AMECOS: A Modular Event-Based Framework for Concurrent Object Specification

Timothé Albouy
Univ Rennes, Inria, CNRS, IRISA, France

Antonio Fernández Anta
IMDEA Networks Institute, Madrid, Spain

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus

Mathieu Gestin
Univ Rennes, Inria, CNRS, IRISA, France

Nicolas Nicolaou
Algolysis Ltd, Nicosia, Cyprus

Junlang Wang
IMDEA Networks Institute, Madrid, Spain
Universidad Carlos III de Madrid, Spain

Abstract

In this work, we introduce a modular framework for specifying distributed systems that we call AMECOS. Specifically, our framework departs from the traditional use of sequential specification, which presents limitations both on the specification expressiveness and implementation efficiency of inherently concurrent objects, as documented by Castañeda, Rajsbaum and Raynal in CACM 2023. Our framework focuses on the interactions between the various system components, specified as concurrent objects. Interactions are described with sequences of object events. This provides a modular way of specifying distributed systems and separates legality (object semantics) from other issues, such as consistency. We demonstrate the usability of our framework by (i) specifying various well-known concurrent objects, such as registers, shared memory, message-passing, reliable broadcast, and consensus, (ii) providing hierarchies of ordering semantics (namely, consistency hierarchy, memory hierarchy, and reliable broadcast hierarchy), and (iii) presenting a novel axiomatic proof of the impossibility of the well-known Consensus problem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Concurrency, Object specification, Consistency conditions, Consensus impossibility

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2024.4

Funding This work has been partially supported by Région Bretagne, the French ANR project ByBloS (ANR-20-CE25-0002-01), the H2020 project SOTERIA, the Spanish Ministry of Science and Innovation under grants SocialProbing (TED2021-131264B-I00) and DRONAC (PID2022-140560OB-I00), the ERDF "A way of making Europe", NextGenerationEU, and the Spanish Government's "Plan de Recuperación, Transformación y Resiliencia".

1 Introduction

Motivation. Specifying distributed systems is challenging as they are inherently complex: they are composed of multiple components that *concurrently* interact with each other in unpredictable ways, especially in the face of asynchrony and failures. Stemming from this complexity, it is very challenging to compose concise specifications of distributed systems and, even further, devise correctness properties for the objects those systems may yield.

28th International Conference on Principles of Distributed Systems (OPODIS 2024).
Editors: Silvia Bonomi, Letterio Galletta, Etienne Rivière, and Valerio Schiavoni; Article No. 4; pp. 4:1–4:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To ensure the correctness of a distributed system, realized by both *safety* ("nothing bad happens") and *liveness* ("something good eventually happens") properties, the specification must capture all of the possible ways in which the system's components can interact with each other and with the system's external environment. This can be difficult, especially when dealing with complex and loosely coupled distributed systems in which components may proceed independently of the actions of others. Another challenge caused by concurrency is to specify the *consistency* of the system or the objects it implements. The order in which processes access an object greatly impacts the evolution of the state of said object. Several types of consistency guarantees exist, from weak ones such as PRAM consistency [26] to stronger ones such as Linearizability [20]. Therefore, one needs to precisely specify the ordering/consistency guarantees expected by each object the system implements.

To address the inherent complexity of distributed systems, researchers often map executions of concurrent objects to their sequential counterparts, using *sequential specifications* [4, 27]. Although easier and more intuitive for specifying how abstract data structures behave in a sequential way [35], as noted in [11], sequential specifications appear unnatural for systems where events may not be totally ordered, hindering the potential of concurrent objects. More precisely, there are concurrent objects that do not have a sequential specification (e.g., set-linearizable objects or Java's exchanger object [11]), or objects that, even if they can be sequentially specified, have an inefficient sequential implementation. For example, it is impossible to build a concurrent queue implementation that eliminates the use of expensive synchronization primitives [11].

Our approach. In this work, we propose a modular framework which we call AMECOS (from *A Modular Event-based framework for Concurrent Object Specification*) that *does not* use sequential specification of objects, but instead offers a relaxed concurrent object specification mechanism that encapsulates concurrency intuitively, alleviating the specifier from complex specifications. Here are some noteworthy *features* of our framework.

Component Identification and Interfacing: Our specification focuses on the *interface* between the various system components specified as concurrent objects. In particular, it considers objects as opaque boxes, specifying them by directly describing the intended behavior and only examining the interactions between the object and its clients. In this way, we do not conflate the specification of an object with its implementation, as is typically the case with formal specification languages such as TLA+ [25] and Input/Output Automata [28]. Specifically, these languages specify a distributed system and its components with a state machine (as a transparent box). In contrast, our formalism specifies an object at the interface level (as an opaque box). Furthermore, we avoid using higher-order logic, which sometimes can be cumbersome, and instead, we use simple logic, rendering our specification "language" simple to learn and use. In some sense, we provide the "ingredients" needed for an object to satisfy specific properties and consistency guarantees.

Modularity: Focusing on the object's interface also provides a *modular way* of specifying distributed systems and separates the object's semantics from other aspects, such as consistency. With our formalism, we can, for example, specify the semantics of a shared register [23], then specify different consistency semantics, such as PRAM [26] and Linearizability [20], *independently*, and finally combine them to obtain PRAM and atomic (i.e., linearizable) shared registers, respectively. This modularity also helps, when convenient (e.g., for impossibility proofs), to abstract away the underlying communication medium used for exchanging information. In fact, we also specify communication media as objects.

Structured Formalism: The formalism follows a *precondition/postcondition* style for specifying an object's semantics, via 3 families of predicates: *Validity*, *Safety*, and *Liveness*. The first one specifies the requirements for the use of the object (preconditions), while the other two specify the guarantees (hard and eventual) provided by the object (postconditions). This makes our formalism easy to use, providing a structured way of specifying object semantics.

Notification Operations: Another feature of the formalism is what we call *notification* operations, that is, operations that are not invoked by any particular process, but that spontaneously notify processes of some information. For example, a broadcast service provides a broadcast operation for disseminating a message in a system, but all processes of this system must be eventually notified that they received a message without invoking any operation. So, a notification is a "callback" made by an object to a process, and not by a process to an object. This feature increases our framework's expressiveness compared to formalisms that are restricted to using only invocation and response events for operations [33].

Contributions. The following list summarizes the contributions of the paper.

- We first present our framework's architecture, basic components, key concepts, and notation (see Section 3). We especially show that our framework can elegantly take into account a wide range of process failures, such as crashes or Byzantine faults (see Sections 3.5 and 6). Then, we demonstrate how concurrent objects can be specified using histories, preconditions, and postconditions (see Section 4).
- Using our formalism, we show how we can specify several classic consistency conditions, from weak ones such as PRAM consistency to strong ones such as Linearizability. Then, we show that we can define consistency conditions (set-linearizability and interval linearizability) for objects that do not have sequential specifications (see Section 5).
- Using the definitions of object specification and consistency, we show how they can be combined to yield correctness definitions for histories, even in the presence of Byzantine failures (see Section 6).
- We demonstrate our framework's effectiveness in constructing axiomatic proofs by presenting a novel, axiomatic proof of the impossibility of resilient consensus objects in an asynchronous system (see Section 7). To our knowledge, this is one of the simplest and most general proofs of this result. Its simplicity and generality stem from the fact that our formalism abstracts away the implementation details of the object or system being specified, allowing us to focus on proving intrinsic fundamental properties. For instance, our impossibility proof abstracts away the communication medium. (For completeness, we show in Section F that SWSR atomic registers and point-to-point message-passing channels satisfy the relevant assumptions of the proof.)
- Finally, to further exemplify the usability of our formalism, we specify shared memory, message passing, and reliable broadcast as concurrent objects (see Appendices C-D). The modularity of our formalism is demonstrated by combining the consistency conditions with these object specifications, obtaining shared memory and broadcast hierarchies.

We also present a comparison with related work (Section 2) and a discussion of our findings (Section 8). Due to page limitations, some developments appear in appendices.

2 Related Work

The present work addresses two different (but not unrelated) problems: object semantics specification and consistency conditions. It also deals with the Consensus impossibility.

Object semantics specification. As we already discussed, traditionally, formal definitions of concurrent objects (e.g., shared stacks or FIFO channels) are given by *sequential specifications*, which define the behavior of some object when its operations are called sequentially. Distributed algorithms are commonly defined using formal specification languages, e.g., input/output (IO) automata [28], temporal logics (e.g., LTL [32], CTL* [13], TLA [25]) and CSP [1], for implementing concurrent objects. Formal proofs are used to show that those implementations satisfy the sequential specifications of the object in any possible execution.

We argue that defining concurrent objects using such formal methods conflates the specification and implementation of said objects. On the contrary, as we already discussed in Section 1, our formalism considers objects as opaque boxes, and the specification stays at the object's interface. Furthermore, formal methods are typically complex and difficult to learn, requiring specialized tools and expertise. Our formalism, instead, relies only on simple logic (no higher-order logic is required), making it easy to learn and use. To this end, we concur that our formalism *complements* existing formal methods by providing an intuitive way to express the necessary properties a concurrent object must satisfy. Moreover, the formalism may reveal the necessary components ("ingredients") needed for an object to satisfy specific guarantees. Armed with our object specifications, formal methods may be used to specify and compose simpler components, drifting away from the inherent complexities of more synthetic distributed structures.

Consistency conditions. Consistency conditions can be seen as additional constraints on the ordering of operation calls that can be applied on top of an object semantics specification, which, in this work, we call *legality*. Over time, several very influential consistency conditions have been presented (e.g., [2, 5, 20, 24, 26]). However, all of these consistency conditions have been introduced in their own notations and context (databases, RAM/cache coherence or distributed systems), which raised the need to have a unified formalism for expressing all types of consistency. Several formalisms have been proposed [8, 9, 30, 36, 38]. We propose a formalism that uses light notation and is very expressive. As we demonstrate in Section 5, we view legality as the lowest degree of consistency, thus making a clear separation between legality and consistency. Furthermore, our formalism helps us specify consistency guarantees incrementally, moving from weaker to stronger ones, yielding a consistency hierarchy. Several works already presented consistency hierarchies and a way to combine consistencies with object specifications [30, 39]. In contrast to our framework, these approaches rely on sequential object specifications, which, as we have already discussed, can constrain expressiveness.

Possibly the work in [38] (derived from [8, 9]) is the closest to ours with respect to "specification style." However, the object specifications in [38] (and [8, 9]) use the artificial notion of *arbitration* to *always* impose a total order on object operation executions, whereas our formalism does not require a total order (unless the consistency imposes it); in general, we only consider partial orders of operation executions. Another notable difference with [38] is that their consistency specification is for storage objects, whereas we specify consistency conditions in general, and then we combine them with a specification of a shared memory object to yield a consistency hierarchy for registers (similar to storage objects considered in [38]). Additionally, compared to other endeavors such as [8, 9, 30], our framework also considers strong failure types such as Byzantine faults.

Impossibility of Asynchronous Consensus. Consensus is a fundamental abstraction where all participants of a distributed system must eventually agree on one of the values that have been proposed [27]. To demonstrate the utility of the framework, we define a Consensus

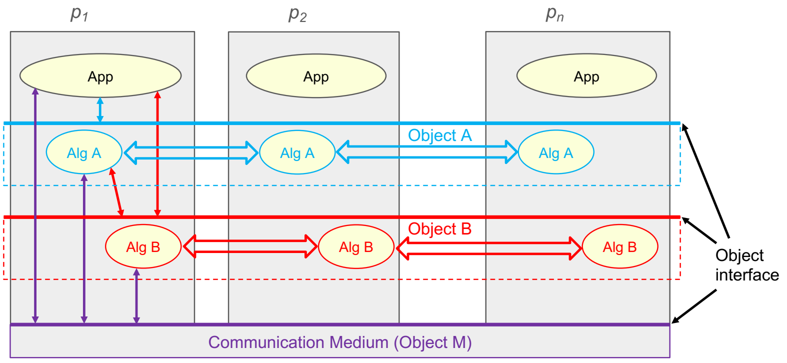

Figure 1 Example of a distributed system architecture with 3 objects. The distributed system is formed by n processes that communicate using a Communication Medium, which is Object M. The system has 2 more objects, Objects A and B, built by local modules (Alg. A and B) in each process, that cooperate using a protocol (horizontal arrows) to implement the object. Each process has an application algorithm that can execute the operations (vertical arrows) of the objects offered by their interface. Also, objects' local modules can execute operations of other objects.

