From Sequential Specifications to Eventual Consistency

Radha Jagadeesan and James Riely

DePaul University

Abstract We address a fundamental issue of *interfaces* that arises in the context of cloud computing. We define what it means for a replicated and distributed implementation to satisfy the standard sequential specification of the data structure. Several extant implementations of replicated data structures already satisfy the constraints of our definition. For example, all of the algorithms discussed in a recent survey of convergent or commutative replicated datatypes [16] satisfy our definition. We show that our definition simplifies the programmer task significantly for a class of clients who conform to the CALM principle [9].

1 Intro

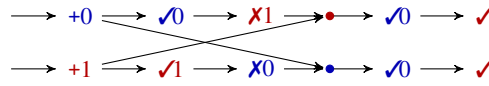
An example serves to motivate the problem addressed in this paper. Consider an integer set interface with mutator methods add and remove and a single, boolean-valued accessor method get. We will assume that mutators do not return values (have return type Unit or void) and that accessors do not alter the state of the object. The sequential behavior of such a set can be defined as a set of strings such as ✗0 +0 ✓0 ✗1 and +0 +1 ✓0 ✓1 -1 ✓0 ✗1, where $+u$ represents a call to add with argument u, $-u$ represents remove(u), ✓u represents get(u) returning true and ✗u represents get(u) returning false. Since accessor methods do not alter the state of the object, the interface is closed under commutation of accessors: if (s ✓0 ✗1) is a valid traces in the interface, for some s, then so is (s ✗1 ✓0).

Consider the implementation of such a set as a cloud service that is implemented by replication of the data structure (eg. see [16]). In this distributed setting, we assume intra-node atomicity and sequencing of state transitions, whereas temporal relations between two computers that are distributed is only induced by the receipt of messages over the network. In this distributed context, there are two impediments to requiring the replicas to achieve consensus on a global total order [12] on the operations on the data structure. Firstly, the associated serialization bottleneck negatively affects performance and scalability (eg, see [5]). Secondly, the CAP theorem [7] imposes a tradeoff between consistency and partition-tolerance.

This has led to the emergence of alternative approaches based on *eventual consistency* and *optimistic replication* [15], [18]. In such approaches, a replica may execute an operation without synchronizing with other replicas. The other replicas are updated asynchronously with the update operation. However, due to the vagaries of the network, even if every replica eventually receives and applies all updates, it could happen in possibly different orders. So, there has to be some mechanism to reconcile conflicting updates (for illustrative examples, see [16], [17]). Thus, such approaches address the issue of efficiency (since any query to the state of the data structure at a replica

is answered locally at the replica without any consensus overhead) and data remains available even in the presence of network partitions.

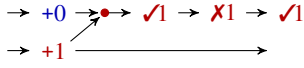
The literature on convergent or commutative replicated datatypes (CRDTs) (see [16] for a survey) provides a systematic attempt to design such datastructures. Consider the following diagram, in the style of [16].



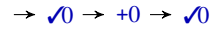
(1)

In this sample execution, the mutators +0 and +1 are executed at distinct replicas. The actions in each replica are temporally ordered from left to right, as indicated by the horizontal arrows. We assume the local updates are atomic. After a local update, the replica forwards messages to the other replicas; in the diagram, the diagonal arrows between replicas indicate messages that propagate such local updates, with the interpretation that the operation is guaranteed to be finished at the recipient at the point the arrow appears on the recipients timeline. The accessors are executed locally and atomically at each replica. Of course, there is a consistent global state, testified by ✓0 and ✓1 at both replicas, after both messages have been delivered. Thus, the literature (eg. see [16] for a precise formalization) deems this implementation to be eventually consistent, since the states of all the replicas eventually converge at quiescent points, when all the messages have been delivered. This view is adequate for examples where we are interested only in the final state of the data structure. For example, in a shopping cart, the client is arguably only interested in the state of the cart at the moment of checkout.

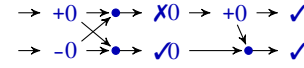
Since eventual consistency only speaks about the quiescent points of the system, it does not address correctness of intermediate states in the evolution of the system. For example, all of the following implementation traces of a putative replicated set are deemed to be eventually consistent, even though we see very problematic behavior.



(2)



(3)



(4)

In figure (2), the accessor results regresses from ✓1 to ✗1 even though there is no remove invocation in the system; in figure (3), the initial accessor ✓0 is not justified; in figure (4), the replicas conflict in their ordering of concurrent add/remove updates.

This problem is addressed by the seminal work of [2]. The key idea in [2] is to view the interface of a replicated data structure as a *concurrent* specification that determines the valid result of an accessor from the context of a prior concurrent history. [1] extends this approach to allow for bounded rollbacks. In this style, the above examples are *declared* invalid; for example in figure (2), the result ✗1 is deemed invalid in the context of its prior history.

In addition to capturing the properties of replicated implementations much more precisely than the traditional definitions of eventual consistency, this line of work has also lead to useful tools and techniques to aid the programmer: [8] proves abstraction and composition theorems, applying it in particular to the replicated implementation of

a graph data structure; [1] develops model checking techniques to reason about implementations relative to these specifications.

In this approach, replicated data structures are specified directly, without any formal comparison to the sequential data structures that they are meant to approximate. This approach is (intentionally) agnostic to the design of the specifications themselves. For example, whereas the result ✗1 in figure (2) is not valid, the result ✗1 in figure (1) is valid. The justification for the different decisions about ✗1 in figures (1) and (2) is the traditional *sequential* specification of Set; namely, if there are no remove operations, ✓1 is acceptable iff there is a preceding +1.

