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

Timothé Albouy, Antonio Fernández Anta, Chryssis Georgiou, Mathieu Gestin, Nicolas Nicolaou, Junlang Wang

To cite this version:

Timothé Albouy, Antonio Fernández Anta, Chryssis Georgiou, Mathieu Gestin, Nicolas Nicolaou, et al.. AMECOS: A Modular Event-based Framework for Concurrent Object Specification. 2024. hal-04577664

HAL Id: hal-04577664
https://inria.hal.science/hal-04577664

Preprint submitted on 16 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

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

Timothé Albouy @
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Antonio Fernández Anta @
IMDEA Networks Institute, Spain

Chryssis Georgiou @
University of Cyprus, Cyprus

Mathieu Gestin @
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Nicolas Nicolaou @
Algolysis Ltd, Cyprus

Junlang Wang @
IMDEA Networks Institute, 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 interface 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 shared memory, asynchronous message-passing, and reliable broadcast, (ii) providing hierarchies of ordering semantics (namely, consistency hierarchy, memory hierarchy, and reliable broadcast hierarchy), and (iii) presenting novel axiomatic proofs of the impossibility of the well-known Consensus and wait-free Set Agreement problems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Concurrency, Object specification, Consistency conditions, Consensus and set agreement impossibility.

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. 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 within the system 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's state 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 [30] to stronger ones such as Linearizability [25]. 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* [31, 6]. Although easier and more intuitive for specifying how abstract data structures behave in a sequential way [41], as noted in [14], 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 [14]), 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 [14].

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, but instead offers a relaxed concurrent object specification mechanism (imposing partial order) that encapsulates concurrency intuitively, alleviating the specifier from complex specifications. Some noteworthy *features* of our specification framework are the following.

Component Identification and Interfacing: Our specification focuses on the *interface* between the various system components specified as concurrent objects. In particular, it considers objects to be black 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 [29] and Input/Output Automata [32]. 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 [27], specify different consistency semantics, such as PRAM [30] and Linearizability [25], *independently* of a specific object, and then combine them to obtain PRAM and atomic (*i.e.*, linearizable) 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, as we demonstrate, we can also specify communication mediums 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 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 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 having to call any operation. So, a notification can be seen as a "callback" that is made not by a process to an object but by the object to a process. This feature increases our framework's expressiveness compared to formalisms that are restricted to using only invocation and response events for operations [38]. **Contributions.** The following list summarizes the contributions of the paper.

- We first present our framework's basic components, key concepts and notation (see Section 2). We especially show that our framework can elegantly take into account a wide range of process failures, such as crash or Byzantine faults (see Section 2.4). Then, we demonstrate how concurrent objects can be specified using histories, preconditions and postconditions (see Section 3).
- 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 (see Section 4).
- To exemplify the usability of our formalism, we specify the reliable broadcast problem as a concurrent object (see Section 5). The modularity of our formalism is demonstrated by combining the consistency conditions with this broadcast specification and obtaining a broadcast hierarchy. An interested reader can find two more examples of specifications in Appendix C, where the asynchronous message-passing and shared memory communication media are specified as concurrent objects.
- We further demonstrate the usability (and simplicity) of our framework by specifying the well-known k-Set Agreement problem [15] as a concurrent object (see Section 6.1). Then, we provide a novel axiomatic proof of the impossibility of solving wait-free k-Set Agreement in an asynchronous system with $n > k$ processes[1]. To our knowledge, this is the first impossibility proof for wait-free set agreement that is agnostic to the communication mechanism (*e.g.*, message passing, shared memory) and is only based on three simple axioms: *Asynchrony*, *WaitFreeResilience*, and *NonTriviality* (see Section 6.2). The simplicity and generality of this proof (see Section 6.3) stem from the fact that our formalism and the axioms relieve us from providing implementation details of the object or system being specified, which allows us to prove intrinsic fundamental properties.

We also present a comparison with related work (Section 7) and a discussion of our findings (Section 8). Due to page limitations, some developments appear in appendices.

2 Framework: Components, Notation and Concepts

This section presents the basic components of the framework with its key notation and concepts.

[1] Similarly, in Appendix E, we define a *Consensus* object [19], and give a novel axiomatic proof showing that it cannot be implemented in an asynchronous system with crashes. We also discuss the benefits of this proof over previous proofs.

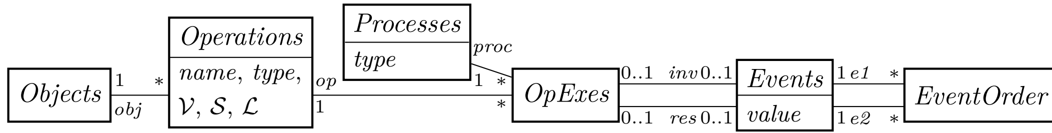

Figure 1 Diagram for the relations and attributes of the elements in the sets of our framework.

2.1 Components

We consider 6 (potentially infinite) sets of distributed system execution components. As shown in Figure 1, these sets are in relation to each other.

Processes: contains all *processes* p_i of the system (i corresponds to the process identity), where the $p_i.type \in \{\texttt{correct}, \texttt{faulty}\}$ attribute is the type of p_i (either correct or faulty, see Section 2.4).

Objects: contains all *objects* of the system (*e.g.*, a register, a stack, a broadcast object, *etc.*). An object is associated with a set of *operations* that processes can use to access it.

Operations: contains all *operations* op that can be performed 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 2.2), 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 3).

Events: contains all *events* e of the system, where the $e.value$ attribute is the value of the event (*i.e.*, the input or output value of an operation execution, see Section 2.2). When an operation is called, it produces an invocation event and/or a response event (see next item). Moreover, *Events* does not contain the $\perp$ value, which denotes the absence of an event.

OpExes: contains all *operation executions* (*op-ex* for short) o of an operation $o.op \in Operations$ by a process $o.proc \in Processes$, and where $o.inv, o.res \in Events$ respectively refer to the invocation (*i.e.*, start event) and the response (*i.e.*, end event) of o (at most one of $o.inv$ or $o.res$ can be equal to $\perp$, signifying the absence of event, see Section 2.2).

EventOrder: corresponds to the order between events of *Events*, represented as a set of event pairs ep where $ep.e1$ and $ep.e2$ are the first and second elements of the pair, respectively.

2.2 Notation

Op-exes. For simplicity of notation, we can represent an op-ex $o \in OpExes$ as a pair (i, r) where $i = o.inv$ and $r = o.res$. The elements of this pair, $o.inv$ and $o.res$, can be events of *Events*, but also the sentinel value $\perp$, which denotes an absence of event. An op-ex $o \in OpExes$ can have the following configurations (the $(\perp, \perp)$ pair is forbidden, like all pairs of identical values, see Constraint 3 below):

- $o = (i, r)$, where $i, r \in Events$: in this case, o is said to be a *complete* op-ex (an operation invocation that has a matching response),
- $o = (i, \perp)$, 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),
- $o = (\perp, r)$, 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 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 writing these elements in the notation: the form "$L.\mathsf{get}_i(j)/v$ op-ex on list L, process p_i, index j and return value v" could simply be written "$\mathsf{get}()$". 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 notification op-exes: 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 operation type: 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 $\perp$ and $\varnothing$ symbol in their return part. For instance, the forms "All pending op-exes" and "All complete op-exes with no return value" can be written "$\mathsf{op}()/\perp$" 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 $R.\mathsf{write}(v) \in O$. Nevertheless, this abuse of notation must be used with care, because two op-exes that have the same form are not necessarily the same op-ex (*e.g.*, a given process can call 2 times the same operation with the same parameters, which would result in one same form for 2 different op-exes).

Event order. For the sake of simplicity, we define the strict partial order $<_G$ over all elements of *Events* as follows[2]: $<_G \triangleq \{(ep.e1, ep.e2) \mid \forall\ ep \in EventOrder\}$. The order $<_G$ defines the temporal ordering of events, that is, $(e, e') \in <_G$, also expressed as $e <_G e'$, means that event e happens before event e'. The G of $<_G$ stands for "global", as it is applied on *Events*, the set of all events.

Histories. We represent a history of a distributed system as a tuple $H = (E, <, O)$, where E is a set of events, $<$ is a strict partial order on events, and O is a set of op-exes, such that:

- $O = H.opexes$: O is the set of op-exes of H,
- $E = \{i, r \mid (i, r) \in O\}$: E is the set of all events appearing in O,
- $< = \{(e, e') \in <_G \mid e, e' \in E\}$: $<$ is the subset of $<_G$ pertaining to events of E.

Given history $H = (E, <, O)$ 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 $<$.

2.3 Constraints

We now state additional constraints on the model that cannot be expressed directly on the diagram of Figure 1. We first provide predicates defining strict order relations as follows.

