Geometric semantics for asynchronous computability

Jérémy Ledent

joint work with Éric Goubault and Samuel Mimram

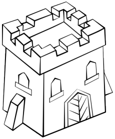
École Polytechnique

CHoCoLa
June 6, 2019

A topological approach for asynchronous computability

The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

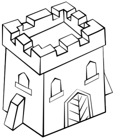
A  B

The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.

A



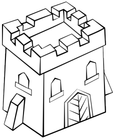
B

The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.

A



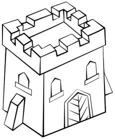
B

The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.
- Messengers might be captured by the enemy, in which case, the message is never received.

A



B

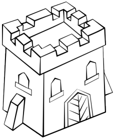
The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.
- Messengers might be captured by the enemy, in which case, the message is never received.

How can they coordinate the attack?

A



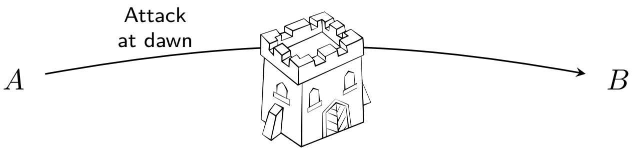
B

The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.
- Messengers might be captured by the enemy, in which case, the message is never received.

How can they coordinate the attack?

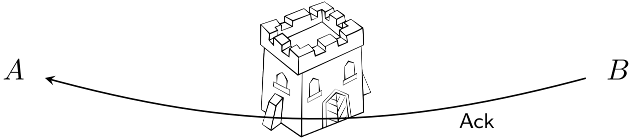


The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.
- Messengers might be captured by the enemy, in which case, the message is never received.

How can they coordinate the attack?

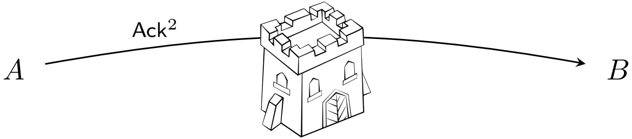


The two generals problem

Two divisions of the same army, commanded by general A and general B, are surrounding an enemy fortress.

- They must attack simultaneously.
- They communicate by sending messengers.
- Messengers might be captured by the enemy, in which case, the message is never received.

How can they coordinate the attack?



Asynchronous computability

a.k.a. Fault-tolerant distributed computing

Goal: Prove that a given concurrent task is unsolvable in a given computational model.

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

Goal: Prove that a given concurrent task is unsolvable in a given computational model.

- Shared memory or message-passing
- *Communication primitives:* read/write, test&set, CAS...
- *Type of errors:* crashes, lost messages, byzantine failures...

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

Goal: Prove that a given concurrent task is unsolvable in a given computational model.

- Shared memory or message-passing
- *Communication primitives:* read/write, test&set, CAS...
- *Type of errors:* crashes, lost messages, byzantine failures...
- *Tasks:* Consensus, weak symmetry breaking, renaming...

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

Goal: Prove that a given concurrent task is unsolvable in a given computational model.

- **Shared memory** or message-passing
- *Communication primitives:* **read/write**, test&set, CAS...
- *Type of errors:* **crashes**, lost messages, byzantine failures...
- *Tasks:* **Consensus**, weak symmetry breaking, renaming...

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

Goal: Prove that a given concurrent task is unsolvable in a given computational model.

- **Shared memory** or message-passing
- *Communication primitives:* **read/write**, test&set, CAS...
- *Type of errors:* **crashes**, lost messages, byzantine failures...
- *Tasks:* **Consensus**, weak symmetry breaking, renaming...

Remark: Usually, impossibility results come from a lack of information about the system, not from a lack of computing power.

A topological approach

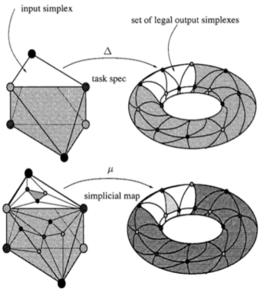


FIG. 13. Asynchronous computability theorem.

THEOREM 3.1 (ASYNCHRONOUS COMPUTABILITY THEOREM). *A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision σ of $\mathcal{I}$ and a color-preserving simplicial map*

$$\mu: \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S in $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(carrier(S, \mathcal{I}))$.

Herlihy and Shavit, 1999
2004 Gödel prize

A topological approach

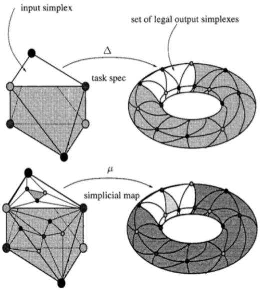


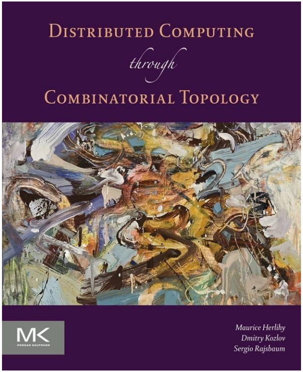
FIG. 13. Asynchronous computability theorem.

THEOREM 3.1 (ASYNCHRONOUS COMPUTABILITY THEOREM). *A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision σ of $\mathcal{I}$ and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S in $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(carrier(S, \mathcal{I}))$.

Herlihy and Shavit, 1999
2004 Gödel prize



Herlihy, Kozlov, Rajsbaum, 2013

Simplicial complexes

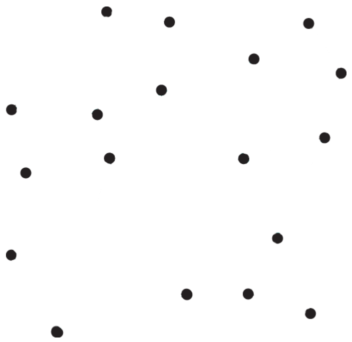
Definition

An (abstract) **simplicial complex** is a pair $\langle V, S\rangle$ where V is a set of *vertices* and S is a downward-closed family of subsets of V called *simplices* (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$).

Simplicial complexes

Definition

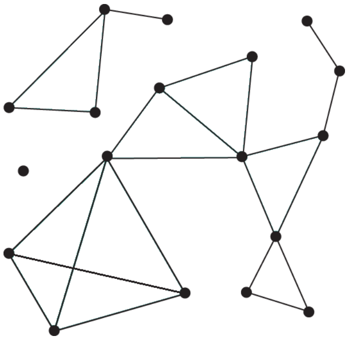
An (abstract) **simplicial complex** is a pair $\langle V, S \rangle$ where V is a set of *vertices* and S is a downward-closed family of subsets of V called *simplices* (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$).



Simplicial complexes

Definition

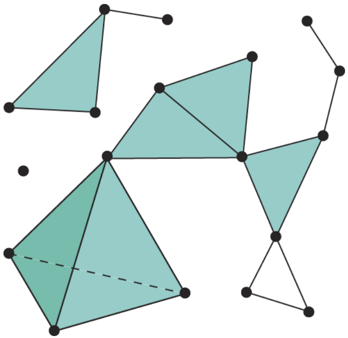
An (abstract) **simplicial complex** is a pair $\langle V, S \rangle$ where V is a set of *vertices* and S is a downward-closed family of subsets of V called *simplices* (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$).



Simplicial complexes

Definition

An (abstract) **simplicial complex** is a pair $\langle V, S \rangle$ where V is a set of *vertices* and S is a downward-closed family of subsets of V called *simplices* (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$).



