Semantics for Concurrency

Ana Sokolova

UNIVERSITY of SALZBURG

MOVEP, 18.7.2018
• Part I: Concurrent data structures correctness and performance

• Part II: Order extension results for verifying linearizability
Concurrent Data Structures
Correctness and Relaxations
Data structures

• Queue FIFO
  
  enq  f  e  d  c  b  a  deq

• Stack LIFO
  
  push x y z  pop

• Pool unordered
  
  ins k  n  l  rem
  j  m  o
Concurrent data structures

- Queue FIFO

- Stack LIFO

- Pool unordered
Semantics of concurrent data structures

- **Sequential specification** = set of legal sequences

- **Consistency condition** = e.g. linearizability / sequential consistency

```plaintext
| t1: | enq(2) | deq(1) |
| t2: | enq(1) | deq(2) |
```

Examples:
- e.g. queue legal sequence: `enq(1)enq(2)deq(1)deq(2)`
- e.g. the concurrent history above is a linearizable queue concurrent history
Consistency conditions

A history is ... wrt a sequential specification iff

Linearizability [Herlihy, Wing '90]

Sequential Consistency [Lamport '79]

there exists a legal sequence that preserves precedence order

there exists a legal sequence that preserves per-thread precedence (program order)

there exists a legal sequence that preserves per-thread precedence (program order)

There have been a legal sequence that preserves precedence order

Linearizability [Herlihy, Wing '90]

Sequential Consistency [Lamport '79]

there exists a legal sequence that preserves precedence order

there exists a legal sequence that preserves per-thread precedence (program order)
Performance and scalability

throughput

# of threads / cores

Ana Sokolova

MOVEP 18.7.18
Relaxations allow trading correctness for performance.
Goal

Stack - incorrect behavior

push(a)push(b)push(c)pop(a)pop(b)

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the error from correct behaviour

correct in a relaxed stack... 2-relaxed? 3-relaxed?
How can relaxing help?

Stack

- top
- c
- b
- a

thread 1
thread 2
...
thread n

k-Relaxed stack

- top
- c
- b

k

thread 1
thread 2
...
thread n

Ana Sokolova
What we have

- Framework
- Generic examples
- Concrete relaxation examples
- Efficient concurrent implementations

for semantic relaxations

out-of-order / stuttering

stacks, queues, priority queues,.. / CAS, shared counter

of relaxation instances
The big picture

$S \subseteq \Sigma^*$

sequential specification
legal sequences

$\Sigma$ - methods with arguments
The big picture

\[ S \subseteq \Sigma^* \]

\[ S_k \subseteq \Sigma^* \]

\[ \Sigma - \text{methods with arguments} \]

\[ \text{sequential specification} \]

\[ \text{legal sequences} \]

\[ \text{relaxed sequential specification} \]

\[ \text{sequences at distance up to } k \text{ from } S \]
Relaxing the Semantics

- **Sequential specification** = set of legal sequences
- **Consistency condition** = e.g. linearizability / sequential consistency

Quantitative relaxations
Henzinger, Kirsch, Payer, Sezgin,S. POPL13

Local linearizability
Haas, Henzinger, Holzer,…, S, Veith CONCUR16
Syntactic distances do not help

$\text{push}(a)[\text{push}(i)\text{pop}(i)]^n\text{push}(b)[\text{push}(j)\text{pop}(j)]^m\text{pop}(a)$

is a 1-out-of-order stack sequence

its permutation distance is $\min(2n,2m)$
Semantic distances need a notion of state

- States are equivalence classes of sequences in $S$

\[ x \equiv y \iff \forall u \in \Sigma^*. (xu \in S \iff yu \in S) \]

- Two sequences in $S$ are equivalent iff they have an indistinguishable future

Example: for stack

\[
\text{push(a)push(b)pop(b)push(c)} \equiv \text{push(a)push(c)}
\]
Semantics goes operational

