Yale

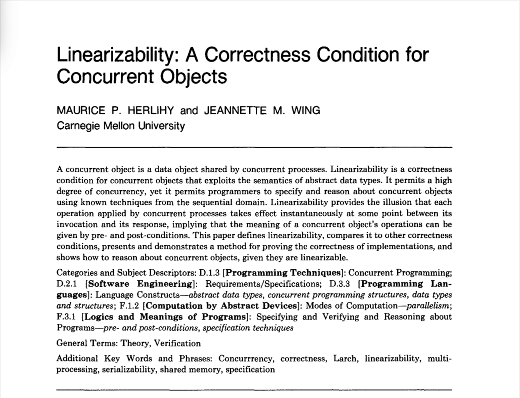
A Compositional Theory of Linearizability

Arthur Oliveira Vale   Zhong Shao   Yixuan Chen

Yale University, USA

January 22, 2023

Herlihy-Wing Linearizability



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

What is a concurrent object?

$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:
  deq
- Concurrent:

Herlihy-Wing Linearizability

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*, *concurrent programming structures*, *data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions*, *specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

What is a concurrent object?

$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:
  deq
- Concurrent:

Herlihy-Wing Linearizability

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

What is a concurrent object?

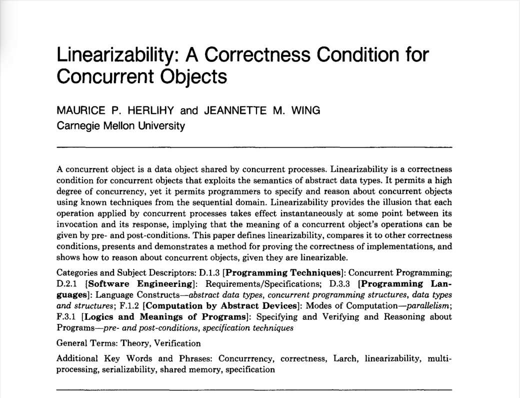
$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:

  $$\text{deq} \longrightarrow \bot$$

- Concurrent:

Herlihy-Wing Linearizability



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

α_0:deq

What is a concurrent object?

$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:
  $\text{deq} \longrightarrow \bot$
- Concurrent:

Herlihy-Wing Linearizability

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

$$\alpha_0\text{:deq} \longrightarrow \alpha_0\text{:}\bot$$

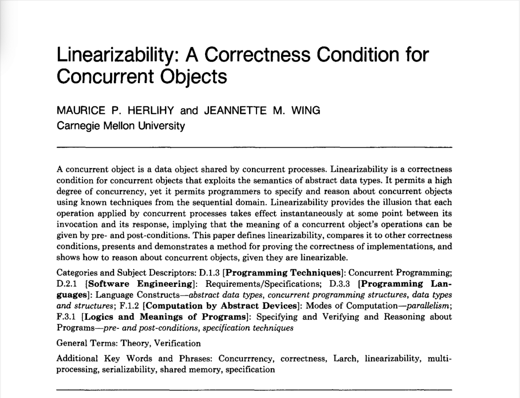
What is a concurrent object?

$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:

  $\text{deq} \longrightarrow \bot$
- Concurrent:

Herlihy-Wing Linearizability



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

What is a concurrent object?

$$\text{Queue} := \{\text{enq} : \mathbb{N} \to \{\text{ok}\}, \text{deq} : \mathbb{N} + \{\bot\}\}$$

- Sequential:
  $\text{deq} \longrightarrow \bot$
- Concurrent:

α_0:deq α_1:enq(1) α_2:enq(2) α_1:ok α_2:ok α_0:$\bot/1/2$

Classical Linearizability: Example

ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok

α_0:deq → α_0:1 → α_0:deq

α_1:enq(1) → α_1:ok

α_2:enq(2) → α_2:ok

α_0:deq → α_0:1 → α_0:deq

α_1:enq(1) → α_1:ok

α_2:enq(2) → α_2:ok

ν_{squeue} : α_1:enq(1) α_1:ok α_0:deq α_0:1 α_2:enq(2) α_2:ok

Classical Linearizability: Example

ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok

---

α_0:deq $\rightarrow$ α_0:1 $\rightarrow$ α_0:deq

α_1:enq(1) $\rightarrow$ α_1:ok

α_2:enq(2) $\rightarrow$ α_2:ok

$\rightsquigarrow$

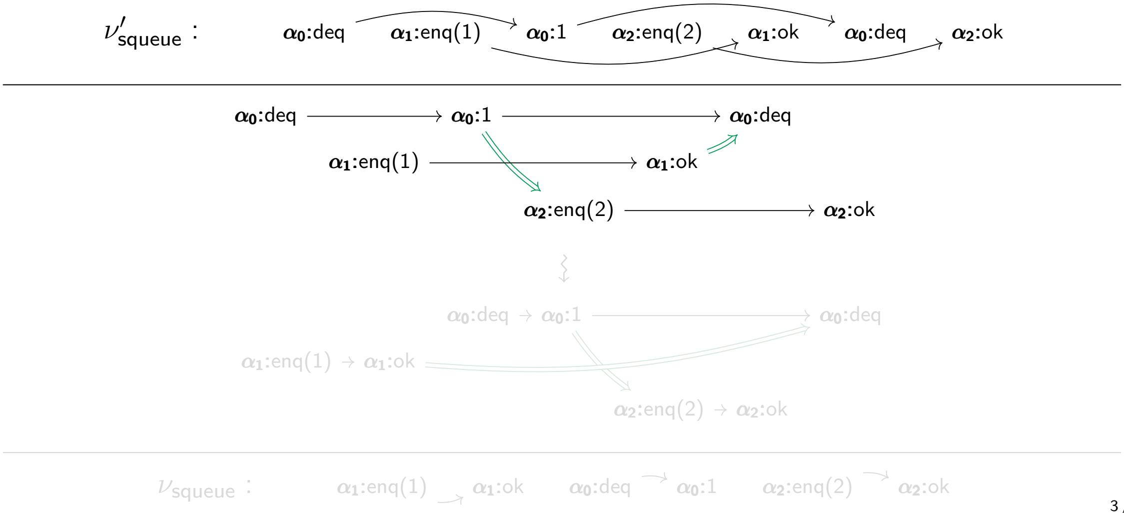
α_0:deq $\rightarrow$ α_0:1 $\rightarrow$ α_0:deq