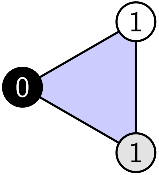
Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

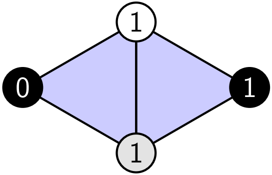
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

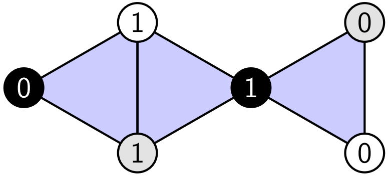
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

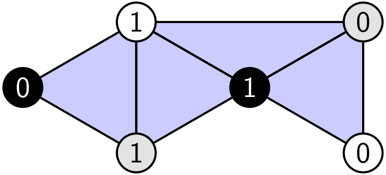
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

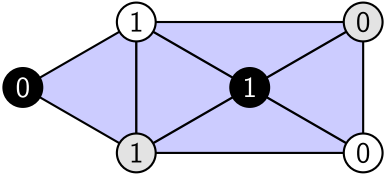
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

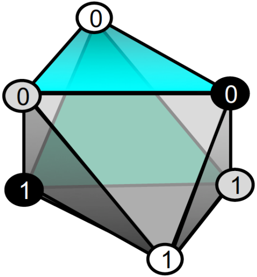
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

In the picture below, the three process names are represented as the colors black, grey, white:



The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$A = \boxed{}\boxed{}\boxed{}$

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$$A = \begin{array}{|c|c|c|}\hline & 2 & \\ \hline\end{array}$$

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$$A = \begin{array}{|c|c|c|}\hline & 2 & 3 \\ \hline\end{array}$$

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$A = \begin{array}{|c|c|c|} \hline \quad & 2 & 3 \\ \hline \end{array}$