In this paper, we provide a definition of eventual consistency that develops precisely such a connection with the sequential specification[1]. We show the utility of our definition by showing that clients satisfying the CALM principle (see [9] for a survey) can in fact abstract away completely from the distributed and replicated implementation and program against the sequential specification realized by the implementation.

Our work is complementary to the research program of [1], [2], [8]. Our methods provide a way to justify the concurrent interfaces described in their approach. In future work, we hope to use our methods to show that CALM clients of their interfaces can also be protected from details of distribution and replication. We also hope to adopt their methods to support a more general class of clients.

An informal outline of our approach. In a replicated data structure, a mutator m is *visible* to an event a if m executes at a's replica before a executes. We say that an implementation trace (such as those in the figures above) satisfies a sequential specification if for each event a, we can associate a string of events $\mathtt{t}(a)$ that satisfies the following.

Mutator closed: $\mathtt{t}(a)$ includes a as well as the mutator events that are visible to a
Validity: $\mathtt{t}(a)$ is a valid sequential trace that ends in a
History consistency: For any events d and e, $\mathtt{t}(d)$ and $\mathtt{t}(e)$ agree on the ordering of mutator events that are visible to both d and e.

Figure (2) does not satisfy validity at the event ✗1 in the top replica since neither +0 +1 ✗1 nor +1 +0 ✗1 is a valid trace of a set. Figure (3) does not satisfy validity at the initial event ✓0 since the trace ✓0 is not a valid trace of a set. In Figure (4), to satisfy validity, we have to associate the trace −0 +0 at the ✓0 event in the top replica and the trace +0 −0 at the ✗0 event in the bottom replica, thus violating history-consistency.

For a positive example, in figure (1), the traces associated to the event ✗1 in the top replica is +0 and the trace associated to the event ✗0 in the bottom replica is +1. There is a choice for the trace associated with the events ✓1 in the top replica and ✓0 in the bottom replica. By consistency, they need to be the same, but they can both be chosen to be either +0 +1 or +0 +1.

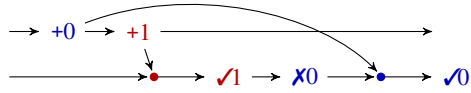
Our definition is flexible enough to accommodate all the data structures discussed in [16], a recent survey of the literature on CRDTs. Such data structures provide a particularly simple programming view for clients located at a replica. In a logically monotone

[1] Traditional criteria, such as linearizability [10] or quiescent consistency [4], [11], do not apply here. For example, figure (1) is considered eventually consistent; however, there is no sequential trace of a set that has this collection of actions in the required order, since we have both +0 ⟶ ✗1 and +1 ⟶ ✗0 .

execution, the arrival time of a concurrent mutator does not alter the evolution of the system[2]. We formalize a weaker monotonicity property: that there is *some* ordering of concurrent mutators that does not alter the evolution of a system. Under this weak monotonicity assumption (that is satisfied by all CALM executions), we prove abstraction [6] and composition [10] theorems. This is particularly relevant because it simplifies the programmer perspective for a large class of programs that includes those written in languages that realize the CALM principle, such as Bloom [3].

2 Bracketed Partial Orders

In this section, we define bracketed partial orders (BPOs). BPOs provide a formalization of diagrams such as those given in the introduction. BPOs are labelled partial orders, enriched with *replicas* and *bracketing*. Bracketing relates the remote execution of a mutator to the initial call of the mutator. Consider the following example.



(5)

This is formalized as a BPO with seven events. There are two replicas: one for each horizontal line. The partial order is given by the arrows. Two events are labelled as mutators: +0 and +1. Three events are labelled as accessors: ✓1, ✗0 and ✓0. The remaining two events (shown without labels in the diagram) are bracketing events. In the formalism, bracketing events are labelled with the name of the preceding mutator event. Generally one is interested in the isomorphism class of labelled partial orders (the pomset), and therefore the event names themselves are uninteresting.

A BPO is *causal* if the order of mutator and bracketing events at each replica respects the partial order of the mutator events themselves. All of the figures in the introduction are causal. Figure (5) is not causal, however, since the mutator order is +0 +1 but the order at the bottom replica is +1 +0.

BPOs directly capture the notion of an *operation-based* CRDT (see [16]). *State-based* CRDTs can be considered a special case of causal BPOs that communicate multiple brackets with a single communication (modelled as an uninterrupted sequence of bracketed events at the receiving replica).

Let $\mathscr{A}$ and $\mathscr{M}$ be disjoint sets of *accessor labels* and *mutator labels*, respectively, and let $\mathscr{L} = \mathscr{A} \cup \mathscr{M}$ be a set of *labels*. We use metavariables s–v to range over various types of relations with labels in $\mathscr{L}$, which we generically refer to as "traces".

Example 2.1. In this paper we consider four implementations of an integer set datatype: the *G-set*, *U-set*, *OR-set*, and *2P-set*. See [16] for implementation details.

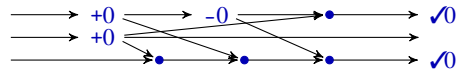
A G-set has mutator labels of the form $+u$, where u is an integer, and accessor labels of the form ✓u and ✗u. A G-set is *grow only*; thus, once ✓u has been observed for a

[2] Our formalization of logically monotone executions is inspired by ideas in [13], [14], where a monotone node is insensitive to the arrival order of the inputs. Further, a concurrent input action (mutator) does not disable an output action (accessor) at a monotone node.

particular u, it is impossible to subsequently observe $\text{✗}u$. It is straightforward to specify the replicated implementation and, therefore, the corresponding BPO.