α_1:enq(1) $\rightarrow$ α_1:ok

α_2:enq(2) $\rightarrow$ α_2:ok

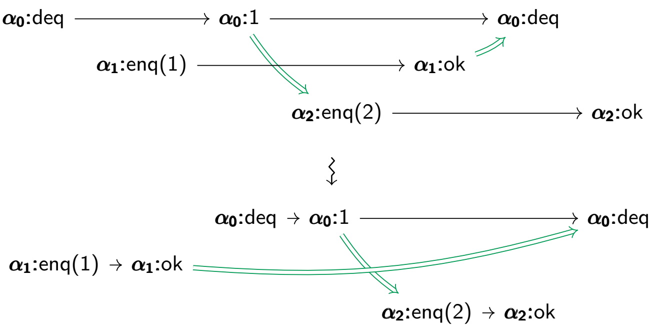
ν_{squeue} : α_1:enq(1) α_1:ok α_0:deq α_0:1 α_2:enq(2) α_2:ok

Classical Linearizability: Example



Classical Linearizability: Example

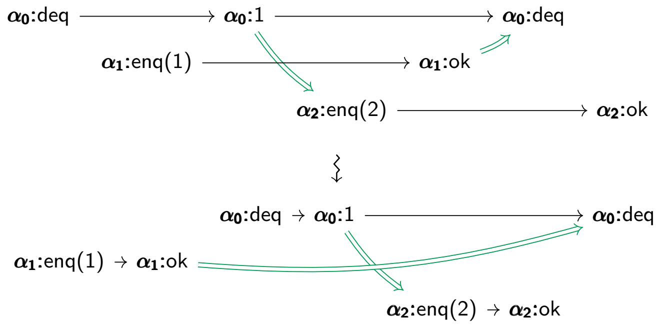
ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok



ν_{squeue} : α_1:enq(1) α_1:ok α_0:deq α_0:1 α_2:enq(2) α_2:ok

Classical Linearizability: Example

ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok

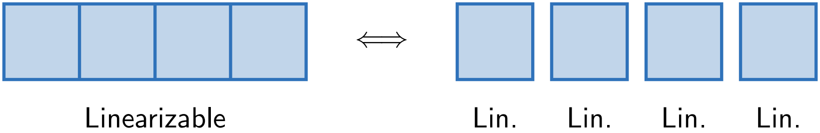


ν_{squeue} : α_1:enq(1) α_1:ok α_0:deq α_0:1 α_2:enq(2) α_2:ok

Locality [Herlihy and Wing, 1990]

PROPOSITION

H is linearizable if and only if, for each object x, $H \mid x$ is linearizable.



Equivalence with Contextual Refinement [Filipović et Al. 2010]

$\mathsf{Obj}_{\mathsf{Conc}}$ observationally refines ($\sqsubseteq$) $\mathsf{Obj}_{\mathsf{Atom}}$ when

$$\forall \text{ programs } P \,.\, \forall \text{ states } s \,.\, [\![P]\!](\mathsf{Obj}_{\mathsf{Conc}})(s) \subseteq [\![P]\!](\mathsf{Obj}_{\mathsf{Atom}})(s)$$

PROPOSITION

$$\mathsf{Obj}_{\mathsf{Conc}} \text{ linearizes to } \mathsf{Obj}_{\mathsf{Atom}} \iff \mathsf{Obj}_{\mathsf{Conc}} \text{ observationally refines } \mathsf{Obj}_{\mathsf{Atom}}$$

Why?

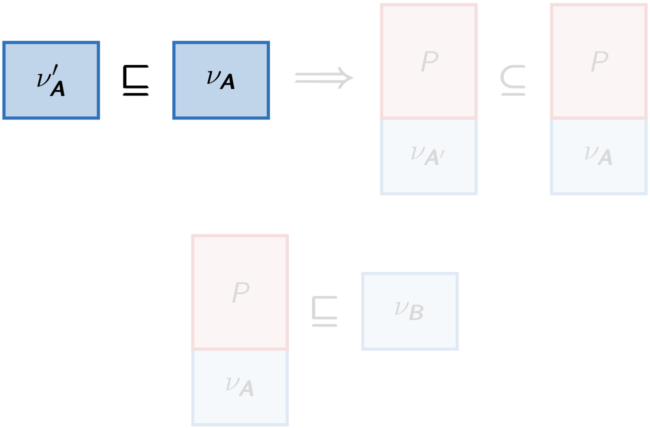
Where does linearizability come from and why does it work?

