Strong Eventual Consistency of the Collaborative Editing Framework WOOT

Emin Karayel and Edgar Gonzàlez

Google, Mountain View

December 14, 2021

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 a CRDT for collaborative text editing introduced by Oster et al. (CSCW 2006) for which the eventual consistency property was verified only for a bounded model to date. We contribute a formal proof for WOOTs strong eventual consistency.

Contents

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*" [22]. 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 [24].

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 without consensus-based concurrency control [13]. They are used in highly scalable robust distributed applications [26, 3].

An RDT is *eventually consistent* when, if after some point in time no further updates are made at any replica, all replicas eventually converge to equivalent states. It is *strongly eventually consistent* when it is eventually

[1]Note that other authors like Shapiro et al. [24] use CmRDT to refer to Commutative RDTs, with CRDT standing for *Conflict-free RDTs*.

consistent and, whenever any two peers have seen the same set of updates (in possibly different order), they reach equivalent states immediately [24].

The WithOut Operational Transforms (WOOT) Framework [19] was the first proposed CRDT for collaborative text editing [2]. It has been implemented as part of several OSS projects [4, 6, 8, 16]. However, the eventual consistency of WOOT has only been verified for a bounded model [19, 18]. A formal proof of WOOTs consistency can rigorously establish that there is no complex counter-example not identified by model checking.

The contribution of this work is one such proof that the WOOT Framework is strongly eventually consistent. Its central idea is the association of a value from a dense totally ordered space to each inserted (and potentially deleted) character, using a recursive definition with respect to the acyclic graph induced by the predecessor and successor relation of the characters. We then show that the strings in each peer remain sorted with respect to that value, i.e., that the values form a sort key for W-characters.[2] This resolves the conjecture posed by Oster et al. [18, conjecture 1] and is also the key lemma to establish that the WOOT Framework has the strong eventual consistency property.

After reviewing related work in the following section, we formalize the WOOT Framework as a distributed application in Section 4. We follow with the complete eventual consistency proof in Section 5 and summarize the established results in Section 6. In Section 8 we given overview of the proof and follow up with a conrete formalized example in Section 9.

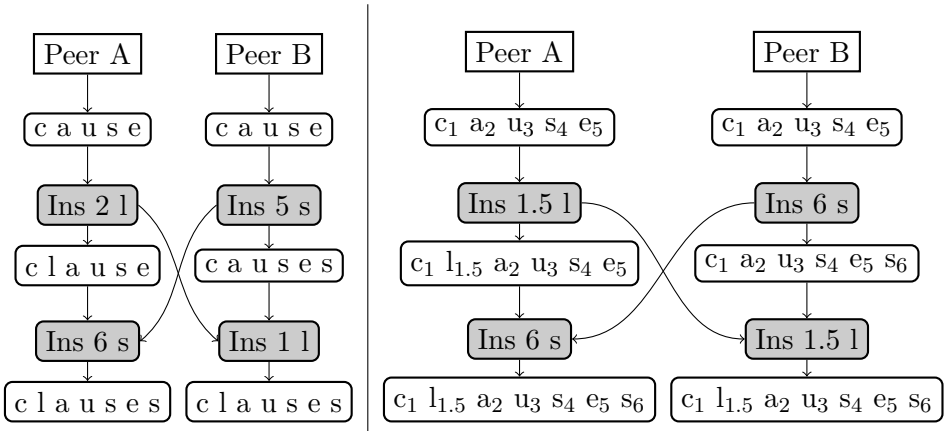
The presentation is structured such that all the definitions necessary to review the established results in Section 6 are part of Section 4. This means it is possible to skip Section 5 entirely.

2 Related Work

The first collaborative text editing tools were based on operational transformations (OT), and introduced by Ellis and Gibbs [5]. The basic idea behind OT-based frameworks is to adjust edit operations, based on the effects of previously executed concurrent operations. For instance, in Figure 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 [14, 17]. Counter-examples have been found in most OT algorithms [22][7, section 8.2].

