Brief Announcement: On the impossibility of detecting concurrency

Éric Goubault, Jérémy Ledent and Samuel Mimram
École Polytechnique
{eric.goubault, jeremy.ledent, samuel.mimram}@lix.polytechnique.fr

Abstract

We identify a general principle of distributed computing: one cannot force two processes running in parallel to see each other. We state this principle formally in the context of asynchronous processes communicating through shared objects, using trace-based semantics. We prove that it holds in a reasonable computational model, and then study the class of concurrent specifications which satisfy this property. This allows us to derive a Galois connection theorem for different variants of linearizability.

2012 ACM Subject Classification Concurrency

Keywords and phrases concurrent specification, concurrent object, linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2016.23

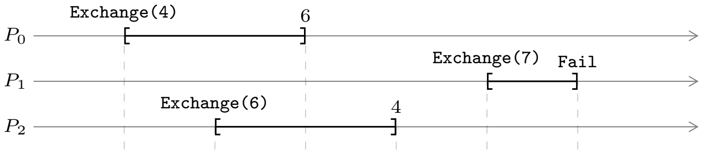
1 Introduction

A common setting to study distributed computing is that of asynchronous processes communicating through shared objects. 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.

This is simple to do when the objects that we consider are concurrent versions of sequential data structures, such as lists or queues. We simply take the usual sequential specification of the object, and extend it to a concurrent setting using one of the many correctness criteria found in the literature: atomicity [8], sequential consistency [5], serializability [10], causal consistency [11] or linearizability [4].

However, we also want to specify objects with an intrinsically concurrent nature, such as those found in the area of distributed computability [3]: consensus and set-agreement objects, immediate snapshot. Another example is Java's *Exchanger* object: two processes that call the Exchanger object concurrently can swap values. A process calling the Exchanger alone fails and receives an error value.

A very general way of specifying such objects was proposed by Lamport in [6]. The specification of a concurrent object is simply the set of all the execution traces that we consider correct for this object. For example, a correct execution trace of the Exchanger object is depicted below:



$$T \;=\; \mathsf{i}_0^4 \;\cdot\; \mathsf{i}_2^6 \;\cdot\; \mathsf{r}_0^6 \;\cdot\; \mathsf{r}_2^2 \;\cdot\; \mathsf{i}_1^7 \;\cdot\; \mathsf{r}_1^{\mathtt{Fail}}$$

The trace T consists of *invocation* events i_i^x meaning that the object was called by process i with input x, and *response* events r_i^y meaning that process i returned with output

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:3

value y. This trace can be seen as an abstraction of the real-time execution pictured above, where the horizontal axis represents global time. Formally, for a fixed set $\mathcal{V}$ of values and n processes, the set of *actions* is:

$$\mathcal{A} \quad = \quad \{\mathsf{i}_i^x \mid 0 \leq i < n \text{ and } x \in \mathcal{V}\} \cup \{\mathsf{r}_i^y \mid 0 \leq i < n \text{ and } y \in \mathcal{V}\}$$

A *trace* is a word $T \in \mathcal{A}^*$ such that for every process i, the projection of T on i starts with an invocation and alternates between invocations and responses. We write $\mathcal{T}$ for the set of all traces. Then, a *concurrent specification* in the sense of [6] is just a subset of $\mathcal{T}$.

This notion of concurrent specification is not convenient to use when reasoning about distributed systems. In fact, the correctness criteria such as linearizability can be regarded as convenient ways of defining such concurrent specifications. Starting from a sequential specification σ, we obtain $\mathsf{Lin}(\sigma) \subseteq \mathcal{T}$ which is the set of all the traces that are linearizable w.r.t. σ. The advantage of this is that sequential specifications are much easier to describe than general concurrent specifications. To specify objects with a more concurrent flavor, variants of linearizability have been described: set-linearizability [9] (a.k.a. concurrency-aware linearizability [2]) and interval-linearizability [1]. Interval-linearizability is the most expressive: it captures all the distributed *tasks*, in the sense of [3].

Contribution. In the following, we restrict to a class of concurrent specifications: those satisfying the *undetectability of concurrency* property. As it turns out, they correspond exactly to the concurrent specifications definable using interval-linearizability. We show that these are the only relevant concurrent specifications, and prove a theorem showing how the different notions of linearizability relate to this property.