A U-set adds mutators of the form $-u$ to the labels of a G-set, denoting removal. A U-set requires that for every u, $+u$ may appear at most once in each execution—each $+u$ is *unique*. This requirement is imposed on the *client* of the U-set; it is not ensured by the U-set itself. The implementation is again straightforward. The client can guarantee uniqueness using various techniques; for example, take $u = 2^c \cdot 3^n$ where c is a globally unique client thread identifier and n is a monotone thread-local counter.

An OR-set (observed-remove set) has the same labels as a U-set, but does not require that $+u$ actions are unique. The implementation uses an underlying U-set and a map from the elements of the U-set to the elements of the OR-set. Consider the following BPO, from [16].



(6)

This BPO is not a valid execution of a G-set (because of the -0) or a U-set (because of the two +0's). However, this is a valid execution of an OR-set. The +0 in the middle replica is concurrent with the -0 of the top replica. Since they are working on top of an underlying U-set, the -0 only removes the +0 added by the top replica; the middle +0 is not affected and eventually prevails.

A 2P-set is implemented using two grow sets; one representing additions and one representing tombstones for removed elements, in the obvious way. Like a U-set, a 2P-set also constrains the behaviour of clients. A client must ensure that no element that is removed is subsequently re-added. The BPO in figure (6) is a valid 2P-set BPO if the events labeled ✓0 are re-labeled to ✗0. In a OR-set, an add "wins" over a concurrent remove, whereas in a 2P-set, the remove wins. Thus these two examples represent different specializations of the set API. The OR-set resolves figure (6) to the sequential specification +0 -0 +0 ✓0 ✓0, whereas the 2P-set resolves it to +0 +0 -0 ✗0 ✗0.

The constraints on the clients of U-set and 2P-set are required for correct functioning *as a set*. The definition of *correctness* is given informally in [16]. The main contribution of this paper is to provide a formalization, which we do in section 4. □

Definition 2.2. A *(replicated) bracketed partial order (BPO)* is a octuple $\langle E_\mathsf{A}, E_\mathsf{M}, E_\mathsf{B}, L, R, \lambda, \rho, \Rightarrow \rangle$ where R is a set of replicas, and the following hold.

(a) sets E_A, E_M and E_B are disjoint, $L \subseteq \mathscr{L}$, and $\langle E_\mathsf{A} \cup E_\mathsf{M} \cup E_\mathsf{B}, \Rightarrow \rangle$ is a partial order,
(b) $\rho \in (E_\mathsf{A} \cup E_\mathsf{M} \cup E_\mathsf{B}) \mapsto R$ and $\lambda \in (E_\mathsf{A} \mapsto L \cap \mathscr{A}) \cup (E_\mathsf{M} \mapsto L \cap \mathscr{M}) \cup (E_\mathsf{B} \mapsto E_\mathsf{M})$,
(c) $\forall e \in E_\mathsf{B}.\ \lambda(e) \Rightarrow e$ and $\rho(\lambda(e)) \neq \rho(e)$
(d) $\forall d, e \in E_\mathsf{B}.$ if $\lambda(d) = \lambda(e)$ then either $d = e$ or $\rho(d) \neq \rho(e)$
(e) $\forall d, e \in E.$ if $\rho(d) = \rho(e)$ then either $d \Rightarrow e$ or $e \Rightarrow d$.

For a BPO s, we write $\mathsf{E}_\mathsf{A}(s)$ for the accessor events of s, $\mathsf{E}_\mathsf{M}(s)$ for the mutator events and $\mathsf{E}_\mathsf{B}(s)$ for the bracketing events. We also define $\mathsf{E}_\mathsf{AM}(s) \triangleq \mathsf{E}_\mathsf{A}(s) \cup \mathsf{E}_\mathsf{M}(s)$. □

Condition (b) establishes the interpretation of the labelling function: The elements of E_A denote local events (accessors), the elements of E_M denote the origination of a global event (mutators), and the elements of E_B denote the remote reception of a global event (brackets). Events $m \in E_\mathsf{M}$ and $b \in E_\mathsf{B}$ are a *bracketed pair* when $\lambda(b) = m$.

Condition (c) establishes that in a bracketed pair, the beginning must precede the end and occur at a separate replica. Condition (d) establishes that each mutator is bracketed at most once per replica. Thus, each mutator event has one "beginning" and as many as $\left|R\right| - 1$ "endings". Condition (e) establishes that events are totally ordered at each replica; concurrency within a replica can be handled via standard means.

Definition 2.3 (Causal). Let s be an BPO. Define $\mathsf{remote}_s(e) \triangleq \{b \in \mathsf{E_B}(s) \mid \lambda_s(b) = e\}$. The BPO s is *causal* when $\forall d, e \in \mathsf{E_M}(s).\ \forall d' \in \mathsf{remote}_s(d).\ \forall e' \in \mathsf{remote}_s(e).$ if $d \Rightarrow_s e$ and $\rho_s(d') = \rho_s(e')$ then $d' \Rightarrow_s e'$. □

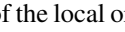
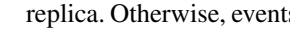
3 Labeled visibility relations and orders

BPOs have a clear operational intuition. In this section we provide an abstract view of BPOs which is sufficient to define correctness. The relations we need are weaker than labeled partial orders. In particular, we do not require transitivity. We refer to these potentially intransitive relations as *labeled visibility relations* (LVRs). For example, starting with the BPO given in figures (5) and (6), we derive the following LVRs.

In these diagrams, we use $\rightsquigarrow$ to represent an intransitive edge and $\rightarrow$ to represent a "transitive" edge. Thus, in the left diagram, the event ✗0 sees +1 and ✓1, but not +0, whereas ✓0 sees all four prior events. Recall from figure (5) that the replica that generates ✗0 sees +1 before +0, even though these are initiated in the reverse order. A causal BPO generates a transitive LVR, as in the right diagram above. Formally, LVRs are defined with a single visibility relation, which may or may not be transitive. We include replica identifiers to define liveness properties; we ignore them except when important.