[2]Note that the values themselves do not have to be actually computed, during the execution of the framework. Their existence and compatibility with the integration algorithm forms a witness for the consistency proof we are presenting.



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

Figure 1: Collaborative text editing

LSEQ [15], LOGOOT [26] and TreeDoc [20] are CRDTs that create and send sort keys for symbols (e.g., 1.5 and 6 in Figure 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.[3] 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. [20] 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. [22]. Contrary to the previous approaches, the identifiers associated to 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 preceeding the new symbol

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

at the time it was created and the actual identifier associated to the new symbol. The integration algorithm starts by finding the preceeding 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. [7] also formalized the eventual consistency property of RGAs using Isabelle/HOL.

In addition to the original design of WOOT by Oster et al. [19], a number of extensions have also been proposed. For instance, Weiss et al. [25] propose a line-based version WOOTO, and Ahmed-Nacer et al. [1] introduce a second extension 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 are no publications that further expand on the correctness of the WOOT Framework. The fact that the general convergence proof is missing is also mentioned by Kumawat and Khunteta [11, Section 3.10].

3 Preliminary Notes

3.1 Algorithms in Isabelle

theory *ErrorMonad*
  imports
    Certification-Monads.Error-Monad
begin

Isabelle's functions are mathematical functions and not necessarily algorithms. For example, it is possible to define a non-constructible function:

fun *non-constructible-function* **where**
  non-constructible-function f = (*if* (∃ *n*. *f n* = *0*) *then 1 else 0*)

and even prove properties of them, like for example:

$$\textit{non-constructible-function}\ (\lambda x.\ \textit{Suc}\ 0) = 0$$

In addition to that, some native functions in Isabelle are under-defined, e.g., [] ! *1*. But it is still possible to show lemmas about these undefined values, e.g.: [] ! *1* = [*a*, *b*] ! *3*. While it is possible to define a notion of algorithm in Isabelle [9], we think that this is not necessary for the purpose of this formalization, since the reader needs to verify that the formalized functions correctly model the algorithms described by Oster et al. [19] anyway. However, we show that Isabelle can generate code for the functions, indicating that non-constructible terms are not being used within the algorithms.

type-synonym *error* = *String.literal*

```
fun assert :: bool ⇒ error + unit
  where
    assert False = error (STR ''Assertion failed.'') |
    assert True = return ()

fun fromSingleton :: 'a list ⇒ error + 'a
  where
    fromSingleton [] = error (STR ''Expected list of length 1'') |
    fromSingleton (x # []) = return x |
    fromSingleton (x # y # ys) = error (STR ''Expected list of length 1'')
```

Moreover, we use the error monad—modelled using the *sum* type—and build wrappers around partially defined Isabelle functions such that the evaluation of undefined terms and violation of invariants expected by the algorithms result in error values.

We are able to show that all operations succeed without reaching unexpected states during the execution of the framework.

end

4 The WOOT Framework

```
theory Data
  imports Main Datatype-Order-Generator.Order-Generator
begin
```

Following the presentation by Oster et al. [19] we describe the WOOT framework as an operation-based CRDT [24].

In WOOT, the shared data type is a string over an alphabet $'\Sigma$. Each peer starts with a prescribed initial state representing the empty string. Users can perform two types of edit operations on the string at their peer:

- Insert a new character.
- Delete an existing character.

Whenever a user performs one of these operations, their peer will create an update message (see Section 4.5), integrate it immediately into its state, and send it to every other peer.

An update message created at a peer may depend on at most two of the previously integrated messages at that peer. A message cannot be delivered to a peer if its antecedents have not been delivered to it yet. In Section 4.7 we describe a few possible methods to implement this requirement, as there is a trade-off between causal consistency and scalability.

