Proving Linearizability Using Partial Orders

Artem Khyzha    Mike Dodds
Alexey Gotsman    Matthew Parkinson

IMDEA Software Seminar Series  Nov 3rd 2016
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 be simulated by their sequential specification

\[
\text{data structure implementation} \subseteq \text{data structure specification}
\]

- History = a well-formed sequence of operation invocations and responses
- Formalised by “linearizability”
**Dummy Queue**

```c
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**

```c
Sequence S;

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

for every concrete history

\[ t_1 \quad e(\text{`a'}) \quad d(\text{`?}) \]

\[ t_2 \quad e(\text{`b'}) \quad e(\text{`d'}) \]

\[ t_3 \quad e(\text{`c'}) \]

find matching behaviour of the seq. spec.
Linearizability

for every concrete history

find matching behaviour of the seq. spec.
for every concrete history

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

$t_1$ e('a') d('b')

$t_2$ e('b')

$t_3$ e('c')

find a linearization

e('b') e('a') e('c') d(?) e('d')

Standard proof technique: linearization points
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();
    }
}

Sequence S;

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

for every concrete history

build a linearization:  <empty sequence>
Forward simulation

for every concrete history

t_1 \quad e('a') \quad d('b')

t_2 \quad e('b')

t_3 \quad e('c') \quad e('d')

build a linearization: e('b')
Forward simulation

for every concrete history

build a linearization: $e('b'), e('a')$
Forward simulation

for every concrete history

\[ t_1 \quad e('a') \quad d('b') \]
\[ t_2 \quad e('b') \quad e('d') \]
\[ t_3 \quad e('c') \]

build a linearization: \[ e('b') \quad e('a') \quad e('c') \]
Forward simulation

for every concrete history

$t_1$ $t_2$ $t_3$

e('a') $d('b')$
e('b')
e('c')
e('d')

current step

build a linearization: $e('b')$, $e('a')$, $e('c')$, $d('b')$, $d('b')$
Forward simulation

for every concrete history

develop diagrams:

\[ t_1 \quad e('a') \quad d('b') \]
\[ t_2 \quad e('b') \]
\[ t_3 \quad e('c') \quad e('d') \]

current step

build a linearization:

\[ e('b'), e('a'), e('c'), d('b'), e('d') \]
Problem

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

for every concrete history

\[
\begin{align*}
t_1 & \quad e('a') \quad d(?) \\
t_2 & \quad e('b') \quad e('c') \\
t_3 & \quad e('d') \\
\end{align*}
\]

find a linearization

\[
\begin{align*}
e('b') & \quad e('a') \quad e('c') \quad d(?) \quad e('d') \\
\ast & \\
\ast & \\
? & = 'b'
\end{align*}
\]
The TS queue

- each thread has its own pool
- pool is an abstract sequence
- (single producer multiple consumers)
enqueue(Val v) {
    ts := newTimestamp();
    insert(this_thread, v, ts);
}
timestamps = intervals

(a, b) < (c, d) iff b < c

enqueue(Val v) {
    ts := newTimestamp();
    insert(this_thread, v, ts);
}
The TS queue

\[
\begin{align*}
\text{int } & \text{ counter } = 1; \\
\text{TS } & \text{ newTimestamp() } \\
\text{ int } & \text{ l := counter; } \\
\text{ if } & \text{ CAS(counter, l, l+1) } \\
\text{ r := } & \text{ l; } \\
\text{ else } & \text{ return (l, r); }
\end{align*}
\]
The TS queue

t₁’s pool  t₂’s pool  t₃’s pool

'a' (1,1)  'b' (1,1)  'c' (2,2)
 'd' (3,3)

t₁  e('a')  d(?)
 t₂  e('b')
 t₃  e('c')

current step
The TS queue

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

'a' (1,1)  'b' (1,1)

t_1  e('a')  d(?)
t_2  e('b')
t_3  e('c')

current step
The TS queue

$\text{t}_1$'s pool $\Rightarrow \text{t}_2$'s pool $\Rightarrow \text{t}_3$'s pool

\begin{align*}
\text{'a'} & \hspace{1cm} (1,1) \\
\text{'b'} & \hspace{1cm} (1,1) \\
\text{'c'} & \hspace{1cm} (2,2)
\end{align*}

t_1 \quad \text{e('a')} \quad \text{d(?)}

t_2 \quad \text{e('b')} 

t_3 \quad \text{e('c')} 

\text{current step}
The TS queue

t_1's pool

't_1's pool

't_2's pool

't_3's pool

't_1's pool

't_2's pool

't_3's pool

The current step
The TS queue

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

by choosing the smallest timestamp
The TS queue

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

- the connection with the real-time order
- \( e_1 \xrightarrow{rt} e_2 \implies \text{timestamp}(e_1) < \text{timestamp}(e_2) \)
by the 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 is implemented in a program logic

✓ 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 (instead of a linearization)

• 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 (instead of a linearization)

1. the abstract history extends the real-time order

2. all linearizations meet the sequential spec
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.

\[
\text{e('a')} \quad \text{d('b')} \\
\text{e('b')} \\
\text{e('c')}
\]
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.

- Events get completed by the end of the operations.
Enqueue’s commitment point

```c
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.

\[ \begin{align*}
\text{t}_1 & \quad e('a') & \quad d('b') \\
\text{t}_2 & \quad e('b') \\
\text{t}_3 & \quad e('c') \\
\end{align*} \]
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 \texttt{enq}) {
  res := try removing \texttt{enq};

  if (res \neq 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
Conclusions

• The technique for proving linearizability of algorithms that are challenging with the linearization points method

• Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set

• Examples to do: the TS stack