Distributed Computing (2022) 35:145–164
https://doi.org/10.1007/s00446-021-00414-6

Strong eventual consistency of the collaborative editing framework WOOT

Emin Karayel[1,2] · Edgar Gonzàlez[1]

Received: 3 July 2020 / Accepted: 4 November 2021 / Published online: 4 January 2022

Abstract
Commutative Replicated Data Types (CRDTs) are a promising new class of data structures for large-scale shared mutable content in applications that only require eventual consistency. The WithOut Operational Transforms (WOOT) framework is the first CRDT for collaborative text editing introduced by Oster et al. (In: Conference on Computer Supported Cooperative Work (CSCW). ACM, New York, pp 259–268, 2006a). Its eventual consistency property was verified only for a bounded model to date. While the consistency of many other previously published CRDTs had been shown immediately with their publication, the property for WOOT remained open for 14 years. We use a novel approach identifying a previously unknown sort-key based protocol that simulates the WOOT framework to show its consistency. We formalize the proof using the Isabelle/HOL proof assistant to machine-check its correctness.

Keywords Distributed Systems · CRDT · Eventual Consistency · WOOT · Formal Verification

1 Introduction

A Replicated (Abstract) Data Type (RDT) consists of *"multiple copies of a shared Abstract Data Type (ADT) replicated over distributed sites, [which] provides a set of primitive operation types corresponding to that of normal ADTs, concealing details for consistency maintenance"* [25]. RDTs can be classified as state-based or operation-based depending on whether full states (e.g., a document's text) or only the operations performed on them (e.g., character insertions and deletions) are exchanged among replicas. Operation-based RDTs are commutative when the integration of any two concurrent operations on any reachable replica state commutes [27]. Commutative (Operation-Based) Replicated Data Types (CRDTs[1] from now on) enable sharing mutable content with optimistic replication—ensuring high-availability, responsive interaction, and eventual consistency in an asynchronous network without consensus-based concurrency control [14]. They are used in highly scalable robust distributed applications [4,31]. An RDT (and, in particular a CRDT) is eventually consistent when, if after some point in time no further updates are made at any replica, all replicas eventually converge to equal states. It is strongly eventually consistent when it is both eventually consistent and strongly convergent, i.e., any pair of peers which have integrated the same set of updates (in possibly different order) are in the same state [27]. The first [3] proposed CRDT for collaborative text editing was the WithOut Operational Transforms (WOOT) Framework [21]. It has been implemented as part of several OSS projects [5,7,9,19]. However its eventual consistency property was verified only for a bounded model to date [20,21]. The usual commutativity of operations based proofs of consistency fail to apply for WOOT, hence we use a novel approach, identifying a previously unknown sort-key based protocol that simulates the WOOT framework. Due to the length and complexity of the proof, we formalized it and machine-checked its correctness using the proof assistant Isabelle/HOL [10]. In summary our novel contributions are:

- the first proof of the strong eventual consistency of WOOT (not limited to a bounded model),

[1] Note that other authors like Shapiro et al. [27] use CmRDT to refer to Commutative RDTs, with CRDT standing for Conflict-free RDTs.

✉ Emin Karayel
me@eminkarayel.de

Edgar Gonzàlez
edgargip@google.com

1 Google, Mountain View, USA

2 Present Address: Karlsruhe, Germany

- the introduction of a new class of sort key based protocols for collaborative editing with logarithmic message size per edit operation,
- the observation that the WOOT framework can be simulated by an instance of the above, revealing previously unknown hidden-structure of the framework.

After reviewing related work in the following section, we start in Sect. 3 with a well-known strongly consistent CRDT making a sequence of refinements until we reach the WOOT framework. This allows us to discuss each idea in the proof individually, instead of presenting a large intricate proof in one go. Section 3 motivates the question of the existence of a certain sort-key space, which we confirm in Sect. 4. In Sect. 5 we present the rigorous results implied by the previous section and their formalized proof. We discuss the high-level proof strategy and key intermediate results. In Sect. 6 we explain how we modelled the framework and proof in Isabelle/HOL. In Sect. 7 we conclude with a summary and discuss open research directions.

2 Related work

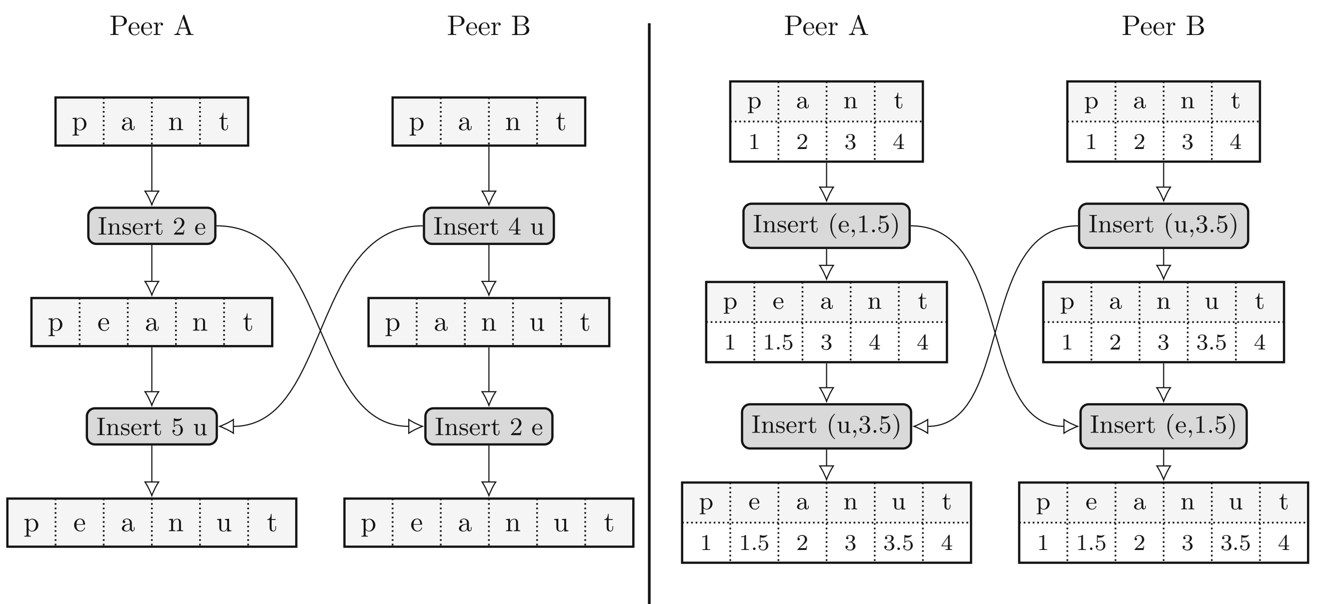
Ellis and Gibbs [6] introduced the first collaborative text editing tools, which were based on operational transformations (OT). The basic idea behind OT-based frameworks is to adjust edit operations, based on the effects of previously executed concurrent operations. For instance, in Fig. 1a, peer B can execute the message received from peer A without correction, but peer A needs to transform the one received from peer B to reach the same state. Proving the correctness of OT-based frameworks is error-prone and requires complicated case coverage [15,22]. Counter-examples have been found in most OT algorithms [25],[8, §8.2]. LSEQ [17], LOGOOT [31] and TreeDoc [23] are CRDTs that create and send sort keys for symbols (e.g., 1.5 and 3.5 in Fig. 1b). These keys can then be directly used to order them, without requiring any transformations, and are drawn from a dense totally ordered space. In the figure rational numbers were chosen for simplicity, but more commonly lexicographically ordered sequences are used.[2] The consistency property of these frameworks can be established easily. However, the space required per sort key potentially grows linearly with the count of edit operations. In LSEQ, a randomized allocation strategy for new identifiers is used to reduce the key growth, based on empirically determined edit patterns—but in the worst-case the size of the keys will still grow linearly with the count of insert operations. Preguica et al. [23] propose a solution for this problem using regular rebalancing operations. However, this can only be done using a consensus-based mechanism, which is only possible when the number of participating peers is small. A benefit of LSEQ, LOGOOT, and TreeDoc is that deleted symbols can be garbage-collected (though delete messages may have to be kept in a buffer if the corresponding insertion message has not arrived at a peer), in contrast to the WOOT Framework, where deleted symbols (tombstones) cannot be removed. Replicated Growable Arrays (RGAs) are another data structure for collaborative editing, introduced by Roh et al. [25]. Contrary to the previous approaches, the identifiers associated with the symbols are not sort keys, but are instead ordered consistently with the happened-before relation. A peer sends the identifier of the symbol immediately preceding the new symbol at the time it was created and the actual identifier associated with the new symbol. The integration algorithm starts by finding the preceding symbol and skipping following symbols with a larger identifier before placing the new symbol. The authors provide a mathematical eventual consistency proof. Recently, Gomes et al. [8] also formalized the eventual consistency property of RGAs using Isabelle/HOL [18]. The message size of both the WOOT framework and of RGAs grows only logarithmically with the number of peers and edit operations.[3] In addition to the original design of WOOT by Oster et al. [21], a number of extensions have also been proposed. For instance, Weiss et al. [30] propose a line-based version called WOOTO. Ahmed-Nacer et al. [2] introduce a second extension called WOOTH, which improves performance by using hash tables. The latter compare their implementation in benchmarks against LOGOOT, RGA, and an OT algorithm. To the best of our knowledge there were no previous publications that further expand on the correctness of the WOOT Framework. The fact that the general convergence proof was missing had also been mentioned by Kumawat and Khunteta [12, §3.10].

3 Deriving WOOT

As we mentioned in the introduction, a rigorous correctness proof for the eventual consistency of the WOOT framework is long and complex, which is the reason we relied on a proof assistant to avoid any subtle flaws in the argument. In this section we want to highlight the key ideas in it: we derive the WOOT framework starting from the well-known consistent CRDT 2P-Set (Two-Phase-Set), making iterative refinements until we reach WOOT. On each refinement, we explain intuitively why it preserves consistency. This way,

[2] In addition, peers draw sort keys from disjoint (but dense) subsets to avoid concurrently choosing the same sort key.

[3] In the original presentation RGAs had message sizes proportional with the number of peers, but Gomes et al. [8, §6.1] discuss a possible implementation using the logical clocks introduced by Lamport [13].



(a) Transformation-based **(b)** Sort-key based

Fig. 1 Collaborative text editing

we can convey each idea from the exhaustive proof, independently and decoupled from each other.

3.1 2P-Set

The 2P-Set [27] is one of the simple CRDTs allowing a shared replicated data structure for a set where elements can be added and removed.

Each peer keeps track of two sets S and R initially both being empty. An element e is added/removed to/from the replicated data structure by broadcasting a message **add e / remove e**. A peer that receives an **add e** message adds the element e to the first set S. If a peer receives a **remove e** it will add it to the second set R. A peer determines whether an element is in the 2P-Set by checking whether it is in the first set and not in the second set. Note that an element which was removed can never be added again.[4] Messages are broadcast to all peers including the one that originated the message and the integration algorithm for a message is identical, irrespective of whether the message originated from the same peer or from a different one.[5] We summarize the CRDT in Protocol 1. We think of the peers as single threaded machines, communicating by broadcasting messages in an asynchronous network. There is no synchronization or shared variables between peers. For simplicity, we do assume a single broadcast message will be received (with a possibly arbitrary delay) at most once by each peer[6] and that messages are neither being altered nor being sent by peers implementing a different protocol. It is easy to see that, once each peer has received all updates, they will all be in the same state, i.e., if we assume eventual delivery than we have eventual consistency. And, similarly, that two peers who have integrated the same set of messages will be in the same state. This essentially follows from the fact that set union is commutative, associative, and idempotent. For example the operations **add e / add f** commute because:

$$(S \cup \{e\}) \cup \{f\} = S \cup \{e, f\} = (S \cup \{f\}) \cup e$$

A more naive implementation with only a single set, where remove (element e) removes the element from that set, would not have the same consistency properties: A peer that receives an insert operation after the remove operation of the same element will be in a different state than a peer that receives the remove operation after the insert operation. Algebraically this happens because:

$$(S \setminus \{e\}) \cup \{e\} \neq (S \cup \{e\}) \setminus \{e\}$$

In the CRDT community the elements of the second set are called tombstones. The information in the set needs to be pre-

[4] In practice this can be circumvented by storing an additional unique identifier per set element. See also U-set [27] for an example of this design.

[5] Modelling a replicated data structure this way enables easier proofs, as there is no need to distinguish those two cases. However, a real-world implementation may have separate code-paths for broadcasting a message to other peers, and integrating the message locally. See for example [8, §5.2], where the same approach has been taken.