Once delivered to a remote peer, an update message will be integrated to the peers' state. The integration algorithm for an update message is the

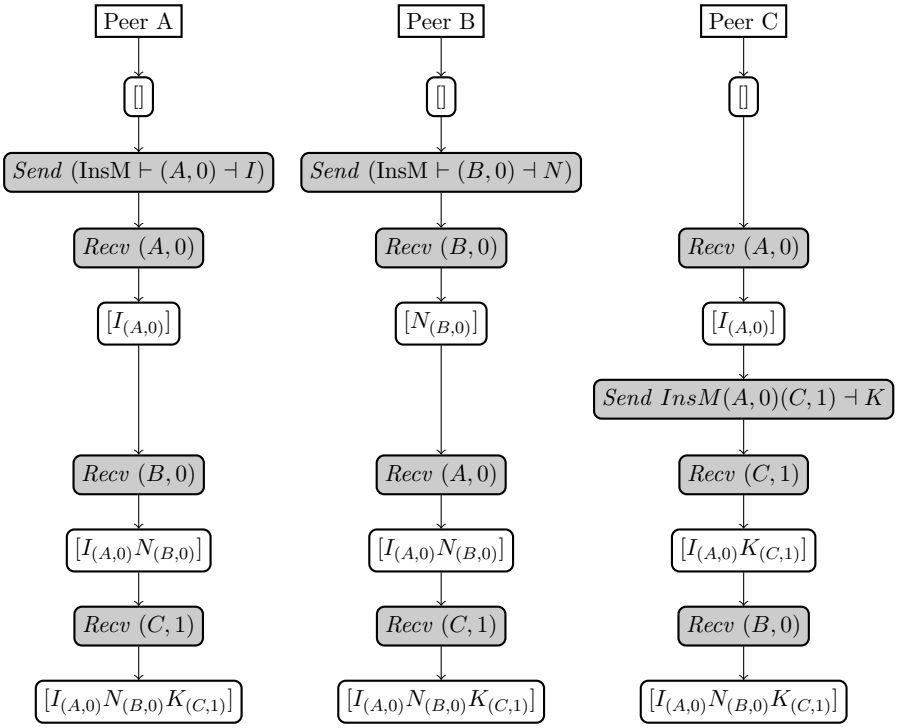


Figure 2: Example session with 3 peers. Each peer creates an update message and sends a copy of it to the other two peers. Each peer integrates the messages in a different order. The white rounded boxes represent states, for brevity we only show the W-character's symbol and identifier. Although a W-character's data structure stores the identifiers of its predecessor and successor from its original creation event. The gray round boxes represent events, we abbreviate the reception events, with the identifier of the W-character, although the peer receives the full insert message.

same whether the message originated at the same or at a different peer (see Section 4.6).

The interaction of the WOOT Framework can be visualized using a space-time diagram [10]. An example session between 3 peers is shown in Figure 2. Note that, each peer sees the edit operations in a different order.

4.1 Symbol Identifiers

The WOOT Framework requires a unique identifier for each insert operation, which it keeps associated with the inserted symbol. The identifier may not be used for another insertion operation on the same or any other peer. Moreover the set of identifiers must be endowed with a total linear order. We will denote the set of identifiers by $'\mathcal{I} :: \mathit{linorder}$.

Note that the order on the identifiers is not directly used as a global order over the inserted symbols, in contrast to the sort-key based approaches: LSEQ, LOGOOT, or TreeDoc. In particular, this means we do not require the identifier space to be dense.

In the modelling in Section 4.7, we will use the pair consisting of a unique identifier for the peer and the count of messages integrated or sent by that peer, with the lexicographic order induced by the Cartesian product of the peer identifier and the counter.

It is however possible to use other methods to generate unique identifiers, as long as the above requirements are fulfilled.

4.1.1 Extended Identifiers

datatype *'ℐ extended*
  = *Begin* (⊢)
  | *InString 'ℐ* ((*1*⟦-⟧))
  | *End* (⊣)
derive *linorder extended*

We embed the set of identifiers in an extension containing two additional elements denoting the smallest (resp. largest) element of the extension. The order of identifiers with respect to each other is preserved. The extended set is used in the corner cases, where a W-character is inserted at the beginning or end of the string - and there is no preceeding resp. succeeding W-character to reference. See also the following section.

