A Sound Foundation for the Topological Approach to Task Solvability

Jérémy Ledent and Samuel Mimram

LIX, École Polytechnique, France

CONCUR'19, Amsterdam
August 30, 2019

Introduction

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, ...

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, ...

Objects:

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

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, ...

Objects:

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

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, . . .

Objects:

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

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, . . .

Objects:

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

Asynchronous computability

a.k.a. Fault-tolerant distributed computing

A fixed number n of asynchronous processes communicate through shared objects in order to solve a concurrent task.

Tasks: Consensus, set agreement, renaming, ...

Objects:

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

Problem

Can we solve the task Θ using the objects $A_1, \ldots, A_n$?

A topological approach

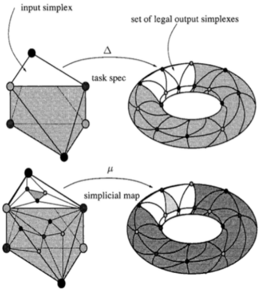


FIG. 13. Asynchronous computability theorem.

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*

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

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

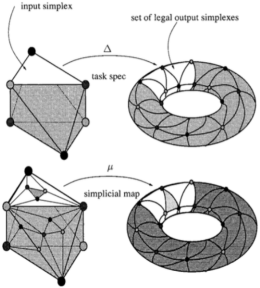
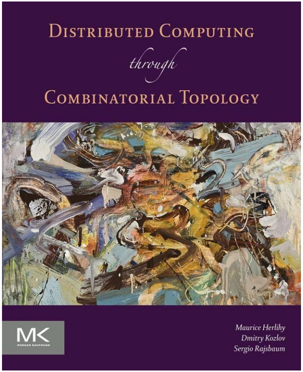


FIG. 13. Asynchronous computability theorem.

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*

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

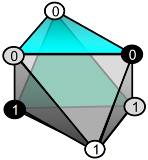
such that for each simplex S in $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(carrier(S, \mathcal{I}))$.



