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 $\Theta$ using the objects $A_1, \ldots, A_n$?
A topological approach

Herlihy and Shavit, 1999
2004 Gödel prize
A topological approach

**Theorem 3.1 (Asynchronous Computability Theorem).** A decision task \((\mathcal{F}, \mathcal{C}, \Delta)\) has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision \(\sigma\) of \(\mathcal{F}\) and a color-preserving simplicial map

\[ \mu : \sigma(\mathcal{F}) \to \mathcal{C} \]

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

Herlihy and Shavit, 1999
2004 Gödel prize

Herlihy, Kozlov, Rajsbaum, 2013
Asynchronous Computability Theorem

Input complex

Protocol complex

Subdivision

Output complex

Task specification

Decision

∃
Asynchronous Computability Theorem

Input complex

Task specification

Output complex
Asynchronous Computability Theorem

Protocol complex

Output complex

Subdivision

Task specification

Input complex
Asynchronous Computability Theorem

Input complex

Protocol complex

Subdivision

Decision

Output complex

Task specification

Input complex
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 Theorems for t-resilient systems, Saraph, Herlihy, Gafni (DISC 2016).

▶ we use other objects instead of read/write registers?

→ This talk.
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”?
  - → *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 \textit{wait-free} protocol using \textit{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”?
  \rightarrow \textit{Asynchronous Computability Theorems for $t$-resilient systems}, Saraph, Herlihy, Gafni (DISC 2016).

- we use other objects instead of read/write registers?
  \rightarrow \text{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

Protocol complex

Input complex

Output complex

Task specification

Protocol specification

Decision: $\exists \ ?$
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.
(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.

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.
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?

- Specify how each method modifies the internal state:
  - `push(3)`
  - `pop()`
    - → `3`
  - `pop()`
    - → `7`
Getting rid of internal states

