Eventual Consistency for CRDTs

Radha Jagadeesan and James Riely

DePaul University

Abstract. We address the problem of *validity* in eventually consistent (EC) systems: In what sense does an EC data structure satisfy the sequential specification of that data structure? Because EC is a very weak criterion, our definition does not describe every EC system; however it is expressive enough to describe any Convergent or Commutative Replicated Data Type (CRDT).

1 Introduction

In a replicated implementation of a data structure, there are two impediments to requiring that all replicas achieve consensus on a global total order of the operations performed on the data structure [Lamport 1978]: (a) the associated serialization bottleneck negatively affects performance and scalability (*e.g.* see [Ellis and Gibbs 1989]), and (b) the CAP theorem imposes a tradeoff between consistency and partition-tolerance [Gilbert and Lynch 2002].

In systems based on *optimistic replication* [Vogels 2009; Saito and Shapiro 2005], a replica may execute an operation without synchronizing with other replicas. If the operation is a mutator, the other replicas are updated asynchronously. Due to the vagaries of the network, the replicas could receive and apply the updates in possibly different orders.

For sequential systems, the correctness problem is typically divided into two tasks: proving *termination* and proving *partial correctness.* Termination requires that the program eventually halt on all inputs, whereas partial correctness requires that the program only returns results that are allowed by the specification.

For replicated systems, the analogous goals are *convergence* and *validity.* Convergence requires that all replicas eventually agree. Validity requires that they agree on something sensible. In a replicated list, for example, if the only value put into the list is 1, then convergence ensures that all replicas eventually see the same value for the head of the list; validity requires that the value be 1.

Convergence has been well-understood since the earliest work on replicated systems. Convergence is typically defined as *eventual consistency*, which requires that once all messages are delivered, all replicas have the same state. *Strong eventual consistency* (SEC) additionally requires convergence for all subsets of messages: replicas that have seen the same messages must have the same state.

Perhaps surprisingly, finding an appropriate definition of validity for replicated systems remains an open problem. There are solutions which use concurrent specifications, discussed below. But, as Shavit [2011] noted:

"It is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting, where there are no interleavings. Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the structure's properties sequentially, and find a way to map its concurrent executions to these 'correct' sequential ones."

In this paper we give the first definition of validity that is both (1) derived from standard sequential specifications and (2) validates the examples of interest.

We take the "examples of interest" to be *Convergent/Commutative Replicated Data Types* (CRDTs). These are replicated structures that obey certain monotonicity or commutativity properties. As an example of a CRDT, consider the *add-wins set*, also called an "observed remove" set in [Shapiro et al. 2011a]. The add-wins set behaves like a sequential set if add and remove operations on the same element are ordered. The concurrent execution of an add and remove result in the element being added to the set; thus the remove is ignored and the "add wins." This concurrent specification is very simple, but as we will see in the next section, it is quite difficult to pin down the relationship between the CRDT and the sequential specification used in the CRDT's definition. This paper is the first to successfully capture this relationship.

Many replicated data types are CRDTs, but not all [Shapiro et al. 2011a]. Notably, Amazon's Dynamo [DeCandia et al. 2007] is not a CRDT. Indeed, interest in CRDTs is motivated by a desire to avoid the well-know concurrency anomalies suffered by Dynamo and other ad hoc systems [Bieniusa et al. 2012].

Shapiro et al. [2011b] introduced the notion of CRDT and proved that every CRDT has an SEC implementation. Their definition of SEC includes convergence, but not validity.

The validity requirement can be broken into two components. We describe these below using the example of a list data type that supports only two operations: the mutator `put`, which adds an element to the end of the list, and the query `q`, which returns the state of the list. This structure can be specified as a set of strings such as "`put(1); put(3); q=[1,3]`" and "`put(1); put(2); put(3); q=[1,2,3]`".

– *Linearization* requires that a response be consistent with some specification string. A state that received `put(1)` and `put(3)`, may report `q=[1,3]` or `q=[3,1]`, but not `q=[2,1,3]`, since `2` has not been put into the list.
– *Monotonicity* requires that states evolve in a sensible way. We might permit the state `q=[1,3]` to evolve into `q=[1,2,3]`, due to the arrival of action `put(2)`. But we would not expect that `q=[1,3]` could evolve into `q=[3,1]`, since the data type does not support deletion or reordering.

Burckhardt et al. [2012] provide a formal definition of validity using partial orders over events: linearizations respect the partial order on events; monotonicity is ensured by requiring that evolution extends the partial order. Similar definitions can be found in [Jagadeesan and Riely 2015] and [Perrin et al. 2015]. Replicated data structures that are sound with respect to this definition enjoy many good properties, which we discuss throughout this paper. However, this notion of correctness is not general enough to capture common CRDTs, such as the add-wins set.

This lack of expressivity lead Burckhardt et al. [2014] to abandon notions of validity that appeal directly to a sequential specification. Instead they work directly with *concurrent* specifications, formalizing the style of specification found informally in [Shapiro et al. 2011b]. This has been a fruitful line of work, leading to proof rules [Gotsman et al. 2016] and extensions [Bouajjani et al. 2014]. See [Burckhardt 2014; Viotti and Vukolic 2016] for a detailed treatment.

Positively, concurrent specifications can be used to validate any replicated structure, including CRDTs as well as anomalous structures such as Dynamo. Negatively, concurrent specifications have no the clear connection to their sequential counterparts. In this paper, we restore this connection. We arrive at a definition of SEC that admits CRDTs, but rejects Dynamo.

The following "corner cases" are a useful sanity-check for any proposed notion of validity.

– The principle of *single threaded semantics* (PSTS) [Haas et al. 2015] states that if an execution uses only a single replica, it should behave according to the sequential semantics.
– The principle of *single master* (PSM) [Budhiraja et al. 1993] states that if all mutators in an execution are initiated at a single replica, then the execution should be linearizable [Herlihy and Wing 1990].
– The principle of *permutation equivalence* (PPE) [Bieniusa et al. 2012] states that "if all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states."

PSTS and PSM say that a replicated structure should behave sequentially when replication is not used. PPE says that the order of independent operations should not matter. Our definition implies all three conditions. Dynamo fails PPE [Bieniusa et al. 2012], and thus fails to pass our definition of SEC.

In the next section, we describe the validity problem and our solution in detail, using the example of a binary set. The formal definitions follow in Section 3. We state some consequences of the definition and prove that the add-wins set satisfies our definition. In Section 4, we describe a collaborative text editor and prove that it is SEC. In Section 5 we characterize the programmer's view of a CRDT by defining the *most general* CRDT that satisfies a given sequential specification. We show that any program that is correct using the most general CRDT will be correct using a more restricted CRDT. We also show that our validity criterion for SEC is *local* in the sense of Herlihy and Wing [1990]: independent structures can be verified independently. In Section 6, we apply these results to prove the correctness of a graph that is implemented using two SEC sets.

Our work is inspired by the study of relaxed memory, such as [Alglave 2012]. In particular, we have drawn insight from the RMO model of Higham and Kawash [2000].

2 Understanding Replicated Sets

In this section, we motivate the definition of SEC using replicated sets as an example. The final definition is quite simple, but requires a fresh view of both executions and specifications. We develop the definition in stages, each of which requires a subtle shift in perspective. Each subsection begins with an example and ends with a summary.

2.1 Mutators and Non-Mutators

An *implementation* is a set of *executions*. We model executions abstractly as labelled partial orders (LPOs). The ordering of the LPO captures the history that precedes an event, which we refer to as *visibility*.

$$\begin{tikzcd}
{+0}^{a} \arrow[r] \arrow[rrrd] & \checkmark 0^{b} \arrow[r] & \text{✗}1^{c} \arrow[r] & \checkmark 0^{d} \arrow[r] & \checkmark 1^{e} \\
{+1}_{f} \arrow[r] \arrow[rrru] & \text{✗}0_{g} \arrow[r] & \checkmark 1_{h} \arrow[r] & \checkmark 0_{i} \arrow[r] & \checkmark 1_{j}
\end{tikzcd} \qquad (1)$$

Here the events are a through j, with labels +0, +1, etc, and order represented by arrows. The LPO describes an execution with two replicas, shown horizontally, with time passing from left to right. Initially, the top replica receives a request to add 0 to the set (+0^a). Concurrently, the bottom replica receives a request to add 1 (+1^b). Then each replica is twice asked to report on the items contained in the set. At first, the top replica replies that 0 is present and 1 is absent (✓0^b✗1^c), whereas the bottom replica answers with the reverse (✗0^g✓1^h). Once the add operations are visible at all replicas, however, the replicas give the same responses (✓0^d✓1^e and ✓0^i✓1^j).

LPOs with non-interacting replicas can be denoted compactly using sequential and parallel composition. For example, the prefix of (1) that only includes the first three events at each replica can be written (+0^a; ✓0^b; ✗1^c) $\parallel$ (+1^f; ✗0^g; ✓1^h).

A *specification* is a set of *strings*. Let SET be the specification of a sequential set with elements 0 and 1. Then we expect that SET includes the string "+0✓0✗1", but not "+0✗0✓1". Indeed, each specification string can uniquely be extended with either ✓0 or ✗0 and either ✓1 or ✗1.

There is an isomorphism between strings and labelled *total* orders. Thus, specification strings correspond to the restricted class of LPOs where the visibility relation provides a total order.

Linearizability [Herlihy and Wing 1990] is the gold standard for concurrent correctness in tightly coupled systems. Under linearizability, an execution is valid if there exists a linearization τ of the events in the execution such that for every event e, the prefix of e in τ is a valid specification string.

Execution (1) is not linearizable. The failure can already be seen in the sub-LPO (+0^a; ✗1^c) $\parallel$ (+1^f; ✗0^g). Any linearization must have either +1^f before ✗1^c or +0^a before ✗0^g. In either case, the linearization is invalid for SET.

Although it is not linearizable, execution (1) is admitted by every CRDT SET in [Shapiro et al. 2011a]. To validate such examples, Burckhardt et al. [2012]

develop a weaker notion of validity by dividing labels into *mutators* and *accessors* (also known as non-mutators). Similar definitions appear in [Jagadeesan and Riely 2015] and [Perrin et al. 2015]. Mutators change the state of a replica, and accessors report on the state without changing it. For SET, the mutators $\mathbf{M}$ and non-mutators $\overline{\mathbf{M}}$ are as follows.

$\mathbf{M} = \{$`+0`, `-0`, `+1`, `-1`$\}$, representing addition and removal of bits 0 and 1.
$\overline{\mathbf{M}} = \{$✗`0`, ✓`0`, ✗`1`, ✓`1`$\}$, representing membership tests returning false or true.

Define the *mutator prefix* of an event e to include e and the *mutators* visible to e. An execution is valid if there exists a linearization of the execution, τ, such that for every event e, the *mutator prefix* of e in τ is a valid specification string.

It is straightforward to see that execution (1) satisfies this weaker criterion. For both ✓0^b and ✗1^c, the mutator prefix is +0^a. This includes +0^a but not +1^f, and thus their answers are validated. Symmetrically, the mutator prefixes of ✗0^g and ✓1^h only include +1^f. The mutator prefixes for the final four events include both +0^a and +1^f, but none of the prior accessors.