Key Contributions

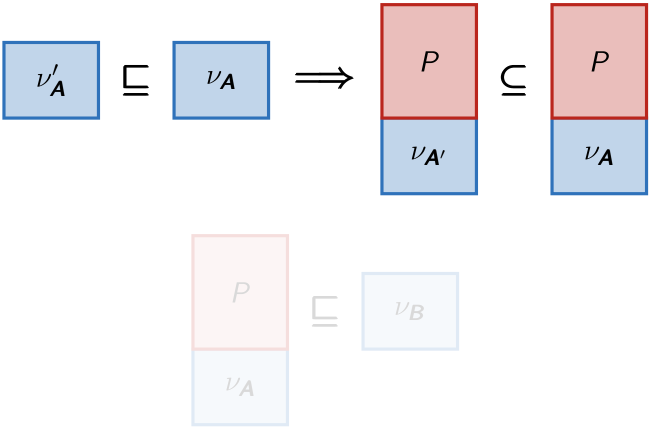
- A new generalized definition of linearizability not tied to atomicity.
- The first model of linearizability that supports refinement, horizontal and vertical composition.
- A general (category-theoretic) methodology for deriving linearizability from a model of concurrent computation.
- New simpler proofs of the locality and refinement properties.
- A new program logic that is sound for our formulation of linearizability.
- Applications to compositional verification.

Outline

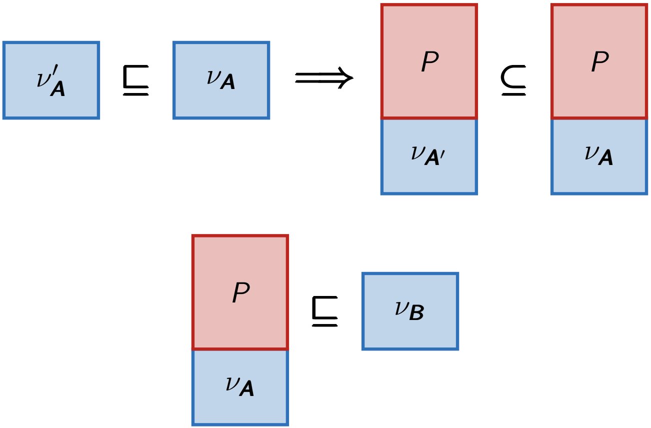
Typical Approach for Verifying Concurrent Objects



Typical Approach for Verifying Concurrent Objects



Typical Approach for Verifying Concurrent Objects

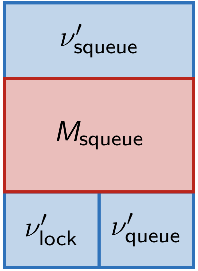


Implementating a Shared Queue

M_{squeue}:

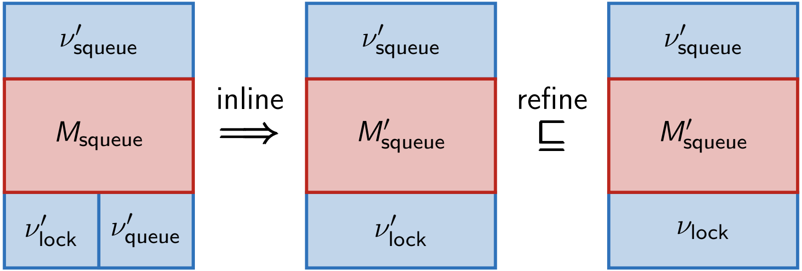
```
Import Q:Queue
Import L:Lock

enq(k) {                deq() {
  L.acq();                    L.acq();
  r <- Q.enq(k);              r <- Q.deq();
  L.rel();                    L.rel();
  ret r                     ret r
}                       }
```

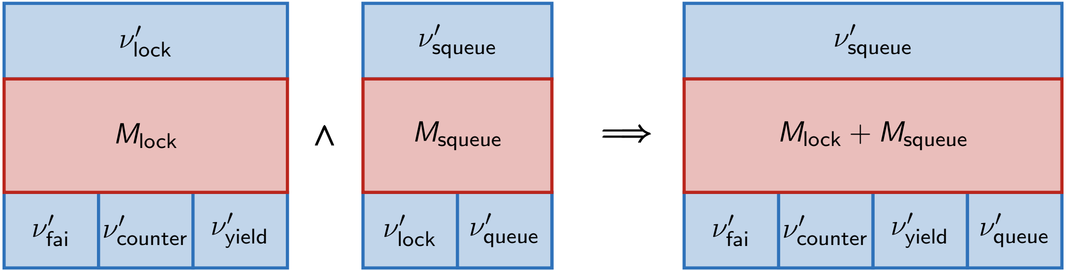


- No account of how locality interacts with refinement.
- Locality doesn't apply! The queue has a race (not linearizable).

Implementing a Shared Queue (Continued)

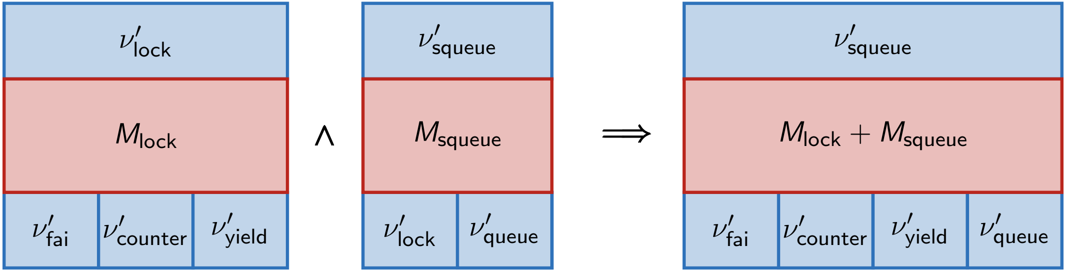


Vertical Composition



Inlining ? Syntactic Linking?