[6] In a real error-prone network, this could be achieved by having a unique identifier per message and keeping track of already received messages.

served to enable order-independent integration of messages and would be avoidable in a single process application.

Protocol 1 2P-Set

Each peer is initialized by calling the **init** function. Here we set up the global state variables S and R.

```
init
    $S \leftarrow \emptyset$ : set
    $R \leftarrow \emptyset$ : set
```

The **query** functions are part of the interface of the CRDT and provide a view to the data structure.

```
query lookup($e$ : element) : boolean
    return $e \in S \wedge e \notin R$
```

Similarly, the **modify** functions are part of the interface of the CRDT and allow modifications to the data structure.

```
modify add($e$ : element)
    broadcast (add $e$)

modify remove($e$ : element)
    broadcast (remove $e$)
```

Modification functions broadcast messages, that are integrated into the state of each peer, when the respective message is received using the **integrate** functions:

```
integrate add($e$ : element)
    $S \leftarrow S \cup \{e\}$

integrate remove($e$ : element)
    $R \leftarrow R \cup \{e\}$
```

3.2 Sort keys

Symbols may occur multiple times and in arbitrary order within text, hence sets do not seem to be a useful CRDT for collaborative text editing, but we can easily support sequences using the 2P-Set as a building block.

To do that we use set elements e that are pairs of:

- A sort key $\alpha(e) \in \mathcal{A}$
- The symbol $\sigma(e) \in \Sigma$

The user only observes the symbols, but in the order induced by the sort keys. In the following we use the term character to refer to the compound tuple consisting of the symbol and additional associated information, like the sort key. In Table 1, we give an example of a state with pairs of sort keys and symbols—for the sequence “pant”.

Characters are inserted into the sequence by creating a sort key that is ordered between those of the preceding and succeeding sequence elements. In the example (Table 1), to insert the symbol ‘l’ between the first and second character, a peer could generate the new character (1.5, ‘l’). The algorithm to query the current string based on the set of non-deleted characters consists of sorting the sequence according to the sort key and presenting the resulting symbol sequence to the user.

Table 1 Sort keys and symbols

Sort key	Symbol
1	p
2	a
3	n
4	t

Protocol 2 Sort Key based Protocol for String Editing

```
init
    $S \leftarrow \emptyset : (\mathcal{A} \times \Sigma)$ set
    $R \leftarrow \emptyset : (\mathcal{A} \times \Sigma)$ set

query view() : $\Sigma$ list
    $w := \text{sort}_\alpha(S \setminus R)$
    return $\sigma(w_1) \cdots \sigma(w_{|S \setminus R|})$                ▷ Symbols of $w$

modify insert($\sigma : \Sigma, k : \mathbb{N}$)                ▷ Insert $\sigma$ at position $k$
    $l := \alpha(\text{rank}_\alpha(S \setminus R, k))$ or $\vdash$ if $k = 0$
    $u := \alpha(\text{rank}_\alpha(S \setminus R, k+1))$ or $\dashv$ if $k = |S \setminus R|$
    $\alpha :=$ build-sort-key($l, u$)
    broadcast(insert $(\alpha, \sigma)$)

modify delete($k : \mathbb{N}$)                ▷ Delete the $k$-th character
    $c := \text{rank}_\alpha(S \setminus R, k))$
    broadcast(delete $c$)

integrate insert($c : (\mathcal{A} \times \Sigma)$)
    $S \leftarrow S \cup \{c\}$

integrate remove($c : (\mathcal{A} \times \Sigma)$)
    $R \leftarrow R \cup \{c\}$
```

In Protocol 2 we summarize the framework: We denote by $\text{rank}_\alpha(S, k)$ the function that returns the k-th smallest[7] element of the set S according to the order induced by α. Similarly $\text{sort}_\alpha(S)$ returns the sequence of the elements in S according to the order induced by α.

The algorithm **build-sort-key** computes a sort key between the pair of sort keys it was given. Note that for the special case where a sort key for the beginning (resp. end) of the string needs to be generated: We allow passing in the special values $\vdash$ (resp. $\dashv$) as first (resp. second) argument representing an element outside the set of sort keys with order strictly smaller (resp. larger) than all of them.

[7] Here k is indexed starting from 1; e.g. $\text{rank}_\alpha(S, 1)$ is the smallest element and $\text{rank}_\alpha(S, |S|)$ is the largest element.

To clarify, the above approach works under the assumption that the available sort keys are elements of a dense totally ordered set, such that it is always possible to find a new sort key between each pair of previously generated sort keys.[8] Additionally, a mechanism needs to be introduced that prevents the possibility of choosing the same sort key twice.

3.3 Avoiding collisions

As mentioned in the previous paragraph, the above solution may lead to collisions, where two distinct characters are inserted at the same position with the same sort key. That prevents them from being ordered in a sequence, as well as any possibility of inserting a character between them. To resolve this, we introduce a unique id $i \in \mathcal{I}$ for each inserted character. We reserve distinct dense subsets for the sort keys of each such unique id.

Protocol 3 New version of the Insert Algorithm

```
modify insert($\sigma : \Sigma, k : \mathbb{N}$)
    $l := \alpha(\text{rank}_\alpha(S \setminus R, k))$ or $\vdash$ if $k = 0$
    $u := \alpha(\text{rank}_\alpha(S \setminus R, k+1))$ or $\dashv$ if $k = |S \setminus R|$
    $i :=$ create-unique-id()
    $\alpha := \Psi(l, i, u)$
    broadcast(insert $(\alpha, \sigma)$)
```

In Protocol 3, we give an updated version of the insertion algorithm. The function **create-unique-id** creates a new globally unique id. This can be achieved by assigning a unique id to each peer and keeping and incrementing a counter on each peer, the unique id of the character would be formed by the pair of peer id and counter. The function

$$\Psi : (\mathcal{A} \cup \{\vdash\}) \times \mathcal{I} \times (\mathcal{A} \cup \{\dashv\}) \to \mathcal{A}$$

is used to generate a new sort key, with the unique id as an additional argument. As before, the first argument (lower bound) may be the special value $\vdash$ to facilitate the creation of a sort key for the beginning of the string. And, similarly, the last argument (the upper bound) may be the special value $\dashv$ to facilitate insertion at the end of the string.

Note that we require Ψ to have at least the following properties:

Condition 1 If $l < u$ then $l < \Psi(l, i, u) < u$.

Condition 2 If $l < u, l' < u'$ and $\Psi(l, i, u) = \Psi(l', i', u')$ then $i = i'$.

The first property ensures that the newly computed sort key is actually between the sort keys of the adjacent characters. The second implies that for distinct unique ids the generated sort keys will be distinct, for any predecessor and successor.

We can give an example for a function fulfilling conditions 1, and 2 in the case where the identifiers are natural numbers strictly between 0 and b, i.e., $\mathcal{I} = \{1, \ldots, b-1\}$. Let $\mathbb{Q}_{b,i}$ be the rational numbers with a finite b-ary representation ending with $i \in \mathcal{I}$:

$$\mathbb{Q}_{b,i} := \left\{ b^{-k}(i + bj) \middle| k \in \mathbb{N}, j \in \mathbb{Z} \right\}$$

and let θ be an injective function from the rational numbers to the natural numbers.[9] We can then define the following function Ψ on $\mathcal{A} := \mathbb{Q} \cap (0, 1)$, the set of rational numbers between 0 and 1:

$$\begin{aligned} \Psi \ : \ & (\mathcal{A} \cup \{0\}) \times \mathcal{I} \times (\mathcal{A} \cup \{1\}) \to \mathcal{A} \\ \Psi(l, i, u) := & \operatorname*{argmin}_{x \in \mathbb{Q}_{b,i} \cap (l,u)} \theta(x) \end{aligned}$$

i.e. if we order the rational numbers according to the enumeration induced by θ, then $\Psi(l, i, u)$ is the first rational number in the sequence that has a finite b-ary representation ending in i and whose value is strictly between l and u.

Note that we identified $\vdash$ with 0 (representing the beginning of the string) and $\dashv$ with 1 (representing the ending of the string), while the sort keys are rational numbers strictly between 0 and 1. We don't give a proof that this indeed works, since we will see below that we need an additional property for Ψ which narrows down the possible definitions for Ψ further. But the curious reader may verify that the candidate sets for each i are disjoint but all are dense in $\mathbb{Q}$.

3.4 Avoiding transfer of sort keys

The above scheme has the drawback that the bit size of the sort keys can grow linearly with the number of edit operations and, since they are part of the transferred operations, the same is true for the message sizes per edit operation. To fix that, we are making a second change to the scheme. Instead of transferring the sort keys themselves, we send the unique ids (from the previous section) of each character, as well as the unique id of its immediate predecessor and successor at the time it was created. Additionally, we require that Ψ is a pure function, used by all peers. This allows every peer to compute the sort keys themselves.

Consider for example the characters in Table 2. We would assign them the sort keys:

- $\alpha_p = \Psi(\vdash, i_1, \dashv)$
- $\alpha_a = \Psi(\alpha_p, i_2, \dashv)$
- $\alpha_e = \Psi(\alpha_p, i_3, \alpha_a)$

[8] This is why we use rational numbers in the example as they are a well-known dense totally ordered set.

[9] Sagher [26] presents an example of such a function.

Table 2 Characters with predecessors and successor identifiers

Predecessor	Identifier	Successor	Symbol
$\vdash$	i_1	$\dashv$	p
i_1	i_2	$\dashv$	a
i_1	i_3	i_2	e

The identifier of a character, as well as the identifiers of its predecessor/successor, i.e., the identifiers of the character that were preceding/succeeding it at the time it was created, are immutable and using the function Ψ it is possible to compute the sort keys of each character recursively.

Note that the unique ids do not have to be order-preserving and can be constructed in way that they only have logarithmic size with respect to the number of participating peers and edit operations.

Protocol 4 Deferred Computation of Sort Keys

```
init
    $S \leftarrow \emptyset : (\mathcal{I} \times \mathcal{A} \times \Sigma)$ set
    $R \leftarrow \emptyset : \mathcal{I}$ set

query view() : $\Sigma$ list
    $w := \text{sort}_\alpha \left(\{c \in S | i(c) \notin R\}\right)$
    return $\sigma(w_1) \cdots \sigma(w_{|w|})$                    ▷ Symbols of $w$

modify insert($\sigma : \Sigma, k : \mathbb{N}$)                    ▷ Insert $\sigma$ at position $k$
    $l := i(\text{rank}_\alpha(S \setminus R, k))$ or $\vdash$ if $k = 0$
    $u := i(\text{rank}_\alpha(S \setminus R, k+1))$ or $\dashv$ if $k = |S \setminus R|$
    $i :=$ create-unique-id()
    broadcast(insert $(l, i, u, \sigma)$)

modify delete($k : \mathbb{N}$)                    ▷ Delete the $k$-th character
    $i := i(\text{rank}_\alpha(S \setminus R, k))$
    broadcast(delete $i$)

integrate insert($l : \mathcal{I} \cup \{\vdash\}, i : \mathcal{I}, u : \mathcal{I} \cup \{\dashv\}, \sigma : \Sigma$))
    $\alpha_l := \alpha \left(\text{find} \left\{c \in S | i(c) = l\right\}\right)$ or $\vdash$ if $l = \vdash$
    $\alpha_u := \alpha \left(\text{find} \left\{c \in S | i(c) = u\right\}\right)$ or $\dashv$ if $u = \dashv$
    $S \leftarrow S \cup \{(i, \Psi(\alpha_l, i, \alpha_u), \sigma\}$

integrate delete($i : \mathcal{I}$)
    $R \leftarrow R \cup \{i\}$
```

In Protocol 4 we summarize the new scheme. In it, a character is represented by a triple in a peer's state; in particular:

- The identifier $i(c) \in \mathcal{I}$
- A sort key $\alpha(c) \in \mathcal{A}$ (as in the previous scheme)
- The symbol $\sigma(c) \in \Sigma$ (as in the previous scheme)

The keen reader will notice that the scheme could fail if insert messages are delivered out-of-order. In particular, when the insert message m_1 for the referenced sort key of an insert message m_2 has not been delivered to a peer, the integration of m_2 will not succeed.

3.5 Semantic causal delivery

One way to ensure that such a failure does not happen is to delay the integration of messages whose dependencies have not been received yet. The authors of WOOT have coined the term *semantic causal delivery* to refer to this delivery mechanism. Indeed it is a weaker form of causal delivery, which would imply a message is only delivered to a peer if all messages that were present during its creation were already delivered.

