A Sound Foundation for the Topological Approach to Task Solvability

Jérémy Ledent
École Polytechnique, Palaiseau, France
jeremy.ledent@lix.polytechnique.fr

Samuel Mimram
École Polytechnique, Palaiseau, France
samuel.mimram@lix.polytechnique.fr

Abstract

The area of fault-tolerant distributed computability is concerned with the solvability of decision tasks in various computational models where the processes might crash. A very successful approach to prove impossibility results in this context is that of combinatorial topology, started by Herlihy and Shavit's paper in 1999. They proved that, for wait-free protocols where the processes communicate through read/write registers, a task is solvable if and only if there exists some map between simplicial complexes satisfying some properties. This approach was then extended to many different contexts, where the processes have access to various synchronization and communication primitives. Usually, in those cases, the existence of a simplicial map from the protocol complex to the output complex is taken as the definition of what it means to solve a task. In particular, no proof is provided of the fact that this abstract topological definition agrees with a more concrete operational definition of task solvability. In this paper, we bridge this gap by proving a version of Herlihy and Shavit's theorem that applies to any kind of object. First, we start with a very general way of specifying concurrent objects, and we define what it means to implement an object B in a computational model where the processes are allowed to communicate through shared objects $A_1, \dots, A_k$. Then, we derive the notion of a decision task as a special case of concurrent object. Finally, we prove an analogue of Herlihy and Shavit's theorem in this context. In particular, our version of the theorem subsumes all the uses of the combinatorial topology approach that we are aware of.

2012 ACM Subject Classification Concurrency

Keywords and phrases Fault-tolerant protocols, Asynchronous computability, Combinatorial topology, Protocol complex, Distributed task

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2019.30

1 Introduction

The typical framework in which one studies distributed computing is that of asynchronous processes communicating through shared objects. A wide variety of computational models have been introduced, depending on the communication primitives the processes are allowed to use, and the assumptions made on the kind of failures that might happen during the computation. The area of fault-tolerant computability [6] studies what kind of *decision tasks* can be solved in such models. To solve a task, each process starts with a private input value, and after communicating with the other processes, it has to decide on an output. For instance, a well-known task is *consensus*, where the processes must agree on one of their inputs.

A very well-studied setting is the one of wait-free protocols communicating through shared read/write registers. In order to prove impossibility results in this context, people started developing powerful mathematical tools based on algebraic topology [1, 13]. The fundamental paper by Herlihy and Shavit [10] provides a topological characterization of the tasks that can be solved by a wait-free protocol using read/write registers: they proved the *asynchronous*

© Jérémy Ledent and Samuel Mimram;
licensed under Creative Commons License CC-BY

computability theorem, which says that a read/write protocol solves a decision task if and only if there exists a map from a subdivision of the input complex to the output complex, which is *carried* by the task specification. The subdivided simplicial complex that appears in this theorem was the first occurrence of a *protocol complex*, a very compact way of expressing combinatorially the partial knowledge that each process acquires during the computation.

Soon thereafter, it became clear that this method actually extends beyond the setting of read/write registers: many other computational models can also be described as protocol complexes. This idea gave birth to the *combinatorial topology* approach to distributed computing [6]. Generalizing the ideas behind the asynchronous computability theorem, one can define an abstract notion of protocol using so-called *carrier maps* between an input complex and a protocol complex. Then, we can take Herlihy and Shavit's characterization as the *definition* of what it means for a protocol to solve a task. This abstract approach is very appealing because it offers a great deal of generality: for example, the set-agreement task cannot be solved in any computational model whose protocol complex is a pseudomanifold [6, Chapter 9]. We are thus able to prove impossibility results for a wide class of computational models, instead of studying them one at a time. In principle, most computational models and synchronization primitives can be formulated using the protocol complex formalism: it has been used in the literature to model processes communicating using test-and-set objects [8], (N, k)-consensus objects [7], weak symmetry breaking and renaming [4], or various synchronous or asynchronous message passing primitives [9].

The drawback of this high level of abstraction is that we have to *trust* that our protocol complex faithfully represents the behavior of the objects that we want to model. When we want to define, for example, the protocol complex which is supposed to model test-and-set computation, we must define it carefully so that the simplicial notion of "solving" a task will agree with the concrete operational semantics of test-and-set objects. Ideally, one would have to prove another version of the asynchronous computability theorem for test-and-set protocols, relating a concrete notion of solvability with the abstract simplicial definition. This is usually not done in practice, since it is often clear intuitively that the proposed notion of protocol complex makes sense operationally.

In this paper, we generalize the asynchronous computability theorem to a large class of concurrent objects. In Section 2, we define the "concrete" notion of solvability that we will later prove equivalent to the simplicial one. Our goal is to model processes which communicate though arbitrary shared objects. In particular, our notion of object is general enough to include all the objects mentioned above. Our notion of object specification is discussed more thoroughly in [5] where, in particular, we prove that it is equivalent to specifications based on interval-linearizability [2]. In Section 3, we discuss the notion of task and compare it to the notion of one-shot object. It had been remarked recently that tasks are less expressive than one-shot objects [2]. We explore more in depth the relationship between the two, and as a result we obtain a characterization of the one-shot objects that correspond to tasks. Finally, in Section 4, we state and prove our generalized asynchronous computability theorem. It encompasses the original theorem of Herlihy and Shavit for read/write registers, as well as all the other variants that have been implicitly used in the literature.

2 A computational model based on trace semantics

In this section, we define a concrete semantics for protocols where the processes communicate through shared objects. What we mean here by "concrete" is that it is based on interleavings of execution traces, as opposed to the abstract topological semantics of Section 4. The

formalism that we use here (i.e., all of Section 2) was developed and discussed more thoroughly in a previous article [5]. It is quite similar to other trace-based operational semantics found in the literature [12, 3]. There are mainly two things that we need to define: in Section 2.1, we give a very general way of specifying concurrent objects, and in Section 2.2, we define what it means to implement a new object B, assuming we have access to objects $A_1, \dots, A_k$.

2.1 Specifying concurrent objects

Our goal here is to define a notion of concurrent object which includes all the objects that are relevant to fault-tolerant computability. As it was remarked in [2], the tasks studied in this field cannot be specified using common techniques such as linerizability [11]. Thus, they introduced the notion of *interval-linearizability* in order to unify tasks and objects. The definition that we use here was introduced in [5] and shown to be equivalent to interval-linearizability. In the rest of the paper, we suppose fixed a number $n \in \mathbb{N}$ of *processes* and write $[n] = \{0, 1, \dots, n-1\}$ for the set of *process names* (a process is identified by its number). We also suppose fixed a set $\mathcal{V}$ of *values* which can be exchanged between processes and objects (typically $\mathcal{V} = \mathbb{N}$). The set $\mathcal{A}$ of possible *actions* for an object is defined as

$$\mathcal{A} \quad = \quad \{\mathsf{i}_i^x \mid i \in [n], x \in \mathcal{V}\} \cup \{\mathsf{r}_i^y \mid i \in [n], y \in \mathcal{V}\}$$

An action is thus either
- i_i^x: an *invocation* of the object by the process i with input value x,
- r_i^y: a *response* of the object to the process i with output value y.

