Arbitration-Free Consistency Is Available (and Vice Versa)

HAGIT ATTIYA, Technion - Israel Institute of Technology, Israel
CONSTANTIN ENEA, LIX - Ecole Polytechnique - CNRS - Institut Polytechnique de Paris, France
ENRIQUE ROMÁN-CALVO, University of Freiburg, Germany

The fundamental tension between *availability* and *consistency* shapes the design of distributed storage systems. Classical results capture extreme points of this trade-off: the CAP theorem shows that strong models like linearizability preclude availability under partitions, while weak models like causal consistency remain implementable without coordination. These theorems apply to simple read-write interfaces, leaving open a precise explanation of the combinations of object semantics and consistency models that admit available implementations.

This paper develops a general semantic framework in which storage specifications combine operation semantics and consistency models. The framework encompasses a broad range of objects (key-value stores, counters, sets, CRDTs, and SQL databases) and consistency models (from causal consistency and sequential consistency to snapshot isolation and bounded staleness).

Within this framework, we prove the *Arbitration-Free Consistency* (AFC) theorem, showing that an object specification within a consistency model admits an available implementation if and only if it is *arbitration-free*, that is, it does not require a total arbitration order to resolve visibility or read dependencies.

The AFC theorem unifies and generalizes previous results, revealing arbitration-freedom as the fundamental property that delineates coordination-free consistency from inherently synchronized behavior.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; • **Computer systems organization** → *Availability*.

Additional Key Words and Phrases: Distributed Systems, Availability, CAP Theorem

ACM Reference Format:
Hagit Attiya, Constantin Enea, and Enrique Román-Calvo. 2026. Arbitration-Free Consistency Is Available (and Vice Versa). *Proc. ACM Program. Lang.* 10, POPL, Article 41 (January 2026), 63 pages. https://doi.org/10.1145/3776683

1 Introduction

Distributed storage systems enable reliable access to objects by replicating them across a wide-area network. Replication is essential for tolerating faults in the system (e.g., machines that crash, network partitions) and for decreasing latency. In such systems, it is crucial to maintain a trade-off between availability (ensuring prompt access to data) and preserving consistency, even in the presence of communication delays. The *CAP theorem* [13, 18] shows that a key-value store cannot provide strong *Consistency* (atomicity) while maintaining *Availability* and tolerating network *Partitions* at the same time. PACELC [1, 19] refines CAP by adding the case of a connected network where strong consistency cannot be achieved with low latency.

Authors' Contact Information: Hagit Attiya, Technion - Israel Institute of Technology, Haifa, Israel, hagit@cs.technion.ac.il; Constantin Enea, LIX - Ecole Polytechnique - CNRS - Institut Polytechnique de Paris, Palaiseau, France, cenea@lix.polytechnique.fr; Enrique Román-Calvo, University of Freiburg, Freiburg im Breisgau, Germany, calvo@informatik.uni-freiburg.de.

ACM 2475-1421/2026/1-ART41
https://doi.org/10.1145/3776683

Many modern storage systems sacrifice strong consistency for availability (or low latency) and ensure weaker notions of consistency. There is plethora of weak consistency models [14] (or *isolation levels* [3] in the context of transactions) that correspond to different trade-offs with respect to availability. Other modern storage systems relax the semantics of the objects they support, e.g., multi-value registers, where a get arbitrarily returns a previously stored value.

Given that the guarantees of a storage system are captured through the subtle combination of its consistency model and its object semantics, a natural question to consider is:

> *What class of consistency models support available implementations of which objects in the presence of network partitions?*

Previous results provided only partial answers to this question. The aforementioned CAP theorem only shows a negative result that an *Atomic (Linearizable) Key-Value Store* is not included in this class. Attiya et al. [7] identify a consistency model, called Observable Causal Consistency (OCC), that is not included in this class; but only for particular objects, Multi-Value Registers. We remark that Causal Consistency, which is strictly weaker than both of them, is in the class.

The goal of this paper is to give a precise answer for the question raised above. To do so, we rely on a very expressive framework for defining consistency models and object semantics that builds on previous work [14]. Using this framework, we give a tight characterization of models and objects that can be expressed within this framework and that support available implementations. Before explaining our characterization, we outline our framework.

A framework for defining storage specifications. A storage system is composed of a collection of objects that can be read or modified using a set of operations (the API of the storage). Specifications are expressed in terms of an abstract model of storage executions, which is defined as a set of binary relations among events—each event corresponding to an invocation of an operation on an object. These relations capture typical control-flow dependencies–such as invocations occurring at the same replica–data-flow dependencies–where certain updates affect the result of a query–and a total order used as a "tie-breaker" to fix an order between concurrent invocations. The latter is called *arbitration order* and it has an important role in our main result.

In a distributed storage system, implementations typically rely on communication protocols to share the effects of invocations among all replicas. They also use specific algorithms to merge the effects received from other replicas into the local replica's state. As a result, each invocation can be viewed as executing within a specific *context*–that is, the set of prior operations, including those received from remote replicas.

A storage specification defines the expected behavior of the system. It consists of two parts:

- a *consistency model*, restricting the possible contexts in which each invocation may execute.
- an *operation specification*, describing the allowable effects of an invocation, given its context.

A consistency model consists of a set of *visibility* formulas saying when an invocation belongs to the context of another invocation. This "being in the context of" binary relation is defined via combinations of the binary relations mentioned above (by standard composition, union, and transitive closure). For instance, a visibility formula may state that all prior invocations at the same replica should be included in the context. An *operation specification* consists of a set of functions that characterize the read and write behavior of an invocation, in particular, the value written by writes. Note that this value is not always fixed since we allow operations that read and write at the same time, e.g., Compare-and-Swap which writes a given object only if the old value equals some other value given as input. We also allow SQL transactions whose effects are even more complex.

We show that our framework covers many possible storage specifications, including Last-Writer-Wins and Multi-Value Key-Value stores, Key-Value stores with Compare-and-Swap operations, Key-Value stores with counters, as well as transactional and non-transactional SQL stores, and many

```
PUT(x,1);   || PUT(y,2);
a = GET(y); || b = GET(x);
```

(a) Sequential Consistency and PUT, GET operations.

```
PUT(x, 1);  || PUT(y, 1);
...         || ...
PUT(x, K);  || PUT(y, K);
a = GET(x); || b = GET(y);
```

(b) Bounded Staleness and PUT, GET operations.

```
FAA(x, 1); || FAA(x, 2);
```

(c) Sequential Consistency and FAA operations.

```
FAA(x, 1); || FAA(y, 2);
FAA(y, 3); || FAA(x, 4);
```

(d) Prefix Consistency and FAA operations.

Fig. 1. Different litmus programs with two concurrent sessions showing the absence of available implementations for selected pairs of consistency models and operation specifications.

possible consistency models including Return-Value Consistency, Causal Consistency, Sequential Consistency, and transactional isolation levels like Snapshot Isolation and Serializability.

*The arbitration-free consistency (*AFC*) theorem.* Our main result states roughly, that a storage system has an available implementation if and only if the visibility formulas that define its consistency model *exclude any meaningful use of the total arbitration order.* Such a consistency model is called *arbitration-free.* As in previous works, we consider an implementation to be *available* if operations can be answered immediately on every replica (without waiting for messages from other replicas).

The proof of the AFC theorem is quite challenging, one reason being the very expressive and abstract specification framework that we consider. Proving that there exist available implementations for arbitration-free consistency models is the easier part since arbitration-freeness implies that the model is weaker than causal consistency, and the latter supports available implementations [9, 10, 25, 26]. The opposite direction is much more difficult and is described in two stages.

We first consider a basic case, in which operations read and/or write a single value from/to a single object. This yields a reasonably simple proof, while still covering consistency models such as Return-Value Consistency, Causal Consistency, Prefix Consistency and Sequential Consistency, and objects such as a key-value store, with ordinary put / get operations or extended with Fetch-and-Add and Compare-and-Swap operations.

Then, we consider a general class of objects where operations can read and/or write multiple objects at the same time, and reads may compute their return value from multiple updates. In this very generic context, we need to introduce some number of restrictions (assumptions) which are however satisfied by all practical cases that we are aware of (see Section 7). This is to exclude pathological cases that arise from starting with a very abstract formal model.

To summarize, we provide the first characterization of distributed storage formal specifications that support available implementations which takes into account both consistency constraints and the semantics of the implemented objects. At a high level, the key insight behind our result is that in an asynchronous system, where replicas coordinate only through the exchange of messages, they can establish at most a causal order between operations. The arbitration order, in contrast, is total: it compares operations that are concurrent and therefore incomparable under causality. Determining such a total order would require additional synchronization between replicas, coordination that cannot be achieved in an always-available manner.

2 Motivating Examples

We illustrate the broad applicability of the AFC theorem through various storage specifications, each reflecting different trade-offs between consistency and operation semantics. We argue about the diversity of reasoning required and motivate the need for a unified framework.

As a starting point, we consider a standard key-value store with PUT and GET operations; PUT(x, v) writes the value v to object (key) x, and GET(x) reads the latest[1] value of object x. As consistency

[1]We assume a standard semantics based on the Last-Writer-Wins conflict resolution policy.

model, we consider the standard *Sequential Consistency* (SC) whose formalization uses arbitration to postulate an order in which different operations interleave. By the AFC theorem, the latter implies that there exists no available implementation that ensures SC. Intuitively, the proof is based on a *litmus* program like in Figure 1a. This program contains two concurrent sessions, each executed at a different replica. Also, x and y are initially 0. An SC available implementation should allow an execution in which, intuitively, the two replicas operate without exchanging any messages, resulting in both final get operations returning 0. However, this outcome violates sequential consistency, as it cannot be produced by any interleaving of the operations—leading to a contradiction.

We remark that this argument proves a version of the CAP theorem that is stronger than the one proved in [18]. The latter proof relies on the *real-time ordering* requirement that is embedded in linearizability — a consistency model stronger than sequential consistency (cf. [21]).

Such a proof can be generalized to the case where PUT / GET operations are replaced for instance, by ADD / CONTAINS operations on a set, i.e., PUT(x, v) and GET(x) in Figure 1a are replaced by ADD(x) and CONTAINS(x) (and similarly for operations on y). As in the previous case, an SC available implementation should allow an execution without exchange of messages, resulting in both final CONTAINS operations returning false (the set does not contain the element), which is an SC violation.

On the other hand, if we consider a weaker consistency model, a straightforward variation of the program in Figure 1a can not be used to prove non-existence of available implementations. For instance, consider *Bounded Staleness* [28] a weakening of SC, which requires that each get operation observes all preceding put operations (on the same object), except possibly the most recent $K - 1$, for some fixed value of K. The put operations are still required to execute following some fixed arbitration order as in SC (see Section 7.1 for a precise definition). This weakening for $K = 2$ admits an execution of t he program in Figure 1a where both final get operations return 0 (the get operations may miss the only put in the program). Therefore, this program cannot be used to show non-existence of available implementations. Instead, one can use the program given in Figure 1b, which contains K put operations in each session. One can follow now the same strategy as above and show that an execution without exchange of messages makes both get operations return 0, and this violates bounded staleness.

If we weaken consistency even further and consider *Causal Consistency* (CC) [29], then the AFC theorem will imply existence of available implementations (which is known [9, 10, 25, 26]).

Now, if we change the set of operations and consider a storage system with only Fetch-and-Add operations (FAA(x, v) returns the old value of x and adds v, atomically), then a proof for non-existence of SC available implementations can be done using the program in Figure 1c with only one FAA in each session. An execution without exchange of messages will imply that both FAA return the initial value of x, and this is a violation of SC.

If we weaken consistency to *Prefix Consistency* (PC) [16], then the previous program is not suitable. An execution where both FAA in Figure 1c return the initial value of x satisfies PC (see section 4.1 for a formal definition). Instead, we need a litmus program like in Figure 1d which contains two FAAs per session. Here, an execution where all FAAs return an initial value does not satisfy PC. This program can also be used to show the non-existence of available implementations of *Parallel Snapshot Isolation* (PSI) [30] or *Conflict-preserving Causal Consistency* (CCC), a consistency model defined using the axioms Conflict and Causal from [11]. As a side remark, note that CCC is equivalent to CC for the key-value store with PUT and GET operations presented at the beginning, and therefore, there exists an available implementation for CCC in that case.

While these cases follow a broadly similar proof strategy, each demands distinct proof artifacts (such as litmus programs) and tailored reasoning. The AFC theorem unifies these diverse arguments within a common theoretical framework, grounded in a formalization of a wide class of storage specifications encompassing all the examples above.

3 Abstracting Storage Executions

We present an abstract model of distributed storage executions that includes the essential components needed to define storage specifications. A *distributed storage* (or simply storage) replicates the state of a set of objects over two or more nodes called *replicas*. We use Objs to denote the infinite set of objects, ranged over x, y, z, and Reps to denote the set of replica identifiers, ranged over r, r_1, r_2. Objects are accessed using a set of *operations* which may write or return values in a set Vals.

An *abstract execution* records operation invocations along with a set of relations that represent control-flow dependencies (two invocations executing on the same replica), and the internal behavior of the storage. The internal behavior includes, broadly, the computation of object states and the return values of invocations, as well as the communication between replicas. The first concerns local computation within each replica, while the second pertains to communication protocols or underlying network assumptions. To distinguish these two aspects, we first introduce the concept of a *history*, which records only the data-flow dependencies relevant to characterizing the local computation. An abstract execution is then defined as an extension of a history, enriched with additional relations that abstract inter-replica communication.

3.1 Histories

The invocation of an operation on some replica is represented using an *event* $e = (\mathsf{id}, \mathsf{r}, \mathsf{op}, \mathsf{wval}, \mathsf{m})$ where id is an event identifier, r is a replica identifier, op is an operation name, wval is a (partial) mapping that associates an object x with a value v that this event writes to x, and m is additional metadata of the invocation. We use $\mathtt{id}(e)$ $\mathtt{rep}(e)$, $\mathtt{op}(e)$, $\mathtt{wval}(e)$, and $\mathtt{md}(e)$ to denote the event identifier, replica identifier, operation, written value mapping and metadata of an event e, respectively. We assume that every event e accesses (reads or writes) a fixed finite set of objects denoted as $\mathsf{obj}(e)$. The set of events is denoted by Events. We assume that Events includes a distinguished type of *initial events* that affect every object, representing the initial state of the storage.

Example 3.1. *As a running example, we consider a Key-value Store with four types of operations:* $\mathtt{PUT}(x, v)$ *that writes* v *to object (key)* x*,* $\mathtt{GET}(x)$ *that reads object* x*,* $\mathtt{FAA}(x, v)$ *that reads the value* v' *of object* x *and writes* $v' + v$*, and* $\mathtt{CAS}(x, v, v')$*, that reads* x *and writes* v' *iff the value read is* v*. We use* faacas *to refer to this storage (from the Fetch-and-Add and Compare-and-Swap operations).*

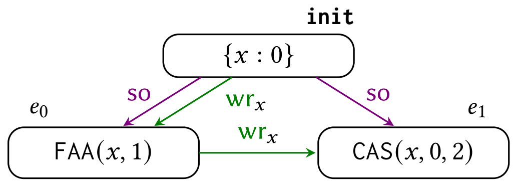
A *history* contains a finite set of events E ordered by a (partial) *session order* so that relates events on the same replica, and a *write-read* relation wr (also known as read-from) representing data-flow dependencies between events that update and respectively, read a same object. Histories contain an initial event, **init**, that precedes every other event in E w.r.t so. We consider a write-read relation $\mathsf{wr}_x \subseteq \mathcal{P}(E) \times E$ for every object $x \in \mathsf{Objs}$. The inverse of wr_x is defined as usual and denoted by wr_x^{-1}. We use $\mathsf{wr} : \mathsf{Objs} \to \mathcal{P}(E) \times E$ to denote the mapping associating each object x with wr_x.

For simplicity, we often abuse the notation and extend wr_x and wr to pairs of events: we say that $(w, r) \in \mathsf{wr}_x$ if $w \in \mathsf{wr}_x^{-1}(r)$, and we say that $(w, r) \in \mathsf{wr}$ if there exists an object x s.t. $(w, r) \in \mathsf{wr}_x$.

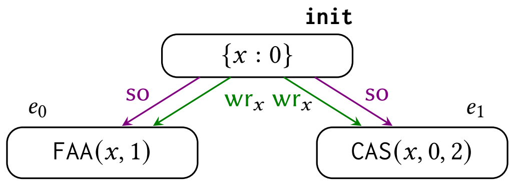
Definition 3.2. *A* history $(E, \mathsf{so}, \mathsf{wr})$ *is a finite set of events* E *along with a strict partial* session order so*, and a* write-read *relation* $\mathsf{wr}_x \subseteq \mathcal{P}(E) \times E$ *for every* $x \in \mathsf{Objs}$ *such that*

- *E contains a single initial event* **init***, which precedes every other event in E w.r.t.* so*,*
- $\forall e, e' \in E \setminus \{\mathbf{init}\}$*,* so *orders e and e' iff* $\mathtt{rep}(e) = \mathtt{rep}(e')$*,*
- *the inverse of* wr_x *is a total function for every* $x \in \mathsf{Objs}$*, and*
- $\mathsf{so} \cup \mathsf{wr}$ *is acyclic (here we use the extension of* wr *to pairs of events).*

Example 3.3. *Figure 2 shows two examples of histories of the storage* faacas *presented in Example 3.1. For readability, we omit replica identifiers from events. The* wr *dependencies can be used to explain the*



(a) e_0 reads 0, writes 1; e_1 reads 1 and does not write.



(b) e_0 reads 0 and writes 1; e_1 reads 0 and writes 2.

Fig. 2. Two examples of histories for faacas. Arrows represent so and wr relations. The initial event **init** defines the initial state where x is 0. Events e_0 and e_1 execute a fetch-and-add and compare-and-swap respectively, at different replicas.

"local" computation in those invocations as follows: (1) on the left, the CAS *should fail (not write to x) because it reads the value written by the* FAA *which should be equal to 1 since* FAA *reads the initial value, (2) on the right, the* CAS *should succeed (write to x) because it reads the initial value (the* FAA *will concurrently write 1 to x).*

We say that the event w is *read* by the event r if $(w, r) \in \mathsf{wr}$. Since we assumed that wr_x^{-1} is a total function, we use $\mathsf{wr}_x^{-1}(r)$ to denote the set W such that $(\mathsf{W}, r) \in \mathsf{wr}_x$. We use $\mathsf{wr}_x^{-1}(e) = \emptyset$ to indicate that e does not read x (resp. $\mathsf{wr}_x^{-1}(e) \neq \emptyset$ to indicate that e reads x).

3.2 Abstract Executions

An *abstract execution* of a distributed storage is a history with a finite set of events E along with a relation $\mathsf{rb} \subseteq E \times E$ called *receive-before*, and a total order $\mathsf{ar} \subseteq E \times E$ called *arbitration*. These relations are an abstraction of the internal communication behavior, i.e., the propagation of operation invocations between different replicas and conflict-resolution policies. The receive-before relation models information exchange between replicas and intuitively, an event w is received-before an event e on a replica r if w has been propagated to replica r before executing e. The arbitration order represents a "last-writer wins" conflict resolution policy between concurrent events and the order in which events take effect in the storage for "strong" consistency models such as Sequential Consistency or Serializability. This order may be ignored by weaker consistency models, where a read is *not* required to read from the latest update that precedes it in arbitration order, or by specific types of storage, e.g., CRDTs (see Section 7), where conflict resolution does not rely on the arbitration order.

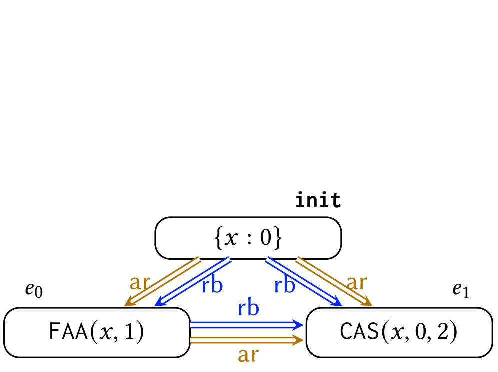
Definition 3.4. *An* abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *is a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *along with an asymmetric, irreflexive relation* receive-before $\mathsf{rb} \subseteq E \times E$ *and a strict total* arbitration order $\mathsf{ar} \subseteq E \times E$, *such that:*

(1) *propagated updates are not "forgotten" within the same replica:* $\mathsf{rb} = \mathsf{rb}; \mathsf{so}^*$[2],
(2) *events at the same replica or events that are read are necessarily received-before, and* ar *is consistent with the receive-before relation:* $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$.

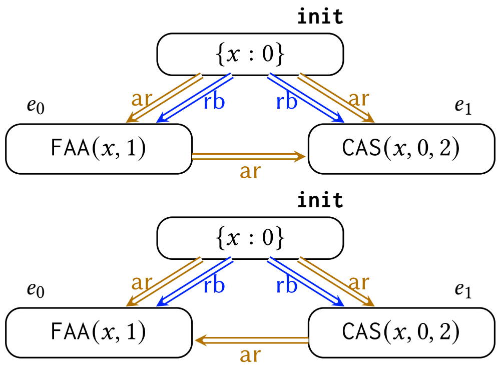
ξ is called an abstract execution of h.

The conditions above are naturally satisfied by storages where replicas execute in a single process, values are not produced "out of thin air", and the arbitration order is implemented using "consistent" timestamps, i.e. timestamps that do not contradict Lamport's clocks [23] or causality. This is the case for implementations where "ties" between concurrent operations are solved based on replica IDs (assumed to be totally ordered), or when using timestamps from a (partially-)synchronized clock – which is most often the case in practice.

[2]The symbol ; denotes the usual composition of relations



(a) Abstract execution of the history in Figure 2a.



(b) Two abstract executions of the history in Figure 2b.

Fig. 3. Abstract executions of the histories from Figure 2. Arrows represent ar and rb relations. For readability, we omit the so and wr relations. The event e_0 is received-before executing e_1 in Figure 3a but not in Figure 3b. The arbitration relation is the same in both executions.

For an event e, we use $e \in \xi$ to denote the fact that $e \in E$.

Example 3.5. *Figure 3 shows abstract executions for the histories in Figure 2. In both cases, the receive-before relation includes only the* wr *dependencies which is anyway required by definition. Reading a value at some replica r produced by an invocation e at some other replica r' should imply that e propagated to r. On the left, the arbitration order includes just the* wr *dependencies which already ensure totality. On the right,* FAA *and* CAS *are concurrent, i.e., both invocations were executed before either had a chance to propagate. We present the two possible arbitration orders. This shows that the arbitration order cannot be always determined based on the information exchanged between the replicas, i.e. by the receive-before.*

The concept of abstract execution defined earlier is subsequently used to formalize the specifications of distributed storage systems. We will start with a so-called basic class that concerns "single-object" operations.

4 Basic Storage Specifications

We present a first class of storage specifications, called *basic*, where operations read and/or write a *single value* from/to a *single object* (the operations in Example 3.1 satisfy this assumption). We will present a more general framework with multi-object operations that read and/or write multiple values or objects in Section 7.

In general, a storage specification has two parts: a *consistency model* characterizing the propagation of invocations between different replicas, and an *operation specification* which defines object states and return values. The definition of consistency models builds on the work of [11, 12] and the definition of operation specifications refines replicated data types as defined in [14]. The first two subsections define these concepts for the class of operations mentioned above, and the last subsection formalizes the validity of an abstract execution w.r.t. such storage specifications.

4.1 Basic Consistency Models

In general, a consistency model is defined as a non-empty set of *visibility* formulas that characterize the *context* in which an event (invocation of an operation) is executed (abstractly speaking). The context of an event e at a replica r is defined as the set of events, potentially from other replicas, that propagated to r prior to executing e. The notion of validity w.r.t. a consistency model defined later will require that the event e which is read by another event e' is the last in the arbitration order ar within the context of e'. This accurately models the Last-Writer-Wins conflict resolution policy (we consider other conflict resolution policies in Section 7). We define hereafter a class of so-called basic consistency models that will be extended later in Section 7.1.

Formally, a visibility formula v describes a binary relation between events which is parametrized by an object in Objs. This is written as a predicate $\mathsf{v}_x(e_1, e_2)$ meaning that v relates e_1 to e_2 for object x (explained below). A *consistency model* (criterion) CMod is a set of visibility formulas.

For a consistency model CMod and an abstract execution ξ, the context of an event r for object x is the set of all events e which are related to r by some visibility formula in CMod along with a projection of rb and ar to this set of events, i.e.,

$$\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) = (E_x, \mathsf{rb}_{E_x \times E_x}, \mathsf{ar}_{E_x \times E_x}) \text{ with } E_x = \{e \in \xi \mid \exists \mathsf{v} \in \mathsf{CMod}.\ \mathsf{v}_x(e, r)\} \tag{1}$$

We use Contexts to denote the set of all possible contexts, i.e., tuples $(E, \mathsf{rb}_E, \mathsf{ar}_E)$ where E is a finite set of events, rb_E is an asymmetric, irreflexive relation over E, and ar_E is a strict total order over E, such that $\mathsf{rb}_E \subseteq \mathsf{ar}_E$.

Basic visibility formulas (used in basic consistency models) have the following form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_n) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}. \bigwedge_{i=1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \ \wedge \varepsilon_0 \text{ writes } x \wedge \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset \tag{2}$$

where each relation $\mathsf{Rel}_i^{\mathsf{v}}$, $1 \leq i \leq n$, is defined by the grammar listed below:

$$\mathsf{Rel} ::= \mathtt{id} \mid \mathsf{so} \mid \mathsf{wr} \mid \mathsf{rb} \mid \mathsf{ar} \mid \mathsf{Rel} \cup \mathsf{Rel} \mid \mathsf{Rel}; \mathsf{Rel} \mid \mathsf{Rel}^? \mid \mathsf{Rel}^+ \mid \mathsf{Rel}^* \tag{3}$$

This formula states that ε_0 (which is e in Eq.1) is connected to ε_n (which is r in Eq.1) by a path of dependencies that go through some intermediate events $\varepsilon_1, \ldots \varepsilon_{n-1}$ (all the ε variables are interpreted as events). The constraint $\mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset$ asks that ε_n reads the object x. Every relation used in the path is a composition of so, wr, rb and ar via union $\cup$, composition of relations ;, and transitive closure $^+$. $\mathsf{Rel}^?$ is syntactic sugar for $\mathtt{id} \cup \mathsf{Rel}$, and Rel^* for $\mathtt{id} \cup \mathsf{Rel}^+$. Since the grammar includes composition the existential quantifiers in Eq.3 do not increase expressivity (one could write $(\varepsilon_0, \varepsilon_{n+1}) \in \mathsf{Rel}_1^{\mathsf{v}}; \ldots ; \mathsf{Rel}_n^{\mathsf{v}}$). These quantifiers are used to simplify proofs in Section 6.

The predicate ε_0 writes x means that ε_0 writes to object x, i.e., $\mathtt{wval}(e)(x) \downarrow$.

We write $\mathsf{v}_x(e_0, \ldots e_n)$ whenever $\mathsf{v}_x(e_0, e_n)$ holds using the events $e_1, \ldots e_{n-1}$ to instantiate the existential quantifiers. The length of v_x, denoted by $\mathsf{len}(\mathsf{v}_x)$, is the number of relations $\mathsf{Rel}_i^{\mathsf{v}}$ used in its definition (n in Equation (2)).

As mentioned above, a *basic consistency model* is a set of basic visibility formulas.

Figure 4 describes several visibility formulas and their corresponding consistency models, inspired by Biswas et al. [11]. The dashed ar edges (leading to e) should be ignored for now. Basic visibility formulas constrain events w.r.t. a single object – x. Later, we will define consistency models whose visibility formulas can impose additional constraints that concern multiple objects.

We say that a consistency model CMod_1 is *weaker than* another consistency model CMod_2, denoted $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ if intuitively, the context of any event w.r.t. CMod_1 is larger than the context w.r.t. CMod_2. Formally, $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ iff for every abstract execution ξ, event $e \in \xi$ and object x, $\mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_2])$ holds. CMod_1 and CMod_2 are *equivalent*, denoted $\mathsf{CMod}_1 \equiv \mathsf{CMod}_2$, when $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ and $\mathsf{CMod}_2 \preccurlyeq \mathsf{CMod}_1$.

We assume that every consistency model CMod includes a visibility formula $\mathsf{v}_x^{\mathsf{so}}$ (resp. $\mathsf{v}_x^{\mathsf{wr}}$) such that $\mathsf{so} \subseteq \mathsf{v}_x^{\mathsf{so}}$ (resp. $\mathsf{wr}_x \subseteq \mathsf{v}_x^{\mathsf{wr}}$) for every object $x \in$ Objs. The constraint $\mathsf{so} \subseteq \mathsf{v}_x^{\mathsf{so}}$ corresponds to the so-called "read-my-own-writes" consistency (i.e., an event "observes" every preceding event at the same replica) and $\mathsf{wr}_x \subseteq \mathsf{v}_x^{\mathsf{wr}}$ is a “well-formedness” constraint since visibility formulas will constrain the write-read relation in a history (see Definition 4.2).

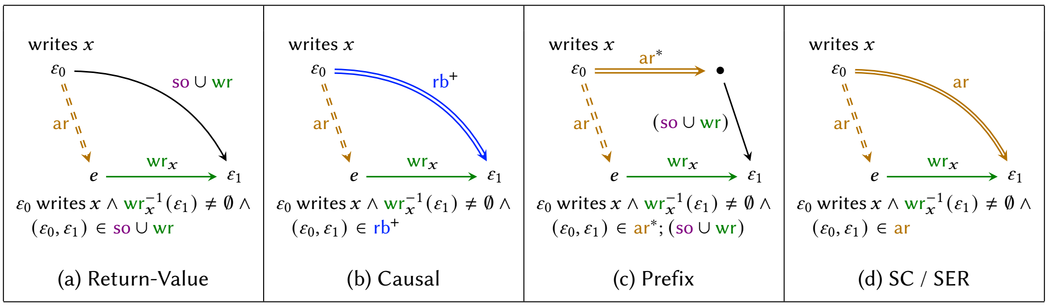

writes x
ε_0
so ∪ wr
ar
e
wr$_x$
ε_1
ε_0 writes $x \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge (\varepsilon_0, \varepsilon_1) \in \mathsf{so} \cup \mathsf{wr}$
(a) Return-Value
rb$^+$
ε_0 writes $x \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge (\varepsilon_0, \varepsilon_1) \in \mathsf{rb}^+$
(b) Causal
ar*
(so ∪ wr)
ε_0 writes $x \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge (\varepsilon_0, \varepsilon_1) \in \mathsf{ar}^*; (\mathsf{so} \cup \mathsf{wr})$
(c) Prefix
ε_0 writes $x \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge (\varepsilon_0, \varepsilon_1) \in \mathsf{ar}$
(d) SC / SER

Fig. 4. Visibility formulas defining the homonymous consistency models *Return-Value Consistency* (RVC, Figure 4a), *Causal Consistency* (CC, Figure 4b), *Prefix Consistency* (PC, Figure 4c) and *Sequential Consistency/Serializability* (SC/SER, Figure 4d). Solid edges describe the dependencies linking ε_0 and ε_1. We include the wr_x edge (and its source e) as a visualization of the constraint $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset$. Dashed ar edges are not part of the visibility formulas. These capture the Last-Writer-Wins conflict resolution policy discussed later, requiring that the event e being read succeeds all other events from the context in ar.

All consistency models in Figure 4 trivially satisfy this constraint as for any abstract execution, $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$. RVC is the weakest consistency model that our framework can describe.

4.2 Basic Operation Specifications

While visibility formulas define the context of an invocation in terms of prior invocations, the effect of an invocation is defined using the following semantical functions: rspec says whether an event reads an object or not, and wspec defines the value written by the invocation, if any. The written value may depend on the value read by the event in the case of atomic read writes like FAA and CAS. Concerning notations, for a partial function $f : A \rightharpoonup B$, we use $f(a) \downarrow$ to say that f is defined for $a \in A$, and $f(a) \uparrow$, otherwise. Similarly, for a predicate p over some set A, we use $p(a) \downarrow$ to say that p is true for a, and $p(a) \uparrow$, otherwise.

A *basic read specification* rspec is a predicate over Events. For example, Equation (4) describes the read specification of faacas. We say that an event e is a *read* event if $\mathsf{rspec}(e) \downarrow$, and in such case, we say that e reads $\mathsf{obj}(e)$.

$$\mathsf{rspec}(r) = \text{true iff } \mathsf{op}(r) = \mathtt{GET}, \mathtt{FAA}, \mathtt{CAS} \tag{4}$$

A *basic write specification* wspec is a partial function wspec : Events $\rightharpoonup$ Vals $\rightharpoonup$ Vals, that associates non-initial events to partial functions that map a read value to a value to be written. For example, Equation (5) describes the write specification of faacas.

$$\mathsf{wspec}(w)(v) = \begin{cases} v' & \text{if } w = \mathtt{PUT}(x, v') \\ v + v' & \text{if } w = \mathtt{FAA}(x, v') \\ v'' & \text{if } w = \mathtt{CAS}(x, v', v'') \wedge v = v' \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5}$$

For an event e, we say that e is a *write* event if $\mathsf{wspec}(e) \downarrow$. We assume that if $\mathsf{wspec}(e) \downarrow$, then the function $\mathsf{wspec}(e) : \mathsf{Vals} \rightharpoonup \mathsf{Vals}$ is defined for at least one value. We say that e writes x given v if $x = \mathsf{obj}(e)$ and $\mathsf{wspec}(e)(v) \downarrow$. We assume that every value v can *enable* at least one event to write, i.e., there exists $e \in$ Events s.t. $\mathsf{wspec}(e)(v) \downarrow$. We also assume that if e is a write event but it is not a read event, e.g., a PUT invocation, then $\mathsf{wspec}(e)$ is a total constant function, i.e. $\mathsf{wspec}(e) : \mathsf{Vals} \rightarrow \mathsf{Vals}$ and $\mathsf{wspec}(e)(v_1) = \mathsf{wspec}(e)(v_2)$ for all v_1, v_2.

Definition 4.1. *A* basic operation specification *is a tuple* OpSpec = $(E, \mathsf{rspec}, \mathsf{wspec})$ *where E is a set of events, such that* $\mathsf{obj}(e)$ *is a singleton for every $e \in E$.*

We use Events[OpSpec] to refer to the set of events E in an operation specification.

Operation Closure. We define some natural assumptions about basic operation specifications (it is easy to check that they hold on the faacas example with the definitions in Equation (4) and Equation (5)). We assume that E contains at least one read and one write event. We also assume that all objects support a common set of operations with identical read and write behavior, and that these operations can be executed at any replica. Formally, for every event $e \in E$, replica r, identifier id and object x there exists an event $e' \in E$ s.t. $\mathtt{rep}(e') = \mathsf{r}$, $\mathtt{id}(e') = \mathsf{id}$, $\mathsf{obj}(e') = x$, $\mathsf{rspec}(e') = \mathsf{rspec}(e)$ and $\mathsf{wspec}(e') = \mathsf{wspec}(e)$.

