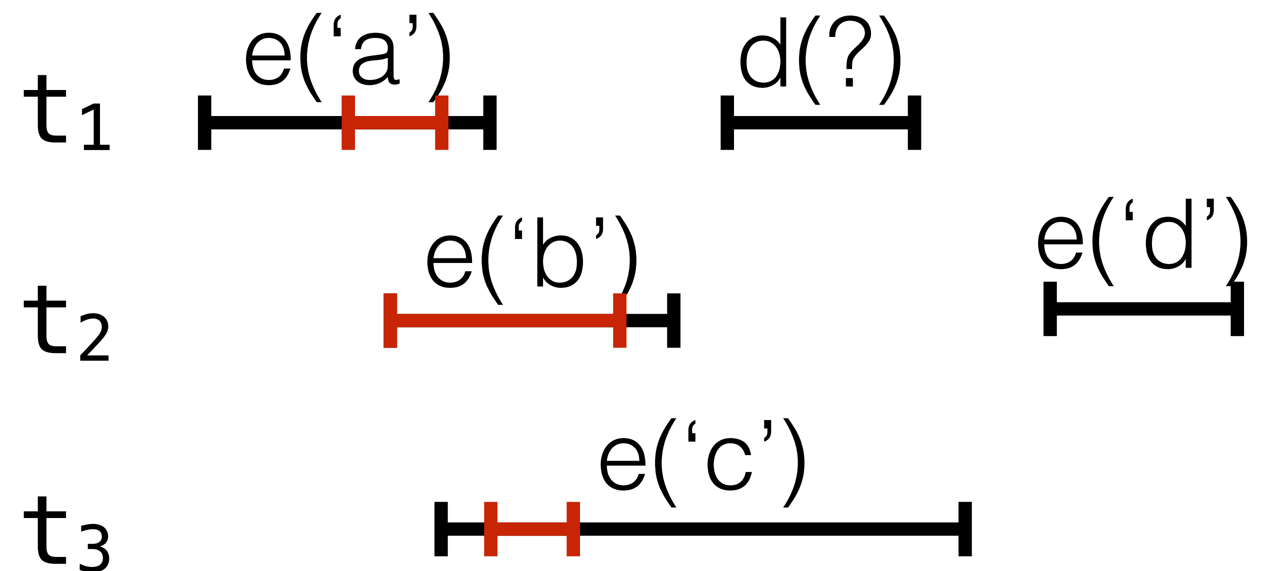
t_2 e('b') e('d')

t_3 e('c')