R's view: $\begin{array}{|c|c|c|} \hline \quad & 2 & 3 \\ \hline \end{array}$

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$A =$

	2	3

Q's view:		2	3
R's view:		2	3

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

$$A = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline\end{array}$$

Q's view: $\begin{array}{|c|c|c|}\hline & 2 & 3 \\ \hline\end{array}$

R's view: $\begin{array}{|c|c|c|}\hline & 2 & 3 \\ \hline\end{array}$

The immediate snapshot object

```
immediate_snapshot :  'a -> 'a array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

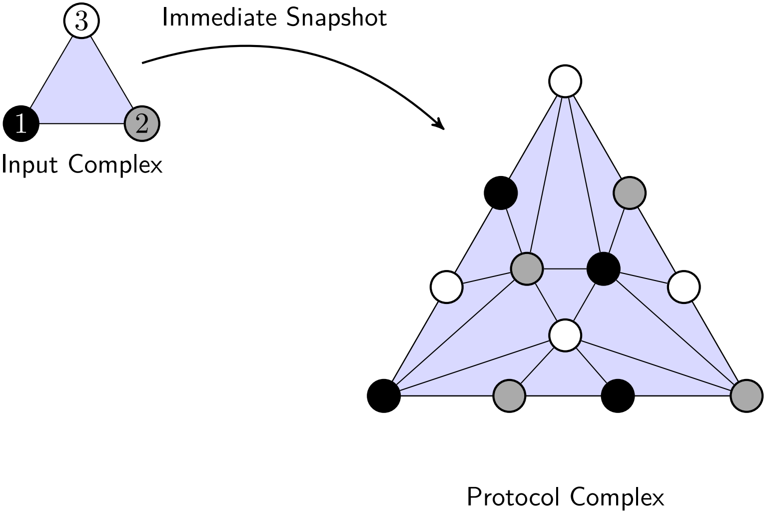
- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs $1, 2, 3$.

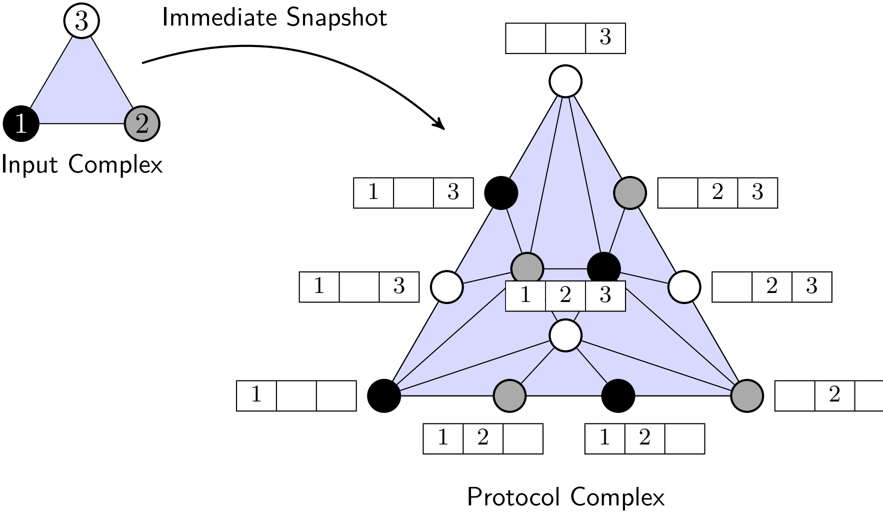
$A = \boxed{1} \boxed{2} \boxed{3}$

P's view:	1	2	3
Q's view:		2	3
R's view:		2	3

Protocol complex for immediate snapshot



Protocol complex for immediate snapshot



The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathsf{in}_i \in \{0, 1\}$.
After communicating, it decides an output $\mathsf{d}_i \in \{0, 1\}$.

The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathsf{in}_i \in \{0,1\}$.
After communicating, it decides an output $\mathsf{d}_i \in \{0,1\}$.

Specification:

- *Agreement:* $\mathsf{d}_i = \mathsf{d}_\mathsf{j}$ for all i, j.
- *Validity:* $\mathsf{d}_i \in \{\mathsf{in}_i \mid 1 \leq i \leq n\}$ for all i.

The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathsf{in}_i \in \{0, 1\}$.
After communicating, it decides an output $\mathsf{d}_i \in \{0, 1\}$.

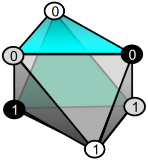
Specification:

- *Agreement:* $\mathsf{d}_i = \mathsf{d}_\mathsf{j}$ for all i, j.
- *Validity:* $\mathsf{d}_i \in \{\mathsf{in}_i \mid 1 \leq i \leq n\}$ for all i.

Examples: for 3 processes

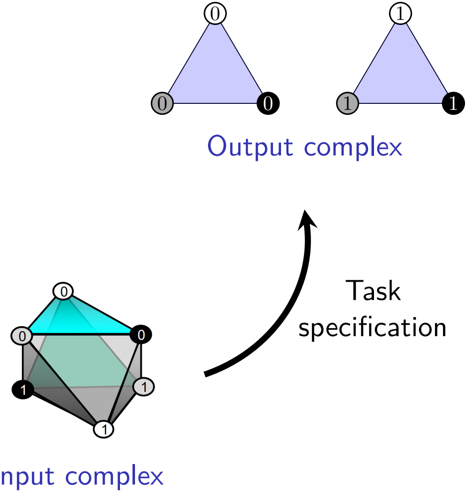
- if the inputs are $(0, 0, 0)$, the outputs must be $(0, 0, 0)$.
- if the inputs are $(1, 0, 1)$, the outputs can be either $(0, 0, 0)$ or $(1, 1, 1)$.

Topological definition of task solvability

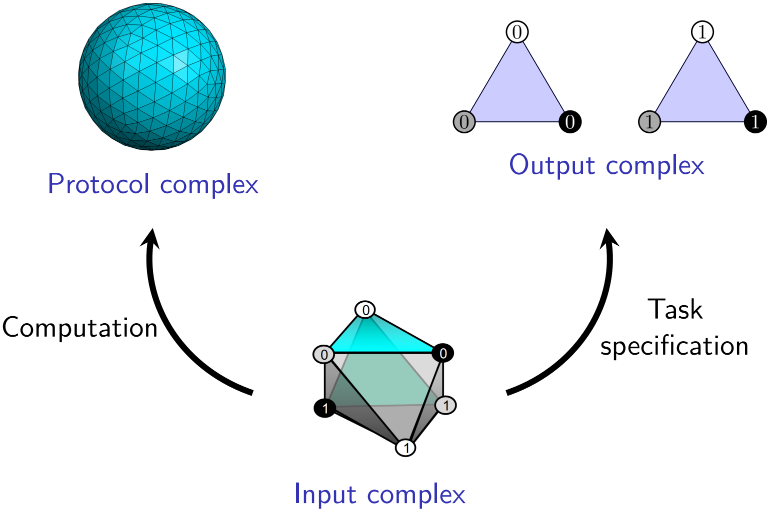


Input complex

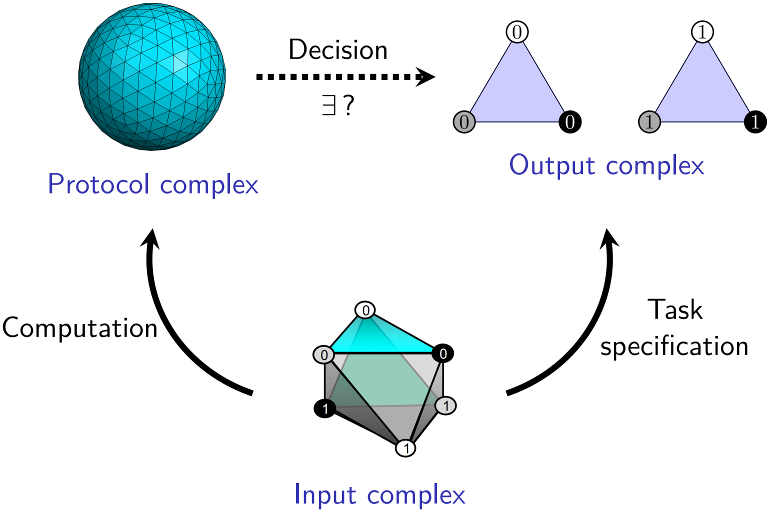
Topological definition of task solvability



Topological definition of task solvability



Topological definition of task solvability



Asynchronous computability theorem

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write** registers if and only if there is a decision map from **a subdivision of** the input complex into the output complex such that [...].

Asynchronous computability theorem

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write** registers if and only if there is a decision map from **a subdivision of** the input complex into the output complex such that [...].

What if:

- we replace "wait-free" by "t-resilient"?
- we use other objects instead of read/write registers?
- we use a message-passing architecture?

Asynchronous computability theorem

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write** registers if and only if there is a decision map from **a subdivision of** the input complex into the output complex such that [...].

What if:

- we replace "wait-free" by "t-resilient"?
- we use other objects instead of read/write registers?
- we use a message-passing architecture?

Goal: an asynchronous computability theorem for any objects.

Specifying concurrent objects

Objects vs Tasks

"Can we solve the task T using the objects $A_1, \ldots, A_k$?"

Objects vs Tasks

"Can we solve the task T using the objects $A_1, \ldots, A_k$?"

Objects:

- Long-lived

Tasks:

- Used only once

Objects vs Tasks

"Can we solve the task T using the objects $A_1, \ldots, A_k$?"