4.2 Messages

datatype (*'ℐ*, *'Σ*) *insert-message* =
  InsertMessage (*P*:*'ℐ extended*) (*I*:*'ℐ*) (*S*:*'ℐ extended*) (*Σ*:*'Σ*)

datatype *'ℐ delete-message* = *DeleteMessage 'ℐ*

datatype (*'ℐ*, *'Σ*) *message* =
  Insert (*'ℐ*, *'Σ*) *insert-message* |
  Delete 'ℐ delete-message

Two kinds of update messages are exchanged in the WOOT Framework, indicating respectively an insertion or a deletion of a character. Thus the set of messages is a sum type *message*.

An insert message *Insert m* has the following four components:

- *P m* and *S m* denote the identifiers of the character immediately preceding (resp. succeeding) the character at the time of its insertion. The special value ⊢ (resp. ⊣) indicates that there was no such character, i.e., that it was inserted at the beginning (resp. end) of the string.

- $I\ m$ denotes the unique identifier associated to the character (as described in Subsection 4.1).
- $\Sigma\ m$ denotes the inserted character.

4.3 States

type-synonym $('\mathcal{I}, '\Sigma)$ *woot-character* $= ('\mathcal{I}, '\Sigma\ option)$ *insert-message*

A W-character w is the representation of an inserted character in the state of a peer. It has the same semantics and notation for its components as an insert message, with the difference that $\Sigma\ w$ can be $Some\ \sigma$ denoting an inserted character, or *None* if the character has already been deleted. Because of this overlap in semantics, we define the type of W-characters as a type synonym.

The state of a peer is then a string of W-characters $s :: ('\mathcal{I}, '\Sigma)$ *woot-character list*. The initial state is the empty string []. The string the user sees is the sequence of symbols omitting *None*s, i.e., the sequence: $[\sigma.\ Some\ \sigma \leftarrow map\ \Sigma\ s]$.

fun *to-woot-char* $:: ('\mathcal{I}, '\Sigma)$ *insert-message* $\Rightarrow ('\mathcal{I}, '\Sigma)$ *woot-character*
  where
    to-woot-char $(InsertMessage\ p\ i\ s\ \sigma) = InsertMessage\ p\ i\ s\ (Some\ \sigma)$

An insert message can be converted into a W-character by applying *Some* to the symbol component.

end

4.4 Basic Algorithms

theory *BasicAlgorithms*
  imports *Data ErrorMonad*
begin

In this section, we introduce preliminary definitions and functions, required by the integration and edit algorithms in the following sections.

definition *ext-ids* $:: ('\mathcal{I}, '\Sigma)$ *woot-character list* $\Rightarrow '\mathcal{I}$ *extended list*
  where *ext-ids* $s = \vdash \#(map\ (\lambda x.\ [\![\ I\ x\ ]\!])\ s) @ [\dashv]$

The function *ext-ids* returns the set of extended identifiers in a string s, including the beginning and end markers $\vdash$ and $\dashv$.

fun *idx* $:: ('\mathcal{I}, '\Sigma)$ *woot-character list* $\Rightarrow '\mathcal{I}$ *extended* $\Rightarrow$ *error* $+$ *nat*
  where
    idx s i $=$ *fromSingleton* $(filter\ (\lambda j.\ (ext\text{-}ids\ s\ !\ j = i))\ [0..<(length\ (ext\text{-}ids\ s))])$

The function *idx* returns the index in w of a W-character with a given identifier i:

- If the identifier i occurs exactly once in the string then $idx\ s\ [\![i]\!] = Inr\ (j + 1)$ where $I\ (s\ !\ j) = i$, otherwise $idx\ s\ [\![i]\!]$ will be an error.
- $idx\ s \vdash = Inr\ 0$ and $idx\ s \dashv = Inr\ (length\ w + 1)$.

```
fun nth :: ('I, 'Σ) woot-character list ⇒ nat ⇒ error + ('I, 'Σ) woot-character
  where
    nth s 0 = error (STR ''Index has to be >= 1.'') |
    nth s (Suc k) = (
      if k < (length s) then
        return (s ! k)
      else
        error (STR ''Index has to be <= length s''))
```