An *execution trace* is a finite sequence of actions, i.e., an element of $\mathcal{A}^*$; we write ε for the empty trace and $T \cdot T'$ for the concatenation of two traces T and T'. Given a process $i \in [n]$, the i-th *projection* $\pi_i(T)$ of a trace $T \in \mathcal{T}$ is the trace obtained from T by removing all the actions of the form i_j^x or r_j^x with $j \neq i$. A trace $T \in \mathcal{T}$ is *alternating* if for all $i \in [n]$, $\pi_i(T)$ is either empty or it begins with an invocation and alternates between invocations and responses, i.e., using the traditional notation of regular expressions:

$$\pi_i(T) \quad \in \quad \Big(\bigcup_{x,y \in \mathcal{V}} \mathsf{i}_i^x \cdot \mathsf{r}_i^y\Big)^* \cdot \Big(\bigcup_{x \in \mathcal{V}} \mathsf{i}_i^x + \varepsilon\Big)$$

We write $\mathcal{T} \subseteq \mathcal{A}^*$ for the set of alternating traces. In the remaining of the paper, we will only consider alternating traces, and often drop the adjective "alternating". If $\pi_i(T)$ ends with an invocation, we call it a *pending* invocation. An alternating trace T is *complete* if it does not have any pending invocation.

▶ **Definition 1.** *A* concurrent specification σ *is a subset of* $\mathcal{T}$ *which is*
(1) prefix-closed*: if* $T \cdot T' \in \sigma$ *then* $T \in \sigma$*,*
(2) non-empty*:* $\varepsilon \in \sigma$*,*
(3) receptive*: if* $T \in \sigma$ *and* $\pi_i(T)$ *has no pending invocation, then* $T \cdot \mathsf{i}_i^x \in \sigma$ *for every* $x \in \mathcal{V}$*,*
(4) total*: if* $T \in \sigma$ *and* $\pi_i(T)$ *has a pending invocation, there exists* $x \in \mathcal{V}$ *such that* $T \cdot \mathsf{r}_i^x \in \sigma$*,*
(5) *is closed under* expansion*:*
- *if* $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T' \in \sigma$ *where* a_j *is an action of process* $j \neq i$*, then* $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T' \in \sigma$*.*
- *if* $T \cdot \mathsf{r}_i^y \cdot a_j \cdot T' \in \sigma$ *where* a_j *is an action of process* $j \neq i$*, then* $T \cdot a_j \cdot \mathsf{r}_i^y \cdot T' \in \sigma$*.*

We write CSpec *for the set of concurrent specifications.*

A concurrent specification σ is the set of all executions that we consider acceptable: a protocol *implements* the specification σ if all the execution traces that it generates belong to σ (this will be detailed in Section 2.2). The axioms (1-3) are quite natural and commonly

considered in the literature (e.g. in [11]). They can be read as follows: (1) an object can do one action at a time, (2) an object can do nothing, (3) it is always possible to invoke an object. The axiom (4) states that objects always answer and in a non-blocking way; this is a less fundamental axiom and more of a design choice, since we want to model wait-free computation. Note that receptivity (3) does not force objects to accept all inputs: we could have a distinguished "error" value which is returned in case of an invalid input. Similarly, an object with several inputs, or several interacting methods (for example, a stack) can be modeled by choosing a suitable set of values $\mathcal{V}$.

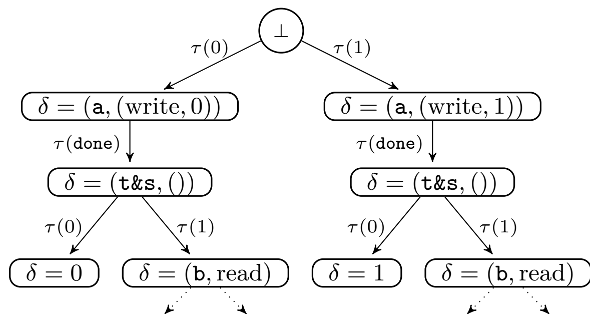
The condition (5) might seem more surprising, and will be crucial to establish the correspondence between tasks and objects in Section 3. Some consequences of this condition, and the reasons why it is a desirable property of concurrent specifications, are discussed more thoroughly in [5]. Intuitively, this condition says that if a given execution is correct, then a similar execution where some process is idle for a while just after invoking, or just before responding, must also be correct, since both executions are indistinguishable. In particular, condition (5) implies that invocations "commute": we have $T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in \sigma$ *iff* $T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$, and similarly for responses. We say that two traces $T, T' \in \mathcal{T}$ are *equivalent*, written $T \equiv T'$, if one is obtained from the other by reordering the actions within each block of consecutive invocations or consecutive responses. Generally, in the rest of the paper, we are only interested in studying traces up to equivalence.

2.2 Program semantics

In this section, we provide an operational model for concurrent programs communicating through shared objects. We assume given a set Obj of objects: they might be, for instance, concurrent data structures that have already been implemented, and that our programs are able to use in order to compute and communicate. We do not want to depend on a particular implementation of these objects, but on their specification. Thus, each object comes with its concurrent specification (as in Definition 1), which is the set of behaviors that it might exhibit. Note that our model does not have any special construct for reading and writing in the shared memory: we assume that the memory itself is given as an object in Obj, with an appropriate specification. Thus, the only meaningful action a program can take is to call one of the objects; and possibly do some local computation to determine what the next call should be.

To abstract away the syntax of the programming language, we use an automata-like representation, which roughly corresponds to the control-flow graph of the program.

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



The example above shows an implementation of binary consensus among two processes, using three objects: two read-write registers `a` and `b`, and a test-and-set object `t&s`. In a given state of the automaton, the decision function δ indicates which is the next object

that the program will call, and the transition function τ says what the next state will be depending on the return value of the call. The pseudo-code and automaton above represent the program run by one of the two processes, the other process should switch the role of `a` and `b`.

We suppose fixed a set Obj of objects, along with their concurrent specification $\mathsf{spec}(o) \in \mathsf{CSpec}$ for each $o \in \mathsf{Obj}$. Here, a program is basically a piece of code (executed by one of the processes) which takes a value as input, makes several calls to the objects in Obj (using algebraic operations to combine their results) and finally returns a value. Formally,

▶ **Definition 2.** *A* program *is a quadruple* $(Q, \bot, \delta, \tau)$ *consisting of:*

- *a (possibly infinite) set Q of* local states *containing an* idle state $\bot \in Q$*,*
- *a* decision function $\delta : Q \setminus \{\bot\} \to (\mathsf{Obj} \times \mathcal{V}) \sqcup \mathcal{V}$*,*
- *a* transition function $\tau : Q \times \mathcal{V} \to Q \setminus \{\bot\}$*.*

The idle state is the one where the program is waiting to be called with some input value x, in which case it will go to state $\tau(\bot, x)$. After that, the decision function gives the next step of the process depending on the current state: either call some object with some input value, or terminate and output some value. In the case where an object is called, the transition function gives the new local state of the process, depending on the previous state and the value returned by the object.

A *protocol* P is given by a program $P_i = (Q_i, \bot_i, \delta_i, \tau_i)$ for each process $i \in [n]$. The *global state* of a protocol P is an element $q = (q_0, \dots, q_{n-1})$ of $\overline{Q} = \prod_i Q_i$, consisting of a state for each process P_i. The initial state is $q_{\text{init}} = (\bot_0, \dots, \bot_{n-1})$ and, given a global state q, we write $q[i \leftarrow q'_i]$ for the state where the i-th component q_i has been replaced by q'_i. We now describe the set $\mathcal{A}$ of possible actions for P, as well as their effect $\Delta : \overline{Q} \times \mathcal{A} \to \overline{Q}$ on global states.

