From Linearizability to Eventual Consistency

Radha Jagadeesan and James Riely

DePaul University

Abstract We address the fundamental issue of interfaces that arises in the context of cloud computing; namely, what does it mean for a replicated and distributed implementation of a data structure to satisfy its standard sequential specification. The main contribution of this paper is a new definition of eventual consistency that liberalizes the linear time regime of linearizability to partial orders. Any implementation that conforms to our definitions satisfies the *Principle of Permutation Equivalence* enunciated in the literature : "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." Our definition also coincides with linearizability when the system is only accessed at a single replica, or when the system follows a single-master regime.
More generally, we establish the following key properties:
Expressiveness: We account for a wide range of extant replicated implementations of distributed data structures, including ORSET [Shapiro, Preguiça, Baquero, and Zawirski 2011] and Collaborative Text Editors [Attiya, Burckhardt, Gotsman, Morrison, Yang, and Zawirski 2016].
Composition: We show how to reason about composite data structures in terms of their components, in the style of Herlihy and Wing [1990]. This enables us to reason with a distributed implementation of a composite object (*e.g.*, a graph) by compositionally building on assumptions about the simpler distributed objects (*e.g.*, sets implementing vertices and sets implementing edges).
Abstraction: We prove an abstraction theorem in the style of Filipovic, O'Hearn, Rinetzky, and Yang [2010]. This demonstrates how a client's view of the distributed data structure can be simplified to reasoning with an automaton generated from the sequential specification.

1 Introduction

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 contains. 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 +k represents a call to add with argument k, -k represents remove(k), ✓k represents contains(k) returning true and ✗k represents contains(k) returning false.

Consider the implementation of such a SET by replication of the data structure. Requiring the replicas to achieve consensus on a global total order [Lamport 1978] on the operations on the data structure faces two impediments: (a) the associated serialization bottleneck negatively affects performance and scalability (eg, see [Ellis and Gibbs 1989]), and (b) the CAP theorem [Gilbert and Lynch 2002] imposes a tradeoff between consistency and partition-tolerance.

In alternative approaches based on *eventual consistency* and *optimistic replication* [Vogels 2009; Saito and Shapiro 2005], 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 (*e.g.*, see [Terry et al. 1995; Shapiro et al. 2011]).

A recent survey by Shapiro et al. [2011] on *convergent or commutative replicated datatypes* (CRDTs) provides a systematic attempt to design such data structures. The most expressive SET considered in this survey is the OR-set. Other examples that are addressed by such a paradigm include collaborative text editing [Attiya et al. 2016].

Consider the following diagram, in the style of this survey. This execution is initiated by a client of the form (`add(0); contains(1); ...`) || (`add(1); contains(0); ...`).

$$\begin{tikzcd}
{} \arrow[r] & +0 \arrow[r] \arrow[drrr] & \text{✗}1 \arrow[r] & \text{✓}0 \arrow[r] & \bullet \arrow[r] & \text{✓}1 \arrow[r] & \text{✓}0 \\
{} \arrow[r] & +1 \arrow[r] \arrow[urrr] & \text{✗}0 \arrow[r] & \text{✓}1 \arrow[r] & \bullet \arrow[r] & \text{✓}0 \arrow[r] & \text{✓}1
\end{tikzcd} \tag{1}$$

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.

The CRDT based analysis of SET proceeds by considering the commutativity properties of mutator operations. Mutator operations on different elements, *e.g.*, +0, +1 commute. For conflicting operations, +0, -0, assuming that we wish the outcome to be defined, there are two possible design choices:

– In OR-set, the add wins; thus, +0|-0 results in ✓0.
– The 2P-set, the remove wins; thus, +0|-0 results in ✗0.

For example, OR-set allows the following execution.

$$\begin{tikzcd}
{} \arrow[r] & +0 \arrow[rr] \arrow[ddrr] & & -0 \arrow[r] \arrow[ddrr] & \text{✗}0 \arrow[r] & \bullet \arrow[r] & \text{✓}0 \\
{} \arrow[r] & +0 \arrow[rrrrr] \arrow[urrrr] \arrow[dr] & & & & & {} \\
{} \arrow[r] & \text{✗}0 \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \text{✓}0 \arrow[r] & \bullet \arrow[r] & \text{✓}0
\end{tikzcd}$$

The choices made by OR-set are deceptively simple. They can result in complicated executions, such as the one below.

$$\begin{tikzcd}
{} \arrow[r] & +0 \arrow[r] \arrow[drr] & -0 \arrow[r] \arrow[drrr] & \bullet \arrow[r] & \text{✓}0 \arrow[r] & \bullet \arrow[r] & \text{✗}0 \\
{} \arrow[r] & +0 \arrow[r] \arrow[urr] & -0 \arrow[r] \arrow[urrr] & \bullet \arrow[r] & \text{✓}0 \arrow[r] & \bullet \arrow[r] & \text{✗}0
\end{tikzcd}$$

On the other hand, the OR-set does *not* permit the following behaviors.

$$\begin{tikzcd} {} \arrow[r] & +0 \arrow[r] & \bullet \arrow[r] & \checkmark 1 \arrow[r] & \text{✗}1 \arrow[r] & {} \\ {} \arrow[r] & +1 \arrow[ur] \arrow[rrrr] & & & & {} \end{tikzcd} \qquad \begin{tikzcd} {} \arrow[r] & \checkmark 0 \arrow[r] & +0 \end{tikzcd} \qquad (2)$$

This motivates the basic question: *In what sense does the* OR-*set implementation realize a* SET*?*

The traditional correctness idea is that of *eventual* consistency: when all messages are delivered, all the replicas agree on the outcome. This view is adequate for examples where we are interested only in the final state of the data structure. However, this standard definition of eventual consistency is quite weak since it ignores the intermediate states in the evolution of the system. Thus, eventual consistency does not rule out the problematic examples of (2). More generally, eventual consistency does not capture the following properties that simplify the client perspective by showing a degree of coherence with the sequential specification.

STS: The principle of *single threaded semantics*: A correct implementation should behave according to the sequential semantics if accessed at a single replica (inspired by [Haas et al. 2015]).

PPE: The principle of *permutation equivalence*: "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." Bieniusa et al. [2012] demonstrate that this principle holds for the OR-set and does *not* hold for the Amazon Dynamo shopping cart [DeCandia et al. 2007] and the C-Set [Aslan et al. 2011].

SINGLE-MASTER: The principle of *client-server linearizability:* Any execution of a correct implementation on a client-server system should be linearizable. [Budhiraja et al. 1993] identify executions where at any point there is a unique server that accepts updates in the form of mutators, and all other servers can only process non-mutators. In such a restricted distributed system, eventual consistency should imply linearizability.

It is noteworthy that OR-set satisfies the above properties; thus, eventual consistency provides a more complex picture for the client of an OR-set than is warranted.

There have been several attempts in the literature to formally characterize and analyze notions of eventual consistency. However, these attempts suffer from at least one of the following inadequacies:

– Inability to address the full expressiveness of CRDTs (*e.g.*, Burckhardt et al. [2012], Jagadeesan and Riely [2015] are unable to fully address OR-set), or
– Inability to validate STS,PPE, SINGLE-MASTER (*e.g.*, Burckhardt et al. [2014],[Bouajjani et al. 2014]).

This motivates a first statement of the problem addressed by this paper: *a definition of eventual consistency that is expressive enough to include all* CRDT*s and yet constrained enough to ensure that any valid implementation satisfies* STS, PPE *and* SINGLE-MASTER.

Of course, the particular choice of the criteria STS, PPE and SINGLE-MASTER is arguably ad-hoc. So, our investigations explore these (and other such criteria) as special cases of an abstraction theorem (in the sense of [Filipovic et al. 2010]) for clients of a data structure that is eventually consistent in our sense.

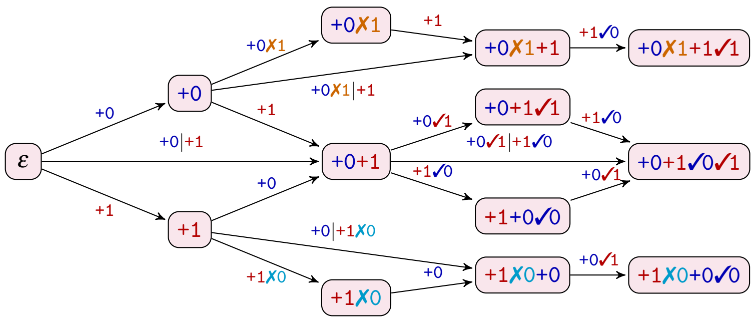
Abstraction theorems. We begin by briefly reviewing linearizability, the standard for correctness criterion of shared memory concurrent systems,

The execution of a concurrent system involves the evolution of multiple sequential threads in global linear time, perhaps communicating with each other via the shared memory. Given a specification Σ, an execution u is considered *linearizable* if it is a permutation of some $\sigma \in \Sigma$ that respects thread order and the order of non-overlapping method calls. An object is linearizable if all its executions are. From a client perspective, the set of linearizations of a linearizable object is an operational refinement of the object [Filipovic et al. 2010], *i.e.*, the client is able to soundly substitute the specification for the implementation. Thus, a client of a linearizable object can abstract away concurrency, take an atomic view of method invocations, and program against the sequential interface.

Let us take an alternative view of linearizability inspired by the proof techniques of [Filipovic et al. 2010]. For each specification Σ, we generate an automaton $\mathsf{lts}_{\mathsf{lin}}(\Sigma)$. Similarly, for each set U of executions, we generate a set of automata $\mathsf{lts}_{\mathsf{lin}}(U)$.

In a partial order notion of time, multiple events do not need to be related. So, the transitions in $\mathsf{lts}_{\mathsf{lin}}(\Sigma)$ are labelled by pomsets of events. The intuition is that the new events that are arriving simultaneously are the maximal elements of the pomset; the rest of the pomset merely elucidates the causal histories of the new arrivals.

Thus, each state of the automaton corresponds to a linearization of a *cut* of the distributed system. The labels of the automaton are partial orders whose maximal elements represent the events that are being executed; the remaining prefix represent the visibility relation at the point of execution. As an example of such an automaton, consider the binary SET.



The diagram above shows the portion of $\mathsf{lts}_{\mathsf{lin}}(\text{SET})$ corresponding to interaction the client (add(0); contains(1)) ∥ (add(1); contains(0)). To keep the diagram small, we quotient by bisimilarity; the state labels in the diagram are chosen from among the bisimilar states.