Objects:

- Long-lived
- Have a sequential flavor

Tasks:

- Used only once
- Intrinsically concurrent

Objects vs Tasks

"Can we solve the task T using the objects $A_1, \ldots, A_k$?"

Objects:

- Long-lived
- Have a sequential flavor

Tasks:

- Used only once
- Intrinsically concurrent

But in practice:

"I can solve consensus using X, and I can solve Y using consensus objects, so I can solve Y using X"

Objects vs Tasks

"Can we solve the task T using the objects $A_1, \dots, A_k$?"

Objects:

- Long-lived
- Have a sequential flavor

Tasks:

- Used only once
- Intrinsically concurrent

But in practice:

"I can solve consensus using X, and I can solve Y using consensus objects, so I can solve Y using X"

$\longrightarrow$ We would like a composable notion of "solving".

Objects

From now on, everything is an object:

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...
- Message-passing interfaces

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...
- Message-passing interfaces
- Immediate-snapshot, consensus, set-agreement, ...

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...
- Message-passing interfaces
- Immediate-snapshot, consensus, set-agreement, ...

"Can we implement the object B using the objects $A_1, \ldots, A_k$?"

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...
- Message-passing interfaces
- Immediate-snapshot, consensus, set-agreement, ...

"Can we implement the object B using the objects $A_1, \ldots, A_k$?"

$\longrightarrow$ How do we specify a concurrent object?

Objects

From now on, everything is an object:

- Hardware: Read/Write registers, test&set, CAS, ...
- Data structures: lists, queues, hashmaps, ...
- Message-passing interfaces
- Immediate-snapshot, consensus, set-agreement, ...

"Can we implement the object B using the objects $A_1, \ldots, A_k$?"

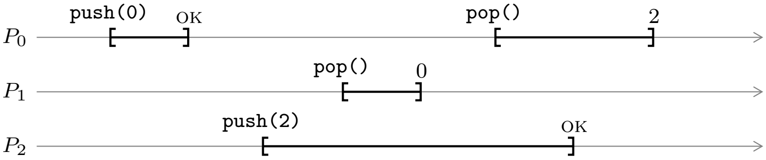
$\longrightarrow$ How do we specify a concurrent object?
$\longrightarrow$ What does it mean to implement an object?

Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).

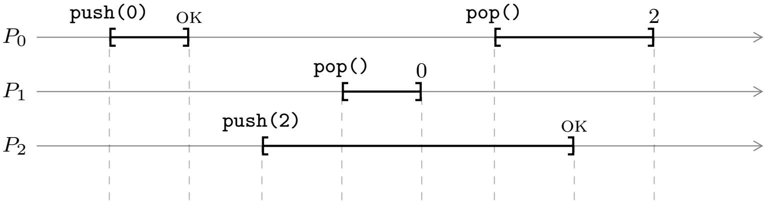
Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).



Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).



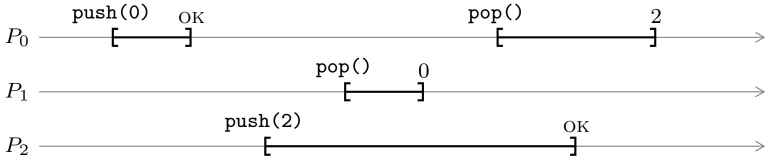
$$T = \mathsf{i}_0^{\mathtt{push},0} \cdot \mathsf{r}_0^{\mathrm{OK}} \cdot \mathsf{i}_2^{\mathtt{push},2} \cdot \mathsf{i}_1^{\mathtt{pop}} \cdot \mathsf{r}_1^{2} \cdot \mathsf{i}_0^{\mathtt{pop}} \cdot \mathsf{r}_2^{\mathrm{OK}} \cdot \mathsf{r}_0^{0}$$

Trace formalism:

- Time is abstracted away.
- Alternation of invocations and responses on each process.

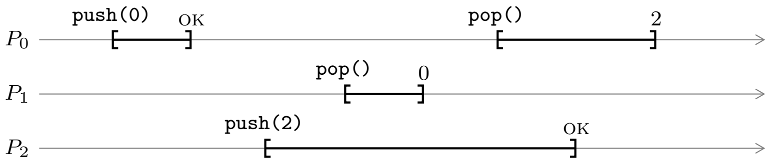
Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).



Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).

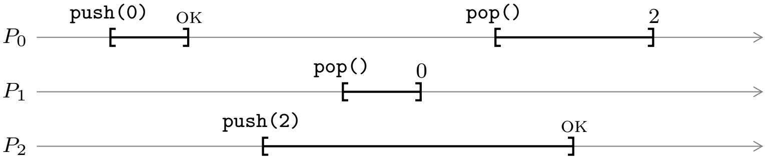


Write $\mathcal{T}$ for the set of all execution traces.

- A *concurrent specification* is a subset $\sigma \subseteq \mathcal{T}$.

Concurrent specifications

Idea: the specification of an object is the set of all the correct execution traces (Lamport, 1986).



Write $\mathcal{T}$ for the set of all execution traces.

- A *concurrent specification* is a subset $\sigma \subseteq \mathcal{T}$.
- A program implements a specification σ if all the traces that it can produce belong to σ.

Linearizability (Herlihy & Wing, 1990)

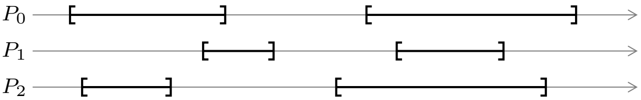
$$\text{SeqSpec} \xrightarrow{\text{Lin}} \text{ConcSpec}$$

- **Input:** a sequential specification σ (e.g. list, queue, ...).
- **Output:** a concurrent specification $\text{Lin}(\sigma)$.