- i_i^x: the i-th process is called with input value $x \in \mathcal{V}$. The local state q_i of process i is changed from $\bot_i$ to $\tau_i(\bot_i, x)$:

  $$\Delta(q, \mathsf{i}_i^x) \quad = \quad q[i \leftarrow \tau_i(\bot_i, x)]$$

  where the state on the right is q where q_i has been replaced by $\tau_i(\bot_i, x)$.
- $\mathsf{i}(o)_i^x$: the i-th process invokes the object $o \in \mathsf{Obj}$ with input value $x \in \mathcal{V}$. This does not have any effect on the global state:

  $$\Delta(q, \mathsf{i}(o)_i^x) \quad = \quad q$$

- $\mathsf{r}(o)_i^x$: the object $o \in \mathsf{Obj}$ returns some output value $x \in \mathcal{V}$ to the i-th process. The local state of process i is updated according to its transition function τ_i:

  $$\Delta(q, \mathsf{r}(o)_i^x) \quad = \quad q[i \leftarrow \tau_i(q_i, x)]$$

- r_i^x: the i-th process has finished computing, returning the output value $x \in \mathcal{V}$. It goes back to idle state:

  $$\Delta(q, \mathsf{r}_i^x) \quad = \quad q[i \leftarrow \bot_i]$$

The actions of the form i_i^x and r_i^x (resp. $\mathsf{i}(o)_i^x$ and $\mathsf{r}(o)_i^x$) are called *outer* (resp. *inner*) *actions*. Given a trace $T \in \mathcal{A}^*$ and an object o, we denote by T_o, called the *inner projection on* o, the trace obtained from T by keeping only the inner actions of the form $\mathsf{i}(o)_i^x$ or $\mathsf{r}(o)_i^y$. The function Δ is extended as expected as a function $\Delta : \overline{Q} \times \mathcal{A}^* \to \overline{Q}$, i.e., $\Delta(q, T \cdot T') = \Delta(\Delta(q, T), T')$ and $\Delta(q, \varepsilon) = q$. A trace is valid if at each step in the execution, the next action is taken according to the decision function δ. Formally:

▶ **Definition 3.** *A trace $T \in \mathcal{A}^*$ is* valid *when for every strict prefix U of T, writing $T = U \cdot a \cdot V$ and $q = \Delta(q_{\text{init}}, U)$, with $a \in \mathcal{A}$, we have:*

- *if $a = \mathsf{i}_i^x$ then $q_i = \bot_i$,*
- *if $a = \mathsf{r}_i^y$ then $q_i \neq \bot_i$ and $\delta_i(q_i) = y$,*
- *if $a = \mathsf{i}(o)_i^x$ then $q_i \neq \bot_i$ and $\delta_i(q_i) = (o, x)$,*
- *if $a = \mathsf{r}(o)_i^y$ then $q_i \neq \bot_i$.*

Moreover, we require that for every object $o \in \mathsf{Obj}$, the inner projection T_o belongs to $\mathsf{spec}(o)$. The set of valid traces for P is written $\mathcal{T}_P \subseteq \mathcal{A}^*$.

A protocol P is *wait-free* if there is no valid infinite trace (i.e., all its prefixes are valid) involving only inner-i-actions after some position: in other words, a process running alone will eventually decide an output value. Given a trace $T \in \mathcal{A}^*$, we write $\pi(T)$ for the trace obtained by keeping only outer actions.

▶ **Definition 4.** *The* semantics *of a protocol P is the set of traces $[\![P]\!] = \{\pi(T) \mid T \in \mathcal{T}_P\}$ and P* implements *a concurrent specification σ whenever $[\![P]\!] \subseteq \sigma$, i.e., all the outer traces that P can produce are correct with respect to σ.*

An important property that was proved in [5] is that $[\![P]\!]$ itself is a concurrent specification in the sense of Definition 1. Indeed, given any protocol P, we should be able to specify abstractly what P is doing: that specification is precisely $[\![P]\!]$. In particular, since $[\![P]\!]$ satisfies all the axioms of concurrent specifications, we are sure that these axioms are reasonable, in the sense that they are validated in our model.

3 Tasks as a particular kind of one-shot objects

The topological approach to distributed computing [6] is not interested in implementing long-lived objects as in the previous section, but in solving *decision tasks*. In a decision task, each process starts with a private input value, it communicates with the other processes by using some shared objects, and then it must eventually decide an output value. The most well-known example is the *consensus* task, where the processes have to agree on a common output value. Thus, in a decision task, all the processes start the computation together, and once a process has decided an output value, it does not take part in the computation anymore. This is contrasted with concurrent objects, where new processes can start and terminate at any point during the computation, and a single process is allowed to make several consecutive calls to the object.

In this section, we compare tasks and one-shot objects, that is, objects which can be called only once by each process. It was already observed in [2] that tasks are weaker than one-shot objects: some objects cannot be expressed as a task. They defined a notion of "extended task" which is as expressive as one-shot objects. Our goal here is dual: we want to characterize the subclass of one-shot objects which correspond to tasks.

We start by giving a formal definition of tasks. The formalism that we use here is inspired of the one used in Herlihy and Shavit's paper [10]. Readers familiar with the topological definition of a task should recognize that this a direct reformulation of it. Recall that the set of values is written $\mathcal{V}$ and n is the number of processes. Let $\bot$ be a fresh symbol which is not in $\mathcal{V}$. A *vector* is a tuple $U \in (\mathcal{V} \cup \{\bot\})^n$, such that at least one component of U is not $\bot$. We write U_i for the i-th component of U, and $U[i \leftarrow x]$ for the vector U where the i-th component U_i has been replaced by $x \in \mathcal{V}$. Given two vectors U and V, we say that U *matches* V when $U_i = \bot$ iff $V_i = \bot$. We say that U is a *face* of V, written $U \preceq V$, when for

all i, either $U_i = V_i$ or $U_i = \bot$. A vector U is *maximal* if none of its components is $\bot$. If X is a set of vectors, its *downward closure* is $\downarrow X = \{U \mid U \preceq V \text{ for some } V \in X\}$.

▶ **Definition 5.** *A* task *is a triple* $\Theta = (I, O, \Delta)$ *such that:*

- $I = \mathcal{V}^n$ *and* $O \subseteq \mathcal{V}^n$ *are sets of maximal vectors. The elements of* I *(resp. of* O*) and their faces are called* input *vectors (resp.* output *vectors).*
- $\Delta \subseteq \downarrow I \times \downarrow O$ *is a relation between input and output vectors, such that:*
  - Δ *only relates matching vectors,*
  - *for every input vector* $U \in \downarrow I$, $\Delta(U) \neq \varnothing$,
  - *for all input vectors* $U \preceq U'$, $\Delta(U) \subseteq \downarrow \Delta(U')$.