\( S \subseteq \Sigma^* \) is the sequential specification

\[ \text{LTS}(S) = (S/\equiv, \Sigma, \rightarrow, \left[ \varepsilon \right]_{\equiv}) \text{ with} \]

- states
- labels
- initial state

transition relation

\[ [s]_{\equiv} \xrightarrow{m} [sm]_{\equiv} \iff sm \in S \]
The relaxation framework

- Start from $\text{LTS}(S)$
- Add transitions with transition costs
- Fix a path cost function

$\text{distance} = \text{minimal cost on all paths labelled by the sequence}$
Generic out-of-order

\[ \text{segment\_cost}(q \overset{m}{\rightarrow} q') = |v| \]

Where \(v\) is a sequence of minimal length s.t.

- removing \(v\) enables a transition
- or
- inserting \(v\) enables a transition

 goes with different path costs
Out-of-order stack

- Canonical representative of a state
- Add incorrect transitions with segment-costs

Sequence of **push**'s with no matching **pop**

- Possible path cost functions $\text{max}$, $\text{sum}$, ...

Also more advanced
Relaxing the Consistency Condition

Local Linearizability (CONCUR16)
Local Linearizability main idea

- **Partition** a history into a set of local histories
- **Require** linearizability per local history

Already present in some shared-memory consistency conditions (not in our form of choice)

Local sequential consistency… is also possible

no global witness
Local Linearizability (queue) example

- t1: `enq(1)` → `deq(2)`
- t2: `enq(2)` → `deq(1)`

- t2-induced history, linearizable
- t1-induced history, linearizable

- (sequential) history not linearizable

- locally linearizable
Local Linearizability (queue) definition

Queue signature $\Sigma = \{\text{enq}(x) \mid x \in V\} \cup \{\text{deq}(x) \mid x \in V\} \cup \{\text{deq}(\text{empty})\}$

For a history $h$ with a thread $T$, we put

$$I_T = \{\text{enq}(x)^T \in h \mid x \in V\}$$
$$O_T = \{\text{deq}(x)^T \in h \mid \text{enq}(x)^T \in I_T\} \cup \{\text{deq}(\text{empty})\}$$

$h$ is locally linearizable iff every thread-induced history $h_T = h \mid (I_T \cup O_T)$ is linearizable.
Local Linearizability for Container-Type DS

Signature $\Sigma = \text{Ins} \cup \text{Rem} \cup \text{SOb} \cup \text{DOb}$

For a history $h$ with a thread $T$, we put

$$I_T = \{m^T \in h \mid m \in \text{Ins}\}$$

$$O_T = \{m(a) \in h \cap \text{Rem} \mid i(a)^T \in I_T\} \cup \{m(e) \mid e \in \text{Emp}\}$$
$$\cup \{m(a) \in h \cap \text{DOb} \mid i(a)^T \in I_T\}$$

$h$ is locally linearizable iff every thread-induced history $h_T = h \mid (I_T \cup O_T)$ is linearizable.
Generalizations of Local Linearizability

Signature $\Sigma$

For a history $h$ with $n$ threads, choose

- $\text{In}_h(i)$
- $\text{Out}_h(i)$

by increasing the in-methods, LL gradually moves to linearizability

- in-methods of thread $i$, methods that go in $h_i$
- out-methods of thread $i$, dependent methods on the methods in $\text{In}_h(i)$ (performed by any thread)

$h$ is locally linearizable iff every thread-induced history $h_i = h | (\text{In}_h(i) \cup \text{Out}_h(i))$ is linearizable.
Where do we stand?

In general

Local Linearizability

Linearizability

Sequential Consistency
Where do we stand?