▶ **Definition 1** (Generic strict orders). *Given an arbitrary set S and an arbitrary relation $\prec$ over the elements of S, the following predicates define* strict partial order *and* strict total order.

$$PartialOrder(S, \prec) \triangleq (\forall\ e \in S : e \not\prec e) \wedge (\forall\ e, e', e'' \in S : e \prec e' \prec e'' \implies e \prec e'').$$
$$TotalOrder(S, \prec) \triangleq PartialOrder(S, \prec) \wedge (\forall\ e, e' \in S, e \neq e' : e \prec e' \vee e' \prec e).$$

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.

[2] Recall that any binary relation can be represented as a set of pairs.

▶ **Constraint 1** (Total order of events). *The $<_G$ relation is a strict total order over Events:*

$$EvTotalOrder() \triangleq TotalOrder(Events, <_G).$$

Constraint 1 enforces that the definition of strict total order is respected by $<_G$. Hence, we place ourselves in the classical (Newtonian) physics model, where events are totally ordered. This particular assumption is critical for defining the linearizability consistency condition (see Section 4.2), which assumes a total order of events.

▶ **Constraint 2** (Event validity). *Every event must be part of exactly one op-ex:*

$$EvValidity() \triangleq \forall\ e \in Events : |\{o = (i, r) \in OpExes \mid e = i \vee e = r\}| = 1.$$

▶ **Constraint 3** (Op-ex validity). *The invocation and the response events of an op-ex must be distinct, and if they are both not $\bot$, then the invocation must precede the response w.r.t. the event order:*

$$OpExValidity() \triangleq \forall\, (i, r) \in OpExes : i \neq r \wedge (i \neq \bot \neq r \implies i <_G r).$$

▶ **Constraint 4** (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:*

$$\begin{aligned} OpValidity() \triangleq \forall\ \mathsf{op} \in Operations, \forall\ o = (i, r) \in OpExes, o.op = \mathsf{op} :& \\ \mathsf{op}.type{=}\texttt{notif} \Rightarrow (i{=}\bot \wedge r \in Events) \wedge& \\ \mathsf{op}.type{=}\texttt{normal} \Rightarrow (i{\in}Events \wedge r \in Events \cup \{\bot\}).& \end{aligned}$$

This model does not forbid concurrent op-exes on the same process. This way, we can consider multi-core processes that can make concurrent operation calls. If a specifier wants to impose that op-exes on the same process behave "in isolation", that is, if they are concurrent, they must behave as if they happened sequentially, then he/she can apply on the history a consistency condition that ensures a per-process total order of op-exes (see Section 4.2).

2.4 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 1, for any $p \in Processes$, omission faults concern p only if $p.type$ = `faulty.omitting`, and Byzantine faults concern p only if $p.type$ = `faulty.byzantine`. Processes p of type $p.type$ = `correct` are subject to none of these faults.

Omissions correspond to operation invocations that do not terminate, for whatever reason, producing pending op-exes. We assume that such omitting processes, although they may produce omissions at any time, follow the protocol. A *crash fault* is a special case of omission fault on a process p, where there exists a point in time τ (the crash) such that, if p has omissions (*i.e.*, pending op-exes), then they concern all op-exes that were invoked before τ but that did not terminate by τ, and only them. Furthermore, after the crash at τ, p has no operation invocations or notifications.

On the contrary, Byzantine processes may arbitrarily deviate from the protocol (for instance, because of implementation errors or attacks). Strictly speaking, given that their implementation can be arbitrary, we cannot say that Byzantine processes make actual op-exes on the same operations and objects as non-Byzantine processes. However, Byzantine processes may simulate op-exes that can appear legitimate to other processes. Hence, 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 2** (Byzantine histories function). *Given history H=(E,<,O), the ByzHistories(H) function returns the set of all possible modified histories $H' = (E', <', O')$ s.t.:*

$$O' = \{o \in O \mid o.proc.type \neq \texttt{faulty.byzantine}\} \cup \{\text{any arbitrary set of pending op-exes by } p \mid \forall\, p \in Processes, p.type = \texttt{faulty.byzantine}\},$$

$$E' = \{i, r \in (i, r) \in O'\},$$

$$<' \subseteq E'^2 \text{ such that } < \subseteq <' \text{ and } TotalOrder(E', <').$$

Informally, given a base history $H = (E, <, O)$ and a modified history $H' = (E', <', O') \in$ *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 need to guarantee anything for Byzantine processes.

This paper is primarily interested in the specification level, so we are not concerned with implementation assumptions for limiting the number of faulty processes. Moreover, by adding more constraints to the model, new failure subtypes can be derived from these two basic failure types. As crashes are modeled as a subtype of omission failures, subtypes of Byzantine faults can also be modeled by restricting the types of histories that the *ByzHistories* function can return.

3 Framework: Object Specification

In this formalism, we define the specification of an object using a list of conditions that are applied to the input and output of this 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. 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, respectively. (For concrete examples of object specifications, see Sections 5.1 and 6.1 and Appendix C.)

Op-ex context. The context of an op-ex o is the set of all op-exes preceding o in the same object.

▶ **Definition 3** (Context of an op-ex). *The* context *of an op-ex $o \in O$ with respect to a binary relation $\rightarrow$ over O is defined as $ctx(o, O, \rightarrow) \triangleq (O_c, \rightarrow_c)$, where $O_c \triangleq \{o' \in O \mid o' \rightarrow o, o.obj = o'.obj\}$ and $\rightarrow_c \triangleq \{(o', o'') \in \rightarrow \mid o', o'' \in O_c \cup \{o\}\}$.*

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 $\rightarrow$ 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 4** (Invocation validity predicate $\mathcal{V}$). *Given an operation* $\mathsf{op} \in$ *Operations, its* invocation validity predicate $\mathsf{op}.\mathcal{V}(o, ctx(o, O, \rightarrow))$ *indicates whether an op-ex* $o = (i{\neq}\bot, r) \in$ *OpExes of* op *(i.e.,* $\mathsf{op} = o.op$*) respects the usage contract of the object given its context* $ctx(o, O, \rightarrow)$*.*

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 returned v by a $R.\mathsf{read}()/v$ is (one of) the last 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 5** (Safety predicate $\mathcal{S}$). *Given an operation* $\mathsf{op} \in$ *Operations, its* safety predicate $\mathsf{op}.\mathcal{S}(o, ctx(o, O, \rightarrow))$ *indicates whether* $r.value$ *is a valid return value for op-ex* $o = (i, r{\neq}\bot) \in$ *OpExes of* op *(i.e.,* $\mathsf{op} = o.op$*) in relation to its context* $ctx(o, O, \rightarrow)$*.*

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

▶ **Definition 6** (Liveness predicate $\mathcal{L}$). *Given an operation* $\mathsf{op} \in$ *Operations, its* liveness predicate $\mathsf{op}.\mathcal{L}(o, O, \rightarrow)$ *indicates whether an op-ex* $o = (i, r) \in$ *OpExes of* op *(i.e.,* $\mathsf{op} = o.op$*) respects the liveness specification of* op*.*

History legality. We now define the notion of history legality.

▶ **Definition 7** (Global validity, safety and liveness predicates). *Given a set of op-exes* O *and a relation* $\rightarrow$ *on* O*, the following predicates define* global validity*,* safety *and* liveness*.*

$$\begin{aligned} Validity(O, \rightarrow) &\triangleq \forall\ o = (i{\neq}\bot, r) \in O : o.op.\mathcal{V}(o, ctx(o, O, \rightarrow)). \\ Safety(O, \rightarrow) &\triangleq \forall\ o = (i, r{\neq}\bot) \in O : o.op.\mathcal{S}(o, ctx(o, O, \rightarrow)). \\ Liveness(O, \rightarrow) &\triangleq \forall\ o = (i, r) \in O : o.op.\mathcal{L}(o, O, \rightarrow). \end{aligned}$$

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

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

We can see that *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`, or in other words, $\rightarrow$ satisfies the invocation validity, safety and liveness of the objects of H.

History correctness. Here, we define the correctness of a history H w.r.t a set of clauses $\mathcal{C}$.

▶ **Definition 9** (Correctness predicate). *Given an arbitrary history $H = (E, <, O)$ and a set of clauses $\mathcal{C}$, the following predicate describes the correctness of H with respect to $\mathcal{C}$:*

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

Intuitively, a history H 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 verified. To create the set of all possible modified histories, we use the *ByzHistories* function introduced in Section 2.4. Depending on the failure model, Byzantine processes may or may not exist in the system, and in the latter case, the modified history H' is equivalent to the base history H. Recall that only pending op-exes are created for Byzantine processes, as we do not need to guarantee safety and liveness for them.

As an example, $\mathit{Correctness}(H, \mathit{Legality})$ denotes the application of *Legality* on a history H. In fact, *Legality* can be seen as the weakest set of constraints that can be applied to a history, upon which we can apply other constraints, which results in consistency conditions, as we show next.

4 Consistency

We first define reusable predicates describing certain constraints on the op-ex order $\rightarrow$ (Section 4.1) and then we define common consistency conditions (Section 4.2).