Above, we write $\Delta(U) = \{V \mid (U, V) \in \Delta\}$. Intuitively, the task specification says that if each process i starts with the input U_i, the set $\Delta(U)$ consists of all the output vectors that are considered acceptable outputs. A '$\bot$' component in a vector represents a process which is not participating in the computation, either because it crashed, or because it was so slow that all the other processes terminated before it could take any steps. With that in mind, the third condition on Δ asserts that, if such a "slow" process wakes up, there is at least one valid output which is compatible with what the other processes already decided. Note that, to match the receptivity property of concurrent specifications, we are requiring $I = \mathcal{V}^n$, i.e., a task must specify what the outputs should be given any input vector. This is not a restriction compared to the usual definition of a task: if we do not care about what happens on some of the input vectors, we can simply consider that every output is correct.

We now describe how we can turn a task into a one-shot object. Given a vector U, the set $\{i \in [n] \mid U_i \neq \bot\}$ is called the *participating set* of U. A pair $(U, V) \in \Delta$, where U and V have participating set $I = \{i_1, \ldots, i_k\}$, corresponds to a correct execution trace of the form $\mathsf{i}_{i_1}^{U_{i_1}} \cdots \mathsf{i}_{i_k}^{U_{i_k}} \cdot \mathsf{r}_{i_1}^{V_{i_1}} \cdots \mathsf{r}_{i_k}^{V_{i_k}}$, that is, a trace where all the participating processes invoke with the input values from U, and then they all respond with the output values from V. Remember that the order of consecutive invocations or consecutive responses does not matter. For convenience, we will write such a trace $\mathsf{i}^U \cdot \mathsf{r}^V$.

Thus, the task specification Δ gives us a set of execution traces $\{\mathsf{i}^U \cdot \mathsf{r}^V \mid (U, V) \in \Delta\} \subseteq \mathcal{T}$. But this set does not satisfy the conditions of Definition 1: we need to specify which traces are correct or not, among the traces of other "shapes" (i.e., traces where invocations and responses are interleaved). For example, consider a trace of the form $\mathsf{i}^U \cdot \mathsf{r}^V \cdot \mathsf{i}_j^x \cdot \mathsf{r}_j^y$, where $(U, V) \in \Delta$ have a participating set I, and $j \notin I$. Intuitively, this represents the situation where all the processes in I ran together without hearing from j, and after all of these processes terminated, the process j ran alone with input x and decided output y. What should be the condition on x and y for this trace to be considered correct? The most sensible answer is that we should require $(U[j \leftarrow x], V[j \leftarrow y]) \in \Delta$. The next definition generalizes this idea to traces of any shape.

▶ **Definition 6.** *Let* $\Theta = (I, O, \Delta)$ *be a task. We define the one-shot concurrent specification* $G(\Theta) \subseteq \mathcal{T}$ *as the set of execution traces* $T \in \mathcal{T}$ *such that:*

- T *is one-shot, i.e., every process has at most one invocation and one response in* T,
- *there exists a completion* T' *of* T *(obtained by appending responses to the pending invocations), such that for every non-empty prefix* S *of* T' *which is complete, we have* $(U_S, V_S) \in \Delta$, *where the* i*-th component of* U_S *(resp.* V_S*) is* x *if* i_i^x *(resp.,* r_i^x*) appears in* S, *and* $\bot$ *otherwise.*

▶ **Proposition 7.** $G(\Theta)$ *is a one-shot concurrent specification, i.e., it satisfies the properties of Definition 1, except that we only require receptivity among one-shot traces.*

Proof. Prefix-closure (1) and non-emptiness (2) are obvious.

- (3'): Let $T \in G(\Theta)$ and assume that no action from process i occurs in T. Let $x \in \mathcal{V}$, we want to show that $T \cdot \mathsf{i}_i^x \in G(\Theta)$. Let $T' = T \cdot \widehat{T}$ be the completion of T that appears in the definition of $G(\Theta)$ ($\widehat{T}$ consists of responses to the pending invocations of T). In particular, since T' is complete and a prefix of itself, we have $(U_{T'}, V_{T'}) \in \Delta$.
  What we have to do is find a response to the invocation i_i^x that obeys the specification of the task Θ. Since i does not participate in T, the vector $U_{T'}$ has a $\bot$ at position i, and so $U_{T'} \preceq U_{T'}[i \leftarrow x]$. Thus, we have $\Delta(U_{T'}) \subseteq {\downarrow}\Delta(U_{T'}[i \leftarrow x])$, since Θ is a task. So in particular $V_{T'} \in {\downarrow}\Delta(U_{T'}[i \leftarrow x])$, which means that there is some $W \in \Delta(U_{T'}[i \leftarrow x])$ which extends $V_{T'}$. Let $y \in \mathcal{V}$ be the i-th component of W. Pick the complete trace $T'' = T \cdot \mathsf{i}_i^x \cdot \widehat{T} \cdot \mathsf{r}_i^y$. Then we can check that $(U_{T''}, V_{T''}) = (U_{T'}[i \leftarrow x], W) \in \Delta$, and since all the other complete prefixes of T'' are also complete prefixes of T, we finally get $T \cdot \mathsf{i}_i^x \in G(\Theta)$.
- (4): Let $T \in G(\Theta)$ and assume that $\pi_i(T)$ has a pending invocation. Using the same notations as before, the suffix $\widehat{T}$ must contain a response r_i^y to this invocation. Then just by switching the order of the responses in $\widehat{T}$, we obtain $T \cdot \mathsf{r}_i^y \in G(\Theta)$.
- (5): Let $T = T_1 \cdot a_j \cdot \mathsf{i}_i^x \cdot T_2 \in G(\Theta)$, with $j \neq i$, and take the same notations as in the previous cases. We claim that $T_1 \cdot \mathsf{i}_i^x \cdot a_j \cdot T_2 \cdot \widehat{T}$ satisfies the conditions of Definition 6. The only way that this could fail is if the complete prefix S is $T_1 \cdot \mathsf{i}_i^x$: all other cases are covered by the fact that $T \in G(\Theta)$. But $T_1 \cdot \mathsf{i}_i^x$ cannot be complete since it has a pending invocation. So $T_1 \cdot \mathsf{i}_i^x \cdot a_j \cdot T_2 \in G(\Theta)$.
  The second half of condition (5) is proved similarly. ◀

There is also a map in the other direction: from a one-shot concurrent specification σ, we can produce a task $F(\sigma)$. This direction is much easier to define, all we have to do is keep all the traces of σ which consist of a sequence of invocations followed by a sequence of responses, and put in Δ the corresponding pair (U, V).

▶ **Definition 8.** *Given a one-shot concurrent specification $\sigma \subseteq \mathcal{T}$, the task $F(\sigma) = (I, O, \Delta)$ is defined as follows. For each trace $T \in \sigma$ of the form $T = \mathsf{i}_{i_1}^{x_1} \cdots \mathsf{i}_{i_k}^{x_k} \cdot \mathsf{r}_{i_1}^{y_1} \cdots \mathsf{r}_{i_k}^{y_k}$ (we say that T is* fully-concurrent*), we define the vectors U_T (resp., V_T) whose i_j-th component is x_j (resp., y_j) and all the other components are $\bot$. Then $\Delta = \{(U_T, V_T) \mid T \in \sigma$ is fully-concurrent$\}$, and I (resp., O) is the set of all the maximal input (resp., output) vectors that appear in Δ.*

▶ **Proposition 9.** *$F(\sigma)$ is a task.*

Proof. First, we justify that $\Delta \subseteq {\downarrow}I \times {\downarrow}O$. Let $(U_T, V_T) \in \Delta$ for some trace T. We want to show that U_T and V_T are faces of maximal vectors that appear in Δ. We will find a fully-concurrent trace $T' \in \sigma$ such that $U_T \preceq U_{T'}$ and $V_T \preceq V_{T'}$. For each process i that does not participate in T, by the receptivity property of σ, we can add an invocation i_i^x at the end of T. By totality, this invocation has an appropriate response r_i^y, that we add at the end of the trace. Then by applying the expansion property several times, we can push all the new invocations to the left to obtain the trace $T' \in \sigma$ which is fully-concurrent.