Herlihy and Shavit, 1999
2004 Gödel prize

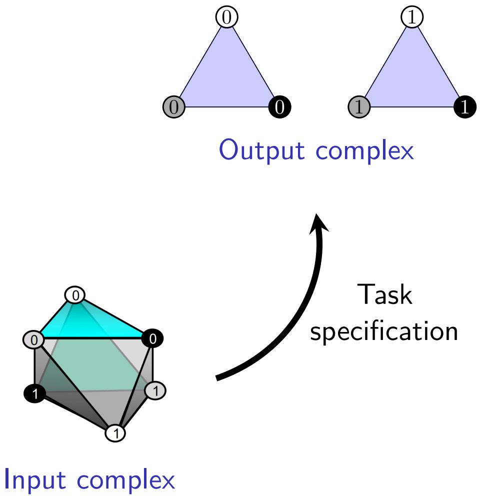
Herlihy, Kozlov, Rajsbaum, 2013

Asynchronous Computability Theorem

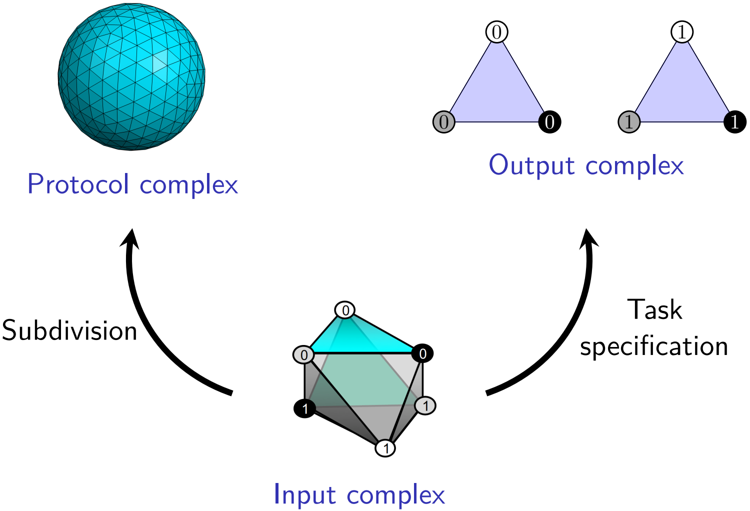


Input complex

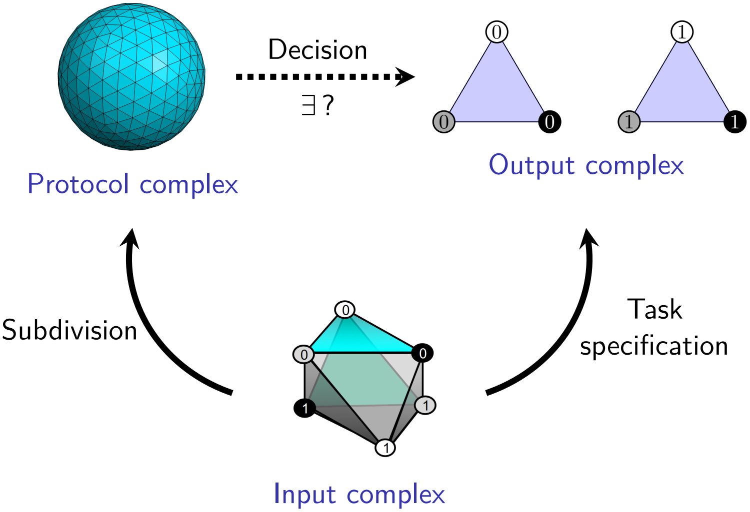
Asynchronous Computability Theorem



Asynchronous Computability Theorem



Asynchronous Computability Theorem



Asynchronous Computability Theorem (2)

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 the protocol complex into the output complex such that [...].

Asynchronous Computability Theorem (2)

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 the protocol complex into the output complex such that [...].

What if:

- we replace "wait-free" by "t-resilient"?

Asynchronous Computability Theorem (2)

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 the protocol complex into the output complex such that [...].

What if:

- we replace "wait-free" by "t-resilient"?
  $\longrightarrow$ *Asynchronous Computability Theorems for t-resilient systems*, Saraph, Herlihy, Gafni (DISC 2016).

Asynchronous Computability Theorem (2)

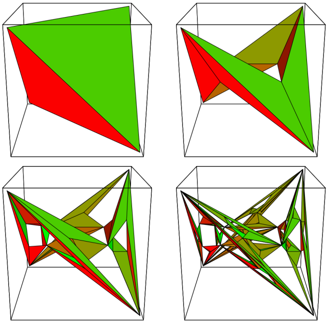
Theorem (Herlihy and Shavit, 1999)

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

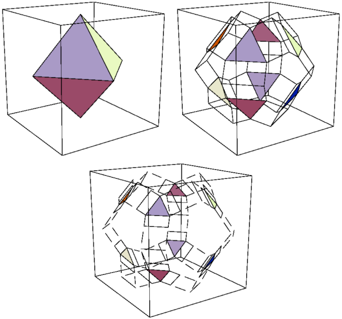
What if:

- we replace "wait-free" by "t-resilient"?
  $\longrightarrow$ *Asynchronous Computability Theorems for t-resilient systems*, Saraph, Herlihy, Gafni (DISC 2016).
- we use other objects instead of read/write registers?
  $\longrightarrow$ This talk.