Summary: Convergent states must agree on the final order of mutators, but intermediate states may see incompatible subsequences of this order. By restricting attention to mutator prefixes, the later states need not linearize these incompatible views of the partial past.

This relaxation is analogous to the treatment of non-mutators in update serializability [Hansdah and Patnaik 1986; Garcia-Molina and Wiederhold 1982], which requires a global serialization order for mutators, ignoring non-mutators.

2.2 Dependency

The following LPO is admitted by the add-wins SET discussed in the introduction.

$$\begin{tikzcd}
\texttt{+0}^{a} \arrow[r] & \texttt{+1}^{b} \arrow[r] & \texttt{-1}^{c} \arrow[dr] & & \\
& & & \checkmark\texttt{0}^{g} \arrow[r] & \checkmark\texttt{1}^{h} \\
\texttt{+1}_{d} \arrow[r] & \texttt{+0}_{e} \arrow[r] & \texttt{-0}_{f} \arrow[ur] & &
\end{tikzcd} \qquad (2)$$

In any CRDT implementation, the effect of +1^b is negated by the subsequent -1^c The same reasoning holds for +0^e and -0^f. In an add-wins set, however, the *concurrent* adds, +0^a and +1^d, win over the deletions. Thus, in the final state both 0 and 1 are present.

This LPO is not valid under the definition of the previous subsection: Since ✓0^g and ✓1^h see the same mutators, they must agree on a linearization of (+0^a; +1^b; -1^c) $\|$ (+1^d; +0^e; -0^f). Any linearization must end in either -1^c or -0^f; thus it is not possible for both ✓0^g and ✓1^h to be valid.

Similar issues arise in relaxed memory models, where program order is often relaxed between uses of independent variables [Alglave et al. 2014]. Generalizing, we write $m \# n$ to indicate that labels m and n are dependent. Dependency is a property of a *specification*, not an implementation. Our results only apply to specifications that support a suitable notion of dependency, as detailed in

Section 3. For SET, # is an equivalence relation with two equivalence classes, corresponding to actions on the independent values 0 and 1.

$$\# = \{\texttt{+0},\ \texttt{-0},\ \text{✗}\texttt{0},\ \checkmark\texttt{0}\}^2 \cup \{\texttt{+1},\ \texttt{-1},\ \text{✗}\texttt{1},\ \checkmark\texttt{1}\}^2,\ \text{where } D^2 = D \times D.$$

While the dependency relation for SET is an equivalence, this is not required: In Section 4 we establish the correctness of collaborative text editing protocol with an intransitive dependency relation.

The *dependent restriction* of (2) is as follows.

$$\begin{tikzcd}
& \boxed{\texttt{+0}}^a \arrow[rd] & & \boxed{\texttt{+1}}^b \arrow[r] & \boxed{\texttt{-1}}^c \arrow[rd] & \\
\boxed{\texttt{+0}}_e \arrow[r] & \boxed{\texttt{-0}}_f \arrow[r] & \boxed{\checkmark\texttt{0}}^g & & \boxed{\texttt{+1}}_d \arrow[r] & \boxed{\checkmark\texttt{1}}^h
\end{tikzcd} \tag{3}$$

In the previous subsection, we defined validity using the *mutator prefix* of an event. We arrive at a weaker definition by restricting attention to the *mutator prefix of the dependent restriction*.

Under this definition, execution (2) is validated: Any interleaving of the strings $\texttt{+0}^e\texttt{-0}^f\texttt{+0}^a\checkmark\texttt{0}^g$ and $\texttt{+1}^b\texttt{-1}^c\texttt{+1}^d\checkmark\texttt{1}^h$ linearizes the dependent restriction of (2) given in (3).

Summary: CRDTs allow independent mutators to commute. We formalize this intuition by restricting attention to mutator prefixes of the dependent restriction. The CRDT must respect program order between dependent operations, but is free to reorder independent operations.

This relaxation is analogous to the distinction between *program order* and *preserved* program order (PPO) in relaxed memory models [Higham and Kawash 2000; Alglave 2012]. Informally, PPO is the suborder of program order that removes order between independent memory actions, such as successive reads on different locations without an intervening memory barrier.

2.3 Puns

The following LPO is admitted by the add-wins SET.

$$\begin{tikzcd}
\boxed{\texttt{+0}}^a \arrow[r] \arrow[rrd] & \boxed{\texttt{-0}}^b \arrow[r] \arrow[rrd] & \boxed{\checkmark\texttt{0}}^c \arrow[r] & \boxed{\text{✗}\texttt{0}}^d \\
\boxed{\texttt{+0}}_e \arrow[r] \arrow[rru] & \boxed{\texttt{-0}}_f \arrow[r] \arrow[rru] & \boxed{\checkmark\texttt{0}}_g \arrow[r] & \boxed{\text{✗}\texttt{0}}_h
\end{tikzcd} \tag{4}$$

As in execution (2), the add $\texttt{+0}^a$ is undone by the following remove $\texttt{-0}^b$, but the concurrent add $\texttt{+0}^e$ wins over $\texttt{-0}^b$, allowing $\checkmark\texttt{0}^c$. In effect, $\checkmark\texttt{0}^c$ sees the order of the mutators as $\texttt{+0}^a\ \texttt{-0}^b\ \texttt{+0}^e$. Symmetrically, $\checkmark\texttt{0}^g$ sees the order as $\texttt{+0}^e\ \texttt{-0}^f\ \texttt{+0}^a$. While this is very natural from the viewpoint of a CRDT, there is no linearization of the events that includes both $\texttt{+0}^a\ \texttt{-0}^b\ \texttt{+0}^e$ and $\texttt{+0}^e\ \texttt{-0}^f\ \texttt{+0}^a$, since $\texttt{+0}^a$ and $\texttt{+0}^e$ must appear in different orders.

Indeed, this LPO is not valid under the definition of the previous subsection. First note that all events are mutually dependent. To prove validity we must find a linearization that satisfies the given requirements. Any linearization of the mutators must end in either $\texttt{-0}^b$ or $\texttt{-0}^f$. Suppose we choose $\texttt{+0}^a\ \texttt{-0}^b\ \texttt{+0}^e\ \texttt{-0}^f$ and look for a mutator prefix to satisfy ✓0^g. (All other choices lead to similar problems.) Since -0^f precedes ✓0^g and is the last mutator in our chosen linearization, every possible witness for ✓0^g must end with mutator -0^f. Indeed the only possible witness is +0^a +0^e -0^f ✓0^g. However, this is not a valid specification string.

The problem is that we are linearizing *events*, rather than *labels*. If we shift to linearizing labels, then execution (4) is allowed. Fix the final order for the mutators to be +0 -0 +0 -0. The execution is allowed if we can find a subsequence that linearizes the labels visible at each event. It suffices to choose the witnesses as follows. In the table, we group events with a common linearization together.

+0^a, +0^e:	+0	✓0^c, ✓0^g:	+0-0+0✓0
-0^b, -0^f:	+0-0	✗0^d, ✗0^h:	+0-0+0-0✗0

Each of these is a valid specification string. In addition, looking only at mutators, each is a subsequence of +0 -0 +0 -0.

In execution (4), each of the witnesses is actually a *prefix* of the final mutator order, but, in general, it is necessary to allow *subsequences*.

$$\begin{tikzcd}
 & \boxed{+0}^{a} \arrow[rr] \arrow[dr] & & \boxed{\checkmark 0}^{b} \\
\boxed{-0}_{c} \arrow[rr] & & \boxed{\checkmark 0}_{d} &
\end{tikzcd} \tag{5}$$

Execution (5) is admitted by the add-wins SET. It is validated by the final mutator sequence -0 +0. The mutator prefix +0 of b is a subsequence of -0 +0, but not a prefix.

Summary: While dependent events at a single replica must be linearized in order, concurrent events may slip anywhere into the linearization. A CRDT may *pun* on concurrent events with same label, using them in different positions at different replicas. Thus a CRDT may establish a final total over the labels of an execution even when there is no linearization of the events.

2.4 Frontiers

In the introduction, we mentioned that the validity problem can be decomposed into the separate concerns of *linearizability* and *monotonicity.* The discussion thus far has centered on the appropriate meaning of linearizability for CRDTs. In this subsection and the next, we look at the constraints imposed by monotonicity.

Consider the prefix {+0^a, -0^b, +0^e, ✓0^c, -0^f} of execution (4), extended with action ✗0^x, with visibility order as follows.

$$\begin{tikzcd}
{}^{a}\boxed{+0} \arrow[r] & \boxed{-0}^{b} \arrow[r] \arrow[ddr] & \boxed{\checkmark 0}^{c} \\
{}_{e}\boxed{+0} \arrow[r] \arrow[urr] \arrow[drr, bend right] & \boxed{-0}_{f} & \\
 & & \boxed{\times 0}_{x}
\end{tikzcd} \tag{6}$$

This execution is *not strong* EC, since ✓0^c and ✗0^x see exactly the same mutators, yet provide incompatible answers.

Unfortunately, execution (6) is valid by the definition given in the previous section: The witnesses for a–f are as before. In particular, the witness for ✓0^c is

"+0-0+0✓0". The witness for ✗0^x is "+0+0-0✗0". In each case, the mutator prefix is a subsequence of the global mutator order "+0-0+0-0".

It is well known that punning can lead to bad jokes. In this case, the problem is that ✗0^x is punning on a concurrent -0 that cannot be matched by a visible -0 in its history: the execution -0 that is visible to ✗0^x must appear *between* the two +0 operations; the specification -0 that is used by ✗0^x must appear *after*. The final states of execution (4) have seen both remove operations, therefore the pun is harmless there. But ✓0^c and ✗0^x have seen only one remove. They must agree on how it is used.

Up to now, we have discussed the linearization of each event in isolation. We must also consider the relationship between these linearizations. When working with linearizations of *events*, it is sufficient to require that the linearization chosen for each event be a subsequence for the linearization chosen for each visible predecessor; since events are unique, there can be no confusion in the linearization about which event is which. Execution (6) shows that when working with linearizations of *labels*, it is insufficient to consider the relationship between individual events. The linearization "+0+0-0✗0" chosen for ✗0^x is a supersequence of those chosen for its predecessors: "+0" for +0^e and "+0-0" for -0^b. The linearization "+0-0+0✓0" chosen for ✓0^c is also a supersequence for the same predecessors. And yet, ✓0^c and ✗0^x are incompatible states.

Sequential systems have a single state, which evolves over time. In distributed systems, each replica has its own state, and it is this *set* of states that evolves. Such a set of states is called a *(consistent) cut* [Chandy and Lamport 1985].