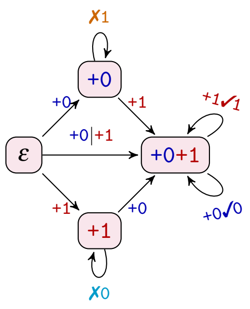
This perspective allows us to rephrase linearizability as a simulation: an object system U is *linearizable* if and only if $\forall u \in U.\ \mathsf{lts}_{\mathsf{lin}}(u) \sqsubseteq\!\!\!\!\sim \mathsf{lts}_{\mathsf{lin}}(\Sigma)$, *i.e.*, $\mathsf{lts}(u)$ is simulated by $\mathsf{lts}_{\mathsf{lin}}(\Sigma)$. The abstraction theorem of [Filipovic et al. 2010] then is seen as a precongruence property of a suitable parallel composition; *i.e.*, for every client C: $\mathsf{lts}(U) \precsim \mathsf{lts}_{\mathsf{lin}}(\Sigma)$ implies $C \parallel\!\lceil\, \mathsf{lts}(U) \precsim C \parallel\!\lceil\, \mathsf{lts}_{\mathsf{lin}}(\Sigma)$.

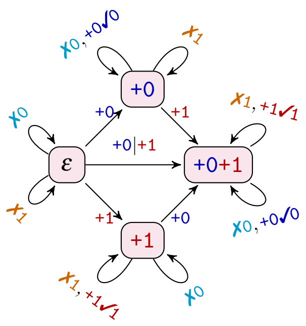
For example, the execution given in (1) is not simulated by the example specification automaton, and therefore is not linearizable.

The peculiarities of linearizability are made evident by this formulation. In a linearizable execution, nothing is ever forgotten: a state of the automaton fully reflects all of the order of all operations that lead to the state. In addition, the source of every edge is a prefix of the target.

Moving onto eventual consistency. Eventual consistency is strictly weaker than linearizability. Thus, when adapting this framework to EC, we expect the label set to be smaller. As a result of shrinking the label set, the number of distinct states in the bisimulation quotient is also smaller.

Rather than recording the entire downclosure in a label, the EC automaton records only the *dependent* downclosure. The dependent downclosure includes only the mutators that precede and are dependent on the maximal events. When applied to the previous example, we arrive at the automaton on the left below. Unlike the automaton for linearizability, this automaton will simulate the execution in (1).



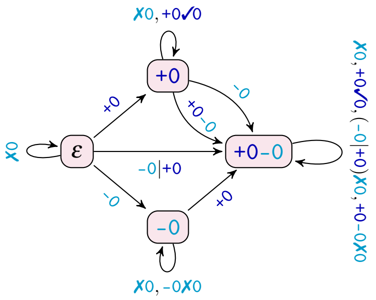


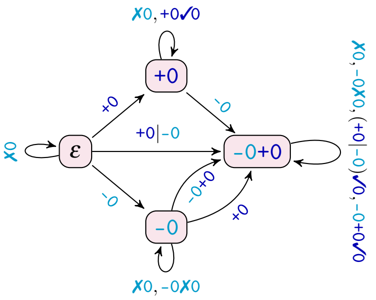
Because the bisimulation collapse of the EC automaton is much smaller than that for linearizability, we can show the individual accessors available in each state without increasing the number of states. We show the enriched automaton on the right above. As expected, the cut after +0 permits the accessor +0✓0. Perhaps surprisingly, it also permits ✗0. The difference between these is manifest in the prefix: ✓0 is possible after +0 has been seen, whereas ✗0 is possible when +0 has not been seen.

In SET, the operators on different values are symmetric, leading to a great deal of symmetry in the preceding figures. The dependencies between operators on the same values lead to asymmetries.

In a distributed system, a newly arriving event is not necessarily an extension of the future: it could also be the receipt of an update generated in the concurrent past, but only received now. Consider the automata for +0-0 and -0+0 given at the top of the next page. Depending on the order chosen, the paths to the final state differ.

Let us focus on the automaton for +0-0, on the left. The transition $+0 \xmapsto{+0-0} +0-0$ corresponds to an execution of -0 *after* +0. The transition $+0 \xmapsto{-0} +0-0$ corresponds to an execution of -0 *concurrently with* +0. While there is also a transition $-0 \xmapsto{+0} +0-0$,





there is no transition of the form -0 $\xrightarrow{\texttt{-0+0}}$ +0-0 since the order on the label would contradict the order at the target. Note also that whereas the source of +0 $\xrightarrow{\texttt{-0}}$ +0-0 is a prefix of the target, this does not hold for -0 $\xrightarrow{\texttt{+0}}$ +0-0. In the latter case, the source is a *subsequence* of the target, rather than a prefix. We will see in section 4 that even using the standard notion of subsequence here is too strong: we must consider subsequences up to stuttering equivalence.

This perspective allows us to characterize eventual consistency as a simulation. As for linearizability, the abstraction theorem provides a simplified programming model for clients: $U \precsim \mathsf{lts}_{\mathsf{ec}}(\Sigma)$ implies $C \,\|\!\lceil\, U \precsim C \,\|\!\lceil\, \mathsf{lts}_{\mathsf{ec}}(\Sigma)$. The properties STS, PPE, SINGLE-MASTER can be viewed as simple corollaries of this abstraction theorem.

The programming model is completely abstract with respect to replica identity. A client of an object that is eventually consistent in this sense is able to abstract away from the mechanics of replication and distribution and take an atomic view of method invocations. However, in contrast to [Filipovic et al. 2010], such general clients must account for the partial order aspects incorporated into $\mathsf{lts}_{\mathsf{ec}}(\Sigma)$.

As a consequence of abstraction, the following implementation strategy is sound: Each client is attached to a replica which fulfills its requests. The service may move clients, *e.g.*, based on network and load-balancing considerations. During execution, the client can be moved to any other replica, with the proviso that the target replica has received all of the update messages that have been received by the source replica.

Prior work. We refer the reader to a recent survey paper by Viotti and Vukolic [2016] for a taxonomy of the various possibilities for consistency in non-transactional distributed systems using the vocabulary of [Burckhardt et al. 2014].

The basic problem of relating replicated datatypes to their sequential specifications is addressed by the seminal paper of Burckhardt et al. [2012]. Intuitively, Burckhardt et al. [2012] define a notion of eventual consistency (EC) for transactions as compatibility with a serialization of them. Our prior paper [Jagadeesan and Riely 2015] builds on [Burckhardt et al. 2012] to provide a weaker definition of EC. Neither definition, however, is able to validate every implementation deemed correct by Shapiro et al. [2011]. In particular, they fail for data structures whose mutators do not fully commute, such as non-monotone sets. For example, Shapiro et al.'s OR-set (observed remove SET), that is not considered EC by either [Burckhardt et al. 2012] or [Jagadeesan and Riely 2015].

Burckhardt et al. [2014] take an alternate approach: to view the interface of a replicated data structure as a *concurrent* specification. In this approach, the valid result of an accessor is determined from the context of a prior concurrent history. [Bouajjani et al. 2014] 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. This approach has the advantage of flexibility. It is possible to validate the structures in [Shapiro et al. 2011] as well as the Amazon Dynamo shopping cart [DeCandia et al. 2007].

However, this gain in flexibility comes at the cost of a clear connection to any sequential specification. In particular, the principles of PPE, STS and SINGLE-MASTER are not necessarily validated in this approach. For example, Bieniusa et al. [2012] demonstrate that this principle holds for the OR-set and does *not* hold for the Amazon Dynamo shopping cart [DeCandia et al. 2007] and the C-Set [Aslan et al. 2011].

A key technical influence on our work comes from the study of relaxed memory. We are inspired in particular by the RMO models that cannot be implemented with buffers and a central store [Higham and Kawash 2000]; thus, in our opinion, capturing one essential artifact of peer-to-peer distributed systems that do not have a master replica. Our particular technical treatment is reminiscent of the approach of Alglave [2012].

While we show that our definitions are applicable to a variety of CRDTs, in this paper, we do not explore systematic proof principles for validating that data structures. The proof rules, in particular the event based proof rule, explored in Gotsman et al. [2016] are highly relevant here.

Organization of paper The rest of the paper is organized as follows.

- In section 2, we define alphabets and specifications over those alphabets. Our alphabets come equipped with a dependency relation. We define equivalence up to stuttering and use this to define a liberalization of traditional subsequence order.
- In section 3, we describe our model of execution for a replicated and distributed data structure. Our model is flexible enough to accommodate non-causal (intransitive) systems. We define the notion of *dependent cut*, which is used throughout the remainder of the paper.
- In section 4, we define our notion of EC and discuss the definition with several examples that illustrate the design decisions, including a collaborative text editor.
- In section 5, we prove that OR-set satisfies our definition.
- In section 6, we provide an alternative characterization of EC as a simulation. Theorem 21 shows that the simulation characterization is sound and complete for EC.
- In section 7, we explore how a client can make use of the fact that it is running against an data structure that is EC. We prove abstraction (Theorem 25) and composition (Proposition 26) and show the use of these results with the example of a graph implemented using two concurrent sets.

2 Specifications

We define alphabets and specifications over those alphabets. We then provide example specifications and define *dependency* between actions, relative to a specification. The central contribution in this section is the notion of stuttering equivalence of strings in a specification that is used to describe an order between strings of a specification that liberalizes the subsequence order.

2.1 Alphabets

An *alphabet* is a quadruple $\langle \mathbf{L}, \mathbf{M}, \mathscr{A}, \# \rangle$ where

- $\mathbf{L}$ is a set of *actions* (also known as *labels*),
- $\mathbf{M} \subseteq \mathbf{L}$ is a distinguished set of *mutator* actions,
- $\mathscr{A} \subseteq 2^{\mathbf{L}}$ is a partitioning[1] of $\mathbf{L}$ into *accessor* sets, and
- $\# \subseteq (\mathbf{L} \times \mathbf{L})$ is a symmetric and reflexive *dependency* relation.

We write $\overline{\mathbf{M}}$ for the set of non-mutators; that is, $\overline{\mathbf{M}} = \mathbf{L} \setminus \mathbf{M}$.

For example, a binary SET uses the following alphabet:

- $\mathbf{M} = \{\texttt{+0}, \texttt{-0}, \texttt{+1}, \texttt{-1}\}$, representing addition and removal of elements 0 and 1,
- $\overline{\mathbf{M}} = \{\text{✗}\texttt{0}, \text{✓}\texttt{0}, \text{✗}\texttt{1}, \text{✓}\texttt{1}\}$, representing membership tests returning false or true,
- $\mathbf{L} = \mathbf{M} \cup \overline{\mathbf{M}}$,
- $\mathscr{A} = \{\{\texttt{+0}\}, \{\texttt{-0}\}, \{\text{✗}\texttt{0}, \text{✓}\texttt{0}\}, \{\texttt{+1}\}, \{\texttt{-1}\}, \{\text{✗}\texttt{1}, \text{✓}\texttt{1}\}\}$, and
- $\# = \{\texttt{+0}, \texttt{-0}, \text{✗}\texttt{0}, \text{✓}\texttt{0}\}^2 \cup \{\texttt{+1}, \texttt{-1}, \text{✗}\texttt{1}, \text{✓}\texttt{1}\}^2$, where $D^2 = D \times D$.