(Conditional) Read-Write Events. We say that OpSpec allows read-writes if E contains an event that is a read and a write event at the same time (e.g., FAA and CAS invocations); we call such events *read-write* events. If OpSpec allows read-writes, then we assume that every value can *enable* some read-write to write, i.e., for every value v, E contains a read-write event e s.t. $\mathsf{wspec}(e)(v)\downarrow$. As an example, this condition is not satisfied by a storage with only GET and TEST&SET operations (TEST&SET writes 1 if it reads 0 and nothing otherwise). Indeed, value 1 cannot enable any write.

A read-write event is called *unconditional* if for every value v, $\mathsf{wspec}(e)(v)\downarrow$ and *conditional* otherwise. For example, a FAA invocation is unconditional and a CAS invocation is conditional. We assume that if OpSpec allows conditional writes, then every value v can *disable* some conditional read-write to write, i.e., E contains a conditional read-write event s.t. $\mathsf{wspec}(e)(v)\uparrow$.

4.3 Validity w.r.t. Basic Storage Specifications

A *basic storage specification* is a pair Spec = (CMod, OpSpec) where CMod is a basic consistency model and OpSpec is a basic operation specification. Next, we formalize the validity of an abstract execution w.r.t. a basic storage specification.

The interpretation of a basic visibility formula $\mathsf{v}_x(\varepsilon_0, \varepsilon_n)$ on an abstract execution ξ is defined as expected.

Definition 4.2. *Let* Spec = (CMod, OpSpec) *be a basic storage specification. An abstract execution* $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *of a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *is* valid *w.r.t.* Spec *iff*

- *it contains events from the operation specification, i.e.,* $E \subseteq \mathsf{Events}[\mathsf{OpSpec}]$,
- *the write-read dependencies of each event* $e \in E$ *for object* x *satisfy the following:*
  - *if* e *reads object* x, *i.e.* $\mathsf{rspec}(e)\downarrow$ *and* $x \in \mathsf{obj}(e)$, e *reads from the write event in its context that is maximal w.r.t. the arbitration order:* $\mathsf{wr}_x^{-1}(e) = \{w_x^e\}$,
  - *if* e *does not read object* x, *i.e.* $\mathsf{rspec}(e)\uparrow$ *or* $x \notin \mathsf{obj}(e)$, *then* $\mathsf{wr}_x^{-1}(e) = \emptyset$.
- *the value written by each event* $e \in E$ *to object* x *is consistent with* wspec:
  - *if* e *reads object* x, *i.e.* $\mathsf{rspec}(e)\downarrow$ *and* $x \in \mathsf{obj}(e)$, *then it writes based on the value read:* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(\mathtt{wval}(w_x^e)(x))$[3],
  - *if* e *does not read object* x, *i.e.* $\mathsf{rspec}(e)\uparrow$ *or* $x \notin \mathsf{obj}(e)$, *then* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(\_)$[4],

  where $w_x^e = \max_{\mathsf{ar}} \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}])$.

A history h *is valid w.r.t.* Spec *iff there exists an abstract execution of* h *which is valid w.r.t.* Spec.

Recall that the value function, and implicitly, the operation specification, are used to interpret the visibility formulas of CMod and thus define invocation contexts.

Example 4.3. *The abstract executions described in Figure 3 are both valid w.r.t.* (CC, faacas) *as every event which is read is also received-before* ($\mathsf{wr} \subseteq \mathsf{rb}$)*. However, only Figure 3a is valid w.r.t.* (SC, faacas)*. In Figure 3a,* e_1 *reads from the writing event that precedes it w.r.t.* ar*. On the other hand, in Figure 3b,* e_1 *reads* x *from* **init** *and not from* e_0 *which is its maximal visible event w.r.t.* ar *that writes* x*. Moreover,*

[3]Since wval and wspec are partial functions, the equality also means that the left side is defined iff the right side is defined.

[4]_ represents any value in Vals.

by the symmetry between e_0 and e_1, it can be proven that any abstract execution of such history is not valid w.r.t. $(\mathsf{SC}, \mathsf{faacas})$.

5 Programs and Storage Implementations

We model programs accessing a storage and storage implementations using *Labeled Transition Systems (LTSs).* Their interaction via invocations of operations will be defined as the usual parallel composition of LTSs. We also present the notions of availability and validity of a storage implementation, key to the AFC theorem.

5.1 Labeled Transition Systems

An LTS $L = (S, A, s_0, \Delta)$ is a tuple formed of a (possibly infinite) set of *states* S, a set of *actions* A, an *initial state* $s_0 \in S$ and a (partial) *transition function* $\Delta : S \times A \rightharpoonup S$. An *execution* of L is an alternating sequence of states and actions $\rho = s_0, a_0, s_1, a_1, s_2, \ldots$ such that $\Delta(s_i, a_i) = s_{i+1}$ for each i. A state s is *reachable* if there exists an execution ending in s. A *trace* of an execution ρ is the projection of ρ over actions (the maximum subsequence of ρ formed of actions). The final state of a finite trace t, denoted by $\mathtt{state}(t)$, is the last state of ρ. The set of all traces of L is denoted by $\mathcal{T}_L$. An LTS is *finite* if all its traces are finite. For any finite trace t and action a, $\Delta(t, a)$ is defined as $\Delta(\mathtt{state}(t), a)$. If $\Delta(t, a) \downarrow$, then $t \oplus a$ is defined by appending a to t.

Let $L_1 = (S_1, A_1, s_0^1, \Delta_1)$ and $L_2 = (S_2, A_2, s_0^2, \Delta_2)$ be two LTSs. We define a parallel composition operator between L_1 and L_2 that is parametrized by a partial function $\pi : A_1 \rightharpoonup A_2$. This function allow us to define a relationship between a subset of A_1 and a subset of A_2, called *synchronized actions* of L_1 and L_2. The set of actions $a \in A_1$ for which $\pi(a)$ is not defined (resp. actions $a \in A_2$ for which $\pi^{-1}(a)$ is not defined) are the *local actions* of L_1 (resp. L_2). Without loss of generality, we assume that the set of local actions of L_1 and L_2 are disjoint.

The parallel composition of L_1 and L_2 w.r.t. π is the LTS $L_1 \parallel_\pi L_2 = (S, A, s_0, \Delta)$ where $S = S_1 \times S_2$, $A = A_1 \cup A_2$, $s_0 = (s_0^1, s_0^2)$, and Δ is defined as follows:

$$\Delta((s_1, s_2), a) ::= \begin{cases} (\Delta(s_1, a), \Delta(s_2, \pi(a))) & \text{if } a \in A_1, \pi(a) \downarrow, \Delta(s_1, a) \downarrow, \text{ and } \Delta(s_2, \pi(a)) \downarrow \\ (\Delta(s_1, a), s_2) & \text{if } a \in A_1, \pi(a) \uparrow, \text{ and } \Delta(s_1, a) \downarrow \\ (s_1, \Delta(s_2, a)) & \text{if } a \in A_2, \pi^{-1}(a) \uparrow, \text{ and } \Delta(s_2, a) \downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

(note the asymmetry due to using the function π). Whenever there is no ambiguity w.r.t. π we simply write $L_1 \parallel L_2$.

5.2 Programs and Storage Implementations

Let E be a set of events. A *program* over E is an LTS $P_E = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$ such that $E \subseteq A_\mathsf{p}$. Intuitively, this LTS models all possible interleavings between invocations on different replicas. Actions in $A_\mathsf{p} \setminus E$ represent computation steps performed by the program locally, before or after invoking operations on the storage. Also, to simplify the technical exposition, we do not consider separate transitions for calling and returning from a storage operation. Intuitively, the transitions labeled by events occur at the return time.

A *storage implementation* over E is an LTS $I_E = (S_\mathsf{i}, A_\mathsf{i}, s_0^\mathsf{i}, \Delta_\mathsf{i})$ such that A_i contains (1) an arbitrary set of local actions (representing computation/communication steps internal to the storage), and (2) pairs of events in E and their read-dependencies, i.e., pairs (e, m) where $e \in E$ and $m : \mathsf{Objs} \rightharpoonup \mathcal{P}(E)$. Intuitively, m represents the write-read dependencies of e. We also assume that each action includes an identifier, denoted by $\mathtt{id}(a)$, so that along an execution every action occurs only once. For any action $a = (e, m)$, $\mathtt{ev}(a)$ and $\mathtt{wr}\text{-}\mathtt{Set}(a)$ denote the event e and the write-read dependencies m

respectively. Also, $\mathtt{op}(a) = \mathtt{op}(e)$ is the operation type of a. To model communication, we assume that A_i includes two types of local actions, `send` actions for sending a message (from one replica to another) and `receive` to receive a message.

The formalization of send/receive actions is straightforward and we omit it. We will say that a send action *matches* a receive action if they concern precisely the same message (messages are associated with unique identifiers). For any `send`, resp., `receive`, action a at some replica r, $\mathsf{rb}\text{-}\mathtt{Set}(a)$ denotes the set of events that r sends in this message, resp., that r receives in this message. We assume that if a trace t contains any such action, for every event $e \in \mathsf{rb}\text{-}\mathtt{Set}(a)$ there must exist an action $(e, \_)$ preceding a in t. As expected, if a_s and a_r match, then $\mathsf{rb}\text{-}\mathtt{Set}(a_s) = \mathsf{rb}\text{-}\mathtt{Set}(a_r)$.

For any action $a \in A_\mathsf{p} \cup A_\mathsf{i}$, $\mathtt{rep}(a)$ denotes the replica executing a.

The interaction between a storage implementation I_E and a program P_E is defined as their asymmetric parallel composition w.r.t. a partial function $\pi : A_\mathsf{i} \rightharpoonup A_\mathsf{p}$ which is defined only for actions of the form (e, m) (as described above) by $\pi(e, m) = e$. The program and the storage implementation synchronize on events representing operation invocations. It is denoted by $I_E \parallel P_E$. By definition, traces of $I_E \parallel P_E$ include actions of the form (e, m) (coming from A_i), and local actions of P_E or I_E.

Traces of I_E (or $I_E \parallel P_E$) induce histories and abstract executions. The *induced history* of a trace t of I_E (or $I_E \parallel P_E$) is the history $h = (E^t, \mathsf{so}^t, \mathsf{wr}^t)$ where E^t is the set events e such that some action $a_e = (e, m)$ occurs in t, so^t orders events from the same replica as they occur in t, and for every object x and event e, $(\mathsf{wr}^t_x)^{-1}(e) = \mathsf{W}$ iff $\mathsf{wr}\text{-}\mathtt{Set}(a_e) = (x, \mathsf{W})$ (a_e is the action that contains e). We implicitly assume that for any event $e \in E$ different from $\mathbf{init}$, $(\mathbf{init}, e) \in \mathsf{so}^t$. We use $\mathsf{h}(t)$ to denote the induced history of a trace t.

The *induced receive-before* of a trace t of I_E (or $I_E \parallel P_E$) is the relation rb^t over events induced by the matching relation between sends and receives: $(e, e') \in \mathsf{rb}^t$ iff $(e, e') \in \mathsf{so}^t$ or there exists matching `send` and `receive` actions, a_s, a_r and a synchronized action $a = (e', \_)$ s.t. $\mathtt{rep}(a_r) = \mathtt{rep}(a)$, a_r occurs before a in t, and $e \in \mathsf{rb}\text{-}\mathtt{Set}(a_s)$ (which coincides with $\mathsf{rb}\text{-}\mathtt{Set}(a_r)$).

A trace t of I_E also induces a set of abstract executions of the form $\xi = (\mathsf{h}(t), \mathsf{rb}^t, \mathsf{ar}^t)$ where ar^t is any total order between the events in ξ that is consistent with rb^t, i.e., $\mathsf{rb}^t \subseteq \mathsf{ar}^t$ (to satisfy the requirements in Definition 3.4).

5.3 Availability and Validity of a Storage Implementation

We say that a storage implementation I_E is *available* if, intuitively, every execution of I_E terminates when interacting with a finite program P_E (executing a single synchronized action does not make a replica enter an infinite loop of local steps), and no invocation is delayed due to a replica waiting for messages.

We say that a replica $r \in \mathsf{Reps}$ is *waiting* in a trace t of some composition $I_E \parallel P_E$ if

- the program can execute some action at replica r: there is an action $a \in A_\mathsf{p}$ s.t. $\mathtt{rep}(a) = r$ and $\Delta_{P_E}(t', a) \downarrow$; where t' is obtained from t by removing all local actions of I_E and replacing every action (e', m) with e', and
- the only actions of replica r that the parallel composition can execute are `receive` actions: for every action $a \in A_\mathsf{p} \cup A_\mathsf{i}$ s.t. a is not a `receive` action and $\mathtt{rep}(a) = r$, $\Delta_{I_E \parallel P_E}(t, a) \uparrow$.

Note that the latter implies that the action a that P_E can execute after t' is necessarily an event in E (otherwise, a is a local action of P_E and the parallel composition could execute it).

Definition 5.1. *An implementation I_E is* available *if the following hold:*

- *for every finite program P_E, the composition $I_E \parallel P_E$ is also finite, and*
- *for every program P_E and every trace t of $I_E \parallel P_E$, there is no replica waiting in t.*

Given a storage specification Spec over a set of events E, a storage implementation I_E is *valid w.r.t.* Spec if every trace t induces some abstract execution which is valid w.r.t. Spec.An implementation valid w.r.t. Spec is simply called a Spec-*implementation* (or implementation of Spec).

6 The Basic Arbitration-Free Consistency Theorem

We present a simpler instance of our main result (the AFC theorem) for basic storage specifications.

To simplify the statement of the theorem, we define a normal form for basic consistency models w.r.t. a basic operation specification OpSpec. A visibility formula is called *simple* if it does *not* use composition operators between relations, i.e., the grammar in Equation (3) is replaced by: $\mathsf{Rel} ::= \mathsf{so} \mid \mathsf{wr} \mid \mathsf{rb} \mid \mathsf{ar}$. Also, a visibility formula v from a consistency model CMod is called *vacuous* w.r.t. OpSpec iff for every abstract execution ξ, ξ is valid w.r.t. (CMod, OpSpec) iff ξ is valid w.r.t. (CMod $\setminus$ {v}, OpSpec). For example, if $\mathsf{Rel}^{\mathsf{v}}_i$ and $\mathsf{Rel}^{\mathsf{v}}_{i+1}$ in Equation (2) are wr (for some i), then any instance of ε_i must be an invocation of a read-write that both reads and writes. If the operation specification does not include read-writes (e.g., a key-value store with only PUT and GET operations), such visibility formulas are vacuous.

Definition 6.1. *A basic consistency model* CMod *is called in* normal form w.r.t. a basic operation specification OpSpec *if it contains only simple visibility formulas and no visibility formula from* CMod *is vacuous w.r.t.* OpSpec.

A normal form of a basic consistency model CMod w.r.t. OpSpec is any basic consistency model CMod′ in normal form, such that for every abstract execution ξ, ξ is valid w.r.t. (CMod, OpSpec) iff ξ is valid w.r.t. (CMod′, OpSpec). We show in Appendix B that every basic consistency model CMod has a normal form. A normal form can be obtained by replacing each visibility formula v with an equivalent (possibly infinite) set of simple visibility formulas S_{v}. Each set S_{v} is obtained by recursively decomposing the union, composition and transitive closure operators in each relation $\mathsf{Rel}^{\mathsf{v}}$ (see Equation (2)).

A visibility formula is called *arbitration-free* if its definition does not use the arbitration relation ar, i.e. the grammar in Equation (3) omits the ar relation. For example, in Figure 4, RVC and CC are arbitration-free while PC and SC are not.

Definition 6.2. *A consistency model is called* arbitration-free *w.r.t. an operation specification* OpSpec *if the visibility formulas contained in some normal form w.r.t.* OpSpec *are arbitration-free.*

Defining arbitration-free via a normal form removes "redundant" occurrences of the arbitration-order, i.e. visibility relations that employ ar but are vacuous w.r.t. OpSpec. We also show in Appendix B that for every basic consistency model CMod, if some normal form consists of arbitration-free visibility formulas, then this holds for any other normal form (this is actually proved for the more general class of consistency models defined in Section 7.1).

Theorem 6.3 (Basic Arbitration-Free Consistency (AFC$_0$)). *Let* Spec = (CMod, OpSpec) *be a basic storage specification. The following statements are equivalent:*

(1) CMod *is arbitration-free w.r.t.* OpSpec,
(2) *there exists an available* Spec-*implementation.*

In the following, we present a summary for the proof of AFC$_0$, which contains a series of lemmas. We refer the reader to Appendix C for a detailed proof. Lemmas 6.4 to 6.6 show that if CMod is arbitration-free then there exists an available Spec-implementation, whereas Lemma 6.7 is used to show the converse.

6.1 Arbitration-Freeness Implies Availability

Assume that CMod is arbitration-free w.r.t. OpSpec. We first show that CMod is weaker than CC.

Lemma 6.4. *Let* Spec = (CMod, OpSpec) *be a basic storage specification. If* CMod *is arbitration-free w.r.t.* OpSpec*, then* CMod *is weaker than* CC*.*

Proof Sketch. If CMod is arbitration-free, then every simple visibility formula v in a normal form of CMod does not use ar, i.e. it only uses so, wr and rb. By Definition 3.4, $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb}$ in any abstract execution ξ. Hence, for every object x, $\mathsf{ctxt}_{\mathsf{CMod}}(r, [\xi, x]) \subseteq \mathsf{ctxt}_{\mathsf{CC}}(r, [\xi, x])$, i.e. $\mathsf{CMod} \preccurlyeq \mathsf{CC}$. □

Lemma 6.5 below implies that if a consistency model CMod is weaker than CC, then any available (CC, OpSpec)-implementation is also an available (CMod, OpSpec)-implementation.

Lemma 6.5. *Let* OpSpec *be a basic operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of basic consistency models s.t.* CMod_2 *is weaker than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$*.*

Lemma 6.6 shows that there exists an available (CC, OpSpec)-implementation, which concludes the proof of this direction.

Lemma 6.6. *Let* OpSpec *be a basic operation specification. There exists an available* (CC, OpSpec)*-implementation.*

Proof Sketch. We define an available storage implementation of (CC, OpSpec) which is an abstraction of existing CC implementations [9, 10, 25, 26].

The storage implementation I_E describes a transition function associating events with the write-read relation obtained by computing the maximum writing event on its causal past (i.e. all write events that are already received in its replica). Each replica r maintains the causal past as follows: (1) every event invoked at r is added to r's causal past, (2) after every invocation, r broadcasts a message to all other replicas that contains its causal past, (3) whenever a replica r' receives this message, it adds the included causal past to its own. Sent messages are not required to be received before executing an invocation. The latter implies trivially that I_E is an available storage implementation. The validity w.r.t. (CC, OpSpec) follows easily from the "transitive" communication of causal pasts between replicas. □

6.2 Availability Implies Arbitration-Freeness

We prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod′ of CMod contains a simple visibility formula involving ar (see Definition 6.2). By Lemma 6.7, such a formula precludes the existence of an available (CMod′, OpSpec)-implementation. Consequently, there is no available (CMod, OpSpec)-implementation, since any such implementation would also be an available (CMod′, OpSpec)-implementation – this is an easy observation as CMod is equivalent to CMod′.

Lemma 6.7. *Let* Spec = (CMod, OpSpec) *be a basic storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* i, $0 \leq i \leq \mathsf{len}(\mathsf{v})$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* (CMod, OpSpec)*-implementation.*

Proof Sketch. We assume by contradiction that there is an available implementation I_E of Spec. . We use the visibility formula v to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no `receive` action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec.

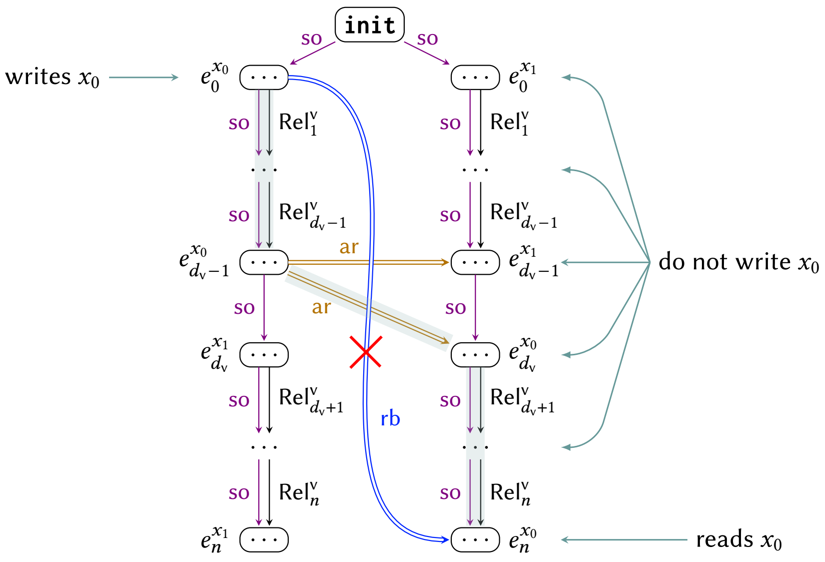

init
so
so
writes x_0
$e_0^{x_0}$
$e_0^{x_1}$
$\mathsf{Rel}_1^{\mathsf{v}}$
$\mathsf{Rel}_{d_\mathsf{v}-1}^{\mathsf{v}}$
$e_{d_\mathsf{v}-1}^{x_0}$
$e_{d_\mathsf{v}-1}^{x_1}$
ar
ar
do not write x_0
$e_{d_\mathsf{v}}^{x_1}$
$e_{d_\mathsf{v}}^{x_0}$
$\mathsf{Rel}_{d_\mathsf{v}+1}^{\mathsf{v}}$
rb
$\mathsf{Rel}_n^{\mathsf{v}}$
$e_n^{x_1}$
$e_n^{x_0}$
reads x_0

Fig. 5. Abstract execution of a trace without `receive` actions for the visibility formula v. If $i \neq d_\mathsf{v}$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$ holds because the two events are executed at the same replica (recall that $\mathsf{so} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$). If $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$, then since $\mathsf{so} \subseteq \mathsf{ar}$ and ar is transitive, we get that $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}}^{x_0}) \in \mathsf{Rel}_{d_\mathsf{v}}^{\mathsf{v}} = \mathsf{ar}$; and therefore, that $\mathsf{v}_{x_0}(e_0^{x_0}, e_n^{x_0})$ holds. However, in the absence of receives, $(e_0^{x_0}, e_n^{x_0}) \notin \mathsf{rb}$.

The program P we construct generalizes the litmus programs presented in Figure 1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq n, l \in \{0, 1\}$. The events are used to "encode" two instances v_{x_0} and v_{x_1} of the visibility formula.

Let d_v be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to $\varepsilon_{d_\mathsf{v}}$ which is not arbitration-free, and the suffix from $\varepsilon_{d_\mathsf{v}}$ up to $\varepsilon_{\mathsf{len}(\mathsf{v})}$, the arbitration-free part. Thus, v is of the form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_n) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}. \bigwedge_{i=1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \;\wedge \varepsilon_0 \text{ writes } x \wedge \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset$$

where $n = \mathsf{len}(\mathsf{v})$, $\mathsf{Rel}_i^{\mathsf{v}} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}\}$ for $i < d_v$, $\mathsf{Rel}_{d_\mathsf{v}}^{\mathsf{v}} = \mathsf{ar}$, and $\mathsf{Rel}_i^{\mathsf{v}} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$ for $i > d_\mathsf{v}$.

Replica r_l executes first events $e_i^{x_l}$ with $i < d_\mathsf{v}$ and then, events $e_i^{x_{1-l}}$ with $i \geq d_\mathsf{v}$ – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. All events in replica r_l access (read and/or write) object x_l except for $e_n^{x_l}$ which reads x_{1-l}. For ensuring that $\mathsf{v}_x(e_0^{x_l}, \ldots e_n^{x_l})$ holds in an induced abstract execution of a trace without `receive` actions, we require that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $e_{i-1}^{x_l}$ is a write event and $e_i^{x_l}$ is a read event. Figure 5 exhibits a diagram of such execution.

Example 6.8. *We illustrate the construction for Prefix Consistency* (PC) *and a Key-Value store with* PUT *and* GET *operations (their specification is defined in Section 4.2).* PC *can be defined as the following set of* simple *visibility formulas (obtained from* Prefix *in Figure 4c):*

$$\begin{array}{lll} \mathsf{v}_x^1(\varepsilon_0, \varepsilon_1) & ::= & \varepsilon_0 \text{ writes } x \;\wedge\; \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \;\wedge\; (\varepsilon_0, \varepsilon_1) \in \mathsf{so} \\ \mathsf{v}_x^2(\varepsilon_0, \varepsilon_1) & ::= & \varepsilon_0 \text{ writes } x \;\wedge\; \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \;\wedge\; (\varepsilon_0, \varepsilon_1) \in \mathsf{wr} \\ \mathsf{v}_x^3(\varepsilon_0, \varepsilon_2) & ::= & \exists \varepsilon_1.\ \varepsilon_0 \text{ writes } x \;\wedge\; \mathsf{wr}_x^{-1}(\varepsilon_2) \neq \emptyset \;\wedge\; (\varepsilon_0, \varepsilon_1) \in \mathsf{ar} \;\wedge\; (\varepsilon_1, \varepsilon_2) \in \mathsf{so} \\ \mathsf{v}_x^4(\varepsilon_0, \varepsilon_2) & ::= & \exists \varepsilon_1.\ \varepsilon_0 \text{ writes } x \;\wedge\; \mathsf{wr}_x^{-1}(\varepsilon_2) \neq \emptyset \;\wedge\; (\varepsilon_0, \varepsilon_1) \in \mathsf{ar} \;\wedge\; (\varepsilon_1, \varepsilon_2) \in \mathsf{wr} \end{array} \quad (6)$$

Observe that v_x^4 is vacuous w.r.t. the specification of PUT *and* GET *since it implies that ε_2 reads from multiple events, and* PUT *and* GET *read a single object at a time. Thus, the normal form of* PC *w.r.t. the specification of* PUT *and* GET *contains only the first three visibility formulas above.*

The only visibility formula which is not arbitration-free is v_x^3. We have that the index $d_\mathsf{v} = 1$ and we consider the following types of events:

$$e_0^{x_0} : \mathtt{PUT}(x_0, \_), e_1^{x_0} : \mathtt{PUT}(x_1, \_), e_2^{x_0} : \mathtt{GET}(x_0)$$
$$e_0^{x_1} : \mathtt{PUT}(x_1, \_), e_1^{x_1} : \mathtt{PUT}(x_0, \_), e_2^{x_1} : \mathtt{GET}(x_1)$$

Replica r_0 executes $e_0^{x_0}$ and then $e_1^{x_1}$ and $e_2^{x_1}$. Replica r_1 executes $e_0^{x_1}$ and then $e_1^{x_0}$ and $e_2^{x_0}$.

Given such a program P, the proof proceeds as follows:

(1) There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma C.5): Since I_E is available, it can always delay receiving messages, and execute other actions instead. Then, as P is a finite program, such an execution must be finite.

(2) The trace t induces a history $h_\mathsf{v} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_\mathsf{v} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$ (ar is arbitrary as long as $\mathsf{rb} \subseteq \mathsf{ar}$). As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_v read the latest value w.r.t. so written on their associated object in ξ_v (Lemma C.6). In particular, we deduce that all traces of P without `receive` events induce the same history and therefore, the induced history does not change when the induced arbitration order changes.

(3) Since ar is a total order, either $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$ or $(e_{d_\mathsf{v}-1}^{x_1}, e_{d_\mathsf{v}-1}^{x_0}) \in \mathsf{ar}$. W.l.o.g., assume that $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$. By Lemma C.7, we deduce that $e_0^{x_0} \in \mathrm{ctxt}_{x_0}(e_n^{x_0}, [\xi_\mathsf{v}, \mathrm{CMod}])$. The proof is explained in Figure 5: if $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$, then all events $e_i^{x_0}$ form a path in such way that $\mathsf{v}_{x_0}(e_0^{x_0}, \dots e_n^{x_0})$ holds in ξ_v.

(4) Since $e_n^{x_0}$ is the only event at r_1 that reads or writes x_0 and events in ξ_v read the latests values w.r.t. so in ξ_v, we deduce that $e_n^{x_0}$ reads x_0 from **init**. However, as $e_0^{x_0} \in \mathrm{ctxt}_{x_0}(e_n^{x_0}, [\xi_\mathsf{v}, \mathrm{CMod}])$ and **init** precedes $e_0^{x_0}$ in arbitration order, we deduce that $e_n^{x_0}$ does not read the latest value w.r.t. ar, i.e. $\mathrm{rspec}(e_n^{x_0}) \downarrow$ but $\mathsf{wr}_{x_0}^{-1}(e_n^{x_0}) \neq \{\max_{\mathsf{ar}} \mathrm{ctxt}_{x_0}(e_n^{x_0}, [\xi_\mathsf{v}, \mathrm{CMod}])\}$. Therefore, ξ_v is not valid w.r.t. Spec (see Definition 4.2). This contradicts the hypothesis that I_E is an implementation of Spec. □

The corollary below is a direct consequence of Theorem 6.3 and Lemma 6.4.

Corollary 6.9. *Let* OpSpec *be a basic operation specification. The strongest consistency model* CMod *for which* (CMod, OpSpec) *admits an available implementation is* CC.

7 Generalized Distributed Storage Specifications

We describe a generalization of the basic storage specifications from Section 4 along three dimensions: a larger class of consistency models, multi-object operations, and more general read behaviors. To rule out anomalous behaviors in this generalization, we introduce a set of additional assumptions. Figure 6 summarizes the structure of storage specifications and the relationship between basic and generalized specifications in terms of assumptions.

7.1 Consistency Models

The set of basic consistency models (Section 4.1) does *not* include (parallel) snapshot isolation, and the version of k-bounded staleness considered in Section 2. Snapshot Isolation, k-Bounded Staleness and Parallel Snapshot Isolation are defined, respectively, using the visibility formulas

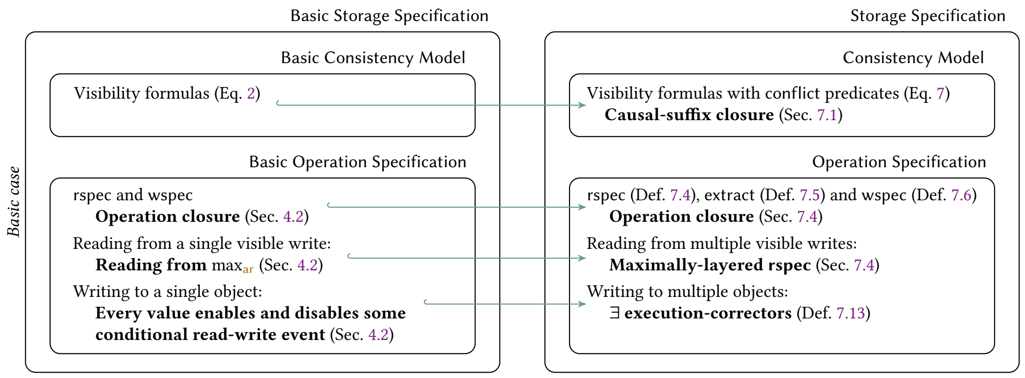

Basic Storage Specification
Basic Consistency Model
Visibility formulas (Eq. 2)
Basic Operation Specification
rspec and wspec
Operation closure (Sec. 4.2)
Reading from a single visible write:
Reading from max_ar (Sec. 4.2)
Writing to a single object:
Every value enables and disables some conditional read-write event (Sec. 4.2)
Basic case
Storage Specification
Consistency Model
Visibility formulas with conflict predicates (Eq. 7)
Causal-suffix closure (Sec. 7.1)
Operation Specification
rspec (Def. 7.4), extract (Def. 7.5) and wspec (Def. 7.6)
Operation closure (Sec. 7.4)
Reading from multiple visible writes:
Maximally-layered rspec (Sec. 7.4)
Writing to multiple objects:
∃ execution-correctors (Def. 7.13)
General case

Fig. 6. Conceptual map relating basic and generalized storage specifications (Sections 4 and 7). Storage specifications are composed of consistency models and operation specifications. Assumptions are written in bold text. Arrows denote how definitions/assumptions translate from the basic case to the general case.

Conflict (Figure 7a), k-Bounded (Figure 7b)[5], and n-PSI (Figure 7c). To include such consistency models in our formalization, we extend the syntax of visibility formulas so that the intermediate events can be further constrained via the wrCons formula:

$$\mathsf{v}_x(\varepsilon_0,\varepsilon_n) ::= \exists \varepsilon_1,\ldots,\varepsilon_{n-1}.\ \bigwedge_{i=1}^{n}(\varepsilon_{i-1},\varepsilon_i)\in \mathsf{Rel}_i^{\mathsf{v}} \wedge \mathsf{wr}_x^{-1}(\varepsilon_n)\neq\emptyset \wedge \mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0,\ldots\varepsilon_n) \tag{7}$$

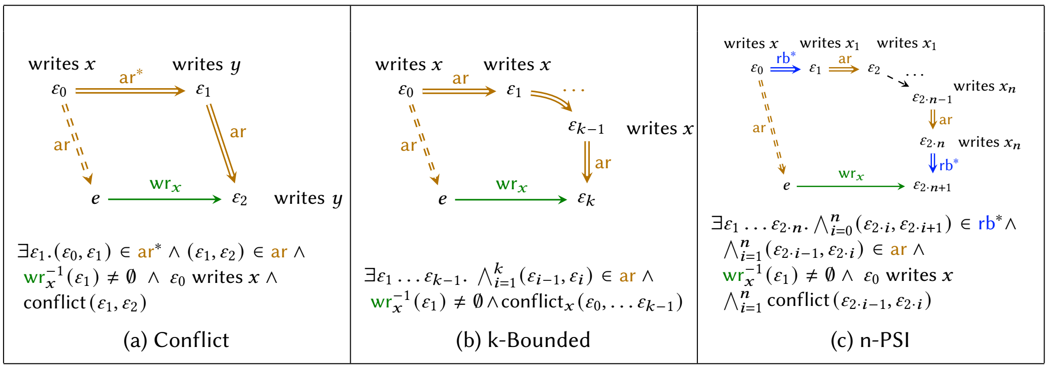
The formula $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0,\ldots\varepsilon_n)$ is a conjunction of predicates $\mathsf{conflict}(E)$ and $\mathsf{conflict}_x(E)$ with $E\subseteq\{\varepsilon_0,\ldots\varepsilon_n\}$. The predicate $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) means that *all* the events in E write on some object y (resp., the object x). Since we want to preserve the constraint ε_0 writes x from basic visibility formulas, we require that there exists a set $E\subseteq\{\varepsilon_0,\ldots\varepsilon_n\}$ s.t. $\varepsilon_0\in E$ and $\mathsf{conflict}_x(E)$ is included in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0,\ldots\varepsilon_n)$ (E can be the singleton ε_0). The interpretation of a conflict predicate in an abstract execution ξ is done as expected: a predicate $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) holds iff there exists an object y s.t. for every $e\in E$, e writes y in ξ (resp. e writes x in ξ). As before, the predicate ε writes y is true iff $\mathtt{wval}(e)(y)\downarrow$.