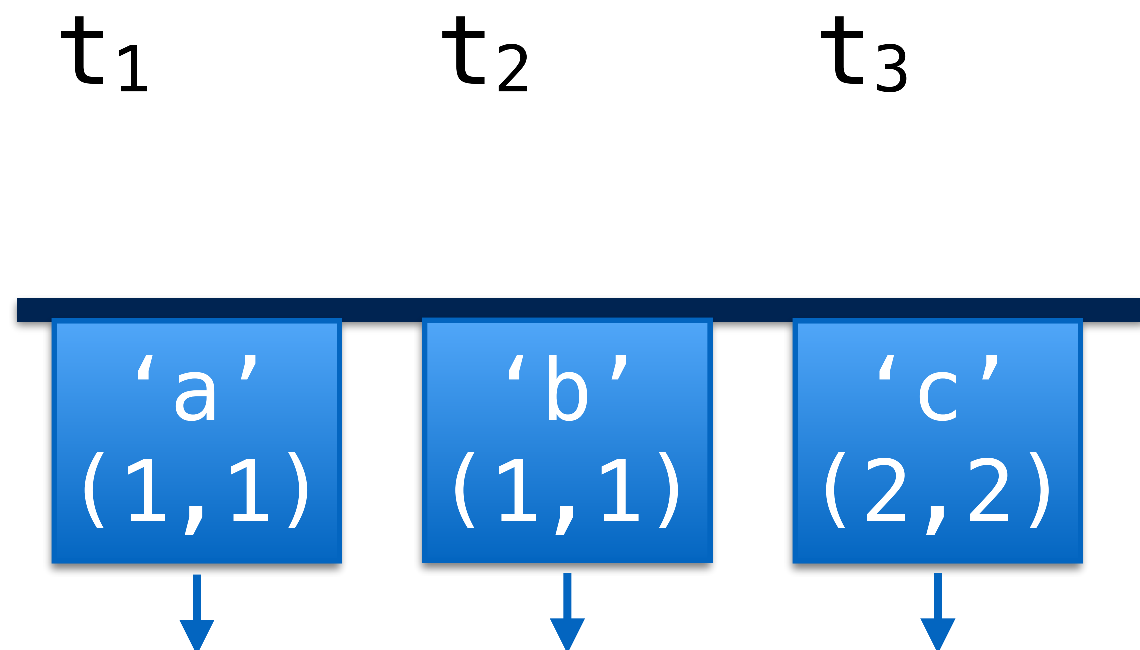
current step

The TS queue



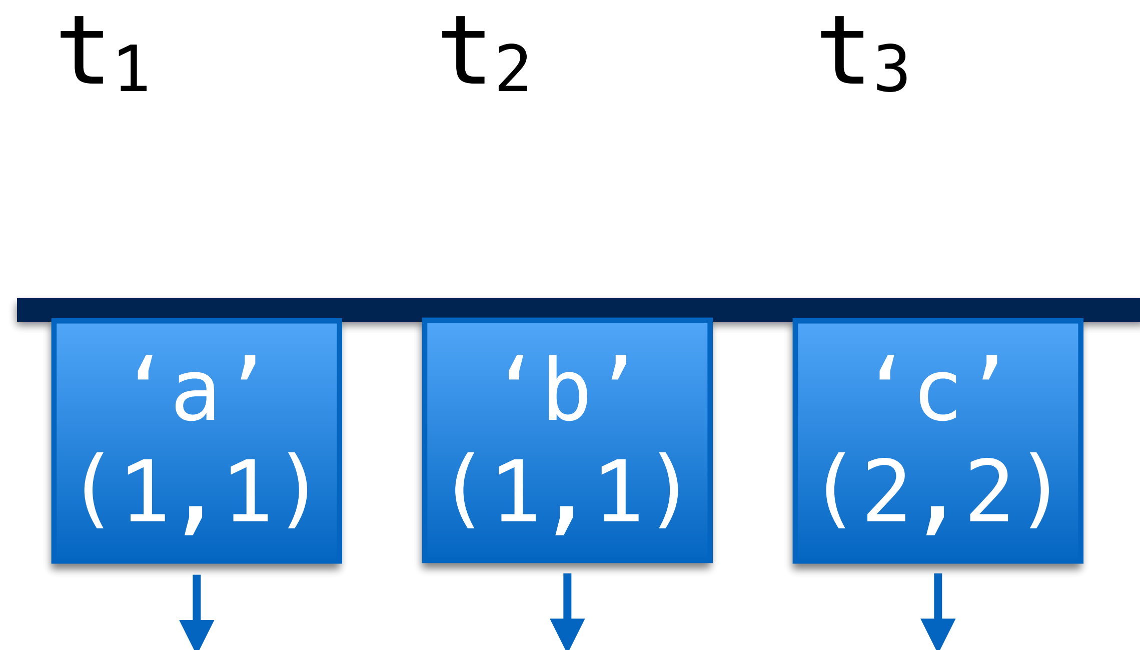


The TS queue




```
Val dequeue() {
  do {
    for each pool do {
      look at the front node;
      update the candidate
                for removal;
    }
    if (there is a candidate)
      try removing it and
      returning its value;
  } while (true);
}
```