Linearizability (Herlihy & Wing, 1990)

$$\text{SeqSpec} \xrightarrow{\text{Lin}} \text{ConcSpec}$$

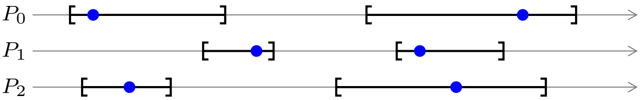
- **Input:** a sequential specification σ (e.g. list, queue, ...).
- **Output:** a concurrent specification $\text{Lin}(\sigma)$.



Linearizability (Herlihy & Wing, 1990)

$$\text{SeqSpec} \xrightarrow{\text{Lin}} \text{ConcSpec}$$

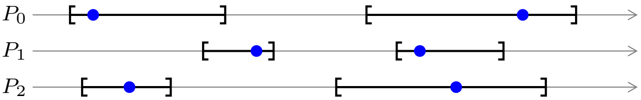
- **Input:** a sequential specification σ (e.g. list, queue, ...).
- **Output:** a concurrent specification $\text{Lin}(\sigma)$.



Linearizability (Herlihy & Wing, 1990)

$$\text{SeqSpec} \xrightarrow{\text{Lin}} \text{ConcSpec}$$

- **Input:** a sequential specification σ (e.g. list, queue, ...).
- **Output:** a concurrent specification $\text{Lin}(\sigma)$.

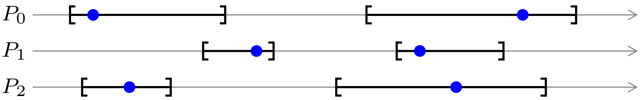


$$\text{Lin}(\sigma) = \{T \text{ concurrent trace} \mid T \text{ is linearizable w.r.t. } \sigma\}$$

Linearizability (Herlihy & Wing, 1990)

$$\text{SeqSpec} \xrightarrow{\text{Lin}} \text{ConcSpec}$$

- **Input:** a sequential specification σ (e.g. list, queue, ...).
- **Output:** a concurrent specification $\text{Lin}(\sigma)$.



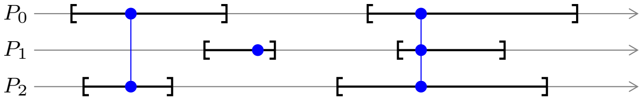
$$\text{Lin}(\sigma) = \{T \text{ concurrent trace} \mid T \text{ is linearizable w.r.t. } \sigma\}$$

Some objects are not linearizable!

Their specification cannot be expressed as $\text{Lin}(\sigma)$, for any σ.

Concurrent variants of linearizability

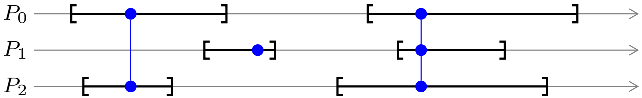
Set-linearizability (Neiger, 1994)



- Can specify: exchanger, immediate snapshot, set agreement.

Concurrent variants of linearizability

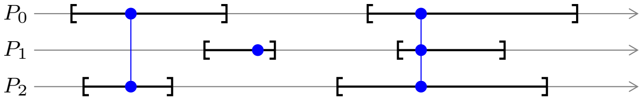
Set-linearizability (Neiger, 1994)



- Can specify: exchanger, immediate snapshot, set agreement.
- Cannot specify: validity, write-snapshot.

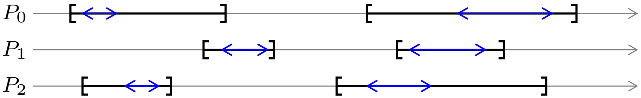
Concurrent variants of linearizability

Set-linearizability (Neiger, 1994)



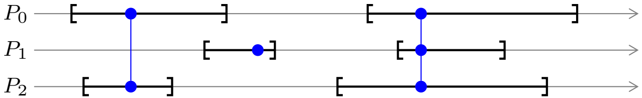
- Can specify: exchanger, immediate snapshot, set agreement.
- Cannot specify: validity, write-snapshot.

Interval-linearizability (Castañeda, Rajsbaum, Raynal, 2015)



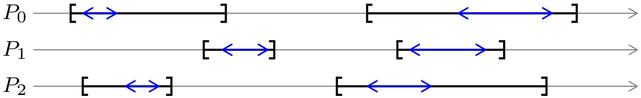
Concurrent variants of linearizability

Set-linearizability (Neiger, 1994)



- Can specify: exchanger, immediate snapshot, set agreement.
- Cannot specify: validity, write-snapshot.

Interval-linearizability (Castañeda, Rajsbaum, Raynal, 2015)



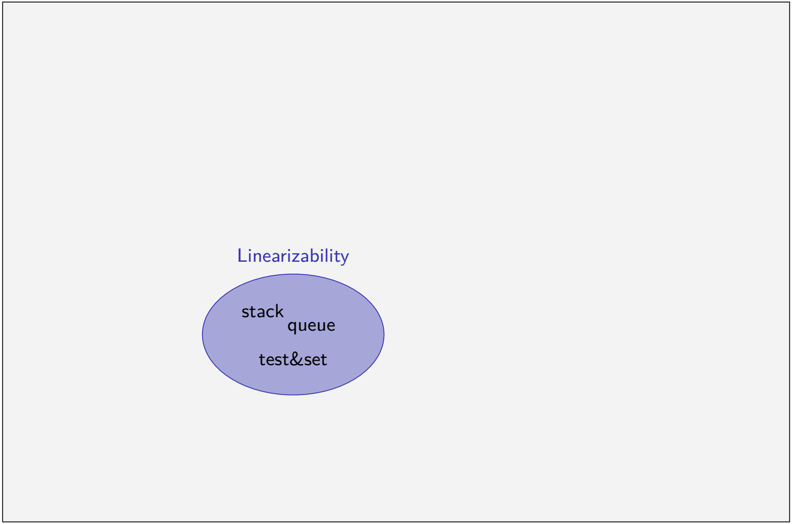
- Can specify every task!

Overview

Concurrent specifications

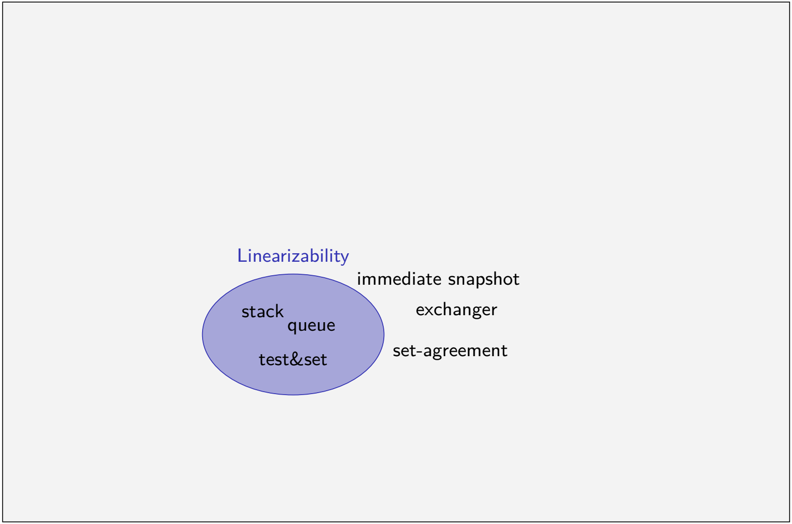
Overview

Concurrent specifications



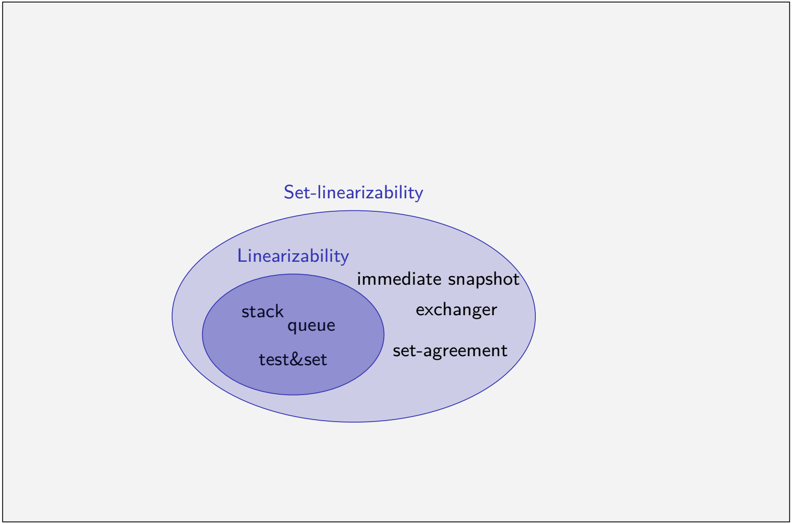
Overview

Concurrent specifications



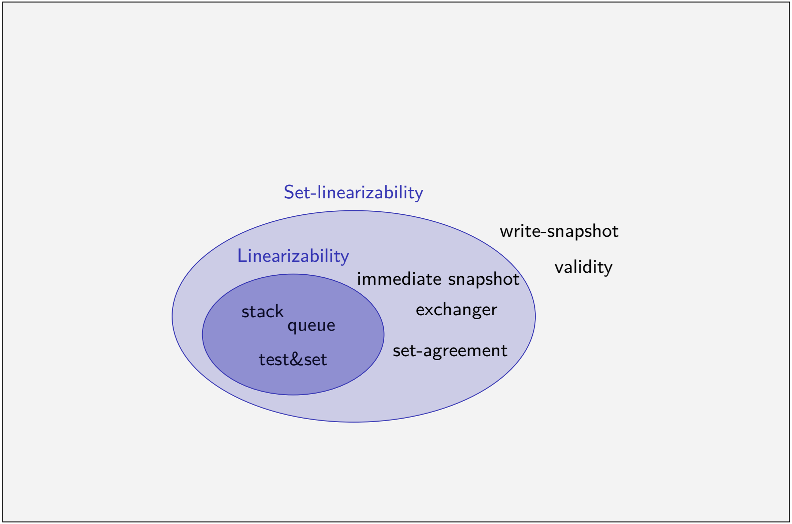
Overview

Concurrent specifications



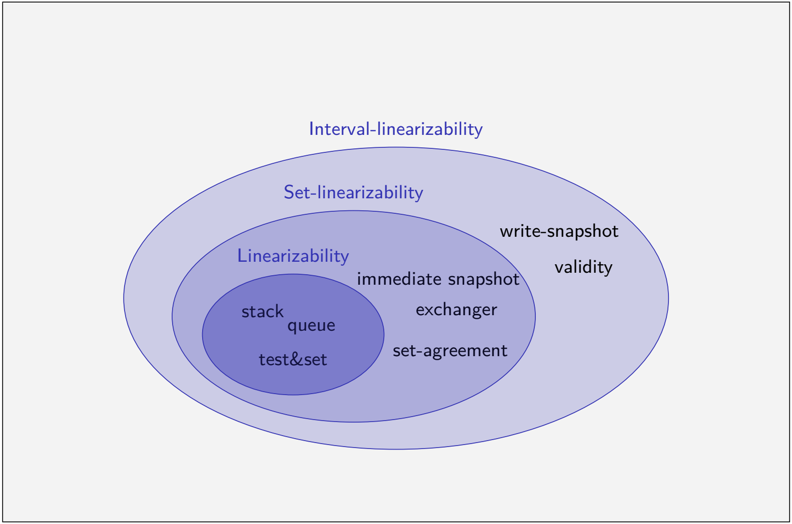
Overview

Concurrent specifications

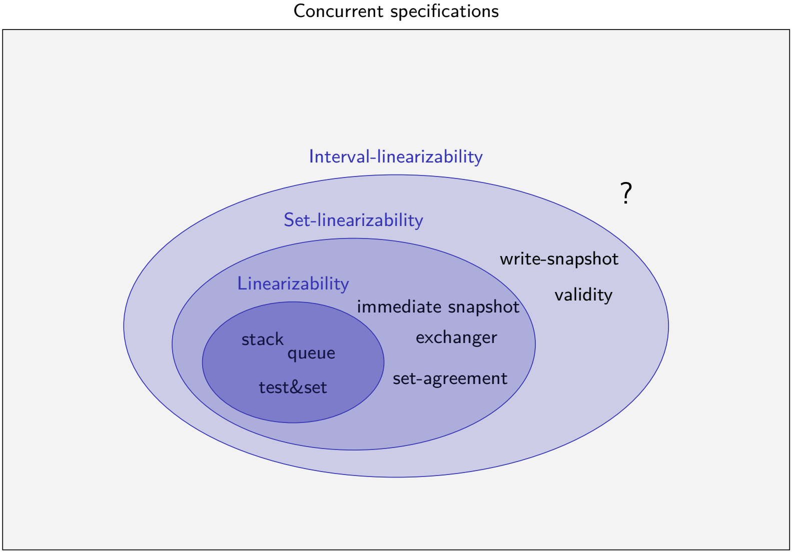


Overview

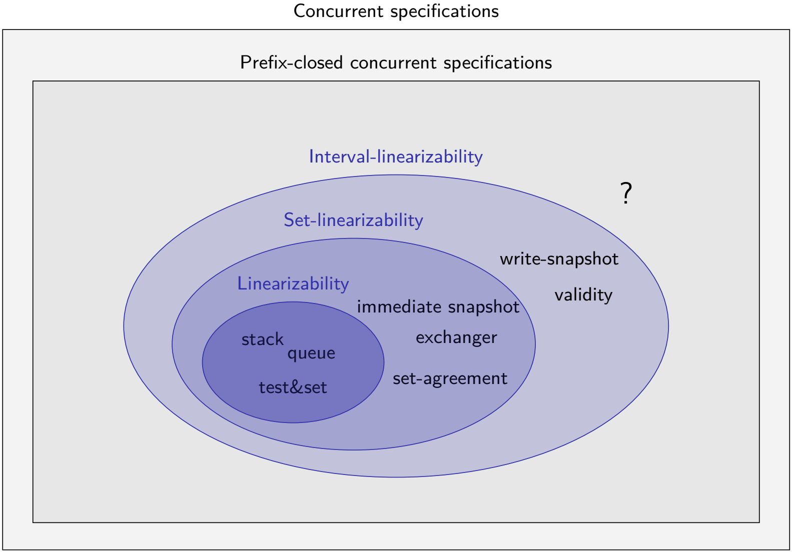
Concurrent specifications



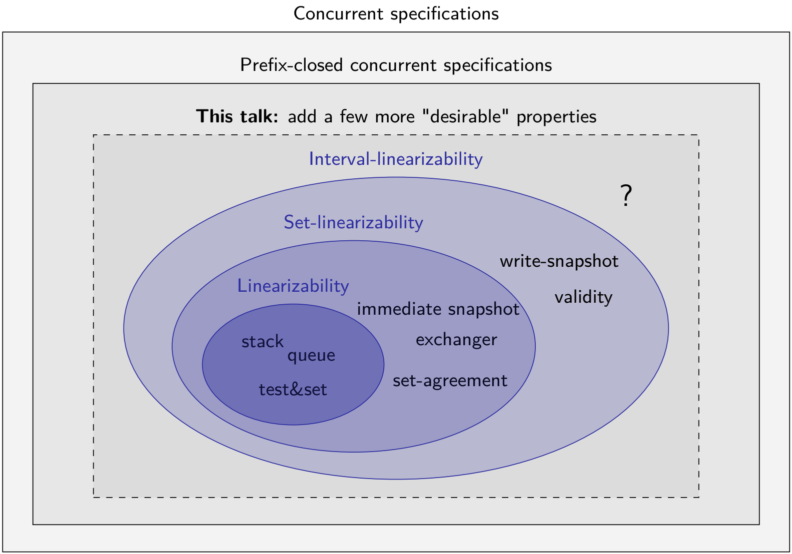
Overview


Concurrent specifications
Interval-linearizability
?
Set-linearizability
write-snapshot
validity
Linearizability
immediate snapshot
exchanger
stack
queue
test&set
set-agreement

Overview

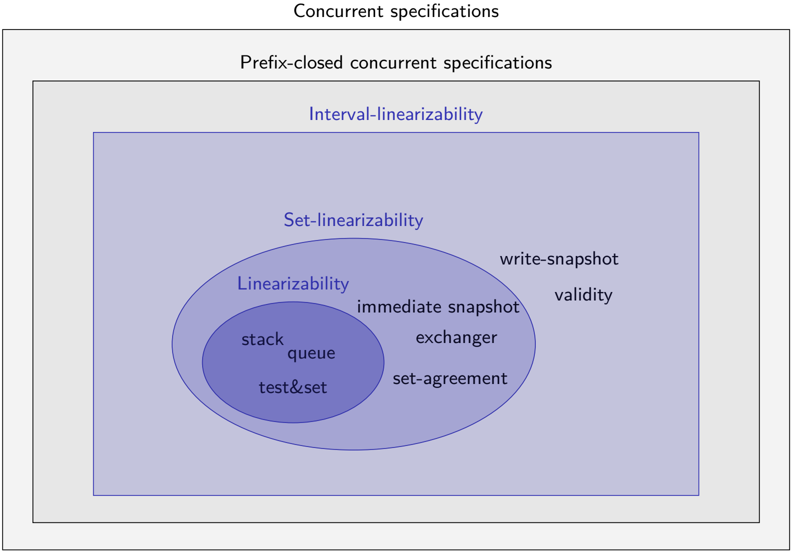

Concurrent specifications
Prefix-closed concurrent specifications
Interval-linearizability
?
Set-linearizability
write-snapshot
validity
Linearizability
immediate snapshot
exchanger
set-agreement
stack
queue
test&set

Overview



Concurrent specifications
Prefix-closed concurrent specifications
This talk: add a few more "desirable" properties
