A Generic Undo Support for State-Based CRDTs

Weihai Yu, Victorien Elvinger, Claudia-Lavinia Ignat

A Generic Undo Support for State-Based CRDTs

Weihai Yu
University of Tromsø - The Arctic University of Norway, Norway
weihai.yu@uit.no

Victorien Elvinger
Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France
victorien.elvinger@inria.fr

Claudia-Lavinia Ignat
Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France
claudia.ignat@inria.fr

Abstract

CRDTs (Conflict-free Replicated Data Types) have properties desirable for large-scale distributed systems with variable network latency or transient partitions. With CRDT, data are always available for local updates and data states converge when the replicas have incorporated the same updates. Undo is useful for correcting human mistakes and for restoring system-wide invariant violated due to long delays or network partitions.

There is currently no generally applicable undo support for CRDTs. There are at least two reasons for this. First, there is currently no abstraction that we can practically use to capture the relations between undo and normal operations with respect to concurrency and causality. Second, using inverse operations as the existing partial solutions, the CRDT designer has to hard-code certain rules and design a new CRDT for almost every operation that needs undo support.

In this paper, we present an approach to generic support of undo for CRDTs. The approach consists of two major parts. We first work out an abstraction that captures the semantics of concurrent undo and redo operations through equivalence classes. The abstraction is a natural extension of undo and redo in sequential applications and is straightforward to implement in practice. By using this abstraction, we then device a mechanism to augment existing CRDTs. The mechanism provides an "out of the box" support for undo without the involvement of the CRDT designers. We also present a practical application of the approach in collaborative editing.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Concurrent algorithms; Information systems → Data replication tools; Human-centered computing → Synchronous editors; Human-centered computing → Asynchronous editors

Keywords and phrases Data replication, eventual consistency, state-based CRDT, delta-state CRDT, concurrent undo

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.2

Funding The research was mostly carried out during Yu's sabbatical stay at Inria, LORIA, financed by University of Tromsø - The Arctic University of Norway.

Acknowledgements The authors thank the members of the COAST team at Inria, LORIA, in particular Matthieu Nicolas and Gérald Oster, for interesting discussions.

1 Introduction

The CAP theorem ([11, 14]) states that in a networked system, it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of that data for update and (P) tolerance to network partition. [7] revisited the theorem and clarified some common misunderstandings. Among these, the three properties are continuous rather than binary and partition is a function of

latency. [7] also referred to two useful approaches to partition management that are highly relevant to the work we are presenting in this paper: (1) CRDT, with which replicated data provably converge after a partition (delay) and (2) compensation (undo), which can be used to fix violation of global invariant during partition.

CRDTs [23], or Conflict-free Replicated Data Types, have properties desirable for large-scale distributed applications. A site independently updates its local replica (i.e. a site is always available for update). The states of replicas converge when they have incorporated the same set of updates (referred to as *strong convergence* in [23]). Applications adopting CRDTs include distributed key-value stores [8] and collaborative editors ([17, 29, 20, 26, 28]). There has also been active research on CRDT-based transaction processing ([2, 15, 19]).

During a network partition or after long network delay, systems' global invariant could be violated, such as overbooking of resources or pre-mature commit of sub-transactions [12]. For applications such as online shopping and collaborative editing, human users introduce additional delay and mistakes. Undo (or compensation [12]) is a generic tool to fix human mistakes or restore global invariant.

Currently, there is no generic support of undo for CRDTs. There are at least two reasons for this. First, there is currently no simple applicable abstraction that sufficiently captures the relations between undo and normal operations with respect to concurrency and causality. Second, using inverse operations as the existing partial solutions, the CRDT designer has to hard-code certain rules and design a new CRDT for almost every operation that needs undo support. We explain these issues in detail in later sections §5.1 and §4.3.

Our first contribution is an abstraction that defines the semantics of concurrent undo and redo operations through equivalence classes. The abstraction correctly captures concurrency and causality of undo and redo operations. It is a natural extension to undo and redo of sequential systems and hence is easy to understand and straightforward to implement in practice. The abstraction applies generally beyond the context of CRDTs.

Our second contribution is a generic approach to augmenting existing CRDTs with an "out of the box" support for undo. Unlike the current partial solutions where the CRDT designer has to design a new CRDT for nearly every inverse operation, with our approach, the CRDT designer does not have to get involved in the design of individual inverse operations.

The original CRDT paper [23] presented two families of CRDT approaches, namely state-based and operation-based. There have been improvement and refinement on both approaches, the most representative being [3] on state-based and [5] on operation-based. Our work focuses on state-based CRDTs and is based on [3].

The paper is organized as follows. §2 describes the model of the systems our work applies to and §3 presents the notations we use. §4 reviews the background of CRDTs. §5 presents our first main contribution, the abstraction for concurrent undo and redo operations. §6 describes our second main contribution, to generically support undo for existing CRDTs. §7 shows a practical application of our work in collaborative editing. §8 discusses related work. §9 concludes.

2 System Model

A distributed system consists of sites with globally unique identifiers. We use $\mathbb{I}$ for the set of site identifiers. Sites do not share memory. They maintain durable states. Sites may crash, but will eventually recover to the durable state at the time of the last crash.

A site can send messages to any other site in the system through an asynchronous and unreliable network. There is no upper bound on message delay. The network may discard,

reorder or duplicate messages, but it cannot corrupt messages. Through re-sending, messages will eventually be delivered. The implication is that there can be network partitions, but disconnected sites will eventually get connected.

3 Notations

$\mathbb{N}$ is the set of natural numbers. $\mathbb{B}$ is the set of Boolean values. $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$. $\mathcal{P}(S)$ denotes the power set on S. Most sets in this paper are partially ordered and have a least element $\bot$ (also known as the bottom element).

Set comprehension is of the form $\{x \in S |\ \mathsf{pred}(x)\}$ or $\{f(x) | x \in S\}$, where f is a function and pred is a predicate.

We use $m \colon K \hookrightarrow V$ to denote a partial function where $\mathsf{dom}(m) \subseteq K$. A partial function can be represented as a set of pairs $\{\langle k, m(k)\rangle | k \in \mathsf{dom}(m)\}$. When $k \in K \wedge k \notin \mathsf{dom}(m)$ and V has a bottom $\bot_V$, we use $m(k) = \bot_V$ for convenience. For example, given a partial function $p \colon \mathbb{N} \hookrightarrow \mathbb{N}$ and $\mathsf{dom}(p) = \varnothing$, we use $p(n) = 0$ for any $n \in \mathbb{N}$, because $\bot_{\mathbb{N}} = 0$. Due to this convenience, we do not need an initialization $p(n) = 0$ as in the case of a total function.

The notation $m\{k \mapsto v\}$ represents an update of the function m for a new value v associated with the key k.

The notation $\mathsf{f}(x)$ is like a function or procedure in a conventional programming language. In this paper, it can be a query, a mutator (an operation) or a predicate. We may write $\mathsf{f}_y(x)$ for $\mathsf{f}(x, y)$ to make the signatures of functions look consistent in different contexts. For example, $\mathsf{inc}(x)$ increments a counter x, while $\mathsf{inc}(x, A)$, or better $\mathsf{inc}_A(x)$, increments a counter x at site A.

4 CRDT Background

A CRDT is a data type specifically designed for data replicated at different sites. A site queries and updates its local replica independently (i.e. without coordination with other sites). The data is always available for update, but the data states at different sites may diverge. From time to time, the sites send their updates asynchronously to other sites with an anti-entropy protocol. To incorporate the updates made at the other sites, a site merges the received updates with its local replica. A CRDT has the property that when all sites have incorporated the same set of updates, the replicas converge.