From this point on, a consistency model is defined as a set of visibility formulas, as in Equation (7).

Normal Form. We generalize the normal form of a consistency model to take into account conflict predicates. A consistency model in normal form only contains visibility formulas that are simple, non-vacuous and "conflict-maximal". A *conflict-strengthening* of a visibility formula v is a visibility formula v' obtained from v by (1) replacing some occurrence of $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) with $\mathsf{conflict}(E')$ (resp., $\mathsf{conflict}_x(E')$) where E' is a strict superset of E or (2) removing predicate $\mathsf{conflict}(E)$ if $\mathsf{conflict}_x(E)$ also belongs to v. A visibility formula v is *conflict-maximal* w.r.t. OpSpec iff there is no conflict-strengthening v' such that for every execution ξ over events in Events[OpSpec], object x, and events $e_0,\ldots e_{\mathsf{len}(\mathsf{v})}$, if $\mathsf{v}_x(e_0,\ldots e_{\mathsf{len}(\mathsf{v})})$ holds in ξ, then $\mathsf{v}'_x(e_0,\ldots e_{\mathsf{len}(\mathsf{v})})$ holds in ξ as well. A consistency model CMod is *conflict-maximal* w.r.t. OpSpec iff all its visibility formulas are conflict-maximal w.r.t. OpSpec.

For example, if $\mathsf{Rel}_i^{\mathsf{v}}=\mathsf{wr}$, any instance of ε_i must write on some object y. In conflict-maximal visibility formulas, this fact is represented with a conflict predicate ($\mathsf{conflict}(E)$ or $\mathsf{conflict}_x(E)$) s.t. $\varepsilon_{i-1}\in E$. If OpSpec requires that every event reading y also writes on y, then in a conflict-maximal visibility formula, both $\varepsilon_{i-1},\varepsilon_i$ belong to E. In general, if in any abstract execution, the events

[5]Our version of k-Bounded Staleness corresponds to the (k,T)-Bounded Staleness with $T=\infty$ as defined in [28].



$\exists \varepsilon_1.(\varepsilon_0, \varepsilon_1) \in \mathsf{ar}^* \wedge (\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge \varepsilon_0$ writes $x \wedge \mathsf{conflict}(\varepsilon_1, \varepsilon_2)$

(a) Conflict

$\exists \varepsilon_1 \dots \varepsilon_{k-1}. \bigwedge_{i=1}^{k} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{ar} \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge \mathsf{conflict}_x(\varepsilon_0, \dots \varepsilon_{k-1})$

(b) k-Bounded

$\exists \varepsilon_1 \dots \varepsilon_{2\cdot n}. \bigwedge_{i=0}^{n} (\varepsilon_{2\cdot i}, \varepsilon_{2\cdot i+1}) \in \mathsf{rb}^* \wedge \bigwedge_{i=1}^{n} (\varepsilon_{2\cdot i-1}, \varepsilon_{2\cdot i}) \in \mathsf{ar} \wedge \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge \varepsilon_0$ writes x $\bigwedge_{i=1}^{n} \mathsf{conflict}(\varepsilon_{2\cdot i-1}, \varepsilon_{2\cdot i})$

(c) n-PSI

Fig. 7. Conflict, k-Bounded and n-PSI visibility formulas used to define *Snapshot Isolation* (SI), *Bounded Staleness* (BS) and *Parallel Snapshot Isolation* (PSI). SI is defined by Prefix (Figure 4c) and Conflict, BS is defined by k-Bounded and Return-Value (Figure 4a), and PSI is defined by Causal (Figure 4b) and the set of visibility formulas {n-PSI | $n \geq 1$}.

instantiating $\varepsilon_{i_1}, \dots, \varepsilon_{i_j}$ from v_x always conflict (resp. they always write x), then the visibility formula v must contain the predicate $\mathsf{conflict}(\varepsilon_{i_1}, \dots, \varepsilon_{i_j})$ (resp. $\mathsf{conflict}_x(\varepsilon_{i_1}, \dots, \varepsilon_{i_j})$).

Definition 7.1. *A consistency model* CMod *is called in* normal form w.r.t. a operation specification OpSpec *if it contains only simple, conflict-maximal visibility formulas and no visibility formula from* CMod *is vacuous w.r.t.* OpSpec.

Under some operation specifications, consistency models can be equivalent due to conflict predicates. For example, in a storage with only FAA operations, SI and SER are equivalent due to the Conflict visibility formula: in this specification, every event is both a read and a write event and so any event reading x conflicts with an event writing x.

Similarly to Section 6, we say that a consistency model CMod is *arbitration-free* w.r.t. an operation specification OpSpec if there exists a consistency model in general normal form w.r.t. OpSpec that is equivalent to CMod and whose visibility formulas are arbitration-free. Appendix B demonstrates the existence of a normal form and shows that it is not possible for two normal forms to differ solely in that one includes only arbitration-free visibility formulas while the other does not. This result confirms that arbitration-freedom is not a property of the chosen normal form, but rather an inherent characteristic of the definitions of CMod and OpSpec.

Causal Suffix Closure. We introduce an assumption about consistency models which is used in the proof of the AFC theorem in order to find counterexamples to availability that involve only two replicas. This assumption is satisfied by all practical cases that we are aware of (see Example 7.3).

Therefore, we assume that every normal form CMod of a consistency model is *closed under causal suffixes*, i.e., for every visibility formula $\mathsf{v}_x \in$ CMod, CMod contains every arbitration-free "suffix" of v_x that starts with an event writing x. Thinking about a visibility formula v as a path of dependencies (between the pairs $(\varepsilon_{i-1}, \varepsilon_i)$), a suffix of v is a suffix of that path. For example, the visibility formulas s and s' described in Equation (9) and Equation (10) are suffixes of the visibility formula in Equation (8).

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_3) = \exists \varepsilon_1, \varepsilon_2.(\varepsilon_0, \varepsilon_1) \in \mathsf{rb} \wedge (\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \wedge (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \wedge \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \mathsf{conflict}_x(\varepsilon_0, \varepsilon_1, \varepsilon_2) \tag{8}$$

$$s_x(\varepsilon_1, \varepsilon_3) = \exists \varepsilon_2.(\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \ \wedge \ (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \ \mathsf{conflict}_x(\varepsilon_1, \varepsilon_2) \tag{9}$$

$$s'_x(\varepsilon_2, \varepsilon_3) = (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \ \mathsf{conflict}_x(\varepsilon_2) \tag{10}$$

Formally, let v_x be a visibility formula defined as in Equation (7). Let $\mathsf{conflict}_x(\mathsf{v}_x)$ be the union of the sets E such that $\mathsf{conflict}_x(E)$ occurs in the definition of v_x. For any variable $\varepsilon_k \in \mathsf{conflict}_x(\mathsf{v}_x)$, the ε_k-suffix of v_x is the formula obtained by (1) removing the quantifiers for the first k quantified events, $e_1 \ldots e_k$, and (2) removing all occurrences of the (now) free variables $e_0, \ldots e_{k-1}$, i.e.:

$$\mathsf{suff}_x(\mathsf{v}_x, k)(\varepsilon_k, \varepsilon_n) ::= \exists \varepsilon_{k+1}, \ldots, \varepsilon_{n-1}. \bigwedge_{i=k+1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \land \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset \land \mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \ldots \varepsilon_n)$$

where $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \ldots \varepsilon_n)$ is obtained from $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \ldots \varepsilon_n)$ by projecting all the conflict predicates over the set of events $E_k = \{\varepsilon_k, \ldots, \varepsilon_n\}$, i.e., a predicate $\mathsf{conflict}(E)$ (resp. $\mathsf{conflict}_x(E)$) occurs in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \ldots \varepsilon_n)$ iff $\mathsf{conflict}(E \cap E_k)$ (resp. $\mathsf{conflict}_x(E \cap E_k)$) occurs in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \ldots \varepsilon_n)$.

We refer to arbitration-free suffixes as *causal*, since the remaining dependencies intuitively reflect broader notions of causality. The intuition behind this notion of closure is that the context of an invocation should be upward-closed with respect to causality—meaning that if an update (writing x) is included, then any later updates (writing x) along the dependency path defined by the visibility formula that lie in its causal past must also be included.

We say that a visibility formula v' *subsumes* a visibility formula v of the same length if for every i, $1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{v'}$ is stronger or equal than $\mathsf{Rel}_i^{\mathsf{v}}$. We say that rb is stronger than so and wr, and ar is stronger than rb, so and wr. The extension of "being stronger" to any relation Rel described using Equation (3) is done as expected, as all our operators are positive (there are no negations).

Definition 7.2. *A consistency model* CMod *is* closed under causal suffixes *if for every* $\mathsf{v}_x \in \mathsf{CMod}$ *and* $\varepsilon_k \in \mathsf{conflict}_x(\mathsf{v}_x)$, CMod *includes some visibility formula* v' *that subsumes every arbitration-free suffix of* v.

Example 7.3. *A consistency model containing the visibility formula* v *in Equation* (8) *must also contain the visibility formula* s' *in order to be closed under causal suffixes. Note that* s *uses arbitration and it is not required to be included.*

Any basic consistency model is closed under causal suffixes because every basic visibility formula has no proper arbitration-free suffix. Indeed, $\mathsf{conflict}_x(\mathsf{v}_x)$ *contains just the first event* ε_0 *(assuming that* ε_0 writes x *is rewritten as* $\mathsf{conflict}_x(\{\varepsilon_0\})$*). The models described in Figures 4 and 7 are trivially closed under causal suffixes because their visibility formulas have no arbitration-free suffixes.*

7.2 Operation Specifications

We generalize operation specifications to allow operations to access (read or write) multiple objects, and to support read values that are not limited to the inputs of individual write operations. For example, this includes multi-value reads that return all concurrently written values for an object, or counter reads that return an aggregated value computed from all observed increments.

The generalized reading behavior is modeled using two functions rspec and extract described hereafter. We also introduce a generalized wspec function. Therefore, rspec selects from a given context the events (updates) which are relevant for a reading invocation, extract defines the value read by an invocation, if any (based on the output of rspec), and wspec defines the value written by an invocation, if any (to model conditional read-writes, this is based on the output of extract).

Definition 7.4. *A* read specification $\mathsf{rspec} : \mathsf{Events} \to \mathsf{Objs} \to \mathsf{Contexts} \to \mathcal{P}(\mathsf{Events})$ *is a function such that for every object* x*, context* $c = (E, \mathsf{rb}, \mathsf{ar})$ *and event* e*:*

(1) *well-formedness:* $\mathsf{rspec}(e)(x, c) \subseteq E$*, and if* e *is an initial event,* $\mathsf{rspec}(e)(x, c) = \emptyset$*, and*
(2) *unconditional reading: if* $\mathsf{rspec}(e)(x, c) \neq \emptyset$ *for some context* c*, then for every non-empty context* c', $\mathsf{rspec}(e)(x, c') \neq \emptyset$

Equations (11) to (13) describe the read specifications of faacas, a key-value store k-mv with PUT(x, v) and multi-value GET(x) operations (Appendix A.2), and a collection of distributed counters counter with `inc`(x) and `rd`(x) operations (Appendix A.3). Concerning the relationship to *basic* read specifications, note that the faacas specification in Equation (4) was simpler because the constraint from Equation (11) was imposed in the notion of validity for abstraction executions (Definition 4.2). For multi-value reads (Equation (12)), the read specification selects the maximal elements in the receive-before relation (which models causality), and for a counter (Equation (13)), it returns all events in the context.

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \{\max_{\mathsf{ar}} E\}, & \text{if } r \in \{\mathsf{GET}(x), \mathsf{FAA}(x, v), \mathsf{CAS}(x, v, v')\} \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{11}$$

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \max_{\mathsf{rb}} E, & \text{if } r = \mathsf{GET}(x) \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{12}$$

$$\mathsf{rspec}(r)(x, c) = \begin{cases} E, & \text{if } r = \mathtt{rd}(x) \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{13}$$

The extract specification below computes the value returned from an object x based on the set of invocations writing x returned by the read specification which are paired with values they write (this will become clearer when defining the application of these functions on an abstract execution).

Definition 7.5. *An* extract specification extract : Events $\longrightarrow$ Objs $\longrightarrow \mathcal{P}$(Events $\times$ Vals) $\longrightarrow$ Vals, *such that* extract(**`init`**) *is defined for every initial event* **`init`**.

Equation (14) describes the extract specification of faacas: the value extracted for GET, FAA and CAS coincides with the value written by some previous PUT/FAA/CAS operation. Equation (15) describes the extract specification of k-mv: the value extracted for GET is the set of values written by some previous PUT. In the case of counter, Equation (16), the value extracted for `rd` returns the number of increment invocations in the input, which equals $|R|$ minus one for the initial event **`init`** which is always included in R (since it is so before all other events).

$$\mathsf{extract}(r)(x, R) = \begin{cases} v & \text{if } r \in \{\mathsf{GET}(x), \mathsf{FAA}(x, v'), \mathsf{CAS}(x, v', v'')\} \text{ and } R = \{(w, v)\} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{14}$$

$$\mathsf{extract}(r)(x, R) = \begin{cases} \{v \mid (\_, v) \in R\} & \text{if } r = \mathsf{GET}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{15}$$

$$\mathsf{extract}(r)(x, R) = \begin{cases} |R| - 1 & \text{if } r = \mathtt{rd}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{16}$$

Finally, the write specification computes the value written by an invocation to an object x, based on the values it reads. This makes it possible to model atomic read-writes, e.g., a compare-and-swap, which may write or not depending on what they read, or the value they write may change depending on what they read, e.g., a Fetch-and-Add.

Definition 7.6. *A* write specification wspec : Events $\longrightarrow$ Objs $\longrightarrow$ Vals $\longrightarrow$ Vals *is a function such that* wspec(**`init`**) *is defined for every initial event* **`init`**.

The write specification of faacas and k-mv, Equation (17), describes that its write operations are PUT, FAA and CAS. PUT and FAA unconditionally writes on x while CAS does it depending on the read-and-extracted value of x; where x is the only object accessed by the invocation. In the case of counter, Equation (19), only the operation `inc`(x) writes, writing a dummy value 1 just to indicate that the write has taken place.

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathsf{PUT}(x,v') \\ v+v' & \text{if } w = \mathsf{FAA}(x,v') \\ v'' & \text{if } w = \mathsf{CAS}(x,v',v'') \wedge v = v' \\ \text{undefined} & \text{otherwise} \end{cases} \tag{17}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathsf{PUT}(x,v') \\ \text{undefined} & \text{otherwise} \end{cases} \tag{18}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} 1 & \text{if } w = \mathtt{inc}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{19}$$

Definition 7.7. *An* operation specification *is a tuple* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *where* E *is a set of events.* $\mathsf{Events}[\mathsf{OpSpec}]$ *refers to the set of events* E *in an operation specification.*

Appendix A contains more examples of operation specifications, including SQL statements.

7.3 Validity w.r.t. Storage Specifications

We extend the notion of validity for abstract executions to (general) storage specifications, in a way that is similar to the case of basic storage specifications (Section 4.3). We use the extension of rspec, extract, and wspec to abstract executions defined below:

$$\mathsf{rspec}(e)(x,[\xi,\mathsf{CMod}]) = \mathsf{rspec}(e)(x,\mathsf{ctxt}_x(e,[\xi,\mathsf{CMod}]))$$

$$\mathsf{extract}(e)(x,[\xi,\mathsf{CMod}]) = \mathsf{extract}(e)\ (x,\{(e',\mathtt{wval}(e)(x)) \mid e' \in \mathsf{rspec}(e)(x,[\xi,\mathsf{CMod}])\})$$

$$\mathsf{wspec}(e)(x,[\xi,\mathsf{CMod}]) = \mathsf{wspec}(e)(x,\mathsf{extract}(e)(x,[\xi,\mathsf{CMod}]))$$

Definition 7.8. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. An abstract execution* $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *of a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *is* valid *w.r.t.* Spec *iff*

- ξ *contains events from the operation specification, i.e.,* $E \subseteq \mathsf{Events}[\mathsf{OpSpec}]$,
- *for every event* $r \in E$, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x,[\xi,\mathsf{CMod}])$, *and*
- *the value written by each event* $e \in E$ *to object* x *is consistent with* wspec, *i.e.,* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(x,[\xi,\mathsf{CMod}])$.

A history h *is valid w.r.t.* Spec *iff there exists an abstract execution of* h *which is valid w.r.t.* Spec.

Observe that Definition 7.8 coincides with Definition 4.2 for storage systems that also admit basic storage specifications, e.g., faacas.

7.4 Assumptions About Operation Specifications

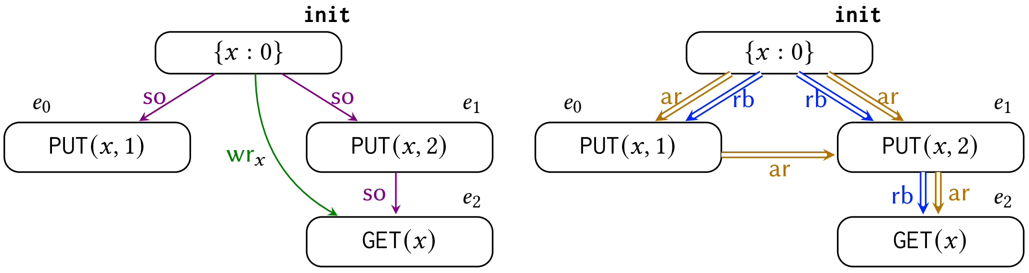
To avoid pathological behaviors in the generalization of specifications, we make several assumptions.

Maximally-Layered Read Specifications. For any basic operation specification OpSpec, the validity of an abstract execution w.r.t. a stronger consistency model (and OpSpec) implies validity w.r.t. a weaker one (see Lemma 6.5). In general, this is not true for operation specifications as described in this section (see Example 7.9). Therefore, we introduce an assumption about read specifications, called *maximally-layered*, which ensures that this property remains true.

Example 7.9. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification of a key-value store with* GET *and* PUT *operations whose read specification is given by Equation (20).*

$$\mathsf{rspec}(e)(x,c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \textit{if} \nexists e' \in E \textit{ s.t. } \mathtt{rep}(e) \neq \mathtt{rep}(e') \textit{ and } c = (E,\mathsf{rb},\mathsf{ar}) \\ \mathbf{init} & \textit{otherwise} \end{cases} \tag{20}$$

We compare the validity of the abstract execution ξ *depicted in Figure 8 w.r.t.* SC *and* CC *(observe that* $\mathsf{CC} \preccurlyeq \mathsf{SC}$*). Under* SC *both* e_0 *and* e_1 *are visible to* e_2*, which implies* $\mathsf{rspec}(e_2)(x,[\xi,\mathsf{SC}]) =$



(a) History of a key-value store with PUT and GET. (b) An abstract execution of the history in Figure 8a.

Fig. 8. A history and an abstract execution of the operation specification in Example 7.9. For readability, we omit the so and wr relations from the abstract execution. Events e_1 and e_2 are executed in the same replica, different from e_0's replica.

$\{$**init**$\}$. *Therefore, ξ is valid w.r.t.* SC. *However, under* CC, *only e_1 is visible to e_2, which implies* $\mathsf{rspec}(e_2)(x, [\xi, \mathsf{CC}]) = \{e_1\}$, *and therefore, ξ is not valid w.r.t.* CC.

Let $\leq$ be a partial order over a set A. A chain of $\leq$ is a subset of A which is totally ordered w.r.t. $\leq$. The *layer* of an element $a \in A$ is the size of the largest chain of $\leq$ which includes a but no elements smaller than a, and a maximal element. For instance, the layer of a maximal element is 1 (the aforementioned largest chain includes just the element itself), the level of a strict predecessor of a maximal element is 2, and so on. A subset $B \subseteq A$ is called *k-maximally layered* w.r.t. $\leq$ if B is the set of all elements in A of layer $k' \leq k$. When $\leq$ is also a total order, the notion of maximally layered is equivalent to being upward closed w.r.t. $\leq$. Otherwise, it is equivalent to being upward closed w.r.t. every total extension of $\leq$.

A read specification rspec is *k-maximally layered* w.r.t. ar (resp. rb^+) if for every object x, context $c = \{E, \mathsf{rb}, \mathsf{ar}\}$, and event e, either $\mathsf{rspec}(e)(x, c) = \emptyset$ or $\mathsf{rspec}(e)(x, c)$ is k-maximally layered w.r.t. ar (resp. rb^+). The *layer bound* of rspec is defined as k. To cover cases where there is no such k, we say that a read specification rspec is *∞-maximally layered* if for every x, context $c = \{E, \mathsf{rb}, \mathsf{ar}\}$, and event e, either $\mathsf{rspec}(e)(x, c) = \emptyset$ or E; and we say that the layer bound is ∞. When the layer bound and the partial order (ar or rb^+) are irrelevant, we simply say that rspec is *maximally layered.*

Example 7.10. *For example,* faacas *is 1-maximally layered w.r.t.* ar, k-mv *is 1-maximally layered w.r.t.* rb^+ *and* counter *is ∞-maximally layered. On the other hand, the read specification in Example 7.9 is not maximally layered since it can sometimes return* **init** *from a non-empty context.*

Lemma 7.11. *Let* OpSpec *be a maximall-layered operation specification and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of consistency models such that* CMod_2 *is stronger than* CMod_1. *Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$.

Operation Closure. As in Section 4.2, we assume that OpSpec contains at least a read and a write event. Also, we assume that all objects support a common set of operations with identical read and write behavior, and that these operations can be executed at any replica. Formally, for every event $e \in E$, replica r, and identifier id, there exists an event e' s.t. $\mathtt{rep}(e') = \mathsf{r}$, $\mathtt{id}(e') = \mathsf{id}$, $\mathsf{obj}(e') = \mathsf{obj}(e)$, $\mathsf{rspec}(e') = \mathsf{rspec}(e)$, $\mathsf{extract}(e') = \mathsf{extract}(e)$, and $\mathsf{wspec}(e') = \mathsf{wspec}(e)$.

We also assume that operations apply uniformly to any set of objects. To formalize this assumption, we define a notion of *domain* for an operation specification OpSpec which is any set of objects D s.t. there is an event $e \in \mathsf{Events}[\mathsf{OpSpec}]$ for which $\mathsf{obj}(e) = D$. We assume that domains are "symmetric", i.e. if D is a domain for OpSpec, then for every pair of objects $x \in D$ and $y \in \mathsf{Objs} \setminus D$, the set $D' = D \setminus \{x\} \cup \{y\}$ is also a domain for OpSpec. If OpSpec allows single-object read/write/read-write events (defined as in Section 4.2), we assume that for every object x, there exists a read/write/read-write event whose domain is $\{x\}$. Also, we assume that if OpSpec allows a *multi-object* read/write/read-write event e such that $\mathsf{obj}(e)$ is a finite set of size at least 2, then for every non-empty finite set $D \subseteq \mathsf{Objs}$, D is a domain of a read/write/read-write event in OpSpec.

Correctors. In addition, we assume that if OpSpec permits conditional read-write events–which write to a set of objects X based on values they read (possibly from other objects) in some context–then any execution can be extended with some conditional read-write event e that writes to every object in X, modulo a so-called correction defined below. This property is only relevant for events with $|\mathsf{obj}(e)| > 1$ (and therefore, irrelevant for basic storage specifications). Our proof will rely on the existence of such extensions.

Example 7.12. *To provide some intuition about the need for corrections, consider a specification formed of prefix consistency (*PC*) and an operation specification with two multi-object operations,* `InsAbs` *and* `DelPre`*, under Last-Writer-Wins (LWW) conflict resolution (i.e., the read specification selects the maximal invocation from the context w.r.t.* ar*) (see Appendix A.4).* `InsAbs`(X, v) *checks for every object $x \in X$ if it is present, and inserts it with value v if not, and* `DelPre`(X) *deletes every object $x \in X$ as long as it was present.Assume an abstract execution ξ, and an event e from ξ whose context implies that x is absent and y is present. If e is an invocation of* `InsAbs` *(resp.,* `DelPre`*), then it can not write both objects since x is absent and y is present.*

We introduce the notion of *corrector*, a set of auxiliary events that modify the context, ensuring the existence of an event that can write to both objects. For instance, in the scenario presented in Example 7.12, if e is an invocation of `InsAbs`$(\{x, y\}, 1)$, the corrector will add a `DelPre`$(\{y\})$ invocation in its context, so both objects are absent.

We start by defining some notations. Let Spec $=$ (CMod, OpSpec) be a storage specification, $\xi = (h, \mathsf{rb}, \mathsf{ar})$ an abstract execution of a history $h = (E, \mathsf{so}, \mathsf{wr})$, and $e \in E$ an event. A *correction* of e in ξ with an event a, denoted by $\xi \overset{a}{\curlyvee} e$, is an abstract execution $\xi' = (h', \mathsf{rb}', \mathsf{ar}')$ associated to a history $h' = (E \cup \{a\}, \mathsf{so}', \mathsf{wr}')$ obtained by adding a as the immediate rb-predecessor and ar-predecessor of e. If $\mathtt{rep}(e) = \mathtt{rep}(a)$, then a is also the immediate so-predecessor of e. The write-read dependencies (wr^{-1}) of every event in ξ remain the same. Multiple corrections exist because the write-read and receive-before dependencies of a are not constrained. This allows flexibility on correcting ξ while preserving validity w.r.t. Spec.

The correction of ξ with a sequence of events $\vec{\mathsf{s}} = (a_1, a_2, \ldots)$, denoted by $\xi \overset{\vec{\mathsf{s}}}{\curlyvee} e$, is defined as expected, by iteratively correcting ξ with all events in $\vec{\mathsf{s}}$ in the order defined by $\vec{\mathsf{s}}$. Therefore, if e' is the immediate ar-predecessor of e in ξ, the ar order in $\xi \overset{\vec{\mathsf{s}}}{\curlyvee} e$ will have $a_1, a_2, \ldots$ inserted in between e' and e (in this order). Similarly for rb and possibly for so.

For a (partial) mapping $f : A \rightarrow B$ and a total order $<$ over A, the sequence of elements in B mapped by f and ordered according to $<$ is denoted by $\mathsf{seq}_<(f)$. Formally, $\mathsf{seq}_<(f) = (f(a_1), f(a_2), \ldots)$ such that $f(a_i) \downarrow$ and $a_i < a_{i+1}$ for all i. We omit the subscript $<$ when it is understood from the context.

Also, if ξ is an abstract execution, then $\xi \oplus e$ is an abstract execution obtained from ξ by appending e to ξ as the last event w.r.t. ar.

Corrector Assumption. If OpSpec allows conditional read-writes, then we assume that for every domain D, $W \subseteq D$, $x \in \mathsf{Objs}$ s.t. $x \in W$ if $W \neq \emptyset$, and abstract execution ξ, there exists

(1) a conditional read-write e with $\mathsf{obj}(e) = D$ which is not contained in ξ, and
(2) a partial mapping $a : D \setminus \{x\} \rightarrow \mathsf{Events}$ called *execution-corrector* for event e in an abstract execution $\xi \oplus e$.

We define execution-correctors as follows.

Definition 7.13. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification,* ξ *an abstract execution,* e *a conditional read-write event from* ξ *with* $\mathsf{obj}(e) = D$*,* $W \subseteq D$ *a set of objects, and* $x \in D$ *an object s.t.* $x \in W$ *if* $W \neq \emptyset$*. Also, let* $<$ *be a fixed total order on the set of objects. An* execution-corrector *for* (e, W, x, ξ) *is a partial mapping* $a : D \setminus \{x\} \rightarrow \mathsf{Events}$ *such that if*

$$\xi' = \xi \overset{\mathsf{seq}(a)}{\vee} e \text{ and } \xi' \upharpoonright y = (\xi \overset{\mathsf{seq}(a \upharpoonright y)}{\vee} e) \setminus \{e\} \text{ where } a \upharpoonright y = a \upharpoonright_{\{z \in \mathsf{Dom}(a) \mid z \leq y\}},$$

then the following hold:

(1) *for every* $y \in D \setminus \{x\}$*, if* $a(y)$ *is defined and the correction up to* $a(y)$ *is valid w.r.t.* Spec*, then* $a(y)$ *writes only* y *in the correction: if* $a(y) \downarrow$ *and* $\xi' \upharpoonright y$ *is valid w.r.t.* Spec*, then for every object* $z \in \mathsf{Objs}$*,* $\mathsf{wspec}(a(y))(z, [\xi' \upharpoonright y, \mathsf{CMod}]) \downarrow$ *iff* $z = y$*, and*

(2) *for every* $y \in D$*, if the correction is valid w.r.t.* Spec*, then* e *reads* y *and additionally,* e *writes* y *only if* $y \in W$*, i.e.,* $\mathsf{rspec}(e)(y, [\xi', \mathsf{CMod}]) \neq \emptyset$ *and* $\mathsf{wspec}(e)(y, [\xi', \mathsf{CMod}]) \downarrow$ *iff* $y \in W$*.*

Example 7.14. *We illustrate execution-correctors for the storage specification presented in Example 7.12, with* `InsAbs` *and* `DelPre` *as operations and* `PC` *as consistency model.*

Let ξ *be an abstract execution,* e *a* $\mathtt{DelPre}(D)$ *event from* ξ*,* $W \subseteq D$ *a non-empty set of objects and* $x \in W$*. For every object* y*, let* w_y *be the last event from the "read" context of* e *w.r.t.* `PC` *which writes* y *(by read context we mean the set of events selected by* rspec *from the context). In the following we assume that* w_x *is an insert event. Note that if* w_x *is a delete event, then there exists no execution-corrector for* e *(intuitively, the correction concerns objects different from* x*, and* $\mathtt{DelPre}(D)$ *will not delete an object which is already deleted).*

An execution-corrector for (e, W, x, ξ) *is the mapping* $a : D \setminus \{x\} \rightarrow \mathsf{Events}$ *defined below. The mapping* a *observes the update on* y *made by* w_y*, and overwrites it when necessary. Thus, when* e *reads* y*,* y *is present iff* $y \in W$*.*

$$a(y) = \begin{cases} \mathtt{InsAbs}(\{y\}, v) & \text{if } y \in W \text{ and } \mathsf{w}_y \text{ deletes } y \text{ in } \xi \\ \mathtt{DelPre}(\{y\}) & \text{if } y \notin W \text{ and } \mathsf{w}_y \text{ inserts } y \text{ in } \xi \\ \text{undefined} & \text{otherwise} \end{cases} \tag{21}$$

Observe that requiring that a is defined for all objects in D is too strict: if the read specification has a layer-bound of 1 and the events read a single object (as faacas), any correction will change the entire context read by e.

8 The Arbitration-Free Consistency Theorem

We now present our main result in its most general form, which extends Theorem 6.3.

Theorem 8.1 (Arbitration-Free Consistency (AFC)). *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. The following statements are equivalent:*

(1) CMod *is arbitration-free w.r.t.* OpSpec*,*

(2) *there exists an available* OpSpec*-implementation.*

The proof of (1) ⇒ (2) is very similar to that in Theorem 6.3 (see Section 6.1). The only difference is replacing Lemma 6.5 with Lemma 7.11 where we use the maximally-layered assumption of read specifications. For the reverse, we follow the reasoning explained in the beginning of Section 6.2 to reduce to consistency models in normal form. Lemma 8.2 extends the arguments in Lemma 6.7 to generalized storage specifications.

Lemma 8.2. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* i*,* $0 \leq i \leq \mathsf{len}(\mathsf{v})$*,* $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* $(\mathsf{CMod}, \mathsf{OpSpec})$*-implementation.*

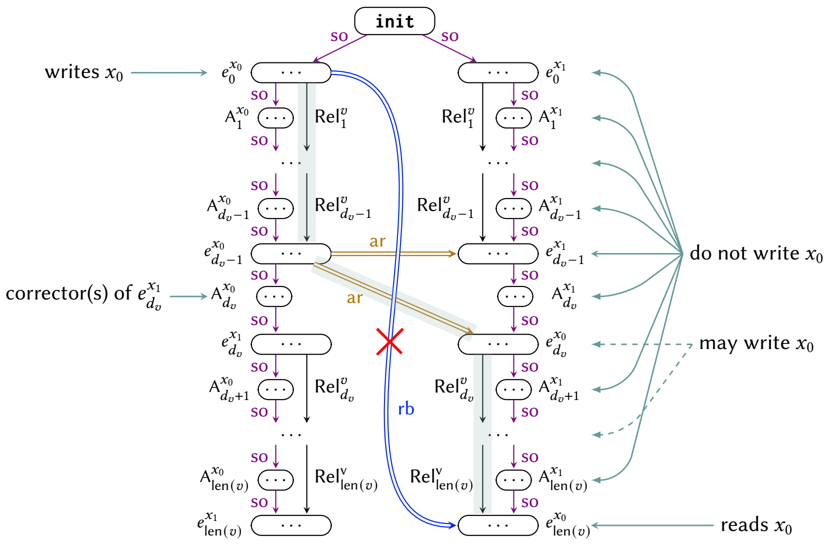


Fig. 9. Abstract execution of a trace without receive actions for the visibility formula v. $\mathsf{A}^{x_l}_i$ represents a sequence of events $a^{x_l}_i(y), y \in \mathsf{obj}(e^{x_l}_i)$ associated to an execution-corrector. The auxiliary events in $\mathsf{A}^{x_l}_i$ allow that, if $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$, $\mathsf{wrCons}^\mathsf{v}_x(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds, and thus $v_{x_0}(e^{x_0}_0, e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds as well.

Proof Sketch. As in Lemma 6.7, we assume by contradiction that there is an available implementation I_E of Spec. We use the visibility formula v to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no `receive` action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

Let d_v be the largest index i s.t. $\mathsf{Rel}^\mathsf{v}_i = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to $\varepsilon_{d_\mathsf{v}}$ which is not arbitration-free, and the suffix from $\varepsilon_{d_\mathsf{v}}$ up to $\varepsilon_{\mathsf{len}(\mathsf{v})}$, the arbitration-free part.

The program P that we construct uses two replicas r_0, r_1, two objects x_0, x_1 and a collection of events $e^{x_l}_i, 0 \le i \le \mathsf{len}(\mathsf{v}), l \in \{0, 1\}$. The events are used to "encode" two instances of v_{x_0} and v_{x_1}. Replica r_l executes first events $e^{x_l}_i$ with $i < d_\mathsf{v}$ and then, events $e^{x_{1-l}}_i$ with $i \ge d_\mathsf{v}$ – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. For every l, the event $e^{x_l}_{\mathsf{len}(\mathsf{v})}$ reads x_{1-l}.