Definition 3.1. Let $s = \langle E, L, R, \lambda, \rho, \rightsquigarrow \rangle$ be a sextuple such that E is a finite set of events, L is a set of labels, R is a set of replicas, $\lambda \in (E \mapsto L)$, $\rho \in (E \mapsto R)$ and $\rightsquigarrow \subseteq (E \times E)$. We say that s is a *labeled visibility relation (LVR)* if $\rightsquigarrow$ is reflexive and acyclic[3]. We say that s is a *labeled partial order (LPO)* if $\rightsquigarrow$ is a partial order. We say that s is a *labeled total order (LTO)* if $\rightsquigarrow$ is a total order.

Given an LVR s, we write $\mathsf{E}(s)$ for the event set of s, $\mathsf{L}(s)$ for the label set, λ_s for the labeling function and $\rightsquigarrow_s$ for the visibility relation. Define $\mathsf{E_A}(s) \triangleq \{e \in \mathsf{E}(s) \mid \lambda(e) \in \mathscr{A}\}$ and $\mathsf{E_M}(s) \triangleq \{e \in \mathsf{E}(s) \mid \lambda(e) \in \mathscr{M}\}$. □

Below, we define the translation from BPOs to LVRs. For a BPO s, the relation $\xRightarrow{\text{local}}_s$ is the union of the local orders at each replica. Whenever $d \xRightarrow{\text{local}}_s e$, we have that $d \rightsquigarrow_s e$. For mutators m and accessors a, we have that $m \rightsquigarrow a$ if m has been received at a's replica. Otherwise, events d and e at different replicas are ordered when they are ordered by $\Rightarrow_s$ and every mutator visible to d is also visible to e. The BPO  translates to the LVR  , which we draw as $m \rightarrow a \rightarrow n \rightarrow b$. The BPO  translates to the LVR $m \rightarrow a \rightarrow n \rightsquigarrow b$.

For a BPO s, we have that $\forall m \in \mathsf{E_M}(s).\ \forall a, b \in \mathsf{E_A}(s).$ if $m \rightsquigarrow_s a \rightsquigarrow_s b$ then $m \rightsquigarrow_s b$.

[3] A relation is acyclic if its transitive closure is anti-symmetric.

Definition 3.2. For any sets $C \subseteq A$ and relation $\mathbf{R} \subseteq A \times A$, define $\mathbf{R} \setminus C \triangleq \mathbf{R} \cap (C \times C)$. Similarly, for $\mathbf{R} \subseteq A \times B$ and $C \subseteq A$, define $\mathbf{R} \setminus C \triangleq \mathbf{R} \cap (C \times B)$.

Let s be a BPO. Define $(d \xrightarrow{\mathsf{local}}_s e) \triangleq (d \Rightarrow_s e)$ and $(\rho_s(d) = \rho_s(e))$. Recall Definition 2.3 of remote. Define $\mathsf{visM}_s(e) \triangleq \{m \in \mathsf{E_M}(s) \mid m \xrightarrow{\mathsf{local}}_s e \text{ or } \exists b \in \mathsf{remote}_s(m).\ b \xrightarrow{\mathsf{local}}_s e\}$. Then we define the LVR derived from s as follows: $\mathsf{lvr}(s) \triangleq \langle \mathsf{E_{AM}}(s), \mathsf{L}(s), \mathsf{R}(s), \rho_s, \lambda_s \setminus \mathsf{E_{AM}}(s), \rightsquigarrow \rangle$ where $\forall d, e \in \mathsf{E_{AM}}(s).\ d \rightsquigarrow e$ iff $d \in \mathsf{visM}_s(e)$ or $d \Rightarrow_s e$ and $\mathsf{visM}_s(d) \subseteq \mathsf{visM}_s(e)$. We write $\rightsquigarrow_s$ for the visibility relation of $\mathsf{lvr}(s)$. □

In a strongly distributed BPO, events at different replicas are only ordered via bracketed pairs; this disallows synchronization between replicas outside of the data structure formalized by the BPO.

Definition 3.3. A BPO is *strongly distributed* if $\forall d, e \in E_\mathsf{A} \cup E_\mathsf{M} \cup E_\mathsf{B}$. if $\rho(d) \neq \rho(e)$ and $d \Rightarrow e$ then $\exists d' \in E_\mathsf{M}, e' \in E_\mathsf{B}.\ \lambda(e') = d'$ and $d \Rightarrow d' \Rightarrow e' \Rightarrow e$ □

Lemma 3.4. Let s be a strongly distributed BPO. *Then the following three statements are equivalent: (a) s is causal, (b) $(\rightsquigarrow_s)$ is transitive, and (c) $(\rightsquigarrow_s) = (\Rightarrow_s \setminus \mathsf{E_{AM}}(s))$.* □

4 Eventual consistency

Definitions of eventual consistency (EC) traditionally include both safety and liveness properties. Liveness is purely a property of implementations. It can be expressed as a simple closure property over sets of LVRs, which we call *eventual delivery*[4].

To define safety, we must first define specifications (Definition 4.1) and give some basic vocabulary for permutations, order extensions and the like (Definition 4.2).

Specifications of sequential structures are typically given as sets of strings of labels. To simplify the definitions, we use isomorphism closed sets of LTOs: the event set identifies a bijection between an implementation LVR and its specification as an LTO. Specification sets are closed with respect to renaming of events and arbitrary replacement of the replica function (replicas don't matter in specifications). In addition, we ask that specification sets be prefix closed, accessor enabled (an specification string can always be extended by some accessor) and closed under reordering of adjacent accessors (if there is no intervening mutator, then accessors commute).