A *cut* of an LPO is a sub-LPO that is down-closed with respect to visibility. The *frontier* of cut is the set of maximal elements. For example, there are 14 frontiers of execution (6): the singletons {+0^a}, {-0^b}, {✓0^c}, {+0^e}, {-0^f}, {✗0^x}, the pairs {+0^a, +0^e}, {+0^a, -0^f}, {-0^b, +0^e}, {-0^b, -0^f}, {✓0^c, -0^f}, {✓0^c, ✗0^x}, {✗0^x, -0^f}, and the triple {✓0^c, ✗0^x, -0^f}. As we explain below, we consider non-mutators in isolation. Thus we do not consider the last four cuts, which include a non-mutator with other events. That leaves 10 frontiers. The definition of the previous section only considered the 6 singletons. Singleton frontiers are generated by *pointed cuts*, with a single maximal element.

When applied to frontiers, the monotonicity requirement invalidates execution (6). Monotonicity requires that the linearization chosen for a frontier be a subsequence of the linearization chosen for any extension of that frontier. If we are to satisfy state ✓0^c in execution (6), the frontier {-0^b, +0^e} must linearize to "+0-0+0". If we are to satisfy state ✗0^x, the frontier {-0^b, +0^e} must linearize to "+0+0-0". Since we require a unique linearization for each frontier, the execution is disallowed.

Since CRDTs execute non-mutators locally, it is important that we ignore frontiers with multiple non-mutators. Recall execution (4):

$$\begin{tikzcd}
\boxed{+0}^{a} \arrow[r] \arrow[drr] & \boxed{-0}^{b} \arrow[r] \arrow[drr] & \boxed{✓0}^{c} \arrow[r] & \boxed{✗0}^{d} \\
\boxed{+0}_{e} \arrow[r] \arrow[urr] & \boxed{-0}_{f} \arrow[r] \arrow[urr] & \boxed{✓0}_{g} \arrow[r] & \boxed{✗0}_{h}
\end{tikzcd}$$

There is no specification string that linearizes the cut with frontier $\{\checkmark 0^c, \checkmark 0^g\}$, since we cannot have ✓0 immediately after -0. If we consider only pointed cuts for non-mutators, then the execution is SEC, with witnesses as follows.

$\{+0^a\}, \{+0^e\}$	: +0	$\{\checkmark 0^c\}, \{\checkmark 0^g\}$	: +0-0+0✓0
$\{+0^a, +0^e\}$	: +0+0	$\{-0^b, -0^f\}$	: +0-0+0-0
$\{-0^b\}, \{-0^f\}$	: +0-0	$\{\text{✗}0^d\}, \{\text{✗}0^h\}$	: +0-0+0-0✗0
$\{-0^b, +0^e\}, \{+0^a, -0^f\}$	: +0-0+0		

In order to validate non-mutators, we *must* consider singleton non-mutator frontiers. The example shows that we *must not* consider frontiers with multiple non-mutators. There is some freedom in the choices otherwise. For SET, we can "saturate" an execution with accessors by augmenting the execution with accessors that witness each cut of the mutators. In a saturated execution, it is sufficient to consider only the *pointed accessor* cuts, which end in a maximal accessor. For non-saturated executions, we are forced to examine each mutator cut: it is possible that a future accessor extension may witness that cut. The status of "mixed" frontiers, which include mutators with a single maximal non-mutator, is open for debate. We choose to ignore them, but the definition does not change if they are included.

Summary: A CRDT must have a strategy for linearizing all mutator labels, even in the face of partitions. In order to ensure *strong* EC, the definition must consider sets of events across multiple replicas. Because non-mutators are resolved locally, SEC must ignore frontiers with multiple non-mutators.

Cuts and frontiers are well-known concepts in the literature of distributed systems [Chandy and Lamport 1985]. It is natural to consider frontiers when discussing the evolving correctness of a CRDT.

2.5 Stuttering

Consider the following execution.

$$\begin{tikzcd}
 & & & \boxed{-0}^{d} & & \boxed{-0}^{y} \\
{}^{a}\boxed{+0} \arrow[r] & {}^{b}\boxed{-0} \arrow[r] & {}^{c}\boxed{+0} \arrow[ur] \arrow[dr] & & {}^{x}\boxed{+0} \arrow[ur] \arrow[dr] & \\
 & & & \boxed{-0}^{e} & & \boxed{-0}^{z}
\end{tikzcd} \tag{7}$$

This LPO represents a partitioned system with events *a*–*e* in one partition and *x*–*z* in the other. As the partition heals, we must be able to account for the intermediate states. Because of the large number of events in this example, we have elided all accessors. We will present the example using the semantics of the add-wins set. Recall that the add-wins set validates ✓0 if and only if there is a maximal +0 beforehand. Thus, a replica that has seen the cut with frontier $\{+0^a, -0^y, -0^z\}$ must answer ✓0, whereas a replica that has seen $\{-0^b, -0^y, -0^z\}$ must answer ✗0.

Any linearization of $\{+0^a, -0^y, -0^z\}$ must end in +0, since the add-win set must reply ✓0: the only possibility is "+0-0-0+0". The linearization of $\{-0^b,$

-0^y, -0^z} must end in -0. If it must be a supersequence, the only possibility is "+0-0-0+0-0". Taking one more step on the left, {+0^c, -0^y, -0^z} must linearize to "+0-0-0+0-0+0". Thus the final state {-0^d, -0^e, -0^y, -0^z} must linearize to "+0 -0-0+0-0+0-0-0". Reasoning symmetrically, the linearization of {-0^d, -0^e, +0^x} must be "+0-0+0-0-0+0", and thus the final {-0^d, -0^e, -0^y, -0^z} must linearize to "+0-0+0-0-0+0-0-0". The constraints on the final state are incompatible. Each of these states can be verified in isolation; it is the relation between them that is not satisfiable.

Recall that monotonicity requires that the linearization chosen for a frontier be a *subsequence* of the linearization chosen for any extension of that frontier. The difficulty here is that subsequence relation ignores the similarity between "+0 -0-0+0-0+0-0-0" and "+0-0+0-0-0+0-0-0". Neither of these is a subsequence of the other, yet they capture exactly the same sequence of *states*, each with six alternations between ✗0 and ✓0. The canonical state-based representative for these sequences is "+0-0+0-0+0-0".

CRDTs are defined in terms of states. In order to relate CRDTs to sequential specifications, it is necessary to extract information about states from the specification itself. Adapting Brookes [1996], we define strings as *stuttering equivalent* (notation $\sigma \sim \tau$) if they pass through the same states. So +0+1+0 $\sim$ +0+1 but +0-0+0 $\not\sim$ +0. If we consider subsequences up to stuttering, then execution (7) is SEC, with witnesses as follow:

$$\begin{array}{ll}
\{a\}, \{x\}, \{a, x\} & : \texttt{+0} \\
\{b\}, \{y\}, \{y, z\}, \{z\} & : \texttt{+0-0} \\
\{a, y\}, \{a, y, z\}, \{a, z\}, \{b, x\} & : \texttt{+0-0+0} \\
\{b, y\}, \{b, y, z\}, \{b, z\}, \{d\}, \{d, e\}, \{e\} & : \texttt{+0-0+0-0} \\
\{c, y\}, \{c, y, z\}, \{c, z\}, \{d, x\}, \{d, e, x\}, \{e, x\} & : \texttt{+0-0+0-0+0} \\
\{d, y\}, \{d, y, z\}, \{d, z\}, & \\
\{e, y\}, \{e, y, z\}, \{e, z\}, \{d, e, y\}, \{d, e, y, z\}, \{d, e, z\} & : \texttt{+0-0+0-0+0-0}
\end{array}$$

Recall that without stuttering, we deduced that {+0^c, -0^y, -0^z} must linearize to "+0-0-0+0-0+0" and {-0^d, -0^e, +0^x} must linearize to "+0-0+0-0-0+0". Under stuttering equivalence, these are the same, with canonical representative "+0 -0+0-0+0". Thus, monotonicity under stuttering allows both linearizations to be extended to satisfy the final state {-0^d, -0^e, -0^y, -0^z}, which has canonical representative "+0-0+0-0+0-0".

Summary: CRDTs are described in terms of convergent states, whereas specifications are described as strings of actions. Actions correspond to labels in the LPO of an execution. Many strings of actions may lead to equivalent states. For example, idempotent actions can be applied repeatedly without modifying the state.

The stuttering equivalence of Brookes [1996] addresses this mismatch. In order to capture the validity of CRDTs, the definition of subsequence must change from a definition over individual specification strings to a definition over *equivalence classes* of strings *up to stuttering*.

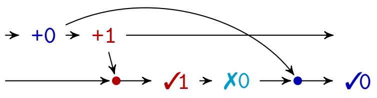
3 Eventual Consistency for CRDTs

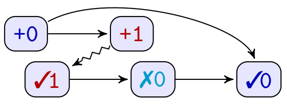
This section formalizes the intuitions developed in Section 2. We define executions, specifications and strong eventual consistency (SEC). We discuss properties of eventual consistency and prove that the add-wins set is SEC.

3.1 Executions

An execution realizes *causal delivery* if, whenever an event is received at a replica, all predecessors of the event are also received. Most of the CRDTs in [Shapiro et al. 2011a] assume causal delivery, and we assumed it throughout the introductory section. There are costs to maintaining causality, however, and not all CRDTs assume that executions incur these costs. In the formal development, we allow non-causal executions.

Shapiro et al. [2011a] draw executions as timelines, explicitly showing the delivery of remote mutators. Below left, we give an example of such a timeline.





This is a non-causal execution: at the bottom replica, +1 is received before +0, even though +0 precedes +1 at the top replica.

Causal executions are naturally described as Labelled Partial Orders (LPOs), which are transitive and antisymmetric. Section 2 presented several examples of LPOs. To capture non-causal systems, we move to *Labelled Visibility Orders* (LVOs), which are merely acyclic. Acyclicity ensures that the transitive closure of an LVO is an LPO. The right picture above shows the LVO corresponding to the timeline on the left. The zigzag arrow represents an intransitive communication. When drawing executions, we use straight lines for "transitive" edges, with the intuitive reading that "this and all preceding actions are delivered".

LVOs arise directly due to non-causal implementations. As we will see in Section 4, they also arise via projection from an LPO.

LVOs are unusual in the literature. To make this paper self-contained, we define the obvious generalizations of concepts familiar from LPOs, including isomorphism, suborder, restriction, maximality, downclosure and cut.

Fix a set $\mathbf{L}$ of labels. A *Labelled Visibility Order* (LVO, also known as an *execution*) is a triple $u = \langle \mathsf{E}_u, \lambda_u, \rightsquigarrow_u \rangle$ where E_u is a finite set of events, $\lambda_u \in (\mathsf{E}_u \mapsto \mathbf{L})$ and $\rightsquigarrow_u \subseteq (\mathsf{E}_u \times \mathsf{E}_u)$ is reflexive and acyclic.

Let u, v range over LVOs. Many concepts extend smoothly from LPOs to LVOs.

- *Isomorphism:* Write $u =_{\mathsf{iso}} v$ when u and v differ only in the carrier set. We are often interested in the isomorphism class of an LVO.
- *Pomset:* We refer to the isomorphism class of an LVO as a *pomset*. Pomset abbreviates *Partially Ordered Multiset* [Plotkin and Pratt 1997]. We stick with the name "pomset" here, since "vomset" is not particularly catchy.