4.1 Op-ex order relations

▶ **Definition 10** (Op-ex order relations). *Given a set of op-exes O, a relation $\rightarrow$ on O and a partial order $<$ on events of O, the following predicates define* op-ex history order*,* op-ex process history order *and* op-ex FIFO order*:*

$$\mathit{OpHistoryOrder}(O, \rightarrow, <) \triangleq \forall\ o = (i, r), o' = (i', r') \in O, r \neq \bot :$$
$$((i' \neq \bot \land r < i') \lor (i' = \bot \land r < r')) \implies (o \rightarrow o' \land o' \not\rightarrow o).$$

$$\mathit{OpProcessOrder}(O, \rightarrow, <) \triangleq \forall\ p_i \in \mathit{Processes} :$$
$$\mathit{OpHistoryOrder}(O|p_i, \rightarrow, <) \land \mathit{TotalOrder}(O|p_i, \rightarrow).$$

$$\mathit{OpFIFOOrder}(O, \rightarrow) \triangleq \forall\ o_i, o'_i \in O|p_i, o_j, o'_j \in O|p_j :$$
$$(o_i \rightarrow o'_i \rightarrow o_j \rightarrow o'_j \land o_i \rightarrow o'_j) \implies (o_i \rightarrow o_j \land o'_i \rightarrow o'_j).$$

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 a set O 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 *OpHistoryOrder*, 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. Let us notice that we distinguish the case where the second op-ex o' is a notification from the case where it is not.

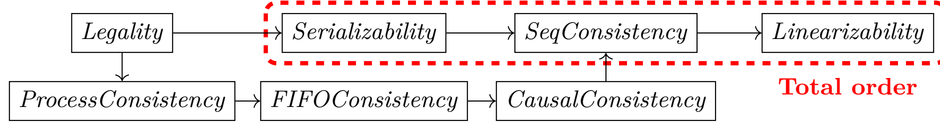

Figure 2 Hierarchy of the consistencies defined in this paper and their relative strengths [34].

- In *OpProcessOrder*, we check if $\rightarrow$ is a per-process op-ex total order respecting the event order.
- In *OpFIFOOrder*, 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 the *OpFIFOOrder* predicate can be found in Appendix B.

4.2 Classic consistency conditions

In this section, we define common consistency conditions from the literature. Like with *Legality*, we represent a consistency condition as a set of constraints on the $\rightarrow$ op-ex order.

▶ **Definition 11** (Classic consistency conditions). *Given a history $H = (E, <, O)$ and a relation $\rightarrow$ on O, the following sets of clauses respectively define* process consistency*,* FIFO consistency [30]*,* causal consistency [2, 35]*,* serializability [7]*,* sequential consistency [28] *and* linearizability [25]*.*

$$\begin{aligned}
\mathit{ProcessConsistency}(H, \rightarrow) &\triangleq \mathit{Legality}(H, \rightarrow) \cup \{\mathit{OpProcessOrder}(O, \rightarrow, <)\}.\\
\mathit{FIFOConsistency}(H, \rightarrow) &\triangleq \mathit{ProcessConsistency}(H, \rightarrow) \cup \{\mathit{OpFIFOOrder}(O, \rightarrow)\}.\\
\mathit{CausalConsistency}(H, \rightarrow) &\triangleq \mathit{FIFOConsistency}(H, \rightarrow) \cup \{\mathit{PartialOrder}(O, \rightarrow)\}.\\
\mathit{Serializability}(H, \rightarrow) &\triangleq \mathit{Legality}(H, \rightarrow) \cup \{\mathit{TotalOrder}(O, \rightarrow)\}.\\
\mathit{SeqConsistency}(H, \rightarrow) &\triangleq \mathit{Serializability}(H, \rightarrow) \cup \mathit{CausalConsistency}(H, \rightarrow).\\
\mathit{Linearizability}(H, \rightarrow) &\triangleq \mathit{SeqConsistency}(H, \rightarrow) \cup \{\mathit{OpHistoryOrder}(O, \rightarrow, <)\}.
\end{aligned}$$

An interested reader can also find in Appendix A definitions of set-linearizability [33] and interval-linearizability [13] in our formalism.

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 [7]) is the same as a single op-ex. Likewise, as discussed in detail in Appendix B, our definition of *FIFOConsistency* differs from the traditional definition of PRAM consistency.

We illustrate in Figure 2 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 interesting consistent object specifications (see Section 5.2).

5 Example of Consistent Object Specification: Reliable Broadcast

This section gives an example of object specification in our formalism by formally defining the celebrated *reliable broadcast* problem [10]. Let us mention that Appendix C provides

additional examples of specifications: *shared memory* and *asynchronous message-passing*, two of the most fundamental communication models of distributed computing, specified as simple objects.

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 [16].

In the following (and in the appendix), 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 $\leftarrow$ symbol to denote an affectation 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 12** (Set of correct processes). *We denote by P_c the subset of Processes containing only correct processes, such that $P_c \triangleq \{p \in Processes \mid p.type = \texttt{correct}\}$.*

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

5.1 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) [10]. 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 $\mathsf{r_broadcast}$**.** If $o \equiv B.\mathsf{r_broadcast}_i(m, id)$, then we have the following.

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

$$\mathsf{r_broadcast}.\mathcal{L}(o, O, \rightarrow) \triangleq OpTermination(o) \land (\forall\ p_j \in P_c, \exists\ o' \equiv B.\mathsf{r_deliver}_j/(m, id, i) \in O : o \rightarrow o').$$

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

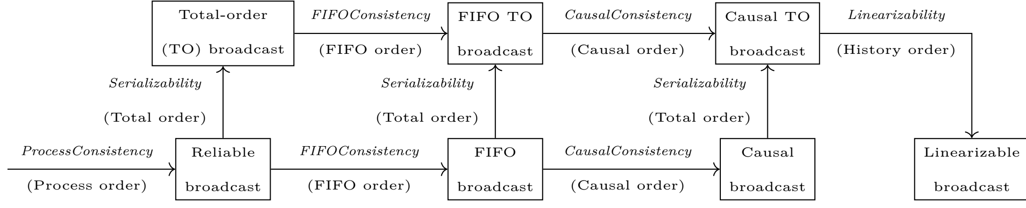

Figure 3 The reliable broadcast hierarchy of [21, 38] extended with linearizable broadcast [16].

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

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

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 4). 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.

5.2 The reliable broadcast consistency hierarchy

The modularity of our formalism allows us to plug any consistency condition (*e.g.*, the ones defined in Section 4.2), or set of consistency conditions, that we want on any given object specification (*e.g.*, reliable broadcast or the ones in Appendix C) 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 3 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).

6 Impossibility of Wait-free Set Agreement in Asynchronous Systems

In this section, we use our framework to define the Set Agreement object and to provide, to the best of our knowledge, the first axiomatic impossibility proof of solving wait-free k-Set Agreement in an asynchronous system[3].

6.1 Object Definition: Set Agreement and k-Set Agreement

Let us first define a Set-Agreement object SA that provides only one notification operation, $SA.\mathsf{decide}_i/v$, which returns a decided value $v \in V$ to process p_i, where V is the set of possible decisions. Observe that we consider a very weak version of Set Agreement which has no propose operation like traditional definitions. Indeed, proving the impossibility of this weaker version of Set Agreement (which applies also to the stronger version, where the decided value must have been previously proposed), makes our proof more general.

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

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

$$\mathsf{decide}.\mathcal{S}(o, (O_c, \rightarrow_c)) \triangleq (v \in V) \wedge (\forall\ SA.\mathsf{decide}_j/v' \in O_c : v = v').$$
$$\mathsf{decide}.\mathcal{L}(o, O, \rightarrow) \triangleq \exists\ SA.\mathsf{decide}_j/- \in O.$$

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.

The k-Set-Agreement object is the special case of Set Agreement in which at most k different values are decided in each history (1-Set Agreement is the Consensus problem). In order to have a k-Set-Agreement object kSA, we define a consistency predicate that we call *kSetTotalOrder* and a correctness condition that we call *kSerializability*:

$$\mathit{kSetTotalOrder}(O, \rightarrow) \triangleq \exists \text{ partition } P_1, P_2, \ldots, P_r \subseteq \mathit{Processes}, r \in [1, k] :$$
$$\forall j \in [1, r],\ \mathit{TotalOrder}(O|P_j, \rightarrow).$$
$$\mathit{kSerializability}(H, \rightarrow) \triangleq \mathit{Legality}(H, \rightarrow) \cup \{\mathit{kSetTotalOrder}(O, \rightarrow)\}.$$

Recall that a partition of the set *Processes* is a collection of disjoint subsets of the set *Processes*, whose union is the original set. Let *Histories* be a distributed system's set of histories containing a k-Set-Agreement object kSA. Then, we have the following assumption.

▶ **Assumption 1.** *For every history $H = (E, <, O) \in \mathit{Histories}$, let $H|kSA = (E|kSA, <|kSA, O|kSA)$ be the subhistory containing only the events of E applied to kSA. Then, it holds $\mathit{Correctness}(H|kSA, \mathit{kSerializability})$.*

From the fact that *kSerializability* guarantees a total order of the op-exes $O|kSA$ of each set of processes $P_j \subseteq \mathit{Processes}, j = 1, \ldots, r$ on object kSA, and the last clause of $\mathsf{decide}.\mathcal{S}()$, we have the following observation.

