Concurrent specifications beyond linearizability

Éric Goubault
École Polytechnique, Palaiseau, France
eric.goubault@lix.polytechnique.fr

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

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

Abstract

With the advent of parallel architectures, distributed programs are used intensively and the question of how to formally specify the behaviors expected from such programs becomes crucial. A very general way to specify concurrent objects is to simply give the set of all the execution traces that we consider correct for the object. In many cases, one is only interested in studying a subclass of these concurrent specifications, and more convenient tools such as linearizability can be used to describe them.

In this paper, what we call a concurrent specification will be a set of execution traces that moreover satisfies a number of axioms. As we argue, these are actually the only concurrent specifications of interest: we prove that, in a reasonable computational model, every program satisfies all of our axioms. Restricting to this class of concurrent specifications allows us to formally relate our concurrent specifications with the ones obtained by linearizability, as well as its more recent variants (set- and interval-linearizability).

2012 ACM Subject Classification Concurrency

Keywords and phrases concurrent specification, concurrent object, linearizability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.81

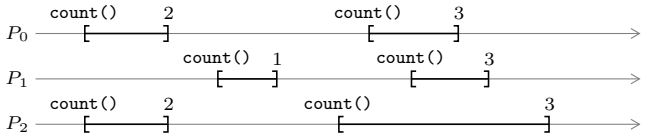
1 Introduction

A common setting to study distributed computing is the one of asynchronous processes communicating through shared objects. It is of particular interest in the area of fault-tolerant distributed computing [9], where we assume that some processes might crash during the computation. The goal is to determine which concurrent tasks are solvable by wait-free programs [8], depending on which shared objects the processes are allowed to use. In this context, the question of how to formally specify the behavior of the shared objects arises: what we want is an abstract, high-level specification, that does not refer to a particular implementation of the object. To illustrate our discussion, let us first introduce a toy (and running) example which we call the COUNT object.

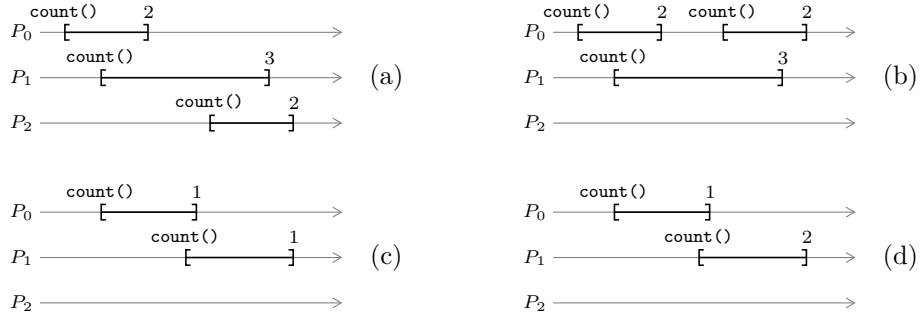
1.1 Counting processes

When a process calls the COUNT object, it should return the number of processes that are currently calling the object in parallel. This is similar, although not completely identical, to Java's `Thread.activeCount` method, which returns "an estimate of" the current number of running threads. A typical execution of this object is depicted below. Each of the three

processes P_0, P_1 and P_2 is calling the COUNT object twice. The horizontal axis represents the real-time scheduling of the operations and the intervals between square brackets depict the time span during which a process is executing the `count()` method: when two intervals vertically overlap, the processes are running the method concurrently. For instance, the last three calls are concurrent: the three processes should see each other and all return 3.

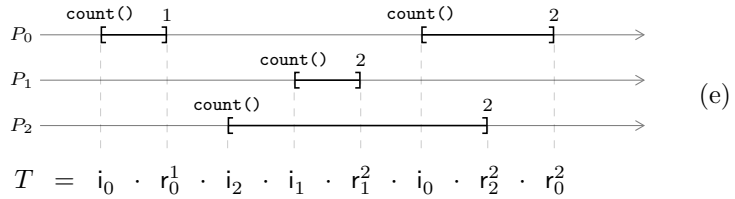


The specification of COUNT seems to be clear on this example. But what about the behaviors depicted below? In execution (a), process P_1 responds 3 since it has seen the two other processes; but there is no point in time when the three processes are running concurrently. Execution (b) represents the same situation, but this time P_1 has seen two different calls of P_0. In execution (c), the two calls are concurrent, but for a very short time, so they did not manage to see each other. In execution (d), process P_1 managed to see process P_0, but not reciprocally. All of these executions may or may not seem correct depending on what exact specification we have in mind. One could for example ask for a variant of the COUNT object that accepts executions (a) and (c), but rejects (b) and (d), and so on.



1.2 Specifying concurrent objects

A simple but very general way of specifying such objects was proposed by Lamport [13]. The specification of a concurrent object is simply the set of all the execution traces that we consider correct for this object. For the COUNT object example, if we draw every possible diagram as in the pictures above and decide which ones we want or not, we will have a well-specified object. Of course, we need a mathematical abstraction of these pictures, we represent them by *execution traces* (Lamport originally used the happens-before partial order; the trace formalism used here was introduced by Herlihy and Wing [10]).



An implicit assumption of this representation is that invocations and responses are totally ordered according to some global time clock. Another assumption is that only the relative

position of the events in time matters: the intervals can be moved around as long as the overlapping pattern is conserved. In particular, we cannot tell how long processes overlap, so execution (c) is indistinguishable from an execution where P_0 and P_1 are fully concurrent.

Although very powerful in terms of expressiveness, this idea of specifying a concurrent object by giving the set of all correct executions can be cumbersome to work with in practice.