- *Suborder:* Write $u \subseteq v$ when $\mathsf{E}_u \subseteq \mathsf{E}_v$, $\lambda_u \subseteq \lambda_v$, $\rho_u \subseteq \rho_v$, and $(\rightsquigarrow_v) \subseteq (\rightsquigarrow_u)$.
- *Restriction:*[1] When $D \subseteq \mathsf{E}_v$, define $v \downharpoonright D = \langle D,\, \lambda_v \downharpoonright D,\, \rightsquigarrow_v \downharpoonright D\rangle$.
  Restriction lifts subsets to suborders: $v \downharpoonright D$ denotes the sub-LVO derived from a subset D of events. See Section 2.2 for an example of restriction.
- *Maximal elements:* $\mathsf{max}(v) = \{d \in \mathsf{E}_v \mid \nexists e \in (\mathsf{E}_v \setminus \{d\}).\ d \rightsquigarrow_v e\}$.
  We say that d is maximal for v when if $d \in \mathsf{max}(v)$.
- *Non-maximal suborder:* $\overline{\mathsf{max}}(v) = v \downharpoonright (\mathsf{E}_v \setminus \mathsf{max}(v))$.
  $\overline{\mathsf{max}}(v)$ is the suborder with the maximal elements removed.
- *Downclosure:* D is *downclosed* for v if $D \subseteq \{e \in \mathsf{E}_v \mid \exists d \in D.\ d \rightsquigarrow_v e\}$.
- *Cut:* u is a *cut* of v if $u \subseteq v$ and E_u is downclosed for v.
  Let $\mathsf{cuts}(v)$ be the set of all cuts of v. A cut is the sub-LVO corresponding to a downclosed set. Cuts are also known as prefixes. See Section 2.4 for an example. A cut is determined by its maximal elements: if $u \in \mathsf{cuts}(v)$ then $u = v \downharpoonright \{d \in \mathsf{E}_v \mid \exists e \in \mathsf{max}(v).\ d \rightsquigarrow_v e\}$.
- *Linearization:* For $a_i \in \mathbf{L}$, we say that $a_1 \dots a_n$ is a *linearization* of $E \subseteq \mathsf{E}_v$ if there exists a bijection $\alpha : E \rightarrow [1,\, n]$ such that $\forall e \in E.\ \lambda_v(e) = a_{\alpha(e)}$ and $\forall d,\, e \in E.\ d \rightsquigarrow_v e$ implies $\alpha(d) \leq \alpha(e)$.

Replica-Specific Properties. In the literature on replicated data types, some properties of interest (such as "read your writes" [Tanenbaum and Steen 2007]) require the concept of "session" or a distinction between local and remote events. These can be accommodated by augmenting LVOs with a replica labelling $\rho_u \in (\mathsf{E}_u \mapsto \mathbf{R})$, which maps events to a set $\mathbf{R}$ of *replica identifiers.*

Executions can be generated operationally as follows: Replicas receive mutator and accessor events from the local client; they also receive mutator events that are forwarded from other replicas. Each replica maintains a set of *seen* events: an event that is received is added to this set. When an event is received from the local client, the event is additionally added to the execution, with the predecessors in the visibility relation corresponding to the current *seen* set. If we wish to restrict attention to causal executions, then we require that replicas forward all the mutators in their *seen* sets, rather than individual events, and, thus, the visibility relation is transitive over mutators.

All executions that are operationally generated satisfy the additional property that $\rightsquigarrow_u$ is per-replica total: if $\rho(d) = \rho(e)$ then either $d \rightsquigarrow_u e$ or $e \rightsquigarrow_u d$. We do not demand per-replica totality because our results do not rely on replica-specific information.

3.2 Specifications and Stuttering Equivalence

Specifications are sets of strings, equipped with a distinguished set of mutators and a dependency relation between labels. Specifications are subject to some constraints to ensure that the mutator set and dependency relations are sensible;

[1] We use the standard definitions for restriction on functions and relations. Given a function $f : E \rightarrow X, \mathcal{R}$: $E \times E$ and $D \subseteq E$, define $f \downharpoonright D = \{\langle d,\, f(d)\rangle \mid d \in D\}$. and $\mathcal{R} \downharpoonright D = \{\langle d_1,\, d_2\rangle \mid d_1,\, d_2 \in D$ and $d_1\ \mathcal{R}\ d_2\}$.

these are inspired by the conditions on Mazurkiewicz executions [Diekert and Rozenberg 1995]. Every specification set yields a derived notion of stuttering equivalence. This leads to the definition of *observational subsequence* ($\leq_{\mathsf{obs}}$).

We use standard notation for strings: Let σ and τ range over strings. Then $\sigma\tau$ denotes concatenation, σ^* denotes Kleene star, $\sigma \mathbin{|||} \tau$ denotes the set of interleavings, ε denotes the empty string and σ^i denotes the ith element of σ. These notations lift to sets of strings via set union.

A *specification* is a quadruple $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$ where

– $\mathbf{L}$ is a set of *actions* (also known as *labels*),
– $\mathbf{M} \subseteq \mathbf{L}$ is a distinguished set of *mutator* actions,
– $\# \subseteq (\mathbf{L} \times \mathbf{L})$ is a symmetric and reflexive *dependency* relation, and
– $\Sigma \subseteq \mathbf{L}^*$ is a set of *valid strings.*

Let $\overline{\mathbf{M}} = \mathbf{L} \setminus \mathbf{M}$ be the sets of *non-mutators.*

A specification must satisfy the following properties:

(a) prefix closed: $\sigma\tau \in \Sigma$ implies $\sigma \in \Sigma$
(b) non-mutators are closed under stuttering, and commutation:
$\forall a \in \overline{\mathbf{M}}\,.\ \sigma a \tau \in \Sigma$ implies $\sigma a^* \tau \subseteq \Sigma$
$\forall a, b \in \overline{\mathbf{M}}\,.\ \{\sigma a, \sigma b\} \subseteq \Sigma$ implies $\{\sigma ab, \sigma ba\} \subseteq \Sigma$
(c) independent actions commute:
$\forall a, b \in \mathbf{L}.\ \neg(a \# b)$ implies $(\sigma ab\tau \in \Sigma$ iff $\sigma ba\tau \in \Sigma)$

Property (b) ensures that non-mutators do not affect the state of the data structure. Property (c) ensures that commuting of independent actions does not affect the state of the data structure.

Recall that the SET specification takes $\mathbf{M} = \{$+0, -0, +1, -1$\}$, representing addition and removal of bits 0 and 1, and $\overline{\mathbf{M}} = \{$✗0, ✓0, ✗1, ✓1$\}$, representing membership tests returning false or true. The dependency relation is $\# = \{$+0, -0, ✗0, ✓0$\}^2 \cup \{$+1, -1, ✗1, ✓1$\}^2$, where $D^2 = D \times D$.

The dependency relation for SET is an equivalence, but this need not hold generally. We will see an example in Section 4.

The definitions in the rest of the paper assume that we have fixed a specification $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$. In the examples of this section, we use SET.

State and Stuttering Equivalence. Specification strings σ and τ are *state equivalence* (notation $\sigma \approx \tau$) if every valid extension of σ is also a valid extension of τ, and vice versa. For example, +0+1+0 $\approx$ +0+1 and +0-0+0 $\approx$ +0, but +0-0 $\not\approx$ +0. In particular, state equivalent strings agree on the valid accessors that can immediately follow them: either ✓0 or ✗0 and either ✓1 or ✗1. Formally, we define state equivalence, $\approx \subseteq \mathbf{L}^* \times \mathbf{L}^*$, as follows[2].

$$(\sigma \approx \sigma') \triangleq (\sigma = \sigma') \text{ or } (\{\sigma, \sigma'\} \subseteq \Sigma \text{ and } \forall \tau \in \mathbf{L}^*.\ \sigma\tau \in \Sigma \text{ iff } \sigma'\tau \in \Sigma).$$

From specification property (b), we know that non-mutators do not affect the state. Thus we have that $ua \approx u$ whenever $a \in \overline{\mathbf{M}}$ and $ua \in \Sigma$. From specification

[2] To extend the definition to non-specification strings, we allow $\sigma \approx \sigma'$ when $\sigma = \sigma'$.

property (c), we know that independent actions commute. Thus we have that $\sigma ab \approx \sigma ba$ whenever $\neg(a \# b)$ and $\{\sigma ab, \sigma ba\} \subseteq \Sigma$

Two strings are *stuttering equivalent*[3] if they only differ in operations that have no effect on the state of the data structure, as given by Σ. Adapting Brookes [1996] to our notion of state equivalence, we define stuttering equivalence, $\sim \subseteq \mathbf{L}^* \times \mathbf{L}^*$, to be the least equivalence relation generated by the following rules, where a ranges over $\mathbf{L}$.

$$\frac{}{\varepsilon \sim \varepsilon} \qquad \frac{\sigma \sim \sigma'}{\sigma a \sim \sigma' a} \qquad \frac{\sigma \approx \sigma a}{\sigma \sim \sigma a} \qquad \frac{\sigma b \sim \sigma \quad \neg(a \# b)}{\sigma a b \sim \sigma a}$$

The first rule above handles the empty string. The second rule allows stuttering in any context. The third rule motivates the name stuttering equivalence, for example, allowing $\texttt{+0+0} \sim \texttt{+0}$. The last case captures the equivalence generated by independent labels, for example, allowing $\texttt{+0+1+0} \sim \texttt{+0+1}$ but not $\texttt{+0-0+0} \sim \texttt{+0-0}$. Using the properties of $\approx$ discussed above, we can conclude, for example, that $\texttt{+0✓0✓0+0-0✗0} \sim \texttt{+0-0}$.

Consider specification strings for a unary SET over value 0. Since stuttering equivalence allows us to remove both accessors and adjacent mutators with the same label we deduce that the *canonical representatives* of the equivalence classes induced by $\sim$ are generated by the regular expression $(\texttt{+0})^?(\texttt{-0+0})^*(\texttt{-0})^?$.

Observational Subsequence. Recall that ac is a *subsequence* of abc, although it is not a *prefix*. We write $\leq_{\mathsf{seq}}$ for subsequence and $\leq_{\mathsf{obs}}$ for *observational subsequence*, defined as follows.

$$\sigma_1 \cdots \sigma_n \leq_{\mathsf{seq}} \tau_0 \sigma_1 \tau_1 \cdots \sigma_n \tau_n \qquad \sigma \leq_{\mathsf{obs}} \tau \text{ if } \exists \sigma' \sim \sigma.\ \exists \tau' \sim \tau.\ \sigma' \leq_{\mathsf{seq}} \tau'$$

Note that observational subsequence includes both subsequence and stuttering equivalence $(\leq_{\mathsf{obs}}) \subseteq (\leq_{\mathsf{seq}}) \cup (\sim)$.