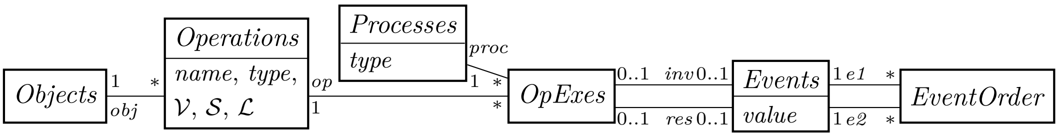

Figure 2 Class diagram for the relations and attributes of the components (sets) of the framework.

object and provide a proof of the FLP impossibility [14]. Like [37], our proof follows an axiomatic approach: the notion of asynchronous resilient consensus is defined as a system of axioms, and then it is proved that this system is inconsistent, i.e., that there is a contradiction. With the framework, the proof becomes agnostic to the communication media, unlike other proofs that assume message-passing [14, 15, 18, 40], shared memory [15, 17, 19] or other models like iterated immediate snapshot [21]. See Appendix A.6 for more details.

3 Framework: Architecture, Components, Notation, and Concepts

3.1 System Architecture

The proposed framework assumes a distributed system as depicted in Figure 1, which has a set of processes that interact with some Communication Medium (e.g., shared memory, message passing), modeled as an object. The processes have applications and local modules that implement other objects. (Processes are essentially computing entities, and they could be modeled as I/O Automata [28].) Each object offers an interface formed by operations that can be invoked by applications and modules of other objects. An execution of this system is an execution of the applications of all the processes and the execution of the object operations these applications invoke (directly or by execution of operations in other objects).

3.2 Components

In the framework, an execution of a distributed system is described with 6 (potentially infinite) sets. As shown in Figure 2, these sets are in relation to each other.

Processes: contains all *processes* p_i (of identity i) of the system execution, where the $p_i.type \in \{\texttt{correct}, \texttt{faulty}\}$ attribute is the *type* of p_i (either correct or faulty, see Section 3.5).

Objects: contains all the *objects* of the system (e.g., a channel, a register, a stack, etc.). An object is associated with a set of *operations* (the interface) that processes can execute on it.

Operations: contains all the *operations op* that can be executed on some object $op.obj \in$ *Objects*, where the *op.name* attribute is the name of *op* (e.g., write or read), the $op.type \in \{\texttt{normal}, \texttt{notif}\}$ attribute is the type of *op* (either it is a normal operation or a notification operation, see Section 3.3), and the $op.\mathcal{V}$, $op.\mathcal{S}$, and $op.\mathcal{L}$ attributes are predicates respectively defining the invocation validity, safety, and liveness of the operation (see Section 4).

OpExes: contains all *operation executions* (*op-ex* for short) o of an operation $o.op \in$ *Operations* by a process $o.proc \in$ *Processes*. When an operation is executed, it produces an invocation event and a response event. Hence, an op-ex o is the pair $(o.inv, o.res)$ of the invocation and the response events of o. If $o.op.type = \texttt{notif}$ then op-ex o has no invocation event, i.e., $o.inv = \bot$ (see Section 3.3).

Events: contains all the *events e* in *OpExes*, where the *e.value* attribute is the value of the event (i.e., the input or output value of an op-ex, see Section 3.3). The set *Events* does not contain the $\bot$ value, which denotes the absence of an event.

EventOrder: corresponds to the total order of the events in *Events*, represented as a set of event pairs $ep = (ep.e1, ep.e2)$. For the sake of simplicity, we denote this total order with $<$ over all elements of *Events* as follows: $< \triangleq \{(ep.e1, ep.e2) \mid \forall\ ep \in EventOrder\}$. The order $<$ defines the temporal ordering of events, i.e., $e{<}e'$ means event e happens before event e'.

3.3 Notation

As described, an op-ex $o \in$ *OpExes* is a pair (i, r) where $i = o.inv$ and $r = o.res$. An op-ex $o \in$ *OpExes* can have the following configurations:

- $o = (i, r)$, where $i, r \in$ *Events*, $i \neq r$, and $i < r$: in this case, o is said to be a *complete* op-ex (an operation invocation that has a matching response),
- $o = (i, \bot)$, where $i \in$ *Events*: in this case, o is said to be a *pending* op-ex (this notation is useful to denote operation calls by faulty processes or operation calls that do not halt),
- $o = (\bot, r)$, where $r \in$ *Events*: in this case, o is said to be a *notification* op-ex (i.e., its operation is not a callable operation, but an operation spontaneously invoked by the object to transmit information to its client).

The "$\equiv$" relation indicates that some op-ex o follows a given form or the same form as another op-ex: an op-ex o could be of the form "read op-ex on register R by process p_i which returned output value v", which we denote "$o \equiv R.\mathsf{read}_i()/v$". If the object, process ID, parameter, or return value are not relevant, we omit these elements in the notation: the form "$L.\mathsf{get}_i(j)/v$ op-ex on list L, process p_i, index j as input value and returned output value v" could simply be written "get()". If only some (but not all) parameters can have an arbitrary value, we can use the "$-$" notation: the form "$S.\mathsf{set}(k, v)$ op-ex on key-value store S for key k and for any value v" could be written "$S.\mathsf{set}(k, -)$". As notifications have no input, the () parameter parentheses are omitted for notifications: the form "receive notification op-ex of message m by receiver process p_i from sender process p_j" is denoted "$\mathsf{receive}_i/(m, j)$". Lastly, we denote by "op" any op-ex operation: the form "any op-ex on register R (write or read)" could thus be written "$R.\mathsf{op}()$". Pending op-exes and complete op-exes with no return value can be respectively denoted with a $\bot$ and $\varnothing$ symbol as their output value. For instance, the forms "All pending op-exes" and "All complete op-exes with no return value" can be written "$\mathsf{op}()/\bot$" and "$\mathsf{op}()/\varnothing$", respectively. By abuse of notation, to refer to any op-ex of a set O that follows a given form f, we can write $f \in O$, for example $R.\mathsf{write}(v) \in O$.

3.4 Histories

The six sets of the framework are used to describe an execution of a distributed system. We are interested in describing all possible executions of the system for a given set of *Objects* and *Operations*. Hence, each such system execution is described with *Events*, *OpExes*, *EventOrder*, and *Processes*. We call this a *history*. Note that a history captures a system execution via the events and op-exes observed in the objects' interfaces.

Hence, a history of a distributed system is a tuple $H = (E, <, O, P)$, where $E = \mathit{Events}$ is the set of events, $< = \mathit{EventOrder}$ is a *strict total order* of events in E, $O = \mathit{OpExes}$ is the set of op-exes, and $P = \mathit{Processes}$ is the set of processes. There are some natural constraints on a history that cannot be expressed directly on the diagram of Figure 2.

- **Event validity**: Every event must be part of exactly one op-ex.
- **Operation validity**: If an operation is a notification, then all its op-exes must be notification op-exes, otherwise, they must all be complete or pending op-exes.

In the sequel, we will often consider subhistories of a history $H = (E, <, O, P)$. For instance, it is useful to consider the subhistory obtained by projecting a history to one particular object. Hence, given history $H = (E, <, O, P)$ and object x, we denote by $H|x$ the subhistory containing only the events of E applied to x and the op-exes of O applied to x, and the corresponding subset of $<$. The concepts of legality, consistency, and correctness defined in the next sections can hence be applied to histories and subhistories.

3.5 Process Faults

Our framework considers two very general types of process failures: *omission faults* and *Byzantine faults*. In the framework's model presented in Figure 2, for any $p \in \mathit{Processes}$, omission faults concern p only if $p.\mathit{type} = \texttt{faulty.omitting}$, and Byzantine faults concern p only if $p.\mathit{type} = \texttt{faulty.byzantine}$. Processes p of type $p.\mathit{type} = \texttt{correct}$ are subject to none of these faults.

Omissions correspond to missing events, like op-ex invocations that do not have a matching response, for whatever reason, producing pending op-exes. We assume that such omitting processes, although they may suffer omissions at any time, follow their assigned algorithm. A *crash fault* is a special case of omission fault on a process p, where there exists a point τ in the sequence of events of the history (the crash point) such that, p has no event after τ in the sequence. Observe that omission faults also account for benign dynamic process failures like crash-recovery models.

On the other hand, Byzantine processes may arbitrarily deviate from their assigned algorithm (for instance, because of implementation errors or attacks). Strictly speaking, given that their behavior can be arbitrary, we cannot say that Byzantine processes execute actual op-exes on the same operations and objects as non-Byzantine processes. For instance, Byzantine processes may simulate op-exes that can appear legitimate to other processes. We further discuss Byzantine faults in Section 6.

Observe that by adding more constraints to the model, new failure subtypes can be derived from these two basic failure types.

4 Object Specification and History Legality

4.1 Object Specification

In our formalism, we specify an object using a set of conditions that are defined for the operations and applied to the op-exes of the object. There are two types of such conditions (that we express as predicates): preconditions (invocation validity) and postconditions (safety and liveness). Every operation of every object has a $\mathcal{V}$ precondition and two $\mathcal{S}$ and $\mathcal{L}$ postconditions (if not given, they are assumed to be satisfied). We will use the register object as an example to understand better the notations defined below. A register R is associated with two operations, $R.\mathsf{read}()/v$ and $R.\mathsf{write}(v)$, where the former returns the value of R and the latter sets the value of R to v, respectively.

Op-ex context. The context of an op-ex o is the set of all op-exes preceding o *in the same object* with respect to a binary relation $\dashrightarrow$ defined over O.

▶ **Definition 1** (Context of an op-ex)**.** *The* context *of an op-ex $o \in O$ with respect to a binary relation $\dashrightarrow$ over O is defined as $ctx(o, O, \dashrightarrow) \triangleq (O_c, \dashrightarrow_c)$, where $O_c \triangleq \{o' \in O \mid o' \dashrightarrow o, o.op.obj = o'.op.obj\}$ and $\dashrightarrow_c \triangleq \{(o', o'') \in \dashrightarrow \mid o', o'' \in O_c \cup \{o\}\}$.*

For instance, the context of a $R.\mathsf{read}()/v$ op-ex is made of all the previous op-exes of register R with respect to a given $\dashrightarrow$ relation. Note that pending op-exes can be part of the context of other op-exes, and thus influence their behavior (and especially their return value in the case of complete op-exes or notifications).

Preconditions. The *preconditions* of an object are the use requirements of this object by its client that are needed to ensure that the object works properly. Typically, a precondition for the operation on an object can require that the input (parameters) of this operation is valid, or that some op-ex required for this operation to work indeed happened before. For instance, we cannot have a read op-ex in register R if there was no preceding write op-ex in R. Another example of precondition for the $\mathsf{divide}(a, b)/d$ operation that returns the result d of the division of number a by number b, is that b must not be zero. These preconditions are given for each operation of an object by the invocation validity predicate $\mathcal{V}$.

▶ **Definition 2** (Invocation validity predicate $\mathcal{V}$)**.** *Given an operation $\mathsf{op} \in$ Operations, its* invocation validity predicate *$\mathsf{op}.\mathcal{V}(o, ctx(o, O, \dashrightarrow))$ indicates whether an op-ex $o = (i{\neq}\bot, r) \in$ OpExes of op (i.e., $o.op \equiv \mathsf{op}$) respects the usage contract of the object given its context $ctx(o, O_c, \dashrightarrow_c)$.*

Postconditions. The *postconditions* of an object are the guarantees provided by this object to its client. The postconditions are divided into two categories: *safety* and *liveness*. Broadly speaking, safety ensures that nothing bad happens, while liveness ensures that something good will eventually happen. For a given op-ex o, safety is interested in the prefix of op-exes of o (i.e., its context), while liveness is potentially interested in the whole history of op-exes. For example, for a register object R, the safety condition is that the value v returned by a $R.\mathsf{read}()/v$ is (one of) the latest written values, while the liveness condition is that the $R.\mathsf{read}()$ and $R.\mathsf{write}()$ op-exes always terminate. These postconditions are given for each operation of an object by the safety predicate $\mathcal{S}$ and the liveness predicate $\mathcal{L}$.