Definition 4.1. Strings may be regarded as labeled total orders (LTOs) up to replica-insensitive isomorphism. LTOs s and t are *replica-insensitive isomorphic* if $\mathsf{L}(s) = \mathsf{L}(t)$ and there exists a bijection $\alpha : \mathsf{E}(s) \to \mathsf{E}(t)$ such that $\forall e \in \mathsf{E}(s).\ \lambda_s(e) = \lambda_t(\alpha(e))$ and $\forall d, e \in \mathsf{E}(s).\ (d \rightsquigarrow_s e)$ iff $(\alpha(d) \rightsquigarrow_t \alpha(e))$.

The following closure properties, defined on sets of strings, lift to isomorphism closed sets of LTOs. For strings $s, t \in \mathscr{L}^*$, let "st" denote concatenation. Let $T \subseteq \mathscr{L}^*$ be a set of strings. We say that T is *prefix closed* when $st \in T$ implies $s \in T$. We say that T is *accessor enabled* when $s \in T$ implies $\exists a \in \mathscr{A}.\ sa \in T$. We say that T is *accessor closed* when $\forall a, b \in \mathscr{A}.\ \{ta, tb\} \subseteq T$ implies $\{tab, tba\} \subseteq T$.

A *specification* is a set of total orders (LTOs) that is replica-insensitive isomorphism closed, prefix closed, accessor enabled and accessor closed. □

[4] See Definition 4.2 of the *extension* of a partial order (notation $\subseteq$). A set S of LVRs satisfies *eventual delivery* if each mutator is eventually seen at every replica: $\forall s \in S.\ \forall m \in \mathsf{E_M}(s).\ \forall p \in \mathsf{R}(s).\ \exists t \in S.\ s \subseteq t$ and $\exists a \in \mathsf{E_A}(t).\ m \rightsquigarrow_t a$.

A specification, as given by Definition 4.1, is "sequential" because the orders are total.

We write $=_\pi$ for permutation equivalence; if $s \leq_\pi t$ then t may contain additional events that are not matched in s. If $s \subseteq t$, then t is an *visibility-extension* of s, with the same events and greater visibility. (For an LPO this is an *order-extension*.) If $s \subseteq t$, then t is an *extension* of s, with both more events and greater visibility. $(s \setminus D)$ denotes the restriction of s to the events in D. $\lfloor e \rfloor^{\mathsf{M}}_s$ denotes the restriction of s to the mutator events visible to e, and $\lceil e \rceil^{\mathsf{M}}_s$ denotes the restriction to the mutator events that are either visible to or "concurrent with" e. Both $\lfloor e \rfloor^{\mathsf{M}}_s$ and $\lceil e \rceil^{\mathsf{M}}_s$ include at most one accessor: e itself.

Definition 4.2. Let s and t be LVRs. Write $s \leq_\pi t$ when $\mathsf{E}(s) \subseteq \mathsf{E}(t)$, $\mathsf{L}(s) \subseteq \mathsf{L}(t)$, and $(\forall e \in \mathsf{E}(s).\ \lambda_s(e) = \lambda_t(e)$ and $\rho_s(e) = \rho_t(e))$. Write $s =_\pi t$ when $s \leq_\pi t$ and $t \leq_\pi s$.

Write $s \subseteq t$ when $s \leq_\pi t$ and $(\forall d, e \in \mathsf{E}(s).\ d \rightsquigarrow_s e$ implies $d \rightsquigarrow_t e)$.

Write $s \subseteq\!\!\!= t$ when $s =_\pi t$ and $s \subseteq t$.

For $D \subseteq \mathsf{E}(s)$, define $(s \setminus D) \triangleq \langle D, \mathsf{L}(s), \mathsf{R}(s), \lambda_s \setminus D, \rho_s, (\rightsquigarrow_s) \setminus D \rangle$.

For $L \subseteq \mathscr{L}$, define $(s \setminus L) \triangleq (s \setminus \{e \mid \lambda_s(e) \in L\})$

Define $\lfloor e \rfloor^{\mathsf{M}}_s \triangleq s \setminus \big(\{e\} \cup \{d \in \mathsf{E}_{\mathsf{M}}(s) \mid d \rightsquigarrow e\}\big)$.

Define $\lceil e \rceil^{\mathsf{M}}_s \triangleq s \setminus \big(\{e\} \cup \{d \in \mathsf{E}_{\mathsf{M}}(s) \mid e \not\rightsquigarrow d\}\big)$. □

To establish eventual consistency of s with respect to T, we must exhibit a function t that maps each event in $\mathsf{E}(s)$ to a specification trace in T. The choice of t is constrained by two conditions.

Fix an event e and let $\mathsf{t}(e) = t$. The first condition requires that t include only events visible to or concurrent with e, and that t respect the order of those events in s. The requirement $\mathsf{E}(\lfloor e \rfloor^{\mathsf{M}}_s) \subseteq \mathsf{E}(t)$ establishes that t includes e, as well as all of the mutators visible to e. The requirement $\mathsf{E}(t) \subseteq \mathsf{E}(\lceil e \rceil^{\mathsf{M}}_s)$ establishes that t only includes mutators that are either visible to or concurrent with e. Finally, the requirement that $(s \setminus \mathsf{E}(t)) \subseteq\!\!\!= t$ establishes that t must respect the order of events in s.

Fix events d and e. The second condition requires that $\mathsf{t}(d)$ and $\mathsf{t}(e)$ agree on the order of mutator events in their intersection.