Very often, one is only interested in studying concurrent objects that have a "sequential flavor", i.e., objects that are concurrent versions of sequential data structures, such as lists or queues. In this scenario, it is usually better to use one of the many correctness criteria found in the literature such as atomicity [16], sequential consistency [12], serializability [19], causal consistency [20], or linearizability [10]. In fact, those correctness criteria can be regarded as convenient ways of defining concurrent specifications in the sense of Lamport. For example, given the sequential specification σ of some object (say, a list), we can generate the set $\mathsf{Lin}(\sigma)$ of *linearizable* concurrent traces, that is, the set of correct executions of a concurrent list (according to linearizability). Thus, $\mathsf{Lin}(\sigma)$ is a concurrent specification (in the sense of Lamport) which is parameterized by a sequential specification σ. Note that σ is a much better-understood mathematical object than general concurrent specifications.

The drawback of these methods is that they cannot specify every concurrent object. The original paper on linearizability [10] already remarked that Lamport's approach is more general, since it can express non-linearizable behavior. The COUNT object described above is a typical example of an object that exhibits intrinsically concurrent behavior, which cannot be specified by sequential means. Many other examples are found in the area of distributed computability [9], such as immediate snapshot, consensus or set-agreement objects. Another notable example is Java's *Exchanger* object. In order to specify those objects, many variants of linearizability have been defined recently: local linearizability [6], set-linearizability [17] (a.k.a. concurrency-aware linearizability [7]) and interval-linearizability [2]. The latter notion is the most expressive one; in particular, they show that their framework allows to specify every concurrent *task*, in the sense of distributed computability [9].

1.3 Contributions

In this paper, we define a notion of *concurrent specification* which is based on Lamport's idea of a set of correct execution traces; but we moreover impose a number of axioms that should hold on this set. We advocate the relevance of this definition as follows.

Firstly, we define a reasonable computational model (thought of as the operational semantics of a concurrent programming language with shared objects), and prove that in this model, every program satisfies our axioms (Theorem 12). This means that our concurrent specifications are the only relevant ones: a specification that does not verify our axioms could never be implemented by a program. Thus, imposing those axioms is not very restrictive; rather, they are desirable properties that concurrent specifications should satisfy.

Secondly, this particular set of axioms allows us to relate formally our specifications and the linearizability-based ones, by constructing Galois connections between them (Theorem 16). In particular, as a corollary, we obtain that our concurrent specifications coincide with the ones that are definable using interval-linearizability. Beyond the applications that we show here, this theorem is also interesting in itself: it states formally in what sense linearizability is the canonical way to turn a sequential specification into a concurrent one. Indeed, given a sequential specification σ, linearizability gives the smallest possible concurrent specification, among the ones that both contain σ and satisfy our axioms.

1.4 Plan

We begin by introducing our definition of concurrent specification (Section 2). Then, we provide an operational model that validates our axioms (Section 3), and we establish Galois connections with various notions of linearizable specification (Section 4). Finally, we conclude (Section 5).

2 Concurrent specifications

In the rest of the paper, we suppose fixed a number $n \in \mathbb{N}$ of *processes* and write $[n] = \{0, 1, \ldots, n-1\}$ 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, and we write $\mathcal{T} = \mathcal{A}^*$ for the set of those; we also 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:

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

In the remaining of the paper, we will only consider alternating traces. 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) *alternating*: every $T \in \sigma$ is alternating,

(2) *prefix-closed*: if $T \cdot T' \in \sigma$ then $T \in \sigma$,

(3) *non-empty*: $\varepsilon \in \sigma$,

(4) *receptive*: if $T \in \sigma$ and $\pi_i(T)$ has no pending invocation, then $T \cdot \mathsf{i}_i^x \in \sigma$ for every $x \in \mathcal{V}$,

(5) *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$,

(6) has *commuting invocations*: if $T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$,

(7) has *commuting responses*: if $T \cdot \mathsf{r}_i^x \cdot \mathsf{r}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot T' \in \sigma$,

(8) is closed under *expansion*: if $T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$ with $i \neq j$ then $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T' \in \sigma$.

We write CSpec for the set of concurrent specifications.

A concurrent specification σ is the set of all executions that we consider acceptable: a program *implements* the specification σ if all the execution traces that it generates belong

to σ (this will be detailed in Section 3). The axioms (1-4) are quite natural and commonly considered in the literature. They can be read as follows: an object gives answers to invocations of processes and a process has to wait for the response before invoking again the object (1), an object can do one action at a time (2), an object can do nothing (3), invocations of objects are always possible (4). The axiom (5) 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 (4) 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 conditions (6), (7) and (8) might seem more surprising, and will be crucial to establish the Galois connection of Section 4. For example, one could expect to specify an object whose behavior depends on which process was invoked first; but that would break condition (6). As we show in Section 3, in an asynchronous model, no program can implement such a specification. Let us explain intuitively why the "closure under expansion" condition holds. Suppose $T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T'$ is an acceptable execution of the object we are specifying. Then, in the trace $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T'$, where process i invokes its operation a little bit earlier, i might be idle for a while and only start computing after j has returned, resulting in the same behavior. Thus, the execution $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T'$ should also be considered acceptable. Alternatively, we can also think of it as process j being idle for a while before it returns. Applied to the COUNT object example, the expansion property implies that execution (c) *must* be accepted, since it is obtained by expanding a correct sequential execution. This expansion condition reflects the idea that invocations and responses do not correspond to actual actions taken by the processes, but rather define an interval in which they are allowed to take steps.

The conditions (6) and (7) above ensure that two invocations (resp. two responses) "commute": we say that two alternating 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.