[3] Proof of the impossibility of solving Consensus in an asynchronous system with crashes is presented in Appendix E.

▶ **Observation 14.** *For every history $H = (E, <, O) \in Histories$, for each $j = 1, \dots, r$, all processes in P_j decide the same value v_j (if they decide). Hence, at most $r \leq k$ different values are decided in* decide *opexes.*

Observe that it is possible to have trivial implementations of a k-Set-Agreement object (in fact a Consensus object) in which all histories decide the same value $v \in V$ by simply hardcoding this in all the processes. Unfortunately, this object is not very useful. We will impose below a non-triviality condition that guarantees that there are histories in which the k-Set-Agreement object decides at least $k + 1$ different values. Additionally, the set of *Histories* must reflect the fact that the object tolerates crash failures and the system is asynchronous.

6.2 Axioms

We consider an asynchronous distributed system with $n = |Processes|$ processes in which up to $n - 1$ processes can crash. The system contains a k-Set-Agreement object, where $k < n$. Hence, we consider a wait-free k-Set-Agreement object denoted *wfSA*.

The (potentially infinite) set *Histories* represents the *system*. From these histories, we will construct a (potentially infinite) set S of possible states of the system. Each *state* $\mathcal{E} \in S$ is a (potentially infinite) set of events. Intuitively, a state $\mathcal{E}$ is the collection of local states of all system processes p_i, represented by the totally-ordered local events p_i has experienced.

To define the set S, we first assign an *index* to each event in a history $H = (E, <, O) \in$ *Histories*. The index assigned to event $e \in E$ is an attribute $e.idx$ that is the position of e in the sequence of events of its process $e.proc$. This sequence is obtained by ordering with $<$ the set $E|e.proc$. Observe that the sets of events of different histories may have common events. After adding the indices, common events 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 sender but in different orders.

A special subset of S is the set of *complete states*, defined as $Complete(S) \triangleq \{E \mid (E, <, O) \in Histories\}$. Consider now any history $H = (E, <, O) \in Histories$. We say that state $\mathcal{E} = E \in Complete(S) \subseteq S$ is a *state extracted from H*. Note that all the events in $\mathcal{E}$ of a process $p \in Processes$ are totally ordered by $<$. Then, we apply iteratively and exhaustively the following procedure to add more states to S: if $\mathcal{E} \in S$ is a state extracted from H, let $e \in \mathcal{E}$ be the last event (w.r.t. $<$) of a process p, then $\mathcal{E} \setminus \{e\}$ is also a state in S extracted from H. Observe that this process ends when the empty state $\mathcal{E} = \varnothing$ is reached (which is also in S). Intuitively, S contains a state $\mathcal{E}$ iff there is a history $H = (E, <, O) \in Histories$ such that the events in $\mathcal{E}$ from every process $p \in Processes$ are a prefix of the sequence of all events from p in E ordered by $<$.

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

$$Continuity(S) \triangleq \forall\ \mathcal{E} \in S \setminus \varnothing, \exists\ e \in \mathcal{E} : (\mathcal{E} \setminus \{e\} \in S).$$

Asynchrony Axiom. Moreover, any asynchronous distributed system S must satisfy the following axiom.

▶ **Definition 15** (Asynchronous distributed system axiom). *The following predicate holds for the set of states S in an* asynchronous distributed system.

$$Asynchrony(S) \triangleq \forall\ \mathcal{E} \in S, \forall E_1, E_2 \subseteq Events : (\mathcal{E} \cup E_1 \in S) \wedge (\mathcal{E} \cup E_2 \in S) \wedge (Procs(E_1) \cap Procs(E_2) = \varnothing) \Longrightarrow (\mathcal{E} \cup E_1 \cup E_2 \in S),$$

where, for each set of events $E \subseteq Events$, $Procs(E)$ is the set of processes with events in E.

Intuitively, *Asynchrony* implies that events from different processes can be added to existing states in any order to obtain new states. We remark that our impossibility proof below is agnostic of the communication medium, as long as the medium satisfies *Asynchrony* as defined above. In Appendix D, we provide simple examples showing that systems with atomic RW Shared Memory and Reliable Broadcast satisfy *Asynchrony* (thus, the impossibility proof holds), whereas Test&Set and Total Order Reliable Broadcast do not satisfy *Asynchrony*.

Wait-free-Resilience Axiom. The following predicates describe the *WaitFreeResilience* Axiom.

▶ **Definition 16** (Wait-free-Resilience Set Agreement axiom)**.** *Given a set of states S of a system with a wait-free k-Set-Agreement object wfSA, WaitFreeResilience is defined as*

$$\begin{aligned} \mathit{WaitFreeResilience}(S) \triangleq & \forall p_i \in \mathit{Processes} : \exists \mathcal{E}_i = \{e_1, ..., e_r\} \in S : \\ & (e_j.\mathit{proc} = p_i, \forall j \in [1, r]) \wedge \mathit{wfSA}.\mathsf{decide}_i/v_i.\mathit{res} \in \mathcal{E}_i \wedge v_i \in V. \end{aligned}$$

This axiom implies that every process can decide by itself even if the other processes do nothing.

Non-Triviality Axiom. Observe that for every process p_i there could be several pairs $(v_i, \mathcal{E}_i)$ that guarantee that $\mathit{WaitFreeResilience}(S)$ holds. Let us denote by F_i the set of values v_i in these pairs. I.e., F_i is the set of values that process p_i can decide without any interaction with the rest of the processes. Then, for each $v \in F_i$, let $\mathcal{E}(v)$ denote the corresponding $\mathcal{E}_i$ in the corresponding pair.

▶ **Definition 17** (Non-triviality Set Agreement axiom)**.** *Consider a set of states S of a system with a wait-free k-Set-Agreement object wfSA that satisfies $\mathit{WaitFreeResilience}(S)$. Let F_i denote the set of values process p_i can decide without interaction. Non triviality is defined as*

$$\mathit{NonTriviality}(S) \triangleq \forall i \in [1, n], \exists v_i \in F_i : |\{v_1, \ldots, v_n\}| \geq k + 1.$$

This *NonTriviality* assumption implies that without any interaction with other processes, the n processes p_i can decide at least $k+1$ different values v_i. This assumption is needed for the impossibility proof because otherwise k-Set Agreement can be trivially satisfied.

▶ **Observation 18.** *If the WaitFreeResilience assumption holds but the NonTriviality assumption does not hold, allowing each process p_i decide any value in F_i without interaction guarantees that at most k different values are decided.*

6.3 Impossibility Theorem

▶ **Theorem 19.** *For $k < n$, there is no asynchronous system with a wait-free k-Set-Agreement object wfSA that satisfies WaitFreeResilience and NonTriviality.*

Proof. Let us assume such a system exists and has a set of states S. By *NonTriviality*, there is a set $\{v_1, \ldots, v_n\}$ as in Definition 17. We show that there is a history in which each process p_i decides value v_i, for each $i \in [1, n]$. This shows that in this history at least $k+1$ different values are decided, which contradicts Observation 14.

Consider the set $\{v_1, \ldots, v_n\}$ and a process p_i. Since $v_i \in F_i$, by *WaitFreeResilience* there exist a state $\mathcal{E}(v_i) \in S$ such that, $\forall e \in \mathcal{E}(v_i), e.\mathit{proc} = p_i$, and $\mathit{wfSA}.\mathsf{decide}_i/v_i.\mathit{res} \in \mathcal{E}(v_i)$. Then, by the *Asynchrony* axiom (Definition 15), the state $\mathcal{E}' = \bigcup_{i \in [1,n]} \mathcal{E}(v_i)$ is also a state in S. From *WaitFreeResilience*, for each process p_i if holds that $\mathit{wfSA}.\mathsf{decide}_i/v_i.\mathit{res} \in \mathcal{E}'$. Hence let H be a history from which $\mathcal{E}'$ can be extracted. In H process p_i decides value v_i

and hence at least $k+1$ different values are decided. This contradicts Observation 14. So there is no asynchronous system with a wait-free k-Set Agreement object *wfSA* that satisfies *WaitFreeResilience* and *NonTriviality*. ◀

7 Related Work

The present work tackles two different (but not unrelated) problems: object semantics specification and consistency conditions. It also deals with the k-set agreement impossibility.

Object semantics specification. As we already discussed in Section 1, 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 [32], temporal logics (*e.g.*, LTL [36], CTL* [17], TLA [29]) 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 black 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 and simple 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.*, [28, 7, 30, 25, 2]). However, all of these consistency conditions have been introduced in their own notations and context (databases, RAM/cache consistency or distributed systems), which raised the need to have a unified formalism for expressing all types of consistency. Several formalisms have been proposed (*e.g.*, [12, 11, 43, 34]). We believe our formalism to have lighter notations and be easier to reason about while being more expressive than the ones presented before. As we demonstrate in Section 4, 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. This hierarchy has already appeared elsewhere in the literature [34]. However, we argue that our specification of it is more elegant and intuitive.