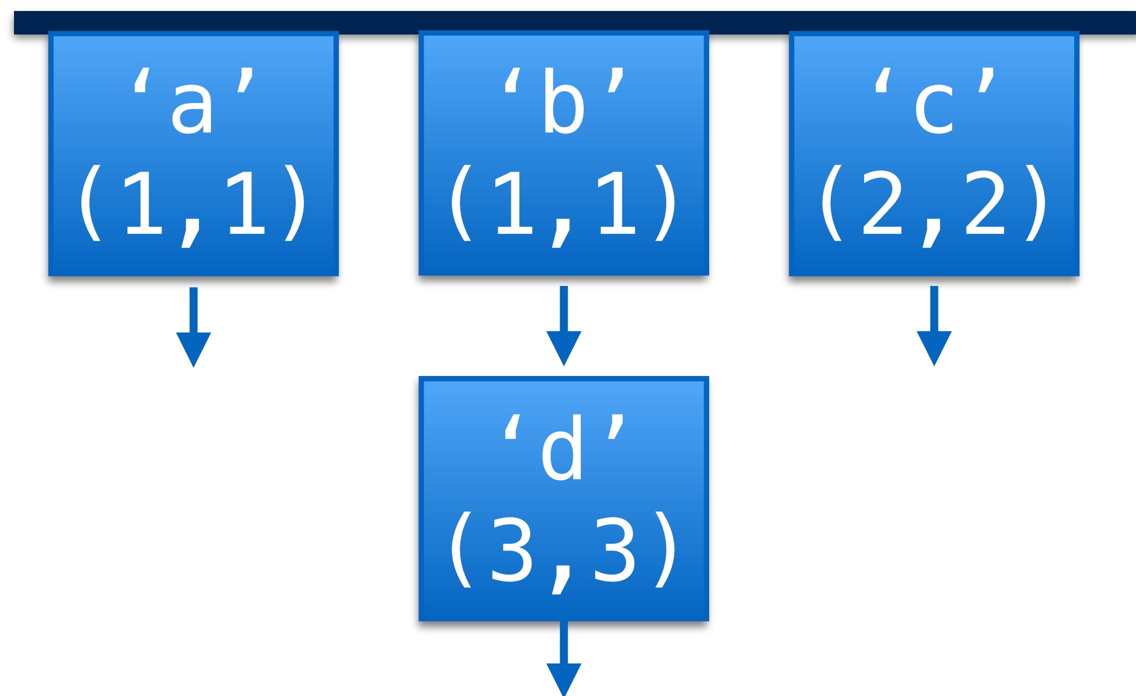
The TS queue

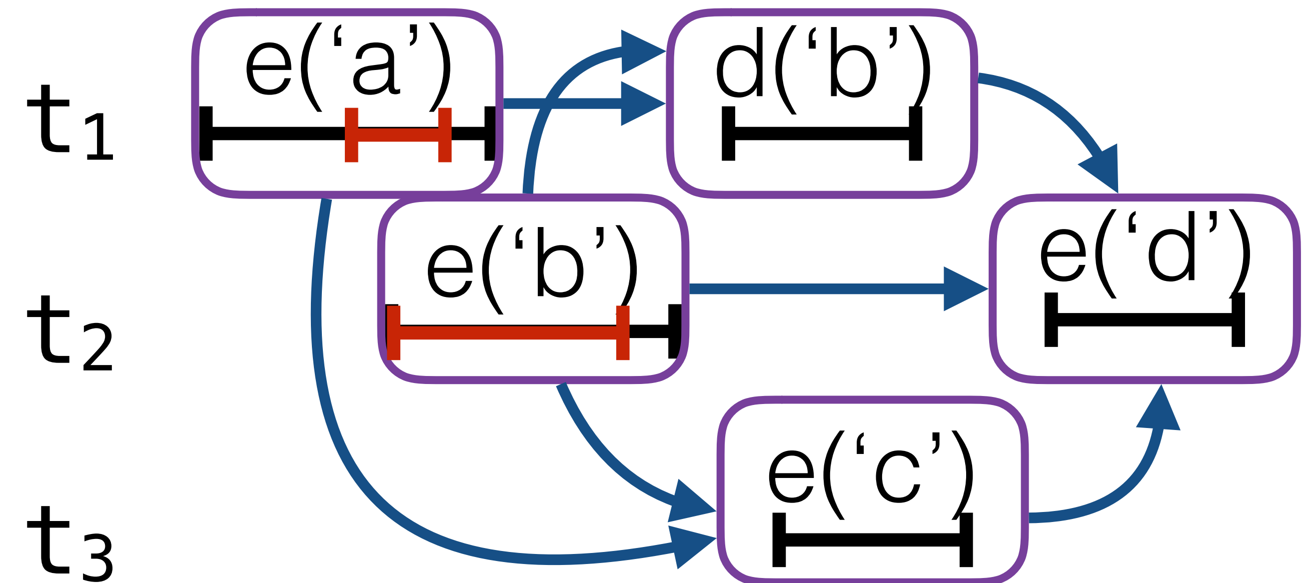



```
Val dequeue() {
  do {
    for each pool do {
      look at the front node;
      update the candidate
                for removal;
    }
    if (there is a candidate)
      try removing it and
      returning its value;
  } while (true);
}
```

Timestamps

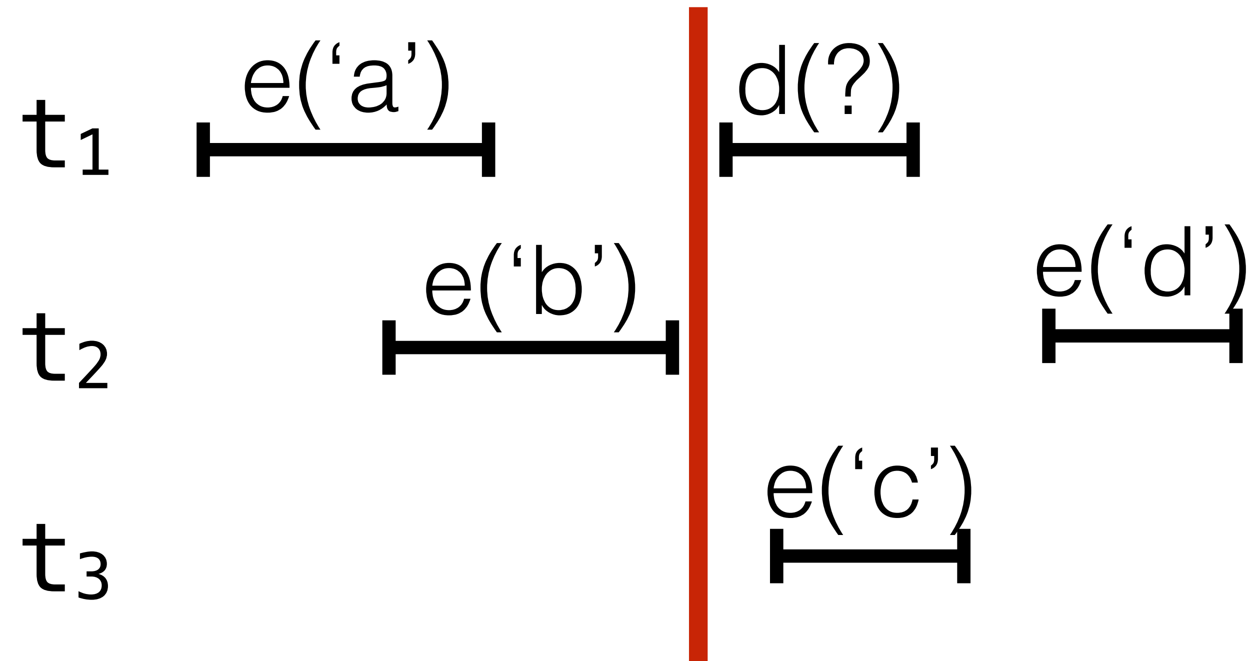
t_1's pool t_2's pool t_3's pool





- the connection with the real-time order
- e('a') —rt—> e('b') ==> timestamp(e('a')) < timestamp(e('a'))

Problem



- by the red moment (*) we still may not know where the linearization points of e('a') and e('b') are

Our technique

- ✓ We alter the standard proof technique to support late choice via partial orders
- ✓ Our proof technique = checking thread/method-modular constraints (via Rely-Guarantee)
- ✓ Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set

Abstract histories

- For each history of a data structure we construct a matching abstract history (multiple linearizations)
- An “abstract” history (E, R):
  - E — a set of events [eid: (tid, op, arg, rval)]
  - R — a partial order

Abstract histories

- For each history of a data structure we construct a **matching** abstract history (multiple linearizations)

1. the abstract history extends the real-time order
2. all linearizations meet the sequential spec
3. the abstract history is acyclic (has linearizations)