▶ **Definition 3** (Safety predicate $\mathcal{S}$)**.** *Given an operation* $\mathsf{op} \in \mathit{Operations}$*, its* safety predicate $\mathsf{op}.\mathcal{S}(o, \mathit{ctx}(o, O, \rightarrow))$ *indicates whether* $r.\mathit{value}$ *is a valid return value for op-ex* $o = (i, r{\neq}\bot) \in \mathit{OpExes}$ *of* op *(i.e.,* $o.\mathit{op} \equiv \mathsf{op}$*) in relation to its context* $\mathit{ctx}(o, O_c, \rightarrow_c)$*.*

We can see in the above definition that the $\mathsf{op}.\mathcal{S}(o, \mathit{ctx}(o, O, \rightarrow))$ predicate is not defined if $o = (i, \bot)$, that is, if o is a pending op-ex.

▶ **Definition 4** (Liveness predicate $\mathcal{L}$)**.** *Given an operation* $\mathsf{op} \in \mathit{Operations}$*, its* liveness predicate $\mathsf{op}.\mathcal{L}(o, H, \rightarrow)$ *indicates whether an op-ex* $o = (i, r) \in \mathit{OpExes}$ *of* op *(i.e.,* $o.\mathit{op} \equiv \mathsf{op}$*) respects the liveness specification of* op*.*

Example. As a complete example, the following is the specification of a single-writer single-reader (SWSR) shared register R using the above conditions. Let p_w and p_r be the writer and reader processes, respectively. Recall that $\mathit{ctx}(o, O, \rightarrow) = (O_c, \rightarrow_c)$ is the context of op-ex o. Let us define predicate $\mathit{OpTermination}(o) \triangleq (o.\mathit{proc.type} = \texttt{correct}) \implies (o \not\equiv \mathsf{op}()/\bot)$, which forces an op-ex o to have a response if executed by a correct process.

Operation read. If $o \equiv R.\mathsf{read}()/v$, then we have the following conditions.

$$\mathsf{read}.\mathcal{V}(o, (O_c, \rightarrow_c)) \triangleq (o.\mathit{proc} = p_r) \wedge (\exists\ o' \equiv R.\mathsf{write}(-) \in O_c).$$

$$\mathsf{read}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq v \in \{v' \mid \exists\ o' {\equiv} R.\mathsf{write}(v') \in O_c, \nexists\ o'' {\equiv} R.\mathsf{write}(-) \in O'_c : o' {\rightarrow_c} o''\}.$$

$$\mathsf{read}.\mathcal{L}(o, H, \rightarrow) \triangleq \mathit{OpTermination}(o).$$

The $\mathcal{V}$ predicate states that only the reader process can read and the register must have been previously written. The $\mathcal{S}$ predicate states that a read must return one of the latest written values. The $\mathcal{L}$ predicate prevents a read op-ex of a correct process to be pending.

Operation write. If $o \equiv R.\mathsf{write}(v)$, then we have the following condition.

$$\mathsf{write}.\mathcal{V}(o, (O_c, \rightarrow_c)) \triangleq o.\mathit{proc} = p_w. \qquad \mathsf{write}.\mathcal{L}(o, H, \rightarrow) \triangleq \mathit{OpTermination}(o).$$

Observe that the $\mathcal{S}$ predicate is not provided and is hence assumed to be always satisfied.

4.2 History Legality

We now define the notion of history legality.

▶ **Definition 5** (History validity, safety, and liveness)**.** *Given a history* $H = (E, <, O, P)$ *and a relation* $\rightarrow$ *on* O*, the following predicates define the* validity*,* safety*, and* liveness *of* H*.*

$$\mathit{Validity}(H, \rightarrow) \triangleq \forall\ o = (i{\neq}\bot, r) \in O : o.\mathit{op}.\mathcal{V}(o, \mathit{ctx}(o, O, \rightarrow)).$$

$$\mathit{Safety}(H, \rightarrow) \triangleq \forall\ o = (i, r{\neq}\bot) \in O : o.\mathit{op}.\mathcal{S}(o, \mathit{ctx}(o, O, \rightarrow)).$$

$$\mathit{Liveness}(H, \rightarrow) \triangleq \forall\ o = (i, r) \in O : o.\mathit{op}.\mathcal{L}(o, H, \rightarrow).$$

Notice that, for an op-ex o, if o is a notification, we do not need to verify its invocation validity, and if o is a pending op-ex, we do not need to verify its safety.

▶ **Definition 6** (Legality condition)**.** *Given a history* $H = (E, <, O, P)$ *and a relation* $\rightarrow$ *on* O*, the* legality condition *is defined as*

$$\mathit{Legality}(H, \rightarrow) \triangleq \{\mathit{Validity}(H, \rightarrow), \mathit{Safety}(H, \rightarrow), \mathit{Liveness}(H, \rightarrow)\}.$$

Legality is defined as a set of clauses (or constraints) on a history H and an op-ex relation $\rightarrow$. Informally, a history H is legal if and only if all clauses of $\mathit{Legality}(H, \rightarrow)$ evaluate to `true`.

5 Consistency Conditions

In the previous section we have presented how object legality can be specified using operation conditions, abstracting the consistency model with a binary order relation $\rightarrow$. In this section we describe how to define order relations $\rightarrow$ to extend legality with different consistency conditions. We first define reusable predicates describing certain constraints on the op-ex order $\rightarrow$ (Section 5.1) and then we define common consistency conditions (Section 5.2) to showcase the power of the formalism. In addition, we provide the definitions of set-linearizability [29] and interval-linearizability [10], which are new interesting consistency conditions; objects with these consistencies do not have sequential specifications [11] (Section 5.3).

5.1 Op-ex Order Relations

We first define partial and total orders within our framework.

▶ **Definition 7** (Generic strict orders). *Given an arbitrary set S and an arbitrary binary relation $\prec$ over the elements of S, the following predicates define* strict partial order *and* strict total order*.*

$$\mathit{PartialOrder}(S,\prec) \triangleq (\forall\ e \in S : e \not\prec e) \land (\forall\ e, e', e'' \in S : e \prec e' \prec e'' \implies e \prec e'').$$
$$\mathit{TotalOrder}(S,\prec) \triangleq \mathit{PartialOrder}(S,\prec) \land (\forall\ e, e' \in S, e \neq e' : e \prec e' \lor e' \prec e).$$

We call these orders "strict" because they are irreflexive. Observe that the asymmetry property is redundant for strict orders because it directly follows from irreflexivity and transitivity. As we can also see, total order adds the connectedness property to partial order.

▶ **Definition 8** (Basic orders). *Given a history $H = (E, <, O, P)$ and a relation $\rightarrow$ on the set of op-exes O, the following predicates define* history order*,* process order *and* FIFO order*:*

$$\mathit{HistoryOrder}(H,\rightarrow) \triangleq \forall\ o = (i,r), o' = (i',r') \in O, r \neq \bot :$$
$$((i' \neq \bot \land r{<}i') \lor (i' = \bot \land r{<}r')) \implies (o{\rightarrow}o' \land o' \not\rightarrow o).$$
$$\mathit{ProcessOrder}(H,\rightarrow) \triangleq \forall\ p_i \in P : \mathit{HistoryOrder}(H|p_i,\rightarrow) \land \mathit{TotalOrder}(O|p_i,\rightarrow).$$
$$\mathit{FIFOOrder}(H,\rightarrow) \triangleq \forall\ o_i, o'_i \in O|p_i, o_j, o'_j \in O|p_j :$$
$$(o_i{\rightarrow}o'_i{\rightarrow}o_j{\rightarrow}o'_j \land o_i{\rightarrow}o'_j) \implies (o_i{\rightarrow}o_j \land o'_i{\rightarrow}o'_j).$$

Note that the above predicates do not define "classic" order relations (strict or not) per se, as they do not guarantee all the required properties. These predicates define how a "visibility" relation $\rightarrow$ between op-exes of history H should look in different contexts, in the sense that the behavior of an op-ex is determined by the set of op-exes it "sees".

- In *HistoryOrder*, we check if $\rightarrow$ respects the event order $<$: if two op-exes are not concurrent with respect to the $<$ order, then the oldest one must precede the newest one. Note that we distinguish whether the second op-ex o' is a notification or not.
- In *ProcessOrder*, we check if $\rightarrow$ totally orders the op-exes of each process p_i while also respecting the event order of p_i.
- In *FIFOOrder*, we check that, if an op-ex of any given process sees some other op-ex of another process, then it also sees all the previous op-exes of the latter process. Furthermore, we also check that the set of op-exes seen by the op-exes of a given process is monotonically increasing, i.e., that a given op-ex sees all the op-exes that its predecessors (of the same process) saw. More details about *FIFOOrder* can be found in Section B.

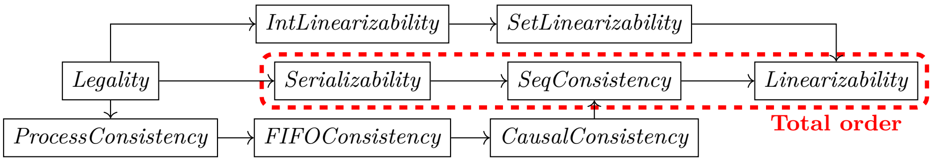

Figure 3 Hierarchy of the consistencies defined in this paper and their relative strengths [30].

5.2 Classic Consistency Conditions

This section defines common consistency conditions from the literature. They are represented as sets of clauses, extending those in *Legality* and constraining the $\rightarrow$ op-ex order.

▶ **Definition 9** (Classic consistency conditions). *Given a history $H = (E, <, O, P)$ and a relation $\rightarrow$ on O, the following sets of clauses respectively define* process consistency*,* FIFO consistency [26]*,* causal consistency [2, 31]*,* serializability [5]*,* sequential consistency [24] *and* linearizability [20]*.*

$$\begin{aligned}
\mathit{ProcessConsistency}(H,\rightarrow) &\triangleq \mathit{Legality}(H,\rightarrow) \cup \{\mathit{ProcessOrder}(H,\rightarrow)\}.\\
\mathit{FIFOConsistency}(H,\rightarrow) &\triangleq \mathit{ProcessConsistency}(H,\rightarrow) \cup \{\mathit{FIFOOrder}(H,\rightarrow)\}.\\
\mathit{CausalConsistency}(H,\rightarrow) &\triangleq \mathit{FIFOConsistency}(H,\rightarrow) \cup \{\mathit{PartialOrder}(O,\rightarrow)\}.\\
\mathit{Serializability}(H,\rightarrow) &\triangleq \mathit{Legality}(H,\rightarrow) \cup \{\mathit{TotalOrder}(O,\rightarrow)\}.\\
\mathit{SeqConsistency}(H,\rightarrow) &\triangleq \mathit{Serializability}(H,\rightarrow) \cup \mathit{CausalConsistency}(H,\rightarrow).\\
\mathit{Linearizability}(H,\rightarrow) &\triangleq \mathit{SeqConsistency}(H,\rightarrow) \cup \{\mathit{HistoryOrder}(H,\rightarrow)\}.
\end{aligned}$$

Note that the above *Serializability* condition strays away from the traditional definition of serializability, as it considers that a transaction (originally defined as an atomic sequence of op-exes [5]) is the same as a single op-ex. Likewise, as discussed in detail in Section B, our definition of *FIFOConsistency* differs from the traditional definition of PRAM consistency.

We illustrate in Figure 3 the relations between all the consistency conditions defined in this section. In this figure, if we have $\mathcal{C}_1 \rightarrow \mathcal{C}_2$ for two consistency conditions $\mathcal{C}_1$ and $\mathcal{C}_2$, then it means that $\mathcal{C}_2$ is stronger than $\mathcal{C}_1$, and thus, that $\mathcal{C}_2$ imposes more constraints on the order of op-exes. The conditions inside the red rectangle are conditions that impose a total order of op-exes. Combining these consistency conditions with other object specifications allows us to obtain multiple consistent object specifications (see Section C.2).