2.2 Specifications

Fix an alphabet $\langle \mathbf{L}, \mathbf{M}, \mathscr{A}, \# \rangle$.

Let σ and τ range over *strings* in $\mathbf{L}^*$ and Σ and T range over *specifications*, which are sets of strings subject to certain closure properties, listed below. We use standard notation for strings and specifications, including the empty string (ε), concatenation (ΣT), Kleene star (Σ^*), choice ($\Sigma \mid T$) and interleaving ($\Sigma \mathbin{|||} T$).

Let $\sigma \downharpoonleft \#a$ be the subsequence of σ that includes exactly the actions dependent on a: $(b_1 \cdots b_n) \downharpoonleft \#a = b_{k_1} \cdots b_{k_m}$ where $k_1, \ldots, k_m$ is the increasing sequence drawn from $\{k \in [1,n] \mid b_k \# a\}$.

Definition 1. A set $\Sigma \subseteq \mathbf{L}^*$ is a *specification* if it satisfies the following.

- prefix closed:
  $\forall \sigma, \tau \in \Sigma.\ \sigma\tau \in \Sigma$ implies $\sigma \in \Sigma$
- non-mutators are closed under stuttering, mumbling and commutation:
  $\forall \sigma, \tau \in \Sigma.\ \forall a \in \overline{\mathbf{M}}.\ \sigma a \tau \in \Sigma$ implies $\sigma a^* \tau \subseteq \Sigma$
  $\forall \sigma \in \Sigma.\ \forall a, b \in \overline{\mathbf{M}}.\ \{\sigma a, \sigma b\} \subseteq \Sigma$ implies $\{\sigma ab, \sigma ba\} \subseteq \Sigma$
- independent labels can be removed:
  $\forall \sigma \in \Sigma.\ \forall a \in \overline{\mathbf{M}}.\ \{\sigma, \sigma \downharpoonleft \#a\} \subseteq \Sigma$ implies $\sigma a \in \Sigma \Leftrightarrow (\sigma \downharpoonleft \#a) a \in \Sigma$
- accessor partitions are deterministic:
  $\forall \sigma \in \Sigma.\ \forall A \in \mathscr{A}.\ \forall a, b \in A.\ \sigma a \in \Sigma$ and $\sigma b \in \Sigma$ imply $a = b$
- accessor partitions are input-enabled:
  $\forall \sigma \in \Sigma.\ \forall A \in \mathscr{A}.\ \exists a \in A.\ \sigma a \in \Sigma$ □

[1] Ie, $\mathbf{L} = \bigcup_{A \in \mathscr{A}} A$ and $\forall A, A' \in \mathscr{A}.\ A \cap A' = \emptyset$.

Independence of two actions requires that there be no single operation that can affect both. Thus, all actions of SET become dependent if we include an single operation that tests whether a pair of elements is present in the set.

The SET specification illustrates an important special case of the above definition when # is also transitive, *i.e.*, it is an equivalence relation. In this special case, the specification is a shuffle of essentially disjoint specifications. Concretely, in the case of set, let $[\![r]\!]$ denote the prefix closure of the set of strings satisfying regular expression r. Valid set strings with values in I are the defined: $|||_{i\in I}\ [\![✗i^{*}\big((+i\ ✓i^{*})\,|\,(-i\ ✗i^{*})\big)^{*}]\!]$.

As a negative example, note that $[\![a^{*}\,|\,b^{*}]\!]$ is not a specification. Commutation of non-mutators requires that at least one of a or b is a mutator, since $\{\varepsilon a, \varepsilon b\} \subseteq [\![a^{*}\mid b^{*}]\!]$ but $\varepsilon ab \notin [\![a^{*}\mid b^{*}]\!]$. If both a and b are mutators, the specification cannot be input enabled: if the actions are assigned the same accessor partition, then determinism fails at prefix ε; if they are assigned separate accessor partitions, then input enabledness fails at every prefix.

Notation 2. Each specification Σ is associated with an alphabet specification. When necessary for clarity, we write the alphabet as $\langle \mathbf{L}_\Sigma, \mathbf{M}_\Sigma, \mathscr{A}_\Sigma, \#_\Sigma\rangle$. To keep the notation light, we drop the subscript when possible. The associated alphabet should be clear from context. □

2.3 Stuttering equivalence

Fix a specification Σ. Let *state equivalence*, $\approx\ \subseteq \mathbf{L}^{*}\times\mathbf{L}^{*}$, be defined as follows.

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

The definition equates specification strings that permit the same suffixes. For non-specification strings, the definition allows $\sigma \approx \sigma'$ only when $\sigma = \sigma'$. For binary sets, we have $\varepsilon \approx$ +0-0 and $\varepsilon \approx$ ✗0 $\approx$ ✗1 $\approx$ ✗0✗1 but $\varepsilon \not\approx$ ✓0.

Let *stuttering equivalence*, $\sim\ \subseteq \mathbf{L}^{*}\times\mathbf{L}^{*}$, be the least equivalence relation generated by the following rules, where a ranges over $\mathbf{L}$.

$$\frac{}{\varepsilon \sim \varepsilon} \qquad\qquad \frac{\sigma \sim \sigma'}{\sigma a \sim \sigma' a} \qquad\qquad \frac{\sigma \sim \sigma' \quad \sigma' \approx \sigma' a}{\sigma \sim \sigma' a}$$

Our definition of stuttering equivalence adapts Brookes [1996] to sequences of labels.

Stuttering equivalence inherits some of the relaxation of state equivalence. Thus, for binary sets, we have $\varepsilon \sim$ ✗0 $\sim$ ✗1 $\sim$ ✗0✗1, +0+0 $\sim$ +0 and $\varepsilon \not\sim$ ✓0.

Stuttering equivalence additionally demands that intermediate states match. Thus $\varepsilon \not\sim$ +0-0.

For non-specification strings, the definition allows stuttering only in the prefix that *is* a specification string. Thus +0 $\sim$ ✗0+0+0 but +0 $\not\sim$ ✓0+0+0.

In combination with the closure of specifications under stuttering of mutators, we can deduce: $\forall a \in \overline{\mathbf{M}}.\ \forall \sigma a \tau \in \Sigma.\ \forall \rho \in [\![\sigma a^{*} \tau]\!].\ \rho \in \Sigma$ and $\rho \sim \sigma\tau$.

2.4 Notions of consistency between strings

The basic combinator for strings is concatenation, from which several relations are derived, including prefix, substring, and subsequence. For example, *abc* has one prefix of length two (*ab*), two substrings of length two (*ab* and *bc*), and three subsequences of length two (*ab*, *ac* and *bc*).

The definitions are as follows. On the left, we give the *strict* (standard) versions of these definitions. On the right, we give the definitions up to stuttering.

$$\begin{array}{rl}
\sigma_1 \leq_{\mathsf{pre}} \sigma_1\tau_1 & \sigma \lesssim_{\mathsf{pre}} \tau \text{ if } \exists\sigma' \sim \sigma.\, \exists\tau' \sim \tau.\, \sigma' \leq_{\mathsf{pre}} \tau' \\
\sigma_1 \leq_{\mathsf{str}} \tau_0\sigma_1\tau_1 & \sigma \lesssim_{\mathsf{str}} \tau \text{ if } \exists\sigma' \sim \sigma.\, \exists\tau' \sim \tau.\, \sigma' \leq_{\mathsf{str}} \tau' \\
\sigma_1 \cdots \sigma_n \leq_{\mathsf{seq}} \tau_0\sigma_1\tau_1 \cdots \sigma_n\tau_n & \sigma \lesssim_{\mathsf{seq}} \tau \text{ if } \exists\sigma' \sim \sigma.\, \exists\tau' \sim \tau.\, \sigma' \leq_{\mathsf{seq}} \tau'
\end{array}$$

The strict relations on the left can be understood in isolation, whereas the non-strict relations on the right can only be understood with respect to a given specification. For example, on binary sets we have `+0+0` $\lesssim_{\mathsf{pre}}$ `+0+1` although `+0+0` $\not\leq_{\mathsf{pre}}$ `+0+1`.

The strict relations are partial orders, but the non-strict relations are only preorders. For example, on sets we have `+0+0` $\lesssim_{\mathsf{pre}}$ `+0` $\lesssim_{\mathsf{pre}}$ `+0+0`. We can recover a partial order by considering the relation over equivalence classes up to $\sim$.

For our purposes, the most important of these relations are the strongest and the weakest. Let $\mathsf{prefix}(\tau) = \{\sigma \mid \sigma \leq_{\mathsf{pre}} \tau\}$ and $\mathsf{subseq}(\tau) = \{\sigma \mid \sigma \lesssim_{\mathsf{seq}} \tau\}$. The partial order $\langle \mathsf{subseq}(\tau), \lesssim_{\mathsf{seq}}\rangle$ satisfies the “M property” of Gunter [1987]: every finite $T \subseteq \mathsf{subseq}(\tau)$ has a finite and complete set of minimal upper bounds in $\langle \mathsf{subseq}(\tau), \lesssim_{\mathsf{seq}}\rangle$.

In general, specifications choose a subset of subsequences, and thus upper bounds may not exists. However, if an upper bound for $T \subseteq \Sigma$ *does* exists in $\langle \Sigma, \lesssim_{\mathsf{seq}}\rangle$, then T has a finite and complete set of minimal upper bounds in $\langle \Sigma, \lesssim_{\mathsf{seq}}\rangle$.

In the case of sets over a single element 0, the canonical representatives of the specification strings of mutators are of the form $(\texttt{+0-0})^*$, since stuttering equivalence allows us to remove duplicate `+0` without an intervening `-0`, and symmetrically `-0` without intervening `+0`.

3 Implementations

We model implementations abstractly as sets of LVOs, which we also call *traces* and define below.

A labelled partial order (LPO) is a labelled relation that is reflexive, antisymmetric, and transitive. We define a labelled visibility order (LVO) to be a potentially intransitive generalization of an LPO, with two labelling functions.

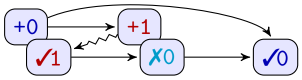
Transitivity can be seen as a *causality* requirement. Many CRDTs satisfy this causality requirement, including state-based CRDTs. Other CRDTs assume a causal delivery model for messages. In particular, all of the examples from Shapiro et al. [2011] either impose transitivity or assume it. However, the framework of CRDTs is more general; we adopt LVOs to capture this generality.