$\leq_{\mathsf{seq}}$ can be understood in isolation, whereas $\leq_{\mathsf{obs}}$ can only be understood with respect to a given specification. In the remainder of the paper, the implied specification will be clear from context. $\leq_{\mathsf{seq}}$ is a partial order, whereas $\leq_{\mathsf{obs}}$ is only a preorder, since it is not antisymmetric.

Let σ and τ be strings over the unary SET with canonical representatives $a\sigma'$ and $b\tau'$. Then we have that $\sigma \leq_{\mathsf{obs}} \tau$ exactly when either $a = b$ and $|\sigma'| \leq |\tau'|$ or $a \neq b$ and $|\sigma'| < |\tau'|$. Thus, observational subsequence order is determined by the number of alternations between the mutators.

Specification strings for the binary SET, then, are stuttering equivalent exactly when they yield the same canonical representatives when restricted to 0 and to 1. Thus, observational subsequence order is determined by the number of alternations between the mutators, when restricted to each dependent subsequence. (The final rule in the definition of stuttering, which allows stuttering across independent labels, is crucial to establishing this canonical form.)

[3] Readers of Brookes [1996] should note that mumbling is not relevant here, since all mutators are visible.

3.3 Eventual Consistency

Eventual consistency is defined using the *cuts* of an execution and the *observational subsequence order* of the specification. As noted in Sections 2.2 and 2.4, it is important that we not consider all cuts. Thus, before we define SEC, we must define *dependent cuts.*

The *dependent restriction* of an execution is defined: $v \downharpoonleft \# = \langle \mathsf{E}_v, \lambda_v, \overset{\#}{\rightsquigarrow}_v \rangle$, where $d \overset{\#}{\rightsquigarrow}_v e$ when $\lambda_v(d) \# \lambda_v(e)$ and $d \rightsquigarrow_v e$. See Section 2.2 for an example of dependent restriction.

The *dependent cuts* of v are cuts of the dependent restriction. As discussed in Section 2.4, we only consider pointed cuts (with a single maximal element) for non-mutators. See Section 2.4 for an example.

$$\mathsf{cuts}_{\#}(v) = \left\{ u \in \mathsf{cuts}(v \downharpoonleft \#) \;\middle|\; \forall e \in \mathsf{E}_u.\ \text{if } \lambda_u(e) \in \overline{\mathbf{M}} \text{ then } \mathsf{max}(u) = \{e\} \right\}$$

An execution v is *Eventually Consistent* (SEC) for specification $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$ iff there exists a function $\tau : \mathsf{cuts}_{\#}(v) \to \Sigma$ that satisfies the following.

Linearization: $\forall p \in \mathsf{cuts}_{\#}(v).\ p$ linearizes to $\tau(p)$, and
Monotonicity: $\forall p, q \in \mathsf{cuts}_{\#}(v).\ p \subseteq q$ implies $\tau(p) \leq_{\mathsf{obs}} \tau(q)$.

A data structure implementation is SEC if all of its executions are SEC.

In Section 2, we gave several examples that are SEC. See Sections 2.4 and 2.5 for examples where τ is given explicitly. Section 2.4 also includes an example that is not SEC.

The concerns raised in Section 2 are reflected in the definition.

- Non-mutators are ignored by the dependent restriction of other non-mutators. As discussed in Section 2.1, this relaxation is similar that of update-serializability [Hansdah and Patnaik 1986; Garcia-Molina and Wiederhold 1982].
- Independent events are ignored by the dependent restriction of an event. As discussed in Section 2.2, this relaxation is similar to preserved program order in relaxed memory models [Higham and Kawash 2000; Alglave 2012].
- As discussed in Section 2.3, punning is allowed: each cut p is linearized separately to a specification string $\tau(p)$.
- As discussed in Section 2.4, we constrain the power puns by considering cuts of the distributed system [Chandy and Lamport 1985].
- Monotonicity ensures that the system evolves in a sensible way: new order may be introduced, but old order cannot be forgotten. As discussed in Section 2.5, the preserved order is captured in the observational subsequence relation, which allows stuttering [Brookes 1996].

3.4 Properties of Eventual Consistency

We discuss some basic properties of SEC. For further analysis, see Section 5.

An important property of CRDTs is *prefix closure*: If an execution is valid, then every prefix of the execution should also be valid. Prefix closure follows immediately from the definition, since whenever u is a prefix of v we have that $\mathsf{cuts}_{\#}(u) \subseteq \mathsf{cuts}_{\#}(v)$.

Prefix closure looks back in time. It is also possible to look forward: A system satisfies *eventual delivery* if every valid execution can be extended to a valid execution with a maximal element that sees every mutator. If one assumes that every specification string can be extended to a longer specification string by adding non-mutators, then eventual delivery is immediate.

The properties PSTS, PSM and PPE are discussed in the introduction. An SEC implementation must satisfies PPE since every dependent set of mutators is linearized: SEC enforces the stronger property that there are no new intermediate states, even when executing all mutators in parallel. For causal systems, where $\rightsquigarrow_u$ is transitive, PSTS and PSM follow by observing that if there is a total order on the mutators of u then any linearization of u is a specification string.

Burckhardt [2014, §5] provides a taxonomy of correctness criteria for replicated data types. Our definition implies NoCircularCausality and CausalArbitration, but does not imply either ConsistentPrefix or CausalVisibility. For LPOs, which model causal systems, our definition implies CausalVisibility. ReadMyWrites and MonotonicReads require a distinction between local and remote events. If one assumes the replica-specific constraints given in Section 3.1, then our definition satisfies these properties; without them, our definition is too abstract.

3.5 Correctness of the Add-Wins Set

The add-wins set is defined to answer ✓k for a cut u exactly when

$$\exists d \in u.\ \lambda_u(d) = \texttt{+k} \ \wedge\ (\nexists e \in u.\ \lambda_u(e) = \texttt{-k} \ \wedge\ d \rightsquigarrow_u e).$$

It answers ✗k otherwise. The add-wins set is called the "observed-remove" set.

We show that any LPO that meets this specification is SEC with respect to SET. We restrict attention to LPOs since causal delivery is assumed for the add-wins set in [Shapiro et al. 2011a].

For SET, the dependency relation is an equivalence. For an equivalence relation R, let $\mathbf{L}/R \subseteq 2^{\mathbf{L}}$ denote the set of (disjoint) equivalence classes for R. For SET, $\mathbf{L}/\# = \{\{\texttt{+0}, \texttt{-0}, ✗\texttt{0}, ✓\texttt{0}\}, \{\texttt{+1}, \texttt{-1}, ✗\texttt{1}, ✓\texttt{1}\}\}$. When dependency is an equivalence, then *every* interleaving of independent actions is valid if *any* interleaving is valid. Formally, we have the following, where $|||$ denotes interleaving.

$$\forall D \in (\mathbf{L}/\#).\ \forall \sigma \in D^*.\ \forall \tau \in (\mathbf{L} \setminus D)^*.\ (\sigma \mathbin{|||} \tau) \cap \Sigma \neq \emptyset \text{ implies } (\sigma \mathbin{|||} \tau) \subseteq \Sigma$$

Using the forthcoming composition result (Theorem 2), it suffices for us to address the case when u only involves operations on a single element, say 0. For any such LVO u, we choose a linearization $\tau(u) \in (\texttt{-0}|\texttt{+0})^*$ that has a maximum number of alternations between $\texttt{-0}$ and $\texttt{+0}$. If there is a linearization that begins with $\texttt{-0}$, then we choose one of these. Below, we summarize some of the key properties of such a linearization.

– $\tau(u)$ ends with `+0` iff there is an `+0` that is not followed by any `-0` in u.
– For any LPO $v \subseteq u$, $\tau(v)$ has at most as many alternations as $\tau(u)$.

The first property above ensures that the accessors are validated correctly, *i.e.*, 0 is deemed to be present iff there is an `+0` that is not followed by any `-0`.

We are left with proving monotonicity, *i.e.*, if $u \subseteq v$, then $\tau(u) \leq_{\mathsf{obs}} \tau(v)$. Consider $\tau(u) = a\sigma$ and $\tau(v) = b\rho$.

– If $b = a$, the second property above ensures that $\tau(u) \leq_{\mathsf{obs}} \tau(v)$.
– In the case that $b \neq a$, we deduce by construction that $b =$ `-0` and $a =$ `+0`. In this case, ρ starts with `+0` and has at least as many alternations as $\tau(u)$. So, we deduce that $\tau(u) \leq_{\mathsf{obs}} \rho$. The required result follows since $\rho \leq_{\mathsf{obs}} \tau(v)$.

4 A Collaborative Text Editing Protocol

In this section we consider a variant of the collaborative text editing protocol defined by Attiya et al. [2016]. After stating the sequential specification, TEXT, we sketch a correctness proof with respect to our definition of eventual consistency. This example is interesting formally: the dependency relation is not an equivalence, and therefore the dependent projection does not preserve transitivity. The generality of intransitive LVOs is necessary to understand TEXT, even assuming a causal implementation.

Specification. Let a, b range over *nodes*, which contain some text, a unique identifier, and perhaps other information. Labels have the following forms:

– Mutator `!`a initializes the text to node a.
– Mutator `+`a`<`b adds node a immediately before node b.
– Mutator `+`a`>`b adds node a immediately after node b.
– Mutator `-`b removes node b.
– Non-mutator query `?`$b_1 \cdots b_n$ returns the current state of the document.

We demonstrate the correct answers to queries by example. Initially, the document is empty, whereas after initialization, the document contains a single node; thus the specification contains strings such as "`?`ε `!`c `?`c", where ε represents the empty document. Nodes can be added either before or after other nodes; thus "`!`c `+`b`<`c `+`d`>`c" results in the document `?`bcd. Nodes are always added adjacent to the target; thus, order matters in "`!`c `+`e`>`c `+`d`>`c" which results in `?`cde rather than `?`ced. Removal does what one expects; thus "`!`c `+`e`>`c `+`d`>`c `-`c" results in `?`de.

Attiya et al. define the interface for TEXT using integer indices as targets, rather than nodes. Using the unique correspondence between the nodes and it indices (since node are unique), one can easily adapt an implementation that satisfies our specification to their interface.

We say that node a is a *added* in the actions `!`a, `+`a`<`b and `+`a`>`b. Node b is a *target* in `+`a`<`b and `+`a`>`b. In addition to correctly answering queries, specifications must satisfy the following constraints:

– Initialization may occur at most once,
– each node may be added at most once,
– a node may be removed only after it is added, and
– a node may be used as a target only if it has been added and not removed.

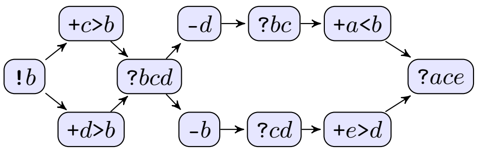
These constraints forbid adding to a target that has been removed; thus "$\texttt{!}c$ $\texttt{+}d\texttt{>}c$ $\texttt{-}c$" is a valid string, but "$\texttt{!}c$ $\texttt{-}c$ $\texttt{+}d\texttt{>}c$" is not. It also follows that initialization must precede any other mutators.