Vertical Composition



Inlining ? Syntactic Linking?

Compositionality

Linearizability	Refinement $- \subseteq -$
Locality $- \otimes -$	(Vertical) Composition $-;-$

Outline

Our Methodology

1. Base Model of Computation
   (A semicategory enriched with a notion of refinement)

2. Choose identity programs
   (Usually obvious)

3. Compute a Compositional Model out of (1) and (2)
   (The Karoubi Envelope)

4. Abstract Linearizability $\iff$ Concrete Linearizability

5. One Extra Axiom $\implies$ Refinement Property

6. Tensor Product + One Extra Axiom $\implies$ Locality

Game Semantics

Types correspond to **Games** A, B, C

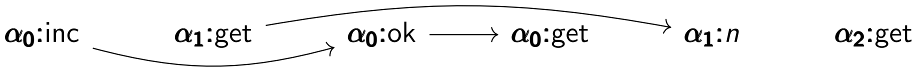
Programs correspond to **strategies** $\sigma : A \multimap B$ of the game $A \multimap B$

Object specifications correspond to strategies $\nu : 1 \multimap A$

Sequentially Consistent Computation

- We start by defining a sequential model of computation.
- A set of agent names $\alpha \in \Upsilon$.
- A concurrent game $\boldsymbol{A}$ is specified by the sequential game A that all agents play.
- A move looks like $\boldsymbol{\alpha}$:m where $\alpha \in \Upsilon$ and m is a move of A.
- The set of plays of $\boldsymbol{A}$ is the set of sequentially consistent interleavings of plays from A.
  Example:

$$\text{Counter} = \{\text{get} : \mathbb{N}, \text{inc} : \text{ok}\}$$



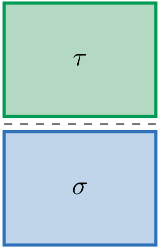
Vertical Composition

There is a composition operation defined per usual by

"Parallel composition + Hiding"

Denoted by

$$\sigma : \boldsymbol{A} \multimap \boldsymbol{B} \qquad \tau : \boldsymbol{B} \multimap \boldsymbol{C} \longmapsto \sigma ; \tau : \boldsymbol{A} \multimap \boldsymbol{C}$$



Which is **associative** ... but there is **no identity element**!

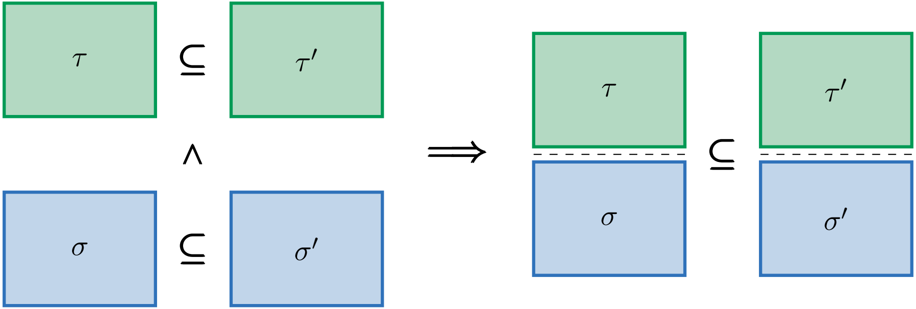
$$\forall \sigma : \boldsymbol{A} \multimap \boldsymbol{B} . \mathbf{id}_{\boldsymbol{A}} ; \sigma ; \mathbf{id}_{\boldsymbol{B}} = \sigma$$

In other words, concurrent games with concurrent strategies assembles into a semicategory

$$\underline{\mathbf{Game}}_{\mathbf{Conc}}$$

Refinement

Our model is enriched over a notion of refinement $\subseteq$ (behavior containment)



Sequential Copycat

```
Import Q:Queue

enq (n : ℕ) {
  r <- Q.enq(n);
  ret r
}

deq () {
  r <- Q.deq();
  ret r
}
```

The copycat strategy $\mathsf{copy}_A : A \multimap A$ behaves as the sequential identity

Concurrent Strategies

$\mathsf{ccopy}_{\boldsymbol{A}} := \|_{\alpha \in \Upsilon} \mathsf{copy}_A$

```
Import Q:Queue

enq (n : ℕ) {
  r <- Q.enq(n);
  ret r
}

deq () {
  r <- Q.deq();
  ret r
}
```

‖ ... ‖

```
Import Q:Queue

enq (n : ℕ) {
  r <- Q.enq(n);
  ret r
}

deq () {
  r <- Q.deq();
  ret r
}
```

Composition can lead to **emergent behavior**.

$\sigma \subseteq \sigma ; \mathsf{ccopy}_{\boldsymbol{B}}$

Concurrent Strategies

$\text{ccopy}_{\boldsymbol{A}} := \|_{\alpha \in \Upsilon} \text{copy}_A$

```
Import Q:Queue

enq (n : ℕ) {
  r <- Q.enq(n);
  ret r
}

deq () {
  r <- Q.deq();
  ret r
}
```

$\|$ $\cdots$ $\|$

```
Import Q:Queue

enq (n : ℕ) {
  r <- Q.enq(n);
  ret r
}

deq () {
  r <- Q.deq();
  ret r
}
```

Composition can lead to **emergent behavior**.