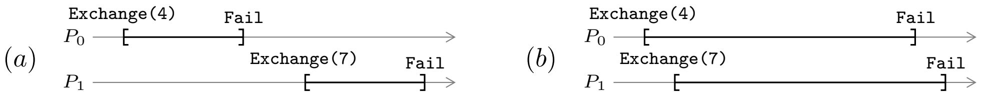
2 Results

A concurrent specification $\sigma \subseteq \mathcal{T}$ satisfies the *undetectability of concurrency* property if the following two conditions hold, where a is an action of some process $j \neq i$.

(1) *invocations commute to the left*: if $T \cdot a \cdot \mathsf{i}_i^x \cdot T' \in \sigma$, then $T \cdot \mathsf{i}_i^x \cdot a \cdot T' \in \sigma$,

(2) *responses commute to the right*: if $T \cdot \mathsf{r}_i^y \cdot a \cdot T' \in \sigma$, then $T \cdot a \cdot \mathsf{r}_i^y \cdot T' \in \sigma$.

Such properties come up in Lipton's reduction [7] proof technique: (1) and (2) assert that invocations and responses are left/right movers, respectively. Pictorially, these two properties mean that if we take a correct execution trace (a) and "expand" the intervals, then the resulting trace (b) must also be considered correct. Intuitively, in (b), the two processes failed to see each other and acted as in the sequential trace (a).



As a naive attempt at specifying the Exchanger object, we might have wanted to allow (a) and forbid (b). But implementing such a specification would have been hopeless, as we show in a reasonable trace-based computational model:

▶ **Theorem 1.** *The semantics $[\![P]\!]$ of any program P satisfies properties (1) and (2).*

Intuitively, the reason why Theorem 1 holds is that calling an object or returning a value does not communicate any information to the other processes. If a process idles right after invoking, or just before returning, it is invisible to the other processes.

The undetectability of concurrency property is naturally enforced by the usual specification techniques such as linearizability, so by using these tools we do not have to worry about this property: we get it for free.

▶ Proposition 2. Let σ be a sequential specification. Then $\mathsf{Lin}(\sigma)$, the set of all linearizable traces, satisfies properties (1) and (2).

We now write SSpec for the set of sequential specifications, and CSpec for the set of concurrent specifications which satisfy the undetectability of concurrency. Proposition 2 says that we can view Lin as a map from SSpec to CSpec. Conversely, there is also a map in the other direction $\mathsf{U} : \mathsf{CSpec} \to \mathsf{SSpec}$ which, given a concurrent specification, forgets about all the concurrent behaviors and keeps only the sequential ones.

▶ **Theorem 3.** *The functions Lin and U are monotonous w.r.t. inclusions, and form a Galois connection, i.e., for every $\sigma \in \mathsf{SSpec}$ and $\tau \in \mathsf{CSpec}$, $\mathsf{Lin}(\sigma) \subseteq \tau \iff \sigma \subseteq \mathsf{U}(\tau)$.*

The fact that we imposed the undetectability of concurrency property on CSpec is crucial in order to establish Theorem 3. This theorem can be understood as follows: given a sequential specification σ, we want to extend it to a concurrent one. Then any $\tau \in \mathsf{CSpec}$ that contains σ must also contain $\mathsf{Lin}(\sigma)$, i.e., $\mathsf{Lin}(\sigma)$ is the smallest extension of σ which is in CSpec. Thus, $\mathsf{Lin}(\sigma)$ can be described as follows: we start with the set of all sequential traces of σ, then close it under the two properties (1) and (2).

Finally, note that analogues of Proposition 2 and Theorem 3 still hold when we replace linearizability by set- or interval-linearizability. In particular, Theorem 3 for interval-linearizability gives us the following characterization of interval-linearizable objects:

▶ **Corollary 4.** *The concurrent specifications which are definable using interval-linearizability are exactly the ones satisfying the undetectability of concurrency.*

References

1 A. Castañeda, S. Rajsbaum, and M. Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *DISC 2015, Proceedings*, pages 420–435, 2015.

2 N. Hemed, N. Rinetzky, and V. Vafeiadis. Modular Verification of Concurrency-Aware Linearizability. In *DISC 2015, Proceedings*, pages 371–387, 2015.

3 M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.

4 M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

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

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

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

8 J. Misra. Axioms for memory access in asynchronous hardware systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 96–110. Springer Berlin Heidelberg, 1985.

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

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

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