We then check that the three conditions on Δ are satisfied. Δ always relates matching vectors because the trace T has matching invocation and responses. For any input vector U, we can use totality of σ to find matching responses, which shows that $\Delta(U) \neq \varnothing$. Finally, given two input vectors $U \preceq U'$, and an element V of $\Delta(U)$, let T be the trace witnessing that $(U, V) \in \Delta$. Then, as in the first paragraph, we can add the missing invocations by receptivity, and matching responses by totality, and get a fully-concurrent trace by expansion. This yields a pair $(U', V') \in \Delta$, with $V \preceq V'$, which is what we want. ◀

The maps F and G are not inverse of each-other: as we stated in the introduction of the section, one-shot objects are more expressive than tasks (an example of a simple object which cannot be expressed as a task is exhibited in [2]). But the next Theorem shows that we still have a close correspondence between tasks and objects. Given two tasks $\Theta = (I, O, \Delta)$ and $\Theta' = (I', O', \Delta')$, we write $\Theta \subseteq \Theta'$ when $\Delta \subseteq \Delta'$.

▶ **Theorem 10.** *The maps F and G are monotonic, and they form a Galois connection between tasks and one-shot objects: for every task Θ and one-shot specification σ, we have*

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

Moreover, the following equality holds: $F \circ G(\Theta) = \Theta$.

Proof. The monotonicity of F and G follows directly from the definitions. In the rest of the proof, we write $F(\sigma) = (I', O', \Delta')$.

- ($\Rightarrow$): Assume $\sigma \subseteq G(\Theta)$. Let $(U_T, V_T) \in \Delta'$, for some fully-concurrent trace $T \in \sigma$. So, we also have $T \in G(\Theta)$. Since T is already complete (and a prefix of itself), we obtain $(U_T, V_T) \in \Delta$.
- ($\Leftarrow$): Conversely, assume $F(\sigma) \subseteq \Theta$, and let $T \in \sigma$. First, we complete T by adding response events to the pending invocations using totality. We get a trace $T' = T \cdot \widehat{T} \in \sigma$. Let S be a prefix of T' which is complete. By prefix-closure, we get $S \in \sigma$. Then, using the expansion property, we can push all the invocations to the left in order to get a trace $S' \in \sigma$ which is fully-concurrent. Moreover, since S' is obtained by reordering the actions of S, we have $U_S = U_{S'}$ and $V_S = V_{S'}$. By definition of $F(\sigma)$, we have $(U_{S'}, V_{S'}) \in F(\sigma)$, so by assumption $(U_S, V_S) = (U_{S'}, V_{S'}) \in G(\Theta)$.
- From the first implication, since $G(\Theta) \subseteq G(\Theta)$, we get $F \circ G(\Theta) \subseteq \Theta$. To prove the other inclusion, take $(U, V) \in \Delta$, and consider the associated trace $T = \mathsf{i}^U \cdot \mathsf{r}^V$. We have (by construction) $U = U_T$ and $V = V_T$, so we just need to check that $T \in G(\Theta)$. But since T is already complete, and the only complete non-empty prefix of T is T itself, with $(U_T, V_T) \in \Delta$, this is the case. ◀

Theorem 10 tells us a lot about the relationship between tasks and one-shot objects. The implication from left to right explains in what sense G is a canonical way to turn a task into a one-shot object: $G(\Theta)$ is the largest one-shot specification whose set of fully-concurrent traces obeys the task Θ. Moreover, the equality $F \circ G = \mathsf{id}$, tells us that G is an injection of tasks into one-shot objects. Thus, the objects that we are interested in are precisely the ones in the image of G.

▶ **Definition 11.** *A* task object *σ is a one-shot concurrent specification which can be written as $\sigma = G(\Theta)$ for some task Θ.*

The maps F and G form a bijection between tasks and task objects: indeed, given a task object $\sigma = G(\Theta)$, we have $G \circ F(\sigma) = G \circ F \circ G(\Theta) = G(\Theta) = \sigma$. Conversely, if a one-shot object σ satisfies $\sigma = G \circ F(\sigma)$, then it is a task object (corresponding to the task $F(\sigma)$). Thus, we have a characterization of task objects, which does not refer to the notion of task: σ is a task object *iff* $G \circ F(\sigma) = \sigma$. In fact, since the inclusion $\sigma \subseteq G \circ F(\sigma)$ holds for any one-shot object (as a consequence of Theorem 10), we even have the equivalence: σ is a task object *iff* $G \circ F(\sigma) \subseteq \sigma$. If we reformulate this inclusion by unfolding the definitions of F and G, we obtain a kind of closure property, in the style of Definition 1:

(6) *Task property:* for every complete one-shot trace $T \in \mathcal{T}$, if for every complete prefix S of T, expanding S gives a fully-concurrent trace $S' \in \sigma$, then $T \in \sigma$.

Then, a task object is a set of one-shot traces that satisfies the axioms (1) – (6).

4 An asynchronous computability theorem for arbitrary objects

The *asynchronous computability theorem* of Herlihy and Shavit [10] states that a task Θ has a wait-free protocol using read/write registers if and only if there exists a chromatic subdivision of the input complex, and a chromatic simplicial map from this subdivision to the output complex, that satisfies some conditions. That statement actually combines two claims: (a) a given protocol P using read/write registers solves the task Θ if and only if there exists a chromatic simplicial map from the protocol complex of P to the output complex, satisfying some conditions; and (b) to study task solvability using read/write registers, we can restrict to a particular class of protocols (iterated immediate snapshots) whose protocol complexes are subdivisions of the input complex.

In this section, we will prove a generalization of claim (a), which works not only for read/write registers, but for protocols which are allowed to use any combination of arbitrary objects, as defined in Section 2.2. We will not have a counterpart of claim (b), since this characterization is specific to the particular case of protocols using read/write registers.

4.1 Preliminaries

We first recall the definitions from combinatorial topology that we will be using. A more thorough account of these notions can be found in [6]. A *simplicial complex* $\mathcal{C} = (V, S)$ consists of a set V of *vertices* and a non-empty set S of finite subsets of V called *simplices*, such that:

- for each vertex $v \in V$, $\{v\} \in S$, and
- S is closed under containment, i.e., if $X \in S$ and $Y \subseteq X$, then $Y \in S$.

We sometimes abuse notations and write $X \in \mathcal{C}$ instead of $X \in S$ to mean that X is a simplex of $\mathcal{C}$. If $Y \subseteq X$, we say that Y is a *face* of X. The simplices which are maximal w.r.t. inclusion are called *facets*. The *dimension* of a simplex $X \in S$ is $\dim(X) = |X| - 1$. The dimension of the simplicial complex $\mathcal{C}$ is $\dim(\mathcal{C}) = \sup\{\dim(X) \mid X \in S\}$. A simplicial complex is *pure* if all its facets have the same dimension (which is also the dimension of the complex). We say that a simplicial complex $\mathcal{C} = (V, S)$ is a *subcomplex* of $\mathcal{C}' = (V', S')$ when $S \subseteq S'$, and we write it $\mathcal{C} \subseteq \mathcal{C}'$.