Definition 3. A quadruple $u = \langle E, \lambda, \rho, \rightsquigarrow\rangle$ is *labeled visibility order* (LVO), if E is a finite set of events, $\lambda \in (E \mapsto \mathbf{L})$, $\rho \in (E \mapsto \mathbf{R})$, and $\rightsquigarrow \subseteq (E \times E)$ is reflexive, acyclic and per-replica total: if $\rho(d) = \rho(e)$ then either $d \rightsquigarrow e$ or $e \rightsquigarrow d$ □

The labels of an LVO have two components, which we divide into two labeling functions: The action labelling $\lambda \in (E \mapsto \mathbf{L})$ maps events to actions. The replica labelling $\rho \in (E \mapsto \mathbf{R})$ maps events to *replica identifiers*, where $\mathbf{R}$ is a set of replica identifiers.

Like an LPO, the most important component of an LVO is a relation $\rightsquigarrow \subseteq (E \times E)$ defined over a carrier set E. Unlike LPOs, LVOs do not require that $\rightsquigarrow$ be transitive and antisymmetric, but merely acyclic. Acyclicity ensures that the transitive closure of an LVOs is an LPO. We do require per-replica transitivity, to model local computation.

When drawing traces, we typically elide event and replica identifiers. We use straight lines for “transitive” edges, with the intuitive reading that “this and all preceding actions are delivered”. We reserve the use of the zigzag arrow to intransitive communications, such as the following. Here `+1` is received before `+0`, even though `+0` precedes `+1`.



Two LPOs are *isomorphic* if they differ only in the carrier set. For LVOs, we additionally ignore the replica identifier.

Definition 4. LVOs u and v are (replica insensitive) *isomorphic* (notation $u =_{\mathsf{iso}} v$) if there exists a bijection $\alpha : \mathsf{E}_u \to \mathsf{E}_v$ such that $\lambda_u(d) = \lambda_v(\alpha(d))$ and $d \rightsquigarrow_u e$ precisely when $\alpha(d) \rightsquigarrow_v \alpha(e)$. □

Many concepts defined for LPOs extend smoothly to LVOs. For example, restriction, downclosure, cut (or prefix) and suborder can be defined as follows.

Definition 5. When $D \subseteq \mathsf{E}_u$, write $u \downharpoonleft D$ for the LVO derived by restricting to the events in D. That is $u \downharpoonleft D = \langle D, \lambda_u \downharpoonleft D, \rho_u \downharpoonleft D, \rightsquigarrow_u \downharpoonleft D \rangle$, where restriction on functions and relations is standard: Given a function $f : E \to X$ and $D \subseteq E$, define $f \downharpoonleft D = \{\langle d, f(d)\rangle \mid d \in D\}$. Given a relation $\mathscr{R}$: $E \to E$ and $D \subseteq E$, define $\mathscr{R} \downharpoonleft D = \{\langle d_1, d_2\rangle \mid d_1, d_2 \in D \text{ and } d_1 \mathrel{\mathscr{R}} d_2\}$.

Event set $D \subseteq \mathsf{E}_u$ is *downclosed* if $\forall e \in D.\ \forall d \in \mathsf{E}_u.\ d \in D$ whenever $d \rightsquigarrow_u e$ and e is maximal in D. Let $\mathsf{cuts}(v) = \{u \mid \exists D \subseteq \mathsf{E}_v.\ D$ is downclosed and $u = v \downharpoonleft D\}$.

Trace u is an *suborder* of v (notation $u \subseteq v$) if $\mathsf{E}_u \subseteq \mathsf{E}_v$, $\lambda_u \subseteq \lambda_v$, $\rho_u \subseteq \rho_v$, and $(\rightsquigarrow_v) \subseteq (\rightsquigarrow_u)$. □

Note that the definition of downclosed (and therefore prefix) is unusual. In particular, note that $\{d, e\}$ is a downclosed subset of $c \rightsquigarrow d \rightsquigarrow e$ when $\neg(c \rightsquigarrow e)$. On LPOs the definition degenerates to the usual one.

Notation 6. Let $\mathscr{L}$ be the set of all LVOs.

We write the components of an LVO u as $\langle \mathsf{E}_u, \lambda_u, \rho_u, \rightsquigarrow_u \rangle$.

Each trace u is associated with a specification. When necessary for clarity, we write Σ_u for the specification associated with u, likewise the alphabet $\langle \mathbf{L}_{\Sigma_u}, \mathbf{M}_{\Sigma_u}, \mathscr{A}_{\Sigma_u}, \#_{\Sigma_u} \rangle$. To keep the notation light, we drop the subscript when possible. The associated specification and alphabet should be clear from context. □

3.1 Dependent cuts

A key technical tool in our approach is the liberalization of the order of an LVO by removing order from non-mutators and between independent labels. This treatment is reminiscent of the approach of Alglave [2012]. In this subsection, we develop this infrastructure.

When considering the correctness of an LVO u, there are two ways that dependence affects the definitions. First we restrict u to the suborder $u \lfloor \#$, which includes only the visibility that must be maintained when checking correctness. Second, we restrict attention to the cuts of $u \lfloor \#$ that contain only independent non-mutators. We call these *dependent cuts*. We define $\mathsf{cuts}_{\#}(u)$ to be the dependent cuts of u and $\mathscr{L}_{\#}$ to be the set of all possible dependent cuts.

We write $d \overset{\#}{\rightsquigarrow}_u e$ when d is a mutator that is both dependent on and visible to e.

$$(d \overset{\#}{\rightsquigarrow}_u e) \text{ iff } (d = e) \text{ or } (\lambda_u(d) \in \mathbf{M},\, \lambda_u(d) \mathrel{\#} \lambda_u(e) \text{ and } d \rightsquigarrow_u e)$$

Let $u \lfloor \# = \langle \mathsf{E}_u, \lambda_u, \rho_u, \overset{\#}{\rightsquigarrow}_u \rangle$ be the *dependent restriction* of u. Note that $u \lfloor \# \subseteq u$. We define $\overset{\#}{\rightsquigarrow}_u$ so that it is reflexive; thus $u \lfloor \#$ is an LVO whenever u is. Except for reflexive edges, non-mutators may only appear on the right of $\overset{\#}{\rightsquigarrow}_u$.

For $p \in \mathscr{L}$, let $\mathsf{E}_{\overline{\mathsf{M}}}(p) = \{e \in \mathsf{E}_p \mid \lambda_p(e) \in \overline{\mathbf{M}}\}$ be the non-mutator events of p. Let

$$\mathsf{cuts}_{\#}(u) = \{p \in \mathsf{cuts}(u \lfloor \#) \mid \forall d, e \in \mathsf{E}_{\overline{\mathsf{M}}}(p).\ \neg(\lambda_p(d) \mathrel{\#} \lambda_p(e))\}$$

be the *dependent cuts* of u. Thus a dependent cut may not contain two dependent non-mutators, such as two occurrences of $\mathsf{✗0}$ in SET.

Define $\mathscr{L}_{\#} = \bigcup_{u \in \mathscr{L}} \mathsf{cuts}_{\#}(u)$ to be the set of all possible dependent cuts.

Note that $\mathscr{L}_{\#} \subseteq \mathscr{L}$.

3.2 Linearization

Partial orders and interval orders are subclasses of LVOs. For example, since the transitive closure of an acyclic relation is antisymmetric, the transitive closure of an LVO is an LPO. A trace u is a *labelled partial order* (LPO) if $\rightsquigarrow_u$ is transitive. An LPO u is an *labelled interval order* (LIO) if $\forall d, e, d', e' \in \mathsf{E}_u.\ (d \rightsquigarrow_u e \text{ and } d' \rightsquigarrow_u e')$ imply $(d \rightsquigarrow_u e'$ or $d' \rightsquigarrow_u e)$. An LPO u is an *labelled total order* (LTO) if $\forall d, e \in \mathsf{E}_u.\ d \rightsquigarrow_u e$ or $e \rightsquigarrow_u d$.

The definitions are related by inclusion: LTO $\subset$ LIO $\subset$ LPO $\subset$ LVO. These restrictions correspond different views of the nature of events over time and space. An LTO captures the idea that events have no duration and time is global. LIOs give events duration, but retain global time — we expand on this in the next subsection. LPOs capture distributed systems without the requirement of globally agreed time. The transitivity requirement of an LPO can be seen as a restriction on communication in such systems: events must be communicated in the order they are seen. LVOs generalize LPOs by allowing out-of-order communication.

Definition 7. For $a_i \in \mathbf{L}$, we say that $a_1 \ldots a_n$ is a *linearization* of $E \subseteq \mathsf{E}_u$ if there exists a bijection $\alpha : E \to [1, n]$ such that $\forall e \in E.\ \lambda_u(e) = a_{\alpha(e)}$ and $\forall d, e \in E.\ d \rightsquigarrow_u e$ implies $\alpha(d) \leq \alpha(e)$. $\square$

Linearization gives rise to a natural definition of *linearizability*, which generalizes the original definition [Herlihy and Wing 1990]. There, linearizability is defined in terms of complete histories, H, which are strings of matching invocations and responses. The order $<_H$ is defined as follows: $e_0 <_H e_1$ if $\mathsf{response}(e_0)$ precedes $\mathsf{invocation}(e_1)$ in

H. It is straightforward to establish that $<_H$ is an interval order. The following definition generalizes linearizability from interval orders to LVOs.

Definition 8. Trace u is *linearizable* if there exists $\sigma \in \Sigma_u$ that linearizes E_u. □

When restricted to interval orders, this notion coincides with that of Herlihy and Wing.

4 Eventual Consistency

The definition of eventual consistency follows from two simple principles.

Linearization: Each "cut" of events should linearize to a specification string.
Monotonicity: The strings chosen for a "future" cut should be "consistent" with the string chosen for the current cut.

Definition 9. Trace u is *eventually consistent* if there exists a function $\tau : \mathsf{cuts}_{\#}(u) \to \Sigma$ such that:

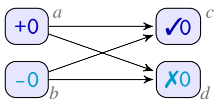
- $\forall p \in \mathsf{cuts}_{\#}(u).\ p$ linearizes to $\tau(p)$, and
- $\forall p, q \in \mathsf{cuts}_{\#}(u).\ p \subseteq q$ implies $\tau(p) \lesssim_{\mathsf{seq}} \tau(q)$. □

Because the range of τ is Σ, the first condition ensures that every event set in $\mathsf{cuts}_{\#}(u)$ linearizes to some specification string. The second condition ensures monotonicity[2].

We consider a series of examples to explain the various choices in this definition. Most of these use the SET specification. We also consider the collaborative text editing protocol of [Attiya et al. 2016]. Many positive examples use OR-set executions, and some of the negative examples use variations of SET.