It will prove quite useful to consider operations in traces, which are pairs consisting of an invocation and its matching response. Formally,

▶ **Definition 2.** Consider an alternating trace $T = T_0 \cdots T_{k-1}$. An *operation* of process i in T is either

- a *complete operation*: a pair (p, q) such that $T_p = \mathsf{i}_i^x$ and $T_q = \mathsf{r}_i^y$ and T_q comes right after T_p in $\pi_i(T)$, or
- a *pending operation*: a pair $(p, +\infty)$ where T_p is a pending invocation,

where $p, q \in \mathbb{N}$, with $0 \leq p, q < k$, are indices of actions in the trace. We write $\mathrm{op}_i(T)$ for the set of operations of the i-th process and $\mathrm{op}(T)$ for the set of all operations.

The operations of a trace can be ordered by the smallest partial order $\preceq$ such that $(p, q) \prec (p', q')$ whenever $q < p'$. This partial order is called *precedence* and two incomparable operations are called *overlapping* or *concurrent*. Note that for every $i \in [n]$, $(\mathrm{op}_i(T), \preceq)$ is totally ordered.

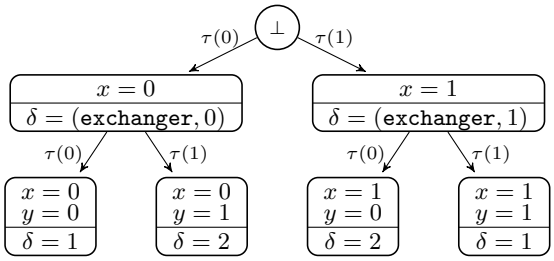
3 A computational model

In this section, we provide an operational model for concurrent programs communicating through shared objects. The model itself is similar to other trace-based operational semantics

that can be found in the literature [4]. We assume given a set $\mathcal{O}$ 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 $\mathcal{O}$, 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. 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 example below shows an implementation of the COUNT object for two processes communicating using a wait-free variant of Java's exchanger object: either two processes call the exchanger concurrently and swap their values, or the process fails after some timeout and gets back its own value.

```
count(x) {
  y = exchanger(x);
  if (y == x)
    return 1;
  else
    return 2;
}
```



We fix a set $\mathcal{O}$ of objects, along with their concurrent specification $\mathsf{spec}(o) \in \mathsf{CSpec}$ for each $o \in \mathcal{O}$. Here, a program (i.e., the implementation of an object) is basically a piece of code which takes a value as input, makes several calls to the objects in $\mathcal{O}$ (using algebraic operations to combine their results) and finally returns a value. Formally,

▶ **Definition 3.** A *program* is given by:

- a (possibly infinite) set Q of *local states* containing an *idle state* $\bot \in Q$,
- a *decision function* $\delta : Q \setminus \{\bot\} \to (\mathcal{O} \times \mathcal{V}) \sqcup \mathcal{V}$,
- 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 $q_{\text{init}} = (\bot_0, \dots, \bot_{n-1})$. 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 $\tau_i(\bot_i, x)$:

$$\Delta(q, \mathsf{i}_i^x) \quad = \quad q[i \leftarrow \tau_i(\bot_i, x)] \tag{9}$$

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 $o \in \mathcal{O}$ with input value $x \in \mathcal{V}$. This does not have any effect on the global state:

$$\Delta(q, \mathsf{i}(o)_i^x) \quad = \quad q \tag{10}$$

$\mathsf{r}(o)_i^x$: the object $o \in \mathcal{O}$ 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:

$$\Delta(q, \mathsf{r}(o)_i^x) \quad = \quad q[i \leftarrow \tau_i(q_i, x)] \tag{11}$$

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

$$\Delta(q, \mathsf{r}_i^x) \quad = \quad q[i \leftarrow \bot_i] \tag{12}$$

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 so on. A trace is valid if at each step in the execution, the next action is taken according to the decision function δ. Formally:

▶ **Definition 4.** 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:

- either $a = \mathsf{i}_i^x$ and $q'_i = \bot_i$;
- or $a = \mathsf{r}_i^y$ and $q'_i \neq \bot_i$ and $\delta_i(q'_i) = y$;
- or $a = \mathsf{i}(o)_i^x$ and $q'_i \neq \bot_i$ and $\delta_i(q'_i) = (o, x)$;
- or $a = \mathsf{r}(o)_i^y$ and $q'_i \neq \bot_i$.

Moreover, we require that for every object $o \in \mathcal{O}$, 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. Given a trace $T \in \mathcal{A}^*$, we write $\pi(T)$ for the trace obtained by keeping only outer actions.

▶ **Definition 5.** 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 its outer traces are valid with respect to σ.

We will now show that if P is wait-free, then $[\![P]\!]$ itself is a concurrent specification (Theorem 12). This means that in our model, the set of traces produced by a program necessarily satisfies all the axioms of concurrent specifications. The wait-free assumption is only used to prove totality; all the other axioms are true for any program. In the following, we use uppercase letters T, T' to denote traces containing only outer actions, and lowercase letters w, w', r, s, t to denote traces that might contain both inner and outer actions.

▶ **Lemma 6.** *The following commutativity properties hold:*

– commutativity of invocations*: for all* $T, T' \in \mathcal{A}^*$*,* $i, j \in [n]$ *and* $x, y \in \mathcal{V}$*,*

$$T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in [\![P]\!] \qquad \textit{implies} \qquad T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in [\![P]\!],$$

– commutativity of responses*: for all* $T, T' \in \mathcal{A}^*$*,* $i, j \in [n]$ *and* $x, y \in \mathcal{V}$*,*

$$T \cdot \mathsf{r}_i^x \cdot \mathsf{r}_j^y \cdot T' \in [\![P]\!] \qquad \textit{implies} \qquad T \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot T' \in [\![P]\!].$$

Proof. Assume there is a word $w \in \mathcal{T}_P$ such that $\pi(w) = T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T'$. So, w is of the form $r \cdot \mathsf{i}_i^x \cdot s \cdot \mathsf{i}_j^y \cdot t$ where s does not contain outer actions. We will show that $w' = r \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s \cdot t \in \mathcal{T}_P$, and the result follows. Since the inner projections are the same, the specifications of the objects are satisfied. We have to show that the conditions regarding the decision functions are respected. Write $q = \Delta(q_{\text{init}}, r)$ and $q' = \Delta(q_{\text{init}}, r \cdot \mathsf{i}_i^x \cdot s)$. Then since w is valid, we have $q_i = \bot_i$ and $q'_j = \bot_j$.