For queues (and most container-type data structures)

Local Linearizability

Linearizability

Sequential Consistency
Properties

Local linearizability is compositional

$\mathbf{h}$ (over multiple objects) is locally linearizable

iff

each per-object subhistory of $\mathbf{h}$ is locally linearizable

Local linearizability is modular / “decompositional”

may allow for modular verification

like linearizability

unlike sequential consistency

uses decomposition into smaller histories, by definition
Verification (queue)

Queue sequential specification (axiomatic)

\[ s \text{ is a legal queue sequence} \iff
\begin{align*}
1. & \ s \text{ is a legal pool sequence, and} \\
2. & \ enq(x) <_s enq(y) \land deq(y) \in s \implies deq(x) \in s \land deq(x) <_s deq(y)
\end{align*} \]

Queue linearizability (axiomatic)

\[ h \text{ is queue linearizable} \iff
\begin{align*}
1. & \ h \text{ is pool linearizable, and} \\
2. & \ enq(x) <_h enq(y) \land deq(y) \in h \implies deq(x) \in h \land deq(y) <_h deq(x)
\end{align*} \]

Henzinger, Sezgin, Vafeiadis

CONCUR13

precedence order
Verification (queue)

Queue sequential specification (axiomatic)

\( s \) is a legal queue sequence
iff
1. \( s \) is a legal pool sequence, and
2. \( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) \in s \Rightarrow \text{deq}(x) \in s \land \text{deq}(x) <_s \text{deq}(y) \)

Queue local linearizability (axiomatic)

\( h \) is queue locally linearizable
iff
1. \( h \) is pool locally linearizable, and
2. \( \text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \Rightarrow \text{deq}(x) \in h \land \text{deq}(y) <_h \text{deq}(x) \)

thread-local precedence order
Relaxations lead to scalable implementations

e.g. k-FIFO, k-Stack

locally linearizable distributed implementation

k-out-of-order queue

local inserts / global removes

Ana Sokolova

MOVEP 18.7.18
Relaxations lead to scalable implementations

e.g. k-FIFO, k-Stack

CAS-based algorithm...

1: loop:
2:   read consistent state
3:   try to add/remove an item (*)
4:   if successful:
5:     return
6:   else:
7:     try to repair the stack
8:     goto loop (retry)
Relaxations lead to scalable implementations

e.g. k-FIFO, k-Stack

locally linearizable distributed implementation

k-out-of-order queue

local inserts / global removes

LLD $\Phi$
LL+$D$ $\Phi$
Performance

(a) Queues, LL queues, and “queue-like” pools

LL+D MS queue performs significantly better than MS queue
Performance

- LLD $\Phi$ performs significantly better than $\Phi$

(a) Queues, LL queues, and “queue-like” pools
Performance

![Performance Graph]

LL+D MS queue performs better than the best known pools

(a) Queues, LL queues, and “queue-like” pools
Linearizability via Order Extension Theorems

joint work with

foundational results for verifying linearizability

Harald Woracek
Queue sequential specification (axiomatic)

**s** is a legal queue sequence

iff

1. **s** is a legal pool sequence, and
2. \(\text{enq}(x) <_{s} \text{enq}(y) \land \text{deq}(y) \in s\) \(\Rightarrow\) \(\text{deq}(x) \in s \land \text{deq}(x) <_{s} \text{deq}(y)\)

Queue linearizability (axiomatic)

**h** is queue linearizable

iff

1. **h** is pool linearizable, and
2. \(\text{enq}(x) <_{h} \text{enq}(y) \land \text{deq}(y) \in h\) \(\Rightarrow\) \(\text{deq}(x) \in h \land \text{deq}(y) <_{h} \text{deq}(x)\)

Henzinger, Sezgin, Vafeiadis CONCUR13

As well as
Reducing Linearizability to State Reachability
[Bouajjani, Emmi, Enea, Hamza]
ICALP15 + …

precedence order
Problems (stack)

Stack sequential specification (axiomatic)

\[ s \text{ is a legal stack sequence} \]
\[ \text{iff} \]
1. \( s \) is a legal pool sequence, and
2. \( \text{push}(x) <_s \text{push}(y) <_s \text{pop}(x) \Rightarrow \text{pop}(y) \in s \land \text{pop}(y) <_s \text{pop}(x) \)