- Example 10 shows that EC requires agreement on the order of mutators.
- Example 11 shows that you must consider only one conflicting accessor at a time. This example also demonstrates that EC is weaker than linearizability.
- Example 12 establishes that we must allow different events to match the same action in the global order σ. It is important to match actions, rather than events.
- Example 13 establishes that we cannot strengthen the definition to consider prefixes instead of subsequences.
- Example 14 shows that we must consider all dependent cuts, including the dependent downclosure of single events.
- Example 15 establishes that the use of independency is necessary.
- Example 16 establishes that the use of stuttering is necessary.
- Example 17 gives an example that requires an intransitive dependency relation.

Example 10. The following execution is not EC for SET.



[2] The corresponding definition for linearizability uses cuts rather than $\mathsf{cuts}_{\#}$ and $\leq_{\mathsf{pre}}$ rather than $\lesssim_{\mathsf{seq}}$.

The cuts $\{+0^a, -0^b, ✓0^c\}$ and $\{+0^a, -0^b, ✗0^d\}$ are both supersets of $\{+0^a, -0^b\}$, and therefore they must agree on the order of these events. There is no SET trace that gives both ✗0 and ✓0 in the same state. □

Example 11. The following execution is EC for SET, but is not linearizable.

$$\begin{tikzcd}
+0^a \arrow[r] & ✗1^c \\
+1_b \arrow[r] & ✗0_d
\end{tikzcd}$$

In the dependent restriction, there is no order. Therefore to demonstrate that the execution is EC, we can linearize the dependent cuts to subsequences of $✗1^c ✗0^d +0^a +1^b$.

Let us now consider a SET variant in which it is possible to atomically test for membership of two elements. As a result, all labels become mutually dependent and the dependent projection is identical to the execution itself. Even in this case, the example remains EC. We can linearize $\{a, b\}$ either as $+0^a+1^b$ or $+1^b+0^a$. Either way, the non-mutators linearize as $+0^a ✗1^c$ and $+1^b ✗0^d$. Note, however, that there is no way to linearize all four events. To allow examples such as this to be EC, we limit dependent cuts so that no two dependent non-mutators can occur in the same cut. □

Example 12. Consider the following OR-set execution.

$$\begin{tikzcd}
+0^a \arrow[r] \arrow[rrd] & -0^b \arrow[r] \arrow[rrd] & ✓0^c \arrow[r] & ✗0^d \\
+0_e \arrow[r] \arrow[rru] & -0_f \arrow[r] \arrow[rru] & ✓0_g \arrow[r] & ✗0_h
\end{tikzcd}$$

This is EC taking the mutator witnesses to be subsequences of $+0 -0 +0 -0$, as follows. (We show $\{a, e\}$ twice to emphasize the symmetry.)

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

Were we to linearize events rather than actions (as in [Jagadeesan and Riely 2015]), this execution would fail to be validated. Suppose we were to pick the event order as $a\,b\,e\,f$. (All other choices lead to similar problems.) Since g sees f, every possible witness for g must end with mutator f. Indeed the only possible witness is $a\,e\,f\,g$. However, $+0+0-0✓0$ is not a valid specification string. □

Example 13. Consider the following OR-set execution.

$$\begin{tikzcd}
+0^a \arrow[r] & +0^b \arrow[r] & -0^c \arrow[r] & ✗0^d \\
 & +0_e \arrow[r] & -0_f \arrow[r] & ✗0_g
\end{tikzcd}$$

This is EC taking the mutator witnesses to be subsequences of any interleaving of $+0+0-0$ and $+0-0$. The witnesses for the two accessors are as follows.

$$+0^a+0^b-0^c\,\text{✗}0^d \qquad\qquad +0^e-0^f\,\text{✗}0^g$$

There is no interleaving that has both of these as prefixes. Thus, we must consider subsequences in the definition, rather than restricting attention to prefixes. □

Example 14. Consider the following execution using the standard SET interface.

$$\begin{tikzcd}
+0^a \arrow[r] \arrow[rrd] \arrow[rrddd] & -0^b & \\
& & \checkmark 0^f \\
+0^c \arrow[rru] \arrow[rrd] & & \\
& & \text{✗}0^g \\
+0^d \arrow[r] & -0^e \arrow[ruuu] \arrow[ru] &
\end{tikzcd}$$

This execution is not EC as a SET. The only hope is to derive witnesses from $+0-0+0+0-0$. Looking only at the accessors, this is fine:

$$+0^d-0^e+0^a+0^c\,\checkmark 0^f \qquad\qquad +0^a+0^c+0^d-0^e\,\text{✗}0^g$$

However, there is no choice for $\{a, c, d, e\}$ that is consistent with both of these strings.

It is important that this execution not be deemed EC, since the prefix without b is not EC. □

Example 15. In the SET specification, the mutators involving 0 and 1 are independent. We do not require preservation of order across independent operations. Consider the following SET trace.

$$\begin{tikzcd}
+0^a \arrow[r] & +1^b \arrow[r] & -1^c \arrow[rd] & & \\
& & & \checkmark 0^g \arrow[r] & \checkmark 1^h \\
+1^d \arrow[r] & +0^e \arrow[r] & -0^f \arrow[ru] & &
\end{tikzcd}$$

The trace is EC, as can be seen by taking the mutator witness for each event set to be subsequences of $+0-0+0$ and $+1-1+1$, as follows.

$+0^a$	$+1^d$
$+0^e$	$+1^b$
$+0^e-0^f$	$+1^b-1^c$
$+0^e+0^a$	$+1^b+1^d$
$+0^e-0^f+0^a$	$+1^b-1^c+1^d$
$+0^e-0^f+0^a\,\checkmark 0^g$	$+1^b-1^c+1^d\,\checkmark 1^h$

This example fails to be EC if we ignore dependency, as in [Jagadeesan and Riely 2015]. There is no interleaving of $+0-0+0$ and $+1-1+1$ that linearizes the accessors in the execution, and thus no way to satisfy both affirmative responses.

Our approach is reminiscent of RMO's treatment of different variables: recall that RMO does not require any preservation of order among operations on different variables. □

Example 16. The aim of this example is to demonstrate that stuttering is essential when looking at subsequences. This example also uses the standard SET interface. It is intended to be an execution of the OR-set. Because of the large number of mutators, we don't describe any accessors, instead recalling that in the OR-set a cut validates ✓0 if and only if it has a maximal +0.

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

This execution is EC. Anticipating our proof for the OR-set, we provide a recipe for the mutator sequence for each cut; namely, when linearizing a SET, we will maximize the number of alternations from -0 to +0 and use stuttering to remove adjacent and identical mutators. This results in the following table, where we show the sequence of labels (on the right) for the cuts (on the left, identified by their set of maximal events).

$\{a\},\{d\}$	: +0
$\{b\},\{c\},\{b,c\},\{e\}$	: +0-0
$\{b,d\},\{c,d\},\{b,c,d\},\{e,a\}$	: +0-0+0
$\{g\},\{h\},\{g,h\},\{b,e\},\{c,e\},\{b,c,e\}$	: +0-0+0-0
$\{g,a\},\{h,a\},\{g,h,a\},\{b,f\},\{c,f\},\{b,c,f\}$	: +0-0+0-0+0
$\{b,c,g\},\{b,c,h\},\{b,c,g,h\}$	: +0-0+0-0+0-0

We will show that this execution is not EC if we do not use stuttering equivalences in the definition of subsequence. Consider the cut with maximal elements $\{c,b,e\}$. The sequence +0-0+0-0-0 cannot be used since it doesn't permit the validation of the sub-cut $\{c,b,d\}$ using a subsequence. Thus, the cut has to be linearized to: +0-0-0+0-0. Thus, monotonicity forces the cut with maximal elements $\{c,b,f\}$ has to be associated with: +0-0-0+0-0+0.

Now, consider the set of all mutators. The sequence +0-0-0+0-0+0-0-0 cannot be used since there is no subsequence to validate the cut with maximal events $\{g,h,a\}$. So, the sequence to linearize the set of all mutators has to be: +0-0+0-0-0+0-0-0. However, this choice does not permit the linearization for $\{c,b,f\}$ as a subsequence. □

Example 17 (Distributed text editors). We consider a variant of the collaborative text editing protocol of [Attiya et al. 2016]. Let a, b range over text identifiers. Labels have the following forms:

– Mutator !a initializes the text to a.
– Mutator a<b adds a immediately before b.
– Mutator a>b adds a immediately after b.
– Mutator -a removes a.
– Non-mutator ?$a_1 \cdots a_n$ returns the current state of the document.

Two labels from this alphabet are dependent if they mention overlapping sets of text identifiers, or if one is a query and the other is a remove.

The following is an example specification string.

!c; b<c; d>c; ?bcd; a<b; e>d; -b; -d; ?ace

The following execution is EC with respect to this specification trace.

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

Crucially, label *-b* is independent of *e>d*, and label *-d* is independent of *a<b*. Thus, the dependent restriction of this execution is as follows.

$$\begin{tikzcd}
 & & & \boxed{a<b} \arrow[ddr] & \\
 & \boxed{b<c} \arrow[urr] \arrow[rr] \arrow[dr, rightsquigarrow] & & \boxed{-b} \arrow[dr] & \\
\boxed{!c} \arrow[ur, rightsquigarrow] \arrow[dr, rightsquigarrow] \arrow[rr, rightsquigarrow] \arrow[rrrr, rightsquigarrow, bend left=60] & & \boxed{?bcd} & & \boxed{?ace} \\
 & \boxed{d>c} \arrow[ur, rightsquigarrow] \arrow[rr] \arrow[drr] & & \boxed{-d} \arrow[ur] & \\
 & & & \boxed{e>d} \arrow[uur] &
\end{tikzcd}$$

We speculate that all executions of this variant of [Attiya et al. 2016] are EC, so long as each text identified is inserted at most once. □

5 Establishing that a data structure is EC

In this section, we look at the problem of establishing that a data structure implementation is EC, using the OR-set as an example. In the process, we show that definition of EC is expressive enough to validate the OR-set. We note that this implicitly also validates the *grow only* G-set that doesn't permit removes.

Rather than speak about a particular implementation of OR-set, we identify a property of the order-relation in the LVOs generated by an OR-set implementation (see [Shapiro et al. 2011] for details).

- Every `-k` is preceded by a `+k` with no intervening `-k`.
- Every ✓`k` is preceded by an `+k` with no intervening `-k`. Every path from an `+k` to ✗`k` contains an intervening `-k`.
- The order relation is transitive.

The first property captures the constraint in OR $-$ *set* that the remove has to specify an "observable" element that is present in the SET. The second constraint captures the priority given to `+k` when it is concurrent with `-k`. The only way for an accessor to k to return false is if every `+k` is masked by an intervening `-k`. The transitivity of the LVO follows from the assumption of causal delivery [Shapiro et al. 2011].