Claim 1: s does not contain any action of process j. Indeed, only an outer action r_j^y can set j's local state to $\bot_j$, and s has no outer action: so j's local state is $\bot_j$ during all of s. But inner actions can only be valid when the local state is not $\bot_j$.

Claim 2: Let u be a prefix of s and $q'' = \Delta(q_{\text{init}}, r \cdot \mathsf{i}_i^x \cdot u) = \Delta(q, \mathsf{i}_i^x \cdot u)$. Then $\Delta(q, \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot u) = q''[j \leftarrow \tau_j(\bot_j, y)]$. This is proved by induction on the length of the prefix u and using Claim 1.

Claim 3: $\Delta(q, \mathsf{i}_i^x \cdot s \cdot \mathsf{i}_j^y) = \Delta(q, \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s)$. Take $u = s$ in Claim 2.

Finally, let u is a prefix of $w' = r \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s \cdot t$. If u ends in r or (by Claim 3) in t, the global state after executing it is the same as for w, so the validity condition is verified. If u ends in s, then by Claim 2 the global state only differs by its j component, but since by Claim 1 the next action is not from j, the validity condition is also verified.

For commutativity of responses, the situation is very similar: suppose that $w = r \cdot \mathsf{r}_i^x \cdot s \cdot \mathsf{r}_j^y \cdot t \in \mathcal{T}_P$, and show that $w' = r \cdot s \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot t \in \mathcal{T}_P$. The analogue of Claim 1 says that there is no action of process i in s. ◀

▶ **Lemma 7.** $[\![P]\!]$ *has the expansion property.*

Proof. The proof is again very similar to that of Lemma 6. Assume that there is a word $w \in \mathcal{T}_P$ such that $\pi(w) = T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T'$. Thus, w is of the form $r \cdot \mathsf{r}_j^y \cdot s \cdot \mathsf{i}_i^x \cdot t$, where s does not contain outer actions.

Then we have two ways of making the invocation and response commute:

– either show that $w' = r \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot s \cdot t \in \mathcal{T}_P$, as in the proof of the commutativity of invocations,
– or show that $w'' = r \cdot s \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot t \in \mathcal{T}_P$, as in the proof of the commutativity of responses.

These two possible proofs correspond to the two ways that we can view the expansion property that were explained in Section 2: either process i is invoked sooner and idles for a while (first proof), or process j idles before returning (second proof). ◀

The key reason that makes Lemmas 6 and 7 go through is the fact that the actions i_i^x and r_i^y do not have any effect besides starting a process or terminating it. Taking these steps does not communicate any information to the other processes. For example, a process might start running and then wait for a while before doing any "real" computation. Such a process cannot be seen by the others. This is an arbitrary choice in how we designed our

computational model, but it reflects what really happens in practice: calling a function, or returning a value, both consume some amount of clock cycles from the processor, and they do not usually have any effect on the shared memory. Thus, one could possibly have a process which has started running, but was immediately preempted by the scheduler before it could perform any meaningful operation.

▶ **Lemma 8.** *$[\![P]\!]$ is closed under prefix.*

Proof. $\mathcal{T}_P$ is closed under prefix since for all $o \in \mathcal{O}$, $\mathsf{spec}(o)$ is closed under prefix by definition, and the conditions on the decision functions are also preserved. Then, $[\![P]\!]$ is also closed under prefix. ◀

▶ **Lemma 9.** *Every trace $T \in [\![P]\!]$ is alternating.*

Proof. T must begin by an invocation because $q_{\mathrm{init}\,i} = \bot_i$ and the only i-action that can occur with local state $\bot_i$ is i_i^x. We then prove by induction on the length of $w \in \mathcal{A}^*$ that $\Delta(q_{\mathrm{init}}, w)_i \neq \bot_i$ if the last outer i-action in w was an invocation, and $\Delta(q_{\mathrm{init}}, w)_i = \bot_i$ if it was a response. This implies that no two invocations nor two responses by the same process can occur consecutively in a valid trace. ◀

▶ **Lemma 10.** *$[\![P]\!]$ is receptive.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ does not have a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is a response, and thus we have $\tau(q_{\mathrm{init}}, w)_i = \bot_i$ (cf. proof of Lemma 9). So $w \cdot \mathsf{i}_i^x$ is valid for all x, and $T \cdot \mathsf{i}_i^x \in [\![P]\!]$. ◀

▶ **Lemma 11.** *$[\![P]\!]$ is total.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ has a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is an invocation, so the local state of i after executing w is $q_i \neq \bot_i$. If $\delta_i(q_i) = y \in \mathcal{V}$, then $w \cdot \mathsf{r}_i^y$ is a valid execution, which concludes the proof. Otherwise, $\delta_i(q_i) = (o, x)$. Then $w \cdot \mathsf{i}(o)_i^x$ is valid because the object o is receptive. And since o is total, there exists y such that $w \cdot \mathsf{i}(o)_i^x \cdot \mathsf{r}(o)_i^y$ is valid. The new local state of i after executing this trace is $q_i' \neq \bot_i$, so we can iterate the previous reasoning. Eventually, we will reach some local state q_i'' with $\delta_i(q_i'') = y'' \in \mathcal{V}$, because P is wait-free (i.e., there is no infinite execution ending with only inner i-actions). ◀

Putting all the previous lemmas together, we obtain Theorem 12:

▶ **Theorem 12.** *The semantics $[\![P]\!]$ of a wait-free protocol is a concurrent specification.*

This theorem ensures that the axioms of concurrent specifications are reasonable, in the sense that they are validated in our model. Thus, our concurrent specifications are the only ones of interest: specifications that do not satisfy these axioms cannot be implemented. For instance, this theorem implies that any protocol implementing a COUNT-like object *must* accept execution (c), since it is obtained by expanding a valid sequential execution, and the protocol's behavior is closed under expansion.