5.3 Set- and Interval-Linearizability Definitions

This section adds set-linearizability [29] and interval-linearizability [10] to the repertoire of consistency conditions supported by the framework. These are interesting because they define objects with no sequential specification [11]. We first define interval and set orders.

▶ **Definition 10** (Set and interval orders). *Given a history $H = (E, <, O, P)$ and a relation $\rightarrow$ on O, the following predicates define* interval *and* set orders*:*

$$\begin{aligned}
\mathit{IntOrder}(H,\rightarrow) \triangleq{}& (\forall\, o \in O : o \not\rightarrow o) \wedge (\forall\, o, o' \in O, o \neq o' : o \rightarrow o' \vee o' \rightarrow o)\\
&\wedge (\forall\, o, o', o'' \in O : o \rightarrow o'' \implies (o \rightarrow o' \vee o' \rightarrow o'')).\\
\mathit{SetOrder}(H,\rightarrow) \triangleq{}& \mathit{IntOrder}(H,\rightarrow) \wedge (\forall\, o, o', o'' \in O, o \neq o'' : o \rightarrow o' \rightarrow o'' \implies o \rightarrow o'').
\end{aligned}$$

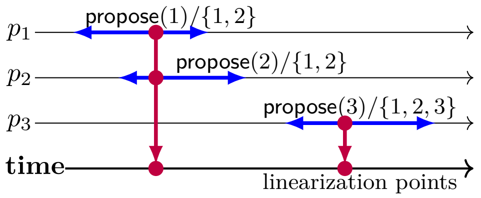

Figure 4 A set-linearizable execution of lattice agreement that is not linearizable.

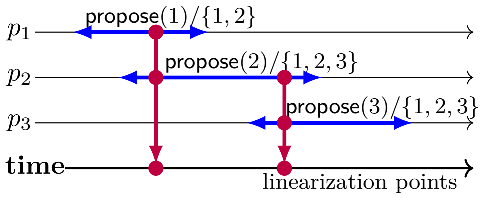

Figure 5 An interval-linearizable execution of lattice agreement that is not set-linearizable.

In *IntOrder*, an op-ex is represented as a time interval, and we check that it can see only all op-exes with which it overlaps, and all previous op-exes. The first clause guarantees irreflexivity (an op-ex cannot see itself), the second connectedness (all op-exes are in relation with each other), and the last one ensures that no forbidden pattern is present.

In *SetOrder*, we check that an op-ex can see only all other op-exes of its equivalence class (except itself), and all previous op-exes. In addition to *IntOrder*, *SetOrder* guarantees a weakened version of transitivity, allowing two-way cycles between two or more op-exes, thus creating equivalence classes. Let us remark that the weakened transitivity property of *SetOrder* implies the last clause of *IntOrder*.

Leveraging the above order relations, we can define set- and interval-linearizability.

▶ **Definition 11** (Set- and interval-linearizability)**.** *Given a history* $H = (E, <, O)$ *and a relation* $\rightarrow$ *on* O*, the following predicates define* set-linearizability [29] *and* interval-linearizability [10].

$$IntLinearizability(H, \rightarrow) \triangleq Legality(H, \rightarrow) \cup \{HistoryOrder(H, \rightarrow), IntOrder(H, \rightarrow)\}.$$
$$SetLinearizability(H, \rightarrow) \triangleq IntLinearizability(H, \rightarrow) \cup \{SetOrder(H, \rightarrow)\}.$$

To illustrate the set- and interval-linearizability consistency conditions, we provide some examples of executions of lattice agreement in Figures 4 and 5, taken from [11]. Lattice agreement is an object that provides a single operation $\mathsf{propose}(v)/V$, where v is a value and V is a set of proposed values. Its only safety property is that V must contain all previously or concomitantly proposed values along with the value being proposed, and its only liveness property is that the propose operation must eventually terminate for correct processes.

In the set-linearizability example of Figure 4, op-exes form two equivalence classes $\{\mathsf{propose}(1), \mathsf{propose}(2)\}$ and $\{\mathsf{propose}(3)\}$. The last clause of *SetOrder* enables the creation of said equivalence classes. Indeed, we have $\mathsf{propose}(1) \rightarrow \mathsf{propose}(2) \rightarrow \mathsf{propose}(3)$ and $\mathsf{propose}(1) \rightarrow \mathsf{propose}(3)$. Besides, we also have $\mathsf{propose}(2) \rightarrow \mathsf{propose}(1) \rightarrow \mathsf{propose}(3)$ and $\mathsf{propose}(2) \rightarrow \mathsf{propose}(3)$. This shows that the forbidden pattern in set-linearizability is, for any op-exes o, o', o'' such that $o \neq o''$, there is $o \rightarrow o' \rightarrow o''$ and $o'' \not\rightarrow o' \not\rightarrow o$. Hence, the weakened transitivity clause of *SetOrder* precludes this pattern. Note that the $o \neq o''$ condition in this clause prevents the contradiction of this clause with the irreflexivity property.

In the interval-linearizability example of Figure 5, equivalence classes can be more complex. More precisely, two equivalence classes can intersect, but it does not necessarily imply that both equivalence classes can "see" each other. Here, op-exes form two different equivalence classes $\{\mathsf{propose}(1), \mathsf{propose}(2)\}$ and $\{\mathsf{propose}(2), \mathsf{propose}(3)\}$. This shows that the forbidden pattern in interval-linearizability is: for any op-exes o, o', o'' that are connected but not concurrent, i.e., $(o \rightarrow o' \rightarrow o'' \wedge o'' \not\rightarrow o' \not\rightarrow o)$, we also have $o'' \rightarrow o$. The clause precluding this pattern is thereby $(o \rightarrow o' \rightarrow o'' \wedge o'' \not\rightarrow o' \not\rightarrow o) \implies o'' \not\rightarrow o$. However, because of the connectedness property, the $o \rightarrow o' \rightarrow o''$ part of the implication is redundant, and the formula can be simplified to $o'' \not\rightarrow o' \not\rightarrow o \implies o'' \not\rightarrow o$. Finally, by applying the contrapositive, we obtain the formulation of the clause that appears in *IntOrder*: $o \rightarrow o'' \implies (o \rightarrow o' \vee o' \rightarrow o'')$.

6 History Correctness

Thus far, we have defined *Legality* (Section 4) and extended it to the *Consistency* (Section 5) of a history H with respect to an op-ex order $\rightarrow$, as a set of clauses $\mathcal{C}$. We now define the correctness of a history H with respect to a set of clauses $\mathcal{C}$ when no process is Byzantine.

▶ **Definition 12** (Correctness predicate). *Given a history $H = (E, <, O, P)$ and a set of clauses $\mathcal{C}$, the following predicate describes the correctness of H with respect to $\mathcal{C}$:*

$$Correctness(H, \mathcal{C}) \triangleq \exists \rightarrow \in O^2 : \bigwedge_{C \in \mathcal{C}} C(H, \rightarrow).$$

Intuitively, a history H is correct with respect to a set of clauses $\mathcal{C}$ if it is possible to find a relation $\rightarrow$ on the op-exes of H, such that all clauses in $\mathcal{C}$ are satisfied. As an example, $Correctness(H, ProcessConsistency)$ is the predicate that decides whether history H is correct under *ProcessConsistency*, which according to its definition in Section 5, requires that the clauses composing *Legality* (see Section 4) and *OpProcessOrder* (see Section 5) are satisfied by H. Note by the above definition of correctness, it is apparent that the more clauses are present, the fewer histories, and thus executions, will satisfy all the clauses. This demonstrates that when stronger, more restrictive, semantics are considered, the more refined is the set of executions that can provide them.

In a similar fashion, we can derive a more general definition where processes may exhibit Byzantine behavior. To model the set of all possible Byzantine behaviors, we introduce the *ByzHistories* function, which, given a history H, returns the set of all modified histories H', where the op-exes by non-Byzantine (i.e., `correct` or `omitting`) processes are the same in H and H', but Byzantine processes are given any arbitrary set of pending op-exes.

▶ **Definition 13** (Byzantine histories function). *Given history $H = (E,<,O,P)$, the $ByzHistories(H)$ function returns the set of all possible histories $H' = (E', <', O', P)$ s.t.*

$$O' = \{o \in O \mid o.proc.type \neq \texttt{faulty.byzantine}\} \cup \{\textit{any arbitrary set of pending}$$
$$\textit{op-exes by } p \mid \forall\, p \in P, p.type = \texttt{faulty.byzantine}\},$$
$$E' = \{i, r \in (i, r) \in O'\}, \textit{ and } <' \subseteq E'^2 : < \subseteq <'.$$

Informally, given a base history $H = (E, <, O, P)$ and a modified history $H' = (E', <', O', P) \in ByzHistories(H)$, the set O' is constructed by keeping all op-exes of O by non-Byzantine processes and creating arbitrary pending op-exes for Byzantine processes, the set E' is the set of all events appearing in O', and the order $<'$ is an arbitrary total order on E' extending $<$. Notice that we only populate the op-exes of Byzantine processes using pending op-exes, and not complete op-exes or notifications, as we do not guarantee anything for Byzantine processes. Hence, we define correctness with Byzantine processes as follows.

▶ **Definition 14** (Byzantine Correctness predicate). *Given a history $H = (E, <, O, P)$ and a set of clauses $\mathcal{C}$, the following predicate describes the Byzantine correctness of H:*

$$Correctness(H, \mathcal{C}) \triangleq \exists\, H' = (E', <', O', P) \in ByzHistories(H), \exists \rightarrow \in O'^2 : \bigwedge_{C \in \mathcal{C}} C(H', \rightarrow).$$

Intuitively, a history H with Byzantine processes is correct with respect to a set of clauses $\mathcal{C}$ if it is possible to construct a modified history H' (where Byzantine processes perform arbitrary op-exes) and an arbitrary relation $\rightarrow$ on the op-exes of H', such that all clauses in $\mathcal{C}$ are satisfied. To create the set of all possible modified histories, we use the *ByzHistories* function. In other words, history H is correct if and only if we can "fix" it by changing only the op-exes of the Byzantine processes to make it correct with respect to $\mathcal{C}$. In the absence of Byzantine processes, Definition 14 collapses to Definition 12.

7 Asynchronous Resilient Consensus Impossibility

This section further exemplifies the framework's utility by showing how it can be used to construct axiomatic proofs. Particularly, we demonstrate the FLP impossibility of having consensus in an asynchronous system with process failures [14]. Due to page limitation, we provide a high-level overview of the proof in this section, but the full developments can be found in Section A.

The Consensus object. Without loss of generality, we consider a simplified version of the consensus object C, which only has one notification operation, $C.\mathsf{decide}_i(v)$, returning a value $v \in V$ (we have binary consensus if $V = \{0, 1\}$) to process p_i. *Proving the impossibility of this weaker version makes our proof more general.* We give in the following the semantics of the decide notification operation. If $o \equiv C.\mathsf{decide}_i/v$, then we have the following predicates.

$$\mathsf{decide}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq (v \in V) \wedge (\forall\ C.\mathsf{decide}_j/v' \in O_c : v = v').$$

$$\mathsf{decide}.\mathcal{L}(o, H, \rightarrow) \triangleq \exists\ C.\mathsf{decide}_j/- \in O.$$

The $\mathcal{S}$ predicate states that the values decided are in the appropriate set V and that, in the context of each op-ex, all decided values are the same. Observe that we allow the same process to decide several times as long as the decided values are the same. The $\mathcal{L}$ predicate states that some process must decide.

Set of states of the system (Σ). In the proof, we consider a system with a consensus object C and a communication medium object M. We use the (potentially infinite) set *Histories* to capture all the histories of the system executions. From each execution, a history is obtained that contains only the events of objects C and M in the order they occurred. From these histories, we will construct a (potentially infinite) set Σ of possible states of the system. Each *state* $\sigma \in \Sigma$ is a (potentially infinite) set of events. Intuitively, a state σ is the collection of local states of all system processes p_i, represented by the totally-ordered local events that p_i has experienced. We also assign an *index* to each event in a history $H = (E, <, O, P) \in \mathit{Histories}$. The index assigned to event $e \in E$ is an attribute $e.\mathit{idx}$ that is the position of e in the sequence of events of its process $e.\mathit{proc}$. This sequence is obtained by ordering with $<$ the set $E|e.\mathit{proc}$.