**Example:** how do we specify a list?
- Specify how each method modifies the internal state:

```
push(3)
pop() → 3
pop() → 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)

![Diagram showing list operations]

1 2 7 3
Getting rid of internal states

Example: how do we specify a list?

- Specify how each method modifies the internal state:
  - `push(3)`
  - `pop()` $\rightarrow$ 3
Getting rid of internal states

**Example:** how do we specify a list?

- Specify how each method modifies the internal state:
  - push(3)
  - pop() → 3
  - pop() → 7
Getting rid of internal states

**Example:** how do we specify a list?

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

![Diagram](image)

- List all the possible execution traces:

![Execution traces](image)
Getting rid of internal states

Example: how do we specify a list?

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

- List all the possible execution traces:

  ![Execution traces diagram]

  - push(1) OK push(2) OK pop() 2 pop() 1
  - push(1) OK push(2) OK pop() 47
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).

- **P_0**
  - push(0) \[ \text{OK} \]
  - pop() \[ \text{2} \]

- **P_1**
  - pop() \[ \text{0} \]
  - push(2)

- **P_2**
  - push(2) \[ \text{OK} \]
**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

Interval-linearizability

Set-linearizability

Linearizability

- list
- queue
- test-and-set

write-snapshot
validity
adopt-commit

immediate snapshot
exchanger
set-agreement

Concurrent Specifications Beyond Linearizability. Goubault, L., Mimram (OPODIS’18)
A computational model
We fix a set \( \{A_1, \ldots, A_k\} \) of shared objects, along with their concurrent specifications.
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.

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

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

\hspace{1cm} P_0: \hspace{1cm} \texttt{consensus}(v) \{ \\
\hspace{1cm} \hspace{1cm} \texttt{a.write}(v); \\
\hspace{1cm} \hspace{1cm} x := \texttt{t.test\&set}(); \\
\hspace{1cm} \hspace{1cm} \text{if } (x = 0) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \texttt{return } v; \\
\hspace{1cm} \hspace{1cm} \text{else} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} v' := \texttt{b.read}(); \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \texttt{return } v'; \\
\hspace{1cm} \} \\

\hspace{1cm} P_1: \hspace{1cm} \texttt{consensus}(v) \{ \\
\hspace{1cm} \hspace{1cm} \texttt{b.write}(v); \\
\hspace{1cm} \hspace{1cm} x := \texttt{t.test\&set}(); \\
\hspace{1cm} \hspace{1cm} \text{if } (x = 0) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \texttt{return } v; \\
\hspace{1cm} \hspace{1cm} \text{else} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} v' := \texttt{a.read}(); \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \texttt{return } v'; \\
\hspace{1cm} \}
Protocol semantics

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

\[
P_1: \quad \text{consensus}(v) \{ \\
\quad \text{b.write}(v); \\
\quad x := \text{t.test\&set}(); \\
\quad \text{if } (x = 0) \\
\quad \quad \text{return } v; \\
\quad \text{else} \\
\quad \quad v' := \text{a.read}(); \\
\quad \quad \text{return } v'; \\
\}
\]
Protocol semantics

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

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

\[ P_0 \longrightarrow \]

\[ P_1 \leftarrow \text{b.write}(1) \longrightarrow \]
Protocol semantics

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

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

\[ \begin{align*}
P_0: & \quad \text{consensus}(0) \\
& \quad \text{b.write}(1) \\
& \quad \text{consensus}(1) \\
& \end{align*} \]
Protocol semantics

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

\[ P_1: \]
\[
\text{consensus}(v) \{ \\
    \text{b.write}(v); \\
    x := \text{t.test\&set}(); \\
    \text{if } (x = 0) \\
    \quad \text{return } v; \\
    \text{else} \\
    \quad v' := \text{a.read}(); \\
    \quad \text{return } v'; \\
\}
\]
Protocol semantics

$P_0$:  
\[
\text{consensus}(v) \{ \\
    \text{a.write}(v); \\
    x := \text{t.test\&set}(); \\
    \text{if } (x = 0) \\
    \quad \text{return } v; \\
    \text{else} \\
    \quad v' := \text{b.read}(); \\
    \quad \text{return } v'; \\
\}
\]

$P_1$:  
\[
\text{consensus}(v) \{ \\
    \text{b.write}(v); \\
    x := \text{t.test\&set}(); \\
    \text{if } (x = 0) \\
    \quad \text{return } v; \\
    \text{else} \\
    \quad v' := \text{a.read}(); \\
    \quad \text{return } v'; \\
\}
\]

$P_0$ consensus(0)

$P_1$ consensus(1)

$P_0$ a.write(0) done

$P_1$ b.write(1) done
Protocol semantics

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

\[ P_1: \]
\[
\text{consensus}(v) \{ \\
    \text{b.write}(v) ; \\
    x := \text{t.test\&set}(); \\
    \text{if } (x = 0) \\
    \quad \text{return } v; \\
    \text{else} \\
    \quad v' := \text{a.read}(); \\
    \quad \text{return } v'; \\
\}
\]
Protocol semantics

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

\[ P_1: \]
\[
\text{consensus}(v) \{ \\
  \text{b.write}(v); \\
  x := \text{t.test\&set}(); \\
  \text{if} \ (x = 0) \\
  \quad \text{return} \ v; \\