We show that any LVO u that satisfies the above properties is eventually consistent against the standard sequential SET specification.

Since the methods on different elements are independent, and the dependency relation for the SET interface is an equivalence, we are able to simplify the verification (in a sense made completely precise by composition Proposition 26. For now, we proceed formally as follows:

Definition 18. $\mathscr{D} \subseteq 2^{\mathbf{L}}$ is a *dependency partitioning* of $\mathbf{L}$ if

– $\forall D \in \mathscr{D}.\ \forall a, b \in D.\ a \# b$, and
– $\forall D, D' \in \mathscr{D}.$ either $D = D'$ or $\forall a \in D.\ \forall b \in D'.\ \neg(a \# b)$. □

Lemma 19. Let $\mathscr{D} \subseteq 2^{\mathbf{L}}$ be a dependency partitioning of $\mathbf{L}$.
$\forall D \in \mathscr{D}.\ \forall \sigma \in D^*.\ \forall \tau \in (\mathbf{L} \setminus D)^*.\ (\sigma \mathbin{|||} \tau) \cap \Sigma \neq \emptyset$ *implies* $(\sigma \mathbin{|||} \tau) \subseteq \Sigma$. □

Thus, without loss of generality, it suffices for us to address the case when u only involves operations on a single element, say 0. We proceed as follows.

To any such LVO u, we associate the linearization, say τ_u, that has the maximum number of adjacent labels of the form `-0+0`, *i.e.*, the maximum number of changes from a `-0` to an `+0` label. Below, we summarize some of the key properties of such a linearization that follow immediately from the definition.

(a) τ_u ends with `+0` if and only if there is an `+0` that is not followed by any `-0` in u.
(b) For any LVO $v \subseteq u$, τ_v has at most as many changes from a `-0` to an `+0` label as τ_u.

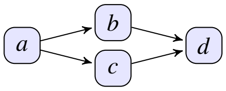
We need to check that these linearizations satisfy all the desired properties. The first property above ensures that the accessors are validated correctly, *i.e.*, 0 is deemed to be present iff there is an `+0` that is not followed by any `-0`. The second property ensures monotonicity, *i.e.*, if $v \subseteq u$, then $\tau_v \lesssim_{\mathsf{seq}} \tau_u$ (see examples in subsection 2.4).

6 Simulation

In this section, we define maps from traces and specifications to LTSs, such that the following holds: u is EC iff $\mathsf{lts}(u) \sqsubseteq\!\!\!\sim \mathsf{lts}(\Sigma_u)$. The purpose of the intermediate LTS is to make explicit all of the paths through the trace (or specification). Whereas branching in an LVO represents concurrency, branching in the corresponding LTS represents nondeterminism.

As usual for true concurrency semantics, the labels of the LTSs are LVOs rather than single actions. We only consider LTSs with labels derived from $\mathscr{L}_\#$ (subsection 3.1).

Neither the carrier set nor the replica identifier matter for simulation, therefore we treat labels up to isomorphism, as defined in section 3. When drawing LTSs, we use standard pomset syntax for labels that are LPOs: concatenation of actions represents sequencing and | is used to represents parallelism. Thus $a(b|c)d$ corresponds to the LPO:



The definitions of LTS and of simulation are standard. We state them here for completeness. Let $\mu, \nu \in \mathscr{L}_\# \uplus \{\varepsilon\}$ range over *labels* of the LTS, where ε represents the silent transition (used in subsection 6.4).

Definition 20. An LTS is a triple $P = \langle S_P, p_0, \longmapsto_P \rangle$ where $p_0 \in S_P$ and $\longmapsto_P : S_P \times (\mathscr{L}_\# \uplus \{\varepsilon\}) \times S_P$. A relation $\mathscr{R}$: $S_P \times S_Q$ is a *(strong) simulation* if $p \overset{\mu}{\longmapsto}_P p'$ and $p \mathrel{\mathscr{R}} q$ implies that there exist $\nu =_{\mathsf{iso}} \mu$ and $q' \in S_Q$ such that $q \overset{\nu}{\longmapsto}_Q q'$ and $p' \mathrel{\mathscr{R}} q'$. Write $P \sqsubseteq\!\!\!\sim Q$ if there exists a simulation $\mathscr{R}$: $S_P \times S_Q$ such that $p_0 \mathrel{\mathscr{R}} q_0$. Write $P \approx Q$ if $P \sqsubseteq\!\!\!\sim Q$ and $Q \sqsubseteq\!\!\!\sim P$. For sets of LTSs, write $\mathscr{P} \sqsubseteq\!\!\!\sim \mathscr{Q}$ if $\forall P \in \mathscr{P}.\ \exists Q \in \mathscr{Q}.\ P \sqsubseteq\!\!\!\sim Q$. □

As usual, we write $p \longmapsto p'$ if $p \overset{\mu}{\longmapsto} p'$ for some μ and $(p \overset{\mu}{\longmapsto})$ if $p \overset{\mu}{\longmapsto} p'$ for some p'.

6.1 Implementation LTS

Let $\mathsf{max}(v)$ be the suborder containing only the maximal elements of v, and $\overline{\mathsf{max}}(v)$ to be the suborder with the maximal elements removed.

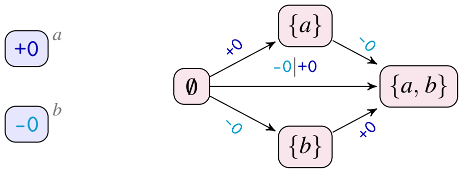
$$\mathsf{max}(v) = v \downharpoonright \{e \mid \nexists d \in \mathsf{E}_v.\ e \leadsto_v d\} \qquad \overline{\mathsf{max}}(v) = v \downharpoonright \{e \mid \exists d \in \mathsf{E}_v.\ e \leadsto_v d\}$$

Let $\longmapsto_{\mathsf{i}} : \mathscr{L} \times \mathscr{L}_{\#} \times \mathscr{L}$ be defined as follows: $p \overset{v}{\longmapsto}_{\mathsf{i}} q$ when[3]

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

These conditions ensure that if $p \overset{v}{\longmapsto}_{\mathsf{i}} q$, then $v \in \mathsf{cuts}_{\#}(q)$.

Let $\mathsf{lts}(u) = \langle \mathsf{cuts}(u), \emptyset, \longmapsto_{\mathsf{i}} \rangle$. As an example, let u be the LPO on the left below, which executes +0 in parallel with -0. Then $\mathsf{lts}(u)$ is given on the right below.



In this simple example, the labels on the transitions do not have any causal history; so, we only see (multisets of) labels rather than full pomsets.

6.2 Specification LTS

Let $\longmapsto_{\mathsf{s}} : \Sigma \times \mathscr{L}_{\#} \times \Sigma$ be defined as follows: $\sigma \overset{v}{\longmapsto}_{\mathsf{s}} \rho$ when[4]

$$\begin{array}{cc} \sigma \lesssim_{\mathsf{seq}} \rho & \exists \rho' \lesssim_{\mathsf{seq}} \rho.\ v \text{ linearizes to } \rho' \\ \rho \in \sigma \mathbin{|||} \mathsf{max}(v) & \exists \sigma' \lesssim_{\mathsf{seq}} \sigma.\ \overline{\mathsf{max}}(v) \text{ linearizes to } \sigma' \end{array}$$

Using the terminology from subsection 2.4, these conditions ensure that if $\sigma \overset{v}{\longmapsto}_{\mathsf{s}} \rho$, then ρ is one of the minimal upper bounds of σ and v in $\langle \Sigma, \lesssim_{\mathsf{seq}} \rangle$. Thus, the sequence at a state reachable from the initial state can be seen as being one of the miminal upper bounds of by the set of labels on the transitions verifying the reachability.

Let $\mathsf{lts}(\Sigma) = \langle \Sigma, \varepsilon, \longmapsto_{\mathsf{s}} \rangle$.

When drawing pictures, it is useful to consider automata on single specification strings. In order to use the smallest possible state set, define $\mathsf{strict\text{-}subseq}(\sigma) = \{\sigma' \mid \sigma' \leq_{\mathsf{seq}} \sigma\}$ and let $\mathsf{lts}(\sigma) = \langle \mathsf{strict\text{-}subseq}(\sigma), \varepsilon, \longmapsto_{\mathsf{s}} \rangle$. We have that $P \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{lts}(\Sigma)$ iff $\exists \sigma \in$

[3] For linearizability, the definition is the same, except labels are chosen from $\mathscr{L}$ rather than $\mathscr{L}_{\#}$.
[4] For linearizability, the definition is the same, except labels are chosen from $\mathscr{L}$ rather than $\mathscr{L}_{\#}$ and the order is defined using $\leq_{\mathsf{pre}}$ rather than $\lesssim_{\mathsf{seq}}$.

$\Sigma.\ P \underset{\sim}{\sqsubset} \mathsf{lts}(\sigma)$. For example, the SET LTS for strings +0-0 and -0+0 are as follows.

$$\begin{tikzcd}
 & \boxed{\texttt{+0}} \arrow[dr, bend left, "\texttt{-0}"] \arrow[dr, bend right, "\texttt{+0-0}"] & & & & \boxed{\texttt{+0}} \arrow[dr, "\texttt{-0}"] & \\
\boxed{\varepsilon} \arrow[ur, "\texttt{+0}"] \arrow[dr, "\texttt{-0}"'] \arrow[rr, "\texttt{+0|-0}"'] & & \boxed{\texttt{+0-0}} & & \boxed{\varepsilon} \arrow[ur, "\texttt{+0}"] \arrow[dr, "\texttt{-0}"'] \arrow[rr, "\texttt{+0|-0}"] & & \boxed{\texttt{-0+0}} \\
 & \boxed{\texttt{-0}} \arrow[ur, "\texttt{+0}"'] & & & & \boxed{\texttt{-0}} \arrow[ur, bend left, "\texttt{-0+0}"] \arrow[ur, bend right, "\texttt{+0}"'] &
\end{tikzcd}$$

The specification LTS is not deterministic. For example there are the two transitions with label +0|-0 from the initial state ε, shown over the two figures.

We draw attention to the two transitions from the -0 state to the -0+0 state in the righthand figure. The transition labeled +0 has no causal history, so is concurrent with the -0 label that is already received at this state. On the other hand, the transition labeled -0+0 reflects a +0 transition that causally follows the -0 label that is already received at this state.

6.3 Soundness and completeness of simulation

Theorem 21. u is EC *iff* $\mathsf{lts}(u) \underset{\sim}{\sqsubset} \mathsf{lts}(\Sigma)$.