Interval-linearizability
?
Set-linearizability
write-snapshot
validity
Linearizability
immediate snapshot
exchanger
set-agreement
stack
queue
test&set

Overview


Concurrent specifications
Prefix-closed concurrent specifications
Interval-linearizability
Set-linearizability
write-snapshot
validity
Linearizability
immediate snapshot
stack
queue
exchanger
test&set
set-agreement

Relevant concurrent specifications

We write ConcSpec for the set of concurrent specifications $\sigma \subseteq \mathcal{T}$ satisfying the following properties.

(1) *prefix-closure*: if $t \cdot t' \in \sigma$ then $t \in \sigma$,

(2) *non-emptiness*: $\varepsilon \in \sigma$,

(3) *receptivity*: if $t \in \sigma$ and t has no pending invocation of process i, then $t \cdot \mathsf{i}_i^x \in \sigma$ for every input value x,

Relevant concurrent specifications

We write ConcSpec for the set of concurrent specifications $\sigma \subseteq \mathcal{T}$ satisfying the following properties.

(1) *prefix-closure*: if $t \cdot t' \in \sigma$ then $t \in \sigma$,

(2) *non-emptiness*: $\varepsilon \in \sigma$,

(3) *receptivity*: if $t \in \sigma$ and t has no pending invocation of process i, then $t \cdot \mathsf{i}_i^x \in \sigma$ for every input value x,

(4) *totality*: if $t \in \sigma$ and t has a pending invocation of process i, then there exists an output x such that $t \cdot \mathsf{r}_i^x \in \sigma$,

Relevant concurrent specifications

We write ConcSpec for the set of concurrent specifications $\sigma \subseteq \mathcal{T}$ satisfying the following properties.

(1) *prefix-closure*: if $t \cdot t' \in \sigma$ then $t \in \sigma$,

(2) *non-emptiness*: $\varepsilon \in \sigma$,

(3) *receptivity*: if $t \in \sigma$ and t has no pending invocation of process i, then $t \cdot \mathsf{i}_i^x \in \sigma$ for every input value x,

(4) *totality*: if $t \in \sigma$ and t has a pending invocation of process i, then there exists an output x such that $t \cdot \mathsf{r}_i^x \in \sigma$,

(5) σ has the *expansion* property.

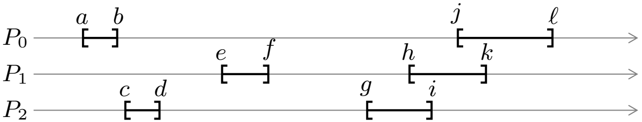
Expansion of intervals

A concurrent specification satisfies the expansion property if:

Expansion of intervals

A concurrent specification satisfies the expansion property if:

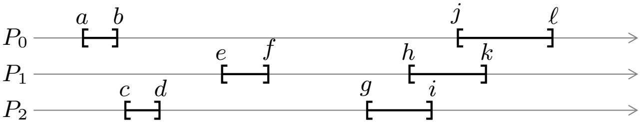
For any correct execution trace,



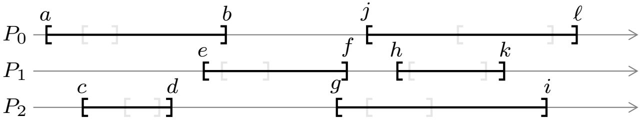
Expansion of intervals

A concurrent specification satisfies the expansion property if:

For any correct execution trace,



if we *expand* the intervals,



then the resulting trace is still correct.

Example: the Exchanger object

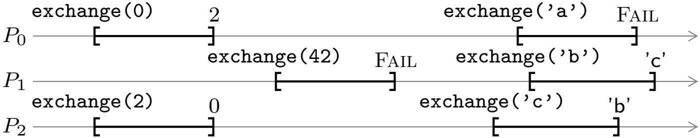
Similar to the one available in Java[1]: *"A synchronization point at which threads can pair and swap elements within pairs".*
Here, we consider a wait-free variant.

[1] `java.util.concurrent.Exchanger<V>`

Example: the Exchanger object

Similar to the one available in Java[1]: *"A synchronization point at which threads can pair and swap elements within pairs".*
Here, we consider a wait-free variant.

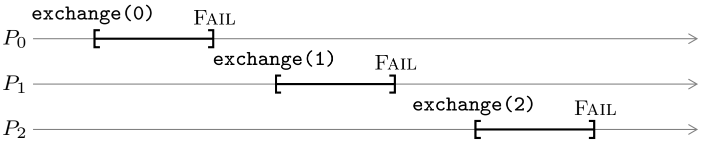
A typical execution of the exchanger looks like this:



[1] java.util.concurrent.Exchanger<V>

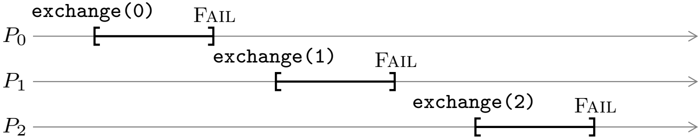
Example: the Exchanger object (2)

The following execution is correct:

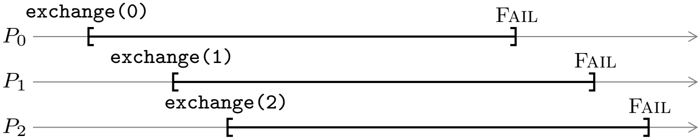


Example: the Exchanger object (2)

The following execution is correct:



Hence, according to the expansion property,



should be considered correct too!

Linearizability gives expansion for free