The function *nth* returns the W-character at a given index in s. The first character has the index 1.

```
fun list-update ::
  ('I, 'Σ) woot-character list ⇒ nat ⇒ ('I, 'Σ) woot-character ⇒
  error + ('I, 'Σ) woot-character list
  where
    list-update s (Suc k) v = (
        if k < length s then
          return (List.list-update s k v)
        else
          error (STR ''Illegal arguments.'')) |
    list-update - 0 - = error (STR ''Illegal arguments.'')
```

The function *list-update* substitutes the W-character at the index k in s with the W-character v. As before, we use the convention of using the index 1 for the first character.

end

4.5 Edit Operations

```
theory CreateAlgorithms
  imports BasicAlgorithms
begin

fun is-visible :: ('I, 'Σ) woot-character ⇒ bool
  where is-visible (InsertMessage - - - s) = (s ≠ None)

fun nth-visible :: ('I, 'Σ) woot-character list ⇒ nat ⇒ error + 'I extended
  where
    nth-visible s k = (let v = ext-ids (filter is-visible s) in
      if k < length v then
        return (v ! k)
      else
        error (STR ''Argument k out of bounds.''))
```

Let l be the count of visible symbols in s. The function *nth-visible s n*:

- Returns the identifier of the n-th visible element in s if $1 \leq n \leq l$.
- Returns $\vdash$ if $n = 0$, and $\dashv$ if $n = l + 1$.
- Returns an error otherwise.

Note that, with this definition, the first visible character in the string has the index 1.

Algorithms *create-insert* and *create-delete* detail the process by which messages are created in response to a user action.

```
fun from-non-extended :: 'I extended ⇒ error + 'I
  where
    from-non-extended ⟦ i ⟧ = Inr i |
    from-non-extended - = error (STR ''Expected InString'')

fun create-insert ::
  ('I, 'Σ) woot-character list ⇒ nat ⇒ 'Σ ⇒  'I ⇒ error + ('I, 'Σ) message
  where create-insert s n σ' i =
    do {
      p ← nth-visible s n;
      q ← nth-visible s (n + 1);
      return (Insert (InsertMessage p i q σ'))
    }
```

In particular, when a user inserts a character σ' between visible position n and its successor of the string of a peer with state s, *create-insert* starts by retrieving the identifiers p of the last visible character before n in w (or $\vdash$ if no such character exists) and q of the first visible one after n (or $\dashv$).

It then broadcasts the message *Insert* (*InsertMessage p i q* σ') with the new identifier i.

```
fun create-delete :: ('I, 'Σ) woot-character list ⇒ nat ⇒ error + ('I, 'Σ) message
  where create-delete s n =
    do {
      m ← nth-visible s n;
      i ← from-non-extended m;
      return (Delete (DeleteMessage i))
    }
```

When the user deletes the visible character at position n, *create-delete* retrieves the identifier i of the n'th visible character in s and broadcasts the message *Delete* (*DeleteMessage i*).

In both cases the message will be integrated into the peer's own state, with the same algorithm that integrates messages received from other peers.