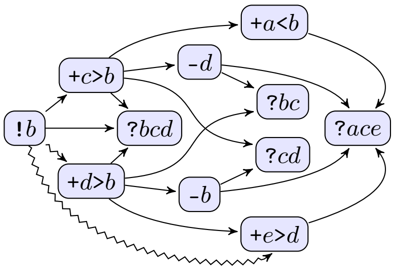
Because add operations use unique identifiers, punning and stuttering play little role in this example. In order to show the implementation correct, we need only choose an appropriate notion of dependency. As we will see, it is necessary that removes be independent of adds with disjoint label sets, but otherwise all actions may be dependent. Let $\mathbf{L}_{\texttt{!+?}}$ be the set of add and query labels, and let nodes return the set of nodes that appear in a label. Then we define dependency as follows.

$$\ell \mathbin{\#} k \text{ iff } \{\ell,\, k\} \subseteq \mathbf{L}_{\texttt{!+?}} \text{ or } \mathsf{nodes}(\ell) \cap \mathsf{nodes}(k) \neq \emptyset$$

Implementation. We consider executions that satisfy the same four conditions above imposed on specifications. We refer the reader to the algorithm of Attiya et al. that provides timestamps for insertions that are monotone with respect to causality.

As an example, Attiya et al. allow the execution given on the left below. In this case, the dependent restriction is an intransitive LVO, even though the underlying execution is an LPO: in particular, $\texttt{!}b$ does not precede $\texttt{-}d$ in the dependent restriction. We give the order considered by dependent cuts on the right—this is a restriction of the dependent restriction: since we only consider pointed accessor cuts, we can safely ignore order out of non-mutators.


$$\begin{tikzcd}
 & \texttt{+}c\texttt{>}b \arrow[dr] & & \texttt{-}d \arrow[r] & \texttt{?}bc \arrow[r] & \texttt{+}a\texttt{<}b \arrow[dr] & \\
\texttt{!}b \arrow[ur] \arrow[dr] & & \texttt{?}bcd \arrow[ur] \arrow[dr] & & & & \texttt{?}ace \\
 & \texttt{+}d\texttt{>}b \arrow[ur] & & \texttt{-}b \arrow[r] & \texttt{?}cd \arrow[r] & \texttt{+}e\texttt{>}d \arrow[ur] &
\end{tikzcd}$$



This execution is not linearizable, but it is SEC, choosing witnesses to be subsequences of the mutator string "$\texttt{!}b$ $\texttt{+}d\texttt{>}b$ $\texttt{+}c\texttt{>}b$ $\texttt{+}a\texttt{<}b$ $\texttt{+}e\texttt{>}d$ $\texttt{-}b$ $\texttt{-}d$". Here, the document is initialized to b, then c and d are added after b, resulting in $\texttt{?}bcd$. The order of c and d is determined by their timestamps. Afterwards, the top replica removes d and adds a; the bottom replica removes b and adds e, resulting in the final state $\texttt{?}ace$. In the right execution, the removal of order out of the non-mutators shows the "update serializability" effect; the removal of order between $\texttt{-}b$ and $\texttt{+}e\texttt{>}d$ (and between $\texttt{-}d$ and $\texttt{+}a\texttt{<}b$) shows the "preserved program order" effect.

Correctness. Given an execution, we can find a specification string $s_1 s_2$ that linearizes the mutators in the dependent restriction of the execution such that s_1 contains only adds and s_2 contains only removes. Such a specification string exists because by the conditions on executions, deletes do not have any outgoing edges to other mutators in the dependent restriction; so, they can be moved to the end in the matching specification string. In order to find s_1 that linearizes the add events, any linearization that respects causality and timestamps (yielded by the algorithm of Attiya et al.) suffices for our purposes. The conditions required by SEC follow immediately.

5 Compositional Reasoning

The aim of this section is to establish compositional methods to reason about replicated data structures. We do so using *Labelled Transition Systems* (LTSs), where the transitions are labelled by dependent cuts. We show how to derive an LTS from an execution, $\mathsf{lts}(u)$. We also define an LTS for the *most general* CRDT that validates a specification, $\mathsf{lts}(\Sigma)$. We show that u is SEC for Σ exactly when $\mathsf{lts}(u)$ is a refinement of $\mathsf{lts}(\Sigma)$. We use this alternative characterization to establish composition and abstraction results.

LTSs. An LTS is a triple consisting of a set a states, an initial state and a labelled transition function between states. We first define the LTSs for executions and specifications, then provide examples and discussion.

For both executions and specifications, the labels of the LTS are dependent cuts: for executions, these are dependent cuts of the execution itself; for specifications, they are drawn from the set $\mathcal{L}_\# = \bigcup_{v \in \mathcal{L}} \mathsf{cuts}_\#(v)$ of all possible dependent cuts. We compare LTS labels up to isomorphism, rather than identity. Thus it is safe to think of LTS labels as (potentially intransitive) pomsets [Plotkin and Pratt 1997].

The states of the LTS are different for the execution and specification. For executions, the states are cuts of the execution u itself, $\mathsf{cuts}(u)$; these are general cuts, not just dependent cuts. For specifications, the states are the stuttering equivalence classes of strings allowed by the specification, $\Sigma/{\sim}$.

There is an isomorphism between strings and total orders. We make use of this in the definition, treating strings as totally-ordered LVOs.

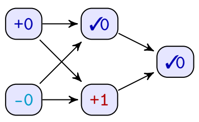
Define $\mathsf{lts}(u) = \langle \mathsf{cuts}(u), \emptyset, \longmapsto_{\mathsf{i}} \rangle$, where $p \overset{v}{\longmapsto}_{\mathsf{i}} q$ if $v \in \mathsf{cuts}_\#(q)$ and

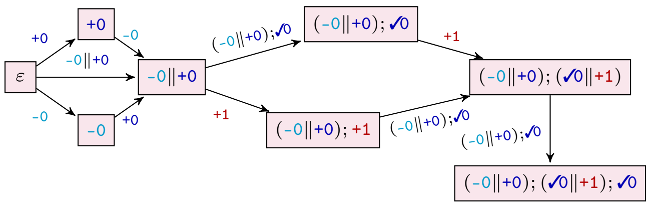
$$\begin{array}{lll}
p \subseteq q & \mathsf{E}_{\mathsf{max}(v)} \cup \mathsf{E}_p = \mathsf{E}_q & \overline{\mathsf{max}}(v) \subseteq p \\
v \subseteq q & \mathsf{E}_{\mathsf{max}(v)} \cap \mathsf{E}_p = \emptyset & \mathsf{E}_{\mathsf{max}(v)} \subseteq \mathsf{E}_{\mathsf{max}(q)}
\end{array}$$

Define $\mathsf{lts}(\Sigma) = \langle \Sigma/{\sim}, \varepsilon, \longmapsto_{\mathsf{s}} \rangle$, where $[\sigma] \overset{v}{\longmapsto}_{\mathsf{s}} [\rho]$ if $v \in \mathcal{L}_\#$ and

$$\begin{array}{lll}
\sigma \subseteq \rho & \mathsf{E}_{\mathsf{max}(v)} \cup \mathsf{E}_\sigma = \mathsf{E}_\rho & \overline{\mathsf{max}}(v) \subseteq \sigma \\
v \subseteq \rho & \mathsf{E}_{\mathsf{max}(v)} \cap \mathsf{E}_\sigma = \emptyset &
\end{array}$$

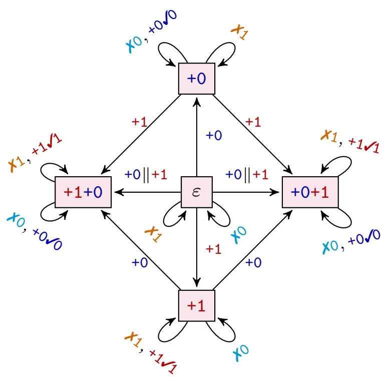
We explain the definitions using examples from SET, first for executions, then for specifications. Consider the execution on the left below. The derived LTS is given on the right.

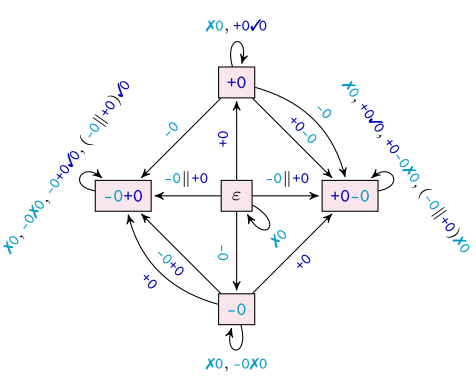




The states of the LTS are cuts of the execution. The labels on transitions are *dependent* cuts. The requirements for execution transitions relate the source p, target q and label v. The leftmost requirements state that the target state must extend both the source and the label; thus the target state must be a combination of events and order from source and label. The middle requirements state that the maximal elements of the label must be new in the target; only the maximal elements of the label are added when moving from source to target. The upper right requirement states that the non-maximal order of the label must be respected by the source; thus the causal history reported by the label cannot contradict the causal history of the source. The lower right requirement ensures that maximal elements of the label are also maximal in the target. The restriction to dependent cuts explains the labels on transitions $(\texttt{-0}\|\texttt{+0}) \xmapsto{\texttt{+1}}_{¡} (\texttt{-0}\|\texttt{+0}); \texttt{+1}$ and $(\texttt{-0}\|\texttt{+0}); (\checkmark\texttt{0}\|\texttt{+1}); \checkmark\texttt{0} \xmapsto{(\texttt{-0}\|\texttt{+0});\checkmark\texttt{0}}_{¡} (\texttt{-0}\|\texttt{+0}); (\checkmark\texttt{0}\|\texttt{+1})$. By definition, there is a self-transition labelled with the empty LVO at every state; we elide these transitions in drawings.

The specification LTS for SET is infinite, of course. To illustrate, below we give two sub-LTSs with limitations on mutators. On the left, we only allow +0 and +1. On the right, we only allow +0 and -0 and only consider the case in which there is at most one alternation between them. The states are shown using their canonical representatives. Because of the number of transitions, we show all dependent accessors as a single transition, with labels separated by commas.





The requirements for specification transitions are similar to those for implementations, but the states are equivalence classes over specification strings: with source $[\sigma]$ and target $[\tau]$. There is a transition between the states if there are members of the equivalence classes, σ and τ, that satisfy the requirements. Since these are total orders, the leftmost requirements state that there must be linearizations of the source and label that are subsequences of the target. Similarly, the upper right requirement states that the non-maximal order of the label must be respected by the source; thus we have $\mathsf{+0} \xmapsto{\mathsf{+0-0}}_{\mathsf{s}} \mathsf{+0-0}$ but not $\mathsf{+0} \xmapsto{\mathsf{-0+0}}_{\mathsf{s}} \sigma$, for any σ. The use of sub-order rather than subsequence allows $\mathsf{+0-0} \xmapsto{\mathsf{+0-0}}_{\mathsf{s}} \mathsf{+0-0-0}$ but prevents nonsense transitions such as $\mathsf{+0-0} \xmapsto{\mathsf{+0-0}}_{\mathsf{s}} \mathsf{-0+0-0}$. Because the states are total orders, we drop the implementation LTS requirement that maximal events of the label must be maximal in the target. If we were to impose this restriction, we would disallow $\mathsf{-0} \xmapsto{\mathsf{+0}}_{\mathsf{s}} \mathsf{+0-0}$.