PROOF. (Sketch) For the forward direction, since u is EC, there exists a function τ : $\mathsf{cuts}_{\#}(u) \to \Sigma$ such that $\forall E \in \mathsf{cuts}_{\#}(u).\ \tau(E)$ is a linearization of E, We define the required simulation $\mathscr{R}$: $\mathsf{cuts}(u) \times \Sigma$ as follows.

$$\forall p \in \mathsf{cuts}(u).\ \ p\,\mathscr{R}\,\tau(p \lfloor \# \lfloor \mathbf{M})$$

Consider a transition $p \overset{v}{\longmapsto}_{\mathsf{i}} q$. It follows from EC-monotonicity that $\tau(p \lfloor \# \lfloor \mathbf{M}) \lesssim_{\mathsf{seq}} \tau$ $(q \lfloor \mathbf{M})$. Since $v \in \mathsf{cuts}_{\#}(q)$, the mutators in $\tau(v)$ linearize to a subsequence that is $\lesssim_{\mathsf{seq}} \tau(q \lfloor \# \lfloor \mathbf{M})$. The non-mutators of $\tau(v)$ are consistent with this subsequence by EC-monotonicity. Using the property that "Independent labels can be removed" in specifications, we deduce that the non-mutators of $\tau(v)$ are consistent with all of $\tau(q \lfloor \# \lfloor \mathbf{M})$. Thus, we deduce that $\tau(p \lfloor \# \lfloor \mathbf{M}) \overset{v}{\longmapsto}_{\mathsf{s}} \tau(q \lfloor \# \lfloor \mathbf{M})$.

For the converse, we are given a simulation $\mathscr{R}$: $\mathsf{cuts}(u) \times \Sigma$. A simple inductive proof demonstrates that: $(\forall p \in \mathsf{cuts}_{\#}(u))$, there is a transition sequence of the form $\emptyset \longmapsto_{\mathsf{i}} p$ which can be taken to be in a special form $\emptyset \longmapsto_{\mathsf{i}} q \overset{v}{\longmapsto}_{\mathsf{i}} p$ in the case where there exists $v \in \mathsf{cuts}_{\#}(u)$ such that $\mathsf{E}_{\mathsf{max}(v)} = \mathsf{E}_{\mathsf{max}(p)}$. In particular, since the initial state $\emptyset$ is in the domain of the simulation relation, every $v \in \mathsf{cuts}_{\#}(u)$ is in the domain of $\mathscr{R}$; the label on the final transition into v ensures that the σ related to v is a linearization of v. We define $\tau(v) = \sigma$, choosing one amongst the possibly many σ that can be constructed this way. A simple inductive proof shows that $\forall p, q \in \mathsf{cuts}(u).\ p \subseteq q$ implies $p \longmapsto_{\mathsf{i}}^{*} q$. Thus $\tau(v) \lesssim_{\mathsf{seq}} \tau(w)$, by the properties of $\mathscr{R}$ and the definition of τ. □

6.4 Client interaction

In [Jagadeesan and Riely 2015] we gave a concrete syntax for clients. Here we model clients abstractly as sets of LPOs. For example, consider sequential SET client that checks for membership of 0, then checks for membership of 1, then adds 2 in the case that the results where the same. This is modeled as the prefix closed set containing the LTOs ✓0✓1+2, ✓0✗1, ✗0✓1, and ✗0✗1+2. As a second example, consider a client with two threads, one adding 0 and testing membership for 0. This is modeled as the prefix closed set containing the LPOs ✓0|+0 and ✗0|+0. Client LPOs can be converted into LTSs, as described in subsection 6.1.

The relation $\Vert\!\lceil$ is defined between LTSs so that $P \Vert\!\lceil Q$ describes the system that results when client P interacts with data structure Q. The definition is then lifted to sets: $\mathscr{P} \Vert\!\lceil \mathscr{Q} = \bigcup_{P\in\mathscr{P},Q\in\mathscr{Q}} P \Vert\!\lceil Q$. The $\Vert\!\lceil$ operator is asymmetric in two ways:

- All of the actions of the client P must be matched by Q. Otherwise $P \Vert\!\lceil Q = \emptyset$. Actions of the data structure Q may not be matched by P; they may instead be propagated to other clients. We expect that $(P_1 \mid P_2) \Vert\!\lceil Q \approx P_1 \Vert\!\lceil (P_2 \Vert\!\lceil Q)$.
- The data structure Q may introduce order not found in the clients. This ensures that the composition of client ✓0|+0 with the SET data structure is nonempty.

The formal definition is as follows. From section 3, recall that we write $\subseteq$ for suborder and $=_{\mathsf{iso}}$ for isomorphism. From subsection 6.1, recall that $\mathsf{max}(u)$ is the suborder including only the maximal elements of u.

Definition 22. For LTSs P and Q, define $\longmapsto_\times$ inductively, as follows.

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

Let $S_\times = \{\langle p,q\rangle \mid \exists\langle p',q'\rangle.\ \langle p,q\rangle \longmapsto_\times^* \langle p',q'\rangle \text{ and } \nexists p''.\ p \longmapsto_P p''\}$
Let $P \Vert\!\lceil Q = \langle S_\times, \langle p_0,q_0\rangle, \longmapsto_\times\rangle$. □

In the case that $S_\times$ is empty, the composition gives the empty LTS.

A couple of comments are in order about the operational consequences of this definition.

- Replica identities do not play a role in the definition. Thus, we are permitting implicit mobility of the client amongst replicas. The only constraint — enforced by the synchronization on the labels — is that the replica has at least as much history on the current item of interaction as the client.
- The definition includes the case where the client itself is replicated. However, in this case, 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.

We can also define restriction, a lá CCS, simply by removing all edges with labels from the given set A

Definition 23. $P\backslash A = \langle S_P, p_0, \{\langle p,a,q\rangle \mid \langle p,a,q\rangle \in (\longmapsto_P) \text{ and } a \notin A\}\rangle$ □

The definitions lift to sets as follows: $\mathscr{P} \Vert\!\lceil \mathscr{Q} = \{(P \Vert\!\lceil Q) \mid P \in \mathscr{P} \text{ and } Q \in \mathscr{Q}\}$ and $\mathscr{P}\backslash A = \{(P\backslash A) \mid P \in \mathscr{P}\}$. Simulation is a precongruence for composition and restriction.

Lemma 24. If $\mathscr{P} \precsim \mathscr{P}'$ and $\mathscr{Q} \precsim \mathscr{Q}'$ then $\mathscr{P} \Vert\!\lceil \mathscr{Q} \precsim \mathscr{P}' \Vert\!\lceil \mathscr{Q}'$ and $\mathscr{P}\backslash A \precsim \mathscr{P}'\backslash A$. □

7 Reasoning with eventual consistency

In this section, we address eventual consistency from the client's perspective. How can a client make use of the fact that it is running against an EC data structure? First we consider a few basic properties.

7.1 Basic Properties

Prefix closure. If v is EC and $u \leq_{\mathsf{pre}} v$, then u is also EC.

Quiescent extension. An EC trace can always be extended to an EC trace in which all mutators are visible at every replica. We formalize this property as follows. Let $\wr_u e = \{d \in \mathsf{E}_u \mid \lambda_u(d) \in \mathbf{M} \text{ and } d \rightsquigarrow_e\}$. Fix trace u and let $D = \{d \in \mathsf{E}_u \mid \lambda_u(d) \in \mathbf{M}\}$ be the set of mutators in u. Then u is *quiescent* if $\forall p \in \mathbf{R}.\ \exists e \in \mathsf{E}_u.\ \rho_u(e) = p$ and $\wr_u e = D$. If u is EC, then there exists a quiescent extension $v \geq_{\mathsf{pre}} u$ that is also EC.

Permutation equivalence. [Bieniusa et al. 2012] state the following principle of permutation equivalence: "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." Any EC implementation satisfies this principle because every dependent set of mutators is linearized — so, in particular, we enforce a stronger property that there are no new intermediate states of the data structure over a purely concurrent system.

Strong consistency Strong consistency, is defined in [Shapiro et al. 2011] to identify those CRDTs that satisfy a kind of "Church-Rosser" theorem and do not permit rollback; thus, in such a replicated implementation of a data structure, the arrival of new mutators does not alter the ordering of old mutators. Every EC implementation is strongly consistent in this sense, as a consequence of EC-monotonicity.

7.2 Abstraction results

A client can program against the specification if the implementation is EC. We demonstrate this by showing that simulation is a congruence for the composition operator. The structure of this proof directly follows the proof techniques of [Filipovic et al. 2010], albeit in ma very different context.

Theorem 25. If u is EC *for Σ, then $\mathscr{P} \mathbin{\|\!\lceil} \mathsf{lts}(u) \underset{\sim}{\sqsubset} \mathscr{P} \mathbin{\|\!\lceil} \mathsf{lts}(\Sigma)$.*

PROOF. (Sketch) By Theorem 21, it suffices to show that: $P \underset{\sim}{\sqsubset} \mathsf{lts}(u)$ implies $\mathscr{P} \mathbin{\|\!\lceil} \mathsf{lts}(u) \underset{\sim}{\sqsubset} \mathscr{P} \mathbin{\|\!\lceil} \mathsf{lts}(\Sigma)$. Let $\mathscr{R}$ be a witness for $P \underset{\sim}{\sqsubset} \mathsf{lts}(u)$. The proof proceeds by constructing a "product" simulation relation of the identity on the states of P with $\mathscr{R}$, *i.e.*:

$$q \mathrel{\mathscr{R}} q' \text{ implies } \langle p, q\rangle \mathrel{\mathscr{S}} \langle p, q'\rangle \qquad \square$$

We view the simplicity of the proof of this theorem as a testament to the efficacy of our framework.

In restricted situations, the client view is simplified further. We discuss briefly below.

Relation to linearizability Every linearizable trace is also EC. The converse also holds under various assumptions. Suppose that u is EC.

If u contains only mutators, all of whom are pairwise dependent, then u is also linearizable. Thus, EC affords extra freedom for non-mutators by ignoring order from them and by not forcing non-mutators to be part of the linearization.

If there is a total order on the mutators of u and $\rightsquigarrow_u$ is transitive, then u is also linearizable. Two special cases, STS and SINGLE-MASTER, are mentioned in the introduction.

In our prior paper [Jagadeesan and Riely 2015], we explored a further special case where the client can indeed assume that they are programming against the sequential interface. Consider clients, all of whose executions are *logically monotone* in the sense that they satisfy the CALM principle [Hellerstein 2010]. In a logically monotone execution, the arrival time of a concurrent mutator does not alter the evolution of the system, *i.e.*, there are no “1races” between concurrent mutators and mutators/accessors[5]. This restriction is an analogue of the DRF property of relaxed memory models, and includes those written in languages that realize the CALM principle, such as Bloom [Conway et al. 2012]. The simplified programmer perspective mimics the guarantees provided for data-race free programs. The treatment of this case follows our prior work Jagadeesan and Riely [2015] and we do not describe it any further.