Possibly the work in [43] (derived from [12, 11]) is the closest to ours with respect to "specification style." However, the object specifications in [43] (and [12, 11]) 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 [43] 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 [43]). Additionally, compared to other endeavors such as [12, 11, 34], our framework also considers strong failure types such as Byzantine faults.

Impossibility of asynchronous k-Set Agreement. k-Set Agreement was first introduced by Chaudhuri [15], with the objective to relax the limitations of the Consensus problem (which is 1-Set Agreement). k-Set Agreement is a fundamental abstraction of distributed computing with a simple premise: the participants of a distributed system must propose values, and all participants must eventually agree on at most k of the values that have been proposed (one in the case of Consensus) [31, 15]. The fact that Consensus cannot be solved in an asynchronous system with crashes, colloquially known as the *FLP* theorem (from the initials of its authors), was first shown in 1983 [19]. In contrast, Chaudhuri showed that a $(k-1)$-resilient k-Set-Agreement algorithm exists, and left open the existence of a k-resilient algorithm. After that, several papers [24, 39, 9] have shown the impossibility of solving wait-free k-Set Agreement with asynchronous read/write shared memory, using topology to model distributed systems. (In [9], Borowsky *et al.* also proposed the BG-simulation technique for shared memory, which can be used to reduce asynchronous non-wait-free t-resilient k-set agreement to its wait-free counterpart.) However, these topological proofs rely heavily on topology results, use sophisticated technologies, and are hard to understand. Proofs without topology for the impossibility of asynchronous wait-free k-Set Agreement with read/write shared memory have also been presented in [4, 5]. In [8], Biely *et al.* present a generic theorem for proving the impossibility of asynchronous k-Set Agreement under a message-passing setup by reduction to the Consensus problem. All the above-mentioned proofs of impossibility of wait-free k-Set Agreement are constrained to certain communication form, either shared memory or message passing. With the help of the framework presented in this paper, we present a proof that is agnostic to the communication means and only relies on basic axioms.

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. Within our formalism, we provide a novel, and to our belief, elegant proof of the impossibility of wait-free k-Set Agreement 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. 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 flourished with object definitions, its usefulness will be apparent both to distributed computing researchers and practitioners seeking for a modular specification of complex distributed objects.

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.

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

4 Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of k-set agreement. *Theor. Comput. Sci.*, 512:41–48, 2013.

5 Hagit Attiya and Ami Paz. Counting-Based Impossibility Proofs for Renaming and Set Agreement. In Marcos K. Aguilera, editor, *Distributed Computing*, pages 356–370, Berlin, Heidelberg, 2012. Springer.

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

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

8 Martin Biely, Peter Robinson, and Ulrich Schmid. Easy impossibility proofs for k-set agreement in message passing systems. In *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, pages 227–228. ACM, 2011.

9 E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

10 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

11 Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, 2014.

12 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.

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

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

15 S. Chaudhuri. More *Choices* Allow More *Faults*: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, July 1993.

16 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.

17 E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Proc. 10th ACM Symposium on Principles of Programming Languages (POPL'83)*, pages 127–140. ACM Press, 1983.

18 Adin D. Falkoff, Kenneth E. Iverson, and Edward H. Sussenguth Jr. A formal description of SYSTEM/360. *IBM Syst. J.*, 3(2):198–262, 1964.

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

20 Eli Gafni and Giuliano Losa. Invited paper: Time is not a healer, but it sure makes hindsight 20:20. In *Proc. 25th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'23)*, volume 14310 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 2023.

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

22 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

23 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.

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

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

26 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.

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

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

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

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

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

32 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.

33 Gil Neiger. Set-linearizability. In *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, page 396. ACM, 1994.

34 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.

35 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.

36 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.

37 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations.* Springer, 2013.

38 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer, 2018.

39 Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, pages 101–110. ACM, 1993.

40 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.

41 Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

42 Gadi Taubenfeld. On the nonexistence of resilient consensus protocols. *Inf. Process. Lett.*, 37(5):285–289, 1991.

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

A Set- and Interval-Linearizability Definitions

In this section, we add set-linearizability [33] and interval-linearizability [13] to the repertoire of consistency conditions supported by the framework. Let us first define interval and set orders among op-exes.

▶ **Definition 20** (Op-ex set and interval order relations). *Given a set of op-exes O, a relation $\dashrightarrow$ on O and a partial order $<$ on events of O, the following predicates define* op-ex interval order *and* op-ex set order*:*

$$\begin{aligned} OpIntOrder(O, \dashrightarrow) &\triangleq (\forall\, o \in O : o \not\dashrightarrow o) \wedge (\forall\, o, o' \in O, o \neq o' : o \dashrightarrow o' \vee o' \dashrightarrow o) \\ &\wedge (\forall\, o, o', o'' \in O : o \dashrightarrow o'' \implies (o \dashrightarrow o' \vee o' \dashrightarrow o'')). \\ OpSetOrder(O, \dashrightarrow) &\triangleq OpIntOrder(O, \dashrightarrow) \\ &\wedge (\forall\, o, o', o'' \in O, o \neq o'' : o \dashrightarrow o' \dashrightarrow o'' \implies o \dashrightarrow o''). \end{aligned}$$

In *OpIntOrder*, 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 *OpSetOrder*, 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 *OpIntOrder*, *OpSetOrder* 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 *OpSetOrder* implies the last clause of *OpIntOrder*.

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

▶ **Definition 21** (Set- and interval-linearizability). *Given a history $H = (E, <, O)$, the following functions respectively define* set-linearizability [33] *and* interval-linearizability [13].

$$\begin{aligned} IntLinearizability(H, \dashrightarrow) &\triangleq Legality(H, \dashrightarrow) \cup \\ &\quad \{OpHistoryOrder(O, \dashrightarrow, <), OpIntOrder(O, \dashrightarrow)\}. \\ SetLinearizability(H, \dashrightarrow) &\triangleq IntLinearizability(H, \dashrightarrow) \cup \{OpSetOrder(O, \dashrightarrow)\}. \end{aligned}$$

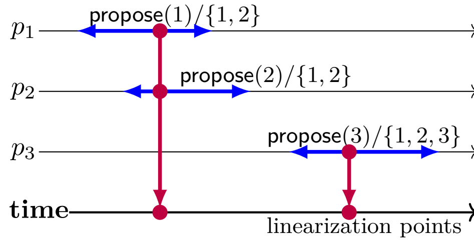

Figure 4 A set-linearizable execution of lattice agreement that is not linearizable.

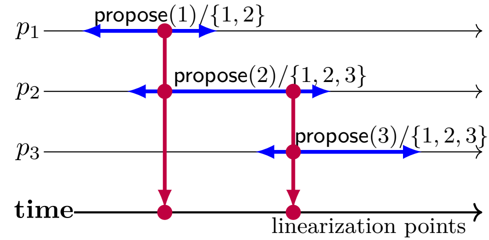

Figure 5 An interval-linearizable execution of lattice agreement that is not set-linearizable.

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 [14]. Lattice agreement is a simple object that provides a single operation propose$(v)/V$, where v is a value and V is a set of proposed values. Its only safety property is that the 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 $\{\text{propose}(1), \text{propose}(2)\}$ and $\{\text{propose}(3)\}$. The last clause of *OpSetOrder* enables the creation of said equivalence classes. Indeed, we have $\text{propose}(1) \rightarrow \text{propose}(2) \rightarrow \text{propose}(3)$ and $\text{propose}(1) \rightarrow \text{propose}(3)$. Besides, we also have $\text{propose}(2) \rightarrow \text{propose}(1) \rightarrow \text{propose}(3)$ and $\text{propose}(2) \rightarrow \text{propose}(3)$. We can see from this example 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, we see that the weakened transitivity clause of *OpSetOrder* 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, on the other hand, we see that the 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 $\{\text{propose}(1), \text{propose}(2)\}$ and $\{\text{propose}(2), \text{propose}(3)\}$. We can understand from this example that the forbidden pattern in interval-linearizability is one in which, 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, let us notice that, because of the connectedness property, the $o \rightarrow o' \rightarrow o''$ part of the left-hand side 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 *OpIntOrder*: $o \rightarrow o'' \implies (o \rightarrow o' \vee o' \rightarrow o'')$.

B FIFO Consistency Addendum

Let us notice that the definition of the *FIFOConsistency* condition, as defined in Section 4.2, 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 11, 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 11.