For ensuring that $\mathsf{v}_x(e^{x_l}_0, \ldots e^{x_l}_n)$ holds in an induced abstract execution of a trace without `receive` actions, we require that if $\mathsf{Rel}^\mathsf{v}_i = \mathsf{wr}$, then $e^{x_l}_{i-1}$ is a write event and $e^{x_l}_i$ is a read event. For ensuring that $\mathsf{wrCons}^\mathsf{v}_x(e_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ holds in such an abstract execution, for each set $E \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ s.t. $\mathsf{conflict}(E)$ occurs in v, we consider a distinct object y_E, which is also distinct from x_0 and x_1. These objects represent each conflict in v in a distinct manner. Then, we require that events $e^{x_l}_i$ write to object y_E iff $\varepsilon_i \in E$ and to object x_l iff ε_i belongs to the set E_x s.t. $\mathsf{conflict}_x(E_x)$ occurs in v (since v is conflict-maximal, there is only one occurrence of a $\mathsf{conflict}_x$ predicate). In the case $e^{x_l}_i$ is a conditional read-write, we add a set of events $A^{x_l}_i$ that form an execution-corrector so $\mathsf{conflict}_x(e^{x_l}_0, \ldots e^{x_l}_{\mathsf{len}(\mathsf{v})})$ holds in an abstract execution of a trace without `receive` actions. These additional events do not write on objects x_0 or x_1.

Figure 9 exhibits a diagram of the abstract execution of the program.

The rest of the proof, which proceeds as follows, is a generalization of the proof of Lemma 6.7 which takes into considerations the assumptions we make about storage specifications:

(1) There exists a finite trace t of $P \mathbin{||} I_E$ that contains no receive action (Lemma C.5).
(2) The trace t induces a history $h_{\mathsf{v}} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_{\mathsf{v}} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$. As I_E is valid w.r.t. Spec, ξ_{v} is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_{v} read the latests value w.r.t. so for any object. In particular, we deduce that ξ_v is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Corollary D.5).
(3) Since ar is a total order, either $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$ or $(e^{x_1}_{d_{\mathsf{v}}-1}, e^{x_0}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$. By Proposition D.6, we deduce that $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(\mathsf{v})}, [\xi_{\mathsf{v}}, \mathsf{CMod}])$. The proof is explained in Figure 9: if $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$, then all events $e^{x_0}_i$ form a path in such way that $\mathsf{v}_{x_0}(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds in ξ_{v}. If some event $e^{x_l}_i$ is a conditional read-write event, the predicate $\mathsf{conflict}_x(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds in ξ_{v} thanks to the corrector events $A^{x_l}_i$.
(4) As $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_{\mathsf{v}}, \mathsf{CMod}])$ but $(e^{x_0}_0, e^{x_0}_{\mathsf{len}(\mathsf{v})}) \notin \mathsf{rb}$ (no message is received), we deduce in Proposition B.16that OpSpec is layered w.r.t. ar. By contrapositive, if OpSpec would be layered w.r.t. rb, as $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(\mathsf{v})}, [\xi_{\mathsf{v}}, \mathsf{CMod}])$, there would exist an event e s.t. $(e^{x_0}_0, e) \in \mathsf{rb}$ and $e \in \mathsf{rspec}(e^{x_0}_{\mathsf{len}(\mathsf{v})})(x_0, [\xi_{\mathsf{v}}, \mathsf{CMod}])$. However, as $\mathsf{rb} = \mathsf{so}$, $\mathtt{rep}(e^{x_0}_0) = \mathtt{rep}(e) = \mathtt{rep}(e^{x_0}_{\mathsf{len}(\mathsf{v})})$ which is false because $\mathtt{rep}(e^{x_0}_0) = r_0$ and $\mathtt{rep}(e^{x_0}_{\mathsf{len}(\mathsf{v})}) = r_1$.
(5) Since rspec is maximally layered, we can show that the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v (Proposition B.17). Observe that an event writes x_0 only if it is **init** or is an event $e^{x_l}_i$ s.t. $\varepsilon_i \in E_x$ and $l = 0$. Any such index i corresponds to a suffix of v. By causal suffix closure, for any arbitration-free suffix v' of v there is a visibility formula that subsumes v' in $\mathsf{nCMod}_{\mathsf{OpSpec}}$. As d_{v} is the maximum index for which $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}$, the number of events writing x_0 in replica r_1 distinct from **init** coincide with the number of arbitration-free suffixes of v. Hence, as rspec is layered w.r.t. ar, if its layer bound would be greater than the number of arbitration-free suffixes, $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ would necessarily read x_0 from **init** (other events writing x_0 are in replica r_0 and $e_{\mathsf{len}(\mathsf{v})}$ only reads from events in r_1). However, as rspec is maximally-layered and $e^{x_0}_0$ succeeds **init** w.r.t. ar and rb^+, we would conclude that $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ would also read x_0 from $e^{x_0}_0$. However, this is impossible as $\mathsf{wr} \subseteq \mathsf{rb} = \mathsf{so}$ but $e^{x_0}_0$ is in replica r_0 and $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ is in replica r_1.
(6) Lastly, we show in Proposition B.18 that if the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v, then v is vacuous w.r.t. OpSpec, which contradicts the fact that v is a visibility formula from the normal form $\mathsf{nCMod}_{\mathsf{OpSpec}}$. □

Corollary 8.3 is an immediate consequence of Theorem 8.1 and Lemma 6.4.

Corollary 8.3. *Let* OpSpec *be an operation specification. The strongest consistency model* CMod *for which* (CMod, OpSpec) *admits an available implementation is* $\mathtt{CC}$.

9 Related Work and Discussion

The CAP conjecture [13] claims that a distributed key-value store cannot be both consistent, available and tolerate partitions. The proof of the *CAP theorem* [18], uses a so-called *split brain* behavior, where two sets of replicas are isolated from each other, and a *get* (read) operation misses the result of an *earlier set* (write) operation (which completes before the get starts). We remark that our proof in section 2 actually extends the proof of the CAP theorem so it holds without the real -time requirement used in the original proof [18].

As pointed by some critiques of the CAP theorem (e.g., [21]), the proof equates consistency with atomicity of read / write variables. Moreover, network partitioning is a stand-in for end-to-end delays in geo-distributed systems. The PACELC (*if Partition then Availability or Consistency, Else Latency or Consistency*) theorem [1] (see [19]) captures these observations; its proof extends results proved for *sequential consistency* [8, 24]. These results are also proved for read / write variables, capturing key-value stores. In the executions we construct, messages between replicas are delayed, in a way that corresponds to split brain behavior, and our emphasis is on constructing the right interaction sequences. We believe this behavior can be used to extend the AFC theorem so it talks about latency, rather than availability, for the same interaction sequences.

The CALM (*consistency as logical monotonicity*) conjecture [20] relates monotonicity of queries to lack of coordination. Informally, it states that a query has a coordination-free execution strategy if and only if it is monotonic. In order to make this statement more concrete, it is necessary do define what coordination freedom means. In their proof of the CALM theorem, Ameloot et al. [6] equate coordination-freedom with the ability of clients to produce an output even when there is no communication between replicas. The proof relies on a split brain behavior, somewhat similar to the one used in the CAP theorem [18]. Extensions of this theorem [5] equip replicas with knowledge of the data distribution. The CALM theorem is motivated, in part, by Bloom [4], a programming language that encourages order-insensitive programming. The applications they present are to key-value stores and to a shopping cart, essentially, a counter. Later work extends the CALM approach to a programming environment for composing small *lattices* [4], and relates it to CRDTs [22].

One key challenge in deriving our result is considering abstract, generic consistency models, while prior work considers specific models. The other challenge is to allow their composition with abstract, generic shared objects, while prior work mostly consider key-value stores. On the possibility side, this is facilitated by the relating arbitration-freeness to causality; the necessity side relies on finding carefully-designed client interactions that “stress” dependencies between replicas.

Defining available implementations for causal consistency has been considered in several works [9, 10, 25, 26]. The work of Attiya et al. [7] and Mahajan et al. [27] show that, in the case of multi-value registers, consistency models stronger than causal consistency cannot support available implementations. In [7] the condition is *observable causal consistency* (OCC) whereas in [27] the condition is *real-time causal consistency* (RTC). The definition of both OCC and RTC are specific to multi-value registers, and the impossibility result depends on several restrictions that we do not consider. Both papers make some (nontrivial, but different) assumptions about the implementations. Furthermore, both of them do not truly prove a tight result: while both [7, 27] prove the positive result for CC, in [7], the impossibility is proved for OCC, and in [27] it is for RTC (both stronger than CC). Besides handling a more general class of operations, the AFC theorem is a strengthening of their results, as it applies to causal consistency and is therefore tight.

Our specification framework builds on previous work [11, 14, 15, 17]. Similarly to Burckhardt et al. [14, 15], storage system specifications decouple consistency from the object semantics. We re-use the same ideas of defining consistency using visibility formulas, contexts, and an arbitration relation. Our object semantics is split into several semantical functions (rspec, extract, and wspec) in order to be more general (modeling transactions), and be able to express “normal” constrains. The extension to transaction isolation levels is similar to Cerone et al. [17] and Biswas and Enea [11].

The works of [11, 12] study the complexity of checking consistency under different scenarios. There is no apparent relation between the complexity of checking consistency and the existence of available implementations: the AFC theorem shows that Read Committed admits available implementations but Sequential Consistency does not whereas [12] shows that checking consistency of an SQL history under Read Committed (equivalent to Return-Value) or Sequential Consistency is NP-complete (inclusion in NP is trivial for any model).

Acknowledgements

We thank the anonymous reviewers for their feedback. Hagit Attiya is supported by the Israel Science Foundation (22/1425 and 25/1849). Constantin Enea is partially supported by the French National Research Agency (project "CENTEANES").

References

[1] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer* 45, 2 (2012), 37–42. https://doi.org/10.1109/MC.2012.33

[2] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation.

[3] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[4] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

[5] Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. 2015. Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture. *ACM Trans. Database Syst.* 40, 4, Article 21 (Dec. 2015), 45 pages. https://doi.org/10.1145/2809784

[6] Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. 2013. Relational transducers for declarative networking. *J. ACM* 60, 2, Article 15 (May 2013), 38 pages. https://doi.org/10.1145/2450142.2450151

[7] Hagit Attiya, Faith Ellen, and Adam Morrison. 2017. Limitations of Highly-Available Eventually-Consistent Data Stores. *IEEE Trans. Parallel Distrib. Syst.* 28, 1 (Jan. 2017), 141–155. https://doi.org/10.1109/TPDS.2016.2556669

[8] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (1994), 91–122. https://doi.org/10.1145/176575.176576

[9] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*. http://doi.acm.org/10.1145/2391229.2391251

[10] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 761–772. http://doi.acm.org/10.1145/2463676.2465279

[11] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. https://doi.org/10.1145/3360591

[12] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2025. On the Complexity of Checking Mixed Isolation Levels for SQL Transactions. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 15934)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, 315–337. https://doi.org/10.1007/978-3-031-98685-7_15

[13] Eric A. Brewer. 2000. Towards robust distributed systems (Invited Talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA. https://doi.org/10.1145/343477.343502

[14] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (2014), 1–150. https://doi.org/10.1561/2500000011

[15] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *41st Symposium on Principles of Programming Languages, POPL*. ACM, 271–284. https://doi.org/10.1145/2535838.2535848

[16] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 568–590. https://doi.org/10.4230/LIPICS.ECOOP.2015.568

[17] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory, CONCUR*. 58–71. https://doi.org/10.4230/LIPICS.CONCUR.2015.58

[18] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[19] Wojciech M. Golab. 2018. Proving PACELC. *SIGACT News* 49, 1 (2018), 73–81. https://doi.org/10.1145/3197406.3197420

[20] Joseph M. Hellerstein. 2010. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.* 39, 1 (Sept. 2010), 5–19. https://doi.org/10.1145/1860702.1860704

[21] Martin Kleppmann. 2015. A Critique of the CAP Theorem. *CoRR* abs/1509.05393 (2015). arXiv:1509.05393 http://arxiv.org/abs/1509.05393

[22] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. 2022. Keep CALM and CRDT On. *Proc. VLDB Endow.* 16, 4 (2022), 856–863. https://doi.org/10.14778/3574245.3574268

[23] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[24] Richard J Lipton and Jonathan S Sandberg. 1988. *PRAM: A scalable shared memory.* Technical Report TR-180-88. Princeton University, Department of Computer Science.

[25] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 401–416. http://doi.acm.org/10.1145/2043556.2043593

[26] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 313–328. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[27] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. 2011. Consistency, availability, and convergence. *University of Texas at Austin Tech Report* 11 (2011), 158.

[28] Microsoft. 2022. *Consistency Levels in Azure Cosmos DB.* https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[29] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems SSS*, Vol. 6976. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[30] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 385–400. https://doi.org/10.1145/2043556.2043592

A Examples of Operation Specifications

We present here several well-know operation specifications.

A.1 Key-Value Store with Fetch-And-Add and Compare-And-Swap Operations

The Key-Value Store with Fetch-And-Add and Compare-And-Swap (faacas) is an operation specification with four operations, $\mathtt{PUT}(x, v)$, that puts value v to object x, $\mathtt{GET}(x)$ that reads object x, $\mathtt{FAA}(x, v)$ that reads the value v' of object x and writes $v' + v$, and $\mathtt{CAS}(x, v, v')$, that reads x and writes v' iff the value read is v.

The following equations, corresponding to Equations (11), (14) and (17), describe the operation specification of faacas.

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \{\mathtt{GET}(x), \mathtt{FAA}(x,v), \mathtt{CAS}(x,v',v'')\} \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{22}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} v & \text{if } r \in \{\mathtt{GET}(x), \mathtt{FAA}(x,v), \mathtt{CAS}(x,v,v')\} \text{ and } R = \{(w,v)\} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{23}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathtt{PUT}(x,v') \\ v + v' & \text{if } w = \mathtt{FAA}(x,v') \\ v'' & \text{if } w = \mathtt{CAS}(x,v',v'') \wedge v = v' \\ \text{undefined} & \text{otherwise} \end{cases} \tag{24}$$

The faacas is maximally layered w.r.t. ar, with 1 as its layer bound. As $\mathtt{CAS}$ is a single-object conditional read-write operation, it trivially allows execution-correctors.

A.2 Key-Value Multi-Value Store

The Key-Value Multi-Value Store (k-mv) [7, 15] is an operation specification with two operations, $\mathtt{GET}(x)$, reading multiple concurrent values on a single object x, and $\mathtt{PUT}(x, v)$, writing on a single object x the value v.

The following equations, corresponding to Equations (12), (14) and (17), describe the operation specification of k-mv.

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{rb}} E\} & \text{if } r = \mathtt{GET}(x) \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{25}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} \{v \mid (\_, v) \in R\} & \text{if } r = \mathtt{GET}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{26}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} v & \text{if } w = \mathtt{PUT}(x,v) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{27}$$

The k-v is maximally layered w.r.t. rb^+, with 1 as its layer bound.

A.3 Distributed Counter

The distributed counter (counter) [15] is an operation specification with two operations, $\mathtt{inc}(x)$, incrementing the value of x by 1, and $\mathtt{rd}(x)$, reading the amount of increments of x.

The following equations, corresponding to Equations (13), (16) and (19), describe the operation specification of counter.

$$\mathsf{rspec}(r)(x,c) = \begin{cases} E & \text{if } r = \mathtt{rd}(x) \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{28}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} |R| - 1 & \text{if } r = \mathtt{rd}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{29}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} 1 & \text{if } w = \mathtt{inc}(x) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{30}$$

The counter is maximally layered w.r.t. ar, with ∞ as its layer bound.

A.4 Insert/Delete Last-Write-Wins

The Insert/Delete Last-Write-Wins (ins/del) is an operation specification with two multi-object operations. $\mathtt{InsAbs}(X, v)$ checks for every object $x \in X$ if it is present, and inserts it with value v if not, and $\mathtt{DelPre}(X)$ deletes every object $x \in X$ as long as it was already present.

Its operation specification is described as follows:

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \left\{ \mathtt{InsAbs}(X,v), \mathtt{DelPre}(X) \right\}, x \in X \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{31}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} v & \text{if } w \in \{\mathtt{InsAbs}(X,\_), \mathtt{DelPre}(X)\}, x \in X \text{ and } R = \{(\_, v)\} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{32}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathtt{InsAbs}(X,v') \land v = \dagger \\ \dagger & \text{if } w = \mathtt{DelPre}(X) \land v \neq \dagger \\ \text{undefined} & \text{otherwise} \end{cases} \tag{33}$$

where $\dagger$ is a special value representing absence. We assume that $\mathtt{InsAbs}(X, \dagger)$ is not defined.

The ins/del is maximally layered w.r.t. ar, with 1 as its layer bound. ins/del allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

Let be v the value that event e reads in $\xi \oplus e$. If $v = \dagger$, we select $e = \mathtt{InsAbs}(D, \_)$ while otherwise, $e = \mathtt{DelPre}(D)$. The mapping a below is an execution-corrector for (e, W, x, ξ):

$$a(y) = \begin{cases} \mathtt{InsAbs}(\{y\}, v') & \text{if } y \in W \land v_y = \dagger \neq v, \text{ or } y \notin W \land v_y = \dagger = v \\ \mathtt{DelPre}(\{y\}) & \text{if } y \in W \land v_y \neq \dagger = v, \text{ or } y \notin W \land v_y \neq \dagger \neq v \\ \text{undefined} & \text{otherwise} \end{cases} \tag{34}$$

where $v_y = \mathsf{wspec}(e_p)(y, [\xi, \mathsf{CMod}])$ and e_p is the maximal event w.r.t. so on the same replica as e.

A.5 Non-Transactional SQL with Last-Writer-Wins Store

The Non-Transactional SQL with Last-Writer-Wins Store (simple-SQL) is an operation specification modelling SQL-like databases [2]. Each object represents a row identifier and the set of values is defined abstractly as Rows. Rows contain a special value denoted $\dagger$, different from $\perp$, indicating that the row is deleted.

This operation specification employs four operations: INSERT, SELECT, UPSERT and DELETE. Each operation has a finite set of objects D as domain. INSERT(R) inserts in the database each row r on an object $d \in D$ using the mapping $\mathsf{R} : D \longrightarrow \mathsf{Rows}$. SELECT(p) selects the rows on the storage satisfying the predicate $\mathsf{p} : D \times \mathsf{Rows} \longrightarrow \{\mathsf{false}, \mathsf{true}\}$. UPSERT(p, U) updates the rows that satisfy p using the mapping $\mathsf{U} : D \times \mathsf{Rows} \longrightarrow \mathsf{Rows}$, inserting them if they are absent. Finally, DELETE(p), deletes the objects satisfying the predicate (i.e. replaces its row by $\dagger$). We assume that in for any predicate p and object x, $\mathsf{p}(x, \dagger) = \mathsf{false}$.

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \{\mathtt{SELECT(p)}, \mathtt{UPSERT(p,\ U)}, \mathtt{DELETE(p)}\} \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{35}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} v & \text{if } r \in \{\mathtt{SELECT(p)}, \mathtt{UPSERT(p,\ U)}, \mathtt{DELETE(p)}\}, \\ & \quad R = \{(w,v)\} \text{ and } \mathsf{p}_x(v) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{36}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} \mathsf{R}(x) & \text{if } w = \mathtt{INSERT(R)} \\ \mathsf{U}_x(v) & \text{if } w = \mathtt{UPSERT(p,\ U)} \\ \dagger & \text{if } w = \mathtt{DELETE(p)} \land v \notin \{\bot, \dagger\} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{37}$$

The simple-SQL is maximally layered w.r.t. ar, with 1 as its layer bound. simple-SQL allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

Let be v the value that event e reads in $\xi \oplus e$. We select the event $e = \mathtt{UPSERT}(\mathsf{p}_{D,W},\ \mathsf{U}_D)$, where $\mathsf{p}_{D,W}$ and U_D are defined below.

$$\mathsf{p}_{D,W}(d,r) = \begin{cases} \text{true} & \text{if } d \in W \\ \text{false} & \text{if } d \in D \setminus W \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\mathsf{U}_D(d,r) = \begin{cases} r & \text{if } d \in D \\ \text{undefined} & \text{otherwise} \end{cases}$$

For such event, we define the execution-corrector $a : D \setminus \{x\} \longrightarrow$ Events as the totally-undefined mapping, i.e. the function that no object $y \in D$ is associated with some event.

A.6 Transactional SQL Multi-Value Store

The Transactional SQL Multi-Value Store (SQL-mvr) is an operation specification modelling SQL-like databases using *transactions*. Each object represents a row identifier and the set of values, Rows, is defined as in Appendix A.5.

Transactions are blocks of simple instructions that are executed sequentially. Transactions start its execution by selecting a *snapshot* of the database (i.e. a mapping associating each object a constant value) from which operations can read. Each instruction may execute a writing operation, but its effect it is only viewed internally. After their completion, the writing effects of the transaction can be seen by other transactions; giving the impression of atomicity.

We model the store with the aid of a unique operation, TRANSACTION(body) that reads the snapshot of the database and then executes the instructions declared in C. C is defined as a sequence of five type of operations: INSERT, SELECT, UPDATE and DELETE. Each operation has a finite set of objects D as domain. INSERT(R) inserts in the database each row r on an object $d \in D$ using the mapping $\mathsf{R} : D \longrightarrow \text{Rows}$. SELECT(p) selects the rows on the storage satisfying the predicate $\mathsf{p} : D \times \text{Rows} \longrightarrow \{\text{false}, \text{true}\}$. UPDATE(p, U) updates the rows that satisfy p using the mapping $\mathsf{U} : D \times \text{Rows} \longrightarrow \text{Rows}$. Finally, DELETE(p), deletes the objects satisfying the predicate (i.e. replaces its row by $\dagger$). abort represents states declared by the user where the transaction should not execute any more instructions and any declared write should be aborted. We assume that in for any predicate p and object x, $\mathsf{p}(x, \dagger) = \text{false}$.

We model snapshots as mappings Objs $\longrightarrow$ Vals. Unlike in Appendix A.5, SQL-mvr requires that local effects of SQL-like instructions are only seen internally, during the execution of the transaction. Such effects are modelled in Equation (38) as a recursive function that simulates the transaction execution w.r.t. a concrete object. The function exe executes one instruction at a time, and it stops whenever all instructions are executed, indicating that the execution was correct, or halting it midway in case some abortion occurred (modelled with the constant value $\bot$).

$$\mathsf{exe}_x(\mathsf{body},\sigma) = \begin{cases} \mathsf{exe}_x(\mathsf{body}',\sigma') & \text{if body} = e;\mathsf{body}', \sigma' = \mathsf{exI}_x(e,\sigma) \text{ and } \sigma' \neq (\bot, \mathsf{false}) \\ \sigma & \text{if body} = \emptyset \\ (\bot, \mathsf{false}) & \text{otherwise} \end{cases} \tag{38}$$

The behavior of each instruction is modelled in Equation (39), updating the snapshot in object x in a similar way as wspec does in Appendix A.5, and indicating if the event e indeed wrote object x.

$$\mathsf{exI}_x(e,(\sigma,\mathrm{w})) = \begin{cases} (\sigma,\mathrm{w}) & \text{if } e = \mathtt{SELECT(p)} \\ (\sigma,\mathrm{w}) & \text{if } e = \mathtt{DELETE(p)} \wedge \neg \mathrm{p}_x(\sigma) \\ (\dagger, \mathsf{true}) & \text{if } e = \mathtt{DELETE(p)} \wedge \mathrm{p}_x(\sigma) \\ (\sigma,\mathrm{w}) & \text{if } e = \mathtt{UPDATE(p,\ U)} \text{ and either } \neg\mathrm{p}_x(\sigma) \text{ or } U_x(\sigma)\uparrow \\ (\mathrm{U}_x(\sigma), \mathsf{true}) & \text{if } e = \mathtt{UPDATE(p,\ U)}, \mathrm{p}_x(\sigma) \wedge U_x(\sigma)\uparrow \\ (\sigma,\mathrm{w}) & \text{if } e = \mathtt{INSERT(R)} \wedge \mathrm{R}(x)\uparrow \\ (\mathrm{R}(x), \mathsf{true}) & \text{if } e = \mathtt{INSERT(R)} \wedge \mathrm{R}(x)\downarrow \\ (\bot, \mathsf{false}) & \text{if } e = \mathtt{abort} \end{cases} \tag{39}$$

The operation specifications of SQL-mvr are an adaptation of those of k-mv:

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{rb}} E\} & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}) \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{40}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} \sigma' & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}), \sigma = \{(v, \mathsf{false}) \mid (w,v) \in R\} \\ & \quad \text{and } \sigma' = \mathsf{exe}_x(\mathsf{body},\sigma) \end{cases} \tag{41}$$

$$\mathsf{wspec}(w)(x,\sigma) = \begin{cases} v & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}),\ \text{and } \sigma = (v, \mathsf{true}) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{42}$$

The SQL-mvr operation specification is maximally layered w.r.t. rb^+, with 1 as its layer bound. SQL-mvr allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

We define $e = \mathtt{TRANSACTION(SELECT(p}_D\mathtt{);\ INSERT(R}_W\mathtt{))}$, where p_W and U_D are defined below.

$$\mathrm{p}_D(d,r) = \begin{cases} \mathsf{true} & \text{if } d \in D \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\mathrm{R}_W(d) = \begin{cases} \_ & \text{if } d \in D \\ \text{undefined} & \text{otherwise} \end{cases}$$

where _ indicates some arbitrary unspecified value.

For such event, we define the execution-corrector $a : D \setminus \{x\} \longrightarrow \mathsf{Events}$ as the totally-undefined mapping, i.e. the function that no object $y \in D$ is associated with some event.

B Normal Form of a Consistency Model w.r.t. an Operation Specification

In this section, we prove the existence of a consistency model in normal form equivalent to a given one (Theorem B.1), and we show as well that arbitration-freeness is well-defined (Theorem B.9), i.e. that either all its normal forms are arbitration-free or none.

For compare consistency models when restricted to an operation specification OpSpec, we introduce the notion of OpSpec-equivalence. Two consistency models CMod_1, CMod_2 are OpSpec-*equivalent*, denoted $\mathsf{CMod}_1 \equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$, if for every abstract execution of OpSpec, ξ, ξ is valid w.r.t. $(\mathsf{CMod}_1, \mathsf{OpSpec})$ iff ξ is valid w.r.t. $(\mathsf{CMod}_2, \mathsf{OpSpec})$. In particular, if CMod_1 and CMod_2 are equivalent, they are also OpSpec-equivalent. The converse is not true: vacuous visibility formulas under an operation specification OpSpec may not be vacuous for every possible operation specification.

B.1 Existence of a Normal Form of a Consistency Model

Theorem B.1 states the existence of a normal form of a consistency model w.r.t. OpSpec.

Theorem B.1. *Let* OpSpec *be an operation specification. For every consistency model* CMod*, there exists a consistency model that is in normal form w.r.t.* OpSpec *and that is* OpSpec*-equivalent to* CMod*.*

The proof of such result is divided in three parts, proving the existence of a consistency model with only simple visibility formulas (Lemma B.4), proving that such model can be refined for removing vacuous visibility formulas (Lemma B.7) and finally, showing that conflict-maximality can be assumed without loss of generality (Lemma B.8).

Monotonicity

Maximally-layered operation specifications are *monotonic*. Intuitively, an operation specification is *monotonic* if (1) the values that are not read under a consistency model CMod_1 should be also not read under a stronger model CMod_2, and (2) whenever some values are read under a consistency model CMod_1 but not under a stronger one CMod_2, some other values must be read under CMod_2 which were not visible under CMod_1.

Definition B.2. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification.* OpSpec *is called* monotonic *if for every pair of consistency models* $\mathsf{CMod}_1, \mathsf{CMod}_2$, $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$*, abstract execution* ξ*, event* $r \in \xi$*, and object* x *the following hold:*

(1) $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \cup (\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]))$.
(2) *if* $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$*, then* $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$

Lemma B.3. *A maximally-layered operation specification is monotonic.*

Proof. Let OpSpec be a maximally-layered operation specification, $\mathsf{CMod}_1, \mathsf{CMod}_2$ be two consistency models s.t. $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, ξ be an abstract execution, r be an event in ξ and x be an object. Observe that by the unconditional read property of OpSpec (Property 2 of Definition 7.4), we can assume w.l.o.g. that r is a read event.

On one hand, we observe that if the layer bound of OpSpec is ∞, OpSpec is trivially monotonic: as r is a read event and the layer bound of OpSpec is ∞, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ and $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. Using the fact that $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$, is easy to see that Properties 1 and 2 hold in this case.

On the other hand, if the layer bound of OpSpec, k, is finite, let R be the relation for which OpSpec is k-maximally layered. For proving Property 1 of Definition B.2, let us partition $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ in the three disjoint sets C_1, C_2 and C_3 described in Equation (43).

$$\begin{aligned} C_1 &\coloneqq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \\ C_2 &\coloneqq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \\ C_3 &\coloneqq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \end{aligned} \tag{43}$$

We note that by Property 1 of Definition 7.4, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. As $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, we deduce that $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$; so $\{C_1, C_2, C_3\}$ is indeed a partition of $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Observe that showing Property 1 of Definition B.2 is equivalent to show that $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq C_1 \cup C_3$. By Property 1 of Definition 7.4, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) = C_1 \cup C_2 \cup C_3$. We conclude the result by showing that $C_2 \cap \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \emptyset$.

For showing it, we observe that the layer of an event w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is less or equal than the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$: as $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$, every chain of events in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ containing w and ordered w.r.t. R belongs to $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Thus, as OpSpec is maximally layered, an event w in C_2 does not belong to $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$: if $w \in C_2$, its layer in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is greater than k; so it is also greater than k in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Hence, as OpSpec has k as layer bound, $w \notin \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$.

For proving Property 2, we observe that if there exists an event $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$, then the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ is greater than k. Let k' be the layer of w and let $\{e_i\}_{i=1}^{k'}$ be a chain of R of length k' s.t. $e_{k'} = w$. As the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is k and R is a partial order, there exists an event e_i, $1 \leq i \leq k$ s.t. $e_i \in C_3$. We observe that as the layer of w is k, the layer of event e_i is i. Hence, as rspec is k-maximally layered, we conclude that $e_i \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. □

Lemma 7.11 shows that for maximally-layered operation specifications, ensuring a strong consistency criteria is enough for ensuring a weaker one. The proof relies on the fact that maximally-layered operation specifications are monotonic (Lemma B.3).

Lemma 7.11. *Let* OpSpec *be a maximall-layered operation specification and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of consistency models such that* CMod_2 *is stronger than* CMod_1. *Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$.

Proof. Let $h = (E, \mathsf{so}, \mathsf{wr})$ be a history and let CMod_1 and CMod_2 be two consistency models s.t. $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$. Let also $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be an abstract execution that witness the validity of h w.r.t. $(\mathsf{CMod}_2, \mathsf{OpSpec})$. To prove that ξ also witnesses h's validity w.r.t. $(\mathsf{CMod}_1, \mathsf{OpSpec})$, by Definition 7.8, it suffices to prove that for every event $r \in h$ and object x, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$.

- $\underline{\mathsf{wr}_x^{-1}(r) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])}$: Let w be a write event in $\mathsf{wr}_x^{-1}(r)$. As $(w, r) \in \mathsf{wr}_x$, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. Moreover, as ξ witnesses h's validity w.r.t. CMod_2, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$. Hence, as $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \cap \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$, by Property 1 of Definition B.2, $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$.
- $\underline{\mathsf{wr}_x^{-1}(r) \supseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])}$: Let $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$ s.t. $w \notin \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$. By property 2 from Definition B.2, there exists $w' \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$ s.t. $w' \notin \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. However, as $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{wr}_x^{-1}(r) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$, this is impossible. Therefore, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{wr}_x^{-1}(r)$. □

An immediate consequence of Lemma 7.11 is the following result.

Lemma 6.5. *Let* OpSpec *be a basic operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of basic consistency models s.t.* CMod_2 *is weaker than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$.

Simple Form

For proving Theorem B.1, we first prove the existence of a consistency model in simple form (i.e. a consistency model with all its visibility formulas are simple) that is equivalent to CMod.

Lemma B.4. *For any consistency model* CMod*, there exists a consistency model in simple form that is equivalent to* CMod.

Intuitively, the proof of Lemma B.4 is as follows: we first unfold union and transitive closure operators, and then trim `id` and compositional operators to obtain a consistency model in simple form. As an intermediate step, we define the consistency model obtained after unfolding union and transitive closure operators. Such consistency model is the *almost simple form* of CMod, $\mathsf{almost}(\mathsf{CMod})$, and it is described as the union of the *almost simple form* of each of its visibility formulas, i.e. $\mathsf{almost}(\mathsf{CMod}) = \bigcup_{v \in \mathsf{CMod}} \mathsf{almost}(v)$. A visibility formula a belongs to the almost simple form of a visibility formula v, $a \in \mathsf{almost}(v)$ if (1) $\mathsf{len}(v) = \mathsf{len}(a)$ and (2) for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^a \in \sigma(\mathsf{Rel}_i^{\mathsf{v}})$; where $\sigma(\mathsf{Rel}i^v)$ is the set of relations described as follows:

$$\sigma(\mathsf{R}) = \begin{cases} \{\mathsf{R}\} & \text{if } \mathsf{R} = \mathtt{id}, \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \sigma(\mathsf{S}) \cup \sigma(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S} \cup \mathsf{T} \\ \sigma(\mathsf{S}); \sigma(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S}; \mathsf{T} \\ \bigcup_{k \in \mathbb{N} \land k \geq 1} \sigma(\mathsf{S})^k & \text{if } \mathsf{R} = \mathsf{S}^+ \end{cases} \tag{44}$$

where the composition of two sets of relations A, B is defined as $A; B ::= \{a; b \mid a \in A, b \in B\}$.

We prove that CMod and $\mathsf{almost}(\mathsf{CMod})$ are equivalent.

Proposition B.5. *For any consistency model* CMod*,* CMod *and* $\mathsf{almost}(\mathsf{CMod})$ *are equivalent.*

Proof. For proving the result, we show that for any abstract execution ξ, object x and event r, $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{almost}(\mathsf{CMod})])$. In particular, it suffices to prove that for every visibility formula $v \in \mathsf{CMod}$ and event w, $v_x(w, r)$ holds in ξ iff there exists a visibility formula $a \in \mathsf{almost}(v)$ s.t. $a_x(w, r)$ holds in ξ. Observe that for every $a \in \mathsf{almost}(v)$, $\mathsf{len}(v) = \mathsf{len}(a)$; so we reduce the proof to show that for every pair of events e, e', $(e, e') \in \mathsf{Rel}_i^{\mathsf{v}}$ iff there exists $\mathsf{R}' \in \sigma(\mathsf{Rel}_i^{\mathsf{v}})$ s.t. $(e, e') \in \mathsf{R}'$.