Fix a finite set A whose elements are called *colors*. A *chromatic simplicial complex* $\mathcal{C} = (V, S, \chi)$ consists of a simplicial complex (V, S) equipped with a *coloring map* $\chi : V \to A$ such that every simplex $X \in S$ has vertices of different colors. A *chromatic simplicial map* $f : \mathcal{C} \to \mathcal{C}'$ from $\mathcal{C} = (V, S, \chi)$ to $\mathcal{C}' = (V', S', \chi')$ is a mapping $f : V \to V'$ between the vertices of the two complexes, such that:

- the image of a simplex is a simplex, i.e., for every $X \in S$, $f(X) := \{f(v) \mid v \in X\} \in S'$,
- f is *color-preserving*, i.e., for every $v \in V$, $\chi'(f(v)) = \chi(v)$.

A *chromatic carrier map* Φ from $\mathcal{C}$ to $\mathcal{C}'$, written $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$, assigns to each simplex $X \in S$ a subcomplex $\Phi(X)$ of $\mathcal{C}'$, such that:

- Φ is monotonic, i.e., if $Y \subseteq X$ then $\Phi(Y) \subseteq \Phi(X)$,
- Φ is rigid, i.e., for every simplex $X \in S$ of dimension d, $\Phi(X)$ is pure of dimension d,
- Φ is chromatic, i.e., for every $X \in S$, $\chi(X) = \chi'(\Phi(X))$, where $\chi(X) = \{\chi(v) \mid v \in X\}$ and $\chi'(\Phi(X)) = \bigcup_{Z \in \Phi(X)} \chi'(Z)$.

Given a chromatic carrier map $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$ and a chromatic simplicial map $f : \mathcal{C}' \to \mathcal{C}''$, their composition $f \circ \Phi : \mathcal{C} \to 2^{\mathcal{C}''}$ is defined as $(f \circ \Phi)(X) = \bigcup_{Z \in \Phi(X)} f(Z)$. Finally, given two chromatic carrier maps Φ and Ψ from $\mathcal{C}$ to $\mathcal{C}'$, we say that Φ is *carried by* Ψ, written $\Phi \subseteq \Psi$, when for every simplex $X \in \mathcal{C}$, $\Phi(X) \subseteq \Psi(X)$.

4.2 Simplicial tasks

The notion of task that we had in Definition 5 was a direct reformulation of the usual definition that is used in the context of combinatorial topology. We now recall the usual definition, and explain briefly why it is the same as Definition 5. Recall that the set of values is $\mathcal{V}$, and n is the number of processes. From now on, when we talk about chromatic complexes, the underlying set of colors will be $[n]$.

▶ **Definition 12.** *A* simplicial task *is a triple* $(\mathcal{I}, \mathcal{O}, \Xi)$*, where:*

- $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I})$ *is the pure chromatic simplicial complex of dimension* $(n-1)$*, whose vertices are of the form* (i, v) *for all* $i \in [n]$ *and* $v \in \mathcal{V}$*, and which contains all the simplices that are well-colored.* $\mathcal{I}$ *is called the* input complex*.*
- $\mathcal{O} = (V_\mathcal{O}, S_\mathcal{O}, \chi_\mathcal{O}, \ell_\mathcal{O})$ *is a pure chromatic simplicial complex of dimension* $(n-1)$*, together with a labeling* $\ell_\mathcal{O} : V_\mathcal{O} \to \mathcal{V}$*, such that every vertex is uniquely identified by its color and its label.* $\mathcal{O}$ *is called the* output complex*.*
- $\Xi : \mathcal{I} \to 2^\mathcal{O}$ *is a chromatic carrier map from* $\mathcal{I}$ *to* $\mathcal{O}$*.*

As in Definition 5, we force the input complex to contain every possible combination of input values, which is unusual but can safely be assumed without loss of generality. There is a straightforward bijection between tasks and simplicial tasks. Consider a task $\Theta = (I, O, \Delta)$ in the sense of Definition 5. A vector $U \in (\mathcal{V} \cup \{\bot\})^n$ corresponds to the simplex $X = \{(i, U_i) \mid i \in [n] \text{ and } U_i \neq \bot\}$, where the vertex (i, U_i) is colored by i and labeled by U_i. Two matching vectors correspond to simplices with the same dimension and set of colors. A maximal vector (i.e., with no $\bot$ component) corresponds to a simplex of dimension $(n-1)$. Thus, the sets of maximal vectors I and O of a task correspond to the facets of the corresponding simplicial complexes $\mathcal{I}$ and $\mathcal{O}$; and their downward closures $\downarrow I$ and $\downarrow O$ correspond to $\mathcal{I}$ and $\mathcal{O}$. The fact that the relation $\Delta \subseteq \downarrow I \times \downarrow O$ relates only matching vectors, along with the non-emptiness of $\Delta(U)$, is equivalent to saying that the corresponding carrier map is rigid and chromatic. The last remaining condition is monotonicity of Δ, which must also hold for carrier maps. With that correspondence in mind, in the rest of the section, we will not distinguish tasks and simplicial tasks.

4.3 Simplicial protocols

The topological version of a protocol is defined similarly as for simplicial tasks:

▶ **Definition 13.** *A* simplicial protocol *is a triple* $(\mathcal{I}, \mathcal{P}, \Psi)$*, where:*

- $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I})$ *is the pure chromatic simplicial complex of dimension* $(n-1)$*, whose vertices are of the form* (i, v) *for all* $i \in [n]$ *and* $v \in \mathcal{V}$*, and which contains all the simplices that are well-colored.* $\mathcal{I}$ *is called the* input complex*.*
- $\mathcal{P} = (V_\mathcal{P}, S_\mathcal{P}, \chi_\mathcal{P}, \ell_\mathcal{P})$ *is a pure chromatic simplicial complex of dimension* $(n-1)$*, together with a labeling* $\ell_\mathcal{P} : V_\mathcal{P} \to \mathsf{Views}$*, where* Views *is an arbitrary set of* views*, such that every vertex is uniquely identified by its color and its label.* $\mathcal{P}$ *is called the* protocol complex*.*
- $\Psi : \mathcal{I} \to 2^\mathcal{P}$ *is a chromatic carrier map from* $\mathcal{I}$ *to* $\mathcal{P}$*.*

The intended meaning of Definitions 12 and 13 is the following: the simplicial protocol $(\mathcal{I}, \mathcal{P}, \Psi)$ *solves* the simplicial task $(\mathcal{I}, \mathcal{O}, \Xi)$ if there exists a chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Psi$ is carried by Ξ. In [6], this is taken as the definition of what it means for a protocol to solve a task. In Section 2, we have given more concrete definitions of protocols and solvability; our goal here is to properly define the protocol complex associated to a given protocol, so that these two definitions of solvability will agree.