Hence, our definition of *FIFOConsistency* relies on our new predicate $\mathit{OpFIFOOrder}(O, \rightarrow)$, 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 *OpFIFOOrder* and *FIFOConsistency* given in Sections 4.1 and 4.2, respectively.

$$\mathit{OpFIFOOrder}(O, \rightarrow) \triangleq \forall\ o_i, o_i' \in O|p_i, o_j, o_j' \in O|p_j : (o_i \rightarrow o_i' \rightarrow o_j \rightarrow o_j' \wedge o_i \rightarrow o_j') \implies (o_i \rightarrow o_j \wedge o_i' \rightarrow o_j').$$

$$\mathit{FIFOConsistency}(H, \rightarrow) \triangleq \mathit{ProcessConsistency}(H, \rightarrow) \cup \{\mathit{OpFIFOOrder}(O, \rightarrow)\}.$$

As said in Section 4.1, intuitively, *OpFIFOOrder* checks that a given op-ex sees all its

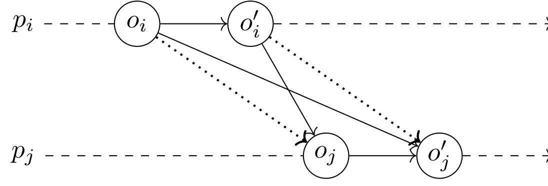

Figure 6 Illustration of *OpFIFOOrder*: if the pattern represented by the 4 hard arrows is present in the $\rightarrow$ op-ex relation, then the 2 dotted arrows must also be present in $\rightarrow$.

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 *OpFIFOOrder* 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 *OpFIFOOrder* 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 of Communication Media Specified as Concurrent Objects

In this section, we provide two additional examples of specifications in our formalism: *shared memory* and *asynchronous message-passing*, two of the most fundamental communication models of distributed computing, specified as simple objects.

C.1 Shared memory

Shared memory is a communication model where system processes communicate by reading and writing on an array of registers, identified by their address. 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 $\rightarrow$ on O, an op-ex $o \in O$ and its context $(O_c, \rightarrow_c) = ctx(o, O, \rightarrow)$.

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

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

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

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

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 $\rightarrow_c$. In $\mathcal{S}$, V denotes the set of values written in M that were not overwritten by new writes according to the $\rightarrow_c$ order. 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.

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

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

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 $\mathsf{write}.\mathcal{V}$ 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.

The shared memory hierarchy. As illustrated by Figure 7, by applying the *FIFOConsistency*, *CausalConsistency*, *SeqConsistency* or *Linearizability* consistency conditions on the specification of shared memory (Appendix C.1), different kinds of memory consistencies can be obtained.

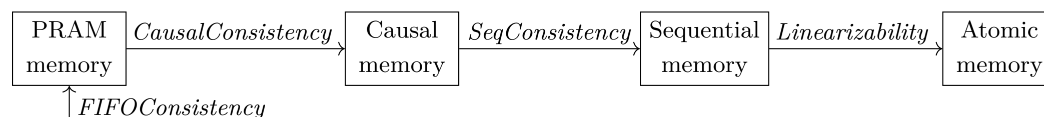

Figure 7 The shared memory hierarchy

C.2 Asynchronous message-passing

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. 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.

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

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.

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

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

Algorithm 1 Distributed system with 3 processes that write and read in a shared atomic register x.

Process 1: $M.\mathsf{write}(1, x)$
Process 2: $M.\mathsf{write}(2, x)$
Process 3: $v \leftarrow M.\mathsf{read}(x)$

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 logics 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 [40]. 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.

D Illustrations of Asynchronous Communication Media

In this section, we give some examples of distributed systems in which processes communicate using asynchronous communication objects, namely atomic shared memory and message-passing (reliable broadcast). We observe that these examples satisfy the *Asynchrony* property defined in Definition 15. Then, we present two systems with objects that do not satisfy *Asynchrony*, namely Test&Set and total-order reliable broadcast.

The first example is an atomic register x in a shared memory M. Algorithm 1 presents the algorithm executed by a distributed system formed by 3 processes, in which two of them write in x while the third one reads from x. Figure 8 presents the set of states S obtained from all the possible histories of the system of Algorithm 1 as described in Section 6.2. Observe in Figure 8 that states do not contain information about how the two write operations in Processes 1 and 2 are ordered in real time. The set is presented as a directed acyclic graph (DAG) with a single source node $\varnothing$, and a directed link labeled with event e from a state R to state $R \cup \{e\}$. This DAG has no cycles and *Continuity* guarantees that it is connected. By inspection, it is possible to observe in Figure 8 that the set S satisfies *Asynchrony* as defined in Definition 15.

The second example also has an atomic register x in a shared memory M. Algorithm 2 presents the algorithm executed by a distributed system formed by 2 processes, in which both of them write in x and then read from x. Figure 9 presents the set of states S obtained from all the possible histories of the system of Algorithm 2 as described in Section 6.2. Observe

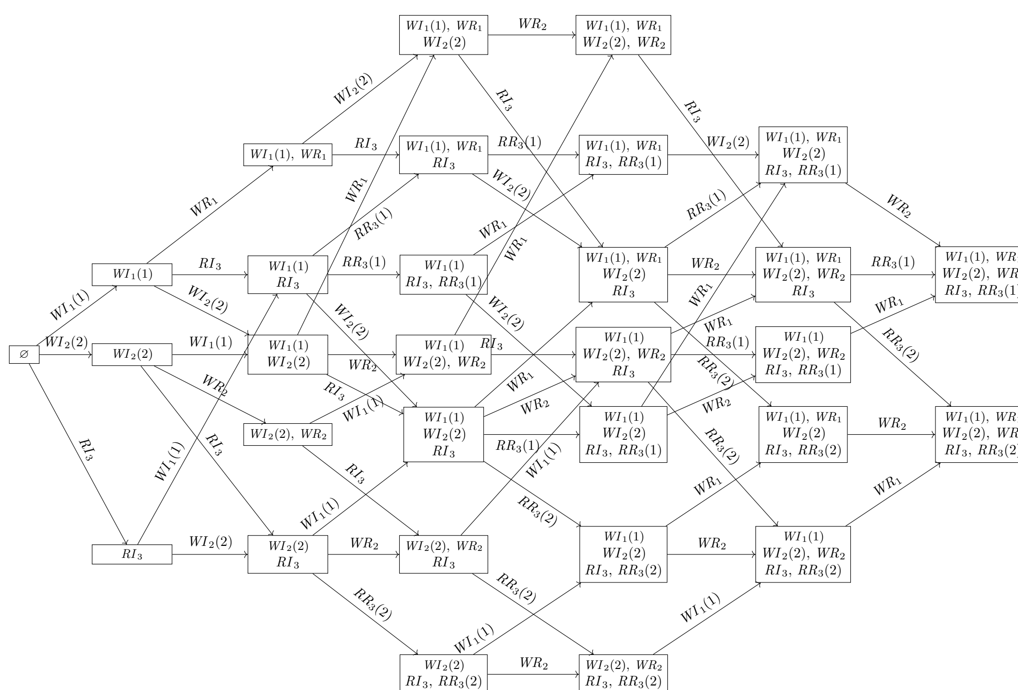

Figure 8 Set of states S as a *state graph* obtained from the histories of the system in Algorithm 1. Each node (box) is a state. A directed arrow labeled with an event e between two states indicates a transition triggered by e (see the *Continuity*(S) property). Events are abbreviated for clarity from their formal definition in Appendix C.1: $WI_i(v)$, WR_i, RI_i and $RR_i(v)$ are short for $M.\mathsf{write}_i(v,x).inv$, $M.\mathsf{write}_i(v,x).res$, $M.\mathsf{read}_i(x).inv$ and $M.\mathsf{read}_i(x)/v.res$, respectively. The event indices are omitted for clarity.

Algorithm 2 Distributed system with 2 processes that write and read in a shared atomic register x.

Process 1: $M.\mathsf{write}(1,x)$; $v_1 \leftarrow M.\mathsf{read}(x)$
Process 2: $M.\mathsf{write}(2,x)$; $v_2 \leftarrow M.\mathsf{read}(x)$

again that in this figure states do not contain information about how the two write operations in Processes 1 and 2 are ordered in real time. For this reason in the last layer from each state there are two arrows labelled with the two possible different values read. By inspection, it can be observed that the set S of Figure 9 satisfies *Asynchrony* as defined in Definition 15.

For comparison, we present in Algorithm 3 the algorithm run by a system with two processes that use a Test&Set object T [18, 37]. This object has an operation $\mathsf{test\&set}$ that returns a value 0 the "first" time it is invoked and a value 1 in any invocation after that. Figure 10 shows the set S as a DAG obtained from all the histories of this system. It can be observed that S does not satisfy *Asynchrony*, since it is violated for state $R = \{T\&SI_1, T\&SI_2\}$ and events $e = T\&SR_1(0)$ and $e' = T\&SR_2(0)$ (using the notation of the figure). Observe that the object T could be used to solve consensus among the 2 processes, since the consensus number of Test&Set is 2 [22, 37].