In the following, we prove that for every relation R over pair of events obtained by the grammar described in Equation (3), the following holds: $(e, e') \in \mathsf{R}$ iff there exists $\mathsf{R}' \in \sigma(R)$ s.t. $(e, e') \in \mathsf{R}'$. We show the result by induction on the depth of R[6]. The base case, when the depth of R is 0, refers to the case $\mathsf{R} = \mathtt{id}, \mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}$. In such case, the result immediately holds by the definition of $\sigma(\mathsf{R})$.

Let us assume that for any relation of depth at most n the result holds, and let us prove that for relations of depth $n + 1$. Three alternatives arise:

- If $\mathsf{R} = \mathsf{S} \cup \mathsf{T}$, $(e, e') \in \mathsf{R}$ if and only if $(e, e') \in \mathsf{S} \cup \mathsf{T}$. By induction hypothesis on both S and T, $(e, e') \in \mathsf{S} \cup \mathsf{T}$ iff there exists $\mathsf{R}' \in \sigma(\mathsf{S}) \cup \sigma(\mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$. Finally, by Equation (44), we conclude that there exists $\mathsf{R}' \in \sigma(\mathsf{S}) \cup \sigma(\mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$ if and only if there exists $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.
- If $\mathsf{R} = \mathsf{S}; \mathsf{T}$, $(e, e') \in \mathsf{R}$ if and only if $(e, e') \in \mathsf{S}; \mathsf{T}$. By the definition of composition, $(e, e') \in \mathsf{S}; \mathsf{T}$ iff there exists e'' s.t. $(e, e'') \in \mathsf{S}$ and $(e'', e') \in \mathsf{T}$. By induction hypothesis on both S

[6]By depth of R we mean the depth of the tree obtained by deriving R using Equation (3).

and T, there exists e'' s.t. $(e, e'') \in \mathsf{S}$ and $(e'', e') \in \mathsf{T}$ iff there exists e'' and relations $\mathsf{S}' \in \sigma(\mathsf{S}), \mathsf{T}' \in \sigma(\mathsf{T})$ e'' s.t. $(e, e'') \in \mathsf{S}'$ and $(e'', e') \in \mathsf{T}'$. By the definition of $\sigma(S); \sigma(T)$, we observe that there exists e'' and relations $\mathsf{S}' \in \sigma(\mathsf{S}), \mathsf{T}' \in \sigma(\mathsf{T})$ e'' s.t. $(e, e'') \in \mathsf{S}'$ and $(e'', e') \in \mathsf{T}'$ iff there exists relation $\mathsf{R}' \in \sigma(\mathsf{S}; \mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$. Finally, by Equation (44), we conclude that there exists relation $\mathsf{R}' \in \sigma(\mathsf{S}; \mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$ if and only if there exists $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.

- If $\mathsf{R} = \mathsf{S}^+$, $(e, e') \in \mathsf{R}$ if and only if there exists $k \in \mathbb{N}^+$ s.t. $(e, e') \in \mathsf{S}^k$. By the previous point, there exists $k \in \mathbb{N}^+$ s.t. $(e, e') \in \mathsf{S}^k$ if and only if there exists $k \in \mathbb{N}^+$ and relation $\mathsf{S}' \in \sigma(\mathsf{S})^k$ s.t. $(e, e') \in \mathsf{S}'$. Finally, by Equation (44), we conclude that there exists $k \in \mathbb{N}^+$ and relation $\mathsf{S}' \in \sigma(\mathsf{S})^k$ s.t. $(e, e') \in \sigma(\mathsf{S})^k$ if and only if there exists relation $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.

□

Obtaining a consistency model in simple form from a consistency model in almost simple form is straightforward: every visibility formula is transformed by splitting composed relations into simpler subrelations and omitting `id` by merging two existentially quantified events. Lemma B.4 formally describes such procedure.

Lemma B.4. *For any consistency model* CMod*, there exists a consistency model in simple form that is equivalent to* CMod.

Proof. We construct a consistency model, simple(CMod), that is in simple form and it is equivalent to CMod. The model is formally defined as follows:

$$\mathsf{simple}(\mathsf{CMod}) = \{\mathsf{simple}(a) \mid a \in \mathsf{almost}(\mathsf{CMod})\} \tag{45}$$

where simple(a) is the *simple visibility formula* of a.

The simple visibility formula of a visibility formula in almost form a is the visibility formula f obtained by supressing `id` and compositional operators. Formally, f is the visibility formula s.t. (1) $\mathsf{len}(f) = \sum_{i=1}^{\mathsf{len}(a)} \mathsf{count}(\mathsf{Rel}_i^a)$ and (2) for every $i, 1 \leq i \leq \mathsf{len}(f)$, $\mathsf{Rel}_i^f = \mathsf{rel}(\mathsf{Rel}_j^a, i - k_j)$; where j is the maximum index s.t. $k_j < i$ and $k_j = \sum_{l=1}^{j} \mathsf{count}(\mathsf{Rel}_l^a)$, and count and rel are the functions described in Equation (46) and Equation (47) respectively.

The function count counts the number of additional quantifiers the correspondant simple form requires:

$$\mathsf{count}(\mathsf{R}) = \begin{cases} 0 & \text{if } \mathsf{R} = \mathtt{id} \\ 1 & \text{if } \mathsf{R} = \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \mathsf{count}(\mathsf{S}) + \mathsf{count}(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S}; \mathsf{T} \end{cases} \tag{46}$$

Also, the function rel, given a relation using compositional operator and an index i, returns the i-th component:

$$\mathsf{rel}(\mathsf{R}, i) = \begin{cases} \mathsf{R} & \text{if } \mathsf{R} = \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \mathsf{rel}(\mathsf{S}, i) & \text{if } i \leq \mathsf{count}(\mathsf{S}) \\ \mathsf{rel}(\mathsf{T}, i - \mathsf{count}(\mathsf{S})) & \text{otherwise} \end{cases} \tag{47}$$

By construction, simple(CMod) is in simple form. Clearly, simple(CMod) is equivalent to almost(CMod). Then, thanks to Proposition B.5, we conclude that simple(CMod) is equivalent to CMod. □

Removing Vacuous Visibility Formulas

After proving the existence of a consistency model CMod in simple form equivalent to a given one, we show how to transform it for obtaining an equivalent consistency model CMod without vacuous visibility formulas (Lemma B.7). We say that any such consistency model is in *basic normal* *form*, extending Definition 6.1 to any consistency model whose visibility formulas are described using Equation (7).

The following result, key to prove Lemma B.7, it is a simple consequence of Definition B.2 and Lemma B.3.

Proposition B.6. *Let* OpSpec *be a maximally-layered operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be two consistency models s.t.* $\mathsf{CMod}_1 \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$ *but* $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$*. There exists an abstract execution* ξ *valid w.r.t.* CMod_1*, an object* x *and events* w, r *s.t.* $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$.

Proof. First of all, as $\mathsf{CMod}_1 \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$ but $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, by Lemma 7.11, there exists an abstract execution ξ valid w.r.t. CMod_1, an object x and an event r s.t. $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$. Thus, either $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \neq \emptyset$ or $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$.

On one hand, if $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \neq \emptyset$, by Property 1 of Definition B.2, then $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$. On the other hand, if $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$, by Property 2 of Definition B.2, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$. □

Lemma B.7. *Let* OpSpec *be an operation specification. For every consistency model* CMod *in simple form, there exists a* OpSpec*-equivalent consistency model,* $\mathsf{bnCMod}_{\mathsf{OpSpec}}$*, that is in basic normal form w.r.t.* OpSpec.

Proof. To prove the result, we construct a consistency model in basic normal form w.r.t. OpSpec, $\mathsf{bnCMod}_{\mathsf{OpSpec}}$, that is OpSpec-equivalent to CMod. Without loss of generality we can assume that CMod is ordered. Let α be an ordinal of cardinality $|\mathsf{CMod}|$. We denote by $v^i, 0 \leq i < \alpha$ to the i-th visibility formula in CMod[7].

We construct a sequence of nested consistency models $\mathsf{CMod}_k, 0 \leq k \leq \alpha$ s.t. (1) CMod_k is OpSpec-equivalent to CMod, (2) CMod_k is more succinct than CMod_i (i.e., for every $i < k$, $v^i \in \mathsf{CMod}_k$ iff $v^i \in \mathsf{CMod}_i$ and for every $i > k$, $v^i \in \mathsf{CMod}_k$), and (3) the first k visibility formulas of CMod_k are simple and non-vacuous w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$ (i.e., for every $i, 0 \leq i < k$, if $v^i \in \mathsf{CMod}_k$, then $\mathsf{CMod}_k \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$).

We construct such sequence using transfinite induction. The base case, $k = 0$, corresponds to $\mathsf{CMod}_0 = \mathsf{CMod}$, which trivially satisfies (1), (2) and (3). For the successor case, let us assume that the property holds for the consistency model CMod_k, and let us prove it for CMod_{k+1}. If $\mathsf{CMod}_k \setminus \{v^k\} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, we denote CMod_{k+1} as $\mathsf{CMod}_k \setminus \{v^k\}$; and otherwise, $\mathsf{CMod}_{k+1} = \mathsf{CMod}_k$.

Clearly, by construction of CMod_{k+1}, (1) and (2) immediately hold. For proving (3), we observe that if $v^i \in \mathsf{CMod}_{k+1}$, $v^i \in \mathsf{CMod}_i$. In such case, $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$. Hence, by Lemma 7.11, there exists an abstract execution valid w.r.t. $(\mathsf{CMod}_i \setminus \{v^i\}, \mathsf{OpSpec})$ that is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. As $\mathsf{CMod}_{k+1} \subseteq \mathsf{CMod}_i$, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \subseteq \mathsf{CMod}_i \setminus \{v^i\}$ and hence, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \preccurlyeq \mathsf{CMod}_i \setminus \{v^i\}$. Therefore, by Lemma 7.11, ξ is valid w.r.t. $(\mathsf{CMod}_{k+1} \setminus \{v^i\}, \mathsf{OpSpec})$. Thus, as ξ is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$; so we conclude (3).

For the limit case, we define CMod_k as the intersection of all consistency models $\mathsf{CMod}_i, i < k$. We observe that in this case, (2) immediately holds by construction of CMod_k.

For proving (3) we observe that $v^i \in \mathsf{CMod}_k$ iff $v^i \in \mathsf{CMod}_i$. In such case, $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$; so by Lemma 7.11, there exists an abstract execution ξ valid w.r.t. $(\mathsf{CMod}_i \setminus \{v^i\}, \mathsf{OpSpec})$ that

[7]Without loss of generality, we can assume that limit ordinals in α are not associated to a visibility formula.

is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. Similarly to the inductive case, we deduce using Lemma 7.11 that ξ is valid w.r.t. $(\mathsf{CMod}_k \setminus \{v^i\}, \mathsf{OpSpec})$. Therefore, we conclude that $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$.

For proving (1), we reason by contradiction, assuming that $\mathsf{CMod}_k \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ and reaching a contradiction. In such case, by Lemma 7.11 there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$ that is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. W.l.o.g., we can assume that ξ is minimal w.r.t. the number of events in it; and let $\mathsf{len}(\xi)$ the number of events in such execution.

For each event $r \in \xi$, we define an ordinal $i(r), i(r) < k$ associated to every visibility formula $v^i, i < k$ that can be applied on ξ. First, we note that for every pair of events, e, e' and object x, if a visibility formula $v_x(e, e')$ holds in ξ, $\mathsf{len}(v) \leq \mathsf{len}(\xi)$. Observe that there exists finite number of visibility formulas v in CMod with at most length $\mathsf{len}(\xi)$: on one hand, for each $j, 1 \leq j \leq \mathsf{len}(v)$, Rel_j^v is either so, wr, rb or ar. On the other hand, wrCons is defined as a conjunction of predicates from a finite set. Thus, the number of possible visibility formulas v of length $\mathsf{len}(v) \leq \mathsf{len}(\xi)$ is finite. Let i_r be the biggest index of a visibility formula $v^i \in \mathsf{CMod}$ s.t. $\mathsf{len}(v^i) \leq \mathsf{len}(\xi)$ and $i < k$; and let $i(r) = i_r + 1$. Observe that k is a limit ordinal, $i(r) < k$.

Let x be an object and r be an event in ξ. We show that $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. As $\mathsf{CMod}_k \subseteq \mathsf{CMod}_{i(r)}$, $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. For showing $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k])$, let $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. In such case, there exists a visibility formula v^i s.t. $v^i(w, r)$ holds in ξ. If $i > k$, by (2) $v^i \in \mathsf{CMod}_k$. Otherwise, $i < i(r)$, so by (2), $v^i \in \mathsf{CMod}_i$. Observe that in this case, applying the induction hypothesis (2) on every consistency model $\mathsf{CMod}_j, j < k$, $v^i \in \mathsf{CMod}_j$, we deduce that $v^i \in \mathsf{CMod}_k$. Either way, we deduce that $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k])$. In conclusion, $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$.

We conclude a contradiction by showing that ξ is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$; which by assumption it is not. Let e be the last event w.r.t. ar in ξ. For reaching such contradiction, as $i(e) < k$ and $\mathsf{CMod}_{i(e)} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, it suffices to show that ξ is valid w.r.t. $(\mathsf{CMod}_{i(e)}, \mathsf{OpSpec})$. We show that $\mathsf{wr}_x^{-1}(e') = \mathsf{rspec}(e')(x, [\xi, \mathsf{CMod}_{i(e)}])$.

On one hand, if $e' = e$, we note that $\mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_{i(e)}])$. As ξ is valid w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$, we conclude that $\mathsf{rspec}(e)(x, [\xi, \mathsf{CMod}_{i(e)}]) = \mathsf{wr}_x^{-1}(e)$.

On the other hand, if $e' \neq e$, let ξ' be the execution obtained by removing e from ξ. By the minimality of ξ, ξ' is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. By induction hypothesis (1), $\mathsf{CMod} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}_{i(e)}$. Hence, ξ' is valid w.r.t. $(\mathsf{CMod}_{i(e)}, \mathsf{OpSpec})$. We thus deduce that $\mathsf{wr}_x^{-1}(e') = \mathsf{rspec}(e')(x, [\xi, \mathsf{CMod}_{i(e)}])$. In conclusion, CMod_k satisfies (1) and thus, the inductive step.

Finally, we define $\mathsf{bnCMod}_{\mathsf{OpSpec}} = \mathsf{CMod}_\alpha$. As CMod_α satisfies (1) and (3), it is a consistency model OpSpec-equivalent to CMod composed of finite, non-vacuous w.r.t. $(\mathsf{CMod}_\alpha, \mathsf{OpSpec})$ visibility formulas; so we conclude that it is a consistency model in basic normal form.

□

Conflict-Strengthening a Consistency Model

Lemma B.8. *Let* OpSpec *be an operation specification. For every consistency model* CMod *in basic normal form w.r.t.* OpSpec *there exists a* OpSpec*-equivalent consistency model that is in normal form.*

Proof. We transform CMod to define $\mathsf{nCMod}_{\mathsf{OpSpec}}$, a consistency model in normal form that is OpSpec-equivalent to CMod.

For every visibility formula $v \in \mathsf{CMod}$, we define v' as the visibility formula that only differs with v on its conflict predicate. More specifically, we require that for every set $E \in \mathcal{P}(\varepsilon_0, \ldots \varepsilon_{\mathsf{len}(v)})$, we require that $\mathsf{conflict}(E) \in v'$ (resp. $\mathsf{conflict}_x(E) \in v'$) iff (1) for every abstract execution ξ, every object x and every collection of events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ_v, there exists an object

$y \neq x$ s.t. if $\varepsilon_i \in E, 0 \leq i \leq \mathsf{len}(v)$, then $\mathsf{wspec}(e_i)(y, [\xi, \mathsf{CMod}]) \downarrow$ (resp. $\mathsf{wspec}(e_i)(x, [\xi, \mathsf{CMod}]) \downarrow$) and (2) there is no strict superset of E satisfying (1). We define $\mathsf{nCMod}_{\mathsf{OpSpec}}$ as the set containing all such visibility formulas. For conclude the result, we first prove that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ for then deduce that $\mathsf{nCMod}_{\mathsf{OpSpec}}$ is indeed a consistency model in normal form.

We show that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$. On one hand, as every visibility formula v' enforces more conflicts than v, $\mathsf{nCMod}_{\mathsf{OpSpec}} \preccurlyeq \mathsf{CMod}$. On the other hand, by the definition of v', for every abstract execution ξ, object x and events w, r, if $v'_x(w, r)$ holds in ξ, $v_x(w, r)$ also holds in ξ. Altogether, we conclude that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$.

To show that $\mathsf{nCMod}_{\mathsf{OpSpec}}$ is a consistency model in normal form, we observe that by construction, every visibility formula $v \in \mathsf{nCMod}_{\mathsf{OpSpec}}$ is in simple form and it is conflict-maximal w.r.t. OpSpec. Hence, it suffices to prove that every visibility formula $v \in \mathsf{nCMod}_{\mathsf{OpSpec}}$ is non-vacuous w.r.t. $\mathsf{nCMod}_{\mathsf{OpSpec}}$.

Let v' be a visibility formula of $\mathsf{nCMod}_{\mathsf{OpSpec}}$. Observe that by construction of $\mathsf{nCMod}_{\mathsf{OpSpec}}$, $\mathsf{nCMod}_{\mathsf{OpSpec}} \setminus \{v'\} \equiv \mathsf{CMod} \setminus \{v\}$. Hence, as $\mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, we deduce that $\mathsf{nCMod}_{\mathsf{OpSpec}} \setminus \{v'\} \equiv \mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod} \equiv \mathsf{nCMod}_{\mathsf{OpSpec}}$. In other words, v' is non-vacuous w.r.t. $(\mathsf{nCMod}_{\mathsf{OpSpec}}, \mathsf{OpSpec})$.

□

B.2 Arbitration-Free Well-Formedness

As described in Section 7.1, a consistency model is arbitration-free if a OpSpec-equivalent consistency model in normal form is arbitration-free. In Theorem B.9, we present a result that states that arbitration-free is well-defined, as either every OpSpec-equivalent consistency model in normal form are arbitration-free or none.

Regarding notations, for a visibility formula v and $i, 0 \leq i \leq \mathsf{len}(v)$ we denote hereinafter $\mathsf{conflictsOf(v, i)} \in \mathcal{P}(\mathcal{P}(\varepsilon_0, \dots \varepsilon_{\mathsf{len}(v)}))$ to the sets of conflicts of ε_i in v, i.e. $E \in \mathsf{conflictsOf(v, i)}$ iff $\varepsilon_i \in E$ and $\mathsf{conflict}(E) \in v$.

Theorem B.9. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification and let* CMod *be a consistency model. For every pair of consistency models in normal form* n_1, n_2 *that are* OpSpec-*equivalent to* CMod, n_1 *is arbitration-free iff* n_2 *is arbitration-free.*

Proof. We prove the result by contradiction, assuming that there exists two consistency models n_1, n_2 in normal form, OpSpec-equivalent to CMod, but one of them arbitration-free and the other one no. W.l.o.g., we can assume that n_1 is arbitration-free and n_2 is not. On one hand, as n_2 is not arbitration-free w.r.t. OpSpec, there exists a visibility formula $v \in n_2$ s.t. v is not arbitration-free. We construct an abstract execution that is valid w.r.t. (n_1, OpSpec) but not valid w.r.t. (n_2, OpSpec) using v, reaching a contradiction.

First of all, observe that by Lemma 6.4, n_1 is weaker than CC. The abstract execution we construct contains a collection events $e_0, \dots e_{\mathsf{len}(v)}$ s.t. ξ is valid w.r.t. (CC, OpSpec) and $v_x(e_0, \dots e_{\mathsf{len}(v)})$ holds on it; for some object x.

Let x be an object. For each set $E \in \mathcal{P}(\varepsilon_0, \dots e_{\mathsf{len}(v)})$ we consider a distinct object y_E, also distinct from x. These objects represents each different conflict in v in an explicit manner.

We denote by $E_x \in \mathcal{P}(\varepsilon_0, \dots e_{\mathsf{len}(v)})$ to the set s.t. $\mathsf{conflict}_x(E_x) \in v$. Also, for every $i, 0 \leq i \leq \mathsf{len}(v)$, we denote by X_i to the set containing objects y_E (resp. x) iff $E \in \mathsf{conflictsOf(v, i)}$ (resp. $E_x \in \mathsf{conflictsOf(v, i)}$). We denote by X to the union of sets $X_i, 0 \leq i \leq \mathsf{len}(v)$.

For obtaining ξ, we construct a sequence of executions $\xi^i, 0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event **`init`**, and incorporating at each time a new event e_i. We use the notation h^{-1} and ξ^{-1} to describe the history and abstract execution containing only **`init`** respectively. We use the convention $e_{-1} = \mathtt{init}$, $\mathsf{conflictsOf}(\mathsf{v}, -1) = \mathsf{Objs}$ and $\tilde{x}_{-1} = o_{-1} = x$ (the usage of such conventions will be clearer later).

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1} \dots e_{i-1}$ and is well-defined (satisfies Definition 3.4) and we construct the history h^i and the abstract execution ξ^i. First of all, we impose the constraint that if $i > 0$, then $r_i = r_{i-1}$ iff $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{so}$, and otherwise $r_i \neq r_j, 0 \leq j < i$.

Also, we define a pair of special objects, $\tilde{x}_i$ and o_i. The purpose of object $\tilde{x}_i$ is control the number of events in ξ that write object x. Equation (48) describes $\tilde{x}_i$; where choice is a function that deterministically chooses an element from a non-empty set. The object o_i is an object different from objects $x, y_E, E \in \mathcal{P}(\varepsilon_0, \dots \varepsilon_{\mathsf{len}(v)})$ and $o_j, -1 \leq j < i$ that we use for ensuring that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $(e_{i-1}, e_i) \in \mathsf{wr}$.

$$\tilde{x}_i = \begin{cases} \tilde{x}_{i-1} & \text{if } X_i = \emptyset \\ x & \text{if } X_i \neq \emptyset \text{ and } x \in X_i \\ \mathsf{choice}\,(X_i) & \text{if } X_i \neq \emptyset \text{ and } x \notin X_i \end{cases} \tag{48}$$

We select a domain D_i, a set of objects $W_i, W_i \subseteq D_i$ that event e_i must write, and a set of objects $C_i \subseteq D_i$ whose value needs to be corrected for e_i in ξ_{i+1} – in the sense of Definition 7.13. We distinguishing between several cases:

- $\underline{i = 0 \text{ or } 0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset}$: In this case, we select e_i to be a write event. If OpSpec only allows single-object atomic read-write events, we define $D_i = X_i$; while if not, we consider a domain containing o_{i-1}, o_i, every object in X_i but no object from $X \setminus X_i$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i-1, i$. Observe that by Proposition B.10, such domain always exist on OpSpec.
  If there is an unconditional write event whose domain is D_i, we define $W_i = D_i$. Otherwise, we define $W_i = X_i \cup \{o_i\}$.
- $\underline{0 < i \leq \mathsf{len}(v), \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr} \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset}$: In this case, by Proposition B.11, OpSpec allows atomic read-write events. If OpSpec only allows single-object atomic read-write events, we define $D_i = X_i$; while if not, we consider a domain containing o_{i-1}, o_i, every object in X_i but no object from $X \setminus X_i$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i-1, i$. Observe that by Proposition B.10, such domain always exist on OpSpec.
  Similarly to the previous case, if there is an unconditional atomic read-write event whose domain is D_i, we define $W_i = D_i$. Otherwise, we define $W_i = X_i \cup \{o_i\}$.
- $\underline{0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset}$: In this case, by Proposition B.11, OpSpec allows events that do not unconditionally write. If OpSpec allows read events that are not write events, we select D_i to be the domain of any such event and $W_i = \emptyset$. Otherwise, OpSpec must allow conditional write events; so we select D_i to be the domain of any such event, $W_i = \emptyset$. Observe that in this case, thanks to the assumptions on OpSpec (see Section 7.4), we can assume without loss of generality that whenever $o_{i-1} \in D_{i-1}$, $o_{i-1} \in D_i$ as well; while otherwise, that $\tilde{x}_{i-1} \in D_i$.

Finally we describe the event e_i thanks to the sets D_i and W_i. If $W_i = D_i$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, we select an unconditional atomic read-write event whose domain is D_i. If $W_i = D_i$ and $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr}$, we select an unconditional write event whose domain is D_i. If $W_i = \emptyset$ and OpSpec allows read events that are not write events, we select a read event whose domain is D_i. Finally, if that is not the case, we select a conditional write event e_i s.t. $\mathsf{obj}(e_i) = D_i$ and s.t. an execution-corrector exists for $(e_i, W_i, \tilde{x}_i, \xi^{i-1} \oplus e_i)$. Such event always exists by the assumptions on operation specifications (Section 7.4). W.l.o.g. we can assume that e_i happens on replica r_i.

For concluding the description of $h^i = (E_i, \text{so}^i, \text{wr}^i)$ and $\xi^i = (h^i, \text{rb}^i, \text{ar}^i)$, we use an auxiliary history and abstract execution, $h^i_0 = (E^i_0, \text{so}^i_0, \text{wr}^i_0)$ and $\xi^i_0 = (h^i_0, \text{rb}^i_0, \text{ar}^i_0)$ respectively. For describing the write-read dependencies of e_i in ξ^0_i, we define the context mapping $c^i : \text{Objs} \longrightarrow \text{Contexts}$, associating each object y to the context $c^i(y)$ described in Equation (49).

$$c^i(y) = (F^i(y), \text{rb}^{i-1}_{\restriction F^i(y) \times F^i(y)}, \text{ar}^{i-1}_{\restriction F^i(y) \times F^i(y)}) \tag{49}$$

where $F^i(y)$ is the mapping associating each object y with the set of events described below:

$$F^i(y) = \begin{cases} \{\textbf{init}\} & \text{if } i = 0 \text{ or if } 0 < i \leq \text{len}(v) \land \text{Rel}_i = \text{ar} \\ \left\{ e \in E^{i-1} \;\middle|\; \begin{array}{l} \text{wspec}(e)(y, [\xi^{i-1}, \texttt{CC}]) \downarrow \text{and} \\ (e, e_{i-1}) \in (\text{rb}^{i-1})^* \end{array} \right\} & \text{otherwise} \end{cases}$$

Then, we define ξ^i_0 as the abstract execution of the history $h^i_0 = (E^i_0, \text{so}^i_0, \text{wr}^i_0)$ obtained by appending e_i to h^i_0 and ξ^i_0 as follows: E^i_0 contains E^{i-1} and event e_i. First of all, we require that the relations so^i_0, wr^i_0, rb^i_0 and ar^i_0 contain so^{i-1}, wr^{i-1}, rb^{i-1} and ar^{i-1} respectively. With respect to event e_i, we impose that e_i is the maximal event w.r.t. so^i_0 among those on the same replica. Also, e_i is maximal w.r.t. wr as we define that for every object z, ${\text{wr}^i_{0z}}^{-1}(e_i) = \text{rspec}(e_i)(z, c_i(z))$. For describing rb^i_0, we require that for every event e s.t. $(e, e_i) \in \text{so}^i_0$, $(e, e_i) \in \text{rb}^i$. Also, if $\text{Rel}^{\text{v}}_i = \text{rb}$, we impose that $(e_{i-1}, e_i) \in \text{rb}^i_0$. Finally, we require that for every pair of events $e, e' \in E^{i-1}$ s.t. $(e, e') \in \text{rb}^{i-1}$ and $(e', e_i) \in \text{so}^i_0$, $(e, e_i) \in \text{rb}^i_0$. With respect to ar^i_0, we impose that e_i is the maximum event w.r.t. ar in ξ^i_0.

We use ξ^i_0 to construct ξ^i. If event e_i is not a conditional write event, $\xi^i = \xi^i_0$. Otherwise, if event e_i is a conditional write event, given W_i and object $\tilde{x}_i$, we select an execution-corrector for e_i w.r.t. $(\texttt{CC}, \text{OpSpec})$ and a_i. W.l.o.g., we assume that every event mapped by a_i happens on replica r_i. Observe that by the choice of sets D_i and W_i, and thanks to the assumptions on storages (see Section 7.4), such event(s) are always well-defined.

In addition, we denote by C_i to the set of objects we need to correct for e_i. More specifically, if e_i is a conditional write-read, we denote by C_i to the set of objects y s.t. $a_i(y)$ is defined, i.e. $C_i = \{y \in \text{Objs} \mid a_i(y) \downarrow\}$. In the case e_i is not a conditional write-read, we use the convention $C_i = \emptyset$. The set of events in ξ^i is the following: $E^i = E^{i-1} \cup \{e_i\} \bigcup_{y \in C_i \setminus \{o_{i-1}\}} a_i(y)$. Observe that by the choice of C_i, the set E^i is well-defined.

Concerning notations, we use $c \oplus a$ to denote the context obtained by appending a to the context $c = \{E, \text{rb}, \text{ar}\}$ as the rb-maximum and ar-maximum event.

From ξ^i_0, we define $\xi^i = \xi^i_0 \overset{\text{seq}(a_i)}{\curlyvee} e_i$ as the corrected execution of ξ and e_i with events a_i. For describing ξ^i, we consider $<$ to be a well-founded order over Objs. ξ^i satisfies the following:

- $\underline{\text{so}^i}$: Let $y \in C_i$. We require that for every event $e \in E^{i-1}$, $(e, a_i(y)) \in \text{so}^i$ iff $\texttt{rep}(e) = r_i, 0 \leq j < i$. We also require that $(\textbf{init}, a_i(y)) \in \text{so}^i$ and $(a_i(y), e_i) \in \text{so}^i$. Finally, we require that for every objects $y' \in C_i, y' < y$, $(a_i(y'), a_i(y)) \in \text{so}^i$.
- $\underline{\text{wr}^i}$: Let y be an object in C_i. For every object z, if $z \in C_i$ and $z < y$, we require that $(\text{wr}^i_z)^{-1}(a_i(y)) = \text{rspec}(a_i(y))(z, c^i(z) \oplus a_i(z))$; while otherwise, we require that $(\text{wr}^i_z)^{-1}(a_i(y)) = \text{rspec}(a_i(y))(z, c^i(z))$. We also require that for every object z, if $z \in C_i$, then $(\text{wr}^i_z)^{-1}(e_i) = \text{rspec}(e_i)(z, c^i(z) \oplus a_i(z))$, while otherwise, $(\text{wr}^i_z)^{-1}(e_i) = \text{rspec}(e_i)(z, c^i(z))$.
- $\underline{\text{rb}^i}$: Let $y \in C_i$. We require that for every object $y \in C_i$ and event e s.t. $(e, a_i(y)) \in \text{so}^i \cup \text{wr}^i$, $(e, a_i(y)) \in \text{rb}^i$. Also, if $\text{Rel}^{\text{v}}_i = \text{rb}$, we impose that $(e_{i-1}, a_i(y)) \in \text{rb}^i$. Finally, we require that for every pair of events $e, e' \in E^{i-1}$ s.t. $(e, e') \in \text{rb}^{i-1}$ and $(e', a_i(y)) \in \text{so}^i$, $(e, a_i(y)) \in \text{rb}^i$.

- $\underline{\mathsf{ar}^i}$: We impose that for every event $e \in E^{i-1}$, $(e, a_i(y)) \in \mathsf{ar}^i, y \in C_i$. We also require that for every pair of objects $y_1, y_2 \in C_i$ s.t. y_1, y_2, $(a_i(y_1), a_i(y_2)) \in \mathsf{ar}^i$.

We then define $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$. Observe that by construction of h^i and ξ^i, they satisfy Definitions 3.2 and 3.4 respectively; so they are a history and an abstract execution respectively. In particular, observe that ξ^i is a correction of the abstract execution ξ_0^{i-1} with events a_i.

Finally, we define $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ as, respectively, the history $h^{\mathsf{len}(v)}$ and the abstract execution $\xi^{\mathsf{len}(v)}$. We prove that ξ is the abstract execution we were looking for.

First, we show that ξ is valid w.r.t. n_2: as ξ is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$ (Corollary B.13), so by Lemma 6.4, it is valid w.r.t. (n_1, OpSpec). As $n_1 \equiv_{\mathrm{OpSpec}} n_2$, ξ is valid w.r.t. (n_2, OpSpec). Next, we deduce in Proposition B.16 that OpSpec is maximally layered w.r.t. ar. For proving such result, we rely on Propositions B.14 and B.15. Finally, we conclude in Proposition B.17 that the layer bound of rspec is bounded by the number of arbitration-free suffixes of v. However, this implies that v is vacuous w.r.t. n_2 (Proposition B.18); which is impossible by the choice of v. The contradiction arises from assuming that n_1 is arbitration-free but n_2 is not; so we conclude the result. □

Proposition B.10. *Let* OpSpec *be a storage that allows multi-object write (resp. read-write) events whose domain is not* Objs*. Then, for every pair of finite disjoint sets F_1, F_2 there exists a domain D in* OpSpec *s.t. $F_1 \subseteq D$ but $F_2 \cap D = \emptyset$.*

Proof. The result is immediate as F_1 is finite. Hence, by the assumptions on operation specifications (Section 7.4), F_1 is a domain on OpSpec. □

Proposition B.11. *Let v be a visibility formula and $i, 0 < i \leq \mathsf{len}(v)$. If $\mathsf{conflictsOf(v,i)} \neq \emptyset$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$,* OpSpec *allows read-write events. If $\mathsf{conflictsOf(v,i)} = \emptyset$ allows events that do not unconditionally write.*

Proof. Observe that as v is non-vacuous w.r.t. $(\mathrm{CMod}, \mathrm{OpSpec})$, $\mathrm{CMod} \setminus \{v\} \not\equiv_{\mathrm{OpSpec}} \mathrm{CMod}$. By Proposition B.6, there exists an execution $\overline{\xi}$ valid w.r.t. $\mathrm{CMod} \setminus \{v\}$, an object z and events $f_0, \dots f_{\mathsf{len}(v)}$ s.t. $v_z(f_0, \dots f_{\mathsf{len}(v)})$ holds in $\overline{\xi}$.

On one hand, if $\mathsf{conflictsOf(v,i)} \neq \emptyset$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, as $\overline{\xi}$ is valid w.r.t. $\mathrm{CMod} \setminus \{v\}$, there exists z s.t. $\mathrm{rspec}(f_i)(z, [\overline{\xi}, \mathrm{CMod} \setminus \{v\}]) \neq \emptyset$. Also, $\mathsf{conflictsOf(v,i)} \neq \emptyset$ iff f_i writes on some object z'. Hence, f_i is a read-write event.