Let Obj be the set of objects that our programs are allowed to use. Let $P = (P_i)_{i\in[n]}$ be a wait-free protocol in the sense of Section 2.2. Recall that each $P_i = (Q_i, \bot_i, \delta_i, \tau_i)$ is the program of process i. As before, we write $\mathcal{A} = \{\mathsf{i}_i^x, \mathsf{r}_i^x, \mathsf{i}(o)_i^x, \mathsf{r}(o)_i^x \mid o \in \mathsf{Obj}, i \in [n], x \in \mathcal{V}\}$ for the set of actions of the protocol. The effect of a trace $T \in \mathcal{A}^*$ on global states is the function denoted by $\Delta : \overline{Q} \times \mathcal{A}^* \to \overline{Q}$.

The states $q \in Q_i \setminus \{\bot_i\}$ such that $\delta_i(q) \in \mathcal{V}$ are called the *final states* of process i. Let $F_i \subseteq Q_i$ denote the set of final states of process i. A trace $T \in \mathcal{A}^*$ is *one-shot* if it contains at most one outer invocation i_i^x and one outer response r_i^x for each process i. In particular, there is no restriction on the number of inner actions, since the objects in Obj are not assumed to be one-shot. A one-shot trace $T \in \mathcal{A}^*$ is *terminating* if after executing it, each process which participates in T ends in a final state. More formally, if we write $q = \Delta(q_{\text{init}}, T)$, for each i such that i_i^x occurs in T, we must have $q_i \in F_i$. Note that in a valid terminating trace, no action r_i^x occurs: when a process is in a final state, the process is ready to return its output value, but it did not return it yet.

We can now define the protocol complex $\mathcal{P} = (V_\mathcal{P}, S_\mathcal{P}, \chi_\mathcal{P}, \ell_\mathcal{P})$ associated to P. The set of views is $\mathsf{Views} = \bigcup_i F_i$. The vertices are of the form (i, q_i) where $i \in [n]$ is a process number and $q_i \in F_i$ is a final state of process i. Such a vertex is colored by i and labeled by q_i. Finally, for each one-shot trace T which is valid and terminating, we get a simplex $Y_T = \{(i, q_i) \mid i \text{ participates in } T\} \in S_\mathcal{P}$, where q_i is the i-th component of $\Delta(q_{\text{init}}, T)$. The carrier map $\Psi : \mathcal{I} \to 2^\mathcal{P}$ is defined as follows. For $X \in \mathcal{I}$ an input simplex, we let $S = \chi_\mathcal{I}(X)$ be the set of participating processes, and $(v_i)_{i\in S}$ their input values. Then, $\Psi(X)$ consists of all simplexes Y_T where T is a valid terminating one-shot trace such that every invocation that occurs in T is of the form $\mathsf{i}_i^{v_i}$ for some $i \in S$.

It is straightforward to see that $\mathcal{P}$ is chromatic, and that Ψ is monotonic and chromatic. To check that $\mathcal{P}$ is pure of dimension $(n-1)$, let Y_T be a simplex of $\mathcal{P}$. We want to extend it to an $(n-1)$-dimensional simplex. To do so, first we append at the end of the trace T invocations i_i^x for each process i that does not participate in T. Since we assumed that the protocol P is wait-free, we can run each of these processes until it reaches a final state. Thus, we get a trace T' which is valid and terminating, and the corresponding simplex $Y_{T'}$ is of dimension $(n-1)$ and contains Y_T. Checking that Ψ is rigid is similar.

4.4 Asynchronous computability theorem

Let $\Theta = (\mathcal{I}, \mathcal{O}, \Xi)$ be a task, $P = (P_i)_{i\in[n]}$ a protocol, and $(\mathcal{I}, \mathcal{P}, \Psi)$ its associated simplicial protocol as described in the previous section. Notice that, since we defined the vertices of $\mathcal{P}$ to be pairs (i, q_i) where $q_i \in F_i$ is a final state of process i, there is a map $\delta : V_\mathcal{P} \to \mathcal{V}$ defined as $\delta(i, q_i) = \delta_i(q_i)$, where δ_i is the decision function of the program P_i.

The following Theorem gives a topological characterization of what it means for the protocol P to implement (in the sense of Definition 4) the task Θ:

▶ **Theorem 14.** *The protocol P implements the task-object $G(\Theta)$ if and only if the map $\delta : V_\mathcal{P} \to \mathcal{V}$ induces a chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Psi$ is carried by Ξ.*

Proof. In this proof, to distinguish between the execution traces of P (on the alphabet $\{\mathsf{i}_i^x, \mathsf{r}_i^x, \mathsf{i}(o)_i^x, \mathsf{r}(o)_i^x\}$) and the outer traces (on the alphabet $\{\mathsf{i}_i^x, \mathsf{r}_i^x\}$), we use lowercase letters t, t', s, s' for the former and capital letters T, T', S, S' for the latter.

($\Rightarrow$): Assume that P implements the object $G(\Theta)$, i.e., every trace $T \in [\![P]\!]$ satisfies the conditions of Definition 6. First, we want to define the map $\delta : V_\mathcal{P} \to V_\mathcal{O}$. Let $(i, q_i) \in V_\mathcal{P}$ be a vertex of $\mathcal{P}$; we would like to take $\delta(i, q_i) = (i, \delta_i(q_i))$, but we do not know yet that it is a vertex of $\mathcal{O}$. The vertex (i, q_i) belongs to a $(n-1)$-dimensional simplex Y_t for some

execution trace t which is valid, one-shot and terminating. Let $q = \Delta(q_{\text{init}}, t)$ be the global state after executing t. In particular, the i-th component of q is q_i. For each $j \in [n]$, write $d_j = \delta_j(q_j)$ the value that process j is about to decide in the trace t. We also write $v_j \in \mathcal{V}$ the input value of process j in t. Let t' denote the trace obtained by appending responses $\mathsf{r}_j^{d_j}$ at the end of t. Then t' is still a valid trace, so if we write $T = \pi(t)$ its projection, we get $T \in [\![P]\!] \subseteq G(\Theta)$. Since T is a complete trace, that means it respects the task specification, i.e., $\{(j, d_j) \mid j \in [n]\} \in \Xi(\{(j, v_j) \mid j \in [n]\})$. In particular, (i, d_i) is in the image of Ξ, so it is a vertex of $\mathcal{O}$.

The map $\delta : V_\mathcal{P} \to V_\mathcal{O}$ is obviously chromatic. Let us show that it is a simplicial map. Let $Y_t \in S_\mathcal{P}$ be a simplex of $\mathcal{P}$. Let $S \subseteq [n]$ be the set of processes participating in t, then Y_t is of the form $Y_t = \{(i, q_i) \mid i \in S\}$, and $\delta(Y_t) = \{(i, d_i) \mid i \in S\}$. As we did in the previous paragraph, we can add responses $\mathsf{r}_i^{d_i}$ for $i \in S$ at the end of the trace t, to obtain a complete valid trace whose projection is in $[\![P]\!]$, and thus also in $G(\Theta)$. This implies that $\delta(Y_t)$ is in the image of Ξ, so it is a simplex of $\mathcal{O}$.