Protocol complexes for other objects

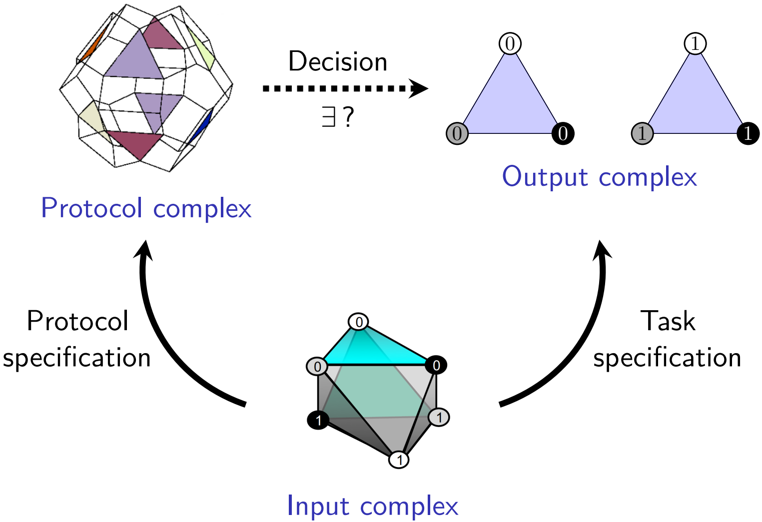


For test-and-set protocols
Herlihy, Rajsbaum, PODC'94



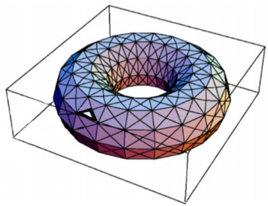
For synchronous message-passing
Herlihy, Rajsbaum, Tuttle, 2001

Topological **definition** of solvability



Benefits and drawbacks

✓ We can prove very general abstract results:



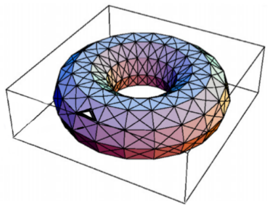
Theorem

Set-agreement is not solvable if the protocol complex is a pseudomanifold.

Herlihy, Kozlov, Rajsbaum (2013)

Benefits and drawbacks

✓ We can prove very general abstract results:



Theorem

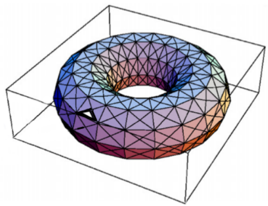
Set-agreement is not solvable if the protocol complex is a pseudomanifold.

Herlihy, Kozlov, Rajsbaum (2013)

✗ Are we still talking about distributed computing?

Benefits and drawbacks

✓ We can prove very general abstract results:



Theorem

Set-agreement is not solvable if the protocol complex is a pseudomanifold.

Herlihy, Kozlov, Rajsbaum (2013)

✗ Are we still talking about distributed computing?

Goal: Give a concrete meaning to “solving a task” using arbitrary objects, and prove that it agrees with the topological definition.

Outline

(1) Define a notion of concurrent object specification which is as general as possible. It should include non-linearizable objects.

Outline

(1) Define a notion of concurrent object specification which is as general as possible. It should include non-linearizable objects.

(2) Define an operational semantics for concurrent processes communicating through arbitrary shared objects.

Outline

(1) Define a notion of concurrent object specification which is as general as possible. It should include non-linearizable objects.

(2) Define an operational semantics for concurrent processes communicating through arbitrary shared objects.

(3) Define the protocol complex associated to a given protocol.

Outline

(1) Define a notion of concurrent object specification which is as general as possible. It should include non-linearizable objects.

(2) Define an operational semantics for concurrent processes communicating through arbitrary shared objects.

(3) Define the protocol complex associated to a given protocol.

(4) Prove the following:

Asynchronous Computability Theorem

A wait-free protocol *solves* a task if and only if there is a simplicial map from the protocol complex to the output complex which is carried by the task specification.

Specifying concurrent objects

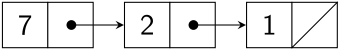
Getting rid of internal states

Example: how do we specify a list?

Getting rid of internal states

Example: how do we specify a list?

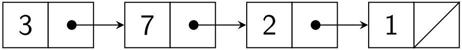
- Specify how each method modifies the internal state:



Getting rid of internal states

Example: how do we specify a list?

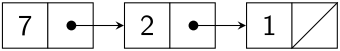
- Specify how each method modifies the internal state:
  - `push(3)`



Getting rid of internal states

Example: how do we specify a list?

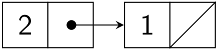
- Specify how each method modifies the internal state:
  - `push(3)`
  - `pop()` $\longrightarrow 3$



Getting rid of internal states

Example: how do we specify a list?

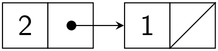
- Specify how each method modifies the internal state:
  - `push(3)`
  - `pop()` $\longrightarrow 3$
  - `pop()` $\longrightarrow 7$



Getting rid of internal states

Example: how do we specify a list?