end

4.6 Integration algorithm

In this section we describe the algorithm to integrate a received message into a peers' state.

```
theory IntegrateAlgorithm
  imports BasicAlgorithms Data
begin

fun fromSome :: 'a option ⇒ error + 'a
  where
    fromSome (Some x) = return x |
    fromSome None = error (STR ''Expected Some'')

lemma fromSome-ok-simp [simp]: (fromSome x = Inr y) = (x = Some y)
  by (cases x, simp+)

fun substr :: 'a list ⇒ nat ⇒ nat ⇒ 'a list where
  substr s l u = take (u − (Suc l)) (drop l s)

fun concurrent ::
  ('a, 's) woot-character list
  ⇒ nat
  ⇒ nat
  ⇒ ('a, 's) woot-character
  ⇒ error + ('a extended list)
  where
    concurrent s l u w =
      do {
        p-pos ← idx s (P w);
        s-pos ← idx s (S w);
        return (if (p-pos ≤ l ∧ s-pos ≥ u) then [[[I w]]] else [])
      }

function integrate-insert
  where
    integrate-insert m w p s =
      do {
        l ← idx w p;
        u ← idx w s;
        assert (l < u);
        if Suc l = u then
          return ((take l w)@[to-woot-char m]@(drop l w))
        else do {
          d ← mapM (concurrent w l u) (substr w l u);
          assert (concat d ≠ []);
          (p', s') ← fromSome (find ((λx.[[I m]] < x ∨ x = s) ∘ snd)
                       (zip (p#concat d) (concat d@[s])));
          integrate-insert m w p' s'
        }
```

```
    }
  by fastforce+

fun integrate-delete ::
  ('a :: linorder) delete-message
  ⇒ ('a, 's) woot-character list
  ⇒ error + ('a, 's) woot-character list
  where
    integrate-delete (DeleteMessage i) s =
      do {
        k ← idx s ⟦i⟧;
        w ← nth s k;
        list-update s k
          (case w of (InsertMessage p i u -) ⇒ InsertMessage p i u None)
      }

fun integrate ::
  ('a, 's) woot-character list
  ⇒ ('a :: linorder, 's) message
  ⇒ error + ('a, 's) woot-character list
  where
    integrate s (Insert m) = integrate-insert m s (P m) (S m) |
    integrate s (Delete m) = integrate-delete m s
```

Algorithm *integrate* describes the main function that is called when a new message m has to be integrated into the state s of a peer. It is called both when m was generated locally or received from another peer. Note that we require that the antecedant messages have already been integrated. See also Section 4.7 for the delivery assumptions that ensure this requirement.

Algorithm *integrate-delete* describes the procedure to integrate a delete message: *DeleteMessage i*. The algorithm just replaces the symbol of the W-character with identifier i with the value *None*. It is not possible to entirely remove a W-character if it is deleted, since there might be unreceived insertion messages that depend on its position.

Algorithm *integrate-insert* describes the procedure to integrate an insert message: $m = \mathit{InsertMessage}\ p\ i\ s\ \sigma$. Since insertion operations can happen concurrently and the order of message delivery is not fixed, it can happen that a remote peer receiving m finds multiple possible insertion points between the predecessor p and successor s that were recorded when the message was generated. An example of this situation is the conflict between $\mathit{InsertMessage} \vdash (A,\ 0) \dashv \mathit{CHR}\ ''I''$ and $\mathit{InsertMessage} \vdash (B,\ 0) \dashv \mathit{CHR}\ ''N''$ in Figure 2.

A first attempt to resolve this would be to insert the W-characters by choosing an insertion point using the order induced by their identifiers to achieve a consistent ordering. But this method fails in some cases: a counterexample was found by Oster et al. [19, section 2].

The solution introduced by the authors of WOOT is to restrict the identifier comparison to the set of W-characters in the range $substr\ l\ u\ s$ whose predecessor and successor are outside of the possible range, i.e. $idx\ s\ (P\ w) \leq l$ and $idx\ s\ (S\ w) \geq u$.

New narrowed bounds are selected by finding the first W-character within that restricted set with an identifier strictly larger than the identifier of the new W-character.

This leads to a narrowed range where the found character forms an upper bound and its immediately preceeding character the lower bound. The method is applied recursively until the insertion point is uniquely determined.

Note that the fact that this strategy leads to a consistent ordering has only been verified for a bounded model. One of the contributions of this paper is to provide a complete proof for it.

end

4.7 Network Model

In the past subsections, we described the algorithms each peer uses to integrate received messages and broadcast new messages when an edit operation has been made on that peer.

In this section, we model the WOOT Framework as a distributed application and set the basis for the consistency properties, we want to establish.

We assume a finite set of peers starting with the same initial state of an empty W-string, each peer reaches a finite set of subsequent states, caused by the integration of received (or locally generated messages). A message is always generated from a concrete state of a peer, using the algorithms described in Section 4.5. Moreover, we assume that the same message will only be delivered once to a peer. Finally, we assume that the happened before relation, formed by

- Subsequent states of the same peer
- States following the reception and states that were the generation sites

do not contain loops. (Equivalently that the transitive closure of the relation is a strict partial order.)