Abstract histories

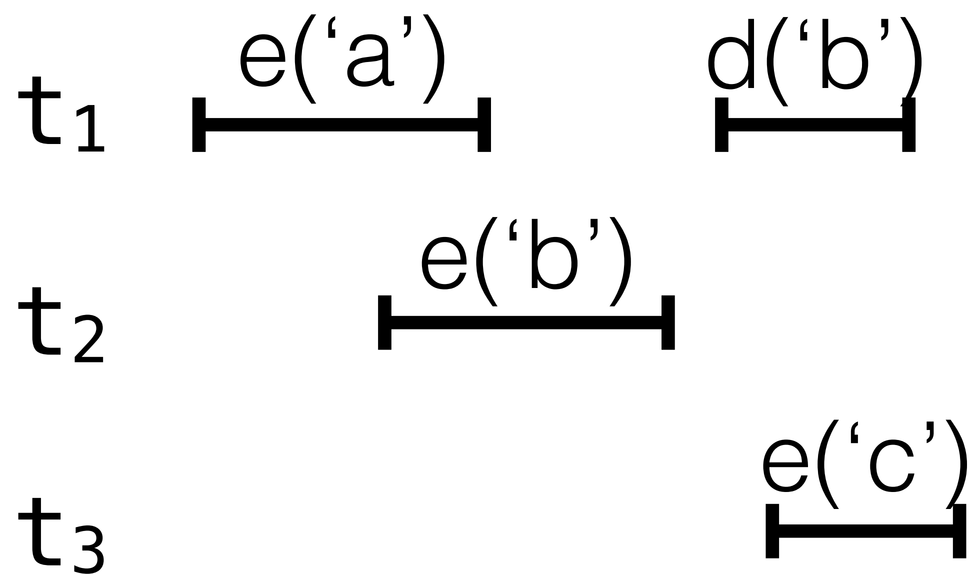
- We prove a number of invariant properties including:
  - all linearizations meet the sequential spec
  - for all enqueues e, e’ with values in the data structure:
    - e —> e’ ==> timestamp(e) < timestamp(e’)
  - …

Commitment points

- Abstract histories are constructed:
  - by adding new events
  - by adding more edges into the partial order
  - by assigning a return value to an event

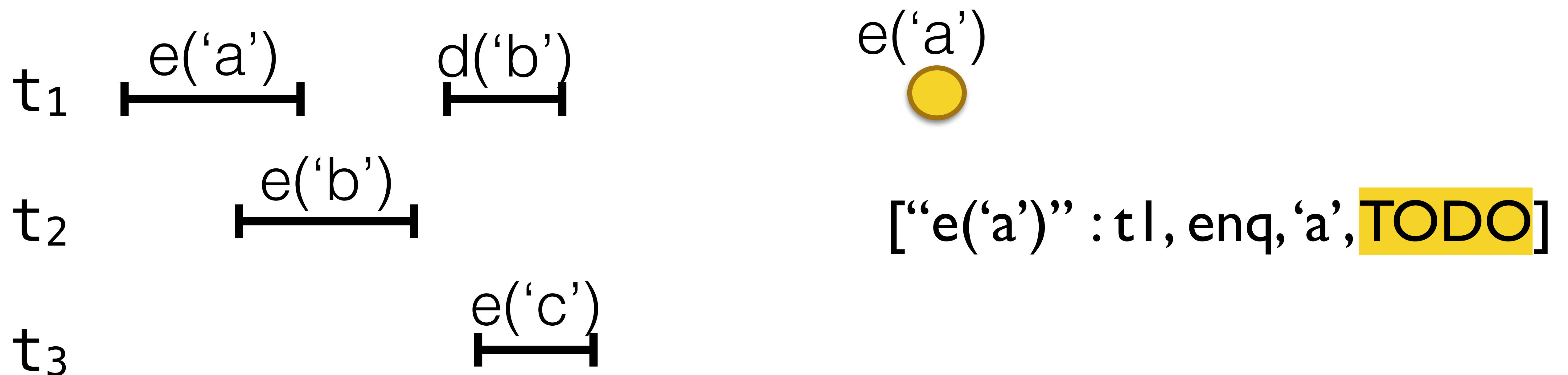
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



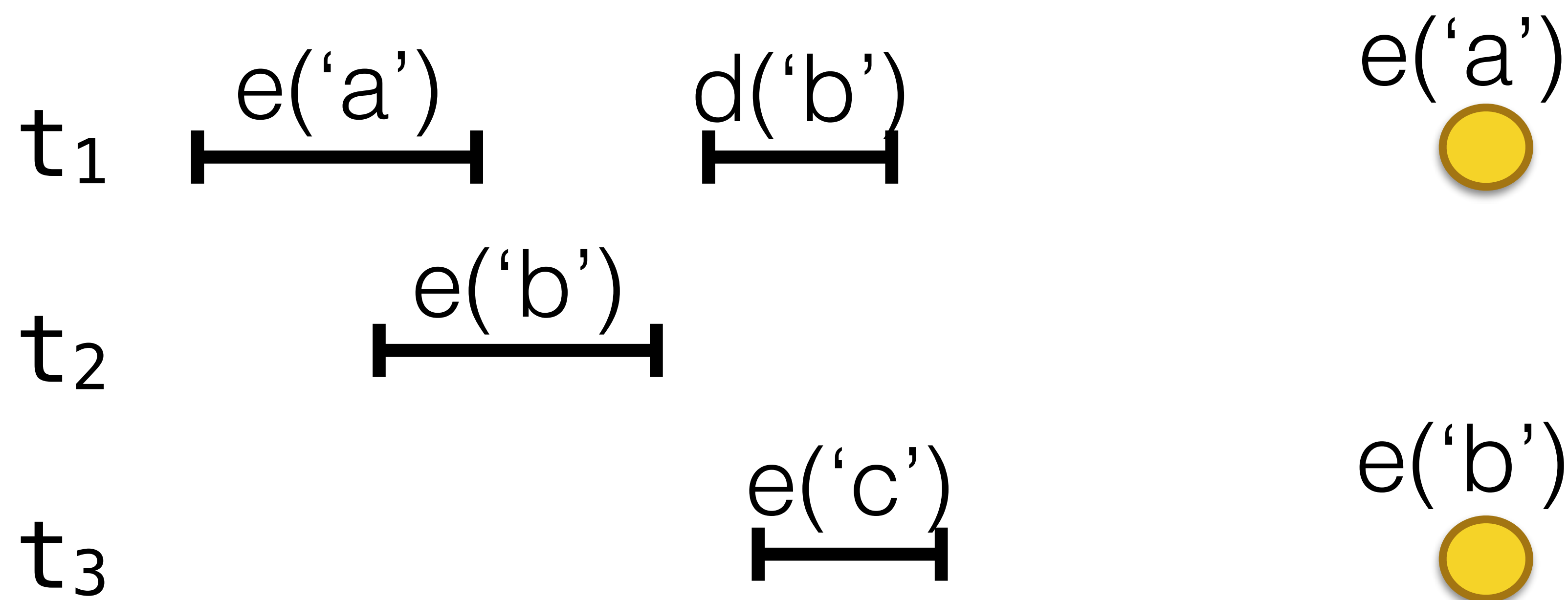
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