It is worth noting that the specification of the add-wins set removes exactly three edges from the right LTS: $\varepsilon \xmapsto{\mathsf{-0|+0}}_{\mathsf{s}} \mathsf{+0-0}$, $\mathsf{+0} \xmapsto{\mathsf{-0}}_{\mathsf{s}} \mathsf{+0-0}$, and $\mathsf{-0} \xmapsto{\mathsf{+0}}_{\mathsf{s}} \mathsf{+0-0}$.

Refinement. Refinement is a functional form of simulation [Hoare 1972; Lamport 1983; Lynch and Vaandrager 1995]. Let $P = \langle S_P, p_0, \longmapsto_P \rangle$ and $Q = \langle S_Q, q_0, \longmapsto_Q \rangle$ be LTSs. A function $f : S_P \to S_Q$ is a *(strong) refinement* if $p \xmapsto{v}_P p'$ and $f(p) = q$ imply that there exist $w =_{\mathsf{iso}} v$ and $q' \in S_Q$ such that $q \xmapsto{w}_Q q'$ and $f(p') = q'$. Then P *refines* Q (notation $P \sqsubseteq Q$) if there exists a refinement $f : S_P \to S_Q$ such that the initial states are related, *i.e.*, $f(p_0) = q_0$.

We now prove that SEC can be characterized as a refinement. We write $p_0 \longmapsto_P^* p_n$ when p_n is reachable from p_0 via a finite sequence of steps $p_i \xmapsto{u_i}_P p_{i+1}$.

Theorem 1. *u is* EC *for the specification Σ iff* $\mathsf{lts}(u) \sqsubseteq \mathsf{lts}(\Sigma)$.

Proof. For the forward direction, assume u is EC and therefore there exists a function $\tau : \mathsf{cuts}_{\#}(u) \to \Sigma$ such that $\forall E \in \mathsf{cuts}_{\#}(u)$. $\tau(E)$ is a linearization of E. For each cut $p \in \mathsf{cuts}(u)$, we start with the dependent restriction, $p \downharpoonleft \#$. We further restriction attention to mutators, $p \downharpoonleft \# \downharpoonleft \mathbf{M}$. The required refinement maps p to the equivalence class of the linearization of $p \downharpoonleft \# \downharpoonleft \mathbf{M}$ chosen by τ: $f(p) \triangleq [\tau(p \downharpoonleft \# \downharpoonleft \mathbf{M})]$. We abuse notation below by identifying each equivalence class with a canonical element of the class.

We show that $p \xmapsto{v}_{\mathsf{i}} q$ implies $f(p) \leq_{\mathsf{obs}} f(q)$. Since $p \subseteq q$, we deduce that $p \downharpoonleft \# \downharpoonleft \mathbf{M} \subseteq q \downharpoonleft \# \downharpoonleft \mathbf{M}$ and by monotonicity, $f(p) = \tau(p \downharpoonleft \# \downharpoonleft \mathbf{M}) \leq_{\mathsf{obs}} \tau(q \downharpoonleft \# \downharpoonleft \mathbf{M}) = f(q)$.

We show that $p \xmapsto{v}_{\mathsf{i}} q$ implies $\tau(v) \leq_{\mathsf{obs}} f(q)$. Suppose v only contains mutators. Since $v \subseteq q$, we deduce that $v \subseteq q \downharpoonleft \# \downharpoonleft \mathbf{M}$ and by monotonicity, $\tau(v) \leq_{\mathsf{obs}} \tau(q \downharpoonleft \# \downharpoonleft \mathbf{M}) = f(v)$. On the other hand, suppose v contains the non-mutator a. Let $A = \mathbf{M} \cup \{a\}$. Since $v \subseteq q$, we deduce that $v \downharpoonleft \mathbf{M} \subseteq q \downharpoonleft \# \downharpoonleft A$. By monotonicity, $\tau(v \downharpoonleft \mathbf{M}) \leq_{\mathsf{obs}} \tau(q \downharpoonleft A)$. Since $\tau(q \downharpoonleft A) = \tau(q \downharpoonleft \mathbf{M})$, we have $\tau(v \downharpoonleft \mathbf{M}) \leq_{\mathsf{obs}} \tau(q \downharpoonleft \mathbf{M}) = f(q)$, as required.

Thus $f(p) \xmapsto{v}_{\mathsf{s}} f(q)$, completing this direction of the proof.

For the reverse direction, we are given a refinement $f : \mathsf{cuts}(u) \to \Sigma/\!\sim$. For any $p \in \mathsf{cuts}_{\#}(u)$, define $\tau(p)$ to be a string in the equivalence class $f(p)$ that includes any non-mutator found in p.

We first prove that $\tau(p)$ is a linearization of p. A simple inductive proof demonstrates that for any $p \in \mathsf{cuts}_{\#}(u)$, there is a transition sequence of the form $\emptyset \longmapsto_{\mathsf{i}}^{*} \overset{p}{\longmapsto}_{\mathsf{i}} p$. Thus, we deduce from the label on the final transition into p that the $\tau(p)$ related to p is a linearization of p.

We now establish monotonicity. A simple inductive proof shows that for any $p, q \in \mathsf{cuts}(u)$, $p \subseteq q$ implies $p \longmapsto_{\mathsf{i}}^{*} q$. Thus $\tau(p) \leq_{\mathsf{obs}} \tau(q)$, by the properties of f and the definition of τ.

Composition. Given two *non-interacting* data structures whose replicated implementations satisfy their sequential specifications, the implementation that combines them satisfies the interleaving of their specifications. We formalize this as a composition theorem in the style of Herlihy and Wing [1990].

Given an execution u and $L \subseteq \mathbf{L}$, write $u \downharpoonright L$ for the execution that results by restricting u to events with labels in L: $u \downharpoonright L = u \downharpoonright \{e \in \mathsf{E}_u \mid \lambda_u(e) \in L\}$. This notation lifts to sets in the standard way: $U \downharpoonright L = \bigcup_{u \in U}\{u \downharpoonright L\}$. Write $u \vDash_{\mathsf{sec}} \Sigma$ to indicate that u is SEC for Σ.

Theorem 2 (Composition). *Let L_1 and L_2 be mutually independent subsets of $\mathbf{L}$. For $i \in \{1, 2\}$, let Σ_i be a specification with labels chosen from L_i, such that $\Sigma_1 \mathbin{|||} \Sigma_2$ is also a specification. If $(U \downharpoonright L_1) \vDash_{\mathsf{sec}} \Sigma_1$ and $(U \downharpoonright L_2) \vDash_{\mathsf{sec}} \Sigma_2$ then $U \vDash_{\mathsf{sec}} (\Sigma_1 \mathbin{|||} \Sigma_2)$ (equivalently $\mathsf{lts}(\Sigma_1 \mathbin{|||} \Sigma_2) \precsim \mathsf{lts}(\Sigma_1) \mathbin{|||} \mathsf{lts}(\Sigma_2)$).*

The proof is immediate. Since L_1 and L_2 are mutually independent, any interleaving of the labels will satisfy the definition.

Abstraction. We describe a process algebra with parallel composition and restriction and establish congruence results. We ignore syntactic details and work directly with LTSs. Replica identities do not play a role in the definition; thus, we permit implicit mobility of the client amongst replicas with the only constraint being that the replica has at least as much history on the current item of interaction as the client. This constraint is enforced by the synchronization of the labels, defined below. While the definition includes the case where the client itself is replicated, it does not provide for out-of-band interaction between the clients at different replicas: All interaction is assumed to happen through the data structure.

The relation $\|\!\lceil$ is defined between LTSs so that $P \|\!\lceil Q$ describes the system that results when client P interacts with data structure Q. For LTSs P and Q, define $\longmapsto_{\times}$ inductively, as follows, where $\emptyset$ represents the empty LVO.

$$\frac{q \overset{v}{\longmapsto}_Q q'}{\langle p, q\rangle \overset{v}{\longmapsto}_{\times} \langle p, q'\rangle} \qquad \frac{p \overset{v}{\longmapsto}_P p' \quad q \overset{w}{\longmapsto}_Q q'}{\langle p, q\rangle \overset{\emptyset}{\longmapsto}_{\times} \langle p', q'\rangle}\ \exists v' =_{\mathsf{iso}} v.\ v' \subseteq w \text{ and } \mathsf{max}(v') = \mathsf{max}(w)$$

Let $S_{\times} = \{\langle p, q\rangle \mid \exists \langle p', q'\rangle.\ \langle p, q\rangle \longmapsto_{\times}^{*} \langle p', q'\rangle \text{ and } \nexists v, p''.\ p' \overset{v}{\longmapsto}_P p''\}$

$$P \|\!\lceil Q = \begin{cases} \{\langle S_{\times}, \langle p_0, q_0\rangle, \longmapsto_{\times}\rangle\} & \text{if } S_{\times} \text{ is non-empty} \\ \emptyset & \text{otherwise} \end{cases}$$

The $\|\!\lceil$ operator is asymmetric between the client and data structure in two ways. First, note that every action of the client must be matched by the data structure.

The condition of client quiescence in the definition of $S_\times$, that all of the actions of the client P must be matched by Q; otherwise $P \parallel\!\lceil\, Q = \emptyset$. However, the first rule for $\longmapsto_\times$ explicitly permits actions of the data structure that may not be matched by the client. This asymmetry permits the composition of the data structure with multiple clients to be described incrementally, one client at a time. Thus, we expect that $(P_1 \mid P_2) \parallel\!\lceil\, Q \approx P_1 \parallel\!\lceil\, (P_2 \parallel\!\lceil\, Q)$.

Second, note that right rule for $\longmapsto_\times$ interaction permits the data structure Q to introduce order not found in the clients. This is clearly necessary to ensure that that the composition of client ✓0|+0 with the SET data structure is nonempty. In this case, the client has no order between +0 and ✓0 whereas the data structure orders ✓0 after +0. In this paper, we do not permit the client to introduce order that is not seen in the data structure. For a discussion of this issue, see [Jagadeesan and Riely 2015].

We can also define restriction for some set $A \subseteq \mathbf{L}$ of labels, a lá CCS. $P \backslash A = \langle S_P, p_0, \{\langle p, v, q\rangle \mid \langle p, v, q\rangle \in (\longmapsto_P) \text{ and } \mathsf{labels}(v) \cap A = \emptyset\}\rangle$. The definitions lift to sets: $\mathcal{P} \parallel\!\lceil\, \mathcal{Q} = \bigcup_{P\in\mathcal{P},\, Q\in\mathcal{Q}} P \parallel\!\lceil\, Q$ and $\mathcal{P}\backslash A = \{(P\backslash A) \mid P \in \mathcal{P}\}$.