- Specify how each method modifies the internal state:
  - push(3)
  - pop() $\longrightarrow 3$
  - pop() $\longrightarrow 7$

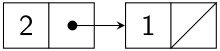


- List all the possible execution traces:

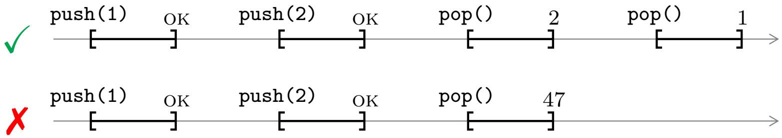
Getting rid of internal states

Example: how do we specify a list?

- Specify how each method modifies the internal state:
  - push(3)
  - pop() $\longrightarrow 3$
  - pop() $\longrightarrow 7$



- List all the possible execution traces:

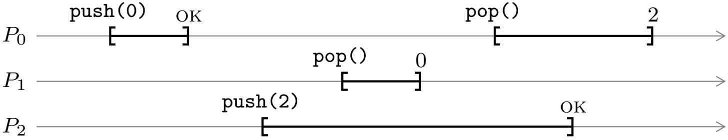


Concurrent specifications

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

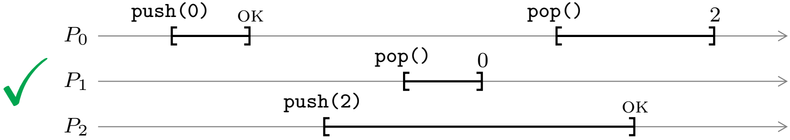
Concurrent specifications

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



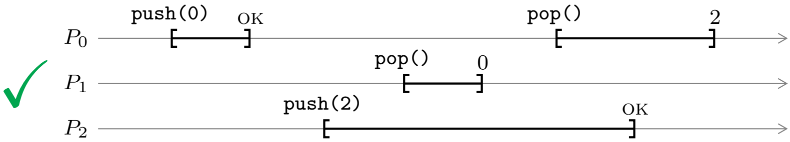
Concurrent specifications

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



Concurrent specifications

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

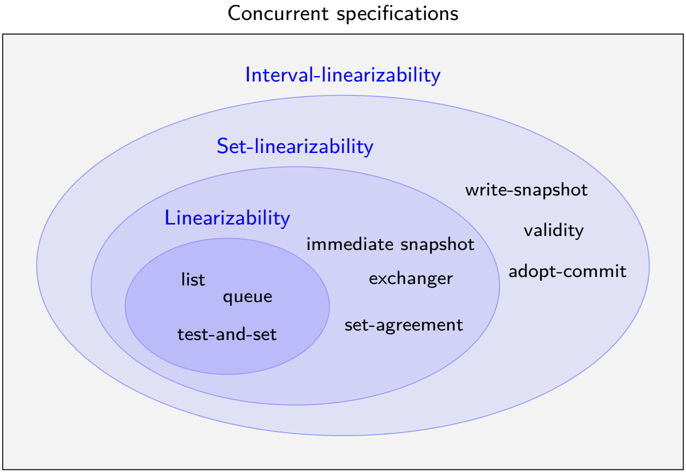


Write $\mathcal{T}$ for the set of all execution traces.

Definition

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

Concurrent specifications (2)



Concurrent Specifications Beyond Linearizability. Goubault, L., Mimram (OPODIS'18)

A computational model

Programs and Protocols

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

Programs and Protocols

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 an object,
- do local computations,
- return an output.

Formally: an infinite state machine.

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

Programs and Protocols

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 an object,
- do local computations,
- return an output.

Formally: an infinite state machine.

A protocol $(P_i)_{i \in [n]}$ consists of one program for each process.

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

Protocol semantics

P_0:

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

P_1:

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

Protocol semantics

P_0:

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

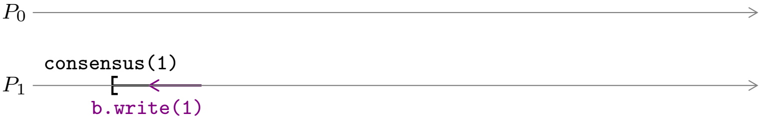
P_1:

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

Protocol semantics

P_0:
```
consensus(v) {
  a.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := b.read();
    return v';
}
```

P_1:
```
consensus(v) {
  b.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := a.read();
    return v';
}
```