There are two families of CRDT approaches, namely operation-based and state-based. For an operation-based CRDT [23], a message for an update is an encoding of the operation that made the corresponding update. A site that receives the message runs a special procedure to incorporate the update. To enforce convergence, the operations of an operation-based CRDT should commute, i.e. the executions of the same set of operations in different orders should have the same effect. A CRDT is *purely* operation-based if the encoding and incorporation of operations are trivial, in the sense that they are independent of the state at which the operation is performed [5]. Pure operation-based CRDTs require reliable causal delivery of messages.

For a state-based CRDT, as originally presented in [23], a message for updates is the data state of the replica in its entirety. The site that receives the message incorporates the updates by merging the received state and its local state. When the possible states of the data form a join-semilattice (see §4.1 below), the merge is the join of the two states. Convergence is implied by the join-semilattice.

$$
\begin{aligned}
\mathsf{GSet}(E) &\stackrel{\text{def}}{=} \mathcal{P}(E) \\
\mathsf{add}(e, s) &\stackrel{\text{def}}{=} \{e\} \cup s \\
\mathsf{add}^{\delta}(e, s) &\stackrel{\text{def}}{=} \begin{cases} \{e\} & \text{if } e \notin s \\ \bot & \text{otherwise} \end{cases} \\
s \sqcup s' &\stackrel{\text{def}}{=} s \cup s' \\
\mathsf{in}(e, s) &\stackrel{\text{def}}{=} e \in s
\end{aligned}
$$

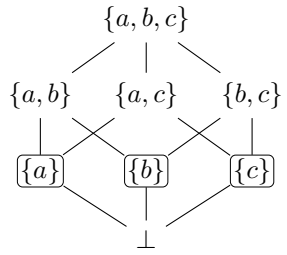

Figure 1 GSet CRDT and Hasse diagram of states

As our work focuses on state-based CRDTs, in the following subsections, we present the main theory underlying this family of CRDTs, including delta-state CRDTs [3], which improve the original state-based CRDTs. We also discuss a typical design of inverse operations in state-based CRDTs and why this is usually not sufficient as a mechanism of undo.

4.1 State-based CRDTs

A *state-based CRDT* is a tuple $\langle \mathcal{S}, \sqsubseteq, s^0, \mathcal{Q}, \mathcal{M}, \sqcup \rangle$, where $\mathcal{S}$ is a poset of states under partial order $\sqsubseteq$, $s^0 \in \mathcal{S}$ is an initial state, $\mathcal{Q}$ is a set of queries on the states, $\mathcal{M}$ is a set of mutators for performing updates on the states, and $\sqcup$ is a join operation on states. Furthermore, the state poset with $\sqcup$ is a join-semilattice. In this paper, we use the term *operation* as a particular instance of state update defined by a mutator. For example, $m \in \mathcal{M}$ is a mutator, whereas $m(s)$ is a state update, hence an operation. Consequently, for two different states s_1 and s_2, $m(s_1)$ and $m(s_2)$ are two different operations.

For a poset P under the partial order $\sqsubseteq$, a join operation $x \sqcup y$ returns the least upper bound of elements x and y in P. The join operation is idempotent, commutative and associative. The poset P is a *join-semilattice* iff $x \sqcup y$ exists for any x and y in P [13]. Some join-semilattices have a least element $\bot$, also known as the bottom element. A power set, under the partial order of set inclusion $\subseteq$ and with set union $\cup$ as join, is a classic example of a join-semilattice that has a bottom element $\bot = \varnothing$. For every CRDT discussed in this paper, we assume a bottom state $\bot$.

For a state-based CRDT, every state update is an inflation. That is, for any mutator $m \in \mathcal{M}$ and state $s \in \mathcal{S}$, $s \sqsubseteq m(s)$. When a local state s merges with a received remote state s', the new local state becomes $s \sqcup s'$. Because local updates are inflations and merges are the results of joins, at each site, state updates are monotonic under $\sqsubseteq$. In other words, every new state s_{n+1} subsumes a previous state s_n, i.e. $s_n \sqsubseteq s_{n+1}$ for any $n \geq 0$.

Figure 1 (left) shows GSet, a state-based CRDT for grow-only sets, where $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}(E)$ for a set E of possible elements, $\sqsubseteq \stackrel{\text{def}}{=} \subseteq$, $s^0 \stackrel{\text{def}}{=} \varnothing$, $\mathcal{Q} \stackrel{\text{def}}{=} \{\mathsf{in}\}$, $\mathcal{M} \stackrel{\text{def}}{=} \{\mathsf{add}\}$ and $\sqcup \stackrel{\text{def}}{=} \cup$. (The figure also shows a delta-mutator add^{δ} that will be explained in §4.2.) Obviously, an update through $\mathsf{add}(e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Figure 1 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states (known as the *cover* relation $\sqsubset_c$ [13]).

GSet is an example of an *anonymous* CRDT. A CRDT is anonymous if its operations are not specific to the sites that perform the operations and hence do not refer to site identifiers. Two sites can concurrently perform the operations defined by the same mutator. We say that these two sites perform *the same anonymous operations* concurrently. For example, when site A performs operation $\mathsf{add}(a, s_1)$ and site B concurrently performs operation $\mathsf{add}(a, s_2)$, the sites perform the same anonymous operation $\mathsf{add}(a)$.

$$\mathsf{GCounter} \stackrel{\text{def}}{=} \mathbb{I} \hookrightarrow \mathbb{N}$$

$$\mathsf{inc}_i(s) \stackrel{\text{def}}{=} s\{i \mapsto s(i)+1\}$$

$$\mathsf{inc}_i^\delta(s) \stackrel{\text{def}}{=} \{\langle i, s(i)+1\rangle\}$$

$$s \sqcup s' \stackrel{\text{def}}{=} \{\langle i, \max\big(s(i), s'(i)\big)\rangle \mid i \in \mathsf{dom}(s) \cup \mathsf{dom}(s')\}$$

$$\mathsf{value}(s) \stackrel{\text{def}}{=} \textstyle\sum_{i \in \mathsf{dom}(s)} s(i)$$

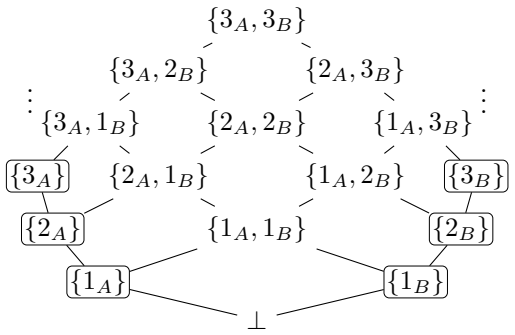

Figure 2 GCounter CRDT and Hasse diagram of states

On the other hand, a CRDT is *named* if a site can only update the part of the state that is specific to that site. Different sites cannot perform the same operation concurrently. Figure 2 (left) shows GCounter, a state-based CRDT for grow-only counters. It uses a partial function (or a key-value map) $\mathbb{I} \hookrightarrow \mathbb{N}$ to simulate a globally replicated counter. The sites update the key-value map similar to a version vector [18]. When site i increments the counter using operation inc_i, only the value mapped from the key i gets incremented. GCounter is named because operation inc_i is specific to site i and only site i can perform it. Figure 2 (right) shows the Hasse diagram of the states in a GCounter. In the figure, 2_A denotes the pair $\langle A, 2\rangle$, to expose the meaning "value 2 at site A".