  \text{else} \\
  \quad v' := \text{a.read}(); \\
  \quad \text{return} \ v'; \\
\}\]
Protocol semantics

**$P_0$:**

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

**$P_1$:**

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

\begin{align*}
P_0: & \quad \text{\texttt{consensus}}(v) \{ \\
& \quad \text{\texttt{a.write}}(v); \\
& \quad x := \text{\texttt{t.test}} & \text{\& set}(); \\
& \quad \text{if (} x = 0 \text{)} \\
& \quad \quad \text{return } v; \\
& \quad \text{else} \\
& \quad \quad v' := \text{\texttt{b.read}}(); \\
& \quad \quad \text{return } v'; \\
& \}\nonumber
\end{align*}

\begin{align*}
P_1: & \quad \text{\texttt{consensus}}(v) \{ \\
& \quad \text{\texttt{b.write}}(v); \\
& \quad x := \text{\texttt{t.test}} & \text{\& set}(); \\
& \quad \text{if (} x = 0 \text{)} \\
& \quad \quad \text{return } v; \\
& \quad \text{else} \\
& \quad \quad v' := \text{\texttt{a.read}}(); \\
& \quad \quad \text{return } v'; \\
& \}\nonumber
\end{align*}
Protocol semantics

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

\[ P_1: \]
\[
\text{consensus}(v) \{ \\
\quad \text{b.write}(v); \\
\quad x := \text{t.test\&set}(); \\
\quad \text{if } (x = 0) \\
\quad \quad \text{return } v; \\
\quad \text{else} \\
\quad \quad v' := \text{a.read}(); \\
\quad \quad \text{return } v'; \\
\}
\]
Protocol semantics

\begin{align*}
P_0: & \quad \text{consensus}(v) \{ \\
& \hspace{1em} \text{a.write}(v); \\
& \hspace{1em} x := \texttt{t.test&set}(); \\
& \hspace{1em} \text{if } (x = 0) \\
& \hspace{2em} \text{return } v; \\
& \hspace{1em} \text{else} \\
& \hspace{2em} v' := \texttt{b.read}(); \\
& \hspace{2em} \text{return } v'; \\
& \}
\end{align*}

\begin{align*}
P_1: & \quad \text{consensus}(v) \{ \\
& \hspace{1em} \text{b.write}(v); \\
& \hspace{1em} x := \texttt{t.test&set}(); \\
& \hspace{1em} \text{if } (x = 0) \\
& \hspace{2em} \text{return } v; \\
& \hspace{1em} \text{else} \\
& \hspace{2em} v' := \texttt{a.read}(); \\
& \hspace{2em} \text{return } v'; \\
& \}
\end{align*}
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 $\llbracket P \rrbracket$ of a protocol is the set of execution traces that can be produced by running the programs together.

**Theorem**

*For any protocol $P$, $\llbracket P \rrbracket$ is a concurrent specification.*
The semantics $\semantics{P}$ of a protocol is the set of execution traces that can be produced by running the programs together.

**Theorem**

For any protocol $P$, $\semantics{P}$ is a concurrent specification.

The protocol $P$ implements an object specification $\sigma$ if $\semantics{P} \subseteq \sigma$. 
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,

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

Tasks are less expressive than objects:
- A task is one-shot (it can be used only once),
A task 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:

\[
\begin{align*}
P_0 \quad \text{consensus}(42) \quad 7 \\
P_1 \quad \text{consensus}(7) \quad 7 \\
P_2 \quad \text{consensus}(3) \quad 7
\end{align*}
\]
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?

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

There is also an obvious function $F: \text{Objects} \rightarrow \text{Tasks}$.

**Theorem**

The functions $F$ and $G$ form a Galois connection:

$$\sigma \subseteq G(\Theta) \iff F(\sigma) \subseteq \Theta$$
Turning a task into an object

How do we specify a consensus object?

This defines a function $G : \text{Tasks} \rightarrow \text{Objects}$. 
Turning a task into an object

How do we specify a consensus object?

This defines a function $G: \text{Tasks} \rightarrow \text{Objects}$.
There is also an obvious function $F: \text{Objects} \rightarrow \text{Tasks}$.
Turning a task into an object

How do we specify a consensus object?

This defines a function $G : \text{Tasks} \rightarrow \text{Objects}$.
There is also an obvious function $F : \text{Objects} \rightarrow \text{Tasks}$.

**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 $(v_0, v_1, v_2)$. 

![Diagram of views](image)
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 $(v_0, v_1, v'_2)$. 

![Diagram showing the view of a process](image)
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 $\Theta$ 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 $\Theta$. 
Asynchronous Computability Theorem
for arbitrary objects

Let $\Theta$ 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 $\Theta$.

- Not surprising: people have been using this for many years.
Asynchronous Computability Theorem
for arbitrary objects

Let $\Theta$ 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 $\Theta$.

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

**Theorem**

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

- 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 \( \Theta \) be a task and \( P \) a wait-free protocol.

**Theorem**

The protocol \( P \) 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 \( \Theta \).

- 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

Refined tasks
- Long-lived tasks
- Study the compositionality of protocols.

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