The latter is a standard assumption in the modelling of distributed systems (compare e.g. [21, Chapter 6.1]) effectively implied by the fact that there are no physical causal loops.

Additionally, we assume that a message will be only received by a peer, when the antecedent messages have already been received by the peer. This is a somewhat technical assumption to simplify the description of the system.

In a practical implementation a peer would buffer the set of messages that cannot yet be integrated. Note that this assumption is automatically implied if causal delivery is assumed.

We establish two properties under the above assumptions

- The integration algorithm never fails.
- Two peers having received the same set of messages will be in the same state.

The model assumptions are derived from Gomes et al.[7] and Shapiro et al.[23] with minor modifications required for WOOT.

theory *DistributedExecution*
imports *IntegrateAlgorithm CreateAlgorithms HOL−Library.Product-Lexorder*
begin

type-synonym *'p event-id = 'p × nat*

datatype *('p,'s) event =*
Send ('p event-id, 's) message |
Receive 'p event-id ('p event-id, 's) message

The type variable *'p* denotes a unique identifier identifying a peer. We model each peer's history as a finite sequence of events, where each event is either the reception or broadcast of a message. The index of the event in a peer's history and its identifier form a pair uniquely identifying an event in a distributed execution of the framework. In the case of a reception, *Receive s m* indicated the reception of the message *m* sent at event *s*.

In the following we introduce the locale *dist-execution-preliminary* from which the *dist-execution* locale will inherit. The reason for the introduction of two locales is technical - in particular, it is not possible to interleave definitions and assumptions within the definition of a locale. The preliminary locale only introduces the assumption that the set of participating peers is finite.

locale *dist-execution-preliminary =*
fixes *events :: ('p :: linorder) ⇒ ('p,'s) event list*
— We introduce a locale fixing the sequence of events per peer.

assumes *fin-peers*: *finite (UNIV :: 'p set)*
— We are assuming a finite set of peers.

begin

fun *is-valid-event-id*
where *is-valid-event-id (i,j) = (j < length (events i))*

```
fun event-pred
  where event-pred (i,j) p  = (is-valid-event-id (i,j) ∧ p (events i ! j))

fun event-at
  where event-at i m = event-pred i ((=) m)

fun is-reception
  where
    is-reception i j = event-pred j (λe. case e of Receive s - ⇒ s = i | - ⇒ False)

fun happened-immediately-before where
  happened-immediately-before i j = (
     is-valid-event-id i ∧
     is-valid-event-id j ∧
     ((fst i = fst j ∧ Suc (snd i) = snd j) ∨ is-reception i j))
```

The *happened-immediately-before* describes immediate causal precedence:

- An events causally precedes the following event on the same peer.
- A message broadcast event causally precedes the reception event of it.

The transitive closure of this relation is the famous happened before relation introduced by Lamport[12].

In the *dist-execution* we will assume that the relation is acyclic - which implies that the transitive closure *happened-immediately-before*$^{++}$ is a strict partial order.

Each peer passes through a sequence of states, which may change after receiving a message. We denote the initial state of peer p as $(p, 0)$ and the state after event (p, i) as $(p, i + 1)$. Note that there is one more state per peer than events, since we are count both the initial and terminal state of a peer.

```
fun is-valid-state-id
  where is-valid-state-id (i,j) = (j ≤ length (events i))

fun received-messages
  where
    received-messages (i,j) = [m. (Receive - m) ← (take j (events i))]

fun state where state i = foldM integrate [] (received-messages i)
```

Everytime a peer receives a message its state is updated by integrating the message. The function *state* returns the state for a given state id.