Since the latter set subsumes the dependencies, causal delivery will automatically imply semantic causal delivery, but not vice versa.

Note that since there is never a dependency on a delete message, and the insert messages already have a unique id associated to them, we denote by deps(m) the set of insert messages a message depends on (more precisely the ids of the inserted characters). In Protocol 5 we depict a possible mechanism to ensure semantic causal delivery, where messages whose dependencies have not been received yet are buffered, until their dependencies are integrated. For the discussion in the following sections, we will assume that some mechanism is employed to ensure semantic causal delivery—or potentially a stronger delivery notion.

Protocol 5 Semantic Causal Delivery

```
function deps($m$ : message) : $\mathcal{I}$ set
    if $m =$ insert$(l, i, u, \sigma)$ then
        return $\{l, u\} \setminus \{\vdash, \dashv\}$
    else if $m =$ delete $i$ then
        return $\{i\}$
    end if

procedure receive($m$ : message)
    $B \leftarrow B \cup \{m\}$
    while $\exists m' \in B$ . deps$(m') \subseteq \left\{i(c) | c \in S\right\}$ do
        $B \leftarrow B \setminus \{m'\}$
        integrate$(m')$
    end while
```

3.6 Acyclic dependency graph

The dependency relation between messages above is acyclic during a run of the framework. To see this, note that a message can only depend on messages already integrated by a peer. In fact it is a consequence of the fact the state graph of a distributed message-passing algorithm is acyclic—when we associate the messages with the states on which they were generated, we can see that the dependency relation is a sub-

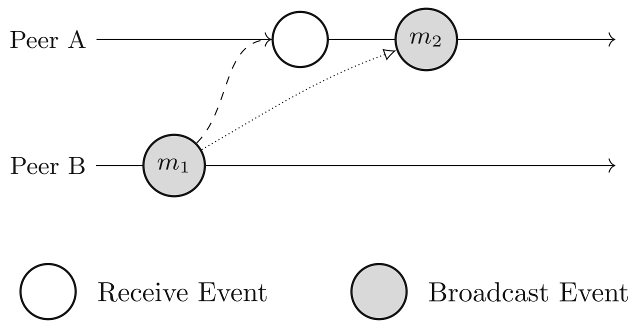


Fig. 2 The causal relationship of messages is a subrelation of the happened-before relation

relation of Lamport's acyclic happened-before relation. In Fig. 2 we illustrate the case of a message m_2 dependent on m_1, which also implies that the state creating m_1 must have happened before the state creating m_2. Note that the converse is not necessarily true, i.e., a message may not semantically depend on all the messages that were created or integrated before it.

3.7 Interleaving anomalies

In Sect. 3.3 we described the minimum requirements on Ψ. The conditions ensure intention preservation, i.e., that the character appears at the place it was inserted. But for concurrent insertion at the same place, the order is unspecified. For example, if two peers concurrently insert a character with symbol x and y between a and b the result, after all messages are received may be "axyb" or "ayxb". A well known anomaly [11] is the situation where entire words are inserted in the same spot concurrently. Consider for example the concurrent insertion of "pea" and "nut" at position 7 in "I␣like␣s" A good outcome would be "I like peanuts" or "I like nutpeas"; a careless definition of Ψ could assign sort keys resulting in something like "I like pneauts".

We solve the issue in two steps. First, similarly to Oster et al. [21], we require that the unique identifiers (not the sort keys) associated to each character form a total order themselves. We cannot require the order to be monotone with respect to the ordering of the characters (that's what the sort keys are there for), but we want to make sure that the unique identifiers generated by distinct peers do not interleave, i.e., identifiers generated by peer A should never be ordered between identifiers generated by peer B[10]. The second step is to add an additional constraint to the function Ψ, requiring that the sort keys preserve the order on the identifier, if it does not violate Condition 1:

[10] This can be easily achieved by creating identifiers of the form (p,n) where p is a unique identifier associated to the peer, and n is a local counter on the peer. Then the tuples can be endowed with the lexicographic order of the product space.

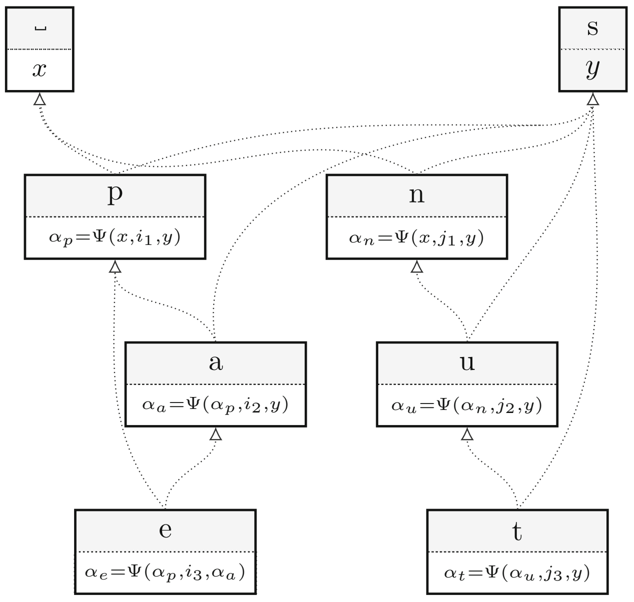

␣
x
s
y
p
$\alpha_p=\Psi(x,i_1,y)$
n
$\alpha_n=\Psi(x,j_1,y)$
a
$\alpha_a=\Psi(\alpha_p,i_2,y)$
u
$\alpha_u=\Psi(\alpha_n,j_2,y)$
e
$\alpha_e=\Psi(\alpha_p,i_3,\alpha_a)$
t
$\alpha_t=\Psi(\alpha_u,j_3,y)$

Fig. 3 Sort keys associated with the concurrent insertion of 'pea' and 'nut'

Condition 3 If $l < \Psi(l', i', u') < u$, $l' < \Psi(l, i, u) < u'$ and $i < i'$ then $\Psi(l, i, u) < \Psi(l', i', u')$.

Given two characters that are to be inserted within each other's boundaries, we require that the order of the identifiers is respected. The condition is non-trivial, but it can be derived from Oster et al. [21, Theorem 3], under the assumption that there is a sort-key mapping Ψ.

Let us see how this avoids the "pneauts" outcome: We call the sort key of the second space (resp. s symbol) in "I␣like␣s" x (resp. y). The identifiers from the first peer generating "pea" are called i_1, i_2 and i_3. The identifiers from the second peer generating "nut" are called j_1, j_2 and j_3. We assume $i_1 < i_2 < i_3 < j_1 < j_2 < j_3$. The first peer types "pa" and inserts the "e" between those in the last step, so that i_2 is associated with "a" and i_3 is associated with "e". In Fig. 3 we present the associated sort keys to each of the symbols.

Using Condition 1 we can easily deduce that:

$$x < \alpha_p < \alpha_e < \alpha_a < y$$
$$x < \alpha_n < \alpha_u < \alpha_t < y$$

Since $x < \alpha_p < y$ and $x < \alpha_n < y$ we can conclude using Condition 3 that α_p and α_n are ordered with respect to the order chosen between i_1 and j_1, i.e., $\alpha_p < \alpha_n$. Using Condition 3 again for α_a and α_n, where we have $\alpha_p < \alpha_a < y$ and $\alpha_p < \alpha_n < y$, we can determine that $\alpha_a < \alpha_n$, which implies: $\alpha_p < \alpha_e < \alpha_a < \alpha_n < \alpha_u < \alpha_t$.

3.8 Avoiding the computation of sort keys

The keen reader may have noticed that, in the example above, there was no need to refer to a concrete instantiation of the function Ψ: we could order the entered symbol just by using conditions 1 and 3 as rules. We will see that this is always possible, and that leads to the last change to the framework we are building.

Instead of computing the sort key of a character, we keep the characters in a sequence according to the order induced by the sort keys. The state of a peer is a sequence of characters: $w_1, \ldots, w_{|w|}$. For each character w_k, we remember the identifiers of the predecessor and successor characters $l(w_k), u(w_k)$ (the characters that were adjacent to the character when it was created), as well as the character's identifier $i(w_k)$ and symbol $\sigma(w_k)$. This information is being stored, so that we can make sure the preconditions of Condition 3 are met. Note that we never remove characters from the sequence but just mark them as deleted, by replacing the characters symbol with $\perp$. The integration algorithm for an insert message becomes the following:

Given a new received character w_{new} to be inserted, we can look up the positions l and u of the preceding and succeeding characters in the sequence, for which we know the identifiers.

$$l = \text{idx}_w(l(w_{\text{new}}))$$
$$u = \text{idx}_w(u(w_{\text{new}}))$$

Note that $\text{idx}_w(i)$ denotes the position of a character in a peer's state w with identifier i. Note also that $l(w_{\text{new}})$ (resp. $u(w_{\text{new}})$) may be $\vdash$ (resp. $\dashv$). Thus, we define $\text{idx}_w(\vdash) := 0$ and $\text{idx}_w(\dashv) := |w| + 1$ for completeness. Due to Condition 1, we know that the sort key of the new character has to be between the sort keys of w_l and w_u. In the easiest case, these two would already be adjacent (i.e. $u = l + 1$) and we can just insert the new character after w_l. If not, we need to narrow down the position further using Condition 3. This is possible for a subset $T_w(l, u)$ of the characters strictly between w_l and w_u whose dependencies are outside of the range w_l and w_u, i.e., both the position of the predecessor (resp. successor) of w_t for $t \in T_w(l, u)$ needs to be less (resp. greater) than or equal to l (resp. u), which implies:

$$\alpha(l(w_t)) \leq \alpha(i(w_l)) < \alpha(i(w_{\text{new}})) < \alpha(i(w_u)) \leq \alpha(u(w_t))$$

Similarly we have:

$$\alpha(l(w_{\text{new}})) \leq \alpha(i(w_l)) < \alpha(i(w_t)) < \alpha(i(w_u)) \leq \alpha(u(w_{\text{new}}))$$

In those cases, we can use Condition 3 to conclude that $\alpha(w_{\text{new}}) < \alpha(w_t)$ or $\alpha(w_{\text{new}}) > \alpha(w_t)$ depending on whether $i(w_{\text{new}}) < i(w_t)$ or $i(w_{\text{new}}) > i(w_t)$ for each $w_t \in T_w(l, u)$. Note that $T_w(l, u)$ cannot be empty: Since the dependency graph is acyclic, we can choose a minimal element according to the dependency relation from the set of characters strictly between w_l and w_u, which by definition cannot have a dependency between w_l and w_u (otherwise it would not be a minimal element). Also, we can apply Condition 3 between pairs of elements in $T_w(l, u)$, which implies that the identifiers in $T_w(l, u)$ will be strictly increasing with the position of the character. This leads to an integration algorithm where a consecutive pair of elements in $T_w(l, u)$ (enclosing the identifier $i(w_{\text{new}})$) is chosen to narrow down l and u.[11]

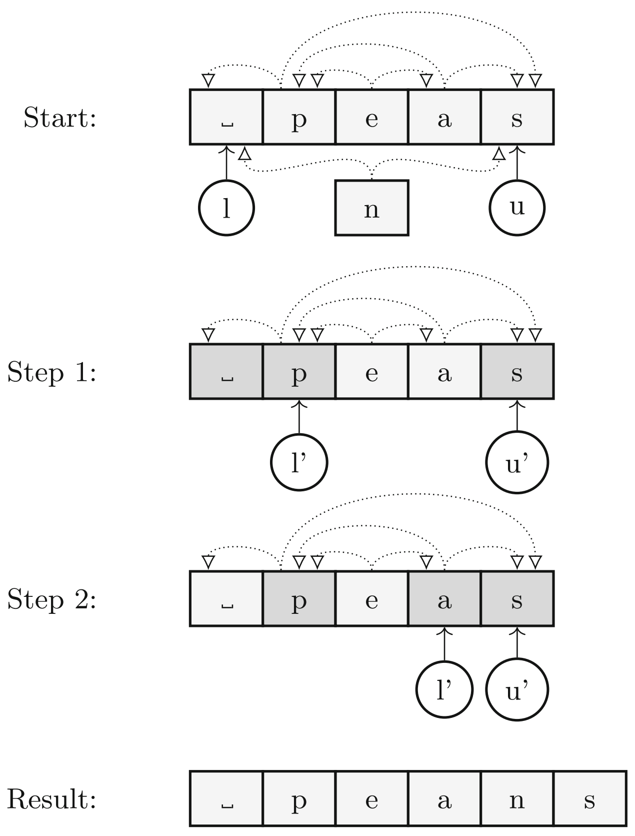


Fig. 4 Example of the insertion of the character with symbol 'n' after the characters 'p', 'e' and 'a' from the previous example in Sect. 3.7. In the first step the integration algorithm determines the possible position for the new character using its predecessor and successor. After that the position is further narrowed down, using identifier comparisons with characters whose dependencies are outside the target range. The characters that are in L in each iteration step are depicted in dark gray. The positions of the narrowed bounds for the next step are depicted using the arrows with the labels l' and u'

[11] In the case where the identifiers of all elements in $T_w(l, u)$ are larger (resp. smaller) than $i(w_{\text{new}})$, the lower bound (resp. upper bound) remains l (resp. u) while the upper bound (resp. lower bound) becomes the first (resp. last) index in $T_w(l, u)$.