A special subset of Σ is the set of *complete states*, defined as $\mathit{Complete}(\Sigma) \triangleq \{E \mid (E, <, O, P) \in \mathit{Histories}\}$. Consider now any history $H = (E, <, O, P) \in \mathit{Histories}$. We say that state $\sigma = E \in \mathit{Complete}(\Sigma) \subseteq \Sigma$ is a *state extracted from H*.

The *Continuity* property. This property ensures that, from every nonempty state $\sigma \in \Sigma$, a previous substate $\sigma' = \sigma \setminus \{e\}$ can be obtained by removing one event e using the following recursive procedure exhaustively. If $\sigma \in \Sigma$ is a state extracted from H, let e be the event in σ with the largest index $e.\mathit{idx}$ of those from process $e.\mathit{proc}$, then $\sigma' = \sigma \setminus \{e\}$ is also *a state in Σ extracted from H*. This procedure ends when the empty state $\varnothing$ is reached (which is also in Σ). Hence, Σ contains a state σ iff there is a history $H = (E, <, O, P) \in \mathit{Histories}$ such that the events in σ from every process $p \in P$ are a prefix of the sequence of all events from p in E ordered by $<$.

The *Asynchrony* axiom. An asynchronous system's set of states Σ must satisfy the *Asynchrony* axiom, which requires that, if two states differ only in their last respective events, which are from different processes, their union is also a state. Observe that this must hold even if the two states are extracted from different histories. We point out that our impossibility proof is agnostic of the communication medium object M, as long as the medium satisfies asynchrony as defined above. For completeness we show in Section F that atomic registers and message-passing objects satisfy this axiom.

The *Val* function. Our impossibility proof relies on the notion of *valence*, which was first introduced in [14]. We represent it as a function $Val(\sigma)$, taking a state σ as a parameter, and returning the set of all possible decision values that can be reached from this state (by extending σ with more events, or because σ already contains some decide events). The valence of a complete state is the set of all values that were decided in the histories from which it was extracted. Moreover, the valence of an incomplete state is the union of the valences of all the states obtained by adding one event to it. We call this last property *Branching*. We say that a state $\sigma \in \Sigma$ is *univalent* iff we have $|Val(\sigma)| = 1$, and we say that it is *multivalent* iff we have $|Val(\sigma)| > 1$.

The *NonEmptyValence* lemma. This lemma uses the $\mathcal{L}$ predicate of the C object to show that there is no state σ with an empty valence, $|Val(\sigma)| = 0$ (proof in Section A).

The *Termination* lemma. This lemma shows that all complete states have a finite univalent sub-state (proof in Section A).

The *Resilience* axiom. This axiom states that, for any process, any multivalent state can be extended by an event that is not from this process. This ensures that even if one process stops taking steps (i.e., crashes), the system can still progress and reach a decision.

The *NonTriviality* axiom. This axiom states that there exist 2 states with different valences, implying that there are histories deciding different values.

The impossibility theorem. The proof of the impossibility of asynchronous resilient consensus consists in showing that a contradiction lies in the system of axioms that we have created. Firstly, we show that *NonTriviality*, *Continuity*, and *Branching* imply the existence of a multivalent state. From this multivalent state and using *Termination*, we can find a *critical* state, i.e., a multivalent state for which all extensions are univalent. By *Branching* from the critical state, we can find two univalent 1-event extensions with different valences. Let us consider the two distinct events, e and e', that extend the critical state into these two superstates with different valences. If $e.proc \neq e'.proc$, then *Asynchrony* applies, and we can create an extension of the critical state containing both e and e', however, by *Branching* and *NonEmptyValence*, this state would be multivalent, which contradicts the definition of a critical state. Otherwise, if $e.proc = e'.proc$, then by *Resilience*, we can create another 1-event extension of the critical state, whose valence is different from at least one of the two previous extensions (that of e or that of e'). This makes us fall back to the case $e.proc \neq e'.proc$, again yielding a contradiction. Therefore, in all cases, the system of axioms leads to a contradiction. (The details of the proof can be found in Section A).

8 Conclusion

In this paper, we have introduced a modular framework for specifying distributed objects. Our approach departs from sequential specifications, and it deploys simple logic for specifying the interface between the system's components as concurrent objects. It also separates the object's semantics from other aspects such as consistency and failures, while providing a structured precondition/postcondition style for specifying objects. We demonstrate the usability of our framework by specifying communication media, services, and even problems, as objects. With our formalism, we also provide a proof of the impossibility of consensus that is agnostic of the medium used for inter-process communication. The simple specification examples we presented in this paper were for illustration and understanding the formalism. Of course, we acknowledge that some combinations of system model, object, and consistency may not be specified with the current version of the framework. We are confident that our framework's expressiveness (via the specification and combination of concurrent objects) enables the specification of more complex distributed systems, including ones with dynamic node participation. As our formalism gets used and flourishes with object definitions, its usefulness will be apparent both to distributed computing researchers and practitioners seeking for a modular specification of complex distributed objects. In addition, we plan to explore how to feed our specification into proof assistants such as Coq [22] and Agda [6].

References

1 Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.

2 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Comput.*, 9(1):37–49, 1995. doi:10.1007/BF01784241.

3 Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani. Asynchronous Byzantine reliable broadcast with a message adversary. *Theor. Comput. Sci.*, 978:114110, 2023. doi:10.1016/J.TCS.2023.114110.

4 Hagit Attiya and Jennifer L. Welch. *Distributed computing – Fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

5 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

6 Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03359-9_6.

7 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

8 Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, 2014. doi:10.1561/2500000011.

9 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, pages 271–284. ACM, 2014. doi:10.1145/2535838.2535848.

10 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

11 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. A linearizability-based hierarchy for concurrent specifications. *Commun. ACM*, 66(1):86–97, 2023. doi:10.1145/3546826.

12 Shir Cohen and Idit Keidar. Tame the wild with Byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. In *Proc. 35th Int'l Symposium on Distributed Computing (DISC'21)*, volume 209 of *LIPIcs*, pages 18:1–18:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.18.

13 E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. In *Proc. 10th ACM Symposium on Principles of Programming Languages (POPL'83)*, pages 127–140. ACM Press, 1983. doi:10.1145/567067.567081.

14 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

15 Eli Gafni and Giuliano Losa. Invited paper: Time is not a healer, but it sure makes hindsight 20:20. In *Proc. 25th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'23)*, volume 14310 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 2023. doi:10.1007/978-3-031-44274-2_6.

16 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

17 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

18 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proc 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 133–142. ACM, 1998. doi:10.1145/277697.277722.

19 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

20 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

21 Gunnar Hoest and Nir Shavit. Towards a topological characterization of asynchronous complexity (preliminary version). In *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, pages 199–208. ACM, 1997. doi:10.1145/259380.259440.

22 Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant a tutorial. *Rapport Technique*, 178, 1997. URL: http://www.itpro.titech.ac.jp/coq.8.2/Tutorial.pdf.

23 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

24 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

25 Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. doi:10.1145/177492.177726.

26 Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton University, 1988.

27 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

28 Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 137–151. ACM, 1987. doi:10.1145/41840.41852.

29 Gil Neiger. Set-linearizability. In *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, page 396. ACM, 1994. doi:10.1145/197917.198176.

30 Matthieu Perrin. *Spécification des objets partagés dans les systèmes répartis sans-attente. (Specification of shared objects in wait-free distributed systems).* PhD thesis, University of Nantes, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01390700.

31 Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Causal consistency: beyond memory. In *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*, pages 26:1–26:12. ACM, 2016. doi:10.1145/2851141.2851170.

32 Amir Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society, 1977. `doi:10.1109/SFCS.1977.32`.

33 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems – An Algorithmic Approach.* Springer, 2018. `doi:10.1007/978-3-319-94141-7`.

34 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proc. 6th Symposium on Theoretical Aspects of Computer Science (STACS'89)*, volume 349 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 1989. `doi:10.1007/BFB0028994`.

35 Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011. `doi:10.1145/1897852.1897873`.

36 Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004. `doi:10.1145/1017460.1017464`.

37 Gadi Taubenfeld. On the nonexistence of resilient consensus protocols. *Inf. Process. Lett.*, 37(5):285–289, 1991. `doi:10.1016/0020-0190(91)90221-3`.

38 Paolo Viotti and Marko Vukolic. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, 2016. `doi:10.1145/2926965`.

39 Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *Proc. 17th Int'l Conference on Distributed Computing (DISC'03)*, volume 2848 of *Lecture Notes in Computer Science*, pages 92–105. Springer, 2003. `doi:10.1007/978-3-540-39989-6_7`.

40 Hagen Völzer. A constructive proof for FLP. *Inf. Process. Lett.*, 92(2):83–87, 2004. `doi:10.1016/J.IPL.2004.06.008`.

A Impossibility of Resilient Consensus in Asynchronous Systems

In this section, we use our framework to define a Consensus object and to provide an axiomatic proof of the FLP impossibility of having reliable deterministic consensus in an asynchronous system with process failures [14]. (This proof is inspired by [37].) After the proof a discussion on its relevance is included.

A.1 Consensus Object

We start by providing the specification of a Consensus object using the conditions defined in Section 4. Our Consensus object C has only one notification operation, $C.\mathsf{decide}_i(v)$, which returns a value $v \in V$ (we have binary consensus if $V = \{0,1\}$) to process p_i. Observe that we consider a simple version of a Consensus object without the common propose operation. Proving the impossibility of this version makes our proof more general[1].

Let *Histories* be the set of histories of a distributed system that contains a Consensus object C. For every history $H = (E, <, O, P) \in \mathit{Histories}$, let $H|C = (E|C, <|C, O|C, P)$ be the subhistory containing only the events of E applied to C. Consider history $H = (E, <, O, P) \in \mathit{Histories}$ with set of op-exes O, a relation $\rightarrow$ on O, an op-ex $o \in O|C$, and its context $\mathit{ctx}(o, O, \rightarrow) = (O_c, \rightarrow_c)$.

Operation decide. If $o \equiv C.\mathsf{decide}_i/v$, then we have the following predicates.

$$\mathsf{decide}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq (v \in V) \wedge (\forall\ C.\mathsf{decide}_j/v' \in O_c : v = v').$$

$$\mathsf{decide}.\mathcal{L}(o, H, \rightarrow) \triangleq \exists\ C.\mathsf{decide}_j/- \in O.$$

[1] We could add a $C.\mathsf{propose}_i(v)$ operation to the Consensus object that returns nothing. The validity predicate of the decide() notification has to be adapted accordingly, but this does not affect the proof.

The $\mathcal{S}$ predicate states that the values decided are in the appropriate set V and that, in the context of each op-ex, all decided values are the same. Observe that we allow the same process to decide several times as long as the decided values are the same. The $\mathcal{L}$ predicate states that some process must decide in every history.

The Consensus object must guarantee that exactly one value can be decided in each history. We achieve this by combining the Consensus object specification with the *Serializability* consistency.

▶ **Assumption 15.** *For every history* $H = (E, <, O, P) \in \mathit{Histories}$*, we must have* $\mathit{Correctness}(H|C, \mathit{Serializability})$*.*

Observe that *Serializability* is only imposed on $H|C$, that is, we only impose a total order on the decide op-exes. From the fact that $\rightarrow|C$ is a total order of the op-exes on object C (imposed by *Serializability*), and the last clause of decide.$\mathcal{S}()$, all decide op-exes in O return the same value v.

▶ **Observation 16.** $\forall\, H = (E, <, O, P) \in \mathit{Histories}, (C.\mathsf{decide}_i/v_i, C.\mathsf{decide}_j/v_j \in O) \implies (v_i = v_j)$*.*

Observe that it is possible to have trivial implementations of a Consensus object in which all histories decide the same hardcoded value $v \in V$. Unfortunately, this object is not very useful. We will impose below a non-triviality condition that guarantees that there are histories in which the Consensus object decides different values. Additionally, the set of *Histories* must reflect the fact that the system is asynchronous and has n processes of which up to one can crash.