Lemma 3. *If $\mathcal{P} \precsim \mathcal{P}'$ and $\mathcal{Q} \precsim \mathcal{Q}'$ then $\mathcal{P} \parallel\!\lceil\, \mathcal{Q} \precsim \mathcal{P}' \parallel\!\lceil\, \mathcal{Q}'$ and $\mathcal{P}\backslash A \precsim \mathcal{P}'\backslash A$.* □

It suffices to show that: $P \precsim \mathsf{lts}(u)$ implies $\mathcal{P} \parallel\!\lceil\, \mathsf{lts}(u) \precsim \mathcal{P} \parallel\!\lceil\, \mathsf{lts}(\Sigma)$. The proof proceeds in the traditional style of such proofs in process algebra. We illustrate by sketching the case for client parallel composition. Let f be the witness for $P \precsim \mathsf{lts}(u)$. The proof proceeds by constructing a "product" refinement $\mathcal{S}$ relation of the identity on the states of P with f, *i.e.*: $f(q) = q'$ implies $\langle p, q\rangle\ \mathcal{S}\ \langle p, q'\rangle$.

Thus, an SEC implementation can be replaced by the specification.

Theorem 4 (Abstraction). *If u is SEC for Σ, then $\mathcal{P} \parallel\!\lceil\, \mathsf{lts}(u) \precsim \mathcal{P} \parallel\!\lceil\, \mathsf{lts}(\Sigma)$.*

6 A Replicated Graph Algorithm

We describe a graph implemented with sets for vertices and edges, as specified by Shapiro et al. [2011a]. The graph maintains the invariant that the vertices of an edge are also part of the graph. Thus, an edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge. In the case of a concurrent addition of an edge with the deletion of either of its vertices, the deletion takes precedence.

The vertices $v, w, \ldots$ are drawn from some universe $\mathcal{U}$. An edge $e, e', \ldots$ is a pair of vertices. Let $\mathsf{vert}(e) = \{v, w\}$ be the vertices of edge $e = (v, w)$. The vocabulary of the set specification includes mutators for the addition and removal of vertices and edges and non-mutators for membership tests.

$$\begin{aligned}
\mathbf{M} &= \{\texttt{+}v, \texttt{-}v, \texttt{+(}v\texttt{,}w\texttt{)}, \texttt{-(}v\texttt{,}w\texttt{)} \mid v, w \in \mathcal{U}\}\\
\overline{\mathbf{M}} &= \{\checkmark v, \text{✗} v, \checkmark\texttt{(}v\texttt{,}w\texttt{)}, \text{✗}\texttt{(}v\texttt{,}w\texttt{)} \mid v, w \in \mathcal{U}\}\\
\# &= \{(e, v), (v, e) \mid v \in \mathsf{vert}(e)\} \cup \{(e, e') \mid \mathsf{vert}(e) \cap \mathsf{vert}(e') \neq \emptyset\}
\end{aligned}$$

Valid graph specification strings answer queries like sets. In addition, we require the following.

- Vertices and edges added at most once: Each add label is unique.
- Removal of a vertex or edge is preceded by a corresponding add.
- Vertices are added before they are mentioned in any edges: If $\sigma^j = \texttt{+(}v\texttt{,}w\texttt{)}$, or $\sigma^j = \texttt{-(}v\texttt{,}w\texttt{)}$ there exists $i, i' < j$ such that: $\sigma^i = \texttt{+}v$, $\sigma^{i'} = \texttt{+}w$.
- Vertices are removed only after they are mentioned in edges: If $\sigma^j = \texttt{+(}v\texttt{,}w\texttt{)}$, or $\sigma^j = \texttt{-(}v\texttt{,}w\texttt{)}$, then for all $i < j$: $\sigma^i \neq \texttt{-}v$ and $\sigma^i \neq \texttt{-}w$.

Graph Implementation. We rewrite the graph program of Shapiro et al. [2011a] in a more abstract form. Our distributed graph implementation is written as a client of two replicate set: for vertices (`V`) and for edges (`E`). The implementation uses USETs, which require that an element be added at most once and that each remove causally follow the corresponding add. Here we show the graph implementation for various methods as client code that runs at each replica. At each replica, the code accesses its local copy of the USETs. All the message passing needed to propagate the updates is handled by the USET implementations of the sets `V`, `E`. For several methods, we list preconditions, which prescribe the natural assumptions that need to satisfied when these client methods are invoked. For example, an edge operation requires the presence of the vertices at the current replica.

```
addVertex(v)               removeVertex(v)          bool ?(v)
  Pre: fresh(v)              Pre: V.?(v)              return V.?(v)
  V.add(v)                   V.remove(v)

addEdge(v,w)               removeEdge(v,w)          bool ?(v,w)
  Pre: V.?(v),V?(w)          Pre: V.?(v),V?(w)        if V.?(v)
  Pre: fresh((v,w))          Pre: E.?((v,w))          then return E.?((v,w))
  E.add((v,w))               E.remove((v,w))          else return false
```

We assume a causal transition system (as needed in Shapiro et al. [2011a]).

Correctness Using the Set Specification. We first show the correctness of the graph algorithm, using the SET specification for the vertex and edge sets. We then apply the abstraction and composition theorems to show the correctness of the algorithm using a set implementation.

Let u be a LVO generated in an execution of the graph implementation. The preconditions ensure that u has the following properties:

(a) For any v, $\texttt{+}v$ is never ordered after $\texttt{-}v$, and likewise for e.
(b) $\texttt{-(}v\texttt{,}w\texttt{)}$ or $\texttt{+(}v\texttt{,}w\texttt{)}$ is never ordered after $\texttt{-}v$ or $\texttt{-}w$.
(c) $\texttt{-(}v\texttt{,}w\texttt{)}$ or $\texttt{+(}v\texttt{,}w\texttt{)}$ is always ordered after some $\texttt{+}v$ and $\texttt{+}w$.

Define σ_1, σ_2 and σ_3 as follows.

- All elements of σ_1 are of the form $\texttt{+}v$. σ_1 exists by (c) above.
- All elements of σ_3 are of the form $\texttt{-}v$. σ_3 exists by (b) above.
- For each edge (v, w) that is accessed in u, let $\sigma_{(v,w)}$ be any interleaving of the events involving (v, w) in u such that no $\texttt{+(}v\texttt{,}w\texttt{)}$ occurs after any $\texttt{-(}v\texttt{,}w\texttt{)}$ in $\sigma_{(v,w)}$. $\sigma_{(v,w)}$ exists by (a) above. σ_2 is any interleaving of all the $s_{(v,w)}$.

Then u is SEC with witness $\sigma_u = \sigma_1\sigma_2\sigma_3$.

Full Correctness of the Implementation. We now turn to proving the correctness of the algorithm when the two sets are replaced by their implementations.

Consider two (distributed implementations of) separate and independent sets for vertices and edges, *i.e.* $\mathbf{L}_{\Sigma_1} \cap \mathbf{L}_{\Sigma_2} = \emptyset$. Suppose we have two implementations, each of which is correct individually: $\mathsf{lts}(U_i) \lesssim \mathsf{lts}(\Sigma_i)$. By composition, we have that they are correct when composed together: $U_1 \parallel\!\mid U_2 \lesssim \Sigma_1 \parallel\!\mid \Sigma_2$. Let $\mathcal{P}$ be the graph implementation, which is a client of the two sets. By abstraction, we know that $\mathcal{P} \parallel\!\lceil\, (\Sigma_1 \parallel\!\mid \Sigma_2) \lesssim T$ implies $\mathcal{P} \parallel\!\lceil\, (U_1 \parallel\!\mid U_2) \lesssim T$. By congruence, we deduce:

$$(\mathcal{P} \parallel\!\lceil\, (\Sigma_1 \parallel\!\mid \Sigma_2))\backslash(\mathbf{L}_{\Sigma_1} \cup \mathbf{L}_{\Sigma_2}) \lesssim T \text{ implies } (\mathcal{P} \parallel\!\lceil\, (U_1 \parallel\!\mid U_2))\backslash(\mathbf{L}_{\Sigma_1} \cup \mathbf{L}_{\Sigma_2}) \lesssim T.$$

Thus, in order to validate the full graph implementation, it is sufficient to establish the correctness of the graph client when interacting with the *specification* of the two independent SETs for edges and vertices, which we have already done in the previous treatment of abstract correctness.

7 Conclusions

We have provided a definition of *strong eventual consistency* that captures *validity* with respect to a *sequential specification.* Our definition reflects an attempt to resolve the tension between expressivity (cover the extant examples in the literature) and facilitating reasoning (by retaining a direct relationship with the sequential specification). The notion of *concurrent specification* developed by Burckhardt et al. [2014] has been used to prove the validity of several replicated data structure implementations. In future work, we would like to discover sufficient conditions relating concurrent and sequential specifications such that any implementation that is correct under the concurrent specification (as defined by Burckhardt et al.) will also be correct under the sequential counterpart (as defined here).

Acknowledgements. This paper has been greatly improved by the comments of the anonymous reviewers.

This material is based upon work supported by the National Science Foundation under Grant No. 1617175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 259–268, 2016.

A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In M. Aguilera, editor, *Distributed Computing*, volume 7611 of *Lecture Notes in Computer Science*, pages 441–442. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33650-8.

A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *POPL '14*, pages 285–296, 2014.

S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems (2Nd Ed.)*, pages 199–216. 1993.

S. Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014. ISSN 2325-1107.

S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In H. Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2012.

S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL '14*, pages 271–284, 2014.

K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995. ISBN 9810220588.

C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, June 1989.

H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Trans. Database Syst.*, 7(2):209–234, June 1982.

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

A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT POPL*, pages 371–384, 2016.

A. Haas, T. A. Henzinger, A. Holzer, C. M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith. Local linearizability. *CoRR*, abs/1502.07118, 2015.

R. C. Hansdah and L. M. Patnaik. Update serializability in locking. In *Proceedings on International Conference on Database Theory*, pages 171–185, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-17187-8.

M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *High Performance Computing - HiPC 2000, 7th International Conference, Proceedings*, pages 355–366, 2000.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, Dec 1972.

R. Jagadeesan and J. Riely. From sequential specifications to eventual consistency. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming*, volume 9135 of *Lecture Notes in Computer Science*, pages 247–259. Springer Berlin Heidelberg, 2015.

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

L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983.

N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214 – 233, 1995.

M. Perrin, A. Mostéfaoui, and C. Jard. Update consistency for wait-free concurrent objects. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 219–228, 2015.

G. Plotkin and V. Pratt. Teams can see pomsets. In *Workshop on Partial Order Methods in Verification*, DIMACS series Vol. 29, pages 117–128. AMS, 1997.

Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1): 42–81, Mar. 2005.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. TR 7506, Inria, 2011a.

M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, Proceedings*, pages 386–400, 2011b.

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

A. Tanenbaum and M. V. Steen. *Distributed systems.* Pearson Prentice Hall, 2007.

P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19, 2016.

W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, Jan. 2009.