Stack linearizability (axiomatic)

\[ h \text{ is stack linearizable} \]
\[ \text{iff} \]
1. \( h \) is pool linearizable, and
2. \( \text{push}(x) <_h \text{push}(y) <_h \text{pop}(x) \Rightarrow \text{pop}(y) \in h \land \text{pop}(x) <_h \text{pop}(y) \)
Problems (stack)

Stack sequential specification (axiomatic)

\[ s \text{ is a legal stack sequence } \iff \]
1. \( s \text{ is a legal pool sequence, and } \)
2. \( \text{push}(x) <_s \text{push}(y) <_s \text{pop}(x) \implies \text{pop}(y) \in s \land \text{pop}(y) <_s \text{pop}(x) \)

Stack linearizability (axiomatic)

\[ h \text{ is stack linearizable } \iff \]
1. \( h \text{ is pool linearizable, and } \)
2. \( \text{push}(x) <_h \text{push}(y) <_h \text{pop}(x) \implies \text{pop}(y) \in h \land \text{pop}(x) \not\in_h \text{pop}(y) \)
Problems (stack)

Stack linearizability (axiomatic)

\( \mathbf{h} \) is stack linearizable
iff
1. \( \mathbf{h} \) is pool linearizable, and
2. \( \text{push}(x) <_\mathbf{h} \text{push}(y) <_\mathbf{h} \text{pop}(x) \implies \text{pop}(y) \in \mathbf{h} \land \text{pop}(x) \not<_\mathbf{h} \text{pop}(y) \)

Ana Sokolova

MOVEP 18.7.18
Linearizability verification

Data structure
- signature $\Sigma$ - set of method calls including data values
- sequential specification $S \subseteq \Sigma^*$, prefix closed

Sequential specification via violations
Extract a set of violations $V$, relations on $\Sigma$, such that $s \in S$ iff $s$ has no violations $\mathcal{P}(s) \cap V = \emptyset$

Linearizability verification
- it is easy to find a large $CV$, but difficult to find a small representative
- we build $CV$ iteratively from $V$
- $\mathcal{P}(s) \cap V = \emptyset$
- concurrent history
- legal sequence
- find a set of violations $CV$ such that: every interval order with no $CV$ violations extends to a total order with no $V$ violations.
Pool without empty removals

Pool sequential specification (axiomatic)

\(s\) is a legal pool (without empty removals) sequence 
iff
1. \(\text{rem}(x) \in s \Rightarrow \text{ins}(x) \in s \land \text{ins}(x) <_s \text{rem}(x)\)

Pool linearizability (axiomatic)

\(h\) is pool (without empty removals) linearizable 
iff
1. \(\text{rem}(x) \in h \Rightarrow \text{ins}(x) \in h \land \text{rem}(x) \prec_h \text{ins}(x)\)
Queue without empty removals

Queue sequential specification (axiomatic)

\( s \) is a legal queue (without empty removals) sequence iff

1. \( \text{deq}(x) \in s \implies \text{enq}(x) \in s \land \text{enq}(x) <_s \text{deq}(x) \)
2. \( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) \in s \implies \text{deq}(x) \in s \land \text{deq}(x) <_s \text{deq}(y) \)

Queue linearizability (axiomatic)

\( h \) is queue (without empty removals) linearizable iff

1. \( \text{rem}(x) \in h \implies \text{ins}(x) \in h \land \text{rem}(x) <_h \text{ins}(x) \)
2. \( \text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \implies \text{deq}(x) \in h \land \text{deq}(y) <_h \text{deq}(x) \)

V violations
\( \text{deq}(x) <_s \text{enq}(x) \) and
\( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) <_s \text{deq}(x) \)

CV violations
= V violations

Ana Sokolova
MOVEP 18.7.18
Pool sequential specification (axiomatic)