Definition 4.3. We say that t *refines* s at e if $\mathsf{E}(\lfloor e \rfloor^{\mathsf{M}}_s) \subseteq \mathsf{E}(t) \subseteq \mathsf{E}(\lceil e \rceil^{\mathsf{M}}_s)$ and $(s \setminus \mathsf{E}(t)) \subseteq\!\!\!= t$. We write $s \approx_{\mathsf{M}} t$ when $\forall m, n \in \mathsf{E}_{\mathsf{M}}(s) \cap \mathsf{E}_{\mathsf{M}}(t).\ m \rightsquigarrow_s n$ iff $m \rightsquigarrow_t n$.

An LVR s is *eventually consistent* (EC) with a specification T (notation $s \vDash_{\mathsf{ec}} T$) when there exists a map $\mathsf{t} : \mathsf{E}(s) \to T$ such that (a) $\forall e \in \mathsf{E}(s).\ \mathsf{t}(e)$ refines s at e, and (b) $\forall d, e \in \mathsf{E}(s).\ \mathsf{t}(d) \approx_{\mathsf{M}} \mathsf{t}(e)$.

Write $S \vDash_{\mathsf{ec}} T$ when $\forall s \in S.\ s \vDash_{\mathsf{ec}} T$. □

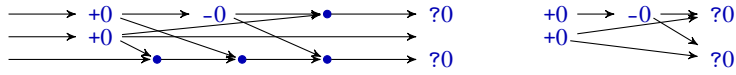
We call this "eventual consistency" because the definition ensures that at quiescent points the same accessors at all the replicas are mapped to the same sequential trace of visible mutator events. Given eventual delivery, then all replicas must eventually agree on the order of all mutators. In the case that specifications are mutator enabled, eventual consistency can be defined in terms of a global order on mutators (u in the proposition below) that all replicas must agree to.

Definition 4.4. A specification T is *mutator enabled* if $\forall s \in T.\ \forall m \in \mathscr{M}.\ sm \in T$. □

Proposition 4.5. Suppose T is mutator enabled specification. Then $s \vDash_{\mathsf{ec}} T$ iff there exists a total order $u =_\pi s \setminus \mathscr{M}$ such that $\forall e \in \mathsf{E}(s).\ \exists t_e \in T.\ t_e$ refines s at e and $t_e \approx_{\mathsf{M}} u$. □

Example 4.6. Consider the execution from Example 2.1, which we repeat below, eliding return values. The BPO is given on the left and the corresponding LVR, on the right.

What are acceptable return values for the get actions? The top replica sees the actions +0 −0 +0 whereas the bottom replica sees +0 +0 −0. They see the same actions, but in different orders. In the 2P-set implementation, both gets return false (remove has priority over add). In the OR-set implementation, both gets return true (add has priority over remove). Both implementations are EC. To see this for the 2P-set, let t map to prefixes of +0 +0 −0. To see this for the OR-set, let t map to prefixes of +0 −0 +0.



An implementation which returns different values for the gets is not EC because there is no t that satisfies the requirements. Since the gets see the same mutators, the specification traces chosen by t must agree on their order. But, since sets are deterministic, there is no set trace that can return both true and false for the same query. □

We end this section with the following simple fact about eventual consistency. The proof uses the fact that we allow events that are concurrent with e to be included in $\mathsf{t}(e)$.

Lemma 4.7. If $v \vDash_{\mathsf{ec}} T$ *and* $s \subseteq v$ *then* $s \vDash_{\mathsf{ec}} T$. □

5 Results

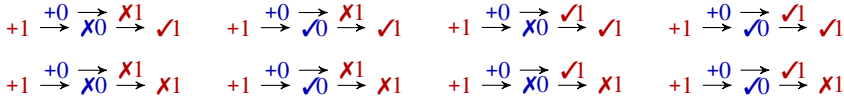
We define a language of clients and define interaction between a client and data structure. We then define monotonicity and state the abstraction and composition results.

Clients. We consider a simple language for clients: parallel composition of sequential processes, which include method call, sequencing and conditional. Let tt and ff represent the boolean constants. Let u and v range over *values*, which include tt and ff. Let o range over *objects*, m over *mutator methods*, and a over *accessor methods*. Then *programs* (P), *configurations* (C) and *labels* (ℓ) are defined as follows.

$$\begin{aligned} P &::= \mathsf{stop} \mid o.m(u);P \mid \mathsf{if}\, o.a(u)\, \mathsf{then}\, P \mid \mathsf{if}\, o.a(u)\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2 \\ C &::= P_1 \parallel \cdots \parallel P_n \\ \ell &::= o.m(u) \mid o.a(u){:}\mathsf{tt} \mid o.a(u){:}\mathsf{ff} \end{aligned}$$

For the most part, we elide occurrences of stop and explicit object references, writing $o.a(u);\mathsf{stop}$ as "$a(u)$". We also write $\mathsf{if}\, a(u)\, \mathsf{then}\, P\, \mathsf{else}\, P$ as "$a(u);P$". In our running example, we have been writing the label $\mathsf{add}(u)$ as "+u", $\mathsf{remove}(u)$ as "−u", $\mathsf{get}(u){:}\mathsf{tt}$ as "✓u" and $\mathsf{get}(u){:}\mathsf{ff}$ as "✗u".

Let $[\![\cdot]\!]$ be a semantic function mapping configurations to sets of LPOs. The definition is the obvious one. For example, let C be the configuration $\mathsf{add}(0);\mathsf{get}(1) \parallel \mathsf{add}(1);\mathsf{get}(0);\mathsf{get}(1)$. Then $[\![C]\!]$ is a set of the following eight LPOs (up to isomorphism).