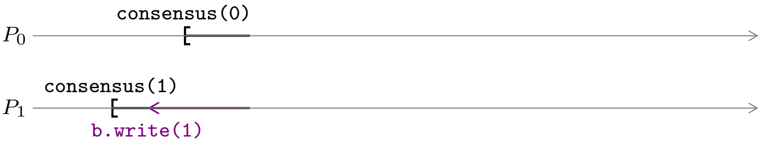
Protocol semantics

P_0:

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

P_1:

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



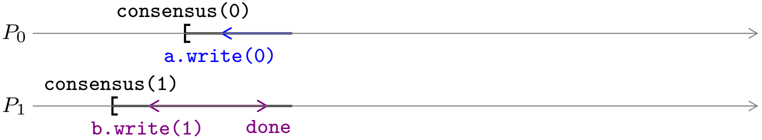
Protocol semantics

P_0:

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

P_1:

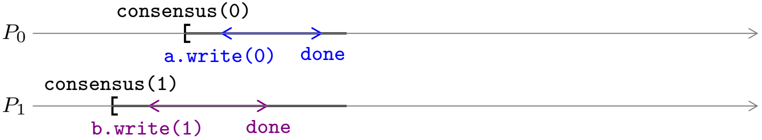
```
consensus(v) {
   b.write(v);
   x := t.test&set();
   if (x = 0)
      return v;
   else
      v' := a.read();
      return v';
}
```



Protocol semantics

P_0:
```
consensus(v) {
  a.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := b.read();
    return v';
}
```

P_1:
```
consensus(v) {
  b.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := a.read();
    return v';
}
```



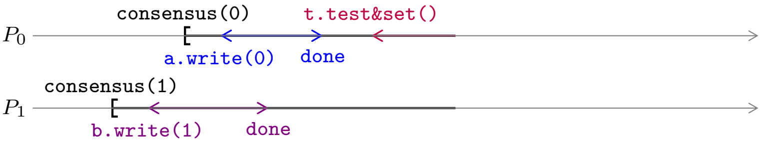
Protocol semantics

P_0:

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

P_1:

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



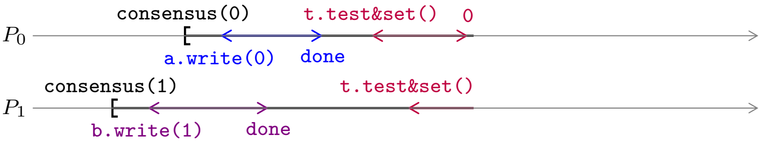
Protocol semantics

P_0:

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

P_1:

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



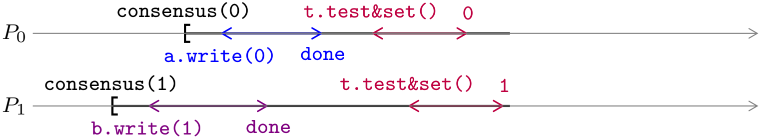
Protocol semantics

P_0:

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

P_1:

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



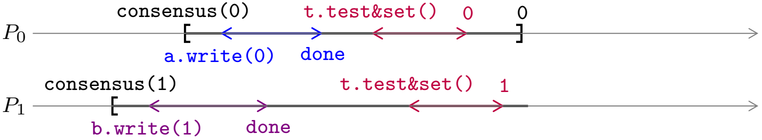
Protocol semantics

P_0:

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

P_1:

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



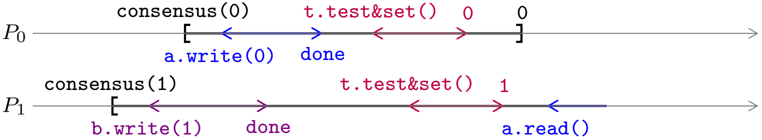
Protocol semantics

P_0:

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

P_1:

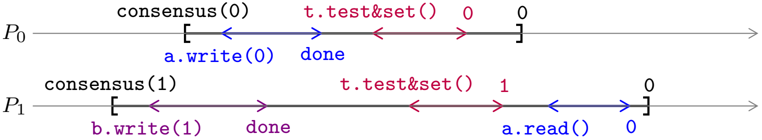
```
consensus(v) {
   b.write(v);
   x := t.test&set();
   if (x = 0)
      return v;
   else
      v' := a.read();
      return v';
}
```