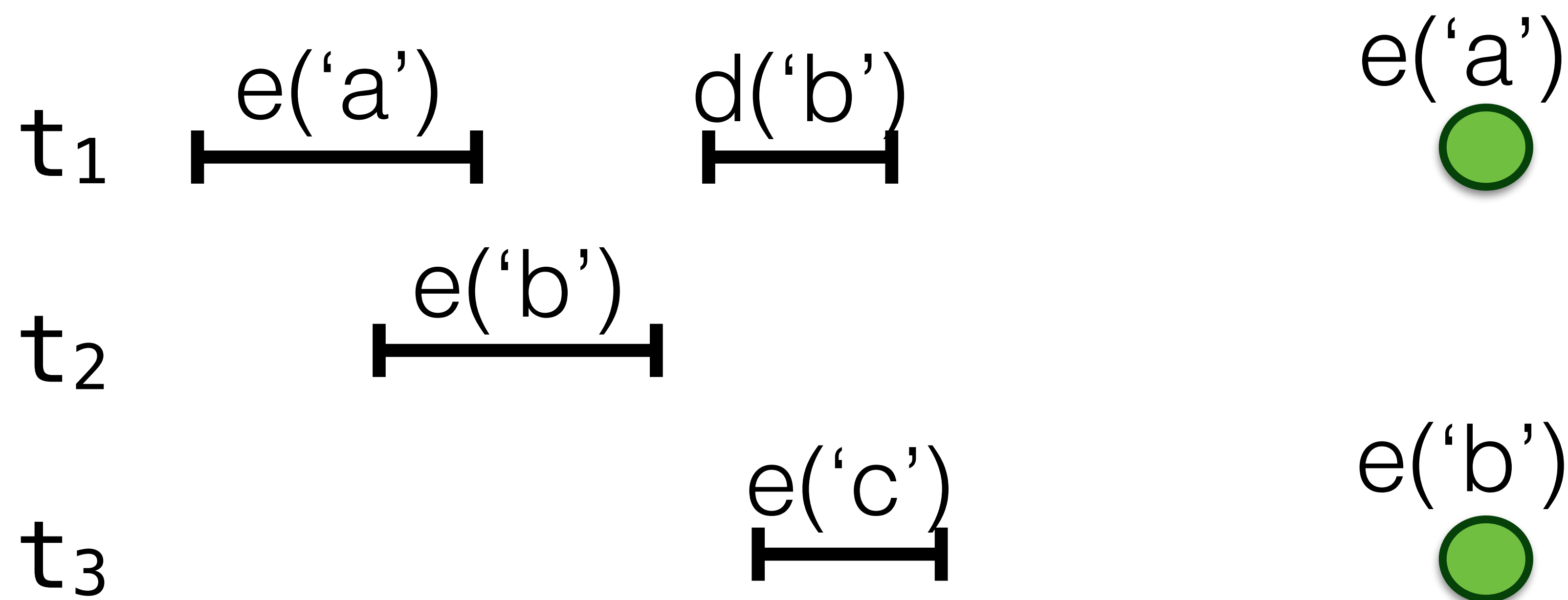
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



e('a')

e('b')

- events get completed by the end of the operations

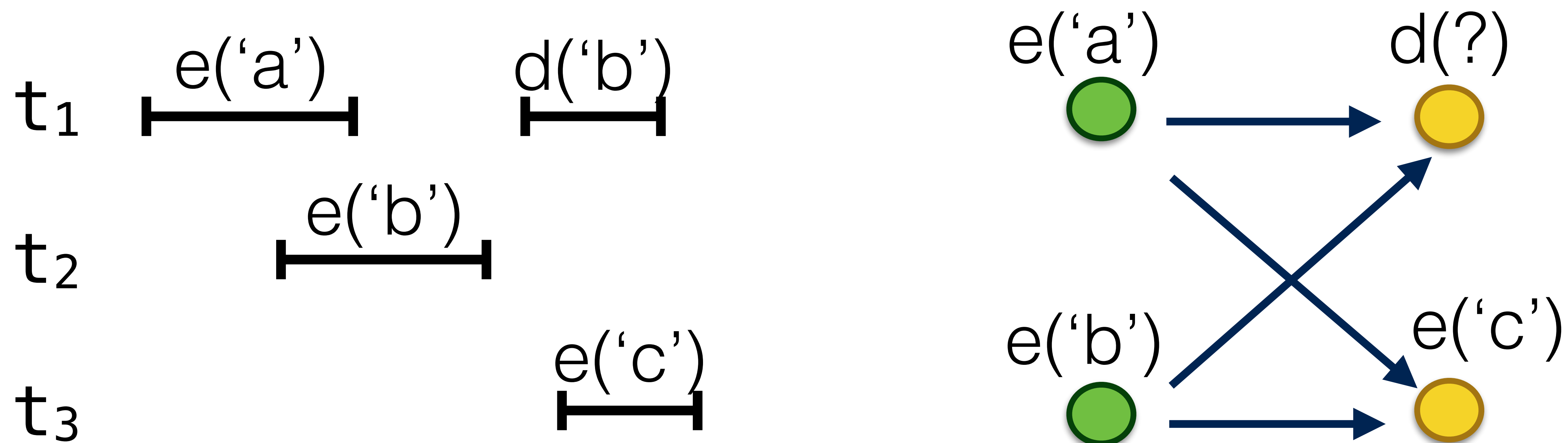
Enqueue's commitment point

```
enqueue(Val v) {
  ts := newTimestamp();
  atomic {
    insert(this_thread, v, ts);
    E(this_event).rval := DONE;
    G[this_event] := ts;
  }
}
```