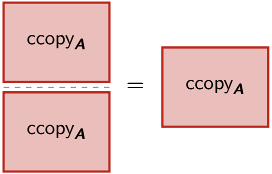
$$\sigma \subseteq \sigma; \text{ccopy}_{\boldsymbol{B}}$$

The Karoubi Envelope

PROPOSITION

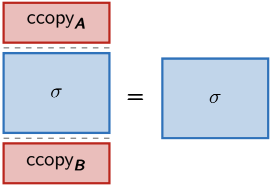
For all concurrent game $\mathbf{A}$ the strategy $\mathsf{ccopy}_{\mathbf{A}} : \mathbf{A} \multimap \mathbf{A}$ is idempotent, i.e.

$$\mathsf{ccopy}_{\mathbf{A}} ; \mathsf{ccopy}_{\mathbf{A}} = \mathsf{ccopy}_{\mathbf{A}}$$



Call a strategy $\sigma : \mathbf{A} \multimap \mathbf{B}$ **saturated** when

$$\mathsf{ccopy}_{\mathbf{A}} ; \sigma ; \mathsf{ccopy}_{\mathbf{B}} = \sigma$$



Composition of saturated strategies is associative and has as identity $\mathsf{ccopy}_{\_}$.
Call the resulting category of concurrent games and saturated strategies

$$\mathbf{Game}_{\mathbf{Conc}}$$

Two Models of Concurrent Computation

$\underline{\mathbf{Game}}_{\mathbf{Conc}}$	$\mathbf{Game}_{\mathbf{Conc}}$
Good for specification	Good for composition

We can convert between models:

$$\sigma : A \multimap B \in \underline{\mathbf{Game}}_{\mathbf{Conc}} \xmapsto{K_{\mathrm{Conc}}-} \mathrm{ccopy}_{A}; \sigma; \mathrm{ccopy}_{B} \in \mathbf{Game}_{\mathbf{Conc}}$$

Outline

Abstract Linearizability

DEFINITION (ABSTRACT LINEARIZABILITY)

We say

$$\nu'_A : A \in \mathbf{Game}_{\mathbf{Conc}}$$

linearizes to

$$\nu_A : A \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$$

when

$$\nu'_A \subseteq K_{\mathsf{Conc}}\ \nu_A$$

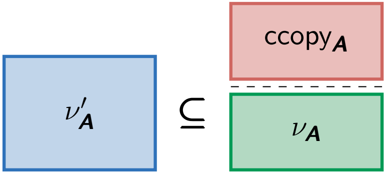
DEFINITION

A linearizable object consists of a pair

$$(\nu'_A : A \in \mathbf{Game}_{\mathbf{Conc}}, \nu_A : A \in \underline{\mathbf{Game}}_{\mathbf{Conc}})$$

such that

$$\nu'_A \subseteq K_{\mathsf{Conc}}\ \nu_A$$



ν'_A is the implementation and ν_A the specification

Rewrites

PROPOSITION (GHICA AND MURAWSKI, 2004)

$\sigma : \boldsymbol{A}$ is saturated if and only if $\forall t \in \sigma. \forall s \in P_{\boldsymbol{A}}. s \rightsquigarrow_{\boldsymbol{A}} t \implies s \in \sigma$

If $t \in \sigma$ and s is "more concurrent" than t then s is also in σ

Linearizability

DEFINITION

$s \in P_{\boldsymbol{A}}$ is linearizable to $t \in P_{\boldsymbol{A}}$ when there exists a sequence s_O of Opponent moves and a sequence s_P of Proponent moves such that

$$s \cdot s_P \rightsquigarrow_{\boldsymbol{A}} t \cdot s_O$$

- t need not be atomic (coincides with Herlihy-Wing when it is);
- s_P = returns;
- s_O = removed pending invocations (not all need be removed);
- $\rightsquigarrow_{\boldsymbol{A}}$ = happens-before order preservation.

Abstract Linearizability

PROPOSITION

Let $\tau : \boldsymbol{A} \in \underline{\textbf{Game}}_{\textbf{Conc}}$ then

$$K_{\mathsf{Conc}}\ \tau = \{s \in P_{\boldsymbol{A}} \mid \exists t \in \tau . s \text{ linearizes to } t\}$$

Abstract Linearizability

PROPOSITION

Let $\tau : \boldsymbol{A} \in \underline{\mathbf{Game}}_{\mathsf{Conc}}$ then

$$K_{\mathsf{Conc}}\ \tau = \{s \in P_{\boldsymbol{A}} \mid \exists t \in \tau . s \text{ linearizes to } t\}$$

COROLLARY

For $\sigma : \boldsymbol{A}$ and $\tau : \boldsymbol{A}$,

$$\sigma \text{ linearizes to } \tau \iff \sigma \subseteq K_{\mathsf{Conc}}\ \tau.$$

Abstract Linearizability

PROPOSITION

Let $\tau : \boldsymbol{A} \in \underline{\textbf{Game}}_{\textbf{Conc}}$ then

$$K_{\mathsf{Conc}}\ \tau = \{s \in P_{\boldsymbol{A}} \mid \exists t \in \tau . s \text{ linearizes to } t\}$$

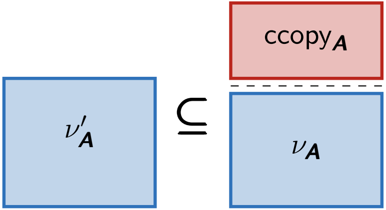
DEFINITION (ABSTRACT LINEARIZABILITY)