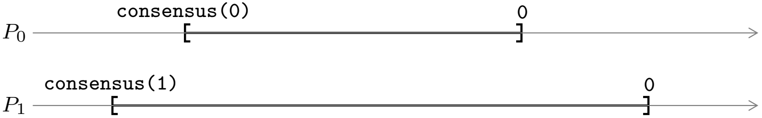
Protocol semantics

P_0:
```
consensus(v) {
  a.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := b.read();
    return v';
}
```

P_1:
```
consensus(v) {
  b.write(v);
  x := t.test&set();
  if (x = 0)
    return v;
  else
    v' := a.read();
    return v';
}
```

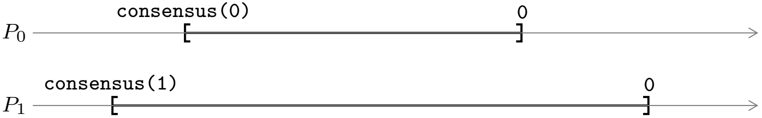


Protocol semantics (2)



The semantics $[\![\mathcal{P}]\!]$ of a protocol is the set of execution traces that can be produced by running the programs together.

Protocol semantics (2)

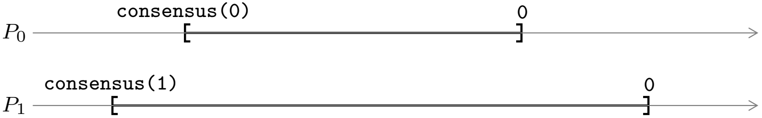


The semantics $[\![\mathcal{P}]\!]$ of a protocol is the set of execution traces that can be produced by running the programs together.

Theorem

For any protocol $\mathcal{P}$, $[\![\mathcal{P}]\!]$ is a concurrent specification.

Protocol semantics (2)



The semantics $[\![\mathcal{P}]\!]$ of a protocol is the set of execution traces that can be produced by running the programs together.

Theorem

For any protocol $\mathcal{P}$, $[\![\mathcal{P}]\!]$ is a concurrent specification.

The protocol $\mathcal{P}$ implements an object specification σ if $[\![\mathcal{P}]\!] \subseteq \sigma$.

Tasks vs Objects

A task for n processes is an input/output relation $\Theta \subseteq \mathcal{V}^n \times \mathcal{V}^n$.

Example: for consensus,

$$\Theta_{\text{consensus}} = \{((v_1, \ldots, v_n), (v_k, \ldots, v_k)) \mid k \in [n] \text{ and } v_1, \ldots, v_n \in \mathcal{V}\}$$

Tasks vs Objects

A task for n processes is an input/output relation $\Theta \subseteq \mathcal{V}^n \times \mathcal{V}^n$.

Example: for consensus,

$$\Theta_{\text{consensus}} = \{((v_1, \ldots, v_n), (v_k, \ldots, v_k)) \mid k \in [n] \text{ and } v_1, \ldots, v_n \in \mathcal{V}\}$$

Tasks are less expressive than objects:

- A task is one-shot (it can be used only once),

Tasks vs Objects

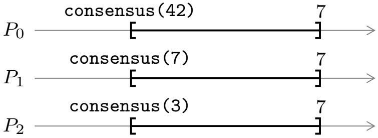
A task for n processes is an input/output relation $\Theta \subseteq \mathcal{V}^n \times \mathcal{V}^n$.