- Ghost state G is a map from events and timestamps
- Helps to establish a bijection between events and elements of the data structure

Commitment points

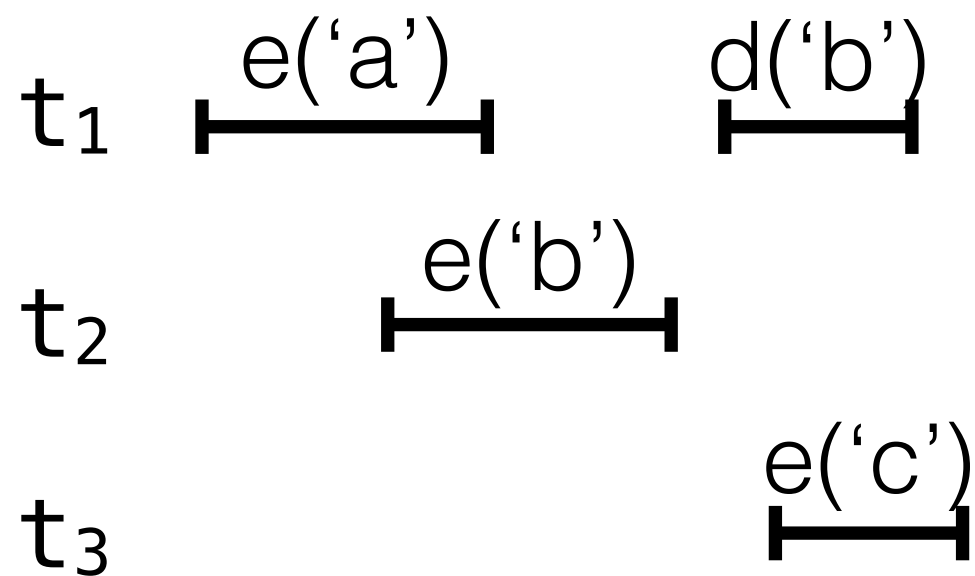
- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history