Linearizability-based techniques always produce specifications which satisfy the expansion property.

Theorem

For every sequential specification σ, $\mathsf{Lin}(\sigma) \in \mathsf{ConcSpec}$*.*

Linearizability gives expansion for free

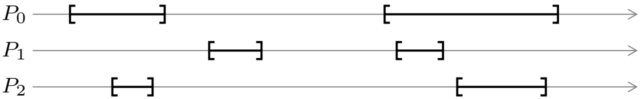
Linearizability-based techniques always produce specifications which satisfy the expansion property.

Theorem

For every sequential specification σ, $\mathrm{Lin}(\sigma) \in \mathrm{ConcSpec}$.

Proof.

If some execution trace is linearizable,



Linearizability gives expansion for free

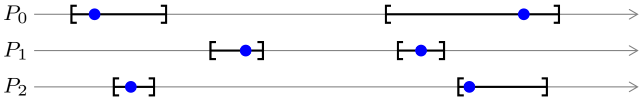
Linearizability-based techniques always produce specifications which satisfy the expansion property.

Theorem

For every sequential specification σ, $\mathsf{Lin}(\sigma) \in \mathsf{ConcSpec}$.

Proof.

If some execution trace is linearizable,



Linearizability gives expansion for free

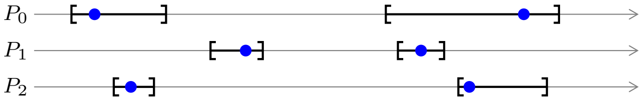
Linearizability-based techniques always produce specifications which satisfy the expansion property.

Theorem

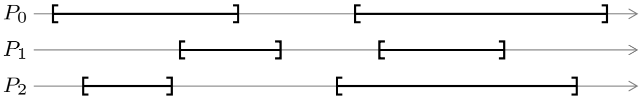
For every sequential specification σ, $\mathsf{Lin}(\sigma) \in \mathsf{ConcSpec}$.

Proof.

If some execution trace is linearizable,



Then any trace obtained by expanding it is still linearizable.



□

Linearizability gives expansion for free

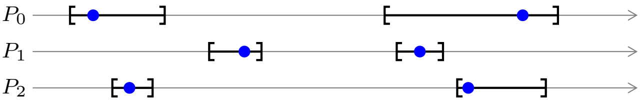
Linearizability-based techniques always produce specifications which satisfy the expansion property.

Theorem

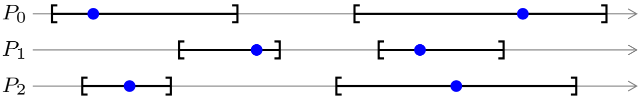
For every sequential specification σ*,* $\mathsf{Lin}(\sigma) \in \mathsf{ConcSpec}$*.*

Proof.

If some execution trace is linearizable,



Then any trace obtained by expanding it is still linearizable.



□

A Galois connection

$$\begin{tikzcd} \text{SeqSpec} \arrow[rr, bend right, "\text{Lin}"'] & & \text{ConcSpec} \end{tikzcd}$$

A Galois connection

$$
\begin{tikzcd}
\mathsf{SeqSpec} \arrow[rr, bend right, "\mathsf{Lin}"'] & & \mathsf{ConcSpec} \arrow[ll, bend right, "\mathsf{U}"']
\end{tikzcd}
$$

A Galois connection

$$\begin{tikzcd} \mathsf{SeqSpec} \arrow[rr, bend right, "\mathsf{Lin}"'] \arrow[rr, phantom, "\top"] & & \mathsf{ConcSpec} \arrow[ll, bend right, "\mathsf{U}"'] \end{tikzcd}$$

Theorem

The maps Lin *and* U *form a Galois connection: for every* $\sigma \in \mathsf{SeqSpec}$ *and* $\tau \in \mathsf{ConcSpec}$,

$$\mathsf{Lin}(\sigma) \subseteq \tau \qquad \iff \qquad \sigma \subseteq \mathsf{U}(\tau)$$

Applications

- By the properties of Galois connections,

$$\mathsf{Lin}(\mathsf{U}(\mathsf{Lin}(\sigma))) = \mathsf{Lin}(\sigma)$$

  This yields a simple criterion to check whether a given specification τ is linearizable: check whether $\mathsf{Lin}(\mathsf{U}(\tau)) = \tau$.

Applications

- By the properties of Galois connections,

$$\mathsf{Lin}(\mathsf{U}(\mathsf{Lin}(\sigma))) = \mathsf{Lin}(\sigma)$$

  This yields a simple criterion to check whether a given specification τ is linearizable: check whether $\mathsf{Lin}(\mathsf{U}(\tau)) = \tau$.

- The Galois connection for interval linearizability has the following corollary:

Theorem

ConcSpec *is the set of interval-linearizable specifications.*

A computational model

We fix a set $\{A_1, \ldots, A_k\}$ of shared objects, along with their concurrent specifications.

A computational model

We fix a set $\{A_1, \ldots, A_k\}$ of shared objects, along with their concurrent specifications.

A program P using these objects can:

- call the objects,
- do local computations,
- use branching, loops.

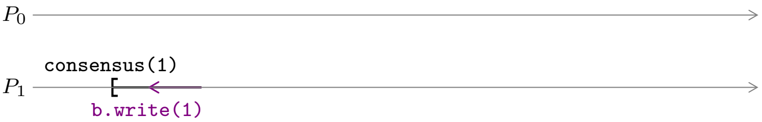
A protocol $\mathcal{P}$ consists of one program for each process.