Protocol 6 The WOOT Framework

```
init
    $w \leftarrow [] : (\mathcal{I} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{I} \cup \{\dashv\} \times \Sigma \cup \{\bot\})$ list

query view() : $\Sigma$ list
    $w' := \text{filter}(\lambda c.\sigma(c) \neq \bot)w$
    return $\sigma(w'_1)\sigma(w'_2)\ldots\sigma(w'_{|w'|})$

modify insert($\sigma : \Sigma, k : \mathbb{N}$)
    $w' := \text{filter}(\lambda c.\sigma(c) \neq \bot)w$
    $l := i(w'_k)$ or $\vdash$ if $k = 0$
    $u := i(w'_{k+1})$ or $\dashv$ if $k = |w'|$
    $i :=$ create-unique-id()
    broadcast(insert $(l, i, u, \sigma)$)

modify delete($k : \mathbb{N}$)
    $w' := \text{filter}(\lambda c.\sigma(c) \neq \bot)w$
    $i := i(w'_k)$
    broadcast(delete $i$)

integrate insert($l_{\text{id}} : \mathcal{I} \cup \{\vdash\}, i : \mathcal{I}, u_{\text{id}} : \mathcal{I} \cup \{\dashv\}, \sigma : \Sigma$)
    $l \leftarrow \text{idx}_w(l_{\text{id}})$
    $u \leftarrow \text{idx}_w(u_{\text{id}})$
    while $u - l \neq 1$ do
        $T := \{k | \text{idx}_w(l(w_k)) \leq l < k < u \leq \text{idx}_w(u(w_k))\}$
        $L := T \cup \{l, u\}$
        $(l, u) \leftarrow \min\{(l', u') | u = u' \vee i \leq i(w_{u'})$ where
                $l'$ and $u'$ are consecutive in $L\}$
    end while
    $w \leftarrow w_1 \ldots w_l \, (l_{\text{id}}, i, u_{\text{id}}, \sigma) \, w_u \ldots w_{|w|}$

integrate delete($i : \mathcal{I}$)
    $k := \text{idx}_w(i)$
    $c' := (l(w_k), i(w_k), u(w_k), \bot)$
    $w \leftarrow w_1 \ldots w_{k-1} \, c' \, w_{k+1} \ldots w_{|w|}$
```

In Protocol 6 we present the resulting framework, which is the WOOT framework as described by Oster et al. In Fig. 4 we depict the integration of the character 'n' from the example of Sect. 3.7 using the integration algorithm. To summarize, if the function Ψ fulfills conditions 1, 2 and 3 then the frameworks in Protocol 4 and Protocol 6 behave identically, exchanging the same set of messages, for the same modifications and providing the same view. While the internal state of Protocol 6 keeps the sequence of characters ordered according the sort keys implicitly, Protocol 4 explicitly computes them.

4 The function Ψ

As seen in the previous section, it is possible to simulate WOOT with the sort-key based Protocol 4, under the assumption that for any totally ordered identifier space $\mathcal{I}$ we can construct a sort key space and a function Ψ fulfilling Conditions 1, 2 and 3. In this section, we prove that this is indeed possible. We start by constructing such a sort key space under the assumption that $\mathcal{I}$ is finite. To extend to the infinite case, we then use the compactness theorem. We omit Condition 2 in the intermediate results, but we conclude at the end of this section that Condition 2 follows from Conditions 1 and 3. We would like to note that this is a novel result, not previously mentioned in publications to the best of our knowledge.

Proposition 1 *Let $b \geq 2$ and $\mathcal{I} := \left\{\frac{1}{b}, \ldots, \frac{b-1}{b}\right\}$ be the set of rational numbers with denominator b strictly between 0 and 1. Then there exists a totally ordered set $\mathcal{A}$ and a function Ψ with domain $\left\{(l, i, u) \middle| l < u \in \mathcal{A} \cup \{\vdash, \dashv\}, i \in \mathcal{I}\right\} \subseteq \mathcal{A} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{A} \cup \{\dashv\}$ and range $\mathcal{A}$ fulfilling conditions 1 and 3.*

Proof We denote by $\mathbb{Q}_b$ the set of rational numbers with finite b-ary expansion, i.e., $\mathbb{Q}_b := \left\{x \in \mathbb{Q} \middle| \exists k \in \mathbb{N}\,.\, b^k x \in \mathbb{Z}\right\}$. Note that $\mathbb{Q}_b$ is a ring, it is closed with respect to addition, negation, multiplication. It is also closed under division by b. With $d_b(x)$, we denote the length of that expansion, i.e., $d_b : \mathbb{Q}_b \to \mathbb{N}$ where $d_b(x) := \min\left\{k \in \mathbb{N} \middle| b^k x \in \mathbb{Z}\right\}$. Note that $d_b(x) = 0$ if and only if $x \in \mathbb{Z}$, moreover $d_b(x + y) \leq \max(d_b(x), d_b(y))$ for all $x, y \in \mathbb{Q}_b$. And $d_b(bx) = d_b(x) - 1$ if $d_b(x) > 0$ for all $x \in \mathbb{Q}_b$.

We denote by $\lfloor x \rfloor$ (resp. $\lceil x \rceil$) the largest integer (resp. smallest integer) smaller or equal (resp. larger or equal) to x. Let $\mu_l(x) := bx - \lfloor bl \rfloor$ and $\nu_u(x) := bx - \lceil bu \rceil + 1$, then

$$0 \leq \mu_l(l) < 1 \tag{1}$$
$$0 < \nu_u(u) \leq 1 \tag{2}$$

for all $l, u \in \mathbb{Q}_b$. Additionally, we can conclude using the properties about d_b we established:

$$\max(d_b(\mu_l(l)), d_b(\mu_l(u))) = \max(d_b(l), d_b(u)) - 1 \tag{3}$$
$$\max(d_b(\nu_u(l)), d_b(\nu_u(u))) = \max(d_b(l), d_b(u)) - 1 \tag{4}$$

if $\max(d_b(l), d_b(u)) > 0$ for all $l, u \in \mathbb{Q}_b$. We write μ_l^{-1}, ν_u^{-1} for the inverse of μ_l and ν_u, i.e.,

$$\mu_l^{-1}(x) = \frac{x + \lfloor bl \rfloor}{b}$$
$$\nu_u^{-1}(x) = \frac{x + \lceil bu \rceil - 1}{b}$$

observing that $\mu_l^{-1}(\mu_l(x)) = x$ and $\nu_u^{-1}(\nu_u(x)) = x$.

Definition of Ψ: Finally, we can define Ψ using recursion on the length of the expansion of l and u, i.e., $\max(d_b(l), d_b(u))$:

$$\Psi(l, i, u) := \begin{cases} i & \text{if } l < i < u, \\ \mu_l^{-1}(\Psi(\mu_l(l), i, \mu_l(u))) & \text{if } i \leq l, \text{ and} \\ \nu_u^{-1}(\Psi(\nu_u(l), i, \nu_u(u))) & \text{if } i \geq u \end{cases}$$

for $i \in \mathcal{I}$ and $l, u \in \mathbb{Q}_b$ such that $l < u$, $l < 1$ and $u > 0$. To provide an intuition for the function Ψ, we refer to Fig. 5. We split the rational numbers between 0 and 1 into b equal-sized intervals. The identifiers correspond to the endpoints of the intervals that are strictly between 0 and 1. During each recursion step, an interval is scaled to the range between 0 and 1. Note that we define Ψ on a larger domain $\left\{(l,i,u) \middle| l < u \in \mathbb{Q}_b, l < 1, u > 0, i \in \mathcal{I}\right\}$ than stated in the proposition. (This is necessary, since the recursion relies on the extended domain, for example when $b = 4$:

$$\Psi\left(\frac{2}{4},\frac{1}{4},1\right) = \mu_{\frac{1}{4}}^{-1}\left(\Psi\left(0,\frac{1}{4},2\right)\right) = \frac{9}{16}.$$

See also the second example in Fig. 5. On the other hand, if Ψ fulfills conditions 1 and 3, this will remain true for any restriction of it.) In the case that $\max(d_b(l), d_b(u)) = 0$ we can conclude $l \leq 0$ from the fact that $l < 1$ and integer, and similarly that $u \geq 1$ and thus $l < i < u$, i.e., $\Psi(l,i,u)$ is directly defined, i.e.,

$$\Psi(l,i,u) = i \quad \text{and } l < i < u \qquad \text{if } \max(d_b(l), d_b(u)) = 0. \tag{5}$$

Otherwise, we can conclude from (3) and (4) that the value $\Psi(l,i,u)$ is either also directly defined or in terms of arguments with smaller $\max(d_b(l), d_b(u))$. That those are still in the domain follow from the monotony of μ_l, ν_u as well as the inequalities (1), (2). Note that since $i \in \mathbb{Q}_b$ and that the application of both μ_l^{-1}, ν^{-1} preserve membership in $\mathbb{Q}_b$, we can deduce that the range of Ψ is in $\mathbb{Q}_b$.

As before we identify 0 with $\vdash$, the sort key associated to the beginning of the string, and 1 with $\dashv$, the sort key associated with the end of the string. The set of sort keys is $\mathcal{A} = \mathbb{Q}_b \cap (0,1)$.

Range of Ψ: We first show that

$$0 < \Psi(l,i,u) < 1 \tag{6}$$
$$l < \Psi(l,i,u) < u. \tag{7}$$

on the domain of Ψ using induction on $\max(d_b(l), d_b(u))$. If $\max(d_b(l), d_b(u)) = 0$ we have $\Psi(l,i,u) = i$ using (5) confirming both (6) and (7). For the induction step let us assume the statements are true if $\max(d_b(l), d_b(u)) = n$. And let $\max(d_b(l), d_b(u)) = n+1$. We consider the three cases from the definition of Ψ separately:

- Case $l < i < u$: Then $\Psi(l,i,u) = i$ and both (6) and (7) follow by definition.
- Case $i \leq l$: Then $\Psi(l,i,u) = \mu_l^{-1}(\Psi(\mu_l(l), i, \mu_l(u)))$, and we can using the induction hypothesis conclude $\mu_l(l) < \Psi(\mu_l(l), i, \mu_l(u)) < \mu_l(u)$ which implies (7). Also using the induction hypothesis we have $0 < \Psi(\mu_l(l), i, \mu_l(u)) < 1$ which implies

$$b^{-1}\lfloor bl \rfloor < \Psi(l,i,u) < b^{-1}(\lfloor bl \rfloor + 1),$$

from which (6) follows using $0 < i \leq l < 1$.