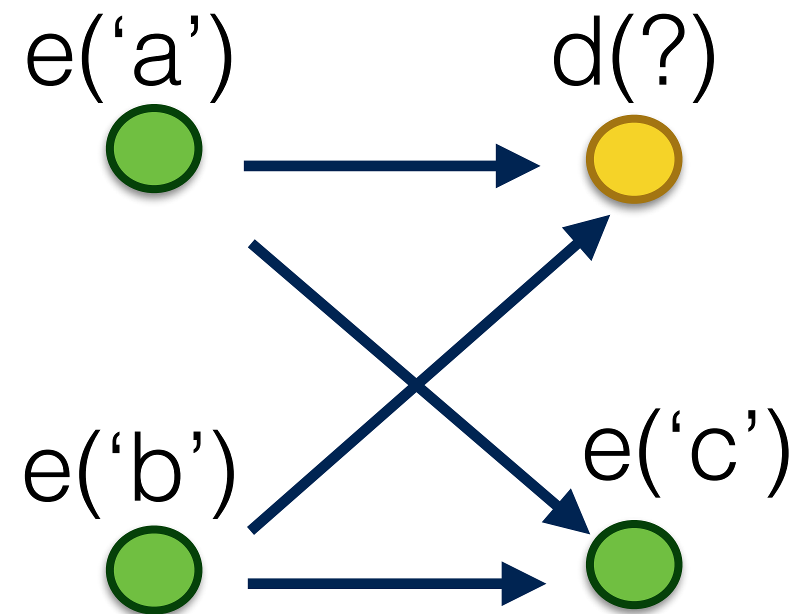


- edges from the completed events are added

Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

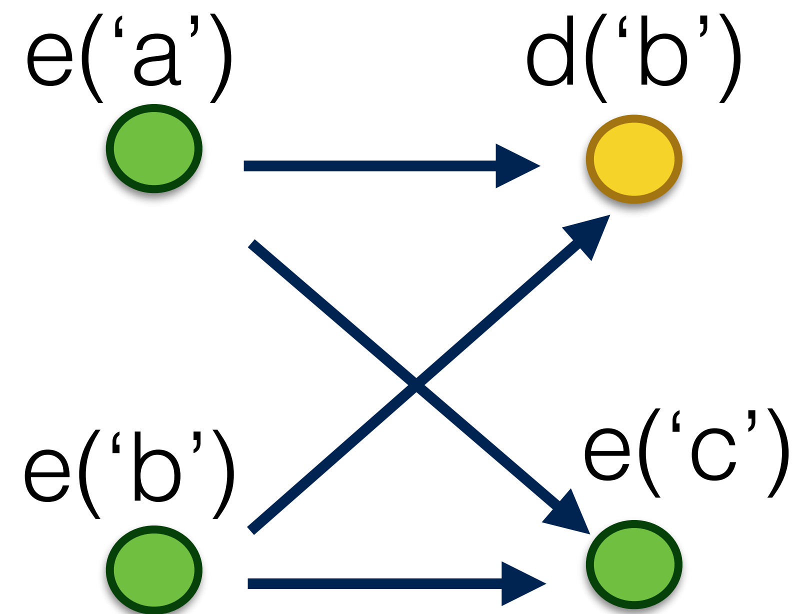




Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

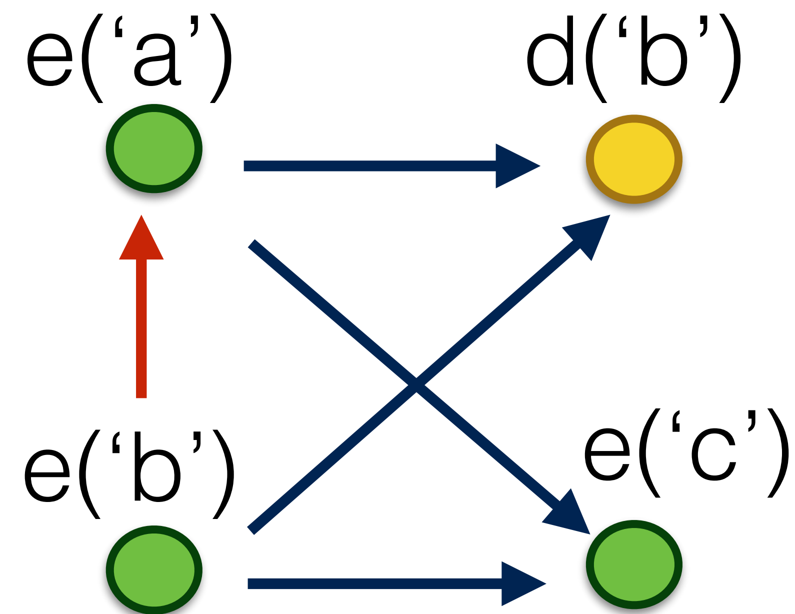
1. e('b') e('a') d('b') e('c')
2. e('b') e('a') e('c') d('b')
3. e('a') e('b') d('b') e('c')
4. e('a') e('b') e('c') d('b')



Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

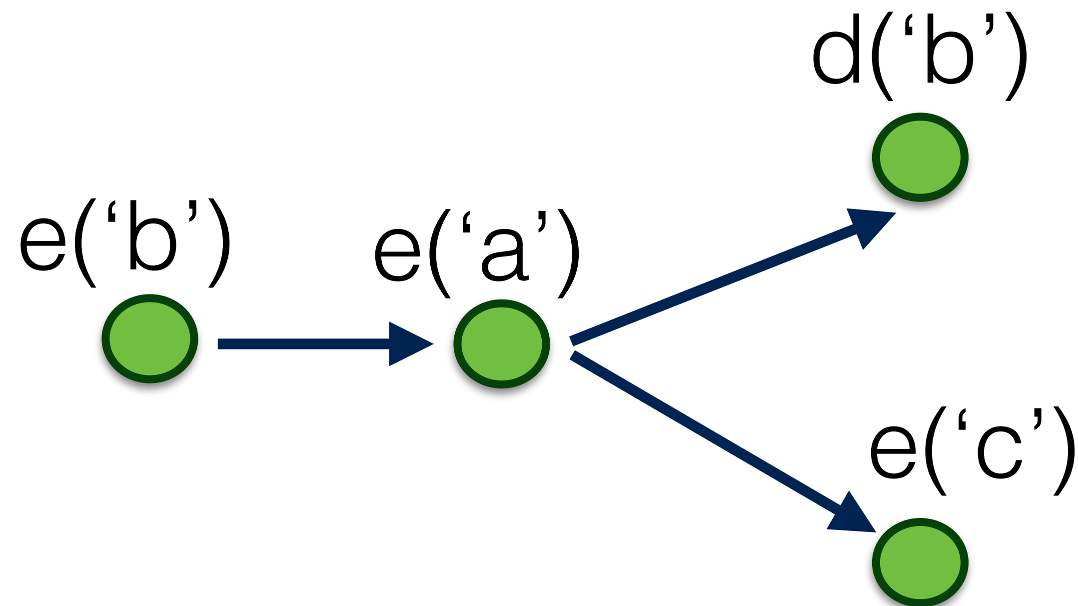
1. e('b') e('a') d('b') e('c')
2. e('b') e('a') e('c') d('b')
3. e('a') e('b') d('b') e('c')
4. e('a') e('b') e('c') d('b')