\( s \) is a legal pool (with empty removals) sequence iff
1. \( \text{rem}(x) \in s \Rightarrow \text{ins}(x) \in s \land \text{ins}(x) <_s \text{rem}(x) \)
2. \( \text{rem}(\bot) <_s \text{rem}(x) \Rightarrow \text{rem}(\bot) <_s \text{ins}(x) \land \text{ins}(x) <_s \text{rem}(\bot) \Rightarrow \text{rem}(x) <_s \text{rem}(\bot) \)

Pool linearizability (axiomatic)

\( h \) is pool (with empty removals) linearizable iff
1. \( \text{rem}(x) \in h \Rightarrow \text{ins}(x) \in h \land \text{rem}(x) <_h \text{ins}(x) \)
2. \( \ldots \)

infinitely many CV violations

\( \text{ins}(x_1) <_h \text{rem}(\bot) \land \text{ins}(x_2) <_h \text{rem}(x_1) \land \ldots \land \text{ins}(x_{n+1}) <_h \text{rem}(x_n) \land \text{rem}(\bot) <_h \text{rem}(x_{n+1}) \)
**Queue sequential specification (axiomatic)**

\( s \) is a legal queue (with empty removals) sequence iff

1. \( \text{deq}(x) \in s \Rightarrow \text{enq}(x) \in s \land \text{enq}(x) \prec_s \text{deq}(x) \)

2. \( \text{deq}(\perp) \prec_s \text{deq}(x) \Rightarrow \text{deq}(\perp) \prec_s \text{enq}(x) \land \text{enq}(x) \prec_s \text{deq}(\perp) \Rightarrow \text{deq}(x) \prec_s \text{deq}(\perp) \)

3. \( \text{enq}(x) \prec_s \text{enq}(y) \land \text{deq}(y) \in s \Rightarrow \text{deq}(x) \in s \land \text{deq}(x) \prec_s \text{deq}(y) \)

**Queue linearizability (axiomatic)**

\( h \) is queue (with empty removals) linearizable iff

1. \( \text{deq}(x) \in h \Rightarrow \text{enq}(x) \in h \land \text{deq}(x) \prec_h \text{enq}(x) \)

2. \( \text{enq}(x_1) \prec_h \text{deq}(\perp) \land \text{enq}(x_2) \prec_h \text{deq}(x_1) \land \ldots \land \text{enq}(x_{n+1}) \prec_h \text{deq}(x_n) \land \text{deq}(\perp) \prec_h \text{deq}(x_{n+1}) \)

3. \( \text{enq}(x) \prec_h \text{enq}(y) \land \text{deq}(y) \in h \Rightarrow \text{deq}(x) \in h \land \text{deq}(y) \prec_h \text{deq}(x) \)
Concurrent Queues

Data independence => verifying executions where each value is enqueued at most once is sound

Reduction to assertion checking = exclusion of "bad patterns"

Value v dequeued without being enqueued
deq => v

Value v dequeued before being enqueued
deq => v  enq(v)

Value v dequeued twice
deq => v  deq => v

Value v1 and v2 dequeued in the wrong order
enq(v1)  enq(v2)  deq = v2  deq = v1

Dequeue wrongfully returns empty
deq => empty

enq(v1)  deq => v1

enq(v2)  deq => v2

enq(v_m-2)  deq => v_{m-2}

enq(v_n)  deq => v_n...
It works for

- Pool without empty removals
- Queue without empty removals
- Priority queue without empty removals
- Pool
- Queue
- Priority queue

But not yet for Stack: infinite CV violations without clear inductive structure

Exploring the space of data structures as well as new ideas for problematic cases
How does it work?
The basics

\[ PO[\mathcal{V}] = \{ R \in PO \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset \} \]
\[ IO[\mathcal{V}] = \{ R \in IO \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset \} \]
\[ TO[\mathcal{V}] = \{ R \in TO \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset \} \]

\[ \forall (a, b), (c, d) \in R. (a, d) \in R \lor (c, b) \in R \]
The problem

Given a set of violations $\mathcal{V}$, find a “small” set of violations $\mathcal{V}'$ such that

$$\forall R \in IO[\mathcal{V}']. \exists \overline{R} \in TO[\mathcal{V}]. \overline{R} \supseteq R$$