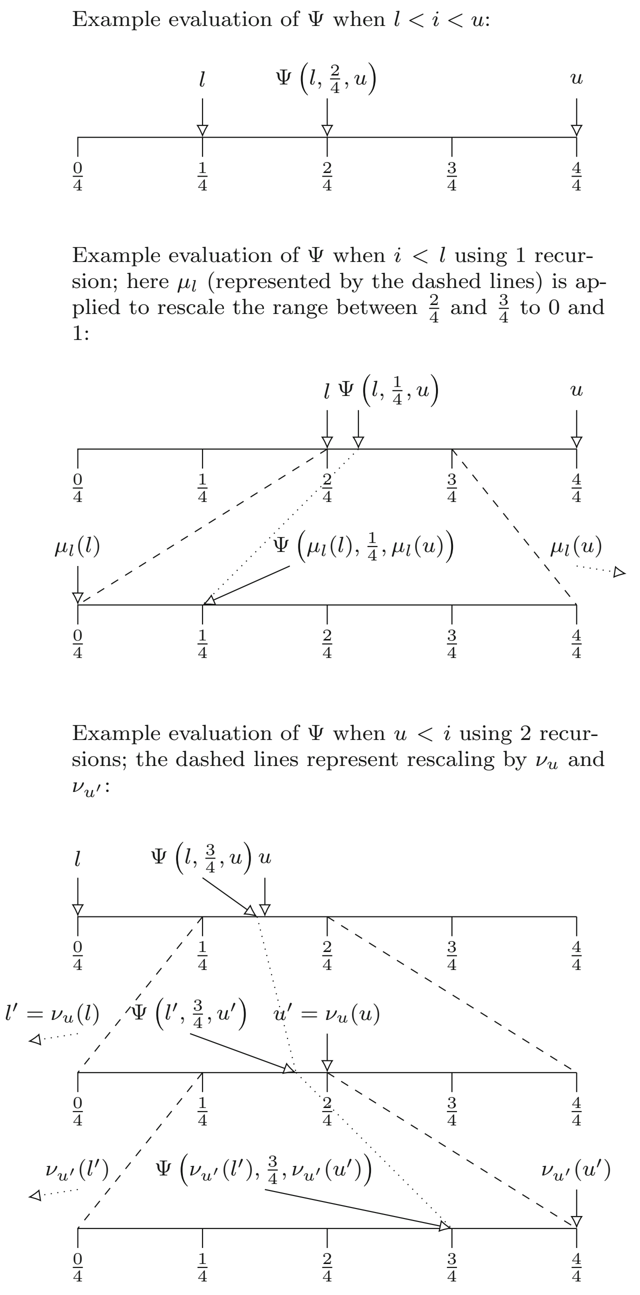

Example evaluation of Ψ when l < i < u:
l
Ψ(l, 2/4, u)
u
0/4
1/4
2/4
3/4
4/4
Example evaluation of Ψ when i < l using 1 recursion; here μ_l (represented by the dashed lines) is applied to rescale the range between 2/4 and 3/4 to 0 and 1:
l Ψ(l, 1/4, u)
u
μ_l(l)
Ψ(μ_l(l), 1/4, μ_l(u))
μ_l(u)
Example evaluation of Ψ when u < i using 2 recursions; the dashed lines represent rescaling by ν_u and ν_u′:
l
Ψ(l, 3/4, u) u
l′ = ν_u(l)
Ψ(l′, 3/4, u′)
u′ = ν_u(u)
ν_u′(l′)
Ψ(ν_u′(l′), 3/4, ν_u′(u′))
ν_u′(u′)

Fig. 5 Example evaluations of Ψ with no, one and two recursions for $b = 4$

- Case $u \leq i$: Then $\Psi(l,i,u) = \nu_u^{-1}(\Psi(\nu_u(l), i, \nu_u(u)))$ and we can again using the induction hypothesis conclude $\nu_u(l) < \Psi(\nu_u(l), i, \nu_u(u)) < \nu_u(u)$ which implies (7). Also using the induction hypothesis we have $0 <$

$\Psi(\nu_u(l), i, \nu_u(u)) < 1$ which implies

$$b^{-1}(\lceil bu \rceil - 1) < \Psi(l,i,u) < b^{-1}\lceil bu \rceil,$$

from which (6) follows using $0 < u \leq i < 1$.

Monotonicity of Ψ: Next we show that

$$\Psi(l,i,u) < \Psi(l,i',u) \quad \text{for } i < i', l < u, l < 1 \text{ and } u > 0 \tag{8}$$

using induction on $\max(d_b(l), d_b(u))$.

If $\max(d_b(l), d_b(u)) = 0$ we can conclude $\Psi(l,i,u) = i < i' = \Psi(l,i',u)$. For the induction step let us assume the statements are true when $\max(d_b(l), d_b(u)) = n$. And let $\max(d_b(l), d_b(u)) = n+1$. We consider 6 separate cases:[12]

- Case $i < i' \leq l < u$: Then

$$\begin{aligned}\Psi(l,i,u) &= \mu_l^{-1}(\Psi(\mu_l(l), i, \mu_l(u))\\ &< \mu_l^{-1}(\Psi(\mu_l(l), i', \mu_l(u))) = \Psi(l,i',u)\end{aligned}$$

  where the equalities are by definition and the inequality follows from the induction hypothesis.
- Case $i \leq l < i' < u$: Note that $l < i'$ implies $\lfloor bl \rfloor < bi'$ which implies $\lfloor bl \rfloor + 1 \leq bi'$ since both sides of the inequality are integer. Using (6) we can now conclude: $\Psi(l,i,u) = \mu_l^{-1}(\Psi(\mu_l(l), i, \mu_l(u))) < b^{-1}(1 + \lfloor bl \rfloor) \leq b^{-1}(bi') = i' = \Psi(l,i',u)$.
- Case $i \leq l < u \leq i'$: If $\lfloor bl \rfloor = \lceil bu \rceil - 1$, we have $\mu_l(x) = \nu_u(x)$ and the result follows using the induction hypothesis analogous to the first case. Otherwise, $\lfloor bl \rfloor < \lceil bu \rceil - 1$ and thus $\lfloor bl \rfloor + 1 \leq \lceil bu \rceil - 1$, since both sides of the inequality are integer. We can use (6) arriving at

$$\begin{aligned}\Psi(l,i,u) &= \mu_l^{-1}(\Psi(\mu_l(l), i, \mu_l(u)))\\ &< b^{-1}(1 + \lfloor bl \rfloor) \leq b^{-1}(\lceil bu \rceil - 1)\\ &< \nu_u^{-1}(\Psi(\nu_u(l), i', \nu_u(u))) = \Psi(l,i',u).\end{aligned}$$

- Case $l < i < i' < u$: We have $\Psi(l,i,u) = i < i' = \Psi(l,i',u)$.
- Case $l < i < u \leq i'$: We have $i < u$ implying $bi < \lceil bu \rceil$ and since both sides of the inequality are integer. We have $bi \leq \lceil bu \rceil - 1$, hence using (6) we can conclude $\Psi(l,i,u) = b^{-1}(bi) \leq b^{-1}(\lceil bu \rceil - 1) < \nu_u^{-1}(\Psi(\nu_u(l), i', \nu_u(u))) = \Psi(l,i',u)$.
- Case $l < u \leq i < i'$: Can be shown using the induction hypothesis analogous to the first case.

Stability of Ψ: Next we show that

$$\begin{aligned}l \leq l' &< \Psi(l,i,u) < u' \leq u\\ &\to \Psi(l,i,u) = \Psi(l',i,u')\end{aligned} \tag{9}$$

using induction on $\max(d_b(l), d_b(u))$. In the case where $\max(d_b(l), d_b(u)) = 0$, we can conclude $\Psi(l,i,u) = i$ and thus $l' < i < u'$ which implies $\Psi(l',i,u') = i$. For the induction step let us assume the statements are true if $\max(d_b(l), d_b(u)) = n$. And let $\max(d_b(l), d_b(u)) = n+1$. We consider the three cases from the definition of $\Psi(l,i,u)$:

- Case $l < i < u$: Then we have $\Psi(l,i,u) = i$ and hence $l' < i < u'$ due to the assumption which implies $\Psi(l',i,u') = i$.
- Case $i \leq l$: Then we have $\lfloor bl \rfloor = \lfloor bl' \rfloor$ since $\lfloor bl' \rfloor \leq bl' < b\Psi(l,i,u) = \Psi(\mu_l(l), i, \mu_l(u)) + \lfloor bl \rfloor < \lfloor bl \rfloor + 1$ which implies $\lfloor bl' \rfloor \leq \lfloor bl \rfloor$ since the left and right hand sides are integers. On the other hand $\lfloor bl \rfloor \leq \lfloor bl' \rfloor$ follows directly from $l \leq l'$, i.e., $\lfloor bl \rfloor = \lfloor bl' \rfloor$. Using that we can rely on the induction hypothesis, to conclude $\Psi(l,i,u) = \Psi(l',i,u')$.
- Case $u \leq i$: Then we have $\lceil bu \rceil = \lceil bu' \rceil$, since $\lceil bu' \rceil \geq bu' > b\Psi(l,i,u) = \Psi(\nu_u(l), i, \nu_u(u)) + \lceil bu \rceil - 1 > \lceil bu \rceil - 1$ which implies $\lceil bu' \rceil \geq \lceil bu \rceil$ since the left and right hand sides are integers. On the other hand $\lceil bu' \rceil \leq \lceil bu \rceil$ follows directly from $u' \leq u$, i.e., $\lceil bu' \rceil = \lceil bu \rceil$. Like in the previous case, we can rely on the induction hypothesis to conclude $\Psi(l,i,u) = \Psi(l',i,u')$.

We have already shown that Ψ fulfills Condition 1 in (7). To show Condition 3, let us assume $l' < \Psi(l,i,u) < u'$ and $l < \Psi(l',i',u') < u$ then:

$$\begin{aligned}\Psi(l,i,u) &= \Psi(\max(l',l), i, \min(u',u))\\ &< \Psi(\max(l',l), i', \min(u',u)) = \Psi(l',i',u')\end{aligned}$$

where the inequality follows from (8) and the equalities from (9). □

It is easy to extend the previous result to arbitrary totally ordered finite identifier sets:

Proposition 2 *Let $\mathcal{I}$ be a finite totally ordered set. Then there exists a totally ordered set $\mathcal{A}$ and a function $\Psi : \mathcal{A} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{A} \cup \{\dashv\} \to \mathcal{A}$ fulfilling conditions 1 and 3.*

Proof Let $b = |\mathcal{I}| + 1$ and $\mathcal{I}' := \{\frac{1}{b}, \ldots, \frac{b-1}{b}\}$, then we can define a strict monotone function between $\mathcal{I}$ and $\mathcal{I}'$

$$\phi(x) = \frac{\left|\{y \in \mathcal{I} \middle| y < x\}\right| + 1}{b}.$$

[12] These are all possible ways i and i' can be in the ranges:

1. smaller or equal to l,
2. strictly between l and u, or
3. larger or equal to u.

For two arbitrary variables, there would be nine cases, but since $i < i'$, three of those are eliminated.

By Proposition 1, there is a totally ordered set $\mathcal{A}$ and a function $\Psi' : X \to \mathcal{A}$ fulfilling conditions 1 and 3, where $X = \{(l,i,u) | l < u \in \mathcal{A}, i \in \mathcal{I}\} \subset \mathcal{A} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{A} \cup \{\dashv\}$. We can define $\Psi(l,i,u) = \Psi'(l,\phi(i),u)$ when $l < u$ and arbitrarily[13] set $\Psi(l,i,u) = 1/b$ if $l \geq u$. □

Proposition 3 *Let $\mathcal{I}$ be a totally ordered set. Then there exists a totally ordered set $\mathcal{A}$ and a function $\Psi : \mathcal{A} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{A} \cup \{\dashv\} \to \mathcal{A}$ fulfilling conditions 1 and 3.*

Proof To extend the result from Proposition 2 to the infinite case, we use the compactness theorem [16, §2.1]: Let us assume that $\mathcal{I}$ is an infinite totally ordered set. We introduce a language $\mathcal{L}$ with two constant symbols $\vdash$ and $\dashv$, one relation symbol $<$ and an infinite set of 2-ary function symbols Ψ_i for each element of $\mathcal{I}$. Consider the following infinite set of first order sentences:

$$\forall a, b \,.\, a < b \to \neg a = b \tag{10}$$
$$\forall a, b \,.\, a < b \to \neg b < a \tag{11}$$
$$\forall a, b \,.\, a = b \to \neg a < b \tag{12}$$
$$\forall a, b, c \,.\, a < b \wedge b < c \to a < c \tag{13}$$
$$\forall a, b \,.\, a < b \vee b < a \vee a = b \tag{14}$$
$$\forall a \,.\, (\vdash < a \vee \vdash = a) \wedge (a < \dashv \vee a = \dashv) \tag{15}$$
$$\forall l, u \,.\, l < u \to l < \Psi_i(l,u) < u \text{ for all } i \in \mathcal{I} \tag{16}$$
$$\forall l, u, l', u' \,.\, l' < \Psi_i(l,u) < u' \wedge l < \Psi_{i'}(l',u') < u \to \Psi_i(l,u) < \Psi_{i'}(l',u') \text{ for all } i < i' \in \mathcal{I} \tag{17}$$