4 Linearizability

The notion of *linearizability* has been introduced by Herlihy and Wing [10] as a correctness criterion for concurrent implementations of sequential data structures. Linearizability is very popular thanks to its *locality* property, which allows programmers to reason modularly about

each object. Here, we adopt a slightly unusual point of view on linearizability: instead of a correctness criterion, it is a map that turns a sequential specification into a concurrent specification, in our sense.

For ease of presentation, we begin by introducing a general notion called $\mathcal{L}$-linearizability which subsumes several variants found in the literature. It is actually quite straightforward to check that the usual notions of linearizability, set-linearizability and interval-linearizability are recovered by instantiating $\mathcal{L}$ with the right set of traces. Thus, the definition of $\mathcal{L}$-linearizability should not be regarded as a contribution of this paper; it is merely a presentation trick to avoid dealing with three variants of the same definition.

The main result of this section is stated in Theorem 16. We then explore its consequences in three particular cases: standard, set- and interval-linearizability.

4.1 $\mathcal{L}$-specifications

We first introduce a notion of specification, akin to the one of Definition 1, which is parameterized by a set $\mathcal{L}$ of traces, which we abstractly consider as the "linear" ones. To recover the standard notion of linearizability, we let $\mathcal{L}$ be the set of sequential traces, in which case $\mathcal{L}$-specifications correspond to sequential specifications.

We always require $\mathcal{L}$ to be alternating, prefix-closed, non-empty and

- *receptive* for complete traces: if $T \in \mathcal{L}$ is complete then $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ for all $i \in [n]$ and $x \in \mathcal{V}$,
- *fully total*: if $T \in \mathcal{L}$ and $\pi_i(T)$ has a pending invocation then $T \cdot \mathsf{r}_i^x \in \mathcal{L}$ for every $x \in \mathcal{V}$.

Intuitively, these conditions mean that $\mathcal{L}$ is a set of traces which have some specific "shape" (e.g., being sequential), but it does not say anything about the values. The set of sequential traces is the smallest set satisfying those properties. Now, given such a set $\mathcal{L}$, an $\mathcal{L}$-specification says which execution traces are correct or not, but only among those of $\mathcal{L}$.

▶ **Definition 13** ($\mathcal{L}$-specification)**.** An *$\mathcal{L}$-specification* σ is a set of traces in $\mathcal{L}$ which is prefix-closed, non-empty and

- *receptive* within $\mathcal{L}$: for all $T \in \sigma$, $i \in [n]$ and $x \in \mathcal{V}$, if $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ then $T \cdot \mathsf{i}_i^x \in \sigma$,
- *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$.

We write $\mathsf{Spec}_{\mathcal{L}}$ for the set of $\mathcal{L}$-specifications.

In particular, a concurrent specification is an $\mathcal{L}$-specification for $\mathcal{L}$ the set of all alternating traces, such that conditions (6), (7) and (8) are moreover satisfied.

4.2 Comparison with concurrent specifications

In order to compare $\mathcal{L}$-specifications and concurrent ones, we define two functions as below:

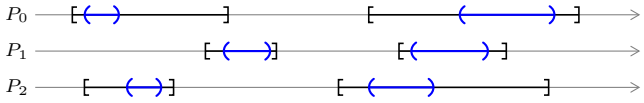
$$\mathsf{Spec}_{\mathcal{L}} \underset{\mathsf{U}_{\mathcal{L}}}{\overset{\mathsf{Lin}_{\mathcal{L}}}{\rightleftarrows}} \mathsf{CSpec} \qquad \mathsf{Spec}_{\mathcal{L}} \perp \mathsf{CSpec}$$

The function $\mathsf{U}_{\mathcal{L}}$ is defined by keeping only linear traces in a specification: $\mathsf{U}_{\mathcal{L}}(\tau) = \tau \cap \mathcal{L}$. The converse will require us to consider traces which are linearizable with respect to $\mathcal{L}$.

We write $T \rightsquigarrow T'$ when the trace T' can be obtained from the trace T by applying the following series of local transformations (forming a string rewriting system):

$$\mathsf{i}_i^x \cdot \mathsf{i}_j^y \rightsquigarrow \mathsf{i}_j^y \cdot \mathsf{i}_i^x \qquad \mathsf{r}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{r}_i^x \qquad \mathsf{i}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{i}_i^x \qquad \text{for } i \neq j.$$

This amounts to contracting the intervals, the opposite of expansion. In the picture below, T is represented in black with square brackets, T' in blue with round brackets:



▶ **Definition 14.** A trace $T \in \mathcal{T}$ is *$\mathcal{L}$-linearizable* w.r.t. an $\mathcal{L}$-specification σ if there exists a completion T' of T, obtained by appending responses to the pending invocations of T, and a trace $S \in \sigma$, such that $T' \rightsquigarrow S$.