4.2 Delta-state CRDTs

Using state-based CRDTs, as originally presented, is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue [3]. They are based on the concept of join-irreducible states.

An element x is *join-irreducible* in a poset P if it cannot be expressed as a join of other elements in P [13]. Formally, x is join-irreducible in P if $\forall y, z \in P : x = y \sqcup z \Rightarrow x = y \vee x = z$. We use $\mathcal{J}(P)$ for the set of join-irreducible elements of P.

For a finite join-semilattice, join-irreducible elements are those that have only one link below in the Hasse diagram [13]. In Figure 1 and Figure 2, the states in boxes are join-irreducible. The set of join-irreducible states of GSet, $\mathcal{J}(\mathcal{P}(E))$, consists of singleton sets. The set of join-irreducible states of GCounter, $\mathcal{J}(\mathbb{I} \to \mathbb{N})$ consists of singleton pair sets.

An important property of join-irreducible elements is that every element in a finite poset can be represented as a join of some join-irreducible elements. More precisely, given a finite poset P, for any $x \in P$, $x = \bigsqcup\{y \in \mathcal{J}(P) | y \sqsubseteq x\}$.

A delta-state CRDT has a delta-mutator m^δ for every mutator m of the corresponding state-based CRDT. Instead of returning the new updated state $m(s)$, m^δ returns a delta representation consisting only of join-irreducible states. The delta representation has the property $m(s) = s \sqcup m^\delta(s)$. For example, in Figure 1, add^δ is the delta counterpart of add. While $\mathsf{add}(e, s)$ returns the whole new state $\{e\} \cup s$, $\mathsf{add}^\delta(e, s)$ returns only a singleton set $\{e\}$ (when e was not in s and the mutation is effectively executed).

Now, instead of sending the whole state $m(s)$, a site only sends the delta representation $m^\delta(s)$, which is typically much smaller in size than $m(s)$. If a remote site has already incorporated s, a merge with $m^\delta(s)$ gives the same result as a merge with $m(s)$. We can thereby regard $m^\delta(s)$ as $m(s)$ where redundancy in s is eliminated.

Because the delta representation is not an inflation, the anti-entropy protocol must do

$$
\begin{aligned}
2\mathsf{P}_{\mathbb{B}}\mathsf{Set}(E) &\stackrel{\text{def}}{=} E \hookrightarrow \mathbb{B} \\
\mathsf{add}(e, s) &\stackrel{\text{def}}{=} \{\langle e, \mathsf{False}\rangle\} \sqcup s \\
\mathsf{add}^{\delta}(e, s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle e, \mathsf{False}\rangle\} & \text{if } e \notin \mathsf{dom}(s) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{remove}(e, s) &\stackrel{\text{def}}{=} \{\langle e, \mathsf{True}\rangle\} \sqcup s \\
\mathsf{remove}^{\delta}(e, s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle e, \mathsf{True}\rangle\} & \text{if } \langle e, \mathsf{True}\rangle \notin s \\ \bot & \text{otherwise} \end{cases} \\
s \sqcup s' &\stackrel{\text{def}}{=} \{\langle e, s(e) \vee s'(e)\rangle | e \in \mathsf{dom}(s) \cup \mathsf{dom}(s')\} \\
\mathsf{in}(e, s) &\stackrel{\text{def}}{=} e \in \mathsf{dom}(s) \wedge s(e) = \mathsf{False}
\end{aligned}
$$

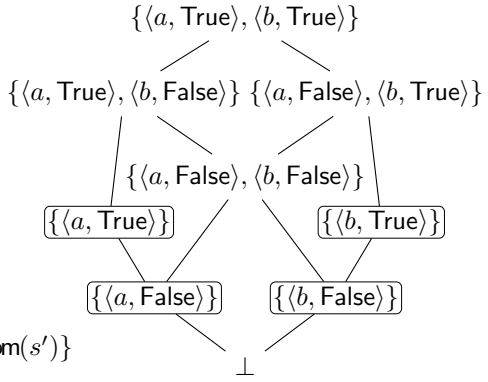

Figure 3 $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ CRDT and Hasse diagram of states

some extra work to achieve certain degree of causality [3]. Otherwise, the replicas will still eventually converge, but the sites may observe states out of causal order. In this paper, we focus on the design aspect of CRDTs and their undo support, and will not discuss the anti-entropy protocols.

4.3 Inverse operation as undo

Sometimes we may want to perform an inverse of an earlier update, for example, to remove an element that was earlier added into a set. Because updates in state-based CRDTs must be inflationary (§4.1), it is relatively easy to design CRDTs for those applications where the data grow in nature, such as grow-only set and grow-only counter. To support operations that make data shrink, such as the inverse operation of inflationary operations, we have to design new CRDTs using some special techniques. For example, we can keep the removed data as a kind of tombstones and let the queries achieve the shrinking effect. [23] and [3] presented different set CRDTs that have both add and remove operations.

Figure 3 (left) shows a set CRDT $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ (two-phase set using Boolean flags) that is a variation of u-set in [23] and two-phase set 2PSet in [3]. We associate every element added to the set with a Boolean flag indicating whether the element has been removed. More precisely, the states are a partial function $E \hookrightarrow \mathbb{B}$. We use pair $\langle e, \mathsf{False}\rangle$ when element e is added and $\langle e, \mathsf{True}\rangle$ when element e is removed. We adopt the conventional order of Boolean values $\mathsf{False} \sqsubset \mathsf{True}$. Hence, when an element is added and removed, the removal wins. (Note in the definitions of remove and $\sqcup$, $s(e) = \mathsf{False}$ when $e \notin \mathsf{dom}(s)$.) Figure 3 (right) shows the Hasse diagram of the states in $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$. For example, when state $\{\langle a, \mathsf{True}\rangle\}$ (i.e. element a has been removed) merges with state $\{\langle a, \mathsf{False}\rangle, \langle b, \mathsf{False}\rangle\}$ (i.e. both elements a and b are in the set), the new state is $\{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ (i.e. only element b is in the set).

Using operation remove as an inverse operation of add in $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ has a problem. The remove operation itself does not have an inverse operation. Once an element has been removed, it cannot be added back again. Actually, this problem is common among many CRDTs that provide some kind of inverse operations.

Causal CRDTs [3] such as OR-Set (observed-remove set [6], [16]) address this problem by associating state elements with causal contexts. A causal context is a set of event identifiers (typically a pair of a site identifier and a site-specific sequence number). Using causal contexts,

we are able to tell explicitly which additions of an element have been later removed. Because there is no upper bound on causal contexts, we can inverse any given (undo or redo) operation by inflation of associated causal contexts. However, maintaining causal contexts for every element can be costly, even though it is possible to compress causal contexts into vector states, especially under causal consistency. In our first contribution (§5), we work out an abstraction that allows us to use a single number as the smallest context without upper bound.

In general, inverse operations must be specially designed for the given operations and the design is normally not directly applicable to other operations or CRDTs. In our second contribution (§6), we present how to support undo in any state-based CRDT through a generic state transformation in the join semilattice space of the CRDT states.

5 Concurrent Undo and Redo Operations

This section presents our first main contribution. We formally characterize the concurrency and causality of undo and redo operations using equivalence classes. We can then represent the equivalence classes with single numbers called undo lengths. The abstraction presented in this section applies generally beyond the context of CRDTs.