We say $\sigma : \boldsymbol{A} \in \textbf{Game}_{\textbf{Conc}}$ linearizes to $\tau : \boldsymbol{A} \in \underline{\textbf{Game}}_{\textbf{Conc}}$ when

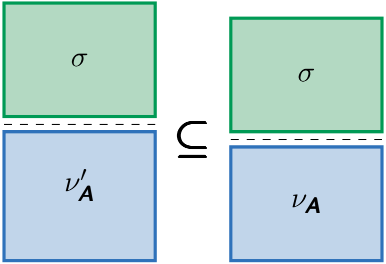
$$\sigma \subseteq K_{\mathsf{Conc}}\ \tau$$

Outline

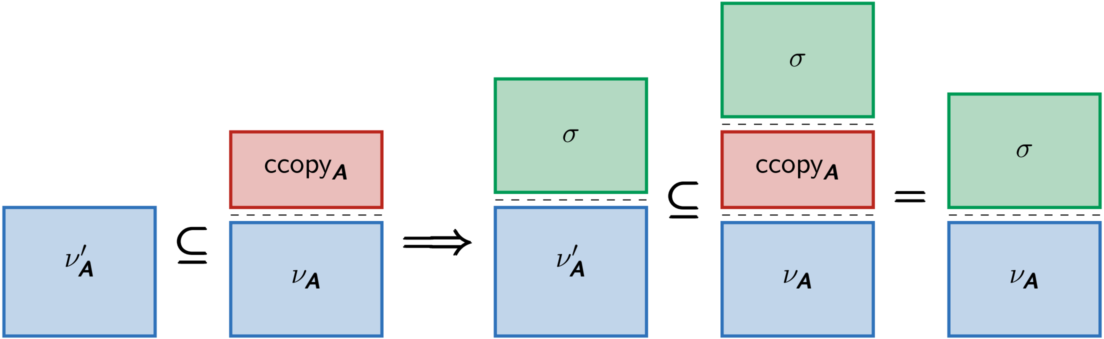
Interaction Refinement



$\iff \forall B. \forall \sigma : A \multimap B.$
$\in \mathbf{Game}_{\mathbf{Conc}}$

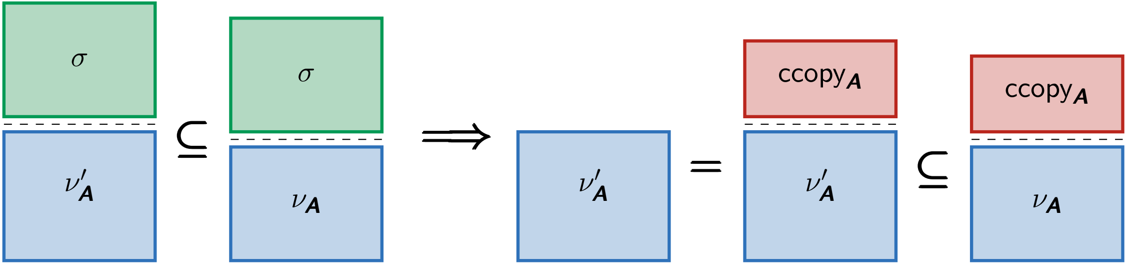


Interaction Refinement: Proof (Forward)



Interaction Refinement: Proof (Backward)

$\forall B. \forall \sigma : A \multimap B.$



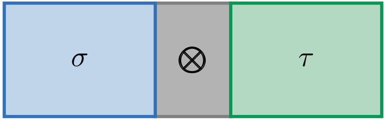
Horizontal Composition

We define a tensor product of strategies:

$$\sigma : \boldsymbol{A} \quad , \quad \tau : \boldsymbol{B} \quad \longmapsto \quad \sigma \otimes \tau : \boldsymbol{A} \otimes \boldsymbol{B}$$

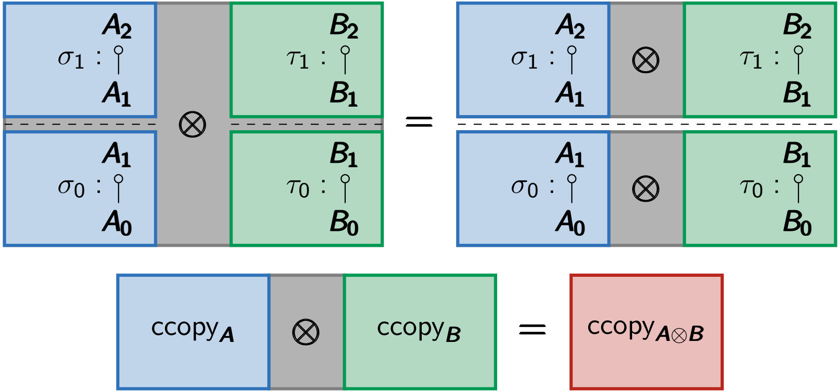
where

$$\sigma \otimes \tau = \text{ all sequentially consistent interleavings of plays of } \sigma \text{ and } \tau$$

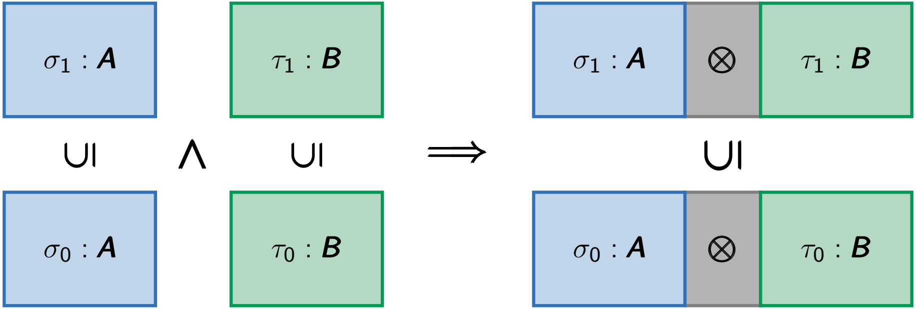


This makes **Game**$_{\mathbf{Conc}}$ into a symmetric monoidal category.
($- \otimes -$ has a unit **1**, is associative and commutative, bifunctor, ...)