(7)

Under what circumstances can such a client interact with a 2P-set or OR-set and expect that the observed behaviour if compatible with a sequential set? This question is

addressed in our first result, known as *abstraction* : when is the actual implementation of a data structure a safe substitute for its "abstract" specification?

We must first define what it means for a client and a data structure to interact.

Interaction. From figure (7) it is clear that the data structure must be able to filter out executions of the client. The set datatype does not include any traces that are compatible with the four LPOs on the second line of figure (7), since ✗1 must follow +1 and there is no −1.

From figure (7) it is equally clear that the data structure must be able to introduce visibility that is not found in the client. For example, to achieve the results on the first line, one must introduce visibility between the client programs, as follows.

It is safe for the data structure to add visibility (and therefore order) to the client; however, the reverse is not true. A client can only introduce order that is compatible with the data structure specification. Consider the sequential client `add(0);get(0);get(0)`. If this client communicates to separate replicas in a G-set, the execution +0 ✓0 ✗0 is possible, via the BPO $\xrightarrow{\quad} \overset{\rightarrow}{+0} \overset{\rightarrow}{\underset{\rightarrow}{\underset{\rightarrow}{\checkmark 0}}} \overset{\rightarrow}{\underset{\rightarrow}{\underset{\rightarrow}{\times 0}}}$. To avoid such anomalies, it is sufficient to require that sequential clients alway move forward in the visibility relation. This can be achieved by restricting each client program to communicate with a single replica, or by other means[5]. We include this requirement in our definition of composition, without specifying how it is fulfilled.

Definition 5.1. Let S be a set of LVRs. $[\![C]\!](S) \triangleq \{s \in S \mid \exists s' \in [\![C]\!].\ s' \subseteq\!\!\!= s\}$ □

One reading of the asymmetry in this definition is that a data structure may introduce order, but not its clients. A more generous reading is that clients may require order that is compatible with the data structure (that the data structure *could* have), but may not introduce incompatible order.

Monotonicity and abstraction. Even with this definition of the semantics, abstraction fails in general. Consider the client `add(0);get(1) ∥ add(1);get(0)`. The BPO $\overset{\rightarrow}{\underset{\rightarrow}{+0}} \overset{\rightarrow}{\underset{\rightarrow}{+1}} \overset{\rightarrow}{\underset{\rightarrow}{\times 1}} \overset{\rightarrow}{\underset{\rightarrow}{\times 0}} \xrightarrow{\quad}$ has order agreeing with the client and is an EC execution of a set, but this behaviour is not observable by a client interacting with a sequential set. Abstraction holds for clients that ensure *monotone* access to the data structure.

A set V is monotone if whenever V contains a trace u with a mutator m that is concurrent with another event e, then V also contains a visibility extension v that orders m and e. Since v is an visibility extension of u, it must contain the same labels.

Definition 5.2. A set V of LVRs is *monotone* when $\forall u \in V.\ \forall m \in \mathsf{E}_{\mathsf{M}}(u).\ \forall e \in \mathsf{E}(u).$ if $(m \not\rightsquigarrow_u e$ and $e \not\rightsquigarrow_u m)$ then $\exists v \in V.\ u \subseteq\!\!\!= v$ and $(m \rightsquigarrow_u e$ or $e \rightsquigarrow_u m)$ □

[5] The "single replica" solution may compromise the goals of replication, such as fault tolerance and performance. Another approach, applicable to causal systems with timestamps, is as follows: In each request and response, the client and data structure exchange timestamps. The data structure sends the timestamp of the last mutator seen by the responding replica, and the client sends the last timestamp it received. A replica may respond to a client request only if its timestamp is greater than the timestamp included in the request. If a replica receives a request that it is incapable of handling, it may wait or forward the request.

Theorem 5.3. Let S be a set of LVR*s and let T be a specification such that $S \vDash_{\mathsf{ec}} T$. Let C be a client such that $[\![C]\!](S)$ is monotone. Then $\forall s \in [\![C]\!](S).\ \exists t \in [\![C]\!](T).\ s \subseteq t$.* □

The theorem states that if there is an execution s in $[\![C]\!](S)$, then is a corresponding execution t in $[\![C]\!](T)$ that has exactly the same labels, and potentially more order. This says that any client behaviour possible with the implementation S is also possible using the sequential specification T.

Example 5.4. The G-set trace $\xrightarrow{\ \ } \overset{+0}{\underset{+1}{\rightarrow}} \xrightarrow{\ \ } \overset{\text{✗}1}{\rightarrow}$ can be allowed in a monotone subset of G-set executions, since we can order ✗1 before +1 and still have a set execution; the events +1 and +0 can be ordered arbitrarily. The G-set trace $\xrightarrow{\ \ } \overset{+0}{\underset{+1}{\rightarrow}} \xrightarrow{\ \ } \overset{\text{✗}1}{\rightarrow} \overset{\text{✗}0}{\rightarrow}$, however, cannot be allowed in a monotone subset of G-set executions. In this case, if we order +1 before ✗1, then the result is clearly not a set execution: 1 has been added, but is not reported present. If we choose the reverse order, we have +0 before ✗0, and again the result fails to be a valid set execution.

Example 5.7 below gives an example of a specific G-set client that satisfies monotonicity, under given assumptions. To design a general class of context-independent monotone clients for a given data structure, it is necessary to limit client programs, as done in languages in the CALM framework [9].

For example, in order to create a monotone subset of G-set traces, it is sound to restrict clients to disallow the two-armed if-then-else. The semantics of the one-armed if-then is blocking—the client must wait until the condition is true. The theorem establishes that such clients can safely us a G-set as though it were a sequential set.