5.1 Problem statement

The basic question is: when a site sees a set of undo and redo operations of an original normal operation *op*, should the site undo or redo *op*?

Example. Site S_1 inserts an element e into a set with operation add_1, undoes the addition with undo_1 and then redoes it with redo_1. Site S_2 receives add_1, undoes it with undo_2, and then receives and integrates undo_1 and redo_1. Is element e in the set at site S_2? The answer should be "yes", because the concurrent undo_1 and undo_2 operations have the same intention, and redo_1, whose intention is to redo the effect of add_1, supersedes both.

In a sequential system, such as a single-user editor, undo and redo of the same normal operation happen in turn. We could simply count the length of the undo-redo chain. If the length is an odd number, the original operation is undone, otherwise, it is redone. In the example, site S_1 alone is like a sequential system. The length of the undo-redo chain at site S_1 is two and the addition of e should be redone.

Undo in concurrent applications has been an active research topic for decades, particularly in the area of collaborative editing ([1, 10, 29, 21, 22, 24, 25, 27, 28]). However, most of the published work does not account for concurrent undo and redo operations correctly.

Some of the latest work also counted the number of undo and redo operations to decide whether an original operation is finally undone or redone, but the result is unsatisfactory.

The approach presented in [25] counts the number of times an operation has been undone or redone. If it is an odd number, the original operation in undone. In the example, the number is 3 at site S_2, so the addition operation add_1 is incorrectly undone.

The approach reported in [27] counts the numbers of undo and redo operations separately. The undo or redo with the higher number wins. In the example, there are two undos and one redo at site S_2. Therefore undo wins and add_1 is incorrectly undone.

The root problem with these earlier approaches is that they do not define the semantics undo and redo operations with respect to concurrency and causality of the operations. In the example, the two concurrent undo operations undo_1 and undo_2 are both meant to undo the same operation add_1. Therefore they should have the same effect as a single undo. On

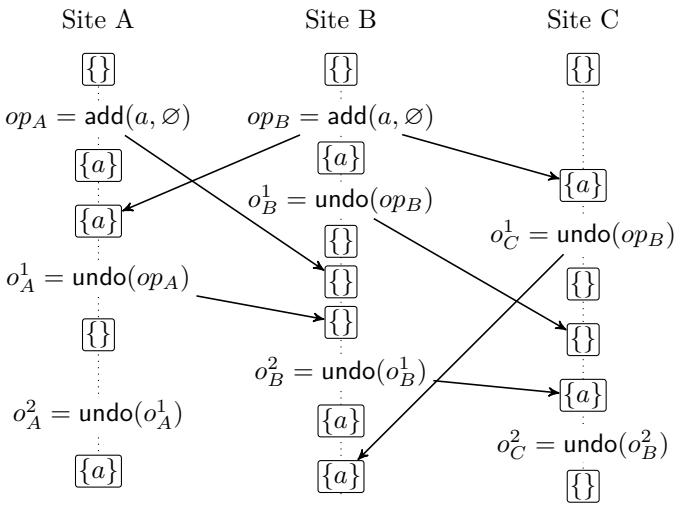

Figure 4 A scenario of concurrent undo operations

the other hand, redo_1 happens causally after undo_1 (which is effectively the same as undo_2) and hence should have the final effect at site S_2.

5.2 Capturing concurrency and causality of undo operations

An application performs operations to modify its data. For example, the add operation adds an element into a set. We call these *normal operations*. In a distributed system, different sites may perform the same normal operations concurrently (or more specifically, the same *anonymous* operations described in §4.1). In Figure 4, site A and site B perform the same $\mathsf{add}(a)$ operation concurrently.

When the application undoes a normal operation, it cancels the effect of the modification. It can even further undo the undo (to achieve a redo), etc. We use *op* for a normal operation and o for any operation, which can be either a normal operation or an undo operation. When the application applies an undo operation on an earlier performed operation o, denoted as $o' = \mathsf{undo}(o)$, we say that o' is an *undo operation* that *directly undoes* o. An application can only directly undo an operation when it has observed the effect of that operation.

In Figure 4, o_A^1 directly undoes op_A, o_B^1 and o_C^1 directly undo op_B, o_A^2 directly undoes o_A^1, o_B^2 directly undoes o_B^1, and o_C^2 directly undoes o_B^2.

We relate the (normal or undo) operations with the same intention through the tie relation. An operation o_1 *ties* with operation o_2, denoted as $o_1 \sim o_2$, if one of the following holds: (i) $o_1 = o_2$, (ii) o_1 and o_2 are the same normal (anonymous) operations, (iii) $o_1 = \mathsf{undo}(o)$ and $o_2 = \mathsf{undo}(o)$, (iv) $o_1 = \mathsf{undo}(o_1')$, $o_2 = \mathsf{undo}(o_2')$ and $o_1' \sim o_2'$.

In Figure 4, $op_A \sim op_B$ because they are the same normal operations $\mathsf{add}(a)$, $o_B^1 \sim o_C^1$ because both directly undo the same operation op_B; $o_A^1 \sim o_B^1$ because $op_A \sim op_B$; $o_A^2 \sim o_B^2$ because $o_A^1 \sim o_B^1$.

Lemma ($\sim$ properties) The $\sim$ relation is reflexive, symmetric and transitive.

Consequently, the tie relation partitions the operations into equivalence groups. For example, the equivalence groups in Figure 4 are $\{op_A, op_B\}$, $\{o_A^1, o_B^1, o_C^1\}$, $\{o_A^2, o_B^2\}$ and $\{o_C^2\}$.

One requirement on handling concurrent normal or undo operations is that the application should observe the same effect of tied operations.

The tie relation $\sim$ captures the concurrency of undo operations. The following undo-supersede relation captures the causality of undo operations. An operation o *undo-supersedes* operation o', denoted as $o \succ_{\text{u}} o'$, if one of the following holds: (i) $o = \mathsf{undo}(o')$, (ii) $o = \mathsf{undo}(o'')$ and $o'' \sim o'$, (iii) $o = \mathsf{undo}(o'')$ and $o'' \succ_{\text{u}} o'$.

In Figure 4, $o_A^2 \succ_{\text{u}} o_A^1$ because $o_A^2 = \mathsf{undo}(o_A^1)$; $o_A^2 \succ_{\text{u}} o_B^1$ because $o_A^2 = \mathsf{undo}(o_A^1)$ and $o_A^1 \sim o_B^1$; $o_C^2 \succ_{\text{u}} o_B^1$ because $o_C^2 = \mathsf{undo}(o_B^2)$ and $o_B^2 \succ_{\text{u}} o_B^1$.

Lemma ($\succ_{\text{u}}$ properties) The $\succ_{\text{u}}$ relation is irreflexive, asymmetric and transitive.

For an operation o, its *original operation*, denoted as $\mathsf{orig}(o)$, is a normal operation op, such that either (i) $o = op$, or (ii) $o = \mathsf{undo}(op)$, or (iii) $o = \mathsf{undo}(o')$ and $\mathsf{orig}(o') = op$.

In Figure 4, $\mathsf{orig}(op_A) = \mathsf{orig}(o_A^1) = \mathsf{orig}(o_A^2) = op_A$, and $\mathsf{orig}(op_B) = \mathsf{orig}(o_B^1) = \mathsf{orig}(o_B^2) = \mathsf{orig}(o_C^1) = \mathsf{orig}(o_C^2) = op_B$.