It is not difficult to check that the above definition is equivalent to the usual one of [10]; this local presentation as a rewriting system is sometimes used in linearizability proofs using Lipton's left/right movers [14]. A minor difference is that the completion T' of T is usually allowed to remove some of the pending invocations of T. This is not useful here since all our specifications are total: a response to the pending invocations can always be found. Finally, given an $\mathcal{L}$-specification σ, we define $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ as the set of traces which are $\mathcal{L}$-linearizable with respect to σ. We now show that it is a concurrent specification.

▶ **Proposition 15.** *Let $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ be an $\mathcal{L}$-specification. Then*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \quad = \quad \{T \in \mathcal{T} \mid T \text{ is linearizable with respect to } \sigma\}$$

is a concurrent specification.

Proof (Sketch). We prove the conditions (6), (7) and (8) of Definition 1, which are the ones of interest in this paper. The closure under prefix is a bit technical.

- Commutativity of invocations. Assume $T = U \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. So there is a sequence of responses $\hat{T}$ and $S \in \sigma$ such that $T \cdot \hat{T} \rightsquigarrow S$. By rewriting one step ($i \neq j$ because of alternation), we get $U \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot V \cdot \hat{T} \rightsquigarrow T \cdot \hat{T} \rightsquigarrow S$. Thus, $U \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$.
- Commutativity of responses is similar.
- For closure under expansion, assume $T = U \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$ with $i \neq j$, and let $\hat{T}$ and $S \in \sigma$ be such that $T \cdot \hat{T} \rightsquigarrow S$. We want to show that $T' = U \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. But since $T' \cdot \hat{T} \rightsquigarrow T \cdot \hat{T} \rightsquigarrow S$, we are done. ◀

Proposition 15 shows that all the axioms that we impose on our concurrent specifications are reasonable in the sense that they are naturally enforced by all the specification techniques based on linearizability. Thus, linearizability is a canonical way of turning a weaker specification σ (e.g., one specifying only sequential behaviors) into a concurrent specification:

▶ **Theorem 16.** *The functions $\mathsf{Lin}_{\mathcal{L}}$ and $\mathsf{U}_{\mathcal{L}}$ are monotonous w.r.t. inclusions, and form a Galois connection: for every $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ and $\tau \in \mathsf{CSpec}$,*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau \qquad \Longleftrightarrow \qquad \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau).$$

Proof. The monotonicity of $\mathsf{U}_{\mathcal{L}}$ is trivial. For $\mathsf{Lin}_{\mathcal{L}}$, assume $\sigma \subseteq \sigma'$ and $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. Let $S \in \sigma$ be a linearization of T. Since $S \in \sigma'$, we also have $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma')$.

Now let σ and τ as in the theorem and assume $\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau$. By monotonicity of $\mathsf{U}_{\mathcal{L}}$, $\mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma)) \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$. But $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma))$ since every $T \in \sigma$ is in $\mathcal{L}$ and is its own linearization (if T has pending invocations, we can add any valid response using totality).

Conversely, assume $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$, and let T be a linearizable trace w.r.t. σ. Let T' be an extension of T and $S \in \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau) \subseteq \tau$ such that $T' \rightsquigarrow S$. So we can go from S to T' through a sequence of expansions and commutations, which gives us $T' \in \tau$ by applying axioms (6-8) of concurrent specifications, and by prefix closure, $T \in \tau$. ◀

In general, the posets $\mathsf{Spec}_{\mathcal{L}}$ and CSpec ordered by inclusion are not isomorphic, but the above theorem shows that the next best thing one could expect happens: every $\mathcal{L}$-specification has a canonical approximation as a concurrent specification and conversely. Note that Galois connections are widely used for comparing semantics of programs and deriving program analysis methods [3], and are a particular case of adjunctions, which have been promoted as the canonical way of comparing models for concurrency [18].

The right-to-left implication of Theorem 16 can be understood as follows: $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ is the smallest concurrent specification which contains σ. Notice that the axioms (6-8) are crucial here: if one of them were missing, we could produce specifications smaller than or incomparable to $\mathsf{Lin}_{\mathcal{L}}(\sigma)$. In fact, the theorem states that $\mathsf{Lin}_{\mathcal{L}}$ is a kind of free construction: starting with the traces in σ, we add all the traces that are required to be in the specification by our axioms, and no other trace than the required ones.

4.3 Sequential linearizability

A trace T is *sequential* when the poset $(\mathrm{op}(T), \preceq)$ is totally ordered, we write seq for the set of sequential traces. A *sequential specification* is a seq-specification (i.e., Definition 13 with $\mathcal{L} = \mathsf{seq}$). Note that a sequential specification is not a particular case of concurrent specification: it only satisfies the receptivity condition of Definition 13, not the stronger one of Definition 1. Intuitively, this means a sequential specification does not specify which behaviors are allowed when some of the processes run in parallel. Sequential linearizability is thus a canonical way to extend a sequential specification to a concurrent one.