On the other hand, if $\mathsf{conflictsOf(v,i)} = \emptyset$, as v is conflict-maximal w.r.t. OpSpec, event f_i does not necessarily write any object. Thus, OpSpec allows events that do not unconditionally write. □

Proposition B.12. *The abstract execution ξ described in Theorem B.9 satisfies that for every $i, 0 \leq i \leq \mathsf{len}(v)$:*

(1) *For every object $y \in C_i$, the following conditions hold:*
  (a) *For every object $z \in$ Objs, if $z \in C_i$ and $z < y$, $G(a_i(y), z) = F^i(z) \cup \{a_i(z)\}$, while otherwise, $G(a_i(y), z) = F^i(z)$.*
  (b) *The execution $\xi^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$.*
(2) *For the event e_i, the following conditions hold:*
  (a) *For every object z, if $z \in C_i$, $G(e_i, z) = F^i(z) \cup \{a_i(z)\}$, while otherwise $G(e_i, z) = F^i(z)$.*
  (b) *The execution ξ^i is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$.*

where $\mathsf{ctxt}_z(e, [\xi, \mathsf{CC}]) = (G(e,z), \mathsf{rb}_{\upharpoonright G(e,z) \times G(e,z)}, \mathsf{ar}_{\upharpoonright G(e,z) \times G(e,z)})$.

Proof. We prove the result by induction. In particular, we show that for every $i, -1 \leq i \leq \mathsf{len}(v)$ and object y, either (0) $i = -1$ or (1) and (2) hold. The base case, $i = -1$, is immediate as (0) holds; so let us suppose that the result holds for every $j, -1 \leq j < i$, and let us prove it for i.

For proving the inductive step, we first prove (1) and then (2). As both (1) and (2) have an identical proof (observe that the role of object y in the former is just to declare that event $a_i(y)$ is well-defined), we present only the proof of (1).

We show (1) by transfinite induction. Let α be an ordinal of cardinality $|\mathsf{Objs}|$. For every $k, 0 \leq k \leq \alpha$, we denote by V_k to the set containing the first k elements in Objs according to $<$. We show that (1) holds for every $y \in V_k \cap C_i$.

The base, V_0 is immediate as $V_0 = \emptyset$. We thus focus on the successor case (i.e., showing that if (1) holds for every object $y \in V_k \cap C_i$ it also holds for V_{k+1}), as the limit case is immediate: if k is a limit ordinal, $V_k = \bigcup_{i,i<k} V_i$; so (1) immediately holds. For showing that (1) holds for every object $y \in V_{k+1} \cap C_i$, as by induction hypothesis it holds for every object $y \in V_k \cap C_i$, it suffices to show it for the only object $y \in V_{k+1} \setminus V_i$. W.l.o.g., we can assume that $y \in C_i$; as otherwise the result is immediate.

We first prove (1a) and then we show (1b). Let $z \in \mathsf{Objs}$ be an object. Two cases arise depending on $\mathsf{Rel}^{\mathsf{v}}_i$.

On one hand, if $i = 0$ or $i > 0 \land \mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}$, $F^i(z) = \{\mathbf{init}\}$. As $\mathbf{init} \in G(a_i(y), z)$, it suffices to show that the only non-initial event in E in $G(a_i(y), z)$ is $a_i(z)$ (whenever $z \in C_i$ and $z < y$). Observe that an event e belongs to $G(a_i(y), z)$ if $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in \mathsf{rb}^+$. As $a_i(y) \in E^i$, by construction of ξ, e must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$.

Observe that as either $i = 0$ or $0 < i \leq \mathsf{len}(v) \land \mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}$, by definition of rb^i, $e \notin E^{i-1}$. Thus, e must be an event in $E^i \setminus E^{i-1}$. Observe that by construction of ξ, as $(e, a_i(y)) \in (\mathsf{rb}^i)^+$, such event must be an event $a_i(w), w \in C_i, w < y$. As $\xi^i = \xi^i_0 \overset{a_i}{\vee} e_i$, by induction hypothesis (1b), we deduce that $\xi^i \restriction w$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Hence, as $\mathsf{wspec}(a_i(w))(z, [\xi^i, \mathsf{CC}]) \downarrow$, we deduce thanks to Property 1 of Definition 7.13 that $z = w$ – so $z \in C_i$ and $z < y$.

On the other hand, if $0 < i \leq \mathsf{len}(v) \land \mathsf{Rel}^{\mathsf{v}}_i \neq \mathsf{ar}$, two sub-cases arise: $z \in C_i, z < y$ or not. Both cases are identical, so we present the former, i.e., if $z \in C_i, z < y$, then $F^i(z) \cup \{a_i(z)\} = G(a_i(y), z)$.

For proving that $F^i(z) \cup \{a_i(z)\} \subseteq G(a_i(y), z)$, we split the proof in two blocks: showing that $F^i(z) \subseteq G(a_i(y), z)$ and showing that $a_i(z) \in G(a_i(y), z)$.

For showing that $F^i(z) \subseteq G(a_i(y), z)$, let e be an event in $F^i(z)$. In such case, to $e \in E^{i-1}$, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}) \in (\mathsf{rb}^i)^*$. By the construction of ξ, it is easy to see that any such event belongs to E^i, $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}) \in \mathsf{rb}^*$. As $\mathsf{Rel}^{\mathsf{v}}_i \neq \mathsf{ar}$, we deduce that $(e_{i-1}, a_i(y)) \in \mathsf{rb}^i \subseteq \mathsf{rb}$. Hence, $(e, a_i(y)) \in \mathsf{rb}^+$; so $e \in G(a_i(y), z)$. This show that $F^i(z) \subseteq G(a_i(y), z)$.

For showing that $a_i(z) \in G(a_i(y), z)$, we observe that $\xi^i = \xi^i_0 \overset{a_i}{\vee} e_i$. As $z < y$, by induction hypothesis (1b), $\xi^i \restriction z$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Thus, by Property 1 of Definition 7.13, $\mathsf{wspec}(a_i(z))(z, [\xi^i, \mathsf{CC}]) \downarrow$. Hence, $\mathsf{wspec}(a_i(z))(z, [\xi, \mathsf{CC}]) \downarrow$. As $z < y$, $(a_i(z), a_i(y)) \in \mathsf{so}^i \subseteq \mathsf{so}$; so we conclude that $a_i(z) \in G(a_i(y), z)$.

We conclude the proof of the inductive step of (1a) by showing the converse i.e. $F^i(z) \cup \{a_i(z)\} \supseteq G(a_i(y), z)$. Let $e \in G(a_i(y), z)$. First of all, by the definition of Causal visibility formula (see Figure 4b), $e \in G(a_i(y), z)$ iff $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in \mathsf{rb}^+$. Observe that if $(e, a_i(y)) \in \mathsf{rb}^+$, by construction of ξ, such event must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$. We prove that if $e \in E^{i-1}$ then $e \in F^i(z)$, while otherwise, if $e \in E^i \setminus E^{i-1}$, then $e = a_i(z)$.

If $e \in E^{i-1}$, as $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$, $\mathsf{wspec}(e)(z, [\xi^{i-1}, \mathsf{CC}]) \downarrow$. Also, as $\mathsf{Rel}^{\mathsf{v}}_i \neq \mathsf{ar}$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$, we deduce that $(e, e_{i-1}) \in (\mathsf{rb}^{i-1})^*$. In other words, $e \in F^i(z)$.

Otherwise, if $e \in E^i \setminus E^{i-1}$, we note that by construction of ξ, the only events in $E^i \setminus E^{i-1}$ s.t. $(e, a_i(y)) \in (\mathsf{rb}^i)^+$ are events $a_i(w), w \in C_i, w < y$. As $\xi^i = \xi^i_0 \overset{\mathsf{seq}(a_i)}{\vee} e_i$ and $z < y$, $\xi^i \restriction z$ is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$. Thus, as $\mathrm{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$, by Property 1 of Definition 7.13 we conclude that $e = a_i(z)$.

For concluding the inductive step, we show that (1b) holds. This is immediate by the definition of wr^i: for every event $e \in \xi^i \restriction y$, by induction hypothesis (1a) or (2a) – depending on whether $e = e_j$ or $a_j(w)$, where $0 \leq j \leq i, w \in C_i$ – $(\mathsf{wr}^i)_z^{-1}(e) = \mathrm{rspec}(e)(\mathsf{CC}, [\xi^i \restriction y, z])$. Thus, $\xi^i \restriction y$ is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$.

□

A consequence of Proposition B.12 is the following result.

Corollary B.13. *The abstract execution ξ described in Theorem B.9 is valid w.r.t.* $(\mathsf{CC}, \mathrm{OpSpec})$.

Proposition B.14. *The predicate $v_{x_0}(e_0, \dots e_{\mathsf{len}(v)})$ holds in the abstract execution ξ described in Theorem B.9.*

Proof. The proof is a simple consequence of ξ's construction. To show that $v_{x_0}(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ, we first show that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}, e_i) \in \mathsf{Rel}_i^{\mathsf{v}}$ and to then prove that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ.

We prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}, e_i) \in \mathsf{Rel}_i^{\mathsf{v}}$. Four cases arise depending on $\mathsf{Rel}_i^{\mathsf{v}}$.

- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{so}}$: In this case, by construction of events e_{i-1}, e_i, we know that $r_i = r_{i-1}$. Hence, $\underline{(e_{i-1}, e_i)} \in \mathsf{so}^i \subseteq \mathsf{so}$.
- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}}$: In this case, we first show that there is an object $y \in D_i \cap W_{i-1} \setminus C_i$, and then show $\underline{\text{that}}$ $(e_{i-1}, e_i) \in \mathsf{wr}_y$. For showing the first part, we distinguish between cases depending on whether $o_{i-1} \in D_i$ or not.
  - $\underline{o_{i-1} \in D_i}$: In this sub-case, we show that $y = o_{i-1}$. On one hand, if $\mathsf{conflictsOf(v, i)} = \emptyset$, by the choice of event e_i, $o_{i-1} \in D_{i-1} \setminus C_i$. On the other hand, if $\mathsf{conflictsOf(v, i)} \neq \emptyset$, as $o_{i-1} \in D_i$, we deduce that OpSpec allows multi-object read-write events. Observe that as v is conflict-maximal w.r.t. OpSpec, $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$. Hence, as OpSpec allows multi-object read-write events, we deduce that $o_{i-1} \in D_{i-1} \setminus C_i$. In both cases, as $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$ and $o_{i-1} \in D_{i-1}$, by the choice of W_{i-1}, we conclude that $o_{i-1} \in W_{i-1}$.
  - $\underline{o_{i-1} \notin D_i}$: In this case, we show that $y = \tilde{x}_i$. On one hand, if $\mathsf{conflictsOf(v, i)} = \emptyset$, $X_i = \emptyset$; so by the choice of $\tilde{x}_i$ (see Equation (48)), $\tilde{x}_i = \tilde{x}_{i-1}$. By the choice of D_i, $\tilde{x}_{i-1} \in D_i \setminus C_i$. Moreover, as v is conflict-maximal w.r.t. OpSpec, $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$; so $\tilde{x}_{i-1} \in X_{i-1}$. By the choice of event e_{i-1}, $X_{i-1} \subseteq W_{i-1}$. Altogether, we conclude that $\tilde{x}_i \in W_{i-1}$.
    On the other hand, if $\mathsf{conflictsOf(v, i)} \neq \emptyset$, we note that $\tilde{x}_i \in D_i \setminus C_i$. As $o_{i-1} \notin D_i$, we deduce that OpSpec only allows single-object read-write events. Thus, $D_i = \{\tilde{x}_i\}$. As v is conflict-maximal w.r.t. OpSpec, we deduce that $X_i \subseteq X_{i-1}$. As by the choice of e_{i-1}, $X_{i-1} \subseteq W_{i-1}$, we conclude that $\tilde{x}_i \in W_{i-1}$.

  We prove now that $(e_{i-1}, e_i) \in \mathsf{wr}_y$. First, we show that e_{i-1} writes y in ξ. On one hand, if e_{i-1} is an unconditional write event, $\mathrm{wspec}(e_{i-1})(y, c^i(y)) \downarrow$. On the other hand, if e_{i-1} is a conditional write event, as ξ is valid w.r.t. $(\mathsf{CC}, \mathrm{OpSpec})$ (Corollary B.13) and $y \in W_i$, by Property 2 of Definition 7.13, we deduce that $\mathrm{wspec}(e_{i-1})(y, c^i(y)) \downarrow$. Then, as $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, $e_{i-1} \in F^i(y)$. Observe that by construction of ξ, e_{i-1} is the ar-maximum event in $c^i(y)$. We note that as $y \notin C_i$, by Proposition B.12, $F^i(y) = G(e_i, y)$. To sum up, e_{i-1} is the ar-maximum event in $\mathsf{ctxt}_y(e_i, [\xi, \mathsf{CC}])$. As rspec is maximally layered, we deduce that $e_{i-1} \in \mathrm{rspec}(e_i)(y, [\xi, \mathsf{CC}])$. Finally, as ξ is valid w.r.t. CC (Corollary B.13), we conclude that $(e_{i-1}, e_i) \in \mathsf{wr}_y$.
- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{rb}}$: In this case, we explicitly stated that $(e_{i-1}, e_i) \in \mathsf{rb}^i \subseteq \mathsf{rb}$ during the construction $\underline{\text{of } \xi}$.

- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}}$: Similarly, by definition of ar^i, we know that $(e_{i-1}, e_i) \in \mathsf{ar}^i \subseteq \mathsf{ar}$.

For showing that show that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \dots e_{\mathsf{len}(v)})$, we show that for every $i, 0 \leq i \leq \mathsf{len}(v)$ and every set $E \in \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i})$, the event e_i writes on object y_E[8]. If e_i is an unconditional write, by the choice of e_i, it writes on every object in D_i. As $y_E \in D_i$, we conclude that e_i writes on y_E. Otherwise, if e_i is a conditional write, we observe that $y_E \in W_i$. Hence, as $\xi^i = \xi_0^i \overset{\mathsf{seq}(a_i)}{\vee} e_i$ and ξ^i is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Proposition B.12), we deduce using Property 2 of Definition 7.13 that $\mathsf{wspec}(e_i)(y_E, [\xi^i, \mathtt{CC}]) \downarrow$. By construction of ξ, we conclude that $\mathsf{wspec}(e_i)(y_E, [\xi, \mathtt{CC}]) \downarrow$. □

Proposition B.15. *Let ξ be the abstract execution described in Theorem B.9. For every $i, 0 \leq i < \mathsf{len}(v)$, if the ε_i suffix of v is non-arbitration-free, then $(e_i, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$.*

Proof. The proof is just an observation about the construction of ξ: for every $j, 0 < j \leq \mathsf{len}(v)$, $(e_{j-1}, e_j) \in \mathsf{rb}$ iff $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{ar}$. Hence, $(e_i, e_{\mathsf{len}(v)} \in \mathsf{rb}^+)$ iff for every $j, i < j \leq \mathsf{len}(v)$, $\mathsf{Rel}_j \neq \mathsf{ar}$. In particular, if the ε_i suffix of v is non-arbitration-free, then $(e_i, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$. □

Proposition B.16. *Let* OpSpec *be a storage,* CMod *be a consistency model in normal form w.r.t.* OpSpec *and v be a visibility formula in* CMod. *If there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t.* CMod, *an object x and events w, r s.t. $v_x(w, r)$ holds in ξ but $(w, r) \notin (\mathsf{rb})^+$, then* OpSpec *is maximally layered w.r.t.* ar.

Proof. First of all, as $v_x(w, r)$ holds in ξ, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$. If OpSpec would be maximally layered w.r.t. $(\mathsf{rb})^+$, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}])$ contains at least the first layer of $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$ w.r.t. rb. Hence, there would exist an event w' s.t. $w' \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}])$ and $(w, w') \in (\mathsf{rb})^+$. As ξ is valid w.r.t. CMod, we deduce that $(w', r) \in \mathsf{wr}$. By Definition 3.4, we deduce that $(w', r) \in \mathsf{rb}$. However, this implies that $(w, w') \in \mathsf{rb}^+$; which contradicts the assumptions. Hence, OpSpec must be maximally layered w.r.t. ar. □

Proposition B.17. *Let* OpSpec *be a storage maximally layered w.r.t.* ar, CMod *be a consistency model in normal form w.r.t.* OpSpec *and v be a non-arbitration free visibility formula in* CMod. *Let us suppose that there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t.* CMod, *an object x and events $e_0, \dots e_{\mathsf{len}(v)}$ satisfying the following:*

1. *for every non-initial event e in ξ, if $e \notin \{e_i \mid 0 \leq i \leq \mathsf{len}(v)\}$, then e does not write on x in ξ,*
2. *$v_x(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ, and*
3. *for every non-arbitration-free ε_k-suffix of v, $(e_k, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$.*

In such case, the layer bound of OpSpec *is bounded by the number of arbitration-free suffixes of v.*

Proof. We reason by contradiction, assuming that k is bigger than the number of saturable suffixes of v. We first show that $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$, for then deduce that $\mathbf{init} \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$. After that, we reach a contradiction by showing that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ but $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{wr}$; which contradicts that ξ is valid w.r.t. CMod.

We first show that $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$. As v contains less than k saturable suffixes, by the Assumption 3, there is less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$ that write on x in ξ and that succeed $e_{\mathsf{len}(v)}$ w.r.t. rb^+. As $\mathsf{wr} \subseteq \mathsf{rb}$ (see Definition 3.4), we deduce that $\mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)})$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$. As ξ is valid w.r.t. CMod, $\mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)}) = \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$; so we prove the first part.

[8]For simplifying the proof, we abuse of notation and say that $y_E = x$ if $E = E_x$. Observe that v is conflict-maximal w.r.t. OpSpec, either $\mathsf{conflict}_x(E_x)$ or $\mathsf{conflict}(E_x)$ do not belong to v.

For showing that **init** $\in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, we observe that by the Assumption 1, no other non-initial event in ξ writes on x in ξ. Hence, $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contain less than k non-initial events. As **init** $\in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$, and OpSpec is maximally layered with layer bound k, we conclude that **init** $\in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$.

For proving that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ but $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{wr}$, we observe that by the Assumption 2, $e_0 \in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$. As OpSpec is maximally layered w.r.t. ar, **init** $\in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, (**init**, $e_0) \in \mathsf{ar}$ and $e_0 \in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$; we conclude that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$.

For reaching a contradiction, we observe that v is non-arbitration-free. Hence, by the Assumption 3, $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{rb}$. Once again, as $\mathsf{wr} \subseteq \mathsf{rb}$ (see Definition 3.4), we deduce that $e_0 \notin \mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)})$. However, as $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, we conclude that ξ is not valid w.r.t. CMod; which is contradicts the hypothesis. Thus, the layer bound of OpSpec is bounded by the number of arbitration-free suffixes of v. □

Proposition B.18. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification maximally layered w.r.t.* ar, CMod *be a consistency model in normal form w.r.t.* OpSpec *and* v *be a simple, conflict-maximal w.r.t.* OpSpec*, non-arbitration-free visibility formula. If the layer bound of* rspec *is smaller or equal by the number of arbitration-free suffixes of* v*, then* $v \notin \mathsf{CMod}$.

Proof. Let v be a simple, conflict-maximal w.r.t. OpSpec, non-arbitration-free visibility formula. We show that v is vacuous w.r.t. CMod; so $v \notin \mathsf{CMod}$.

We reason by contradiction, assuming that v is non-vacuous w.r.t. CMod. In such case, $\mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ but $\mathsf{CMod} \setminus \{v\} \preccurlyeq \mathsf{CMod}$. By Proposition B.6, there exists an abstract execution on OpSpec, and object x, and events w, r s.t. $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$.

We observe that by Property 2 of Definition 7.4, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$. Hence, as $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$, we deduce that $v_x(w, r)$ holds in ξ. As v is simple, there exist events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $e_0 = w$, $e_{\mathsf{len}(v)} = r$ and $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ.

First of all, as rspec is maximally layered w.r.t. ar and $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$, every event in $\{e_0, \ldots e_{\mathsf{len}(v)}\}$ that writes x is also in $\mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$. As v is conflict-maximal w.r.t. OpSpec, at least $|E_x|$ events write on x; where $E_x \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(v)})$ s.t. $\mathsf{conflict}_x(E_x) \in v$. Observe that for every event e_i s.t. $\varepsilon_i \in E_x$ and $\mathsf{suff}_x(\mathsf{v}_x, i)$ is arbitration-free, as CMod is closed under causal suffixes, there exists a visibility formula $v' \in \mathsf{CMod}$ s.t. $v'_x(e_i, e_{\mathsf{len}(v)})$. Thus, $|E_x| \geq \mathsf{af}(v)$, where $\mathsf{af}(v)$ is the number of arbitration-free suffixes of v. Moreover, as $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$, and v is not arbitration-free, $|E_x| > \mathsf{af}(v)$. However, as the layer bound of rspec, k, is smaller or equal than the number of arbitration-free suffixes of v, the number of events read by $f_{\mathsf{len}(v)}$ is at most $\mathsf{af}(v)$. Hence, $|E_x| \leq \mathsf{af}(v)$, which contradicts that $|E_x| > \mathsf{af}(v)$. We reach a contradiction; so the initial hypothesis, that v is non-vacuous w.r.t. CMod, is false. Thus, $v \notin \mathsf{CMod}$. □

C Proof of the Basic Arbitration-Free Consistency Theorem

Let in the folloeing Spec = (CMod, OpSpec) be a basic storage specification. We show that there exists an available Spec-implementation iff CMod is arbitration-free w.r.t. OpSpec.

C.1 Arbitration-Freeness Implies Availability

As discussed in Section 6, the proof of such result is decomposed in three steps:

(1) We show that arbitration-free consistency models w.r.t. OpSpec are weaker than CC (Lemma 6.4).
(2) We deduce that available (CC, OpSpec)-implementations are also available (CMod, OpSpec)-implementations as an immediate consequence of Lemma 6.5.
(3) We prove that there exists available (CC, OpSpec)-implementations (Lemma 6.6).

Lemma 6.4. *Let* Spec = (CMod, OpSpec) *be a basic storage specification. If* CMod *is arbitration-free w.r.t.* OpSpec*, then* CMod *is weaker than* CC.

Proof. For showing that CMod is weaker than CC, let $h = (E, \mathsf{so}, \mathsf{wr})$ be a history and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be an abstract execution of h valid w.r.t. Spec. Let n be a consistency model in normal form that is OpSpec-equivalent to CMod. By Theorem B.1, such model always exists. As CMod is arbitration-free, every visibility formula $v \in n$ is arbitration-free. We conclude the result by showing that $n \preccurlyeq \mathsf{CC}$, i.e. showing that for every object x and every pair of distinct events $e, e' \in E$, if $v_x(e, e')$ holds in ξ then $v_x^{\mathsf{CC}}(e, e')$ holds in ξ as well; where v^{CC} is Causal, the visibility formula of CC (Figure 4b).

First, as $v_x(e, e')$ holds in ξ, e writes x in ξ and $\mathsf{wr}_x^{-1}(e) \neq \emptyset$. Moreover, as v is simple, for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$. By Property 2 of Definition 3.4, we deduce that $(e, e') \in \mathsf{rb}^+$. Altogether, we conclude that $v_x^{\mathsf{CC}}(e, e')$ holds in ξ. □

Lemma 6.6. *Let* OpSpec *be a basic operation specification. There exists an available* (CC, OpSpec)*-implementation.*

Proof. We define an available implementation of $\mathsf{Spec}^{\mathsf{CC}} = (\mathsf{CC}, \mathsf{OpSpec})$.

As discussed in Section 5, any implementation $I_E = (S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ can be characterized by describing its set of states S_{i}, its actions A_{i}, its initial state σ_0^{i} and its transition function Δ_{i}.

First, we define S_{i} as the set of possible values that each object may have; and the declare the initial state any possible state in S_{i}. Next, we define A_{i} via the synchronized actions Events × (Objs × Events ∪ {∅}), as well as the local actions `send` and `receive`. We assume local actions are defined in a similar way to Events, as tuples $a = (\mathsf{id}, \mathsf{r}, \mathsf{op}, \mathsf{m})$, where id is an action identifier, r is a replica identifier, op an operation identifier and m is additional metadata of the action. As for events, we use $\mathtt{id}(a)$, $\mathtt{rep}(a)$, $\mathtt{op}(a)$ and $\mathtt{md}(a)$ for indicating the identifier, replica, operation and metadata of an action a.

For describing its transition function, we rely on the definition of CC. As we design $(S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ to be an available $\mathsf{Spec}^{\mathsf{CC}}$-implementation, we require that any induced abstract execution must be valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. However, Definition 4.2 describes validity "a posteriori", i.e. validity can only be checked once the event is executed; while transition functions describe validity "a priori", i.e. describe a procedure to compute a write-read of a given, not yet added event. For solving this issue, we observe that under CC, that the context of an event e belonging to a synchronized action $a = (e, m)$ only depends on (a) the transitive set of received actions before the last action in its replica and (b) the synchronized actions executed in its own replica. Ensuring transitive communication, i.e. ensuring that every send action on replica r transmits information about all synchronized actions executed or received on replica r before such `send` action suffices to provide CC.

More in detail, for describing the transition function $\Delta_{\mathsf{i}}(t, a)$, we require that (1) a is not present in t and (2) transitive communication is ensured. Also, we require a third condition depending on the type of a:

- if a is a synchronized action, we require that (3a) if a represents a read operation, $a = (e, m)$, then e must read from the latest writing event w.r.t. ar (which coincides with the trace order) received before l^t_r,
- if a is a send action, then (3b) it precedes a synchronized action, and
- if a is a `receive` action, then (3c) there exists a unique preceding send action that matches it.

where $r = \mathtt{rep}(a)$ and l^t_r to the last action in trace t whose replica is r.

On one hand, (1) ensures that $\Delta_{\mathsf{i}}(t, e)$ is well-defined, i.e. in every trace of Δ_{i}, each action contains each action exactly once. On the other hand, (2) and (3a) ensure that I_E is a $\mathsf{Spec}^{\mathsf{CC}}$-storage implementation while (3b) and (3c) ensure that I_E is an available storage implementation.

Formally, $\Delta_{\mathsf{i}}(t, a) \downarrow$ if and only if $a \notin t$ and $\mathsf{sat}(t, a)$ holds; and in such case $\Delta_{\mathsf{i}}(t, a) = t \oplus a$. The predicate $\mathsf{sat}(t, a)$ is described in Equation (50).

$$\mathsf{sat}(t,a) = \begin{cases} a = (e, M_t(e)) & \text{if } \mathsf{op}(a) \neq \mathsf{send}, \mathtt{receive} \\ \mathsf{sendIfData}(t,a) & \text{if } \mathsf{op}(a) = \mathsf{send} \\ \quad \mathsf{sendAllData}(t,a) & \\ \quad \text{and } \mathsf{maxSend}(t,a) & \\ \mathsf{minRcv}(t,a) & \text{if } \mathsf{op}(a) = \mathtt{receive} \\ \quad \text{and } \mathsf{maxRcv}(t,a) & \end{cases} \tag{50}$$

where $M_t(e)$ is the mapping assigning to the objext $x = \mathsf{obj}(e)$ the last event that writes on x received by e, formally defined using Equations (51) and (52); and the predicates sendIfData, sendAllData, maxSend, minRcv and maxRcv are defined in Equations (53) to (56).

$$\begin{aligned} M_t(e) &= \left[x \mapsto \begin{cases} \{\max_{\mathsf{ar}^t_e} E^x_t(e)\} & \text{if } x = \mathsf{obj}(e) \\ \emptyset & \text{otherwise} \end{cases} \right]_{x \in \mathsf{Objs}} \\ E^x_t(e) &= \left\{ e' \;\middle|\; \begin{array}{l} e' \in \mathsf{Events} \cap t \;\wedge\; e' \text{ writes } x \text{ in } \mathsf{exec}(t) \;\wedge \\ (\mathtt{rep}(e') = \mathtt{rep}(e) \vee \mathsf{rec}_t(e', e)) \end{array} \right\} \\ \mathsf{ar}^t_e &= \mathsf{ar}_{\restriction E^x_t(e) \times E^x_t(e)} \end{aligned} \tag{51}$$

$$\mathsf{rec}_t(e', e) = \exists r, s \in t \text{ s.t.} \bigwedge \begin{array}{l} \mathsf{op}(r) = \mathtt{receive}, \mathtt{rep}(r) = \mathtt{rep}(e), \\ \mathsf{op}(s) = \mathsf{send}, \mathtt{rep}(s) = \mathtt{rep}(e'), \\ \mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(r), e' <_t s <_t r < e' \end{array} \tag{52}$$

$$\mathsf{sendIfData}(t, a) ::= \mathsf{op}(a'') \neq \mathsf{send} \tag{53}$$

$$\text{where } a'' = \max_{<_t} \left\{ a' \in t \;\middle|\; \mathtt{rep}(a') = \mathtt{rep}(a) \;\wedge\; \mathsf{op}(a') \neq \mathtt{receive} \right\}$$

$$\begin{aligned} \mathsf{sendAllData}(t, a) ::= \forall a' \in t. \mathtt{rep}(a') = \mathtt{rep}(a) \wedge \mathsf{op}(a') \neq \mathsf{send} \\ \implies \mathsf{RV}^x_{a'} \subseteq \mathsf{rb}\text{-}\mathsf{Set}(a) \end{aligned} \tag{54}$$

$$\text{where } \mathsf{RV}^x_{a'} = \begin{cases} \{e\} & \text{if } \mathsf{op}(a') \neq \mathsf{send}, \mathtt{receive} \wedge \\ & a' = (e, \_) \\ \mathsf{rb}\text{-}\mathtt{Set}(a') & \text{if } \mathsf{op}(a') = \mathtt{receive} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{maxSend}(t, a) ::= \quad \nexists a' \in t.\mathtt{op}(a') = \mathtt{send} \land \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{55}$$

$$\mathsf{minRcv}(t, a) ::= \quad \exists a' \in t.\mathtt{op}(a') = \mathtt{send} \land \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{56}$$

$$\mathsf{maxRcv}(t, a) ::= \nexists a' \in t.\,\mathtt{op}(a') = \mathtt{receive} \ \land\ \mathtt{rep}(a) = \mathtt{rep}(a') \land \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{57}$$

Note that as I_E contains send and receive actions, as well as events along with their write-read dependencies, I_E is a storage implementation. For proving that I_E is the searched implementation, we introduce the following notation: for a trace t and an event $e \in t$, $\mathsf{prefix}(t, e)$ to the trace s.t. $\Delta(\mathsf{prefix}(t, e), e)$ is a prefix of t.

The rest of the proof, showing that I_E is an available $\mathsf{Spec}^{\mathsf{CC}}$-implementation, is a consequence of Lemmas C.1 to C.3. □

Lemma C.1. *The implementation I_E is an $\mathsf{Spec}^{\mathsf{CC}}$-implementation.*

Proof. Let $P_E = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$ be a program. We prove by induction on the length of all traces in $\mathcal{T}_{P_E \| I_E}$ that any trace t is feasible and its induced abstract execution is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. The base case, when $t = \{(\mathbf{init}_{P_E}, \mathbf{init}_{I_E})\}$ is immediate as t contains exactly one event that does not read any object. Hence, let us assume that for any trace $t' \in \mathcal{T}_{P_E \| I_E}$ of at most length k, $\mathtt{exec}(t')$ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$; and let us show that for any trace t of length $k + 1$, $\mathtt{exec}(t)$ is also valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. Let $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be respectively the induced history $\mathsf{history}(t)$ and the induced abstract execution $\mathtt{exec}(t)$ where ar coincides with the trace order. We denote sr to the induced order between send-receive actions with the same rb-Set on t. Before proving that ξ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$, we show that t is feasible, i.e. ξ satisfies Definition 3.4.

- $\underline{\mathsf{rb} = \mathsf{rb}; \mathsf{so}^*}$: This is immediate by the definition of induced receive-before.
- $\underline{\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{rb}}$: By definition of rb, $\mathsf{so} \subseteq \mathsf{rb}$, so we focus on proving that $\mathsf{wr} \subseteq \mathsf{rb}$. Let w, r be events and x be an object s.t. $(w, r) \in \mathsf{wr}_x$. In such case, there is a pair of actions a_r, a_w s.t. $r \in a_r$, $w \in a_w$ and $w \in \mathsf{wr}\text{-}\mathtt{Set}(a_r)(x)$. Hence, $\{w\} = \max_{\mathsf{ar}_e^t} E_t^x(e)$. We deduce then that $\mathsf{rec}_t(w, r)$ must hold; which implies that there exists a send action s and a receive action v s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$ and $w <_t s <_t v <_t r$. By sendAllData predicate, $w \in \mathsf{rb}\text{-}\mathtt{Set}(s)$. As $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$, $w \in \mathsf{rb}\text{-}\mathtt{Set}(v)$. By the definition of induced abstract execution, $(w, r) \in \mathsf{rb}$.
- $\underline{\mathsf{rb} \subseteq \mathsf{ar}}$: For proving that $\mathsf{rb} \subseteq \mathsf{ar}$, as rb can be derived by sr and so, it suffices to prove that both $\mathsf{so}, \mathsf{sr} \subseteq \mathsf{ar}$. First, as so is the partial order induced by the total order $<_t$ on actions executed on the same replica, $\mathsf{so} \subseteq \mathsf{ar}$.
  Next, for proving that $\mathsf{sr} \subseteq \mathsf{ar}$, let s be a send action and let v be a receive action s.t. $(s, v) \in \mathsf{sr}$. Let us consider $p_v^t = \mathsf{prefix}(t, v)$ be the prefix of t before v. On one hand, as p_v^t is a prefix of t', $\Delta_\mathsf{i}(p_v^t, v) \downarrow$. In particular, $\mathsf{minRcv}(p_v^t, v)$ holds; so there is a send action s' in p_v^t s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s') = \mathsf{rb}\text{-}\mathtt{Set}(v)$. We show that $s' = s$. Otherwise, then w.l.o.g. $s <_t s'$. Note that $\Delta_\mathsf{i}(\mathsf{prefix}(t, s'), s') \downarrow$ as $\mathsf{prefix}(t, s') \oplus s'$ is a prefix of t'. In such case, $\mathsf{maxSend}(\mathsf{prefix}(t, s'), s')$ does not hold; which is impossible as $\Delta_\mathsf{i}(\mathsf{prefix}(t, s'), s') \downarrow$. Therefore, $s = s'$. As $s' \in p_v^t$, s precedes v in t; so $(s, v) \in \mathsf{ar}$.

After proving that t is feasible, we show that ξ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. By Definition 4.2, we need to show that for every event r and object x, if $\mathsf{rspec}(r) \uparrow$, $\mathsf{wr}_x^{-1}(r) = \emptyset$, and otherwise, $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_x(r, [\xi, \mathtt{CC}])\}$. Let r be a read event, x be the object it affects and $p = \mathsf{prefix}(t, r)$. We know by Equation (51) that $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}_r^p} E_p^x(r)\}$. Observe then that by Equation (51) and rb's definition, $E_p^x(r) = \mathsf{ctxt}_x(r, [t, \mathtt{CC}])$. Thus, we conclude that $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_x(r, [\xi, \mathtt{CC}])\}$. □

Lemma C.2. *For every program P_E and every trace t of $I_E \parallel P_E$, there is no replica waiting in t.*

Proof. Let $P_E = (S_p, A_p, s_0^p, \Delta_p)$ be a program, $r \in$ Reps be a replica and $t \in \mathcal{T}_{P_E \| I_E}$ be a reachable trace. Let also be $t_1 \in \mathcal{T}_{P_E}$ and $t_2 \in \mathcal{T}_{I_E}$ traces s.t. $t = (t_1, t_2)$. To prove that r is not waiting in t, let us suppose that there exists an event $e \in \mathsf{Events}_{P_E}$ s.t. $\mathsf{op}(e) \neq \mathsf{end}$, $\mathsf{rep}(e) = r$ and $\Delta_{P_E}(t_1, e) \downarrow$, and let us prove that there exists a non-receive action a s.t. $\Delta_{I_E \| P_E}(t, a) \downarrow$.