Two operations o_1 and o_2 have the *same origin*, denoted as $o_1 \overset{\text{orig}}{=} o_2$, if either $\mathsf{orig}(o_1) = \mathsf{orig}(o2)$ or $\mathsf{orig}(o_1) \sim \mathsf{orig}(o2)$.

In Figure 4, $o_A^1 \overset{\text{orig}}{=} o_A^2$ because $\mathsf{orig}(o_A^1) = \mathsf{orig}(o_A^2)$; $o_A^1 \overset{\text{orig}}{=} o_C^2$ because $\mathsf{orig}(o_A^1) \sim \mathsf{orig}(o_C^2)$.

Lemma (origin and undo relations) Undo operations have the same origin iff they are related with tie or undo-supersede relations. Formally, $o_1 \overset{\text{orig}}{=} o_2 \Leftrightarrow o_1 \sim o_2 \vee o_1 \succ_{\text{u}} o_2 \vee o_2 \succ_{\text{u}} o_1$.

For two concurrent undo operations o_1 and o_2 that have the same origin, a merge of o_1 and o_2, $\mathsf{merge}(o_1, o_2)$, should result in either (i) o_1 or o_2 if $o_1 \sim o_2$ (which one does not matter), (ii) o_1 if $o_1 \succ_{\text{u}} o_2$, or (iii) o_2 if $o_2 \succ_{\text{u}} o_1$.

When a site merges two concurrent operations, if the two operations tie with each other, they should have the same effect and the result of the merge can be either of them. In Figure 4, $o_A^1 \sim o_B^1$, hence $\mathsf{merge}(o_A^1, o_B^1) = o_A^1$ (or equally o_B^1). If one operation undo-supersedes the other, $o_1 \succ_{\text{u}} o_2$, o_1 has already seen the effect of o_2 and is causally dependent on o_2. Therefore the result of the merge is o_1. In Figure 4, $o_B^2 \succ_{\text{u}} o_C^1$ and o_B^2 has seen the effect of o_C^1 (which is equivalent to the effect of o_B^1 because $o_B^1 \sim o_C^1$), hence $\mathsf{merge}(o_C^1, o_B^2) = o_B^2$.

Now we define the *undo length* of an operation o as:

$$\mathsf{ulen}(o) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } o \text{ is a normal operation} \\ \mathsf{ulen}(o') + 1 & \text{if } o = \mathsf{undo}(o') \end{cases}$$

In Figure 4, $\mathsf{ulen}(op_A) = \mathsf{ulen}(op_B) = 0$, $\mathsf{ulen}(o_A^1) = \mathsf{ulen}(o_B^1) = \mathsf{ulen}(o_C^1) = 1$, $\mathsf{ulen}(o_A^2) = \mathsf{ulen}(o_B^2) = 2$, and $\mathsf{ulen}(o_C^2) = 3$.

Lemma (undo length) Let $o_1 \overset{\text{orig}}{=} o_2$. $\mathsf{ulen}(o_1) = \mathsf{ulen}(o_2)$ iff $o_1 \sim o_2$; $\mathsf{ulen}(o_1) > \mathsf{ulen}(o_2)$ iff $o_1 \succ_{\text{u}} o_2$.

Lemma (undo merge) Let $o_1 \overset{\text{orig}}{=} o_2$ and $o = \mathsf{merge}(o_1, o_2)$. The undo length of o is $\mathsf{ulen}(o) = \mathsf{max}\big(\mathsf{ulen}(o_1), \mathsf{ulen}(o_2)\big)$.

We could name the equivalence groups under $\sim$ in such a way that G_{op}^0 contains original normal operations and every operation in G_{op}^{n+1} directly undoes an operation in G_{op}^n. Then for any operation $o \in G_{op}^n$, $\mathsf{ulen}(o) = n$. For example, in Figure 4, $G_{\mathsf{add}(a)}^0 = \{op_A, op_B\}$, $G_{\mathsf{add}(a)}^1 = \{o_A^1, o_B^1, o_C^1\}$, $G_{\mathsf{add}(a)}^2 = \{o_A^2, o_B^2\}$ and $G_{\mathsf{add}(a)}^3 = \{o_C^2\}$.

In applications like editors, people often use the terms undo or redo with respect to the original normal operations. When $\mathsf{orig}(o) \sim op$, we say that

- o *undoes* op if either (i) $o = \mathsf{undo}(op)$, or (ii) $o = \mathsf{undo}(\mathsf{undo}(o'))$ and o' undoes op;
- o *redoes* op if $o = \mathsf{undo}(o')$ and o' undoes op.

In Figure 4, o_A^1, o_B^1, o_C^1 and o_C^2 undo $\mathsf{add}(a)$ (either op_A or op_B), whereas o_A^2 and o_B^2 redo $\mathsf{add}(a)$.

Obviously, if o undoes op, then $\mathsf{undo}(o)$ redoes op. Similarly, if o redoes op, then $\mathsf{undo}(o)$ undoes op.

Theorem (undo-redo) Given $\mathsf{orig}(o) \sim op$,

- o undoes op iff $\mathsf{ulen}(o)$ is a positive odd number;
- o redoes op iff $\mathsf{ulen}(o)$ is a positive even number.

We can use the *undo-redo* theorem to answer the question in §5.1.

We omit the proofs of the lemmas and theorem in this section as they are trivial, simply by permutation on the different cases or by induction on undo lengths.

An application at a site always behaves according to the observation of its latest local state. An undo operation o is a *latest undo* of a normal operation op at a site, if $\mathsf{orig}(o) \sim op$ and there does not exist o' at the site such that $o' \succ_{\mathrm{u}} o$.

Locally, an application can only generate a normal operation, directly undo a normal operation if it has not been undone at the site, or directly undo a latest undo operation at the site. In Figure 4, when site B has received o_A^1, the latest undo operations of op_A (or equally op_B) are o_A^1 and o_B^1. Thus site B can only directly undo o_A^1 or o_B^1. It does not matter which of them to undo, because $o_A^1 \sim o_B^1$.

To incorporate the effect of a remote undo operation o, a site merges o with a latest operation o_l that has the same origin with o. If the remote operation o undo-supersedes the local operation o_l, the result of the merge is o and the site incorporates the effects of o; otherwise the result is o_l that the site has already incorporated.

6 Generically Supporting Undo for CRDTs

This section presents our second main contribution, our approach to generically supporting undo for existing CRDTs using the abstraction presented earlier in §5.

6.1 State Deltas as Operations

Every state in a state-based CRDT can be generated from a set of join-irreducible states (see §4.2). In Figures 1–3, states in boxes are join-irreducible. Given a mutator m, the states before and after applying m are s and $m(s)$. Let J_s and $J_{m(s)}$ be the sets of join-irreducible states that generate s and $m(s)$. The *state delta* caused by the execution of m on s is the set of join-irreducible states $J_{m(s)} - J_s$. For example, for the GSet CRDT (Figure 1), the state delta of the operation $\mathsf{add}(e, s)$ is $(s \cup \{e\}) - s = \{e\}$ when $e \notin s$. When e is already in s, the state delta is an empty set and no operation is actually executed.

It is a common and intuitive practice that a state-based CRDT is designed in such a way that every state delta consists of a single join-irreducible state. Or in the case of delta-state CRDTs, every delta-mutator returns a single join-irreducible state. For example, the state delta of $\mathsf{add}(e, s)$ of GSet is $\{e\}$ and the state delta of $\mathsf{inc}_i(s)$ of GCounter is $\{\langle i, s(i) + 1\rangle\}$. We observe that all delta-state CRDTs presented in [3] show this property. With such design, we can use join-irreducible states to represent operations of the CRDT.