```
consensus(v) {
  b.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := a.read();
    return v';
}
```

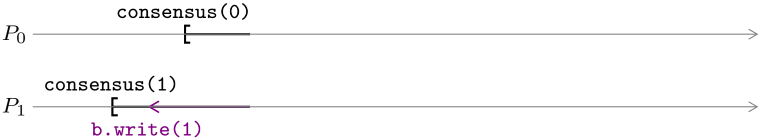
A computational model (2)

A computational model (2)

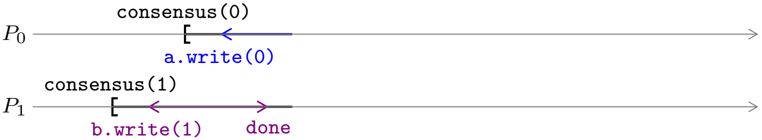
A computational model (2)



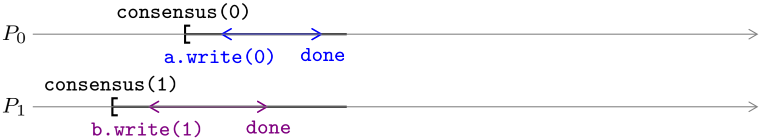
A computational model (2)



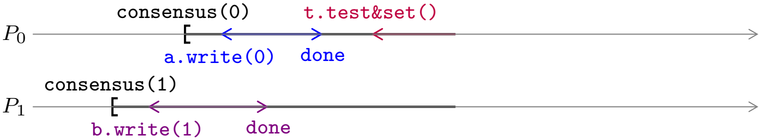
A computational model (2)



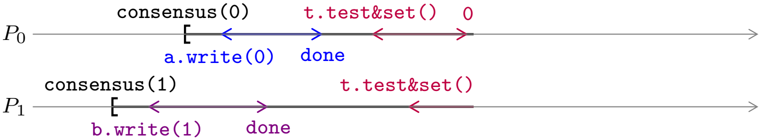
A computational model (2)



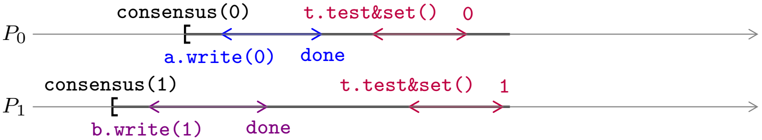
A computational model (2)



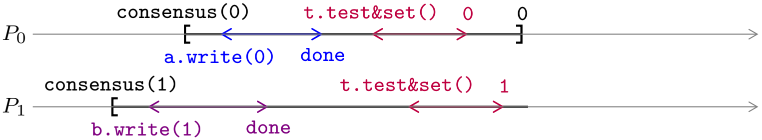
A computational model (2)



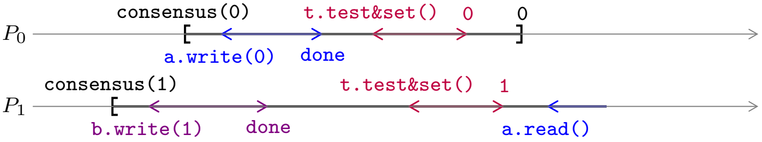
A computational model (2)



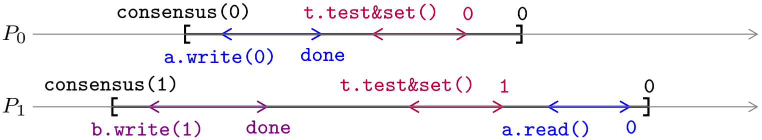
A computational model (2)



A computational model (2)

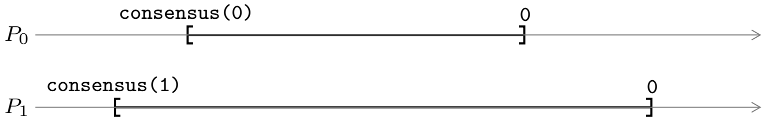


A computational model (2)



A computational model (2)

A computational model (2)



The semantics $[\![\mathcal{P}]\!]$ of a protocol is the set of execution traces that it can produce.

It implements an object specification $S \in \text{ConcSpec}$ if $[\![\mathcal{P}]\!] \subseteq S$.

A computational model (2)

The semantics $[\![\mathcal{P}]\!]$ of a protocol is the set of execution traces that it can produce.
It implements an object specification $S \in \mathsf{ConcSpec}$ if $[\![\mathcal{P}]\!] \subseteq S$.

Theorem

For any wait-free protocol $\mathcal{P}$, $\quad [\![\mathcal{P}]\!] \in \mathsf{ConcSpec}$.

Asynchronous computability theorem

- Tasks are now a particular kind of concurrent object:

$$\mathsf{Tasks} \hookrightarrow \mathsf{ConcSpec}$$

Asynchronous computability theorem

- Tasks are now a particular kind of concurrent object:

$$\text{Tasks} \hookrightarrow \text{ConcSpec}$$

- Define the protocol complex for a given protocol $\mathcal{P}$:

$$\text{Views of process } P_i \quad \simeq \quad \text{States of the CFG of its program}$$

Asynchronous computability theorem

- Tasks are now a particular kind of concurrent object:

$$\text{Tasks} \hookrightarrow \text{ConcSpec}$$

- Define the protocol complex for a given protocol $\mathcal{P}$:

$$\text{Views of process } P_i \quad \simeq \quad \text{States of the CFG of its program}$$

Theorem

A wait-free protocol implements a task if and only if there exists a decision map from the protocol complex to the output complex that makes the diagram commute.

Future work

- Get rid of the wait-free requirement:
  - t-resilient protocols
  - allows to model objects such as semaphores, barriers, ...

Future work

- Get rid of the wait-free requirement:
  - t-resilient protocols
  - allows to model objects such as semaphores, barriers, ...
- All tasks are objects, but not all objects are tasks: find a topological characterization for the other objects.
  $\rightarrow$ Rajsbaum et al. proposed a notion of *refined tasks*.

Future work

- Get rid of the wait-free requirement:
  - t-resilient protocols
  - allows to model objects such as semaphores, barriers, ...
- All tasks are objects, but not all objects are tasks: find a topological characterization for the other objects.
  $\rightarrow$ Rajsbaum et al. proposed a notion of *refined tasks*.
- Game semantics perspective
  - our notion of implementation looks like the composition of strategies
  - can we characterize the immediate-snapshot strategies, and deduce impossibility results from it?

Thanks!