Finally, we need to show that $\delta \circ \Psi$ is carried by Ξ. Let $X \in \mathcal{I}$ be an input simplex, and let $Z \in (\delta \circ \Psi)(X)$ be an output simplex. Then Z must be of the form $\delta(Y_t)$ for some $Y_t \in \Psi(X)$. We write $S \subseteq [n]$ the set of participating processes of X, and $(v_i)_{i \in S}$ their input values. So, t is a valid terminating one-shot trace such that every invocation in t is of the form $\mathsf{i}_i^{v_i}$ for some $i \in S$. Let $S' \subseteq S$ be the participating set of t. Let $q = \Delta(q_{\text{init}}, t)$ be the global state after executing t; we have $Y_t = \{(i, q_i) \mid i \in S'\}$, and $\delta(Y_t) = \{(i, d_i) \mid i \in S'\}$. Once again, we can append appropriate responses $\mathsf{r}_i^{d_i}$ to the trace t, and then we obtain $\pi(t') = T \in [\![P]\!] \subseteq G(\Theta)$. So, we obtain $\delta(Y_t) \in \Xi(\{(i, v_i) \mid i \in S'\})$. Since $\{(i, v_i) \mid i \in S'\} \subseteq X$, by monotonicity of Ξ, we finally get $\delta(Y_t) \in \Xi(X)$.

($\Leftarrow$): Assume that the induced map $\delta : V_\mathcal{P} \to V_\mathcal{O}$ defined as $\delta(i, q_i) = \delta_i(q_i)$ is a chromatic simplicial map from $\mathcal{P}$ to $\mathcal{O}$, and that $\delta \circ \Psi$ is carried by Ξ. We want to prove that P implements the object $G(\Theta)$, i.e., that $[\![P]\!] \subseteq G(\Theta)$. Let $T \in [\![P]\!]$ be a one-shot outer trace, and t such that $\pi(t) = T$ a valid execution trace for P. To show that $T \in G(\Theta)$, we must first complete it, that is, find valid responses to the pending invocations of T. Since the protocol P is wait-free, we can just run the pending processes one by one until they all reach a final state: formally, this amounts to appending inner actions to the trace t according to its program, until a final state q_i is reached. Then, we add the appropriate response $\mathsf{r}_i^{\delta_i(q_i)}$ to obtain a trace t', which extends t, is still valid, and which does not have pending invocations. The projection $T' = \pi(t') \in [\![P]\!]$ is an extension of T where we added responses to the pending invocations of T.

Now that we have T', we have to prove the following property on every non-empty prefix S of T' which is complete (including $S = T'$): the simplex Z_S must belong to $\Xi(X_S)$, where X_S is the input simplex $X_S = \{(i, v_i) \mid \mathsf{i}_i^{v_i} \text{ occurs in } S\}$ and Z_S is the output simplex $Z_S = \{(i, d_i) \mid \mathsf{r}_i^{d_i} \text{ occurs in } S\}$. Our goal is now to decompose Z_S as $Z_S = \delta(Y)$ for some $Y \in \Psi(X_S)$. Let s be the prefix of t' whose projection is $\pi(s) = S$. We write s' for the trace obtained by removing from s all the outer responses, i.e., actions of the form $\mathsf{r}_i^{d_i}$. We claim that s' is still a valid trace: indeed, no action from process i can occur after the outer response (otherwise s would not be one-shot), and the only effect of $\mathsf{r}_i^{d_i}$ is to change the local state of i, which does not affect the validity of the actions of other processes. Moreover, in the global states $q = \Delta(q_{\text{init}}, s')$, every process that participates in s' is in a final state, in other words, s' is terminating. Thus, we have a simplex $Y_{s'} \in S_\mathcal{P}$ in the protocol complex consisting of all the vertices (i, q_i) where i participates in s'. Since s', s and S all have the same participating set, $Y_{s'} \in \Psi(X_S)$. And since $d_i = \delta_i(q_i)$ (because in s', process i is ready to decide $\mathsf{r}_i^{d_i}$), $\delta(Y_{s'}) = Z_S$.

This decomposition of Z_S shows that $Z_S \in \delta \circ \Psi(X_S)$. Since we assumed that $\delta \circ \Psi$ is carried by Ξ, this implies $Z_S \in \Xi(X_S)$, which concludes the proof. ◀

Despite the verbosity of the proof, nothing complicated is going on: we are just putting together all the definitions of the paper. In particular, the crucial definitions that allow the proof to go through are the expansion property (5) in Definition 1; the map G (Definition 6) that characterizes the objects which correspond to tasks; and the definition of the protocol complex $\mathcal{P}$ associated to a protocol P in Section 4.3.

If we instantiate Theorem 14 with Obj containing only an iterated immediate snapshot object, combined with the fact that immediate snapshot protocol complexes are subdivisions of the input complex, we obtain Herlihy and Shavit's asynchronous computability theorem for read/write registers. In general, if we fix a particular set of objects Obj, to prove that a task cannot be solved using the objects of Obj, one needs to find a topological invariant which holds in *any* protocol complex $\mathcal{P}$ associated to any protocol P. This is usually where all the difficulty of the proof lies, and of course Theorem 14 does not help with that part. What the Theorem says is merely that solvability in the sense of protocol complexes agrees with the more basic notion of solvability defined in Section 4.

5 Conclusion

We have extended Herlihy and Shavit's asynchronous computability theorem, which gives a topological characterization of the solvability of tasks, not only in the context of read/write registers, but for a large class of arbitrary objects. This shows the soundness of the topological approach to fault-tolerant computability, as described for example in [6], where the existence of a simplicial map from the protocol complex to the output complex is taken as the *definition* of task solvability. A practical benefit of our proof is that we gave a general definition of what the protocol complex should be to model arbitrary objects. One can now instantiate this definition with a given protocol, without risk of making a mistake. This work paves the way towards a better understanding of the protocol complex: rather than seeing it as a huge, monolithic combinatorial object, we would like to construct it in a modular way, by decomposing it into more basic components.

To establish this theorem, we had to study the distinction between tasks and one-shot objects. We characterized the class of one-shot objects which correspond to tasks, and this result might be of independent interest. A natural continuation of this work would be to try to characterize the solvability of all one-shot objects, possibly using the notion of *refined task* introduced in [2]. Another generalization would be to extend it to t-resilient solvability, instead of focusing on wait-free protocols.

References

1. Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM. URL: `http://doi.acm.org/10.1145/167088.167119`.
2. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. URL: `http://doi.acm.org/10.1145/3266457`.
3. Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379 – 4398, 2010. European Symposium on Programming 2009.

4 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In Shlomi Dolev, editor, *Distributed Computing*, pages 329–338. Springer Berlin Heidelberg, 2006.
5 Éric Goubault, Jérémy Ledent, and Samuel Mimram. Concurrent specifications beyond linearizability. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018*, pages 28:1–28:16, 2018. URL: `https://doi.org/10.4230/LIPIcs.OPODIS.2018.28`.
6 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.
7 Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 324–333, New York, NY, USA, 1994. ACM. URL: `http://doi.acm.org/10.1145/197917.198119`.
8 Maurice Herlihy and Sergio Rajsbaum. *Algebraic topology and distributed computing: a primer*, pages 203–217. Springer Berlin Heidelberg, 1995. URL: `https://doi.org/10.1007/BFb0015245`.
9 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 133–142, New York, NY, USA, 1998. ACM. URL: `http://doi.acm.org/10.1145/277697.277722`.
10 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999. URL: `http://doi.acm.org/10.1145/331524.331529`.
11 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
12 Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7751`.
13 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 101–110, New York, NY, USA, 1993. ACM. URL: `http://doi.acm.org/10.1145/167088.167122`.