The theorem provides guidance about how to design safe clients. In order to allow a two-armed conditional with the G-set, we must ensure that events occurring concurrently with a negative response cannot invalidate that response. One way to achieve this, following [9], is for the G-set to insert a barrier before returning a negative response. □

Composition of data structures. We now turn our attention to reasoning about compound data structures.

Definition 5.5. Given disjoint LTOs t_1 and t_2 (that is, $\mathsf{E}(t_1) \cap \mathsf{E}(t_2) = \emptyset$), let $t_1 \mid\mid\mid t_2$ denote the set of their interleavings. This notion lifts to sets as follows: $(T_1 \mid\mid\mid T_2) \triangleq \{t \in (t_1 \mid\mid\mid t_2) \mid t_1 \in T_1 \text{ and } t_2 \in T_2 \text{ and } (\mathsf{E}(t_1) \cap \mathsf{E}(t_2) = \emptyset)\}$.

Given an LVR s and $L \subseteq \mathsf{L}(s)$, write $s \setminus L$ for the LVR that results by restricting s to events with labels in L. This notation lifts to sets in the obvious way: $S \setminus L \triangleq \bigcup_{s \in S} s \setminus L$.□

Theorem 5.6. Let $[\![C]\!](S)$ be a monotone set of LVR*s. Let L_1 and L_2 be disjoint subsets of $\mathscr{L}$. For $i \in \{1,2\}$, let T_i be a specification with labels chosen from L_i. If $([\![C]\!](S) \setminus L_1) \vDash_{\mathsf{ec}} T_1$ and $([\![C]\!](S) \setminus L_2) \vDash_{\mathsf{ec}} T_2$ then $[\![C]\!](S) \vDash_{\mathsf{ec}} (T_1 \mid\mid\mid T_2)$.* □

Example 5.7. The following definitions implement a 2P-set p, using two G-sets, a for "added" and t for "tombstone": $p.\mathsf{add}(u) \triangleq a.\mathsf{add}(u)$, $p.\mathsf{remove}(u) \triangleq t.\mathsf{add}(u)$, and $p.\mathsf{get}(u) \triangleq a.\mathsf{get}(u) \wedge \neg t.\mathsf{get}(u)$. If we can establish the necessary monotonicity properties, then we can reason with the sequential specifications of a and t in proving p correct. An execution of a grow set g is monotone so long as for any $g.\text{✗}u$, there is no concurrent $g.{+}u$. We must show that both a and t are accessed monotonically, so long as p is accessed monotonically. An execution of a 2P-set p is monotone so long as (1) for any $p.\text{✓}u$, there is no concurrent $p.{-}u$, and (2) for any $p.\text{✗}u$, there is no concurrent $p.{+}u$.

The conditions for monotonicity of p are sufficient to establish monotonicity of a and t. There are two cases: (1) Suppose $p.\checkmark u$. By monotonicity, we know there is no concurrent $p.\text{-}u$, therefore no concurrent $t.\text{+}u$. By definition of p.get, we must have $a.\checkmark u$ and $t.\text{✗}u$. Monotonicity imposes no constraints on $a.\checkmark u$; to satisfy $t.\text{✗}u$, we must have no concurrent $t.\text{+}u$, but this is exactly guaranteed by monotonicity of p. (2) Suppose $p.\text{✗}u$. Then we know there is no concurrent $p.\text{+}u$, therefore no concurrent $a.\text{+}u$. By definition of p.get, we must have either $a.\text{✗}u$ or $t.\checkmark u$. The argument is as before. □

References

[1] A. Bouajjani, C. Enea, and J. Hamza, "Verifying eventual consistency of optimistic replication systems," in *POPL '14*, 2014, pp. 285–296.

[2] S. Burckhardt, A. Gotsman, H. Yang, *et al.*, "Replicated data types: Specification, verification, optimality," in *POPL '14*, 2014, pp. 271–284.

[3] N. Conway, W. R. Marczak, P. Alvaro, *et al.*, "Logic and lattices for distributed programming," in *ACM Symposium on Cloud Computing*, 2012, 1:1–1:14.

[4] J. Derrick, B. Dongol, G. Schellhorn, *et al.*, "Quiescent consistency: Defining and verifying relaxed linearizability," in *Formal Methods*, 2014, pp. 200–214.

[5] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *ACM SIGMOD Record*, vol. 18, no. 2, pp. 399–407, Jun. 1989.

[6] I. Filipovic, P. O'Hearn, N. Rinetzky, *et al.*, "Abstraction for concurrent objects," *Theoretical Comp. Sci.*, vol. 411, pp. 4379–4398, 2010.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, pp. 51–59, 2002.

[8] A. Gotsman and H. Yang, "Composite replicated data types," To appear, 2015.

[9] J. M. Hellerstein, "The declarative imperative: Experiences and conjectures in distributed logic," *SIGMOD Rec.*, vol. 39, no. 1, pp. 5–19, Sep. 2010.

[10] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.

[11] R. Jagadeesan and J. Riely, "Between linearizability and quiescent consistency - quantitative quiescent consistency," in *ICALP '14*, 2014, pp. 220–231.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[13] P. Panangaden, V. Shanbhogue, and E. W. Stark, "Stability and sequentiality in dataflow networks," in *ICALP '90*, 1990, pp. 308–321.

[14] P. Panangaden and E. W. Stark, "Computations, residuals, and the power of indeterminacy," in *ICALP '88*, 1988, pp. 439–454.

[15] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005.

[16] M. Shapiro, N. Preguiça, C. Baquero, *et al.*, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria, TR 7506, 2011.

[17] D. B. Terry, M. M. Theimer, K. Petersen, *et al.*, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *SOSP*, 1995.

[18] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.