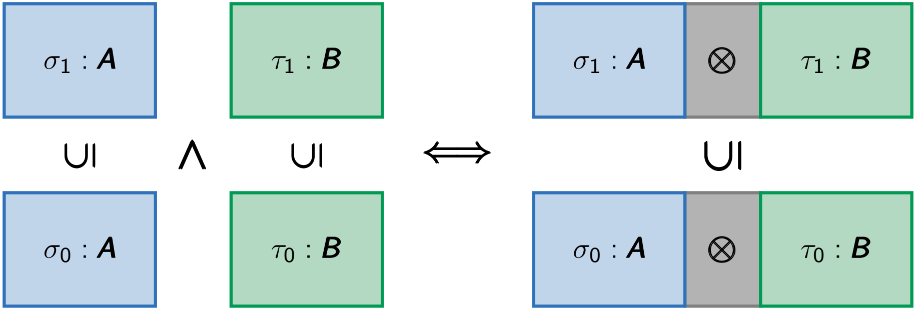
Horizontal Composition: Functorial



Horizontal Composition: Monotonicity

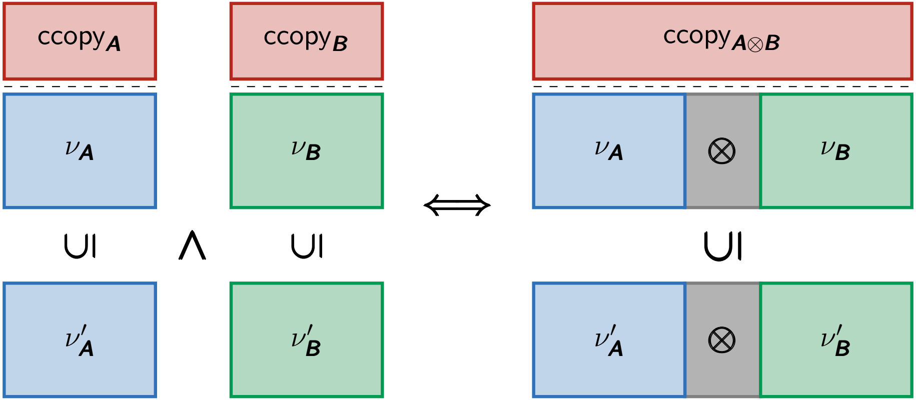


Horizontal Composition: Order-Isomorphism

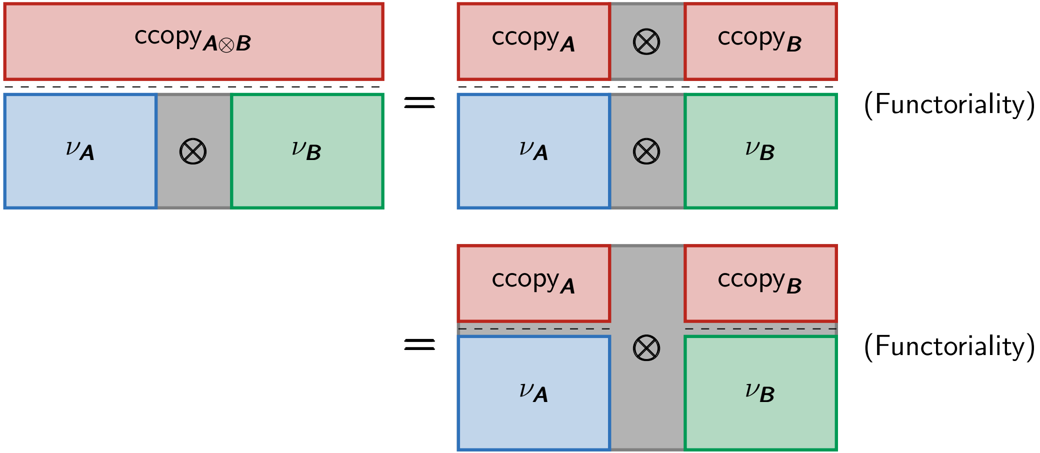


Locality

THEOREM

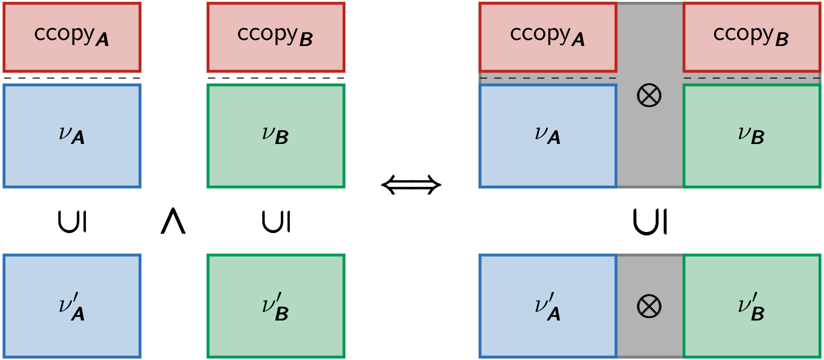


Locality: Proof



Locality: Proof

THEOREM



Holds by the order-isomorphism

Locality Proof [Herlihy and Wing, 1990]

Let $<$ be the transitive closure of the union of all $<_x$ with $<_H$. It is immediate from the construction that $<$ satisfies Conditions (1) and (2), but it remains to be shown that $<$ is a partial order. We argue by contradiction. If not, then there exists a set of operations $e_1, \ldots, e_n$, such that $e_1 < e_2 < \cdots < e_n$, $e_n < e_1$, and each pair is directly related by some $<_x$ or by $<_H$. Choose a cycle whose length is minimal.