A.2 Asynchronous Distributed System

We consider an asynchronous distributed system with n processes in which up to one process can crash. This means that in any history $H = (E, <, O, P) \in \mathit{Histories}$ of the system $|P| = n$ and at most one process $p \in P$ has $p.\mathit{type} = \texttt{faulty}$. For convenience we assume that $P = \{p_1, p_2, \ldots, p_n\}$ in all histories.

The set of objects *Objects* contains a crash-resilient Consensus object C. In order to be able to solve consensus, it also contains some object M that allows processes to communicate. Observe that the events of this communication medium M are in the system histories.

The (potentially infinite) set *Histories* represents the system executions. From these histories, we will construct a (potentially infinite) set Σ of possible states of the system. Each *state* $\sigma \in \Sigma$ is a (potentially infinite) set of events. Intuitively, a state σ is the collection of local states of all system processes p_i, represented by the totally-ordered local events that p_i has experienced.

To define the set Σ, we first assign an *index* to each event in a history $H = (E, <, O, P) \in \mathit{Histories}$. The index assigned to event $e \in E$ is an attribute $e.\mathit{idx}$ that is the position of e in the sequence of events of its process $e.\mathit{proc}$. This sequence is obtained by ordering with $<$ the set $E|e.\mathit{proc}$. Observe that the sets of events of different histories may have common events. After adding the indices, common events with the same index are the same, but with different indices are different. For instance, the indices distinguish events in two histories in which the same process receives the same messages from the same senders but in different orders.

A special subset of Σ is the set of *complete states*, defined as $\mathit{Complete}(\Sigma) \triangleq \{E \mid (E, <, O, P) \in \mathit{Histories}\}$. Consider now any history $H = (E, <, O, P) \in \mathit{Histories}$. We say that state $\sigma = E \in \mathit{Complete}(\Sigma) \subseteq \Sigma$ is a *state extracted from* H. Then, we apply iteratively and exhaustively the following procedure to add more states to Σ: if $\sigma \in \Sigma$ is a state extracted from H, let e be the event in σ with the largest index $e.idx$ of those from process $e.proc$, then $\sigma \setminus \{e\}$ is also a state in Σ extracted from H. This procedure ends when the empty state $\sigma = \varnothing$ is reached (which is also in Σ). Hence, Σ contains a state σ iff there is a history $H = (E, <, O, P) \in \mathit{Histories}$ such that the events in σ from every process $p \in \mathit{Processes}$ are a prefix of the sequence of all events from p in E ordered by $<$.

Observe that this construction of the set Σ guarantees the following property:

$$\mathit{Continuity}(\Sigma) \triangleq \forall\ \sigma \in \Sigma \setminus \varnothing, \exists\ e \in \sigma : (\sigma \setminus \{e\} \in \Sigma).$$

Moreover, the set of states Σ of the asynchronous system must satisfy the following axiom.

▶ **Definition 17** (Asynchronous distributed system axiom)**.** *The following predicate holds for the set of states Σ in an asynchronous distributed system.*

$$\begin{aligned}\mathit{Asynchrony}(\Sigma) \triangleq \forall\ \sigma \in \Sigma, (&\sigma \cup \{e\} \in \Sigma \wedge \sigma \cup \{e'\} \in \Sigma \wedge e.proc \neq e'.proc) \\ &\implies (\sigma \cup \{e, e'\} \in \Sigma).\end{aligned}$$

Asynchrony requires that if two states differ only in their last respective events, which are from different processes, their union is also a state. Observe that this must hold even if the two states are extracted from different histories. We point out that our impossibility proof is agnostic of the communication medium object M, as long as the medium satisfies asynchrony as defined above.

A.3 Valence

Our impossibility proof relies on the notion of valence, which was first introduced in [14].

▶ **Definition 18** (Valence function Val)**.** *Given a state $\sigma \in \Sigma$, the* valence *of σ is a set of values given by $\mathit{Val}(\sigma)$ as follows.*

- *If state $\sigma \in \mathit{Complete}(\Sigma)$ and σ is extracted from history $H = (E, <, O, P)$, then $\mathit{Val}(\sigma) = \{v \mid C.\mathsf{decide}()/v \in O\}$.*
- *$\mathit{Branching}(\Sigma)$: $\forall\ \sigma \in \Sigma \setminus \mathit{Complete}(\Sigma)$, $\mathit{Val}(\sigma) = \bigcup_{\sigma' \in \{\sigma \cup \{e\} \in \Sigma \mid e \notin \sigma\}} \mathit{Val}(\sigma')$.*

Intuitively, the valence of a complete state is the set of all values that were decided in the histories from which it was extracted, and, by *Branching*, the valence of an incomplete state is the union of the valences of all its one-event extensions. We say that a state $\sigma \in \Sigma$ is *univalent* iff we have $|\mathit{Val}(\sigma)| = 1$, and we say that it is *multivalent* iff we have $|\mathit{Val}(\sigma)| > 1$. Observe that it is not possible that $|\mathit{Val}(\sigma)| = 0$, due to the $\mathcal{L}$ predicate of the Consensus object.

▶ **Lemma 19.** $\mathit{NonEmptyValence}(\Sigma) \triangleq \forall\ \sigma \in \Sigma, |\mathit{Val}(\sigma)| \geq 1$.

Proof. From the $\mathcal{L}$ predicate (liveness) of the Consensus object specification, the valence $\mathit{Val}(\sigma)$ of a complete state $\sigma \in \mathit{Complete}(\Sigma)$ extracted from some history H contains at least one value. Let us consider now a state $\sigma \in \Sigma \setminus \mathit{Complete}(\Sigma)$ and assume by induction that all its one-event extension states $\sigma' \in \{\sigma \cup \{e\} \in \Sigma \mid e \notin \sigma\}$ have $|\mathit{Val}(\sigma')| \geq 1$. By construction of the set of states Σ, there is at least one such one-event extension state σ'. By $\mathit{Branching}(\Sigma)$, if holds that $\mathit{Val}(\sigma') \subseteq \mathit{Val}(\sigma)$. Then, $|\mathit{Val}(\sigma)| \geq 1$. ◀

Moreover, it holds that all complete states have a finite univalent sub-state.

▶ **Lemma 20.** $\mathit{Termination}(\Sigma) \triangleq \forall\, \sigma \in \mathit{Complete}(\Sigma), \exists\, \sigma' \in \Sigma, \sigma' \subseteq \sigma, |\sigma'| < +\infty : \mathit{Val}(\sigma) = \mathit{Val}(\sigma') \wedge |\,\mathit{Val}(\sigma')| = 1.$

Proof. Assume $\sigma \in \mathit{Complete}(\Sigma)$ is extracted from history $H = (E, <, O, P)$. First note that $|\,\mathit{Val}(\sigma)| = 1$ from Observation 16. Let $\mathit{Val}(\sigma) = \{v\}$ and $C.\mathsf{decide}_i/v \in O$. Then, in σ there is a decide event e_d from process p_i that returns v. Then, $\sigma' = \{e \in \sigma \mid (e.\mathit{proc} = p_i) \wedge (e{<}e_d)\} \cup \{e_d\}$ is finite and has $\mathit{Val}(\sigma') = \{v\}$. ◀

A.4 Resilient Non-trivial Consensus

Let us now define the properties we require for a non-trivial Consensus object that is resilient to any stopping process.

▶ **Definition 21** (Resilient Non-trivial Consensus axioms). *Given a system* Σ*, the following predicates describe* resilient non-trivial consensus*.*

$$\mathit{NonTriviality}(\Sigma) \triangleq \exists\, \sigma, \sigma' \in \Sigma : \mathit{Val}(\sigma) \neq \mathit{Val}(\sigma').$$

$$\mathit{Resilience}(\Sigma) \triangleq \forall\, \sigma \in \Sigma, |\,\mathit{Val}(\sigma)| > 1, \forall\, p \in P, \exists\, \sigma' {=} \sigma \cup \{e\} \in \Sigma : (e {\notin} \sigma) \wedge (p {\neq} e.\mathit{proc}).$$

NonTriviality states that there exist 2 states with different valences, implying that there are histories deciding different values. *Resilience* states that, for any process, any multivalent state can be extended by an event that is not from this process. This guarantees that even if one process stops taking steps (i.e., crashes), the system can still progress and eventually reach a decision.

A.5 Impossibility Theorem

▶ **Theorem 22.** *There cannot be a resilient non-trivial Consensus object in an asynchronous system.*

Proof. By way of contradiction, let us assume that we have a resilient non-trivial Consensus object C in an asynchronous distributed system, and let Σ be the set of states obtained from the set *Histories* of the system as described above. By construction, the properties *Continuity*, *Branching*, *NonEmptyValence*, and *Termination* hold for Σ. By assumption of an asynchronous distributed system, the axiom *Asynchrony* of Definition 17 holds. We also assume that the axioms *NonTriviality* and *Resilience* of Definition 21 hold for Σ, since C is resilient and non-trivial.

We first show that at least one multivalent state exists, i.e., $\exists\, \sigma \in \Sigma : |\,\mathit{Val}(\sigma)| > 1$. From *NonTriviality*, we have two states σ_u and $\sigma_{u'}$ with different valences. From $\mathit{NonEmptyValence}(\Sigma)$, σ_u and $\sigma_{u'}$ do not have empty valences, so they are either multivalent or univalent. If either σ_u or $\sigma_{u'}$ is multivalent, we are done, so let us assume that they are both univalent. By *Continuity* and *Branching*, we can iteratively remove one event in these states, until we reach a state σ_m (σ_m can be the empty set) that is contained in both σ_u and $\sigma_{u'}$ such that $\mathit{Val}(\sigma_u) \cup \mathit{Val}(\sigma_{u'}) \subseteq \mathit{Val}(\sigma_m)$. Hence, σ_m is multivalent.

From the fact that there is some multivalent state σ_m, we can inductively show that there exists what we call a *critical* state σ_c, i.e., a multivalent state for which all extensions are univalent:

$$\exists\, \sigma_c \in \Sigma : (|\,\mathit{Val}(\sigma_c)| > 1) \wedge (\forall\, \sigma' \in \Sigma, \sigma_c \subset \sigma' : |\,\mathit{Val}(\sigma')| = 1).$$

Observe that σ_m is incomplete (by $\mathit{Termination}(\Sigma)$) and hence has one-event extensions. If all extensions are univalent, σ_m satisfies the property of a critical state and we set $\sigma_c = \sigma_m$. Otherwise, σ_m has some one-event extension that is multivalent. Then, we make σ_m this new multivalent extension and repeat this procedure. Observe that this process must eventually end by finding a critical state, since otherwise, it means an infinite multivalent state exists, which contradicts $\mathit{Termination}(\Sigma)$.

Let us remark that, given that σ_c is a critical state, extending it by only one event results in a univalent state. By *Branching*, there exists (at least) two univalent states $\sigma_v, \sigma_{v'} \in \Sigma$, with different valences and obtained extending σ_c with one event: $\sigma_v = \sigma_c \cup \{e\}$ and $\sigma_{v'} = \sigma_c \cup \{e'\}$ such that $|\mathit{Val}(\sigma_v)| = |\mathit{Val}(\sigma_{v'})| = 1$ and $\mathit{Val}(\sigma_v) \neq \mathit{Val}(\sigma_{v'})$. Let us consider the two following cases.