In this paper, we assume that the state delta of a normal operation op consists of a single join-irreducible state, written as $\downarrow^{\delta} op$. Due to space limit, we do not deal with composite operations consisting of multiple join-irreducible states.

6.2 Undo-State CRDT

We maintain the undo states of operations as meta-data using the undo-state CRDT UState (Figure 5). For an existing CRDT with possible join-irreducible states $\mathcal{S}$, the undo state is a partial function $u \colon \mathcal{S} \hookrightarrow \mathbb{N}$. For a normal operation op of that CRDT, whose state delta is the join-irreducible state $s = \downarrow^{\delta} op$, $s \in \mathsf{dom}(u)$ means the operation op has been performed

$$
\begin{aligned}
\mathsf{UState}(\mathcal{S}) &\stackrel{\text{def}}{=} S \hookrightarrow \mathbb{N} \\
\mathsf{reg}_u(s) &\stackrel{\text{def}}{=} \begin{cases} u\{s \mapsto u(s)+1\} & \text{if } s \in \mathsf{dom}(u) \\ u\{s \mapsto 0\} & \text{otherwise} \end{cases} \\
\mathsf{reg}^{\delta}_u(s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle s, u(s)+1\rangle\} & \text{if } s \in \mathsf{dom}(u) \\ \{\langle s, 0\rangle\} & \text{otherwise} \end{cases} \\
u \sqcup u' &\stackrel{\text{def}}{=} \{\langle s, \mathsf{max}\big(u(s), u'(s)\big)\rangle \mid s \in \mathsf{dom}(u) \cup \mathsf{dom}(u')\} \\
\mathsf{undone}_u(s) &\stackrel{\text{def}}{=} \mathsf{odd}(u(s)) \\
\mathsf{valid}_u(s) &\stackrel{\text{def}}{=} s \in \mathsf{dom}(u) \wedge \mathsf{even}(u(s)) \\
\mathsf{valid}^{+}_{\mathsf{u}}(s) &\stackrel{\text{def}}{=} \bigwedge_{x \in \mathsf{dom}(u), x \sqsubseteq s} \mathsf{valid}_u(x)
\end{aligned}
$$

Figure 5 CRDT for undo states

and $u(s)$ is the undo length of a latest undo operation of *op* (see §5.2 for the respective definitions). Notice that the bottom of $\mathbb{N}$, $\bot_{\mathbb{N}} = 0$. If an operation *op* has not been performed and thus $\downarrow^{\delta}\!op \notin \mathsf{dom}(u)$, applying $u(\downarrow^{\delta}\!op)$ (for example, when performing a join or a query), the result is 0 (§3).

For a normal operation *op* of the existing CRDT, $\downarrow^{\delta}\!op = s$, the operation $\mathsf{reg}_u(s)$ of UState registers the new latest undo state of *op*. The normal operation itself is registered with the addition of a new pair $\langle s, 0\rangle$ into the undo state. A new direct undo of a latest undo operation of *op* is registered with an incremental of $u(s)$ with one.

A join $\sqcup$ of two undo states u and u' merges the undo lengths of all operations that are registered in either u or u' (according to Lemma *undo merge* in §5.2).

Notice that UState is an anonymous CRDT. To see how this works, remember that we can partition the set of operations into equivalence groups under the tie relation $\sim$ (§5.2). Imagine that we register a new undo operation by adding it into the corresponding equivalence group. We can have an anonymous CRDT for the equivalence groups because they are grow-only sets. Using undo lengths in place of equivalence groups is just a way of compressing the undo states. The compression is possible because we are only interested in whether an equivalence group exists, rather than the specific elements in the groups. In addition, a site can only add an element in a new empty group, because it can only directly undo the latest undo operation of that site.

In Figure 4, after site C has incorporated received operation o^2_B, it sees the operations in equivalence groups $G^0_{\mathsf{add}(a)} = \{op_B\}$, $G^1_{\mathsf{add}(a)} = \{o^1_B, o^1_C\}$ and $G^2_{\mathsf{add}(a)} = \{o^2_B\}$. When performing o^2_C, it creates an empty group $G^3_{\mathsf{add}(a)}$ and adds o^2_C into it. Thereby $u(\{a\})$ in the UState becomes 3.

Another way to look at the undo state is to regard it as a log of the operations that have been performed. For every operation in the log, the recorded information is compressed into a single number, the undo length.

The predicate $\mathsf{undone}_u(s)$ states that the normal operation whose state delta is s is currently undone (according to Theorem *undo-redo* in §5.2).

The predicate $\mathsf{valid}_u(s)$ states that a state s in the existing CRDT is valid in u (i.e. $\mathsf{valid}_u(s)$ evaluates to True) if the corresponding normal operation has been performed (i.e.

$s \in \mathsf{dom}(u)$), and the operation either has not been undone (i.e. $u(s) = 0$), or it has been undone but is finally redone (i.e. $u(s) > 0 \wedge \mathsf{even}(u(s))$.

The predicate valid_u^+ takes the dependencies of join-irreducible states into account. When a join-irreducible state becomes invalid due to undo (i.e. valid_u evaluates to False), all states depending on it also become invalid (i.e. valid_u^+ evaluates to False). For example, the state 3_A of $\mathsf{Gcounter}$ (Figure 2) depends on state 2_A. $\mathsf{valid}_u^+(3_A) = \mathsf{False}$ when $\mathsf{valid}_u(\{2_A\}) = \mathsf{False}$.

Notice that the states in GSet form an anti-chain. That is, every join-irreducible state is independent of any other join-irreducible state. For such CRDTs, valid_u^+ gives the same result as valid_u.

To compute the predicate valid_u^+, we need to find out the dependencies among join-irreducible states, using the links in the Hasse diagrams (i.e. the cover relation $\sqsubset_c$). For some CRDTs, the dependencies can be derived. For example, for $\mathsf{GCounter}$, $n_i \sqsubset_c (n+1)_i$ where $n \geq 0$. In case the dependencies cannot be derived, we have to materialize the dependencies, for instance, using a list or tree data structure.

6.3 Augmenting Existing CRDTs with Undo

For an existing CRDT T with possible states in $\mathcal{S}_T$, the CRDT augmented with undo support T_U is a composition of $\mathcal{S}_T$ and $\mathsf{UState}(\mathcal{J}(\mathcal{S}_T))$. Figure 6 shows the T_U CRDT.

The operation $\mathsf{do}_{\langle s,u\rangle}(op)$ performs a normal operation op in state s and registers op in undo state u. The operation $\mathsf{undo\_latest}_{\langle s,u\rangle}(op)$ directly undoes the latest undo operation of op in state s: it registers the new latest undo in undo state u and has *no effect* on s. A site can only perform an undo when the original normal operation op has been performed or incorporated (i.e. the state delta $\downarrow^{\delta} op$ is registered in u). Otherwise, performing an undo has no effect on undo state u.

To join two augmented states $\langle s, u\rangle$ and $\langle s', u'\rangle$, we join independently the states s and s' in $\mathcal{S}_T$ and the states u and u' in $\mathsf{UState}(\mathcal{J}(\mathcal{S}_T))$.

Queries in the original CRDT T are now performed on states transformed from augmented states. $\nu_u(s)$ defines a transformation that transforms a state s in the original CRDT using the undo state u. To see how it works, remember that the following holds for every state s in $\mathcal{S}_T$ (§4.2):

$$s = \bigsqcup \{x \in \mathcal{J}(\mathcal{S}_T) | x \sqsubseteq s\}$$

The transformation ν_u first filters out the invalid join-irreducible states and then joins the valid join-irreducible states to bring back the up-to-date state that reflects the undone effects.