The third example we present is a system with two processes that use a reliable broadcast object B to send respective messages m_1 and m_2, as presented in Algorithm 4. Figure 11 presents a subset of the set S obtained from the histories of this system. For clarity, we have

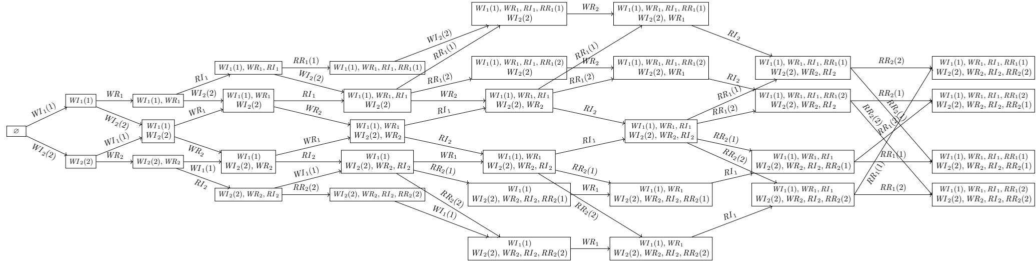

Figure 9 Set of states S as a *state graph* obtained from the histories of the system in Algorithm 2. Events are abbreviated for clarity from their formal definition in Appendix C.1: $WI_i(v)$, WR_i, RI_i and $RR_i(v)$ are short for $M.\mathsf{write}_i(v,x).inv$, $M.\mathsf{write}_i(v,x).res$, $M.\mathsf{read}_i(x).inv$ and $M.\mathsf{read}_i(x)/v.res$, respectively. The event indices are omitted for clarity.

Algorithm 3 Distributed system with 2 processes that use a Test&Set object T to coordinate.

Process 1: $v_1 \leftarrow T.\mathsf{test\&set}()$
Process 2: $v_2 \leftarrow T.\mathsf{test\&set}()$

omitted the response events of the r_broadcast op-exes, the states in which a process receives a message before invoking its local r_broadcast op-ex, and the event indices. Observe that the four sink states only differ in the order in which they receive the messages (captured by the indices of the r_deliver events). It can also be observed that *Asynchrony* is satisfied.

Finally, we present a system with a similar message-passing algorithm, presented in Algorithm 5, but that in this case uses a total order reliable broadcast object B instead of simply r_broadcast. Figure 12 shows a subset of the set S, in which some states and events are omitted as before for clarity. However, it can be seen in the resulting DAG that now only 2 sink states exist since both processes receive the messages in the same order. This prevents the *Asynchrony* axiom from being satisfied.

E Impossibility of Resilient Consensus in Asynchronous Systems

In this section, we use our framework to define a consensus object and to provide a simple proof of the FLP impossibility of having reliable deterministic consensus in an asynchronous system with process failures [19]. This proof is inspired in [42]. After the proof a discussion on its relevance is included.

E.1 The Consensus Object

A Consensus object is a special case of a Set Agreement object defined in Section 6.1 in which exactly one value can be decided in each history. (If $V = \{0, 1\}$ we have binary consensus.) We achieve this by combining the Set Agreement object specification with the *Serializability* consistency.

▶ **Assumption 2.** *Let C be a Consensus object. Then, C is a Set Agreement object and for every $H = (E, <, O) \in Histories$, it holds $Correctness(H|C, Serializability)$.*

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 $\mathsf{decide}.\mathcal{S}()$, all decide op-exes in O return the same value v.

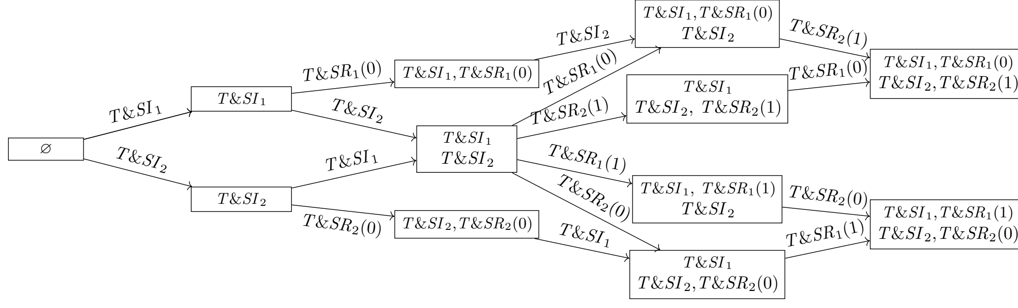

Figure 10 Set of states S as a *state graph* obtained from the histories of the system in Algorithm 3. Events are abbreviated for clarity: $T\&SI_i$ and $T\&SR_i(v)$ are short for $T.\mathsf{test\&set}_i().inv$ and $T.\mathsf{test\&set}_i()/v.res$, respectively. The event indices are omitted for clarity.

Algorithm 4 Distributed system with 2 processes that use a reliable broadcast object B to send 2 messages m_1 and m_2.

Process 1: $B.\mathsf{r_broadcast}(m_1, 1)$
Process 2: $B.\mathsf{r_broadcast}(m_2, 2)$

▶ **Observation 22.** $\forall\ H = (E, <, O) \in \mathit{Histories} : (C.\mathsf{decide}_i/v_i, C.\mathsf{decide}_j()/v_j \in O) \implies (v_i = v_j)$.

In the rest of this section we assume a distributed system that contains a Consensus object C. The system is described by its set of histories *Histories*. From this set we obtained the set of states S as described in Section 6.2. We also assume that the system is asynchronous, as defined by the *Asynchrony* axiom (Definition 15).

E.2 Valence

Our impossibility proof relies on the notion of valence, which was first introduced in [19].

▶ **Definition 23** (Valence function Val). *Given a state $\mathcal{E}$, the* valence *of $\mathcal{E}$ is a set of values given by $\mathit{Val}(\mathcal{E})$ as follows.*

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

Intuitively, the valence of a complete state is the set of all values that were decided, 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 $\mathcal{E} \in S$ is *univalent* iff we have $|\mathit{Val}(\mathcal{E})| = 1$, and we say that it is *multivalent* iff we have $|\mathit{Val}(\mathcal{E})| > 1$. Observe that it is not possible that $|\mathit{Val}(\mathcal{E})| = 0$.

▶ **Lemma 24.** $\mathit{NonEmptyValence}(S) \triangleq \forall\ \mathcal{E} \in S, |\mathit{Val}(\mathcal{E})| \geq 1$.

Proof. From the $\mathcal{L}$ predicate (liveness) of C, the valence $\mathit{Val}(\mathcal{E})$ of a complete state $\mathcal{E} \in \mathit{Complete}(S)$ extracted from some history $H = (E, <, O)$ contains at least one value. By construction of the set S and $\mathit{Branching}(S)$, any other state $\mathcal{E}' \in S \setminus \mathit{Complete}(S)$ extracted from H has $\mathit{Val}(\mathcal{E}) \subseteq \mathit{Val}(\mathcal{E}')$. ◀

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

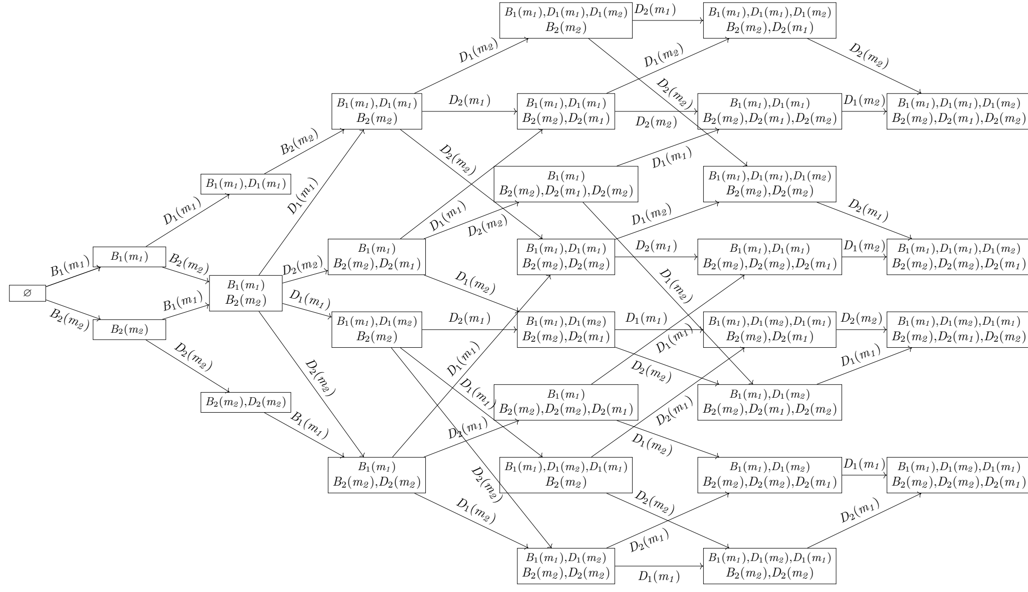

Figure 11 Set of states S as a *state graph* obtained from the histories of the system in Algorithm 4. Events are abbreviated for clarity: $B_i(m_i)$ and $D_i(m_j)$ are short for $B.\mathsf{r_broadcast}_i(m_i, i).inv$ and $B.\mathsf{r_deliver}_i/(M_j, j, j).res$, respectively. The states in which a process receives a message before invoking r_broadcast, the $B.\mathsf{r_broadcast}_i(m_i, i).res$ events, and the event indices are omitted for clarity.