Example: for consensus,

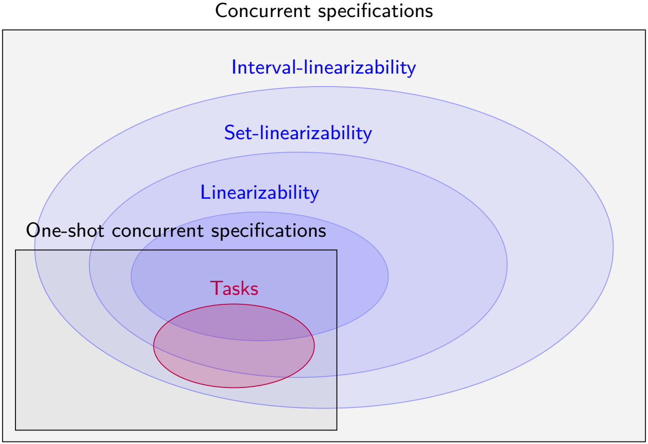
$$\Theta_{\text{consensus}} = \{((v_1, \ldots, v_n), (v_k, \ldots, v_k)) \mid k \in [n] \text{ and } v_1, \ldots, v_n \in \mathcal{V}\}$$

Tasks are less expressive than objects:

- A task is one-shot (it can be used only once),
- A task only specifies traces of the following form:



Tasks vs Objects (2)

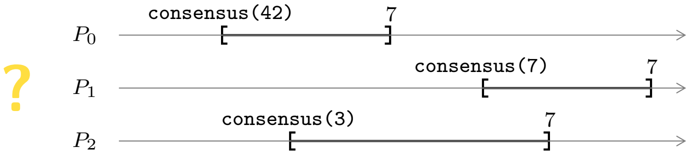


Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability.

Castañeda, Rajsbaum, Raynal (2018).

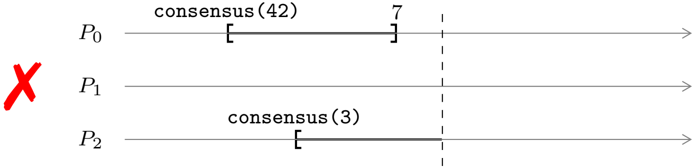
Turning a task into an object

How do we specify a consensus object?



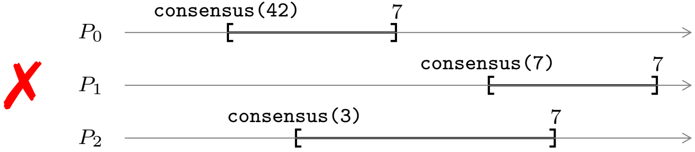
Turning a task into an object

How do we specify a consensus object?



Turning a task into an object

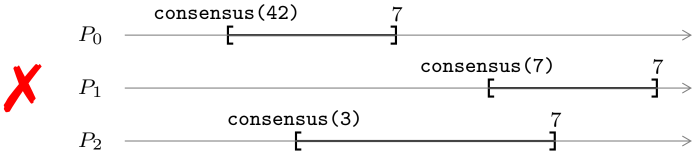
How do we specify a consensus object?



This defines a function G : Tasks $\rightarrow$ Objects.

Turning a task into an object

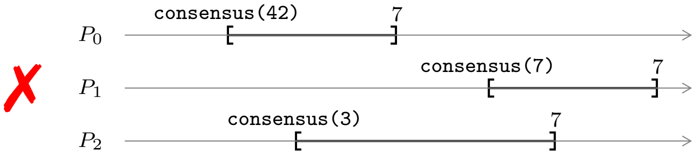
How do we specify a consensus object?



This defines a function G : Tasks $\to$ Objects.
There is also an obvious function F : Objects $\to$ Tasks.

Turning a task into an object

How do we specify a consensus object?



This defines a function G : Tasks $\to$ Objects.
There is also an obvious function F : Objects $\to$ Tasks.

Theorem

The functions F and G form a Galois connection:

$$\sigma \subseteq G(\Theta) \iff F(\sigma) \subseteq \Theta$$

The protocol complex

The notion of “view”

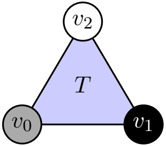
Informally, the view of a process at the end of an execution represents the *partial information* that it gathered.

The notion of “view”

Informally, the view of a process at the end of an execution represents the *partial information* that it gathered.

Example: for 3 processes.

- a trace T gives views (v_0, v_1, v_2).

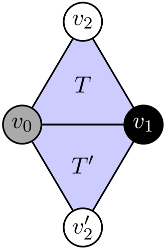


The notion of “view”

Informally, the view of a process at the end of an execution represents the *partial information* that it gathered.

Example: for 3 processes.

- a trace T gives views (v_0, v_1, v_2).
- a trace T' gives views (v_0, v_1, v_2').