- Case 1: $e.\mathit{proc} \neq e'.\mathit{proc}$. Given that the processes of the two events are distinct, from *Asynchrony*, we have $\sigma' = \sigma_v \cup \sigma_{v'} \in \Sigma$. Since $\sigma' = \sigma_v \cup \{e'\}$, from *Branching* it holds that $\mathit{Val}(\sigma') \subseteq \mathit{Val}(\sigma_v)$, and since $|\mathit{Val}(\sigma')| \geq 1$ (*NonEmptyValence*), then we have $\mathit{Val}(\sigma') = \mathit{Val}(\sigma_v)$. However, a similar argument yields that $\mathit{Val}(\sigma') = \mathit{Val}(\sigma_{v'})$, which contradicts $\mathit{Val}(\sigma_v) \neq \mathit{Val}(\sigma_{v'})$.
- Case 2: $e.\mathit{proc} = e'.\mathit{proc}$. By *Resilience*, we can extend σ_c with one event not from $e.\mathit{proc}$ to get a state $\sigma'' = \sigma_c \cup \{e''\} \in \Sigma$, such that $e''.\mathit{proc} \neq e.\mathit{proc}$. From the criticality of σ_c, σ'' is univalent. Then, either $\mathit{Val}(\sigma'') \neq \mathit{Val}(\sigma_v)$ or $\mathit{Val}(\sigma'') \neq \mathit{Val}(\sigma_{v'})$. Without loss of generality, assume that $\mathit{Val}(\sigma'') \neq \mathit{Val}(\sigma_v)$. Then, the contradiction follows from Case 1. ◀

A.6 Discussion on Impossibility of Asynchronous Resilient Consensus

Consensus is a fundamental abstraction of distributed computing with a simple premise: all participants of a distributed system must propose a value, and all participants must eventually agree on one of the values that have been proposed [27]. But just as fundamental is the impossibility theorem associated with consensus in the presence of asynchrony and faults. This result of impossibility, colloquially known as the *FLP* theorem (for the initials of its authors), was first shown in 1983 [14]. Later on, different approaches for proving similar theorems were proposed (e.g., [15, 18, 21]). Notably, the impossibility of asynchronous resilient consensus can be proved using algebraic topology and, more specifically, the asynchronous computability theorem [19]. Constructive proofs follow another interesting approach [15, 40]: they explicitly describe how a non-terminating execution of consensus can be constructed.[2] Like [37], our proof follows an axiomatic approach: the notion of asynchronous resilient consensus is defined as a system of axioms, and then it is proved that this system is inconsistent, i.e., that there is a contradiction.

Compared to previous impossibility proofs of asynchronous resilient consensus, we believe our proof to be one of the most general, partly due to the more natural notations offered by our specification formalism. In particular, unlike [14], which assumes that processes communicate through a message-passing network, our proof is agnostic on the communication medium (as long as such communication medium is asynchronous), hence it holds both for systems using send/receive or RW shared memory. In addition, our proof is more general than many previous proofs, in the sense that it shows an impossibility for a very weak version of the problem.

[2] In their paper [15], Gafni and Losa show the equivalence of 4 different models in terms of computability power: asynchronous 1-resilient atomic shared memory, asynchronous 1-resilient message passing, synchronous fail-to-send message passing, and synchronous fail-to-receive message passing. They then present a constructive impossibility proof in the synchronous fail-to-send message-passing model, and the impossibility in the other models directly follows. However, unlike ours, their proof still makes assumption of the communication medium.

For instance, our proof differs from [14, 19], which assumes that, at least in some specific cases, the value decided by the consensus instance is a value proposed by some process. In contrast, our proof does not make this assumption, as it does not need to relate the inputs (proposals) to the outputs (decisions) of the consensus execution.

B FIFO Consistency Addendum

Let us notice that the definition of the *FIFOConsistency* condition, as defined in Section 5.2, differs from the traditional definition of PRAM consistency we encounter in the literature, which is: "For each process p_i, we can construct a total order of op-exes containing op-exes of p_i, and the *update* op-exes of all processes." Here, *update* op-exes refer to the op-exes that change the object's internal state. For instance, for a register object with read and write operations, the updates would be the write op-exes. However, this initial definition is not completely accurate, because when we are constructing the total order of op-exes for a process p_i, if some update op-ex of another process p_j returns a value, we do not want to verify the validity of this value. Furthermore, adding a "$\forall\ p_i \in \mathit{Processes}$" quantifier at the start of the *FIFOConsistency* condition would make this condition structurally different from the other conditions of Definition 9, as it would create a potentially different $\rightarrow$ relation for every process of the system, instead of having a single global $\rightarrow$ relation like the other conditions of Definition 9.

Hence, our definition of *FIFOConsistency* relies on our new predicate $\mathit{FIFOOrder}(H, \rightarrow)$, which enforces a specific pattern on the $\rightarrow$ relation that characterizes the FIFO order of op-exes. As a reminder, here are the definitions of *FIFOOrder* and *FIFOConsistency* given in Sections 5.1 and 5.2, respectively.

$$\begin{aligned}\mathit{FIFOOrder}(H, \rightarrow) &\triangleq \forall\ o_i, o_i' \in O|p_i, o_j, o_j' \in O|p_j : \\ &\qquad (o_i \rightarrow o_i' \rightarrow o_j \rightarrow o_j' \land o_i \rightarrow o_j') \implies (o_i \rightarrow o_j \land o_i' \rightarrow o_j'). \\ \mathit{FIFOConsistency}(H, \rightarrow) &\triangleq \mathit{ProcessConsistency}(H, \rightarrow) \cup \{\mathit{FIFOOrder}(H, \rightarrow)\}.\end{aligned}$$

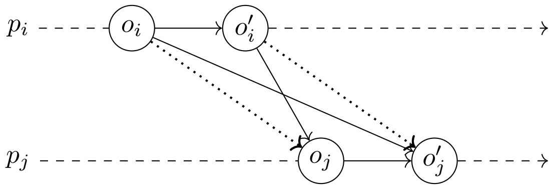

Figure 6 Illustration of *FIFOOrder*: if the pattern represented by the 4 hard arrows is present in the $\rightarrow$ op-ex relation, then the 2 dotted arrows must also be present in $\rightarrow$.

As said in Section 5.1, intuitively, *FIFOOrder* checks that a given op-ex sees all its predecessors on the same process, plus all the predecessors of the op-exes it sees on other processes. Furthermore, the "knowledge" of the op-exes of a given process is monotonically increasing with time: all the op-exes seen by a given op-ex must also be seen by its successors on the same process. Figure 6 illustrates the *FIFOOrder* predicate: we consider two processes, p_i and p_j (that can be the same process), that both have two op-exes (o_i, o_i', o_j and o_j'). If the first op-ex of p_j sees the second op-ex of p_i, and the second op-ex of p_j sees the first op-ex of p_i, then the first and second op-ex of p_j must respectively see the first and second op-ex of p_i.

In the end, we believe that, with this *FIFOOrder* predicate, the resulting definition of *FIFOConsistency* that we obtain is simpler than the original definition of PRAM consistency, while still achieving the same goal.

C Examples: Reliable Broadcast Object

This section gives an example of object specification by formally defining the celebrated *reliable broadcast* problem [7]. Let us remark that our formalism allows us to create object specifications and consistency hierarchies that are completely independent of the failure model: they hold both for omission (e.g., crashes) and Byzantine faults. Let us also observe that our framework's modularity enables us to define various consistent object specifications effortlessly by simply combining an object definition with a consistency condition. For example, by combining *Linearizability* with reliable broadcast (as specified in this section), we obtain another abstraction, linearizable broadcast [12].

In the following, the specifications consist of a list of operations with their correctness predicates, $\mathcal{V}$, $\mathcal{S}$, and $\mathcal{L}$. For concision, if we do not explicitly specify the $\mathcal{V}$, $\mathcal{S}$ or $\mathcal{L}$ predicates for some operation, then it means that implicitly, these predicates always evaluate to `true`. Furthermore, we use in the following logical formulas the ← symbol to denote an assignment of a value to a variable in the predicates. For convenience, we define below a shorthand for referring to the set of correct processes.

▶ **Definition 23** (Set of correct processes). *For a given history $H = (E, <, O, P)$, we define the function $P_C(H)$ that returns the set of correct processes of H, i.e., $P_C(H) \triangleq \{p \in P \mid p.type = \texttt{correct}\}$.*

Below, we define a reusable specification property for liveness checking that, if the process of an op-ex is correct, then this op-ex must terminate.

▶ **Definition 24** (Op-ex termination). *For an op-ex o, the* op-ex termination *liveness property is defined as $OpTermination(o) \triangleq o.proc.type = \texttt{correct} \implies o \not\equiv \mathsf{op}()/\bot$.*

C.1 Reliable Broadcast Specification

Reliable broadcast is a fundamental abstraction of distributed computing guaranteeing an *all-or-nothing* delivery of a message that a sender has broadcast to all processes of the system, and this despite the potential presence of faults (crashes or Byzantine) [7]. This section considers the multi-sender and multi-shot variant of reliable broadcast, where every process can broadcast multiple messages (different messages from the same process are differentiated by their message ID). A *reliable broadcast* object B provides the following operations:

- $B.\mathsf{r\_broadcast}(m, id)$: broadcasts message m with ID id,
- $B.\mathsf{r\_deliver}/(m, id, i)$ (notification): delivers message m with ID id from process p_i.

In the following, we consider a multi-shot reliable broadcast object B, a set of op-exes O, a relation $\rightarrow$ on O, an op-ex $o \in O$ and its context $(O_c, \rightarrow_c) = ctx(o, O, \rightarrow)$.

Operation r_broadcast. If $o \equiv B.\mathsf{r\_broadcast}_i(m, id)$, then we have the following.

$$\mathsf{r\_broadcast}.\mathcal{V}(o, (O_c, \rightarrow_c)) \triangleq \nexists\ \mathsf{r\_broadcast}_i(-, id) \in O_c.$$

$$\mathsf{r\_broadcast}.\mathcal{L}(o, H, \rightarrow) \triangleq OpTermination(o)$$
$$\wedge\ (\forall\ p_j \in P_C(H), \exists\ o' \equiv B.\mathsf{r\_deliver}_j/(m, id, i) \in O : o \rightarrow o').$$

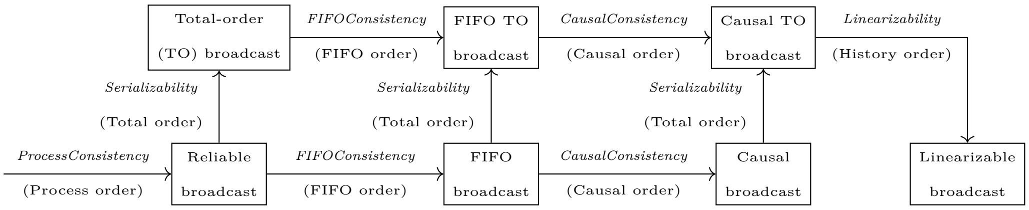
Total-order (TO) broadcast
FIFOConsistency (FIFO order)
FIFO TO broadcast
CausalConsistency (Causal order)
Causal TO broadcast
Linearizability (History order)
Serializability (Total order)
Serializability (Total order)
Serializability (Total order)
ProcessConsistency (Process order)
Reliable broadcast
FIFOConsistency (FIFO order)
FIFO broadcast
CausalConsistency (Causal order)
Causal broadcast
Linearizable broadcast

Figure 7 The reliable broadcast hierarchy of [16, 33] extended with linearizable broadcast [12].

The $\mathcal{V}$ predicate states that a process cannot broadcast more than once with a given ID. The $\mathcal{L}$ predicate states that a r_broadcast op-ex must terminate if a correct process made it, and must trigger matching r_deliver op-ex on every correct process.

Operation r_deliver. If $o \equiv B.\mathsf{r\_deliver}_i/(m, id, j)$, then we have the following.

$$\begin{aligned}
&\mathsf{r\_deliver}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq \\
&\quad C \leftarrow \{B.\mathsf{r\_broadcast}_j(m', id') \in O_c \mid \nexists\, B.\mathsf{r\_deliver}_i/(m', id', j) \in O_c\},\\
&\quad F \leftarrow \{(m', id') \mid \forall\, b,b' \in C, b \equiv B.\mathsf{r\_broadcast}_j(m', id') : b' \not\rightarrow_c b\} : (m, id) \in F.\\
&\quad \mathsf{r\_deliver}.\mathcal{L}(o, H, \rightarrow) \triangleq p_i \in P_C(H)\\
&\qquad\qquad\qquad\qquad \implies (\forall\, p_j \in P_C(H), \exists\, B.\mathsf{r\_deliver}_j/(m, id, k) \in O).
\end{aligned}$$

The $\mathcal{S}$ predicate states that a delivery must return one of the first broadcasts that have not been delivered with respect to $\rightarrow_c$. In $\mathcal{S}$, C denotes the set of candidate broadcast op-exes that have not been delivered, and F denotes the set of "first" broadcast values (message and ID) of op-exes of C that are not preceded (w.r.t. $\rightarrow_c$) by other op-exes in C. Notice that this does not necessarily mean that broadcasts must be delivered in FIFO order, as $\rightarrow$ does not necessarily follow FIFO order (to have this property, $\rightarrow$ would have to follow *FIFOConsistency*, see Section 5). This is the reverse of registers, where you can only read one of the last written values according to $\rightarrow$. The $\mathcal{L}$ predicate states that, if a correct process delivers a message, then all correct processes deliver this message.