Let a be the action $(e, M_t(e))$; where $M_t(e)$ is described using Equation (51). We observe that as $\Delta_{P_E}(t_1, e) \downarrow$, $\Delta_{P_E \| I_E}(t, \mathsf{ex}) \downarrow$. Moreover, $\mathsf{op}(a) \neq \mathtt{receive}$. Hence, r is not waiting in t; so I_E is available. □

Lemma C.3. *For every finite program P_E, the composition $I_E \| P_E$ is also finite.*

Proof. Let $P_E = (S_p, A_p, s_0^p, \Delta_p)$ be a finite program. The implementation I_E is conditionally finite on P_E if for every trace $t \in \mathcal{T}_{P_E \| I_E}$ there exists a constant $k_t \in \mathbb{N}$ s.t. $\mathsf{len}(t) \leq k_t$. Let thus $t \in \mathcal{T}_{P_E \| I_E}$, $t_1 \in \mathcal{T}_{P_E}$, $t_2 \in \mathcal{T}_{I_E}$ be traces s.t. $t = (t_1, t_2)$. As P_E is finite, the length of t_1, $\mathsf{len}(t_1)$, is finite. We show that $k_t ::= 3 \cdot \mathsf{len}(t_1)$ is the constant we search.

Three cases arise, depending on the type of action we consider. First, by maxRcv predicate, the number of receive actions coincides with the number of receive actions with distinct metadata; which by minRcv, is bounded by the number of send actions in the trace. Then, by sendIfData, the number of send actions is bounded by the number of synchronized actions. Finally, by the parallel composition definition, the number of synchronized actions in t and t_1 coincide; so such number is bounded by $\mathsf{len}(t_1)$. Altogether, we deduce that $\mathsf{len}(t) \leq 3 \cdot \mathsf{len}(t_1) = k_t$. □

C.2 Availability Implies Arbitration-Freeness

As explained in Section 6, we prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod′ of CMod contains a simple visibility formula involving ar (see Definition 6.2). By Lemma 6.7, such a formula precludes the existence of an available (CMod′, OpSpec)-implementation. Consequently, there is no available (CMod, OpSpec)-implementation, since any such implementation would also be an available (CMod′, OpSpec)-implementation – this is an easy observation as CMod is equivalent to CMod′ (see Theorem B.1).

Proof. We assume by contradiction that there is an available implementation I_E of Spec but CMod contains a visibility formula v s.t. for some $i, 0 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$. We use the latter fact to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no receive action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

The program P we construct generalizes the litmus program presented in Figure 1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$. The events are used to “encode” two instances v_{x_0} and v_{x_1} of the visibility formula.

Let d_v be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to ε_{d_v} which is not arbitration-free, and the suffix from ε_{d_v} up to $\varepsilon_{\mathsf{len}(v)}$, the arbitration-free part. Thus, v is of the form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_{\mathsf{len}(v)}) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}. \bigwedge_{i=1}^{\mathsf{len}(v)} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \wedge \varepsilon_0 \text{ writes } x \wedge \mathsf{wr}_x^{-1}(\varepsilon_{\mathsf{len}(v)}) \neq \emptyset$$

where $\mathsf{Rel}_i^v \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}\}$, for all $i < d_v$, $\mathsf{Rel}_{d_v}^v = \mathsf{ar}$, and $\mathsf{Rel}_i^v \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$ for all $i > d_v$.

In the construction, we require that replica r_l executes events $e_i^{x_l}$ if $i < d_v$ and events $e_i^{x_{1-l}}$ otherwise – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. All objects in replica r_l access (read and/or write) x_l except $e_{\mathsf{len}(v)}^{x_l}$, which access with x_{1-l}. We denote by $\tilde{x}_i^{x_l}$ to the unique object that event $e_i^{x_l}$ reads and/or writes.

More in detail, we construct a set of events, E^i, histories, $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$, and executions, $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, $0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event $\mathbf{init}$, and incorporating at each time a pair of new events, $e_i^{x_0}$ and $e_i^{x_1}$. For simplifying notation, we use the convention $\mathbf{init} = e_{-1}^{x_0} = e_{-1}^{x_1}$.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \dots e_{i-1}^{x_0}, e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 3.4) and we construct the history h^i and the abstract execution ξ^i.

We distinguish between cases depending on the value i:

- $\underline{i = 0}$: In this case, we consider e_0 be an event s.t. $\mathsf{wspec}(e_0^{x_l})(\mathtt{wval}(\mathbf{init})(\tilde{x}_i^{x_l})) \downarrow$.
- $\underline{0 < i < \mathsf{len}(v), \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr} \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} = \mathsf{wr}}$: In this case, it is easy to see that by Proposition B.11, OpSpec allows atomic read-write events. We consider $e_i^{x_l}$ be an event s.t. $\mathsf{rspec}(e_i^{x_l}) \downarrow$ and $\mathsf{wspec}(w_i^{x_l})(\mathtt{value}_{\mathsf{wr}^{i-1}}(w_i^{x_l}, \tilde{x}_i^{x_l})) \downarrow$.
- $\underline{0 < i < \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} = \mathsf{wr}}$: In this case, if OpSpec allows unconditional writes, then we select $e_i^{x_l}$ as an unconditional write event on object $\tilde{x}_i^{x_l}$. Otherwise, we select event $e_i^{x_l}$ s.t. $\mathsf{rspec}(e_i^{x_l}) \downarrow$ and $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l})) \downarrow$.
- $\underline{0 < i < \mathsf{len}(v) \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} \neq \mathsf{wr}}$: In this case, we select $e_i^{x_l}$ to not write $\tilde{x}_i^{x_l}$ unless it is necessary. If OpSpec allows read events that are not write events, or if allows conditional atomic read-write events, we select $e_i^{x_l}$ as an event such that $\mathsf{rspec}(e_i^{x_l}) \downarrow$ but $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l})) \uparrow$. Otherwise, we select event e_i such that $\mathsf{rspec}(e_i^{x_l}) \downarrow$ and $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l})) \downarrow$.
- $\underline{i = \mathsf{len}(v)}$: In this case, we consider $e_{\mathsf{len}(v)}^{x_l}$ to be an event that reads object $\tilde{x}_i^{x_l}$, i.e. $\mathsf{rspec}(e_{\mathsf{len}(v)}^{x_l}) \downarrow$.

where $l \in \{0, 1\}$ and $w_i^{x_l} = \max_{\mathsf{ar}^{i-1}}\{e \in E^{i-1} \mid \mathsf{wspec}(e)(\mathsf{obj}(e_i^{x_l})) \downarrow \land (e, e_i^{x_l}) \in \mathsf{so}^i\}$. We note that as $\mathbf{init}$ writes on every object, $w_i^{x_l}$ is well-defined.

First of all, observe that event $e_i^{x_l}$ is well-defined thanks to Lemma C.4 and the assumptions on OpSpec (Section 4.3). We denote $E^i = E^{i-1} \cup \{e_i^{x_0}, e_i^{x_1}\}$. We observe that w.l.o.g., we can assume that the $\mathtt{id}(e_i^{x_0})$ is bigger than every identifier of an event in E^{i-1} and that $\mathtt{id}(e_i^{x_0}) < \mathtt{id}(e_i^{x_1})$.

We conclude the description of h^i and ξ^i by specifying the relations $\mathsf{so}^i, \mathsf{wr}^i, \mathsf{rb}^i, \mathsf{ar}^i$. We require that so^i (resp. wr^i, rb^i, ar^i) contains so^{i-1} (resp. wr^{i-1}, rb^{i-1}, ar^{i-1}). Also, we require additional constrains on them due to event e_i:

- $\underline{\mathsf{so}^i}$: We require that $(e, e_i^{x_l}) \in \mathsf{so}^i$ iff $\mathtt{rep}(e) = \mathtt{rep}(e_i^{x_l})$; as well as $(\mathbf{init}, e_i^{x_l}) \in \mathsf{so}^i$.
- $\underline{\mathsf{wr}^i}$: If $e_i^{x_l}$ is not a read event, we require that ${\mathsf{wr}_{x_i}^i}^{-1}(e_i^{x_l}) \neq \emptyset$. Otherwise, we require that $(\{w_i^{x_l}\}, e_i^{x_l}) \in \mathsf{wr}_{x_i}^i$.
- $\underline{\mathsf{rb}^i}$: We require that $\mathsf{rb}^i = \mathsf{so}^i$.
- $\underline{\mathsf{ar}^i}$: We impose that for every event $e \in E^i$, $(e, e_i^{x_l}) \in \mathsf{ar}^i$. Also, we impose that $(e_i^{x_0}, e_i^{x_1}) \in \mathsf{ar}^i$.

Then, we define $\mathsf{Events}_{\mathsf{p}} = E^{\mathsf{len}(v)}$ as the set our program employs. The set $\mathsf{Events}_{\mathsf{p}}$ induces the set of traces $\mathcal{T}_{\mathsf{p}}$.

We define the program $P = (S_{\mathsf{p}}, A_{\mathsf{p}}, s_0^{\mathsf{p}}, \Delta_{\mathsf{p}})$, where $\mathbf{init}_{\mathsf{p}} = \mathbf{init}$ and Δ_{p} is the transition function defined as follows: for every trace $t \in \mathcal{T}_{\mathsf{p}}$ and event $e \in \mathsf{Events}_{\mathsf{p}}$, $\Delta_{\mathsf{p}}(t, e) \downarrow$ if and only if $e \notin t$ and every event in $\mathsf{Events}_{\mathsf{p}}$ whose replica coincide with e and has smaller identifier than e is included in t.

Given such a program P, the proof proceeds as follows:

(1) There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma C.5): Since I_E is available, it can always delay receiving messages, and execute other actions instead. Then, as P is a finite program, such an execution must be finite.

(2) The trace t induces a history $h_{\mathsf{v}} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_{\mathsf{v}} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$ (ar is arbitrary as long as $\mathsf{rb} \subseteq \mathsf{ar}$). As I_E is valid w.r.t. Spec, ξ_{v} is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_{v} read the latest value w.r.t. so written on their associated object in ξ_{v} (Lemma C.6). In particular, we deduce that all traces of P without `receive` events induce the same history and therefore, the induced history does not change when the induced arbitration order changes.

(3) Since ar is a total order, either $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$ or $(e^{x_1}_{d_{\mathsf{v}}-1}, e^{x_0}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$. By Lemma C.7, we deduce that $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_{\mathsf{v}}, \mathsf{CMod}])$. The proof is explained in Figure 5: if $(e^{x_0}_{d_{\mathsf{v}}-1}, e^{x_1}_{d_{\mathsf{v}}-1}) \in \mathsf{ar}$, then all events $e^{x_0}_i$ form a path in such way that $\mathsf{v}_{x_0}(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(v)})$ holds in ξ_{v}.

(4) Since $e^{x_0}_{\mathsf{len}(v)}$ is the only event at r_1 that reads or writes x_0 and events in ξ_{v} read the latests values w.r.t. so in ξ_{v}, we deduce that $e^{x_0}_{\mathsf{len}(v)}$ reads x_0 from **init**. However, as $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_{\mathsf{v}}, \mathsf{CMod}])$ and **init** precedes $e^{x_0}_0$ in arbitration order, we deduce that $e^{x_0}_{\mathsf{len}(v)}$ does not read the latest value w.r.t. ar, i.e. $\mathsf{rspec}(e^{x_0}_{\mathsf{len}(v)}) \downarrow$ but $\mathsf{wr}^{-1}_{x_0}(e^{x_0}_{\mathsf{len}(v)}) \neq \{\max_{\mathsf{ar}} \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_{\mathsf{v}}, \mathsf{CMod}])\}$. Therefore, ξ_{v} is not valid w.r.t. Spec (see Definition 4.2). This contradicts the hypothesis that I_E is an implementation of Spec. □

Lemma C.4. *Let* Spec = (CMod, OpSpec) *be a storage specification s.t.* CMod *is in normal form w.r.t.* OpSpec. *For every visibility formula $v \in$ CMod, there exists an abstract execution valid w.r.t.* Spec, *ξ, an object x and events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $\mathsf{rspec}(e_{\mathsf{len}(v)}) \downarrow$ and $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ.*

Proof. Let $v \in \mathsf{CMod}$ be a visibility formula. As CMod is normal form w.r.t. OpSpec, v is non-vacuous; so $\mathsf{CMod} \not\equiv \mathsf{CMod} \setminus \{v\}$. Hence, there exists an abstract execution valid w.r.t. Spec, ξ, an object x and a read event r s.t. $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) \neq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$. As $\mathsf{CMod} \setminus \{v\} \preccurlyeq \mathsf{CMod}$, we conclude that there exists events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $r = e_{\mathsf{len}(v)}$ and $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ. □

Lemma C.5. *For every available storage implementation, I_E, there exists finite reachable trace $t \in \mathcal{T}_{P \parallel I_E}$ s.t.*

(1) *t does not contain any action a s.t.* $\mathsf{op}(a) = \mathtt{receive}$.
(2) *for every event $e \in \mathsf{Events}_{\mathsf{p}}$ there exists exactly one action $a \in t$ s.t. $\mathsf{ev}(a) = e$ and,*
(3) *for every two actions $a, a' \in t$ in the same replica, if $\mathsf{ev}(a) \downarrow$, $\mathsf{ev}(a') \downarrow$ and $\mathtt{id}(\mathsf{ev}(a)) < \mathtt{id}(\mathsf{ev}(a'))$, then $a <_t a'$*

Proof. Let I_E be an available storage implementation. We construct a sequence of traces $\{t^i\}_{i \in \mathbb{N}}$ s.t. for each $i \in \mathbb{N}$ (1) t^i does not contain any `receive` action, (2a) for every event $e \in \mathsf{Events}_{\mathsf{p}}$ s.t. $\mathtt{id}(e) \leq \mathtt{id}(\mathtt{last}_{\mathtt{rep}(e)}(\pi_1(t^i)))$ there is exactly one action $a \in t^i$ s.t. $\mathsf{ev}(a) = e$, (2b) for every event $e \in \mathsf{Events}_{\mathsf{p}}$ s.t. $\mathtt{id}(e) > \mathtt{id}(\mathtt{last}_{\mathtt{rep}(e)}(\pi_1(t^i)))$ there is no action $a \in t^i$ s.t. $\mathsf{ev}(a) = e$, and (3) for every two actions $a, a' \in t$, if $\mathsf{ev}(a) \downarrow$, $\mathsf{ev}(a') \downarrow$ and $\mathtt{id}(\mathsf{ev}(a)) < \mathtt{id}(\mathsf{ev}(a'))$, then $a <_{t^i} a'$.

Let $t^0 = \mathbf{init}_{P \parallel I_E}$ be the first trace of our sequence. Clearly, t^0 satisfy properties (1), (2a), (2b) and (3). Then, let $n \in \mathbb{N}$ and, assuming that the trace t^n satisfy properties (1), (2a), (2b) and (3), we define t^{n+1}. If for every replica r and every event $e \in \mathsf{Events}_{\mathsf{p}}$, $\Delta_{\mathsf{p}}(\pi_1(t^n), e) \uparrow$, we define $t^{n+1} = t^n$. If not, let r_n be a replica and $e_n \in \mathsf{Events}_{\mathsf{p}}$ be an event s.t. $\Delta_{\mathsf{p}}(\pi_1(t^n), e_n) \downarrow$. As I_E is available, there exists an action a_n s.t. $\mathsf{op}(a'_n) \neq \mathtt{receive}$ and $\Delta_{P \parallel I_E}(t^n, a_n) \downarrow$. We then define $t^{n+1} = \Delta_{P \parallel I_E}(t^n, a_n)$.

By induction hypothesis on t^n, t^n satisfies properties (1), (2a), (2b) and (3). We show that t^{n+1} also satisfies (1), (2a), (2b) and (3). Without loss of generality, we assume that $t^{n+1} \neq t^n$ as otherwise the result immediately holds. First, as t^n satisfies (1) and a_n is not a `receive` action, t^{n+1} satisfies property (1). Properties (2a) and (2b) immediately hold from the definition of $\Delta_{P||I_E}$.

Finally, for proving that t^{n+1} satisfies (3), let $a, a' \in t^n$ be distinct actions s.t. $\mathtt{ev}(a)\downarrow$, $\mathtt{ev}(a')\downarrow$ and $\mathtt{id}(\mathtt{ev}(a)) < \mathtt{id}(\mathtt{ev}(a'))$. If $a, a' \neq a_n$, as t^n satisfies (3), $a <_{t^n} a'$ and therefore, $a <_{t^{n+1}} a'$. Otherwise, note that as t^n satisfies (2b), for every event $e \in \pi_1(t^n)$, $\mathtt{id}(e) \leq \mathtt{id}(\mathtt{ev}(a_n))$. Moreover, as no two events in $\mathsf{Events}_\mathsf{p}$ have identical identifier, traces do not contain the same event twice and $a \neq a'$, we deduce that $a' = a_n$. As $a_n = \mathtt{last}_{r_n}(t^{n+1})$, we conclude that $a <_{t^{n+1}} a'$.

By construction, t^∞ is a trace in $\mathcal{T}_{P||I_E}$. As P is finite and I_E is available, every trace $t \in \mathcal{T}_{P||I_E}$ is finite. We show by contradiction that there exists some $k \in \mathbb{N}$ s.t. $t^k = t^{k+1}$. Consider the sucession of traces $\{t^n\}_{n\in\mathbb{N}}$ and let us assume that $t^k \neq t^{k+1}$ for any $k \in \mathbb{N}$. In such case, we define t^∞ as the limit of such sucession, i.e., the trace obtained by executing events actions $a_i, 0 \leq i \leq \mathbb{N}$ (which are well-defined by construction). Such infinite trace belongs to $\mathcal{T}_{P||I_E}$. However, as P is finite and I_E is available, every trace $t \in \mathcal{T}_{P||I_E}$ is finite. Thus, t^∞ must be finite; which contradicts its construction. Hence, such k exists.

We show that the trace t^k is the searched trace. Clearly, as t^k satisfies (1) and (3), it suffices to prove that it also satisfies (2). On one hand, as $t^k = t^{k+1}$, for every event $e \in P$, $\Delta_\mathsf{p}(\pi_1(t^k), e)\uparrow$. Hence, for every replica $r_l, l \in \{0, 1\}$, $\mathtt{last}_r(\pi_1(t^k)) = e^{x_{1-l}}_{\mathsf{len}(v)}$. By construction of $\mathsf{Events}_\mathsf{p}$, every event $e \in \mathsf{Events}_\mathsf{p}$ with replica r_l has smaller identifier than $e^{x_{1-l}}_{\mathsf{len}(v)}$. Therefore, as t^k satisfies (2a), there is exactly one action $a' \in t^k$ s.t. $\mathtt{ev}(e') = e$; so t^k satisfies (2).

□

Lemma C.6. *For every pair of indices $i, -1 \leq i \leq \mathsf{len}(v)$, $l \in \{0, 1\}$,*

- *If $e^{x_l}_i$ is a read event, then $(\{w^{x_l}_i\}, e^{x_l}_i) \in \mathsf{wr}_{\tilde{x}^{x_l}_i}$.*
- *If $e^{x_l}_i$ is a write event s.t. $\mathtt{wval}(e^{x_l}_i)(\tilde{x}^{x_l}_i)\downarrow$, then $\mathtt{wspec}(e^{x_l}_i)(\mathtt{wval}(w^{x_l}_i)(\tilde{x}^{x_l}_i))\downarrow$.*

Proof. We prove the result by induction on i; where the base case, $i = -1$, trivially holds. For showing the inductive case, let us assume that the result holds for every event $e^{x_{l'}}_{i'}, -1 \leq i' < i, l' \in \{0, 1\}$, and let us show it for events $e^{x_0}_i, e^{x_1}_i$. We divide the proof in two blocks, whether $e^{x_l}_i$ is a read event, and $e^{x_l}_i$ is a write event.

For the first part, we note that by construction of ξ_v using Lemma C.5 we know that ξ_v does not contain any `receive` event, $\mathsf{rb} = \mathsf{so}$. Hence, as ξ_v is valid w.r.t. Spec, $\mathsf{wr} \subseteq \mathsf{rb} = \mathsf{so}$. Thus, $e^{x_l}_i$ reads $\tilde{x}^{x_l}_i$ from an event that precedes it in session order. In particular, by Definition 4.2, $\mathsf{wr}^{-1}_{\tilde{x}^{x_l}_i}(e^{x_l}_i) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_{\tilde{x}^{x_l}_i}(e^{x_l}_i, [\xi_v, \mathsf{CMod}])\}$; so $\mathsf{wr}^{-1}_{\tilde{x}^{x_l}_i}(e^{x_l}_i) = \{w^{x_l}_i\}$.

For the second part, we can assume w.l.o.g. that $e^{x_l}_i$ is a conditional write, as otherwise the result immediately holds. By the choice of $e^{x_l}_i$, in this case, we conclude that $\mathtt{wspec}(e^{x_l}_i)(\mathtt{wval}(w^{x_l}_i)(\tilde{x}^{x_l}_i))\downarrow$
.

□

Lemma C.7. *For every $l \in \{0, 1\}$, if $(e^{x_l}_{d_v-1}, e^{x_{1-l}}_{d_v-1}) \in \mathsf{ar}$, then $e^{x_l}_0 \in \mathsf{ctxt}_{x_l}(e^{x_l}_{\mathsf{len}(v)}, [\xi_v, \mathsf{CMod}])$.*

Proof. For proving that $e^{x_l}_0 \in \mathsf{ctxt}_{x_l}(e^{x_l}_{\mathsf{len}(v)}, [\xi_v, \mathsf{CMod}])$, we show that $v_{x_l}(e^{x_l}_0, e^{x_l}_{\mathsf{len}(v)})$ holds in ξ_v. Observe that by the choice of events and Lemma C.6 $e^{x_l}_0$ writes x_l in ξ_v and $\mathsf{wr}^{-1}_{x_l}(e^{x_l}_{\mathsf{len}(v)}) \neq \emptyset$ holds in ξ_v. Therefore, to conclude the result, we prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{Rel}^v_i$.

For proving it, we observe that CMod is in simple form. Thus, for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}^\mathsf{v}_i$ is either so, wr, rb or ar; which simplify our analysis. First, if $i = d_v$, by definition of d_v, $\mathsf{Rel}^\mathsf{v}_i = \mathsf{ar}$.

By hypothesis, $(e^{x_l}_{d_v-1}, e^{x_{1-l}}_{d_v-1}) \in \mathsf{ar}$. In such case, as $\mathtt{id}(e^{x_{1-l}}_{d_v-1}) < \mathtt{id}(e^{x_l}_{d_v})$ and $\mathtt{rep}(e^{x_{1-l}}_{d_v-1}) = \mathtt{rep}(e^{x_l}_{d_v})$, $(e^{x_{1-l}}_{d_v-1}, e^{x_l}_{d_v}) \in \mathsf{so}$. Therefore, as $\mathsf{so} \subseteq \mathsf{ar}$ and ar is a transitive relation, we deduce that $(e^{x_l}_{d_v-1}, e^{x_l}_{d_v}) \in \mathsf{ar}$.

Next, if $i \neq d_v$, we notice that $(e^{x_0}_{i-1}, e^{x_0}_{i}) \in \mathsf{so} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$. Hence, if $\mathsf{Rel}^{\mathsf{v}}_i$ is either so, rb or ar, the result immediately holds. Otherwise, if $\mathsf{Rel}^{v}_i = \mathsf{wr}$, we show that $e^{x_0}_i$ is a read event and $e^{x_0}_{i-1} = w^{x_0}_i$; which let us conclude that $(e^{x_0}_{i-1}, e^{x_0}_{i}) \in \mathsf{wr}$ thanks to Lemma C.6.

First, we show that if $i \neq \mathsf{len}(v)$ and $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, then $w^{x_l}_i = e^{x_l}_{i-1}$. Thanks to the choice of P, if $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, then $e^{x_l}_i$ is a write event s.t. $\mathtt{wval}(e^{x_l}_{i-1})(\tilde{x}^{x_l}_i) \downarrow$. By Lemma C.6, we deduce that $e^{x_l}_{i-1}$ writes $\tilde{x}^{x_l}_{i-1}$ in ξ_v. As $i \neq \mathsf{len}(v)$, $\tilde{x}^{x_l}_{i-1} = \tilde{x}^{x_l}_i$. Also, as $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, $\mathtt{rep}(e^{x_l}_i) = \mathtt{rep}(e^{x_l}_{i-1})$. Altogether, we deduce that $e^{x_l}_{i-1}$ is an event writing $\tilde{x}^{x_l}_i$ that is the immediate predecessor of $e^{x_l}_i$ w.r.t. so. Hence, $w^{x_l}_i = e^{x_l}_{i-1}$.

Finally, we show that $\mathsf{Rel}^{v}_{\mathsf{len}(v)} \neq \mathsf{wr}$ and conclude the result. We prove the contrapositive, that if $\mathsf{Rel}^{\mathsf{v}}_{\mathsf{len}(v)} = \mathsf{wr}$, v is vacuous w.r.t. Spec. If $\mathsf{Rel}^{\mathsf{v}}_{\mathsf{len}(v)} = \mathsf{wr}$, for every abstract execution ξ' valid w.r.t. Spec, object x and a collection of events $f_0, \dots f_{\mathsf{len}(v)}$, if $v_x(f_0, \dots, f_{\mathsf{len}(v)})$ holds in ξ', then $(f_{\mathsf{len}(v)-1}, f_{\mathsf{len}(v)}) \in \mathsf{wr}$. Thus, ξ' is valid w.r.t. $(\mathsf{CMod} \setminus \{v\}, \mathsf{OpSpec})$. Hence, v is vacuous w.r.t. OpSpec. □

D Proof of the Arbitration-Free Consistency Theorem

Lemmas 8.2 and D.1 prove the AFC theorem.

D.1 Arbitration-Freeness Implies Availability

The proof of (1) $\implies$ (2), essentially coincides with that of Lemma 6.6: we present an available Spec-implementation that guarantees CC. As in Lemma 6.6, CMod is arbitration-free, so by Lemma 6.4, this implies that CMod is weaker than CC. Thanks to Lemma 7.11, any implementation of CC also ensures CMod.

Lemma D.1 ((1) $\implies$ (2))**.** *Let* OpSpec *be a basic operation specification. There exists an available* (CC, OpSpec)*-implementation.*

Proof. The main difference in the construction w.r.t. the implementation shown in Lemma 6.6 corresponds to the transition function, Δ_{i}. More specifically, the main and only change arise in Equation (51), which is substituted by Equation (58).

$$\begin{aligned} M_t(e) &= [x \mapsto \mathsf{rspec}(e)(x, E_t^x(e))]_{x\in\mathsf{Objs}} \\ E_t^x(e) &= \left\{ e' \;\middle|\; \begin{array}{l} e' \in \mathsf{Events} \cap t \;\wedge\; e' \text{ writes } x \text{ in } \mathsf{exec}(t) \;\wedge \\ (\mathsf{rep}(e') = \mathsf{rep}(e) \vee \mathsf{rec}_t(e', e)) \end{array} \right\} \\ \mathsf{ar}_e^t &= \mathsf{ar}_{\restriction E_t^x(e)\times E_t^x(e)} \end{aligned} \tag{58}$$

Is immediate to show that I_E is a storage implementation. Showing that I_E is an available Spec-implementation is done as in Lemma 6.6. Observe that Lemmas C.2 and C.3 apply to this implementation; so $(S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ is an available implementation. In Lemma D.2 we show that indeed I_E is an implementation of (CC, Spec), concluding the result.

□

Lemma D.2. *The implementation I_E is an implementation of* Spec$'$ = (CC, OpSpec).

Proof. Let $P_E = (S_{\mathsf{p}}, A_{\mathsf{p}}, s_0^{\mathsf{p}}, \Delta_{\mathsf{p}})$ be a program. We prove by induction on the length of all traces in $\mathcal{T}_{P_E \| I_E}$ that any trace t is valid w.r.t. Spec$'$. The base case, when t contains a single action, is immediate as such action corresponds to the initial event, which does not read any object. Let us assume that for any trace $t' \in \mathcal{T}_{P_E \| I_E}$ of at most length k, $\mathsf{exec}(t')$ is valid w.r.t. Spec$'$; and let us show that for any trace t of length $k+1$, $\mathsf{exec}(t)$ is also valid w.r.t. Spec$'$.

Let $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be respectively the history $\mathsf{history}(t)$ and the abstract execution $\mathsf{exec}(t)$. We denote sr to the induced order between send-receive actions with the same metadata on t. For proving that ξ is valid w.r.t. Spec$'$, we first prove that ξ is indeed an abstract execution, i.e., ξ satisfies Definition 3.4. In particular, by the construction of $(S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ (compared with that of Lemma C.1), it suffices showing that $\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{rb}$.

By definition of rb, $\mathsf{so} \subseteq \mathsf{rb}$, so we focus on proving that $\mathsf{wr} \subseteq \mathsf{rb}$. Let w, r be events and x be an object s.t. $(w, r) \in \mathsf{wr}_x$. In such case, there is a pair of actions a_r, a_w s.t. $r \in a_r$, $w \in a_w$ and $w \in \mathsf{wr}\text{-}\mathtt{Set}(a_r)(x)$. Hence, $w \in \mathsf{rspec}(r)(x, E_t^x(r))$. We deduce then that $\mathsf{rec}_t(w, r)$ must hold; which implies that there exists a $\mathtt{send}$ action s and a $\mathtt{receive}$ action v s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$ and $w <_t s <_t v <_t r$. By sendAllData predicate, $w \in \mathsf{rb}\text{-}\mathtt{Set}(s)$; so by minRcv, $w \in \mathsf{rb}\text{-}\mathtt{Set}(v)$. By the induced abstract execution definition, $(w, r) \in \mathsf{rb}$.

Finally, we show that ξ is valid w.r.t. Spec$'$. By Definition 7.8, it suffices to show that for every event r and object x, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, \mathsf{ctxt}_x(r, [\xi, \mathtt{CC}]))$. Let r be a read event, x be an object and $p = \mathsf{prefix}(t, r)$. We know by Equation (58) that $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, E_t^x(r))$. Observe that by Equation (58) and rb's definition, $E_p^x(r)$ coincides with $\mathsf{ctxt}_x(r, [t, \mathtt{CC}])$. Thus, so we conclude that $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, \mathsf{ctxt}_x(r, [t, \mathtt{CC}]))$. □

D.2 Availability Implies Arbitration-Freeness

The proof of this result mimics that of Lemma 6.7. We prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod′ of CMod contains a simple visibility formula involving ar, and such formula precludes the existence of an available (CMod, OpSpec)-implementation (see Lemma 8.2).

Lemma 8.2. *Let* Spec = (CMod, OpSpec) *be a storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* $i, 0 \leq i \leq \mathsf{len}(\mathsf{v})$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* (CMod, OpSpec)*-implementation.*

Proof. We assume by contradiction that there is an available implementation I_E of Spec but CMod contains a visibility formula v non-vacuous w.r.t. OpSpec s.t. for some $i, 0 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$. We use the latter fact to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no `receive` action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

The program P we construct generalizes the litmus program presented in Figure 1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$. The events are used to "encode" two instances of v_{x_0} and v_{x_1}.

Let d_v be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to ε_{d_v} which is not arbitration-free, and the suffix from ε_{d_v} up to $\varepsilon_{\mathsf{len}(v)}$, the arbitration-free part.

For ensuring that $\mathsf{v}_x(e_0^{x_l}, \ldots e_n^{x_l})$ holds in an induced abstract execution of a trace without `receive` actions, we require that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $e_{i-1}^{x_l}$ is a write event and $e_i^{x_l}$ is a read event. For ensuring that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \ldots e_{\mathsf{len}(v)})$ holds in such abstract execution, we consider a distinct object y_E, also distinct from x_0, x_1. These objects represents each different conflict in v in an explicit manner. Intuitively, we require that events $e_i^{x_l}$ write on object y_E (resp. x_l) iff $\varepsilon_i \in E$.

More formally, we denote by $E_x \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(v)})$ to the set s.t. $\mathsf{conflict}_x(E_x) \in v$. Also, for every $i, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$, we denote by $X_i^{x_l}$ to the set containing objects y_E (resp. $\hat{x}_i^{x_l}$) iff $E \in \mathsf{conflictsOf(v, i)}$ (resp. $E_x \in \mathsf{conflictsOf(v, i)}$); where $\hat{x}_i^{x_l} = x_l$ if $i < d_v$ and x_{1-l} otherwise. We denote by X to the union of sets $X_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$.

In the construction, we require that replica r_l executes events $e_i^{x_l}$ if $i < d_v$ and events $e_i^{x_{1-l}}$ otherwise – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. We denote by $r_i^{x_l}$ to such replica.

More in detail, we construct a set of events, E^i, histories, $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$, and executions, $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i), 0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event **init**, and incorporating at each time a pair of new events, $e_i^{x_0}$ and $e_i^{x_1}$. We use the notation h^{-1} and ξ^{-1} to describe the history and abstract execution containing only **init** respectively. For simplifying notation, we use the convention **init** $= e_{-1}^{x_0} = e_{-1}^{x_1}$.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \ldots e_{i-1}^{x_0}, e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 3.4) and we construct the history h^i and the abstract execution ξ^i.

The construction of ξ^i follows the structure of that constructed in Lemma 6.7's proof, but with the technical details of that used in Theorem B.9's proof.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \ldots e_{i-1}^{x_0} e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 3.4) and we construct the history h^i and the abstract execution ξ^i.

In the following, let $l \in \{0, 1\}$.