7.3 Composition

We turn our attention to a composition result in the style of [Herlihy and Wing 1990]. Given two *non-interacting* data structures whose replicated implementations satisfy their sequential specifications, we show that the implementation that combines them satisfies the interleaving of their specifications.

Given an trace u and $L \subseteq \mathbf{L}$, write $u \downharpoonright L$ for the trace 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: $U \downharpoonright L = \bigcup_{u \in U} \{u \downharpoonright L\}$.

Proposition 26 (Composition). Write $u \vDash_{\mathsf{ec}} \Sigma_u$ to mean that u is EC *with respect to Σ_u.*

Let L_1 and L_2 be mutually independent subsets of $\mathbf{L}$, using the notion of dependency from subsection 2.1. 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 $(U \downharpoonright L_1) \vDash_{\mathsf{ec}} \Sigma_1$ and $(U \downharpoonright L_2) \vDash_{\mathsf{ec}} \Sigma_2$ then $U \vDash_{\mathsf{ec}} (\Sigma_1 \mathbin{|||} \Sigma_2)$. □

It is also possible to formalize this result as using the interleaving operator on LTSs. If the labels of Σ_1 and Σ_2 are independent, we have $\mathsf{lts}(\Sigma_1 \mathbin{|||} \Sigma_2) \approx \mathsf{lts}(\Sigma_1) \mathbin{|||} \mathsf{lts}(\Sigma_2)$

7.4 Graph is correct

We now show an example of the use of the results in the previous sections. Shapiro et al. [2011] give a construction of a graph using OR-sets. We show that it is sound to program the graph against the specification of SET.

[5] Our formalization of logically monotone executions was inspired by Panangaden et al. [1990]; Panangaden and Stark [1988], where a monotone node is insensitive to the arrival order of the inputs and a concurrent input action (mutator) does not disable an output action (accessor) at a monotone node.

We have two separate and independent sets: $\mathbf{L}_{\Sigma_1} \cap \mathbf{L}_{\Sigma_2} = \emptyset$. Suppose we have two implementations, each of which is correct individually: $\mathsf{lts}(U_i) \precsim \mathsf{lts}(\Sigma_i)$. By composition, we have that they are correct when composed together: $U_1 \mid\mid\mid U_2 \precsim \Sigma_1 \mid\mid\mid \Sigma_2$.

Let $\mathscr{P}$ be the graph implementation, which is a client of the two sets. By abstraction, we know that $\mathscr{P} \parallel\!\!\lceil\, (\Sigma_1 \mid\mid\mid \Sigma_2) \precsim T$ implies $\mathscr{P} \parallel\!\!\lceil\, (U_1 \mid\mid\mid U_2) \precsim T$. Thus, by congruence of these properties, we deduce:

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

The hypothesis of the above implication involves the graph client interacting with the specification automaton for the composition of two independent sets. The investigation of the proof of the hypothesis is the matter of future work. In this paper, we merely note that the methods of traditional concurrency apply. Indeed, this portion of our paper is that it is provides an alternative perspective of the proof of the proof rules in particular, the event based proof rule of Figure 8) of Gotsman et al. [2016].

Bibliography

J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012. doi: 10.1007/s10703-012-0161-5. URL http://dx.doi.org/10.1007/s10703-012-0161-5.

K. Aslan, P. Molli, H. Skaf-Molli, and S. Weiss. C-set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on REsource Discovery*, 2011.

H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In G. Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 259–268. ACM, 2016. ISBN 978-1-4503-3964-3. doi: 10.1145/2933057.2933090. URL http://doi.acm.org/10.1145/2933057.2933090.

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. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3. URL http://dl.acm.org/citation.cfm?id=302430.302438.

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 Berlin Heidelberg, 2012.

ISBN 978-3-642-28868-5. URL http://dx.doi.org/10.1007/978-3-642-28869-2_4.

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

N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing*, pages 1:1–1:14, 2012.

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. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL http://doi.acm.org/10.1145/1323293.1294281.

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

I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Comp. Sci.*, 411:4379–4398, 2010.

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 R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384. ACM, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837625. URL http://doi.acm.org/10.1145/2837614.2837625.

C. A. Gunter. Universal profinite domains. *Inf. Comput.*, 72(1):1–30, 1987. doi: 10.1016/0890-5401(87)90048-4. URL http://dx.doi.org/10.1016/0890-5401(87)90048-4.

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.

J. M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5–19, Sept. 2010.

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 M. Valero, V. K. Prasanna, and S. Vajapeyam, editors, *High Performance Computing - HiPC 2000, 7th International Conference, Bangalore, India, December 17-20, 2000, Proceedings*, volume 1970 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2000. ISBN 3-540-41429-0. doi: 10.1007/3-540-44467-X_32. URL http://dx.doi.org/10.1007/3-540-44467-X_32. To appear in ACM Transactions on Computer Systems.

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. ISBN 978-3-662-47665-9.

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

P. Panangaden and E. W. Stark. Computations, residuals, and the power of indeterminacy. In *ICALP '88*, pages 439–454, 1988.

P. Panangaden, V. Shanbhogue, and E. W. Stark. Stability and sequentiality in dataflow networks. In *ICALP '90*, pages 308–321, 1990.

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, 2011.

D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19, 2016. doi: 10.1145/2926965. URL http://doi.acm.org/10.1145/2926965.

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

A Example specifications

The results of the paper are not limited to sets. In this appendix, we give other example specifications. Some of these are parameterized by countable sets of values, I, and variables, X.

Define specification $\textsc{reg}_X$ for registers over variables X and values $\mathbb{N}$. Let $\mathtt{W}x i$ denote a write to x with value i and $\mathtt{R}x i$ denote a read of x returning i.

$$\begin{aligned}
\mathbf{M} &= \cup_{x\in X}\ \{\mathtt{W}x0, \mathtt{W}x1, \ldots\}\\
\overline{\mathbf{M}} &= \cup_{x\in X}\ \{\mathtt{R}x0, \mathtt{R}x1, \ldots\}\\
\mathscr{A} &= \cup_{x\in X}\ \big\{\{\mathtt{W}x0\}, \{\mathtt{W}x1\}, \ldots, \{\mathtt{R}x0, \mathtt{R}x1, \ldots\}\big\}\\
\mathscr{D} &= \cup_{x\in X}\ \big\{\{\mathtt{W}x0, \mathtt{W}x1, \ldots, \mathtt{R}x0, \mathtt{R}x1, \ldots\}\big\}\\
\Sigma &= |||_{x\in X}\ [\![\mathtt{R}x0^{*}\ \textstyle\sum_{i\in\mathbb{N}} \big(\mathtt{W}x i\ \ \mathtt{R}x i^{*}\big)^{*}]\!]
\end{aligned}$$

Actions on different variables are independent.

It is interesting to consider the memory model that result by applying $\textsc{ec}$ to this specification. The $\textsc{ec}$ model gives per-variable $\textsc{sc}$ (*a.k.a.*, coherence). For example (`wx1 | wx2 | rx1;rx2 | rx2;rx1`) is not $\textsc{ec}$. Across variables, however, it is very permissive, allowing out-of-thin-air. This is not surprising, since our model does not track data or control dependencies. One could imaging adding such things to the dependent restriction in order to forbid thin-air behaviors.

We now present some more inherently sequential specifications. While the definition of $\textsc{ec}$ applies here, as to any specification, the flexibility afforded by $\textsc{ec}$ is greatly reduced by the fact that accessors and mutators overlap. In this light, it appears that the concurrency allowed by CRDTs is inherently tied to the separation of accessors as non-mutators.

Define specification STACK$_I$ for stacks. Let $\texttt{+}i$ denote the push of i, $\texttt{-}i$ denote pop returning i, and $\checkmark i$ denote top returning i. We let $\texttt{-}_{\text{✗}}$ represents an underflow of pop and $\checkmark_{\text{✗}}$ represent an underflow of top.

$$\begin{aligned}
\mathbf{M} &= \cup_{i\in I}\ \{\texttt{+}i, \texttt{-}i\}\\
\mathbf{L} &= \{\texttt{-}_{\text{✗}}, \checkmark_{\text{✗}}\} \cup \bigcup_{i\in I}\ \{\checkmark i\}\\
\mathscr{A} &= \big\{\{\texttt{+}i\}\ \mid\ i\in I\big\} \cup \big\{\{\texttt{-}_{\text{✗}}\} \cup \{\texttt{-}i \mid i\in I\}\big\} \cup \big\{\{\checkmark_{\text{✗}}\} \cup \{\checkmark i \mid i\in I\}\big\}\\
\mathscr{D} &= \{\mathbf{L}\}\\
\Sigma &= [\![\mathbb{S}]\!] \text{ where } \mathbb{S}\ ::=\ (\checkmark_{\text{✗}} \mid \texttt{-}_{\text{✗}})^*\ (\mathbb{B}\ (\checkmark_{\text{✗}} \mid \texttt{-}_{\text{✗}})^*)^*\\
&\qquad\qquad\quad\ \mathbb{B}\ ::=\ \varepsilon \mid \textstyle\sum_{i\in I} \texttt{+}i\ \checkmark i^*\ \mathbb{B}\ \checkmark i^*\ \texttt{-}i\ \mathbb{B}
\end{aligned}$$

Define specification INC for atomic get-and-increment over the naturals. Let $\texttt{+}i$ denote get-and-increment returning i.

$$\begin{aligned}
\mathbf{L} = \mathbf{M} &= \{\texttt{+0}, \texttt{+1}, \ldots\}\\
\mathscr{A} = \mathscr{D} &= \{\mathbf{L}\}\\
\Sigma &= [\![\texttt{+0 +1 +2}\ \cdots]\!]
\end{aligned}$$

Define specification COUNT of increment/decrement counters, bottoming out at 0. Let + denote increment, - denote decrement, and $\checkmark i$ denote a query of the value of the counter, returning i. We presume that a decrement silently does nothing at 0; thus we have strings such as $\checkmark 0 + \checkmark 1 - \checkmark 0 - \checkmark 0$.

$$\begin{aligned}
\mathbf{M} &= \{+, -\}\\
\overline{\mathbf{M}} &= \{\checkmark 0, \checkmark 1, \ldots\}\\
\mathscr{A} &= \big\{\{+\}, \{-\}, \{\checkmark 0, \checkmark 1, \ldots\}\big\}\\
\mathscr{D} &= \{\mathbf{L}\}
\end{aligned}$$