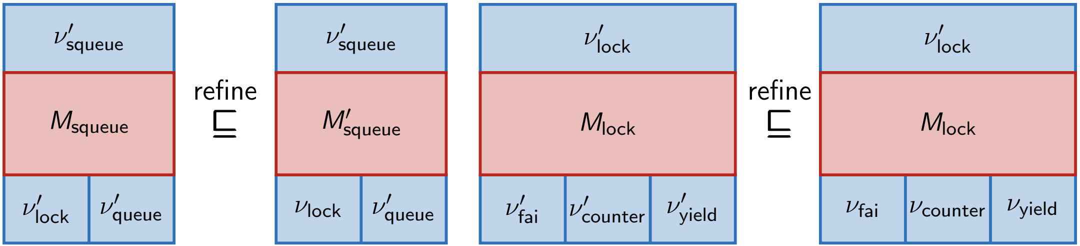
Suppose all operations are associated with the same object x. Since $<_x$ is a total order, there must exist two operations e_{i-1} and e_i such that $e_{i-1} <_H e_i$ and $e_i <_x e_{i-1}$, contradicting the linearizability of x.

The cycle must therefore include operations of at least two objects. By reindexing if necessary, let e_1 and e_2 be operations of distinct objects. Let x be the object associated with e_1. We claim that none of $e_2, \ldots, e_n$ can be an operation of x. The claim holds for e_2 by construction. Let e_i be the first operation in $e_3, \ldots, e_n$ associated with x. Since e_{i-1} and e_i are unrelated by $<_x$, they must be related by $<_H$; hence the response of e_{i-1} precedes the invocation of e_i. The invocation of e_2 precedes the response of e_{i-1}, since otherwise $e_{i-1} <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_{i-1}$. Finally, the response of e_1 precedes the invocation of e_2, since $e_1 <_H e_2$ by construction. It follows that the response to e_1 precedes the invocation of e_i, hence $e_1 <_H e_i$, yielding the shorter cycle $e_1, e_i, \ldots, e_n$.

Since e_n is not an operation of x, but $e_n < e_1$, it follows that $e_n <_H e_1$. But $e_1 <_H e_2$ by construction, and because $<_H$ is transitive, $e_n <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_n$, the final contradiction. □

Outline

Implementing a Shared Queue



Program Logic

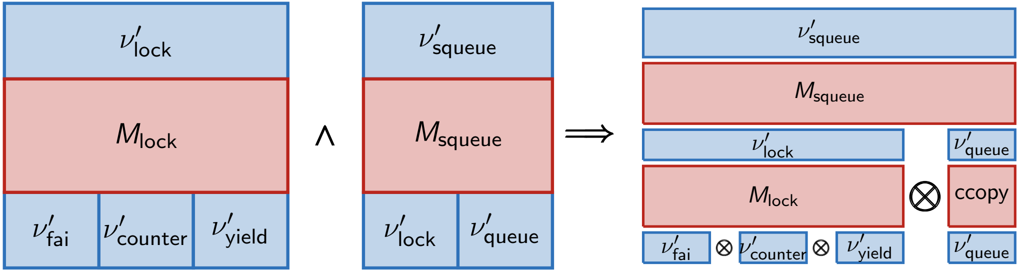
- We define a program logic for showing individual programs implement linearizable objects.
- Sound for our notion of linearizability (and in particular for, interval-linearizability).
- Directly connects with our compositional theory.

PROPOSITION (SOUNDNESS)

If $\mathcal{R}[A], \mathcal{G}[A] \models_A \{P[A]\}\ M[A]\ \{Q[A]\}$ and $(\nu'_E : \dagger\boldsymbol{E}, \nu_E : \dagger\boldsymbol{E})$ is a linearizable concurrent object then

$$\nu'_E ; [\![M[A]]\!] \cap \nu'_F \subseteq K_{\mathsf{Conc}}\ \nu_F$$

Composing Verified Components



Conclusion

Conclusion

- New foundations for linearizability and its properties.
- A compositional theory for linearizability.
- Promising applications for compositional verification.

Check our paper and TR for more:

- The concurrent game semantics model
- The category-theoretic axiomatization
- Thorough comparison with previous work
- The example we described in this talk
- Full program logic description
- More...

Thank you!

Conclusion

Conclusion

- New foundations for linearizability and its properties.
- A compositional theory for linearizability.
- Promising applications for compositional verification.

Check our paper and TR for more:

- The concurrent game semantics model
- The category-theoretic axiomatization
- Thorough comparison with previous work
- The example we described in this talk
- Full program logic description
- More...

Thank you!

Conclusion

Conclusion

- New foundations for linearizability and its properties.
- A compositional theory for linearizability.
- Promising applications for compositional verification.

Check our paper and TR for more:

- The concurrent game semantics model
- The category-theoretic axiomatization
- Thorough comparison with previous work
- The example we described in this talk
- Full program logic description
- More...

Thank you!