C.2 The Reliable Broadcast Consistency Hierarchy

The modularity of our formalism allows us to plug any consistency condition (e.g., the ones defined in Section 5.2), or set of consistency conditions, that we want on any given object specification (e.g., reliable broadcast) to yield a *consistent object specification.* This section demonstrates this fact by applying different consistency conditions on the previously defined reliable broadcast specification.

A reliable broadcast object can provide different ordering guarantees depending on which consistency conditions it is instantiated with. Figure 7 illustrates the reliable broadcast hierarchy, and how reliable broadcasts of different strengths can be obtained by using *ProcessConsistency*, *FIFOConsistency*, *CausalConsistency*, *Serializability* or *Linearizability*.

As we can see, to obtain simple reliable broadcast, we must use the *ProcessConsistency* condition to guarantee that the op-exes of a given process are totally ordered. This assumption is necessary for the invocation validity (the precondition) of the r_broadcast operation, defined by the r_broadcast.$\mathcal{V}$ predicate. Indeed, this predicate states that a process cannot broadcast twice with the same ID; however, if op-exes of a process are not totally ordered, then there can be two r_broadcast op-exes from the same process and with the same ID that would not be in the context of one another, and thus the r_broadcast.$\mathcal{V}$ would not be violated when it should be. This is why a per-process total order of op-exes (imposed by *ProcessConsistency*) is often required for some object specifications (and in this case, for reliable broadcast).

D Examples: Shared Memory Object

Shared memory is a communication model where system processes communicate by reading and writing on an array of registers, identified by their address.

D.1 Shared Memory Object Specification

A shared memory M provides the following operations:

- $M.\mathsf{read}(a)/v$: returns one of the latest values v written in M at address a,
- $M.\mathsf{write}(v, a)$: writes value v in M at address a.

In the following, we consider a shared memory M, a set of op-exes O, a relation $\dashrightarrow$ on O, an op-ex $o \in O$ and its context $(O_c, \dashrightarrow_c) = ctx(o, O, \dashrightarrow)$.

Operation read. If $o \equiv M.\mathsf{read}_i(a)/v$, then we have the following.

$$\mathsf{read}.\mathcal{V}(o, (O_c, \dashrightarrow_c)) \triangleq \exists\ o' \equiv M.\mathsf{write}(-, a) \in O_c.$$

$$\mathsf{read}.\mathcal{S}(o, (O_c, \dashrightarrow_c)) \triangleq v \in \{v' \mid \exists\ o' \equiv M.\mathsf{write}(v', a) \in O_c, \nexists\ o'' \equiv M.\mathsf{write}(-, a) \in O'_c : o' \dashrightarrow_c o''\}.$$

$$\mathsf{read}.\mathcal{L}(o, H, \dashrightarrow) \triangleq OpTermination(o).$$

The $\mathcal{V}$ predicate states that a process cannot read an address never written into. The $\mathcal{S}$ predicate states that a read must return one of the last written values at that address with respect to $\dashrightarrow_c$. The $\mathcal{L}$ predicate states that a read op-ex must terminate if a correct process made it.

Operation write. If $o \equiv M.\mathsf{write}_i(v, a)$, then we have the following.

$$\mathsf{write}.\mathcal{L}(o, H, \dashrightarrow) \triangleq OpTermination(o).$$

The $\mathcal{L}$ predicate states that a write op-ex must terminate if a correct process made it.

D.2 Possible Variants

In the above, we have defined a version of shared memory constituted of multi-writer multi-reader registers (abridged MWMR), where everyone can read and write all the registers. But if we want to restrict the access of some registers to some processes, we can use the $\mathcal{V}$ precondition of the read and write operations. For example, if we want to design a single-writer multi-reader register (abridged SWMR), we can impose in the write.$\mathcal{V}$ predicate that only the invocations of write by a single process are considered valid. More generally, we can design asymmetric objects that provide different operations to different system processes using this technique.

D.3 The Shared Memory Hierarchy

As illustrated by Figure 8, by applying the *FIFOConsistency*, *CausalConsistency*, *SeqConsistency* or *Linearizability* consistency conditions on the specification of shared memory (Section D), different kinds of memory consistencies can be obtained.

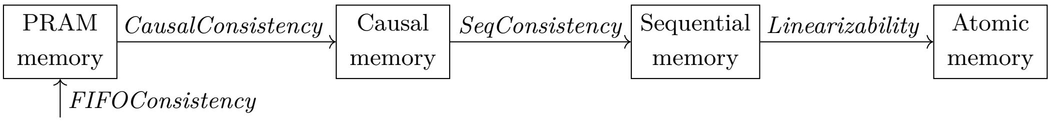

Figure 8 The shared memory hierarchy.

E Examples: Asynchronous Message-passing Object

Asynchronous message-passing is a communication model where system processes communicate by sending and receiving messages. This model is said to be asynchronous because messages can have arbitrary delays.

E.1 Asynchronous Message-passing Object Specification

A message-passing object M provides the following operations:

- $M.\mathsf{send}(m, i)$: sends message to receiver p_i,
- $M.\mathsf{receive}/(m, i)$ (notification): receives message m from process p_i.

In the following, we consider an asynchronous message-passing object M, a set of op-exes O, a relation $\rightarrow$ on O, an op-ex $o \in O$ and its context $(O_c, \rightarrow_c) = \mathit{ctx}(o, O, \rightarrow)$.

Operation send. If $o \equiv M.\mathsf{send}_i(m, j)$, then we have the following.

$$\begin{aligned}\mathsf{send}.\mathcal{L}(o, H, \rightarrow) \triangleq\ & \mathit{OpTermination}(o) \\ & \wedge (p_j \in P_C(H), \exists\ o' \equiv M.\mathsf{receive}_j/(m, i) \in O : o \rightarrow o').\end{aligned}$$

The $\mathcal{L}$ predicate states that a send op-ex must terminate if a correct process made it, and that the receiver, if it is correct, must eventually receive the message. For simplicity, we assume that a given message is only sent once (so we do not have to guarantee that it is received as often as it has been sent.

Operation receive. If $o \equiv M.\mathsf{receive}_i/(m, j)$, then we have the following.

$$\begin{aligned}\mathsf{receive}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq\ & (m, j) \in \{(m', k) \mid \\ & \exists\ M.\mathsf{send}_k(m', i) \in O_c, \nexists\ M.\mathsf{receive}_i/(m', k) \in O_c\}.\end{aligned}$$

The $\mathcal{S}$ predicate states that if a process receives a message, then this message has been sent before.

E.2 Possible Variants

We considered in this specification the asynchronous message-passing model, in which messages have arbitrary delays. But let us mention that this model's *synchronous* counterpart, where messages have a maximum delay known by all processes, can also be represented in our formalism as a concurrent object. The synchronous message-passing model can be represented as having rounds of communication, where all the messages sent in a round are received in the same round. Hence, we see that a synchronous message-passing object S can be represented as providing two operations $S.\mathsf{send}(m, i)$ and $S.\mathsf{end\_round}/M$, where end_round is a notification delivering to the process at hand p_i all the set M of messages sent to p_i during the round that ended. Again, let us notice that our formalism can specify the behavior of complex distributed systems without relying on higher-order logic such as temporal logic.

Furthermore, we assumed a message-passing specification over reliable channels; that is, there is no message corruption, deletion, duplication, etc., for instance, due to interference or disconnections. We classify this kind of network failure under the message adversary model [34]. However, we can easily imagine variants of this specification that consider a message adversary. In particular, for message deletions, the techniques introduced in [3] can help us to design a message-adversary-prone asynchronous message-passing object.

Finally, we considered an authenticated message-passing object because, when a message is received, the recipient knows the sender's identity (there is no identity spoofing), but we can easily design an unauthenticated variant that does not provide this information.

F Asynchrony of SWSR Atomic Registers and Point-to-point Message Passing

In this section, we prove that SWSR atomic registers and message passing, as communication media, satisfy the *Asynchrony* axiom of Definition 17. We make the following natural assumption about op-ex invocations.

▶ **Assumption 25** (Process consistent behavior). *A process decides whether to invoke an op-ex based only on its local view. Formally,*

$$(\sigma \in \Sigma \wedge \sigma \cup \{e\} \in \Sigma \wedge e \equiv \mathit{opex.inv}) \implies (\forall \sigma' \in \Sigma : \sigma|e.\mathit{proc} = \sigma'|e.\mathit{proc}, \sigma' \cup \{e\} \in \Sigma).$$

F.1 Asynchrony of an SWSR Atomic Register

We prove that an SWSR atomic register satisfies asynchrony as defined in Definition 17.

▶ **Theorem 26.** *A linearizable Single Writer Single Reader (SWSR) atomic register R satisfies the asynchronous distributed system axiom of Definition 17.*

Proof. Let us consider a system that contains a SWSR register R as specified in Section 4 with *Linearizability* consistency. Let us consider the set *Histories* of all the correct histories of this system projected to object R. Let Σ be the set of states extracted from *Histories*. Observe that the states in Σ only contain events from two processes: the writer p_w and reader p_r processes. Let us assume by way of contradiction that R does not satisfy asynchrony in Σ, then there is a $\sigma \in \Sigma$ and events e_w and e_r from writer and reader respectively such that

$$(\sigma \cup \{e_w\} \in \Sigma) \wedge (\sigma \cup \{e_r\} \in \Sigma) \wedge (\sigma \cup \{e_w, e_r\} \notin \Sigma). \tag{1}$$

This implies that σ can be extracted from a history H_w from which $\sigma_w = \sigma \cup \{e_w\}$ can also be extracted, but no such history H_w has event e_r. Similarly, σ can be extracted from a history H_r from which $\sigma_r = \sigma \cup \{e_r\}$ can also be extracted, but no such history H_r has event e_w. We have that e_w is an event from a write op-ex, and hence $e_w \equiv R.\mathsf{write}.\mathit{inv}$ or $e_w \equiv R.\mathsf{write}.\mathit{res}$. On its hand, e_r is an event from a read op-ex, and $e_r \equiv R.\mathsf{read}.\mathit{inv}$ or $e_r \equiv R.\mathsf{read}.\mathit{res}$. We have the following possibilities:

(1) First, consider a situation in which one of the events is an invocation event (i.e., $e_r \equiv R.\mathsf{read}.\mathit{inv}$ or $e_w \equiv R.\mathsf{write}.\mathit{inv}$). Let us assume, without loss of generality, that $e_w \equiv R.\mathsf{write}.\mathit{inv}$. We have that $\sigma|p_w = \sigma_r|p_w$. Then, from the process consistent behavior assumption (Assumption 25) applied to σ, σ_r, and e_w, we have that $\sigma_r \cup \{e_w\} = \sigma \cup \{e_w, e_r\}$ belongs to Σ, which contradicts the assumption. The case $e_r \equiv R.\mathsf{read}.\mathit{inv}$ is similar.

(2) Next, consider the situation where both events are responses, i.e., $e_r \equiv R.\mathsf{read}.res$ and $e_w \equiv R.\mathsf{write}.res$. Consider σ_r, which must contains the invocation e of the write op-ex $o = (e, e_w)$. Let us consider any history H from which σ_r can be extracted in which p_w is correct. Then by the *Legality* of H (and in particular the *Liveness* predicate of the write operation), op-ex o has to terminate in H. That is, p_w will have e_w as its next event in H. Then $\sigma_r \cup \{e_w\} = \sigma \cup \{e_w, e_r\} \in \Sigma$ which is a contradiction. ◀

F.2 Asynchrony of a Point-to-point Message-passing Object

We can also prove that a message-passing object as defined in Section E satisfies the *Asynchrony* axiom. We consider here a message passing object M used by two processes, a sender p_s and a receiver p_r, to communicate.

▶ **Theorem 27.** *A point-to-point message-passing object M satisfies the asynchronous distributed system axiom of Definition 17.*

The proof is similar to the proof of Theorem 26 replacing the writer with the sender and the reader with the receiver, and is omitted.