The state transformation can be very costly if applied for every query. To address this, every site maintains a buffer of the transformed state. Every time the site updates the undo state, it also updates the buffered state. For example, when state $\{a\}$ of GSet becomes invalid, we remove element a from the buffered state. Indeed, the buffered states do not form a join-semilattice. This, however, does not lead to inconsistencies, because the buffered states are only local to the sites and are not propagated to remote sites.

Now we use some examples to illustrate how the augmentation works.

We first augment GSet (Figure 1) to GSet_U for undo support. $\mathsf{add}_u(e, s)$ performs $\mathsf{do}_{\langle \mathsf{s,u}\rangle}(\mathsf{add(e)})$. The query in_u in GSet_U is equivalent to the following:

$$\mathsf{in}_u(e, s) \stackrel{\text{def}}{=} e \in s \wedge \neg \mathsf{undone}_u(\{e\})$$

The query now takes the undo effect into account. In Figure 4, the latest undo operation of $\mathsf{add}(a)$ at site B is o_B^2. Because $\mathsf{ulen}(o_B^2) = 2$, the undo state at site B is $\{\langle\{a\}, 2\rangle\}$.

$$
\begin{aligned}
\mathsf{T}_U &\stackrel{\text{def}}{=} \mathcal{S}_T \times \mathsf{UState}(\mathcal{J}(\mathcal{S}_T)) \\
\mathsf{do}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \langle s \sqcup \downarrow^{\delta} op, \mathsf{reg}_u(\downarrow^{\delta} op)\rangle \\
\mathsf{do}^{\delta}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \langle \downarrow^{\delta} op, \mathsf{reg}^{\delta}_u(\downarrow^{\delta} op)\rangle \\
\mathsf{undo\_latest}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \begin{cases} \langle s, \mathsf{reg}_u(\downarrow^{\delta} op)\rangle & \text{if } \downarrow^{\delta} op \in \mathsf{dom}(u) \\ \langle s, u\rangle & \text{otherwise} \end{cases} \\
\mathsf{undo\_latest}^{\delta}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \begin{cases} \langle \bot, \mathsf{reg}^{\delta}_u(\downarrow^{\delta} op)\rangle & \text{if } \downarrow^{\delta} op \in \mathsf{dom}(u) \\ \langle \bot, \bot\rangle & \text{otherwise} \end{cases} \\
\langle s, u\rangle \sqcup \langle s', u'\rangle &\stackrel{\text{def}}{=} \langle s \sqcup s', u \sqcup u'\rangle \\
\nu_u(s) &\stackrel{\text{def}}{=} \bigsqcup \{x \in \mathcal{J}(\mathcal{S}_T) | x \sqsubseteq s \wedge \mathsf{valid}^{+}_u(s)\} \\
\mathsf{query}_u(\ldots, s, \ldots) &\stackrel{\text{def}}{=} \mathsf{query}(\ldots, \nu_u(s), \ldots)
\end{aligned}
$$

Figure 6 CRDT augmented with undo

Therefore the join-irreducible state $\{a\}$ is valid and $\mathsf{in}_u(a, \{a\})$ evaluates to True. On the other hand, the latest undo operation of $\mathsf{add}(a)$ at site C is o^2_C. Because $\mathsf{ulen}(o^2_C) = 3$, the undo state at site C is $\{\langle\{a\}, 3\rangle\}$. Therefore the join-irreducible state $\{a\}$ is invalid and the query $\mathsf{in}_u(a, \{a\})$ evaluates to False.

Now, let us augment $\mathsf{2P_{\mathbb{B}}Set}$ (Figure 3) with undo support. In the first scenario, a site adds a, b, removes a and then undoes the removal. The state in $\mathsf{2P_{\mathbb{B}}Set}$ is $s^1 = \{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ and the undo state is $u^1 = \{\langle\{\langle a, \mathsf{False}\rangle\}, 0\rangle, \langle\{\langle a, \mathsf{True}\rangle\}, 1\rangle, \langle\{\langle b, \mathsf{False}\rangle\}, 0\rangle\}$. The predicate $\mathsf{valid}_{u^1}(\{\langle a, \mathsf{True}\rangle\}) = \mathsf{False}$. Transforming the state results in $s^1_u = \nu_{u^1}(s^1) = \{\langle a, \mathsf{False}\rangle\} \cup \{\langle b, \mathsf{False}\rangle\} = \{\langle a, \mathsf{False}\rangle, \langle b, \mathsf{False}\rangle\}$. The results of queries on s^1_u are as expected, $\mathsf{in}(a, s^1_u) = \mathsf{in}(b, s^1_u) = \mathsf{True}$. That is, both a and b are in the set.

In the second scenario, a site adds a, b, removes a and then undoes the addition of a. The state in $\mathsf{2P_{\mathbb{B}}Set}$ is $s^2 = \{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ and the undo state is $u^2 = \{\langle\{\langle a, \mathsf{False}\rangle\}, 1\rangle, \langle\{\langle a, \mathsf{True}\rangle\}, 0\rangle, \langle\{\langle b, \mathsf{False}\rangle\}, 0\rangle\}$. Observe that $s^2 = s^1$ and $u^2 \neq u^1$, meaning that an undo does not alter the state of the original CRDT. The predicates $\mathsf{valid}_{u^2}(\{\langle a, \mathsf{False}\rangle\}) = \mathsf{False}$ and $\mathsf{valid}^{+}_{u^2}(\{\langle a, \mathsf{True}\rangle\}) = \mathsf{False}$. Transforming the state results in $s^2_u = \nu_{u^2}(s^2) = \{\langle b, \mathsf{False}\rangle\}$. Again, the results of queries on s^2_u are as expected, $\mathsf{in}(a, s^2_u) = \mathsf{False}$ and $\mathsf{in}(b, s^1_u) = \mathsf{True}$. That is, b is in the set but a is not (as if a had never been added).

The last examples with $\mathsf{2P_{\mathbb{B}}Set}$ indicate that the generic undo support works well with CRDTs that themselves support inverse operations.

7 Collaborative Editing with Undo Support

In this section, we show a practical application of the undo support for collaborative editing.

The collaborative editing system is based on the CRDT reported in [29]. It consists of several peers, each of which has a replica of the shared document under editing. At each peer, a user edits the local copy of the document via the *document view*, which is simply a string of characters. Under the hood, there is a *document model*, which is a CRDT of characters. The view is the concatenation of *visible* characters in the model. We could regard the view

$$
\begin{aligned}
\mathsf{Doc}(C) &\stackrel{\text{def}}{=} C \hookrightarrow \mathcal{P}(\mathbb{I}) \\
\mathsf{ins}(c, m) &\stackrel{\text{def}}{=} m\{c \mapsto \varnothing\} \\
\mathsf{ins}^{\delta}(c, m) &\stackrel{\text{def}}{=} \{\langle c, \varnothing\rangle\} \\
\mathsf{del}_i(c, m) &\stackrel{\text{def}}{=} \begin{cases} m\{c \mapsto m(c) \cup \{i\})\} & \text{if } c \in \mathsf{dom}(m) \\ m & \text{otherwise} \end{cases} \\
\mathsf{del}_i^{\delta}(c, m) &\stackrel{\text{def}}{=} \begin{cases} \{\langle c, \{i\}\rangle\} & \text{if } c \in \mathsf{dom}(m) \\ \bot & \text{otherwise} \end{cases} \\
m \sqcup m' &\stackrel{\text{def}}{=} \{\langle c, m(c) \cup m'(c)\rangle \mid c \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\} \\
\mathsf{visible}(c, m) &\stackrel{\text{def}}{=} c \in \mathsf{dom}(m) \wedge m(c) = \varnothing
\end{aligned}
$$

Figure 7 CRDT for a collaborative text editor

as the buffer of transformed state discussed in §6.3.