Theorem (singleton violations)

Let $\mathcal{V}$ consist only of singletons, and let $V = \bigcup \mathcal{V}$. If $V$ is transitive and not a cycle, then the problem is solved with $\mathcal{V}' = \mathcal{V}$. This solves the case of pool without empty removals.
The closures

$$\text{Clos}_O(\mathcal{V}) = \bigcap_{S \in O[\mathcal{V}]} \mathcal{P}(S)^c$$

O-closure of a set of violations

monotone, extensive, idempotent

Proposition

$$\forall R \in IO[\mathcal{V}']. \exists \overline{R} \in TO[\mathcal{V}]. \overline{R} \supseteq R$$

iff

$$\text{Clos}_{TO}(\mathcal{V}) = \text{Clos}_{IO}(\mathcal{V}')$$
The axioms

**Proposition PO**

\( \mathcal{V} \) is PO-closed iff

(C1) \( \mathcal{D} \subseteq \mathcal{V} \)

(C2) \( \forall N \in \mathcal{V}. \forall M. (N \subseteq \text{tr}(M) \Rightarrow M \in \mathcal{V}) \)

**Proposition IO**

\( \mathcal{V} \) is IO-closed iff (C1) and (C2) and

(C3) \( \forall M \in X \setminus \mathcal{E}. \forall a, b, c, d \in X. a \neq d \land c \neq b \Rightarrow \\
[ (a, b) \in M \land (c, d) \in M \land M \cup \{(a, d)\} \in \mathcal{V} \land M \cup \{(c, b)\} \in \mathcal{V} \Rightarrow M \in \mathcal{V} ] \)

**Proposition TO**

\( \mathcal{V} \) is TO-closed iff (C1) and (C2) and

(C4) \( \forall N \in \mathcal{V}, N \cap \Delta \neq \emptyset. \forall M \in X. \forall a \in N \setminus M. \exists a_1, a_2 \in X. \\
a = (a_1, a_2) \land M \cup \{(a_2, a_1)\} \in \mathcal{V} \Rightarrow M \in \mathcal{V} \)
How does it work?

Theorem

Let $\mathcal{V}$ consist only of finite sets and assume

1. $\forall N, M \in \mathcal{V}. \forall (a_1, a_2) \in N. |\{(b_1, b_2) \in M \mid a_2 = b_1\}| \leq 1$

then the problem is solved.

if we manage to construct such a set of violations, we are done

we provide an algorithm that produces a set of violations such that

holds

if we are lucky, (2) holds too
The algorithm

Take two violations \( N_1, N_2 \in \mathcal{V} \) and an element \( x \in X \) and produce a new violation
\[
\{(a, b) \mid (a, x) \in N_1, (x, b) \in N_2\}
\]
\[
\cup \{(a, b) \in N_1 \mid b \neq x\}
\]
\[
\cup \{(a, b) \in N_2 \mid a \neq x\}
\]

Take two violations \( N_1, N_2 \in \mathcal{V} \) and a pair \((x, y) \in X \times X\) and produce a new violation
\[
\{(a, y) \mid (a, x) \in N_2\}
\]
\[
\cup \{(x, b) \mid (y, b) \in N_2\}
\]
\[
\cup \{(a, b) \in N_2 \mid b \neq x \land a \neq y\}
\]
\[
\cup N_1 \setminus \{(x, y)\}
\]

until no new violations are produced
It works for

- Pool without empty removals
- Queue without empty removals
- Priority queue without empty removals
- Pool
- Queue
- Priority queue

But not yet for Stack: infinite CV violations without clear inductive structure

Exploring the space of data structures as well as new ideas for problematic cases
It works for

- Pool without empty removals
- Queue without empty removals
- Priority queue without empty removals
- Pool
- Queue
- Priority queue

But not yet for Stack: infinite CV violations without clear inductive structure

Exploring the space of data structures as well as new ideas for problematic cases

Thank You!