The sentences (10-15) express that $<$ is a strict total order and that the constants $\vdash$ (resp. $\dashv$) are the smallest (resp. largest) elements in that order. Note that (16) and (17) constitute infinite sets of first order sentences. If the set of sentences has a model, it is easy to see that there is a totally ordered set $\mathcal{A}$ and a function Ψ fulfilling Condition 1 and 3. (Note that $\mathcal{A}$ would be identified with the elements of the model excluding the values associated to $\vdash$, $\dashv$ and $\Psi(l,i,u)$ would be the value associated with the value of $\Psi_i(l,u)$). To show that (10-17) have a model, we rely on the compactness theorem, which is asking us to check whether any finite subset of those sentences have a model. For such a finite subset F, there will be a finite subset I of $\mathcal{I}$ such that the sentences (10-15) and the restriction of (16) and (17) to the sentences associated with the identifiers in I are a superset of F. For I we can rely on the previous result on finite identifier spaces to find a model, which implies using the compactness theorem, that this theorem is true even if the identifier space is infinite. □

Theorem 1 *Let $\mathcal{I}$ be a totally ordered set. Then there exists a totally ordered set $\mathcal{A}$ and a function $\Psi : \mathcal{A} \cup \{\vdash\} \times \mathcal{I} \times \mathcal{A} \cup \{\dashv\} \to \mathcal{A}$ fulfilling conditions 1, 2 and 3.*

Proof Because of Proposition 3, we only need to show that Condition 2 is true. Let $l, l' \in \mathcal{A} \cup \{\vdash\}$ and $u, u' \in \mathcal{A} \cup \{\dashv\}$ such that $l < u, l' < u'$ and:

$$\Psi(l,i,u) = \Psi(l',i',u') \tag{18}$$

We show that $i = i'$ by contradiction. Let us hence assume that $i \neq i'$, which implies either $i < i'$ or $i > i'$ since $\mathcal{I}$ is totally ordered. We can then infer using Condition 1 and (18) that l, l', i, i', u, u' fulfill the premise of Condition 3 and hence:

- $i < i'$ would imply $\Psi(l,i,u) < \Psi(l',i',u')$
- $i > i'$ would imply $\Psi(l,i,u) > \Psi(l',i',u')$

Both conclusions are in conflict with (18). Hence, the assumption that $i \neq i'$ must have been false. □

5 Strong eventual consistency of WOOT

Section 3 gave an informal derivation of the WOOT Framework and its consistency as a sequence of simulation arguments. We identified that Theorem 1 should imply that WOOT is strongly eventually consistent. However, a rigorous proof of the implication requires a large number of lemmas and definitions.[14] We decided to use Isabelle/HOL to carry out the rigorous proof to avoid subtle flaws in the arguments. The resulting machine-checked proof [10] was open-sourced to the Archive of Formal Proofs (AFP)[1].

In this section, we describe the exact distributed execution model and the results we have verified in Isabelle/HOL and give a brief overview of the proof. Definitions, assumptions and theorems are accompanied by footnotes that reference the corresponding entities in the formalized proof.

5.1 Distributed system

To rigorously express the consistency properties, we need to formally define the distributed execution model used in the proof. We followed the modelling laid out by Gomes et al. [8] and Raynal [24, Chapter 6] for distributed message-passing algorithms, but refined it for the case of the WOOT framework. Let $\mathcal{P}$ be a finite set of peer identifiers.[15] Similarly to Shapiro et al. [27, §2], we assume the participating peers are non-Byzantine. We model an execution of the framework as sequences of events for each peer,[16] where an event can either be a broadcast event and or the reception of a message,

[13] Since the antecedents of both conditions 1 and 3 imply $l < u$.

[14] We needed more than 100 lemmas and 60 definitions across 66 pages in [10].

[15] **assumes** *fin_peers* [10, §4.7]

[16] **fixes** *events* [10, §4.7]

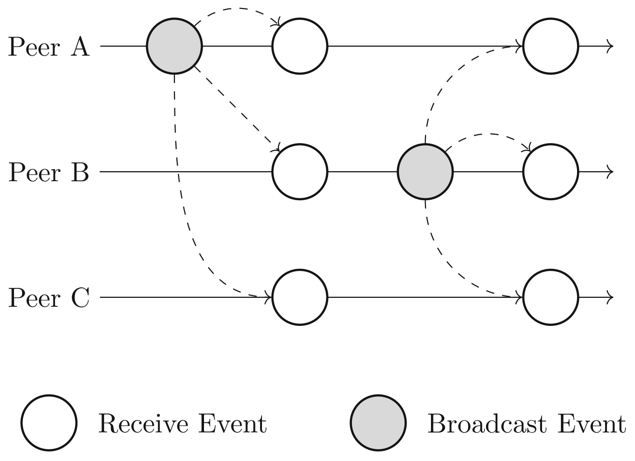


Fig. 6 Example execution of the framework

and we call this sequence the *history* of the peer $h(p)$.[17] We write $h(p)_i = \text{broadcast}(m)$ if the i-th event of peer p was broadcasting the message m and $h(p)_i = \text{receive}(m, q, j)$ if it was the reception of the message m broadcast at the j-th event of peer q. Figure 6 provides an example of such histories. Indices for histories start from 0 and we will use the notation $|h(p)|$ for the number of events.

An event can be uniquely identified by a pair comprised of a peer identifier $p \in \mathcal{P}$ and its index in the history of that peer; we call such a tuple an *event id*.[18]

In Raynal [24, Chapter 7], peers can also have internal events and, instead of broadcast events that disseminate a message to all peers, messages are directed to individual peers. We omit these cases for simplicity. Another difference is in the assumption that all messages are distinct. Since in the WOOT framework it is possible for two peers to send the same message,[19] we avoid that requirement and instead use 3-tuples for receive events, capturing the event index and peer the message originated from. In this way, the links between a broadcast event and its corresponding reception events are still represented.

Note that while we assume event indices count successive events for the same peer, there is no synchronicity assumption between indices from distinct peers. The only ordering of event ids between peers is induced by the causality implied by message transmission. To that end, we introduce a relation on the event ids[20], the *happened-immediately-before* relation $\rightarrow_{\text{hib}}$.[21] The relation $(p, i) \rightarrow_{\text{hib}} (q, j)$ holds if $i < |h(p)|$, $j < |h(q)|$ and either:

- $p = q$ and $j = i + 1$, i.e., they are successive events on the same peer, or
- there exists m such that $h(q)_j = \text{receive}(m, p, i)$, i.e., the latter event is the reception of a message sent by the former event.

The transitive closure of $\rightarrow_{\text{hib}}$ is the *happened-before* relation $\rightarrow_{\text{hb}}$, which was introduced by Lamport [13] to order events of asynchronous distributed systems.

Histories which describe an actual execution of a distributed algorithm fulfill additional conditions. For example, a peer can only receive a message from another peer if the latter broadcast that message. We summarize the assumptions about histories of a distributed system in the following condition:

Condition 4 (Distributed Execution)

- If a message m was received from peer q event j, that event must be a broadcast event, e.g., if $h(p)_i = \text{receive}(m, q, j)$ then $h(q)_j = \text{broadcast(m)}$.[22]
- A broadcast event will deliver a message to each peer at most once, i.e., for all p and $i, j < |h(p)|$, if there exists m, q, k such that $h(p)_i = h(p)_j = \text{receive}(m, q, k)$ then $i = j$. In practice, if the network communication mechanism does not guarantee at-most-once delivery, this condition can also be simulated by keeping track of all received messages in the implementation. Note that we do not yet require that a message will be received at all by all peers.[23]
- The happened-immediately-before relation $\rightarrow_{\text{hib}}$ is acyclic. This is equivalent to the fact that the happened-before relation $\rightarrow_{\text{hb}}$ is a strict partial order. This condition about the execution of distributed systems is a consequence of the fact that the distributed system runs on physical machines and that events cannot cause themselves. See for example Lamport [13, §The Partial Ordering].[24]

We will write $R(p, i)$[25] to denote the set of messages received by a peer before event i, i.e.,

$$R(p, i) := \left\{m \middle| \exists q, k, j < i. h(p)_j = \text{receive}(m, q, k)\right\}$$

[17] **datatype** *('p, 's) event*

[18] **type_synonym** *'p event_id*

[19] This happens when two peers delete the same character concurrently.

[20] More commonly in other work, this relation is defined between events. However because of the assumptions that messages are not distinct, and thus it is possible to have the same broadcast event multiple times, we instead define the relation on event ids.

[21] **fun** *happened_immediately_before* [10, §4.7]

[22] **assumes** *send_correct* [10, §4.7]

[23] **assumes** *at_most_once* [10, §4.7]

[24] **assumes** *acyclic_happened_before* [10, §4.7]

[25] **fun** *received_messages* [10, §4.7]

For results where we need to assume that all broadcast messages will be delivered to each peer, we introduce the eventual delivery condition:

Condition 5 (Eventual Delivery) A message will be delivered to all peers, i.e., for all $p, q \in \mathcal{P}$ and $j < |h(p)|$, if $h(p)_j = \text{broadcast}(m)$ then $m \in R(q, |h(q)|)$.

The following condition expresses the semantic delivery condition we introduced in Sect. 3.5, i.e., a message is not received before its dependencies are. If the underlying communication protocol does not meet this condition, an implementation can buffer messages until their dependencies are received. See Protocol 5 for an example implementation of such an algorithm.

Condition 6 (Semantic Causal Delivery) A message will only be delivered to a peer if its dependencies have already been delivered to it, i.e., for all $p \in \mathcal{P}$ and $j < |h(p)|$, if $h(p)_j = \text{receive}(m, q, k)$ then for each $i' \in \text{deps}(m)$ there exists an $m' \in R(p, j)$ such that $i(m') = i'$. Here we refer to the function deps[26] which is defined in Protocol 5.[27]

In addition to events, we also associate a sequence $s(p)$ of states to each peer $p \in \mathcal{P}$. Similarly to the notation for history, we write $|s(p)|$ for the number of states and index the states starting from 0. Whenever a peer receives a message, it will update its state using the corresponding integration algorithm for the type of message it received as described in Protocol 6. Similarly, a broadcast message is created using one of the modify algorithms described in Protocol 6.

To be able to express properties about the histories of states, we represent the integration and modification algorithms in Protocol 6 as mathematical functions. For the integration algorithms, if m is a message and s a preceding state then $\text{integrate}(m, s)$[28] returns the new state after integrating the message into state s. Depending on the type of message, this is the result of applying the algorithm **integrate insert** or **integrate delete**. In cases where the integration algorithm fails or does not terminate, the resulting state is $\bot$ and we define $\text{integrate}(m, \bot) = \bot$.

Both modification algorithms **modify insert** and **modify delete** read but do not modify the state and their last statement is a broadcast. We introduce functions that return the message that would be broadcast by them. Given a state s, we define the function $\text{create-insert}(i, \sigma, k, s)$,[29] which returns the message that would be broadcast by the algorithm **modify insert**(σ, k) if the state of the peer were s and the unique id returned by the **create-unique-id** function were i. Similarly, $\text{create-delete}(k, s)$[30] returns the message that would be broadcast by **modify delete**(k) if the state of the peer were s.

In the following, we summarize the conditions that express that each peer implements Protocol 6.

Condition 7 (Peers execute WOOT Protocol)

1. The number of states is exactly one larger than the number of events, i.e., $|s(p)| = |h(p)| + 1$. This is because we use $s(p)_0$ for the initial state of a peer, before any event has happened on the peer.
2. Each peer's initial state is the empty string, i.e., $s(p)_0 = \varepsilon$.
3. If a peer receives a message, the resulting state is the output of applying the integration algorithm for that message on the previous state, i.e., $s(p)_{i+1} = \text{integrate}(m, s(p)_i)$ if there exist m, q, j such that $h(p)_i = \text{receive}(m, q, j)$.
4. In the case of a broadcast event, the state of the peer remains the same, i.e., $s(p)_{i+1} = s(p)_i$ if there exists m such that $h(p)_i = \text{broadcast}(m)$. Deferring the update of the state allows us to simplify the correctness proof: we can model the broadcast event as transmitting the message to all peers including the source peer itself; its state will be updated when it receives its own message. An actual implementation would usually introduce a separate code path to update the peer's own state. See also Fig. 6 and Sect. 3.1.[31]
5. In the case of a broadcast event, either the message was created by applying the **modify insert** or **modify delete** algorithm on the state $s(p)_i$, i.e., if $h(p)_i = \text{broadcast}(m)$ then $s(p)_i \neq \bot$ and either:
   - there exists $n < |s(p)_i| + 1$ and $\sigma \in \Sigma$ such that $m = \text{create-insert}((p, i), n, \sigma, s(p)_i)$, or
   - there exists $n < |s(p)_i|$ such that $m = \text{create-delete}(n, s(p)_i)$.