The notion of seq-linearizability coincides with the usual notion of linearizability, that we call here *sequential linearizability* to avoid confusion. The Galois connection of Theorem 16 says that $\mathsf{Lin}_{\mathsf{seq}}(\sigma)$ is the smallest concurrent specification whose set of sequential traces contains σ. Moreover, it is a Galois insertion:

▶ **Proposition 17.** *For every $\sigma \in \mathsf{Spec}_{\mathsf{seq}}$, we have $\mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma)) = \sigma$.*

Proof. The inequality $\sigma \subseteq \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$ is implied by the Galois connection. Let $T \in \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$, i.e., T is both sequential and linearizable w.r.t. σ. Let T' be a completion of T and $S \in \sigma$ such that $T' \rightsquigarrow S$. Since T' is sequential, it is a normal form of the rewriting system (i.e., no rule can be applied): so we must have $T' = S$. Moreover S is required to be in σ, so $T' \in \sigma$ and by prefix-closure, $T \in \sigma$. ◀

This implies that $\mathsf{Spec}_{\mathsf{seq}}$ is a subposet of CSpec, which justifies calling *linearizable* a concurrent specification in the image of $\mathsf{Lin}_{\mathsf{seq}}$. This is however a strict subposet, as an application of Theorem 16:

▶ **Proposition 18.** *There are non-linearizable concurrent specifications.*

Proof. From Proposition 17, any linearizable concurrent specification $\tau = \mathsf{Lin}_{\mathsf{seq}}(\sigma)$ satisfies $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\tau)) = \tau$. Now, consider the concurrent specification $\textsc{set-count} \subseteq \mathcal{T}$ which is the set of traces whose set of operations has a partition $(E_i)_{i \in I}$ such that every $e, e' \in E_i$ are concurrent and their response value is the cardinal of E_i.

Informally, E_i is a set of pairwise-concurrent processes that "saw" each other. Note that, because of expansion, we cannot require that all processes running in parallel should see each

other: the execution (b) is accepted. All other executions (a), (c), (d) and (e) are rejected. In a sequential trace $T \in$ SET-COUNT, every response returns 1. Thus, $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\text{SET-COUNT}))$ only contains traces whose response is 1. But SET-COUNT also has traces with different responses, e.g. $\mathsf{i}_i \cdot \mathsf{i}_j \cdot \mathsf{r}_j^2 \cdot \mathsf{r}_i^2$. Therefore, SET-COUNT is not linearizable. ◀

4.4 Set-linearizability

The idea behind set-linearizability [17] is to specify what happens when a set of processes call an object at the same time. In this setting, an execution trace will be a sequence of sets of processes. In each of these sets, all the processes start executing, then all of them must terminate before we proceed with the next set of processes. Set-linearizability was also recently re-discovered by Hemed et al. [7], who call it *concurrency-aware linearizability*. A typical example of a set-linearizable (but not linearizable) object is the *immediate-snapshot* protocol [1] widely used in distributed computability [9]. Another example is Java's *exchanger* that allows two concurrent threads to atomically swap values.

A trace T is *set-sequential* if it is of the form $T = I_1 \cdot R_1 \cdots I_k \cdot R_k$, where each of the I_i is a non-empty sequence of invocations, R_i for $i < k$ is a sequence of responses with the same set of process numbers as I_i, and R_k is a (possibly empty) sequence of responses whose process numbers are included in those of I_k. We write set for the set of such traces. If we consider $\mathcal{L}$-linearizability with $\mathcal{L} = \mathsf{set}$, we recover the previously defined notion of set-linearizability [17]. Of course, Theorem 16 still holds, but we do not have an analogue of Proposition 17 here: given $\sigma \in \mathsf{Spec}_{\mathsf{set}}$, $\mathsf{U}_{\mathsf{set}}(\mathsf{Lin}_{\mathsf{set}}(\sigma))$ might actually contain more set-sequential traces than σ. This is because set-sequential specifications are not required to satisfy the expansion property nor the commutativity of invocations and of responses. The linearizability map adds just enough set-sequential traces to make these properties verified. A concurrent specification is *set-linearizable* if it is of the form $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ for some $\sigma \in \mathsf{Spec}_{\mathsf{set}}$.

▶ **Example 19.** The SET-COUNT specification defined in the proof of Proposition 18 is set-linearizable. It is obtained as $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ where σ is the set of set-sequential traces of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where every response in R_i is returning the value $|I_i|$.

▶ **Proposition 20.** *There are non-set-linearizable concurrent specifications.*

Proof. This is again an application of Theorem 16: from the properties of Galois connections, a set-linearizable concurrent specification τ must be such that $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau)) = \tau$. Define the concurrent specification INTERVAL-COUNT as the set of alternating traces such that every operation e has a response of the form r_i^k, where k is smaller or equal to the number of operations that are overlapping with e. This is a very permissive version of counting: all executions (a)–(e) are allowed. In a set-sequential trace, all return values are at most n, the total number of processes. Therefore, $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau))$ only contains traces whose response is smaller than n. But τ also contains traces with greater responses, as in execution (b) where process P_1 responds 3 even though only two processes are running. We deduce that INTERVAL-COUNT is not set-linearizable. ◀

4.5 Interval-linearizability

Interval-linearizability was introduced in [2] with the aim of going beyond linearizability and set-linearizability, in order to be able to specify every distributed *task*. They prove that every task can be obtained as the restriction to one-shot executions of an interval-linearizable object. We will show that this result actually extends beyond one-shot tasks:

every concurrent specification is interval-linearizable. An example of an interval-linearizable (but not set-linearizable) object is the (non-immediate) write-snapshot object.

We write int for the set of all alternating traces. The notions of *interval-sequential specification* and *interval-linearizability* defined in [2] coincide with $\mathcal{L}$-specifications and $\mathcal{L}$-linearizability where $\mathcal{L} = \mathsf{int}$. Interval-sequential specifications are almost the same as concurrent specifications, except that they do not require the commutativity of invocations, commutativity of responses and expansion property to be satisfied. In fact, it is mentioned in [2] that one can without loss of generality restrict to interval-sequential executions of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where the I_i (resp., R_i) are non-empty sets of invocations (resp., responses). This amounts to enforcing the commutativity of invocations and of responses. We do not include it here since it will be enforced anyway after we apply linearizability.

$\mathsf{U}_{\mathsf{int}}$ is by definition the "identity" function. As in the case of set-linearizability, an analogue of Proposition 17 does not hold, but we have the converse:

▶ **Proposition 21.** *For every* $\tau \in \mathsf{CSpec}$, $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) = \tau$.