Figure 7 shows the (simplified) CRDT of the document model. The CRDT is a function from the set of characters C to the power set of site identifiers $\mathbb{I}$.

The characters of the CRDT have globally unique and ordered identifiers that are specific to the sites that inserted the character ([26, 4]). Therefore the Doc CRDT is named—every character is unique and cannot be concurrently inserted at different peers. However, different peers can concurrently delete the same character.

When a character c is inserted, c maps to an empty set. When site i deletes c, i is added to the set that c maps to. A character c is visible in the document if it is inserted but not deleted, that is, when c is in the domain and maps to the empty set.

To support undo, we simply augment Doc to Doc_U. The designer of the Doc CRDT does not need to manually design anything in addition. In the augmented CRDT, $\mathsf{visible}_u$ is equivalent to the following:

$$
\mathsf{visible}_u(c, m) \stackrel{\text{def}}{=} \mathsf{valid}_u(\{\langle c, \varnothing\rangle\}) \wedge \bigwedge_{i \in m(c)} \neg\mathsf{valid}_u(\{\langle c, \{i\}\rangle\})
$$

A character c is visible in the document if it is inserted and the insertion is not undone, and if it is deleted, all deletions are undone.

To see why a character should be visible only when all deletions are undone, consider the situation where site A deletes a character "x" and then undoes the deletion. Meanwhile, site B also deletes "x". The final effect should be as if site A had done nothing and site B performed a deletion. So character "x" should not appear in the document.

Some researchers (for example [22]) regard concurrent deletions of the same character as the same operation (which may lead to some confusing semantics of the undo of string-wise operations [29]). We could achieve this by using partial function $C \hookrightarrow \mathbb{B}$ (similar to the $\mathsf{2P}_{\mathbb{B}}\mathsf{Set}$ CRDT in Figure 3) rather than $C \hookrightarrow \mathcal{P}(\mathbb{I})$. With this re-design, a character is visible when only one deletion is undone (because all deletions of the same character are regarded as the same anonymous operation).

8 Related Work

Supporting inverse operations was already a topic when CRDTs were first presented [23], such as a counter that can be both incremented and decremented, a set where elements can be both added and removed, etc. The CRDT designer has to design new customized CRDTs in order to support inverse operations. A common problem is that the designer has to decide a "winner" between an operation and its inverse counterpart, for example, a removal always wins (see §4.3 for an example). Furthermore, a "loser" has never got a chance to "win back".

A causal CRDT [3] associates causal contexts with every operation (or element) to achieve the effect such as adding a removed element back to a set. The CRDT designer has to write a new causal CRDT for a given CRDT to get this support. Furthermore, maintaining causal contexts for every operation could be costly when the number of replicas is large.

Our work provides a generic support of undo for any (to our knowledge) state-based CRDT. The CRDT designer does not need to write a new specialize CRDT to get the undo feature. Furthermore, the undo state for an operation is only a single number.

Undo has been a research topic in the area of collaborative editing for decades ([10, 21, 22, 24, 25, 27]). Most of the work was not able to define the semantics of undo and redo operations with respect to concurrency and causality, and therefore showed incorrect behavior as discussed in §5.1.

The abstraction we proposed, although seemingly simple, correctly captures the semantics of concurrent undo an redo operations and does not have the aforementioned issues.

The Doc CRDT (§7) is a simplification of the work presented in [29]. The model CRDT in [29] represents undo relations using equivalence classes (§5.2) rather than the more compact undo lengths. This allows the editor to support additional features such as displaying who performed a particular undo operation.

The system presented in [9] supports cascading undo of selected operations by explicitly defining dependencies among operations using a process specification language. In our work, operation dependencies are implied by the state order $\sqsubset$ of join-semilattices.

9 Conclusion

In this work, we have presented how to provide undo features to existing CRDTs. Our work consists of two major parts.

The first part is an abstraction that captures the semantics concurrent undo and redo operations using equivalence classes. The abstraction can be compacted into single numbers (undo lengths) that are straightforward to implement in practice. The abstraction is generally applicable (not restricted to CRDTs) to any system that demands concurrent undo and redo of earlier performed operations.

The second part is a generic approach to augmenting existing state-based CRDTs with the capability of undo. The augmentation transforms the states in an original CRDT to the ones with the undo effects. Unmodified queries can be applied to the transformed states. The states of the augmented CRDTs converge eventually, because the state transformation is local to the replicas and does not propagate to the global system.

We have shown a practical application of our work in collaborative editing.

Operation-based CRDTs have also found their ways in applications that demand undo support. Supporting undo features for operation-based CRDTs is an open research topic.

References

1 Gregory D Abowd and Alan J Dix. Giving undo attention. *Interacting with Computers*, 4(3):317 – 342, 1992.

2 Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *36th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 405–414, 2016.

3 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.

4 Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *CollaborateCom*, 2013.

5 Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *14th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, volume 8460 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2014.

6 Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. A optimized conflict-free replicated set. *Rapport de recherche*, 8083, October 2012.

7 Eric A. Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2):23–29, 2012.

8 Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak DT map: a composable, convergent replicated dictionary. In *The First Workshop on the Principles and Practice of Eventual Consistency*, pages 1–1, 2014.

9 Aaron G. Cass and Chris S. T. Fernandes. Using task models for cascading selective undo. In Karin Coninx, Kris Luyten, and Kevin A. Schneider, editors, *Task Models and Diagrams for Users Interface Design*, pages 186–201. Springer Berlin Heidelberg, 2007.

10 Jean Ferrié, Nicolas Vidot, and Michèle Cart. Concurrent undo operations in collaborative environments using operational transformation. In *CoopIS/DOA/ODBASE (1)*, volume 3290 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2004.

11 Armando Fox and Eric A. Brewer. Harvest, yield and scalable tolerant systems. In *The Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

12 Hector Garcia-Molina and Kenneth Salem. Sagas. In *1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM, 1987.

13 Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

14 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

15 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, 2012.

16 Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, coordination-free programming. In *the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195, 2015.

17 Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *CSCW*, pages 259–268. ACM, 2006.

18 Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

19 Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops (SRDS Workshops 2014)*, pages 30–33, 2014.

20 Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 395–403, 2009.

21 Matthias Ressel and Rul Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139. ACM, 1999.

22 Bin Shao, Du Li, and Ning Gu. An algorithm for selective undo of any operation in collaborative applications. In *GROUP*, pages 131–140. ACM, 2010.

23 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)*, pages 386–400, 2011.

24 Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, 2002.

25 David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, 2009.

26 Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *29th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 404–412, 2009.

27 Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. Parallel Distrib. Syst.*, 21(8):1162–1174, 2010.

28 Weihai Yu. Supporting string-wise operations and selective undo for peer-to-peer group editing. In *GROUP*, pages 226–237. ACM, 2014.

29 Weihai Yu, Luc André, and Claudia-Lavinia Ignat. A CRDT supporting selective undo for collaborative text editing. In *15th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, volume 9038 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2015.