   Note that, this means we are assuming a peer which reached the failure state will not broadcast any more messages.[32]

We would like to note a simplification made regarding the unique identifier. In the description of Protocol 6, we define the **create-unique-id** procedure to be an arbitrary algorithm that returns unique identifiers. In the above conditions, we use the combination of peer id and event index as the unique id for newly created characters (see Condition 7 Clause 5). This is a valid implementation for **create-unique-id**, but implementations could of course choose other methods to create unique identifiers.

[26] **fun** *deps* [10, §4.7]

[27] **assumes** *semantic_causal_delivery* [10, §4.7]

[28] **fun** *integrate* [10, §4.6]

[29] **fun** *create_insert* [10, §4.5]

[30] **fun** *create_delete* [10, §4.5]

[31] **fun** *state* [10, §4.7]

[32] **assumes** *send_correct* [10, §4.7]

5.2 Results

With the definitions in this section, we have verified the following two theorems using the Isabelle/HOL interactive theorem prover.

Theorem 2 *(No failure*[33]*) During the distributed execution of the WOOT framework with semantic causal delivery, the integration algorithms will never fail, i.e., if the conditions 4, 6 and 7 are met, then for all $p \in \mathcal{P}$ and $i < |s(p)|$ we have $s(p)_i \neq \bot$.*

Proof Verified in [10, §6]. □

We recall the definition of Strong Convergence from Sect. 1 and Shapiro et al. [27, §2.2]:

Definition 1 A CRDT is *strongly convergent* if any pair of peers who have received the same set of messages will be in equal states, i.e., for all $p, q \in \mathcal{P}, i < |h(p)|, j < |h(q)|$, if $R(p, i) = R(q, j)$ then $s(p)_i = s(q)_j$.

Theorem 3 *(Strong Convergence*[34]*) During the execution of the WOOT framework with semantic causal delivery, if two peers have received the same set of messages they will be in the same state, i.e., if the conditions 4, 6 and 7 are met then for all $p, q \in \mathcal{P}, i < |h(p)|, j < |h(q)|$ if $R(p, i) = R(q, j)$ then $s(p)_i = s(q)_j$.*

Proof Verified in [10, §6]. □

We recall the definition of Eventual Consistency from Sect. 1 and Shapiro et al. [27, §2.2]:

Definition 2 A CRDT is *eventually consistent* when, if after some point no further updates are made at any peer, then the peers will eventually reach the same final state, i.e., there is a state s such that for all $p \in \mathcal{P}$: $s(p)_{|s(p)|-1} = s$.

Except for trivial cases, an operation-based CRDT can only be eventually consistent if all messages are actually delivered. On the other hand, strong convergence can be proved even without that assumption. If we additionally assume eventual delivery, we can conclude that the WOOT framework is eventually consistent:

Corollary 1 *If the WOOT framework is executed with semantic causal delivery and eventual delivery, then it is eventually consistent, i.e., if the conditions 4, 5, 6 and 7 are met, then there is a state s such that for all $p \in \mathcal{P}$: $s(p)_{|s(p)|-1} = s$.*

Proof Note that $|h(p)| = |s(p)| - 1$. Under the assumption of eventual delivery, we can prove that all peers will eventually have received the same set of messages, i.e., $R(p, |h(p)|) = R(q, |h(q)|)$ for all $p, q \in \mathcal{P}$.

We start by proving that $R(p, |h(p)|) \subseteq R(q, |h(q)|)$. For $m \in R(p, |h(p)|)$, there exist r, i and j such that $\text{receive}(m, r, j) = h(p)_i$. From Condition 4, we have $h(r)_j = \text{broadcast}(m)$, and from Condition 5, we can conclude $m \in R(q, |h(q)|)$. Inclusion in the reverse direction $R(q, |h(q)|) \subseteq R(p, |h(p)|)$ can be proven analogously. Using Theorem 3 we conclude that $s(p)_{|h(p)|} = s(q)_{|h(q)|}$. This is true for any pair $p, q \in \mathcal{P}$, implying the corollary. □

We recall the definition for strong eventual consistency from Sect. 1 and Shapiro et al. [27, §2.2]:

Definition 3 A CRDT is *strongly eventually consistent* if it is eventually consistent and it is strongly convergent.

Theorem 4 *If the WOOT framework is executed with semantic causal delivery and eventually delivery, then it is strongly eventually consistent.*

Proof Follows from Theorem 3 and Corollary 1. □

5.3 Details

In the following, we describe the high-level approach (and the reasoning behind it) to prove Theorems 2 and 3 in our formalization [10] using Isabelle/HOL.

Usually, eventual consistency of a CRDT can be proven by checking that any pair of operations commute, i.e., given two messages, the successive integration of them to a given starting state will lead to the same resulting state irrespective of the order of integration. In that scenario, a general theorem about CRDTs (see Shapiro et al. [27, Theorem 2.2]) implies strong eventual consistency for the CRDT at hand.

In some cases, such as in the WOOT framework or RGAs, integration operations are partial functions where the state needs to satisfy a precondition to enable integration of a received message. An integration operation will *fail* if the precondition is not satisfied when it is invoked. For such CRDTs, in addition to consistency, we also need to prove such failures will not occur during the execution of the framework. For example, Gomes et al. [8] establish that the insert operation for RGAs will not fail because the dependency of an element will already be in the array as a consequence of causal delivery.

Interestingly, a proof of non-failure for WOOT must necessarily establish consistent ordering of characters between the participating peers. This can be seen by considering examples where two peers that have a permutation of each other's state will not be able to create messages that can be successfully integrated into the other peers state. However, invariants that imply consistent ordering of the characters between peers can be found. This makes it more favorable in

[33] **theorem** *no_failure* [10, §6]

[34] **theorem** *strong_convergence* [10, §6]

the case of WOOT to directly show consistency from these established invariants, instead of verifying commutativity of operations.

To describe the invariant we establish during the execution of the WOOT framework, we first define the notion of *consistent* sets of messages. Let $\mathcal{A}$ be a sort key space and let $\Psi : (\mathcal{A} \cup \{\vdash\}) \times \mathcal{I} \times (\mathcal{A} \cup \{\dashv\}) \to \mathcal{A}$ be a map fulfilling conditions 1, 2 and 3. We have seen in Sect. 4 that such a space and function exists. Then we can define consistent sets of messages:

Definition 4 (Consistent Sets of Messages[35]) Let M be a set of messages, consisting of the insert messages M_{insert} and delete messages M_{delete}. We say such a set of messages is consistent, if:

1. Each message has a distinct identifier, i.e., i is injective on M_{insert}.
2. The dependencies of each message in the set are met, i.e., $\text{deps}(m) \subseteq \left\{i(m) \middle| m \in M_{\text{insert}}\right\}$ for all $m \in M$.
3. Let $\rightarrow_{\text{dep}}$ be the relation[36] induced by the deps function on the insert messages, i.e., for $m_1, m_2 \in M_{\text{insert}}$, $m_1 \rightarrow_{\text{dep}} m_2$ iff $i(m_1) \in \text{deps}(m_2)$. This relation $\rightarrow_{\text{dep}}$ is well-founded.
4. There exists a function α from the identifiers in M_{insert} to the sort key space $\mathcal{A}$, i.e., $\alpha : i(M_{\text{insert}}) \to \mathcal{A}$, such that:

$$\begin{aligned}&\alpha(\vdash) < \alpha(\dashv),\\ &\alpha(l(m)) < \alpha(u(m)) \text{ and}\\ &\alpha(i(m)) = \Psi\left(\alpha(l(m)), i(m), \alpha(u(m))\right)\end{aligned}$$

   for all $m \in M_{\text{insert}}$.

Roughly, a set of messages is consistent if it would be possible to inductively associate sort keys to each insert message (i.e., created character) according to the scheme described in Protocol 4. See also Fig. 3.

In addition to that, we introduce a relation between sets of messages and states of peers. The state of a peer consists of characters whose symbols are replaced by $\perp$ if they were deleted but are otherwise identical to the corresponding insert message. The state corresponding to a set of insert messages is a sequence of such characters, with the ordering induced by the sort key function. More precisely:

Definition 5 (Association[37]) Let M be a consistent set of messages, where $M = M_{\text{insert}} \cup M_{\text{delete}}$, and s be a sequence of characters. Let d be a function defined on M_{insert} by:

$$d(l, i, u, \sigma) = \begin{cases}(l, i, u, \sigma) & \text{if } i \notin M_{\text{delete}}\\ (l, i, u, \perp) & \text{otherwise.}\end{cases}$$

Then we say s and M are *associated* if:

- M_{insert} and s represent the same set of characters, up to possible substitutions of symbols with $\perp$ due to delete messages, i.e., $|s| = |M_{\text{insert}}|$ and

$$\{s_1, \cdots, s_{|s|}\} = \{d(m) | m \in M_{\text{insert}}\}.$$

- For all α fulfilling the conditions of Clause 3 of Definition 4, the sequence $\alpha(i(s_1)), \cdots, \alpha(i(s_{|s|}))$ is strictly increasing.

Observe that, given a consistent set of messages, there can be at most one state associated to it.[38] A key result we establish is that the integration algorithm commutes with set insertions under that relation:

Proposition 4 *If both M and $M \cup \{m\}$ are consistent sets of messages, s is the state associated to M, and either:*

- *$m \notin M$, or*
- *m is a delete message*

then integrate(m, s) is the state associated to $M \cup \{m\}$.[39]

Proof Verified in [10, §5.6]. □

The arguments of the proof rely on the insights we established in Sect. 3.8. Note that the above result in particular implies that the integration algorithm will not fail. Since the starting state (the empty sequence) is associated with the empty set,[40] it is possible to prove using induction and Proposition 4 that, with the semantic causal delivery condition, a peer that receives a consistent set of messages will be in the state associated to the received set.

Having established that, we proceed [10, §5.7] by proving that all the messages broadcast during the execution of the WOOT framework are a consistent set, using induction according to some causal ordering of the events.[41] It should be noted that during induction, we use Proposition 4 and thus use the fact that the messages broadcast so far, according

[35] **definition** *consistent* [10, §5.3]

[36] **fun** *depends_on* [10, §5.3]

[37] **definition** *is_associated_string* [10, §5.3]

[38] **lemma** *associated_string_unique* [10, §5.3]

[39] **proposition** *integrate_insert_commute* [10, §5.6]

[40] **lemma** *empty_associated* [10, §5.3]

[41] More precisely, since the events are partially ordered according to the $\rightarrow_{\text{hb}}$ relation, we choose an arbitrary total order that is an extension of it. This can be done using topological sorting. In general, there may be many possible such extensions.

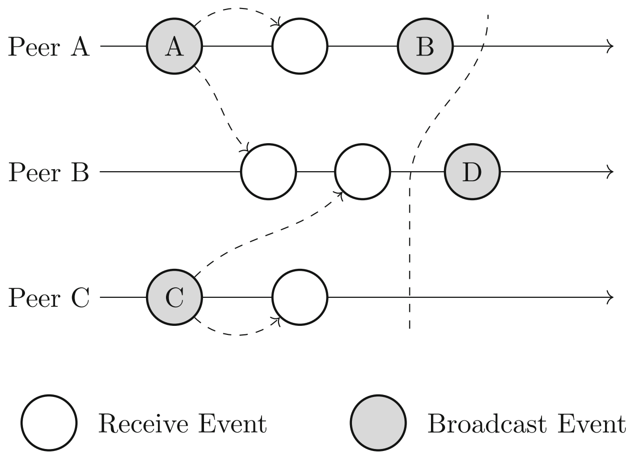


Fig. 7 Example induction step, where the creation of message D keeps the set of messages consistent

to the chosen causal ordering, must be consistent. In Fig. 7 we depict an example induction step: Assuming the set of messages generated by the events left of the dashed line, i.e., the messages A, B and C, are consistent, we want to show that including the message D preserves consistency. Because the induction proceeds according to the happened-before relation, the peer can only have received a subset of the messages—that were already shown to be consistent. In the depicted example, these are A and C. Because of the semantic causal delivery condition, such a subset must itself be consistent. Hence the state of the second peer, when it creates message D is the state associated to the messages A, C. To complete the induction step, we rely on the following proposition:

Proposition 5 *Let M be a consistent set of messages and N be a consistent subset of M and let s be the state associated to N and m be a message, such that either*

- *there exists $n < |s| + 1$, $\sigma \in \Sigma$ and $m = \textit{create-insert}(i, n, \sigma, s)$, where i is an identifier distinct from all identifiers in M, or*
- *there exists $n < |s|$ and $m = \textit{create-delete}(n, s)$.*

Then $M \cup \{m\}$ is consistent.[42]

Proof Verified in [10, §5.4]. □

6 Formalization in Isabelle/HOL

This section gives a brief overview of the machine-checked proof [10] we open-sourced in the Archive of Formal Proofs (AFP)[1]. The AFP is a refereed publication containing formal documents verified by Isabelle. Contrary to ordinary publications, it is being updated with each release of Isabelle, and results are always checked with its most recent release. Authors can improve and add additional content to their entries provided the updates can be verified. All prior versions are accessible. Another distinctive feature of the AFP is that it allows entries to be used as a library, i.e., an entry can depend and use results established by previously published entries.