Like in Theorem B.9, we define a pair of special objects, $\tilde{x}_i^{x_l}$ and $o_i^{x_l}$. The purpose of object $\tilde{x}_i^{x_l}$ is control the number of events in ξ that write object x_l. Equation (59) describes $\tilde{x}_i^{x_l}$; where choice is a function that deterministically chooses an element from a non-empty set. The object $o_i^{x_l}$ is an object different from objects $x, y_E, E \in \mathcal{P}(\varepsilon_0, \dots \varepsilon_{\mathsf{len}(v)})$ and $o_j^{x_{l'}}, -1 \leq j < i, l' \in \{0, 1\}$ that we use for ensuring that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $(e_{i-1}, e_i) \in \mathsf{wr}$. W.l.o.g., we can assume that $o_i^{x_0} \neq o_i^{x_1}$.

$$\tilde{x}_i^{x_l} = \begin{cases} \tilde{x}_{i-1}^{x_l} & \text{if } X_i^{x_l} = \emptyset \\ \hat{x}_i^{x_l} & \text{if } X_i^{x_l} \neq \emptyset \text{ and } \hat{x}_i^{x_l} \in X_i^{x_l} \\ \text{choice}\,(X_i) & \text{if } X_i^{x_l} \neq \emptyset \text{ and } \hat{x}_i^{x_l} \notin X_i^{x_l} \end{cases} \tag{59}$$

We select a domain $D_i^{x_l}$, a set of objects $W_i^{x_l}, W_i^{x_l} \subseteq D_i^{x_l}$ that event $e_i^{x_l}$ must write, and a set of objects $C_i^{x_l} \subseteq D_i^{x_l}$ whose value needs to be corrected for events $e_i^{x_0}, e_i^{x_1}$ in ξ_{i+1} – in the sense of Definition 7.13. We distinguishing between several cases:

- $\underline{i = 0 \text{ or } 0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{conflictsOf(v, i)} \neq \emptyset}$: In this case, we select $e_i^{x_l}$ to be a write event. If OpSpec only allows single-object atomic read-write events, we define $D_i^{x_l} = X_i^{x_l}$; while if not, we consider a domain containing $o_{i-1}^{x_l}, o_i^{x_l}$, every object in $X_i^{x_l}$ but no object from $X \setminus X_i^{x_l}$ nor objects $o_j^{x_{l'}}, 0 \leq j < \mathsf{len}(v), l' \in \{0, 1\} j \neq i - 1, i$. Observe that by Proposition B.10, such domain always exist on OpSpec.
  If there is an unconditional write event whose domain is $D_i^{x_l}$, we define $W_i^{x_l} = D_i^{x_l}$. Otherwise, we define $W_i^{x_l} = X_i^{x_l} \cup \{o_i^{x_l}\}$.
- $\underline{0 < i \leq \mathsf{len}(v), \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr} \text{ and } \mathsf{conflictsOf(v, i)} \neq \emptyset}$: In this case, by Proposition B.11, OpSpec allows atomic read-write events. If OpSpec only allows single-object atomic read-write events, we define $D_i^{x_l} = X_i^{x_l}$; while if not, we consider a domain containing o_{i-1}, o_i, every object in $X_i^{x_l}$ but no object from $X \setminus X_i^{x_l}$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i - 1, i$. Observe that by Proposition B.10, such domain always exist on OpSpec.
  Similarly to the previous case, if there is an unconditional atomic read-write event whose domain is $D_i^{x_l}$, we define $W_i^{x_l} = D_i^{x_l}$. Otherwise, we define $W_i^{x_l} = X_i^{x_l} \cup \{o_i^{x_l}\}$.
- $\underline{0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{conflictsOf(v, i)} = \emptyset}$: In this case, by Proposition B.11, OpSpec allows events that do not unconditionally write. If OpSpec allows read events that are not write events, we select $D_i^{x_l}$ to be the domain of any such event and $W_i^{x_l} = \emptyset$. Otherwise, OpSpec must allow conditional write events; so we select $D_i^{x_l}$ to be the domain of any such event, $W_i^{x_l} = \emptyset$. Observe that in this case, thanks to the assumptions on OpSpec (see Section 7.4), we can assume without loss of generality that whenever $o_{i-1}^{x_l} \in D_{i-1}$, $o_{i-1}^{x_l} \in D_i^{x_l}$ as well; while otherwise, that $\tilde{x}_{i-1}^{x_l} \in D_i^{x_l}$.

Finally we describe the event $e_i^{x_l}$ thanks to the sets $D_i^{x_l}$ and $W_i^{x_l}$. If $W_i^{x_l} = D_i^{x_l}$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, we select an unconditional atomic read-write event whose domain is $D_i^{x_l}$. If $W_i^{x_l} = D_i^{x_l}$ and $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr}$, we select an unconditional write event whose domain is $D_i^{x_l}$. If $W_i^{x_l} = \emptyset$ and OpSpec allows read events that are not write events, we select a read event whose domain is $D_i^{x_l}$. Finally, if that is not the case, we select a conditional write event $e_i^{x_l}$ s.t. $\mathsf{obj}(e_i^{x_l}) = D_i^{x_l}$ and s.t. an execution-corrector exists for $(e_i^{x_l}, W_i^{x_l}, \tilde{x}_i^{x_l}, \xi^{i-1} \oplus e_i^{x_0} \oplus e_i^{x_1})$. Such event always exists by the assumptions on operation specifications (Section 7.4). W.l.o.g. we can assume that $e_i^{x_l}$ happens on replica $r_i^{x_l}$.

For concluding the description of $h^i = (E_i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, we use an auxiliary history and abstract execution, $h_{-1}^i = (E_{-1}^i, \mathsf{so}_{-1}^i, \mathsf{wr}_{-1}^i)$ and $\xi_{-1}^i = (h_{-1}^i, \mathsf{rb}_{-1}^i, \mathsf{ar}_{-1}^i)$ respectively. For specifying wr_{-1}^i, we define the context mapping $c^i : \mathsf{Objs} \longrightarrow \mathsf{Contexts}$ in the same fashion as in Theorem B.9:

$$c_i^{x_l}(y) = (F_i^{x_l}(y), \mathsf{rb}^{i-1}_{\restriction F_i^{x_l}(y) \times F_i^{x_l}(y)}, \mathsf{ar}^{i-1}_{\restriction F_i^{x_l}(y) \times F_i^{x_l}(y)}) \tag{60}$$

where $F_i^{x_l}(y)$ is the mapping associating each object y with the set of events described below:

$$F_i^{x_l}(y) = \begin{array}{lll} \{e \in E^{i-1} \mid & \mathsf{wspec}(e)(y, [\xi^{i-1}, \mathtt{CC}]) \downarrow \text{ and } (e, e_{i-1}^{x_{1-l}}) \in (\mathsf{rb}^{i-1})^* \ \} & \text{if } i = d_v \\ \{e \in E^{i-1} \mid & \mathsf{wspec}(e)(y, [\xi^{i-1}, \mathtt{CC}]) \downarrow \text{ and } (e, e_{i-1}^{x_l}) \in (\mathsf{rb}^{i-1})^* \ \} & \text{otherwise} \end{array}$$

Then, we define ξ_{-1}^i as the abstract execution of the history $h_{-1}^i = (E_{-1}^i, \mathsf{so}_{-1}^i, \mathsf{wr}_{-1}^i)$ obtained by appending $e_i^{x_0}, e_i^{x_1}$ to h_{-1}^i and ξ_{-1}^i as follows: E_{-1}^i contains E^{i-1} and events $e_i^{x_0}, e_i^{x_1}$. First of all, we require that the relations so_{-1}^i, wr_{-1}^i, rb_{-1}^i and ar_{-1}^i contain so^{i-1}, wr^{i-1}, rb^{i-1} and ar^{i-1} respectively.

With respect to events $e_i^{x_0}, e_i^{x_1}$, we impose that $e_i^{x_l}$ is the maximal event w.r.t. so_{-1}^i among those on the same replica. Also, $e_i^{x_l}$ is maximal w.r.t. wr as we define that for every object z, $((\mathsf{wr}_{-1}^i)_z)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z))$. We also require that $\mathsf{rb}_{-1}^i = \mathsf{so}_{-1}^i$. With respect to ar_{-1}^i, we impose that $e_i^{x_0}$ succeeds every event in E^i w.r.t. ar_{-1}^i and that $e_i^{x_1}$ is the maximum event w.r.t. ar in ξ_{-1}^i.

We use ξ_{-1}^i to construct ξ^i. If event $e_i^{x_l}$ is not a conditional write event, $\xi^i = \xi_{-1}^i$. Otherwise, if event $e_i^{x_l}$ is a conditional write event, given $W_i^{x_l}$ and object $\tilde{x}_i^{x_l}$, we select an execution-corrector for $e_i^{x_l}$ w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ and $a_i^{x_l}$. W.l.o.g., we assume that every event mapped by $a_i^{x_l}$ happens on replica $r_i^{x_l}$. Observe that by the choice of sets $D_i^{x_l}$ and $W_i^{x_l}$, and thanks to the assumptions on storages (see Section 7.4), such event(s) are always well-defined.

In addition, we denote by $C_i^{x_l}$ to the set of objects we need to correct for $e_i^{x_l}$. More specifically, if $e_i^{x_l}$ is a conditional write-read, we denote by $C_i^{x_l}$ to the set of objects y s.t. $a_i^{x_l}(y)$ is defined, i.e. $C_i^{x_l} = \{y \in \mathsf{Objs} \mid a_i^{x_l}(y) \downarrow\}$. In the case $e_i^{x_l}$ is not a conditional write-read, we use the convention $C_i^{x_l} = \emptyset$. The set of events in ξ^i is the following: $E^i = E^{i-1} \cup \bigcup_{l \in \{0,1\}} (\{e_i^{x_l}\} \bigcup_{y \in C_i \setminus \{o_{i-1}^{x_l}\}} a_i^{x_l}(y))$. Observe that by the choice of $C_i^{x_l}$, the set E_i is well-defined.

From ξ_{-1}^i, we define $\xi^i = \xi_0^i \overset{\mathsf{seq}(a_i)}{\vee} e_i$ as the corrected execution of ξ and $e_i^{x_0}, e_i^{x_1}$ with events $a_i^{x_0}, a_i^{x_1}$. For describing ξ^i, we consider $<$ to be a well-founded order over Objs. ξ^i satisfies the following:

- $\underline{\mathsf{so}^i}$: Let $y \in C_i^{x_l}$. We require that for every event $e \in E^{i-1}$, $(e, a_i^{x_l}(y)) \in \mathsf{so}^i$ iff $\mathtt{rep}(e) = r_i^{x_l}, 0 \leq j < i$. We also require that $(\mathtt{\textbf{init}}, a_i^{x_l}(y)) \in \mathsf{so}^i$ and $(a_i^{x_l}(y), e_i^{x_l}) \in \mathsf{so}^i$. Finally, we require that for every objects $y' \in C_i^{x_l}, y' < y$, $(a_i^{x_l}(y'), a_i^{x_l}(y)) \in \mathsf{so}^i$.
- $\underline{\mathsf{wr}^i}$: Let y be an object in $C_i^{x_l}$. For every object z, if $z \in C_i^{x_l}$ and $z < y$, we require that $(\mathsf{wr}_z^i)^{-1}(a_i^{x_l}(y)) = \mathsf{rspec}(a_i^{x_l}(y))(z, c_i^{x_l}(z) \oplus a_i^{x_l}(z))$; while otherwise, we require that $(\mathsf{wr}_z^i)^{-1}(a_i^{x_l}(y)) = \mathsf{rspec}(a_i^{x_l}(y))(z, c_i^{x_l}(z))$. We also require that for every object z, if $z \in C_i^{x_l}$, then $(\mathsf{wr}_z^i)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z) \oplus a_i^{x_l}(z))$, while otherwise, $(\mathsf{wr}_z^i)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z))$.
- $\underline{\mathsf{rb}^i}$: Let $y \in C_i^{x_l}$. We require that for every object $y \in C_i^{x_l}$ and event e s.t. $(e, a_i^{x_l}(y)) \in \mathsf{so}^i$, s.t. $(e, a_i^{x_l}(y)) \in \mathsf{so}^i \cup \mathsf{wr}^i$, $(e, a_i^{x_l}(y)) \in \mathsf{rb}^i$.
- $\underline{\mathsf{ar}^i}$: We impose that for every event $e \in E^{i-1}$, $(e, a_i^{x_l}(y)) \in \mathsf{ar}^i, y \in C_i^{x_l}$. We also require that for every pair of objects $y_1, y_2 \in C_i$ s.t. y_1, y_2, $(a_i^{x_l}(y_1), a_i^{x_l}(y_2)) \in \mathsf{ar}^i$. As a tie-breaker between events associated to x_0 and x_1, we require that for every pair of events $e \in \{e_i^{x_0}, a_i^{x_0}(y) \mid y \in C_i^{x_0}\}$, $e' \in \{e_i^{x_1}, a_i^{x_1}(y) \mid y \in C_i^{x_0}\}$, $(e, e') \in \mathsf{ar}^i$.

We then define $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$. Observe that by construction of h^i and ξ^i, as $\mathsf{wr}^i \subseteq \mathsf{rb}^i = \mathsf{so}^i$, they satisfy Definitions 3.2 and 3.4 respectively; so they are a history and an abstract execution respectively. Also, observe that ξ^i is the corrected abstract execution of ξ_{-1}^i for events $e_i^{x_0}, e_i^{x_1}$ with events $a_i^{x_0}, a_i^{x_1}$, i.e. $\xi^i = \xi_1^i = \xi_0^i \overset{\mathsf{seq}(a_i^{x_1})}{\vee} e_i^{x_1}$, where $\xi_0^i = \xi_{-1}^i \overset{\mathsf{seq}(a_i^{x_0})}{\vee} e_i^{x_0}$.

Then, we define $\text{Events}_\text{p} = E^{\text{len}(v)}$ as the set our program employs. The set Events_p induces the set of traces $\mathcal{T}_\text{p}$.

We define the program $P = (S_\text{p}, A_\text{p}, s_0^\text{p}, \Delta_\text{p})$, where $\mathbf{init}_\text{p} = \mathbf{init}$ and Δ_p is the transition function defined as follows: for every trace $t \in \mathcal{T}_\text{p}$ and event $e \in \text{Events}_\text{p}$, $\Delta_\text{p}(t, e) \downarrow$ if and only if $e \notin t$ and every event in Events_p whose replica coincide with e and has smaller identifier than e is included in t.

The rest of the proof, which proceeds as follows, essentially combines previous results obtained while proving Lemma 6.7 and Theorem B.9:

(1) There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma C.5)
(2) The trace t induces a history $h_\text{v} = (E, \text{so}, \text{wr})$ and an abstract execution $\xi_\text{v} = (h, \text{rb}, \text{ar})$ where $\text{rb} = \text{so}$. As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\text{rb} = \text{so}$, events in ξ_v read the latests value w.r.t. so for any object. In particular, we deduce that ξ_v is valid w.r.t. $(\mathsf{CC}, \text{OpSpec})$ (Corollary D.5).
(3) Since ar is a total order, either $(e^{x_0}_{d_\text{v}-1}, e^{x_1}_{d_\text{v}-1}) \in \text{ar}$ or $(e^{x_1}_{d_\text{v}-1}, e^{x_0}_{d_\text{v}-1}) \in \text{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_\text{v}-1}, e^{x_1}_{d_\text{v}-1}) \in \text{ar}$. By Proposition D.6, we deduce that $e_0^{x_0} \in \text{ctxt}_{x_0}(e^{x_0}_{\text{len}(\text{v})}, [\xi_\text{v}, \text{CMod}])$. The proof is explained in Figure 9: if $(e^{x_0}_{d_\text{v}-1}, e^{x_1}_{d_\text{v}-1}) \in \text{ar}$, then all events $e_i^{x_0}$ form a path in such way that $\text{v}_{x_0}(e_0^{x_0}, \dots e^{x_0}_{\text{len}(\text{v})})$ holds in ξ_v. If some event $e_i^{x_l}$ is a conditional read-write event, the predicate $\text{conflict}_x(e_0^{x_0}, \dots e^{x_0}_{\text{len}(\text{v})})$ holds in ξ_v thanks to the corrector events $A_i^{x_l}$.
(4) As $e_0^{x_0} \in \text{ctxt}_{x_0}(e^{x_0}_{\text{len}(v)}, [\xi_\text{v}, \text{CMod}])$ but $(e_0^{x_0}, e^{x_0}_{\text{len}(\text{v})}) \notin \text{rb}$ (no message is received), we deduce in Proposition B.16 that OpSpec is layered w.r.t. ar. By contrapositive, if OpSpec would be layered w.r.t. rb, as $e_0^{x_0} \in \text{ctxt}_{x_0}(e^{x_0}_{\text{len}(\text{v})}, [\xi_\text{v}, \text{CMod}])$, there would exist an event e s.t. $(e_0^{x_0}, e) \in \text{rb}$ and $e \in \text{rspec}(e^{x_0}_{\text{len}(\text{v})})(x_0, [\xi_\text{v}, \text{CMod}])$. However, as $\text{rb} = \text{so}$, $\mathtt{rep}(e_0^{x_0}) = \mathtt{rep}(e) = \mathtt{rep}(e^{x_0}_{\text{len}(\text{v})})$ which is false because $\mathtt{rep}(e_0^{x_0}) = r_0$ and $\mathtt{rep}(e^{x_0}_{\text{len}(\text{v})}) = r_1$.
(5) Since rspec is maximally layered, we can show that the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v (Proposition B.17). Observe that an event writes x_0 only if it is $\mathbf{init}$ or is an event $e_i^{x_l}$ s.t. $\varepsilon_i \in E_x$ and $l = 0$. Any such index i corresponds to a suffix of v. By causal suffix closure, for any arbitration-free suffix v' of v there is a visibility formula that subsumes v' in $\text{nCMod}_\text{OpSpec}$. As d_v is the maximum index for which $\text{Rel}_i^\text{v} = \text{ar}$, the number of events writing x_0 in replica r_1 distinct from $\mathbf{init}$ coincide with the number of arbitration-free suffixes of v. Hence, as rspec is layered w.r.t. ar, if its layer bound would be greater than the number of arbitration-free suffixes, $e^{x_0}_{\text{len}(\text{v})}$ would necessarily read x_0 from $\mathbf{init}$ (other events writing x_0 are in replica r_0 and $e_{\text{len}(\text{v})}$ only reads from events in r_1). However, as rspec is maximally-layered and $e_0^{x_0}$ succeeds $\mathbf{init}$ w.r.t. ar and rb^+, we would conclude that $e^{x_0}_{\text{len}(\text{v})}$ would also read x_0 from $e_0^{x_0}$. However, this is impossible as $\text{wr} \subseteq \text{rb} = \text{so}$ but $e_0^{x_0}$ is in replica r_0 and $e^{x_0}_{\text{len}(\text{v})}$ is in replica r_1.
(6) Lastly, we show in Proposition B.18 that if the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v, then v is vacuous w.r.t. OpSpec, which contradicts the fact that v is a visibility formula from the normal form $\text{nCMod}_\text{OpSpec}$. □

Proposition D.3. *The abstract execution $\xi^{\text{len}(v)}$ described in Lemma 8.2 satisfies that for every $i, 0 \leq i \leq \text{len}(v), l \in \{0, 1\}$:*

(1) *For every object $y \in C_i^{x_l}$, the following conditions hold:*
   (a) *For every object $z \in \text{Objs}$, if $z \in C_i^{x_l}$ and $z < y$, $G(a_i^{x_l}(y), z) = F_i^{x_l}(z) \cup \{a_i^{x_l}(z)\}$, while otherwise, $G(a_i^{x_l}(y), z) = F_i^{x_l}(z)$.*
   (b) *The execution $\xi_l^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \text{OpSpec})$.*
(2) *For the event $e_i^{x_l}$, the following conditions hold:*

(a) *For every object* z*, if* $z \in C_i^{x_l}$, $G(e_i^{x_l}, z) = F_i^{x_l}(z) \cup \{a_i^{x_l}(z)\}$*, while otherwise* $G(e_i^{x_l}, z) = F_i^{x_l}(z)$.
(b) *The execution* ξ_l^i *is valid w.r.t.* $(\mathsf{CC}, \mathsf{OpSpec})$.

where $\mathsf{ctxt}_z(e, [\xi^{\mathsf{len}(v)}, \mathsf{CC}]) = (G(e, z), \mathsf{rb}_{\restriction G(e,z)\times G(e,z)}, \mathsf{ar}_{\restriction G(e,z)\times G(e,z)})$.

Proof. The proof of this result essentially coincides with that of Proposition B.12.

We prove the result by induction. In particular, we show that for every $i, -1 \leq i \leq \mathsf{len}(v)$ and object y, either (0) $i = -1$ or (1) and (2) hold. The base case, $i = -1$, is immediate as (0) holds; so let us suppose that the result holds for every $j, -1 \leq j < i$, and let us prove it for i.

For proving the inductive step, we first prove (1) for $l = 0$, then (2) for $l = 0$, and then (1) and (2) for $l = 1$. As both (1) and (2) have an identical proof (observe that the role of object y in the former is just to declare that event $a_i^{x_l}(y)$ is well-defined and the role of l is to determine which session must be proven first), we present only the proof of (1) for $l = 0$.

We show (1) by transfinite induction. Let α be an ordinal of cardinality $|\mathsf{Objs}|$. For every $k, 0 \leq k \leq \alpha$, we denote by V_k to the set containing the first k elements in Objs according to $<$. We show that (1) holds for every $y \in V_k \cap C_i^{x_0}$.

The base, V_0 is immediate as $V_0 = \emptyset$. We thus focus on the successor case (i.e., showing that if (1) holds for every object $y \in V_k \cap C_i^{x_0}$ it also holds for V_{k+1}), as the limit case is immediate: if k is a limit ordinal, $V_k = \bigcup_{i,i<k} V_i$; so (1) immediately holds. For showing that (1) holds for every object $y \in V_{k+1} \cap C_i^{x_0}$, as by induction hypothesis it holds for every object $y \in V_k \cap C_i^{x_0}$, it suffices to show it for the only object $y \in V_{k+1} \setminus V_i$. W.l.o.g., we can assume that $y \in C_i^{x_0}$; as otherwise the result is immediate.

We first prove (1a) and then we show (1b). Let $z \in \mathsf{Objs}$ be an object. Two cases arise: $z \in C_i, z < y$ or not. Both cases are identical, so we present the former, i.e., if $z \in C_i^{x_0}, z < y$, then $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} = G(a_i^{x_0}(y), z)$.

For proving that $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} \subseteq G(a_i^{x_0}(y), z)$, we distinguish whether $i = d_v$ or not. However, the proof essentially coincides in both cases, so we present the case $i = d_v$. We split the proof in two blocks: showing that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$ and showing that $a_i^{x_0}(z) \in G(a_i(y), z)$.

For showing that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$, let e be an event in $F_i^{x_0}(z)$. In such case, $e \in E^{i-1}$, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^i)^*$. By the construction of ξ, it is easy to see that any such event belongs to E^i, $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^{\mathsf{len}(v)})^*$. As $i = d_v$, we deduce that $(e_{i-1}^{x_1}, a_i^{x_0}(y)) \in \mathsf{rb}^i \subseteq \mathsf{rb}^{\mathsf{len}(v)}$. Hence, $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$; so $e \in G(a_i^{x_0}(y), z)$. This show that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$.

For showing that $a_i^{x_0}(z) \in G(a_i^{x_0}(y), z)$, we observe that $\xi_0^i = \xi_{-1}^i \overset{a_i^{x_0}}{\vee} e_i^{x_0}$. We note that as $z < y$, by induction hypothesis (1b), $\xi_0^i \restriction z$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Thus, by Property 1 of Definition 7.13, $\mathsf{wspec}(a_i(z))(z, [\xi_0^i, \mathsf{CC}]) \downarrow$. Hence, $\mathsf{wspec}(a_i^{x_0}(z))(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $\mathsf{wspec}(a_i^{x_0}(z))(z, [\xi^{\mathsf{len}(v)}, \mathsf{CC}]) \downarrow$. As $z < y$, $(a_i^{x_0}(z), a_i^{x_0}(y)) \in \mathsf{so}^i \subseteq \mathsf{so}^{\mathsf{len}(v)}$; so we conclude that $a_i^{x_0}(z) \in G(a_i^{x_0}(y), z)$.

We conclude the proof of the inductive step of (1a) by showing the converse i.e. $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} \supseteq G(a_i^{x_0}(y), z)$. Let $e \in G(a_i^{x_0}(y), z)$. First of all, by the definition of Causal visibility formula (see Figure 4b), $e \in G(a_i^{x_0}(y), z)$ iff $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$. Observe that if $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$, by construction of $\xi^{\mathsf{len}(v)}$, such event must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$. We prove that if $e \in E^{i-1}$ then $e \in F_i^{x_0}(z)$, while otherwise, if $e \in E^i \setminus E^{i-1}$, then $e = a_i^{x_0}(z)$.

If $e \in E^{i-1}$, as $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$, $\mathsf{wspec}(e)(z, [\xi^{i-1}, \mathsf{CC}]) \downarrow$. Also, as $i = d_v$ and $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^i)^+$, we deduce that $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^{i-1})^*$. In other words, $e \in F_i^{x_0}(z)$.

Otherwise, if $e \in E^i \setminus E^{i-1}$, we note that by construction of $\xi^{\mathsf{len}(v)}$, the only events in $E^i \setminus E^{i-1}$ s.t. $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^i)^+$ are events $a_i^{x_0}(w), w \in C_i, w < y$. As $\xi_0^i = \xi_{-1}^i \overset{\mathsf{seq}(a_i^{x_0})}{\curlyvee} e_i^{x_0}$ and $z < y$, $\xi_0^i \upharpoonright z$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Hence, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ iff $\mathsf{wspec}(e)(z, [\xi_0^i, \mathsf{CC}]) \downarrow$. Thus, by Property 1 of Definition 7.13 we conclude that $e = a_i^{x_0}(z)$.

For concluding the inductive step, we show that (1b) holds. This is immediate by the definition of wr^i: for every event $e \in \xi^i \upharpoonright y$, by induction hypothesis (1a) or (2a) – depending on whether $e = e_j^{x_{l'}}$ or $a_j^{x_{l'}}(w)$, where $0 \leq j \leq i, w \in C_i, l' \in \{0, 1\}$ – $(\mathsf{wr}^i)_z^{-1}(e) = \mathsf{rspec}(e)(\mathsf{CC}, [\xi_{l'}^j \upharpoonright y, z]) = \mathsf{rspec}(e)(\mathsf{CC}, [\xi_l^i \upharpoonright y, z])$. Thus, $\xi^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. □

A consequence of Proposition D.3 is the following result.

Corollary D.4. *The abstract execution ξ described in Lemma 8.2 is valid w.r.t.* $(\mathsf{CC}, \mathsf{OpSpec})$.

Corollary D.5 is an immediate result from Corollary D.4, obtained by simply observing that $\mathsf{rb}^{\mathsf{len}(v)} = \mathsf{so}^{\mathsf{len}(v)} = \mathsf{so} = \mathsf{rb}$.

Corollary D.5. *The abstract execution ξ_v described in Lemma 8.2 is valid w.r.t.* $(\mathsf{CC}, \mathsf{OpSpec})$.

Proposition D.6. *For every $l \in \{0, 1\}$, if $(e_{d_v-1}^{x_l}, e_{d_v-1}^{x_{1-l}}) \in \mathsf{ar}$, then the predicate $v_{x_0}(e_0^{x_l}, \ldots e_{\mathsf{len}(v)}^{x_l})$ holds in the abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ described in Theorem B.9.*

Proof. The proof of this result essentially coincides with that of Proposition B.14.

The proof is a simple consequence of $\xi^{\mathsf{len}(v)}$'s construction. To show that $v_{x_0}(e_0^{x_l}, \ldots e_{\mathsf{len}(v)}^{x_l})$ holds in ξ, we first show that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$ and to then prove that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0^{x_l}, \ldots e_{\mathsf{len}(v)}^{x_l})$ holds in ξ.

We prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$. Four cases arise depending on $\mathsf{Rel}_i^{\mathsf{v}}$.

- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{so}}$: In this case, by construction of events $e_{i-1}^{x_l}, e_i^{x_l}$, we know that $r_i^{x_l} = r_{i-1}^{x_l}$. Hence, $\underline{(e_{i-1}^{x_l}, e_i^{x_l})} \in \mathsf{so}^i \subseteq \mathsf{so}$.
- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}}$: In this case, we first show that there is an object $y \in D_i^{x_l} \cap W_{i-1}^{x_l} \setminus C_i^{x_l}$, and then show that $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{wr}_y$. For showing the first part, we distinguish between cases depending on whether $o_{i-1}^{x_l} \in D_i^{x_l}$ or not.
  - $\underline{o_{i-1}^{x_l} \in D_i^{x_l}}$: In this sub-case, we show that $y = o_{i-1}^{x_l}$. On one hand, if $\mathsf{conflictsOf(v, i)} = \emptyset$, by the choice of event $e_i^{x_l}$, $o_{i-1}^{x_l} \in D_{i-1}^{x_l} \setminus C_i^{x_l}$. On the other hand, if $\mathsf{conflictsOf(v, i)} \neq \emptyset$, as $o_{i-1}^{x_l} \in D_i^{x_l}$, we deduce that OpSpec allows multi-object read-write events. Observe that as v is conflict-maximal, $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$. Hence, as OpSpec allows multi-object read-write events, we deduce that $o_{i-1}^{x_l} \in D_{i-1}^{x_l} \setminus C_i^{x_l}$. In both cases, as $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$ and $o_{i-1}^{x_l} \in D_{i-1}^{x_l}$, by the choice of $W_{i-1}^{x_l}$, we conclude that $o_{i-1}^{x_l} \in W_{i-1}^{x_l}$.
  - $\underline{o_{i-1}^{x_l} \notin D_i^{x_l}}$: In this case, we show that $y = \tilde{x}_i^{x_l}$. On one hand, if $\mathsf{conflictsOf(v, i)} = \emptyset$, $X_i^{x_l} = \emptyset$; so by the choice of $\tilde{x}_i^{x_l}$ (see Equation (59)), $\tilde{x}_i^{x_l} = \tilde{x}_{i-1}^{x_l}$. By the choice of $D_i^{x_l}$, $\tilde{x}_{i-1}^{x_l} \in D_i^{x_l} \setminus C_i^{x_l}$. Moreover, as v is conflict-maximal, $\mathsf{conflictsOf(v, i-1)} \neq \emptyset$; so $\tilde{x}_{i-1}^{x_l} \in X_{i-1}^{x_l}$. By the choice of event $e_{i-1}^{x_l}$, $X_{i-1}^{x_l} \subseteq W_{i-1}^{x_l}$. Altogether, we conclude that $\tilde{x}_i^{x_l} \in W_{i-1}^{x_l}$.
    On the other hand, if $\mathsf{conflictsOf(v, i)} \neq \emptyset$, we note that $\tilde{x}_i^{x_l} \in D_i^{x_l} \setminus C_i^{x_l}$. As $o_{i-1}^{x_l} \notin D_i^{x_l}$, we deduce that OpSpec only allows single-object read-write events. Thus, $D_i^{x_l} = \{\tilde{x}_i^{x_l}\}$. As v is conflict-maximal, we deduce that $X_i^{x_l} \subseteq X_{i-1}^{x_l}$. As by the choice of $e_{i-1}^{x_l}$, $X_{i-1}^{x_l} \subseteq W_{i-1}^{x_l}$, we conclude that $\tilde{x}_i^{x_l} \in W_{i-1}^{x_l}$.

  We prove now that $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{wr}_y$. First, we show that $e_{i-1}^{x_l}$ writes y in ξ. On one hand, if $e_{i-1}^{x_l}$ is an unconditional write event, $\mathsf{wspec}(e_{i-1}^{x_l})(y, c_i^{x_l}(y)) \downarrow$. On the other hand, if $e_{i-1}^{x_l}$ is a conditional write event, as ξ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$ (Corollary D.5) and $y \in W_i^{x_l}$, by

Property 2 of Definition 7.13, we deduce that $\mathsf{wspec}(e^{x_l}_{i-1})(y, c^{x_l}_i(y)) \downarrow$. Then, as $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, $i \neq d_v$, so $e^{x_l}_{i-1} \in F^{x_l}_i(y)$. Observe that by construction of ξ, $e^{x_l}_{i-1}$ is the *so*-maximum event in $c^{x_l}_i(y)$. As every event in $F^{x_l}_i(y)$ is so-related, we deduce that $e^{x_l}_{i-1}$ is the ar-maximum event in $F^{x_l}_i(y)$. We note that as $y \notin C^{x_l}_i$, by Proposition D.3, $F^{x_l}_i(y) = G(e^{x_l}_i, y)$. Altogether, $e^{x_l}_{i-1}$ is the ar-maximum event in $\mathsf{ctxt}_y(e^{x_l}_i, [\xi^{\mathsf{len}(v)}, \mathsf{CC}])$. As $\mathsf{rb}^{\mathsf{len}(v)} = \mathsf{rb}$, we conclude that $e^{x_l}_{i-1}$ is the ar-maximum event in $\mathsf{ctxt}_y(e^{x_l}_i, [\xi, \mathsf{CC}])$. As rspec is maximally layered, we deduce that $e^{x_l}_{i-1} \in \mathsf{rspec}(e^{x_l}_i)(y, [\xi, \mathsf{CC}])$. Finally, as ξ is valid w.r.t. CC (Corollary D.5), we conclude that $(e^{x_l}_{i-1}, e_i) \in \mathsf{wr}_y$.

- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{rb}}$: In this case, $i \neq d_v$. Then, $\mathsf{rb} = \mathsf{so}$ and $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{so}$, we conclude that $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{rb}$.
- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}}$: On one hand, if $i = d_v$, by hypothesis, $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{ar}$. On the other hand, if $i \neq d_v$, $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{so}$. Thus, $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{ar}$.

For showing that show that $\mathsf{wrCons}^{\mathsf{v}}_x(e_0, \ldots e_{\mathsf{len}(v)})$, we show that for every $i, 0 \leq i \leq \mathsf{len}(v)$ and every set $E \in \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i})$, the event $e^{x_l}_i$ writes on object y_E[9]. If $e^{x_l}_i$ is an unconditional write, by the choice of $e^{x_l}_i$, it writes on every object in $D^{x_l}_i$. As $y_E \in D^{x_l}_i$, we conclude that $e^{x_l}_i$ writes on y_E. Otherwise, if $e^{x_l}_i$ is a conditional write, we observe that $y_E \in W^{x_l}_i$. Hence, as $\xi^i_0 = \xi^i_{-1} \overset{\mathsf{seq}(a^{x_0}_i)}{\vee} e^{x_0}_i$ and ξ^i_0 is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$ (resp. $\xi^i_1 = \xi^i_0 \overset{\mathsf{seq}(a^{x_1}_i)}{\vee} e^{x_1}_i$ and ξ^i_1 is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$) (Proposition D.3), we deduce using Property 2 of Definition 7.13 that $\mathsf{wspec}(e^{x_l}_i)(y_E, [\xi^i, \mathsf{CC}]) \downarrow$. By construction of ξ, we conclude that $\mathsf{wspec}(e^{x_l}_i)(y_E, [\xi, \mathsf{CC}]) \downarrow$. □

Received 2025-07-10; accepted 2025-11-06

[9]For simplifying the proof, we abuse of notation and say that $y_E = x_l$ if $E = E_x$. Observe that v is conflict-maximal, either $\mathsf{conflict}_x(E_x)$ or $\mathsf{conflict}(E_x)$ do not belong to v.