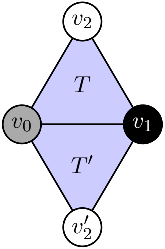


The notion of "view"

Informally, the view of a process at the end of an execution represents the *partial information* that it gathered.

Example: for 3 processes.

- a trace T gives views (v_0, v_1, v_2).
- a trace T' gives views (v_0, v_1, v_2').



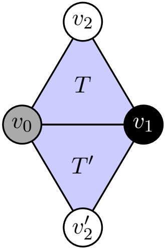
Putting all the possible executions together, we obtain the protocol complex.

The notion of “view”

Informally, the view of a process at the end of an execution represents the *partial information* that it gathered.

Example: for 3 processes.

- a trace T gives views (v_0, v_1, v_2).
- a trace T' gives views (v_0, v_1, v_2').



Putting all the possible executions together, we obtain the protocol complex.

Definition

The view of process P_i in a trace T is simply its final local state at the end of the execution.

Asynchronous Computability Theorem

for arbitrary objects

Let Θ be a task and $\mathcal{P}$ a wait-free protocol.

Theorem

The protocol $\mathcal{P}$ implements the object $G(\Theta)$ if and only if there exists a decision map from the protocol complex to the output complex which is carried by Θ.

Asynchronous Computability Theorem

for arbitrary objects

Let Θ be a task and $\mathcal{P}$ a wait-free protocol.

Theorem

The protocol $\mathcal{P}$ implements the object $G(\Theta)$ if and only if there exists a decision map from the protocol complex to the output complex which is carried by Θ.

- Not surprising: people have been using this for many years.

Asynchronous Computability Theorem

for arbitrary objects

Let Θ be a task and $\mathcal{P}$ a wait-free protocol.

Theorem

The protocol $\mathcal{P}$ implements the object $G(\Theta)$ if and only if there exists a decision map from the protocol complex to the output complex which is carried by Θ.

- Not surprising: people have been using this for many years.
- Benefits:
  - We have a clearly-defined setting in which it works

Asynchronous Computability Theorem

for arbitrary objects

Let Θ be a task and $\mathcal{P}$ a wait-free protocol.

Theorem

The protocol $\mathcal{P}$ implements the object $G(\Theta)$ if and only if there exists a decision map from the protocol complex to the output complex which is carried by Θ.

- Not surprising: people have been using this for many years.
- Benefits:
  - We have a clearly-defined setting in which it works
  - We studied the properties of concurrent specifications

Asynchronous Computability Theorem

for arbitrary objects

Let Θ be a task and $\mathcal{P}$ a wait-free protocol.

Theorem

The protocol $\mathcal{P}$ implements the object $G(\Theta)$ if and only if there exists a decision map from the protocol complex to the output complex which is carried by Θ.

- Not surprising: people have been using this for many years.
- Benefits:
  - We have a clearly-defined setting in which it works
  - We studied the properties of concurrent specifications
  - We understand better the difference between tasks and objects

Future work

We can still generalize this theorem a bit more:

- ACT for t-resilient protocols using arbitrary objects

Future work

We can still generalize this theorem a bit more:

- ACT for t-resilient protocols using arbitrary objects
- ACT for synchronous computation

Future work

We can still generalize this theorem a bit more:

- ACT for t-resilient protocols using arbitrary objects
- ACT for synchronous computation
- ACT for stronger notions of tasks (Castañeda et al.)
    - *Refined tasks*
    - *Long-lived tasks*

Future work

We can still generalize this theorem a bit more:

- ACT for t-resilient protocols using arbitrary objects
- ACT for synchronous computation
- ACT for stronger notions of tasks (Castañeda et al.)
  - *Refined tasks*
  - *Long-lived tasks*

Study the compositionality of protocols.

- Links with game semantics

Future work

We can still generalize this theorem a bit more:

- ACT for t-resilient protocols using arbitrary objects
- ACT for synchronous computation
- ACT for stronger notions of tasks (Castañeda et al.)
  - *Refined tasks*
  - *Long-lived tasks*

Study the compositionality of protocols.

- Links with game semantics
- Can we build the protocol complex modularly?

Thanks!