Algorithm 5 Distributed system with 2 processes that use a total-order reliable broadcast object B to send 2 messages m_1 and m_2.

Process 1: $B.\mathsf{TO_broadcast}(m_1, 1)$
Process 2: $B.\mathsf{TO_broadcast}(m_2, 2)$

▶ **Lemma 25.** $Termination(S) \triangleq \forall\, \mathcal{E} \in Complete(S), \exists\, \mathcal{E}' \in S, \mathcal{E}' \subseteq \mathcal{E}, |\mathcal{E}'| < +\infty : |Val(\mathcal{E})| = |Val(\mathcal{E}')| = 1.$

Proof. Assume $\mathcal{E} \in Complete(S)$ is extracted from history $H = (E, <, O)$. First note that $|Val(\mathcal{E})| = 1$ from Observation 22. Let $Val(\mathcal{E}) = \{v\}$ and $C.\mathsf{decide}_i/v \in O$. Then, in $\mathcal{E}$ there is a decide event e_d from process p_i that returns v. Then, $\mathcal{E}' = \{e \in \mathcal{E} \mid (e.proc = p_i) \wedge (e{<}e_d)\} \cup \{e_d\}$ is finite and has $Val(\mathcal{E}') = \{v\}$. ◀

E.3 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 26** (Resilient Non-trivial Consensus axioms)**.** *Given a system S, the following predicates describe* resilient non-trivial consensus*.*

$$NonTriviality(S) \triangleq \exists\, \mathcal{E}, \mathcal{E}' \in S : Val(\mathcal{E}) \neq Val(\mathcal{E}').$$

$$Resilience(S) \triangleq \forall\, \mathcal{E} \in S, |Val(\mathcal{E})| > 1, \forall\, p \in Processes,$$
$$\exists\, \mathcal{E}' = \mathcal{E} \cup \{e\} \in S : (e \notin \mathcal{E}) \wedge (p \neq e.proc).$$

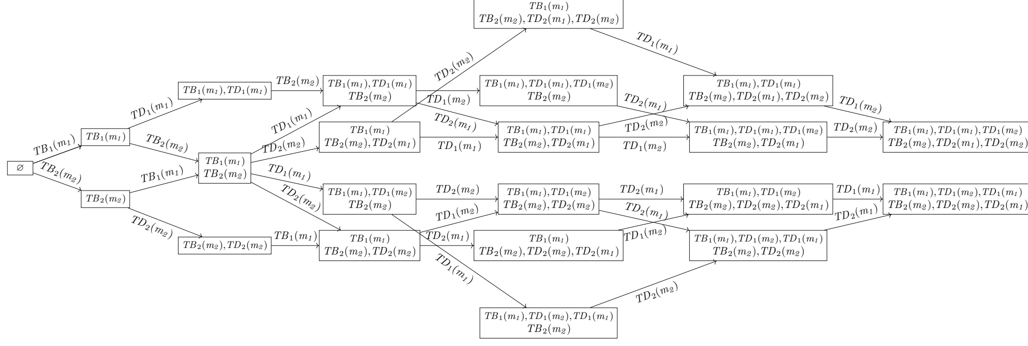

Figure 12 Set of states S as a *state graph* obtained from the histories of the system in Algorithm 5. Events are abbreviated for clarity: $TB_i(m_i)$ and $TD_i(m_j)$ are short for $B.\mathsf{TO_broadcast}_i(m_i, i).inv$ and $B.\mathsf{TO_deliver}_i/(M_j, j, j).res$, respectively. The states in which a process receives a message before invoking TO_broadcast, the $B.\mathsf{TO_broadcast}_i(m_i, i).res$ events, and the event indices are omitted for clarity.

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.

E.4 Impossibility theorem

▶ **Theorem 27.** *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 S be the set of states obtained from the set *Histories* of the system as described in Section 6.2. By construction, the properties *Continuity*, *Branching*, *NonEmptyValence*, and *Termination* hold for S. By assumption of an asynchronous distributed system, the axiom *Asynchrony* of Definition 15 holds. We also assume that the axioms *NonTriviality* and *Resilience* of Definition 26 hold for S, since C is resilient and non-trivial.

We first show that at least one multivalent state exists, *i.e.*, $\exists\, \mathcal{E} \in S : |Val(\mathcal{E})| > 1$. From *NonTriviality*, we have two states $\mathcal{E}_u$ and $\mathcal{E}_{u'}$ with different valences. From $NonEmptyValence(S)$, $\mathcal{E}_u$ and $\mathcal{E}_{u'}$ do not have empty valences, so they are either multivalent or univalent. If either $\mathcal{E}_u$ or $\mathcal{E}_{u'}$ 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 $\mathcal{E}_m$ ($\mathcal{E}_m$ can be the empty set) that is contained in both $\mathcal{E}_u$ and $\mathcal{E}_{u'}$ such that $Val(\mathcal{E}_u) \cup Val(\mathcal{E}_{u'}) \subseteq Val(\mathcal{E}_m)$. Hence, $\mathcal{E}_m$ is multivalent.

From the fact that there is some multivalent state $\mathcal{E}_m$, we can inductively show that there exists what we call a *critical* state $\mathcal{E}_c$, *i.e.*, a multivalent state for which all extensions are univalent: $\exists\, \mathcal{E}_c \in S, |Val(\mathcal{E}_c)| > 1, \forall\, \mathcal{E}' \in S, \mathcal{E}_c \subset \mathcal{E}' : |Val(\mathcal{E}')| = 1$. Observe that $\mathcal{E}_m$ is incomplete (by $Termination(S)$) and hence has extensions. If all extensions are univalent, $\mathcal{E}_m$ satisfies the property of a critical state and we set $\mathcal{E}_c = \mathcal{E}_m$. Otherwise, $\mathcal{E}_m$ has some extension that is multivalent. Then, we make $\mathcal{E}_m$ 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 $\mathit{Termination}(S)$.

Let us remark that, given that $\mathcal{E}_c$ is a critical state, extending it by only one event results in a univalent state. By *Branching*, there exists (at least) two univalent states $\mathcal{E}_v, \mathcal{E}_{v'} \in S$, with different valences and obtained extending $\mathcal{E}_c$ with one single event: $\mathcal{E}_v = \mathcal{E}_c \cup \{e\}$ and $\mathcal{E}_{v'} = \mathcal{E}_c \cup \{e'\}$ such that $|\mathit{Val}(\mathcal{E}_v)| = |\mathit{Val}(\mathcal{E}_{v'})| = 1$ and $\mathit{Val}(\mathcal{E}_v) \neq \mathit{Val}(\mathcal{E}_{v'})$. 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 $\mathcal{E}' = \mathcal{E}_v \cup \mathcal{E}_{v'} \in S$. Since $\mathcal{E}' = \mathcal{E}_v \cup \{e'\}$, from *Branching* it holds that $\mathit{Val}(\mathcal{E}') \subseteq \mathit{Val}(\mathcal{E}_v)$, and since $|\mathit{Val}(\mathcal{E}')| \geq 1$ (*NonEmptyValence*), then we have $\mathit{Val}(\mathcal{E}') = \mathit{Val}(\mathcal{E}_v)$. However, a similar argument yields that $\mathit{Val}(\mathcal{E}') = \mathit{Val}(\mathcal{E}_{v'})$, which contradicts $\mathit{Val}(\mathcal{E}_v) \neq \mathit{Val}(\mathcal{E}_{v'})$.
- Case 2: $e.\mathit{proc} = e'.\mathit{proc}$. By *Resilience*, we can extend $\mathcal{E}_c$ with one event not from $e.\mathit{proc}$ to get a state $\mathcal{E}'' = \mathcal{E}_c \cup \{e''\} \in S$, such that $e''.\mathit{proc} \neq e.\mathit{proc}$. From the criticality of $\mathcal{E}_c$, $\mathcal{E}''$ is univalent. Then, either $\mathit{Val}(\mathcal{E}'') \neq \mathit{Val}(\mathcal{E}_v)$ or $\mathit{Val}(\mathcal{E}'') \neq \mathit{Val}(\mathcal{E}_{v'})$. Without loss of generality, assume that $\mathit{Val}(\mathcal{E}'') \neq \mathit{Val}(\mathcal{E}_v)$. Then, the contradiction follows from Case 1. ◀

E.5 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 [31]. 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 [19]. Later on, different approaches for proving similar theorems were proposed (*e.g.*, [26, 23, 20]). Notably, the impossibility of asynchronous resilient consensus can be proved using algebraic topology and, more specifically, the asynchronous computability theorem [24]. Like [42], 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 simplest, partly due to the more natural notations offered by our specification formalism. In particular, unlike [19], 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. For instance, our proof differs from [19, 24], which assume 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. This way, our proof also shows the impossibility of other abstractions, such as shared random coins, which allow the system processes to agree on some value chosen at random.