Our entry uses the Certification Monads [28] library to express partial functions. These are used for example to handle illegal indices during array lookups and missing identifiers during find operations in sequences, or to capture non-termination cases.[43] Partial functions return an error result in these cases, which can then be propagated. Hence, when we prove that an algorithm will not return an error state, it implies that such runtime errors will not happen.

We also use the Data Type Order Generator [29] library to automatically derive total orders for data types, for example during the construction of the sort key space.

We organized the AFP entry such that all necessary definitions are summarized in Sections 1 to 4 (18 pages) with thorough explanations. Section 5 (36 pages) contains the actual proof, but readers who are only interested in the results can skip it.[44] The resulting Theorems 2 and 3 of this document are presented in Section 6 of the AFP entry.

As mentioned before, Sect. 5 of this document includes footnotes that link the definitions and assumptions to the corresponding definitions and assumptions in the AFP entry. We would also like to refer to the documentation of Isabelle/HOL [18] for an introduction to its semantics and syntax.

In the following subsections, we mention notable methods we used while formalizing the proof, including notions that are uncommon in standard formal mathematics such as type parameters or sum types.

6.1 The function Ψ

Proposition 3 is represented using 2 propositions in our AFP entry [10, §5.1]:

- **proposition** *psi_elem*
- **proposition** *psi_preserve_order*

[42] **lemma** *create_insert_consistent* and **lemma** *create_delete_consistent* [10, §5.4]

[43] This may happen during the **while**-loop in the integration algorithm for insert messages in Protocol 6, where we return an error if $u - l$ does not decrease during an iteration.

[44] Any intermediate definitions within Section 5 are not (neither directly nor indirectly) used in the statement of the resulting theorems.

Instead of using the compactness theorem, we prove the propositions constructively. It was however not easy to present that version of the proof in Sect. 4 as it uses case distinctions with more than 25 separate cases. In Isabelle, the case distinctions result in goals that are automatically resolved. We would like to note that the existence of a constructive version is interesting for possible implementations of Protocol 4.

6.2 Types

Since Isabelle's type system allows the construction of new types based on existing types, we use that mechanism to abstract over the set of symbols and identifiers. Type parameters are indicated using a prime prefix, and type constructors are suffix operators in Isabelle. For example the type of a WOOT character is:

('$\mathcal{I}$, 'Σ) woot_character
where '$\mathcal{I}$ denotes the type of identifiers and 'Σ denotes the type of symbols. Then the type of a list of characters, i.e., a state of a peer is:

('$\mathcal{I}$, 'Σ) woot_character list
In cases where we use special elements such as $\vdash$, $\dashv$ or $\perp$, the representation in Isabelle uses a sum type. For example, when to include the special state $\perp$ denoting the failure of the integration algorithm, the formalization uses the sum type:

error + ('$\mathcal{I}$, 'Σ) woot_character list
This means the result of the *integrate* function is either an *error* or a sequence of characters. To extend types with the special $\vdash$ and $\dashv$ elements, we use the type constructor *extended*, defined by:

dataype '$\mathcal{I}$ extended
= Begin ($\vdash$)
| InString '$\mathcal{I}$ ((1[[-]]))
| End ($\dashv$)

We can read this as an element of '$\mathcal{I}$ *extended* is either $\vdash$, $\dashv$ or $[\![i]\!]$ where i is an element of '$\mathcal{I}$. The terms in parenthesis, such as $\vdash$ and $\dashv$, denote abbreviations; they make the formal document more concise and closer to the notation in this manuscript.

6.3 Locales

Locales are parametric theories, where a number of theorems can be shown for a common set of assumptions and definitions. A good use case for locales are algebraic structures such as groups, rings or fields. Locales can extend each other, for example the locale for a field would extend the locale for a ring.

We use locales to model the distributed system. In particular, we fix a finite set of peers and history of events, and establish their assumptions, such as semantic causal delivery or the absence of causal cycles. For technical reasons, we use two locales:

- *dist_execution_preliminary*
- *dist_execution*

The first one establishes the assumption that the set of peers is finite and introduces definitions such as the sequence of received messages for a given state:

fun *received_messages* **where**
received_messages (i,j) =
[m. (*Receive* _ m) $\leftarrow$ (*take j* (*events i*))]

This corresponds to our definition of R for received message in Sect. 5 (only with the slight difference that messages are returned in the order they were received by the peer). The second locale extends the first locale and introduces the remaining assumptions, as described in Conditions 4, 6 and 7. Interestingly, we could express Clauses 1 to 4 of Condition 7 using a single definition. We recall that we required that the starting state of a peer is the empty sequence and that the state is updated by application of the *integrate* algorithm whenever a message is received. In the Isabelle formalization, we could express these constraints using the *foldM* function:

fun *state* **where**
state i = *foldM integrate* [] (*received_messages i*)

In Section 9 of the AFP entry, we define an example execution of the framework consisting of histories of 3 peers where each peer broadcasts a message, and show that it is an example instance for the *distributed-execution* locale, i.e., that those fulfill the assumptions of it. Note that this implies, that the assumptions are consistent.

7 Conclusion

We have shown that WOOT is strongly eventually consistent. This property was an open conjecture in the original presentation of the framework in 2006. The fact that the general convergence proof was missing had also been mentioned by Kumawat and Khunteta [12, §3.10]. To achieve this result we relied on an association of sort keys to the characters. The proof is verified using the interactive theorem prover Isabelle/HOL.

Having machine-checked our proof, we have strong confidence in its correctness. By open-sourcing our formalization and framework and having it accepted in the Archive of Formal Proofs [10], our work is accessible, reproducible and available in a long-term maintained way to the community.

A key insight we could derive about WOOT is that it can be simulated by a specific instance of a broad class of algorithms, parameterized by the function Ψ (See Protocol 4). We think it is worthwhile to investigate how properties of Ψ are related to properties of the CRDT, for example with respect to interleaving anomalies. This also implies the existence of an implementation by explicitly computing the sort keys. That would allow an integration algorithm with a runtime of $O(n \log n)$ with the same behaviour, compared to the $O(n^2)$ worst-case performance of WOOT (where n is the number of previous insert operations on the document). A second insight we contribute is that the communication-cost of sort-key based protocols could be significantly improved by performing a program transformation that defers some of the computation to the integration site. We think that similar modifications could also improve performance properties of other distributed data structures. It would be interesting to develop a formal theory of how program transformations can be defined on CRDTs and which conditions are required to preserve convergence and consistency properties. We found that the conventional commutativity argument, commonly used to show convergence of CRDTs, does not work with the WOOT framework. Instead we showed consistency using induction over the events of the distributed system. An interesting question for further work is whether there may be stronger fundamental theorems for CRDTs that could apply for WOOT and similar cases.

Acknowledgements Special thanks to Cornelius Diekmann for providing a lot of insightful suggestions and expertise on Isabelle/HOL and to the anonymous reviewers for their careful review and many insightful suggestions and comments.

Declarations

Conflict of Interest The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Archive of Formal Proofs. https://isa-afp.org. Accessed 11 Nov 2021
2. Ahmed-Nacer, M., Ignat, C.-L., Oster, G., Roh, H.-G., Urso, P.: Evaluating CRDTs for real-time document editing. In: Proceedings of the 11th ACM Symposium on Document Engineering, pp. 103–112 (2011)
3. Briot, L., Urso, P., Shapiro, M.: High responsiveness for group editing CRDTs. In: Proceedings of the 19th International Conference on Supporting Group Work, pp. 51–60. ACM, New York (2016)
4. Brown, R., Cribbs, S., Meiklejohn, C., Elliott, S.: Riak DT map: A composable, convergent replicated dictionary. In: Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. ACM, New York (2014)
5. Dallaway, R.: WOOT Model for Scala and JavaScript via Scala.js. https://github.com/d6y/wootjs. Accessed 13 Nov 2021
6. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, vol. 18, pp. 399–407 (1989)
7. Emanouilov, V.: Collaborative Rich Text Editor. https://github.com/kroky/woot. Accessed: 13 Nov 2021
8. Gomes, V.B.F., Kleppmann, M., Mulligan, D.P., Beresford, A.R.: Verifying strong eventual consistency in distributed systems. Proceedings of the ACM on Programming Languages **1**(OOPSLA) (2017). Article 109
9. Kaplan, R.: A Real Time Collaboration Toy Project Based on WOOT. https://github.com/ryankaplan/woot-collaborative-editor. Accessed 13 Nov 2021
10. Karayel, E., Gonzà lez, E.: Strong eventual consistency of the collaborative editing framework WOOT. Archive of Formal Proofs (2020). http://isa-afp.org/entries/WOOT_Strong_Eventual_Consistency.html Formal proof development
11. Kleppmann, M., Gomes, V.B.F., Mulligan, D.P., Beresford, A.R.: Interleaving anomalies in collaborative text editors. In: Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data. ACM, New York (2019). Article 6
12. Kumawat, S., Khunteta, A.: A survey on operational transformation algorithms: challenges, issues and achievements. Int. J. Comput. Appl. **3**(12), 30–38 (2010)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
14. Letia, M., Preguiça, N., Shapiro, M.: Consistency without concurrency control in large, dynamic systems. SIGOPS Oper. Syst. Rev. **44**(2), 29–34 (2010)
15. Li, D., Li, R.: An admissibility-based operational transformation framework for collaborative editing systems. Comput. Supp. Cooper. Work (CSCW) **19**(1), 1 (2010)
16. Marker, D.: Model Theory: An Introduction, edition1st edn. Graduate Texts in Mathematics, vol. 217. Springer (2002)
17. Nédelec, B., Molli, P., Mostefaoui, A., Desmontils, E.: LSEQ: an adaptive structure for sequences in distributed collaborative editing. In: Proceedings of the 2013 ACM Symposium on Document Engineering, pp. 37–46
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283, 1st edn. Springer (2002)
19. Olson, T.: Real Time Group Editor without Operational Transformation. https://github.com/TGOlson/woot-haskell. Accessed 13 Nov 2021
20. Oster, G., Urso, P., Molli, P., Imine, A.: Real time group editors without operational transformation. Technical Report RR-5580, INRIA (2005)
21. Oster, G., Urso, P., Molli, P., Imine, A.: Data consistency for P2P collaborative editing. In: Conference on Computer Supported Cooperative Work (CSCW), pp. 259–268. ACM (2006a)
22. Oster, G., Molli, P., Urso, P., Imine, A.: Tombstone transformation functions for ensuring consistency in collaborative editing systems. In: International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom) (2006b). IEEE

23. Preguiça, N., Marques, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: 29th International Conference on Distributed Computing Systems, pp. 395–403 (2009). IEEE
24. Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer (2013)
25. Roh, H.-G., Jeon, M., Kim, J.-S., Lee, J.: Replicated abstract data types: building blocks for collaborative applications. J. Parall. Distrib. Comput. **71**(3), 354–368 (2011)
26. Sagher, Y.: Counting the rationals. Am. Math. Mon. **96**(9), 823–823 (1989)
27. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Stabilization. Safety, and Security of Distributed Systems, pp. 386–400. Springer (2011)
28. Sternagel, C., Thiemann, R.: Certification monads. Archive of Formal Proofs (2014). https://isa-afp.org/entries/Certification_Monads.html Formal proof development
29. Thiemann, R.: Generating linear orders for datatypes. Archive of Formal Proofs (2012). https://isa-afp.org/entries/Datatype_Order_Generator.html Formal proof development
30. Weiss, S., Urso, P., Molli, P.: Wooki: a P2P wiki-based collaborative writing tool. In: Web Information Systems Engineering – WISE 2007, pp. 503–512. Springer
31. Weiss, S., Urso, P., Molli, P.: Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In: 29th IEEE International Conference on Distributed Computing Systems, pp. 404–412 (2009)