Proof. The inclusion $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) \subseteq \tau$ follows from the Galois connection. For the other inclusion, recall that $\mathsf{U}_{\mathsf{int}}$ is the identity, so we want to prove $\tau \subseteq \mathsf{Lin}_{\mathsf{int}}(\tau)$. But this is trivial since every trace is its own linearization. ◀

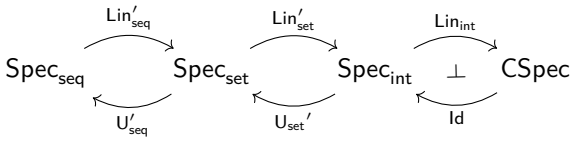
A concurrent specification is *interval-linearizable* if it is in the image of $\mathsf{Lin}_{\mathsf{int}}$.

▶ **Proposition 22.** *Every concurrent specification is interval-linearizable.*

Proof. This is an immediate corollary of Proposition 21: $\tau = \mathsf{Lin}_{\mathsf{int}}(\tau)$, and τ is (in particular) an interval-sequential specification. ◀

Thus, interval-linearizable objects and concurrent specifications are one and the same. One can see interval-linearizability as a convenient way of defining concurrent specifications in practice: we do not have to worry about the expansion and commutativity conditions (6-8), linearizability does that for us.

The results of the last three sections can be summed up in the following diagram, by factoring $\mathsf{Lin}_{\mathsf{seq}}$ and $\mathsf{Lin}_{\mathsf{set}}$ as follows:



where the $\dashv$ symbol between $\mathsf{Lin}_{\mathsf{int}}$ and Id indicates that they are related by a Galois connection, and the primed functions are the expected ones. Moreover, the two other Galois connections are obtained by composition:

$$\mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} = \mathsf{Lin}_{\mathsf{set}} \dashv \mathsf{U}_{\mathsf{set}} \qquad \text{and} \qquad \mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} \circ \mathsf{Lin}'_{\mathsf{seq}} = \mathsf{Lin}_{\mathsf{seq}} \dashv \mathsf{U}_{\mathsf{seq}}.$$

4.6 Other variants

Using the results of Section 4.2, it is easy to come up with new notions of linearizability. For instance, say a trace T is *k-concurrent* if every prefix of T has at most k pending invocations. Intuitively, a trace is k-concurrent if at any time, no more than k processes are running in parallel. Let $\mathcal{L}$ be the set of k-concurrent traces. Then, given a specification that says how

an object behaves when it is accessed concurrently by at most k processes, $\mathcal{L}$-linearizability is a canonical way of extending this specification to any number of processes. In their paper about concurrency-aware linearizability [7], Hemed et al. specify Java's exchanger object on traces that are both 2-concurrent and set-sequential, then they apply linearizability to obtain the full concurrent specification.

5 Conclusion and perspectives

We have studied a class of concurrent specifications satisfying a number of desirable properties, and argued that those properties make sense computationally. This has allowed us to formally compare different notions of linearizability in Theorem 16. Finally, we have explored some consequences and applications of this theorem. We believe that this should be the starting point of a series of future works.

While we have illustrated the robustness of this notion, there are many possible small interesting variants, such as removing the totality condition to model programs which are not wait-free. In particular, it is often desirable to impose specifications to be deterministic: the most simple definition (if the specification contains two traces $T \cdot \mathsf{r}_i^x$ and $T \cdot \mathsf{r}_i^y$ then $x = y$ should hold) does not play well with asynchrony, we think that a definition in the spirit of [15] is however possible. Another direction worth considering is to get rid of the notion of global time, inherent to the trace formalism. Very recent work generalize linerizability in this direction, with applications to weak memory models [21] and relativistic effects [5].

We would also like to use our formalism in order to be able to reason about concurrent programs in a compositional way. The model we introduce in Section 3 is (purportedly) very close to the models of programming languages traditionally considered in game semantics [11], and in particular the asynchronous variants [15], which are able to express the main operational features of the languages while enjoying compositionality. We shall investigate the compositional (categorical) structures enjoyed by our concurrent specifications in the future.

References

1. Elizabeth Borowsky and Eli Gafni. Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51. ACM, 1993.
2. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, Proceedings*, pages 420–435, 2015.
3. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.
4. 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.
5. Seth Gilbert and Wojciech Golab. Making sense of relativistic distributed systems. In Fabian Kuhn, editor, *Distributed Computing, DISC 2015*, volume 8784 of *LNCS*, pages 361–375. Springer Berlin Heidelberg, 2014.
6. Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for

concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016.

7 Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular Verification of Concurrency-Aware Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, Proceedings*, pages 371–387, 2015.

8 Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

9 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.

10 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.

11 J Martin E Hyland and C-HL Ong. On full abstraction for PCF: I, II, and III. *Information and computation*, 163(2):285–408, 2000.

12 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

13 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.

14 Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

15 Paul-André Melliès and Samuel Mimram. Asynchronous games: innocence without alternation. In *International Conference on Concurrency Theory*, pages 395–411. Springer, 2007.

16 J. Misra. Axioms for memory access in asynchronous hardware systems. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 96–110. Springer Berlin Heidelberg, 1985.

17 Gil Neiger. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

18 Mogens Nielsen. Models for concurrency. In *International Symposium on Mathematical Foundations of Computer Science*, pages 43–46. Springer, 1991.

19 Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

20 M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO Conference*, pages 314–321, 1997.

21 G. Smith, K. Winter, and R. J. Colvin. A sound and complete definition of linearizability on weak memory models. *ArXiv e-prints*, February 2018. `arXiv:1802.04954`.