Commitment points

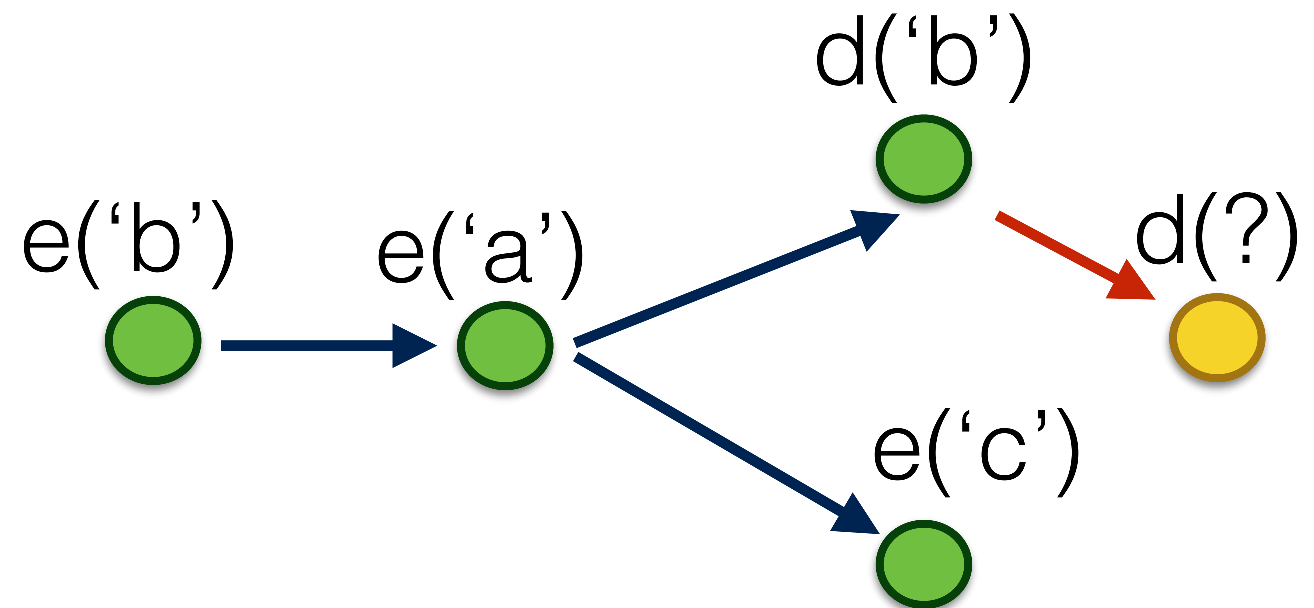
- The late choice is resolved by a dequeue with the FIFO policy in mind

1. e(‘b’) e(‘a’) d(‘b’) e(‘c’)
2. e(‘b’) e(‘a’) e(‘c’) d(‘b’)



Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind



Dequeue’s commitment point

```
if (there is a candidate enq) {
   res := try removing enq using CAS;
   if (res != FAIL) {
     E(this_event).rval := res;
     R := (R
      U {(enq, this_event)}
      U {(enq, e’) | the value of e’ is in queue}
      U {(deq, d’} | d’ is an uncompleted dequeue})+;
  }
}
```

Proof obligations

- For each operation, come up with commitment points:
  - adding a new event and real-time order edges
  - by the end of operation, assign a return value
  - (optionally) extend the order
    - preserving "all linearizations meet seq. spec."
    - preserving acyclicity of the order

Proof obligations

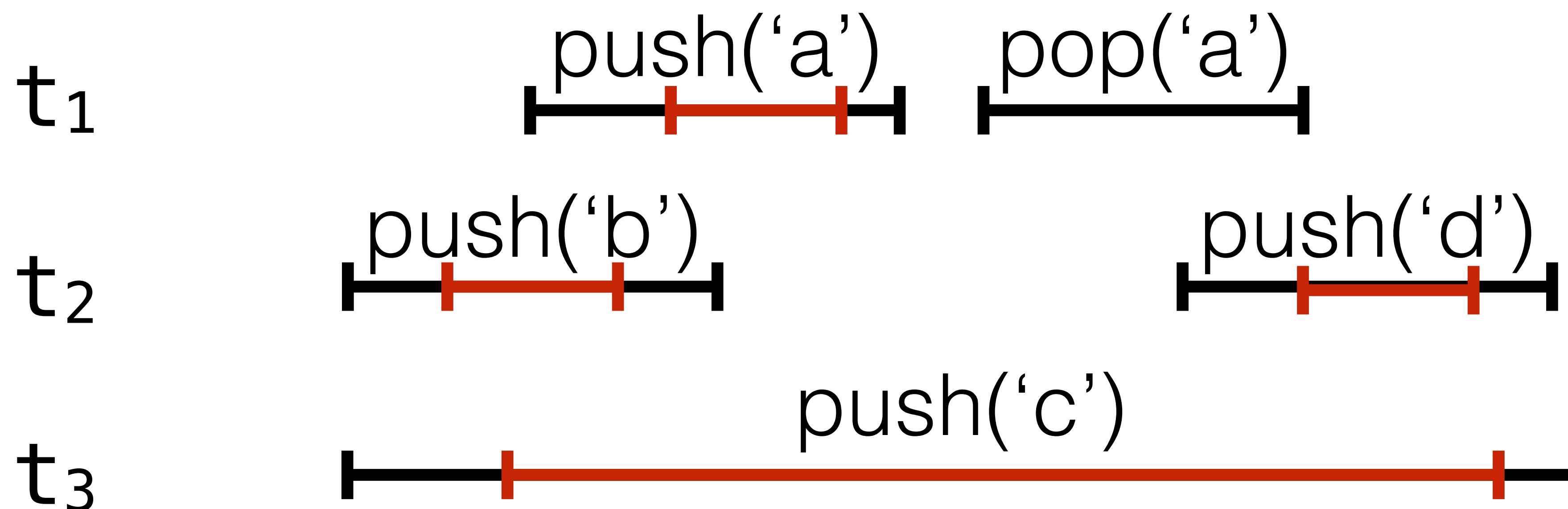
- for each **t** and **op** find $\mathbf{R_t}$, $\mathbf{G_t}$ and **INV**:
  - $R_t, G_t \vdash_t \{INV \wedge started(t, op)\}\ op\ \{INV \wedge finished(t, op)\}$
  - INV ==> all linearizations satisfy the sequential spec.
  - $\forall T.\ stable(INV, R_T)$
  - $\forall T.\ G_t \subseteq R_T$

Summary

- The technique for proving linearizability of algorithms that are challenging with the linearization points method
- Examples:
  - the TS stack (fiasco)
  - the TS queue (this talk),
  - the Herlihy-Wing queue (similar proof),
  - the Optimistic Set (alternative to the Hindsight proof),

TS Stack

a counter-example:



- At the commitment point of pop('a'):
  - need to prevent push('c') from occuring in between
  - but also allow it to happen both before and after