end

The function *deps* computes the identifiers a message depends on.

```
fun extended-to-set :: 'a extended ⇒ 'a set
```

where
extended-to-set $[\![i]\!] = \{i\}$ |
extended-to-set - = {}

fun *deps* :: ('*id*, '*s*) *message* ⇒ '*id set*
where
deps (*Insert* (*InsertMessage l* - *u* -)) = *extended-to-set l* ∪ *extended-to-set u* |
deps (*Delete* (*DeleteMessage i*)) = {*i*}

locale *dist-execution* = *dist-execution-preliminary* +
assumes *no-data-corruption*:
⋀*i s m*. *event-at i* (*Receive s m*) ⟹ *event-at s* (*Send m*)
— A received message must also have been actually broadcast. Note that, we do not assume that a broadcast message will be received by all peers, similar to the modelling made by [7, Section 5.2].

assumes *at-most-once*:
⋀*i j s m*.
event-at i (*Receive s m*) ⟹
event-at j (*Receive s m*) ⟹
fst i = *fst j* ⟹ *i* = *j*
— A peer will never receive the same message twice. Note that this is something that can be easily implemented in the application layer, if the underlying transport mechanism does not guarantee it.

assumes *acyclic-happened-before*:
acyclicP happened-immediately-before
— The immediate causal precendence relation is acyclic, which implies that its closure, the *happened before* relation is a strict partial order.

assumes *semantic-causal-delivery*:
⋀*m s i j i'*. *event-at* (*i*,*j*) (*Receive s m*) ⟹ *i'* ∈ *deps m* ⟹
∃ *s' j' m'*. *event-at* (*i*,*j'*) (*Receive s'* (*Insert m'*)) ∧ *j'* < *j* ∧ *I m'* = *i'*
— A message will only be delivered to a peer, if its antecedents have already been delivered. (See beginning of this Section for the reason of this assumption).

assumes *send-correct*:
⋀*m i*. *event-at i* (*Send m*) ⟹
(∃ *n σ*. *return m* = *state i* ⤜ (λ*s*. *create-insert s n σ i*)) ∨
(∃ *n*. *return m* = *state i* ⤜ (λ*s*. *create-delete s n*))
— A peer broadcasts messages by running the *create-insert* or *create-delete* algorithm on its current state. In the case of an insertion the new character is assigned the event id as its identifier. Note that, it would be possible to assume, a different choice for allocating unique identifiers to new W-characters. We choose the event id since it is automatically unique.

begin

Based on the assumptions above we show in Section 6:

- *Progress*: All reached states *state i* will be successful, i.e., the algorithm *integrate* terminates and does not fail.
- *Strong Eventual Consistency*: Any pair of states *state i* and *state j* which have been reached after receiving the same set of messages, i.e., *set* (*received-messages i*) = *set* (*received-messages j*) will be equal.

end

end

5 Formalized Proof

theory *SortKeys*
imports *Data HOL−Library.List-Lexorder HOL−Library.Product-Lexorder*
begin

datatype *sort-dir* =
Left |
Right
derive *linorder sort-dir*

lemma *sort-dir-less-def* [*simp*]: (*x* < *y*) = (*x* = *Left* ∧ *y* = *Right*)
by (*cases x*, *case-tac* [!] *y*, *simp-all add*:*less-sort-dir-def*)

datatype *'a sort-key* =
NonFinal (*'a* × *sort-dir*) *'a sort-key* |
Final 'a

type-synonym *'id position* = *'id sort-key extended*

fun *embed-dir* **where** *embed-dir* (*x*,*Left*) = (*x*, *0*) | *embed-dir* (*x*,*Right*) = (*x*, *Suc* (*Suc 0*))

lemma *embed-dir-inj* [*simp*]: (*embed-dir x* = *embed-dir y*) = (*x* = *y*)
by (*cases x*, *cases y*, *case-tac* [!] *snd x*, *case-tac* [!] *snd y*, *simp*+)

lemma *embed-dir-mono* [*simp*]: (*embed-dir x* < *embed-dir y*) = (*x* < *y*)
by (*cases x*, *cases y*, *case-tac* [!] *snd x*, *case-tac* [!] *snd y*, (*simp add*:*less-sort-dir-def*)+)

fun *sort-key-embedding* :: *'a sort-key* ⇒ (*'a* × *nat*) *list*
where
sort-key-embedding (*NonFinal x y*) = *embed-dir x*#(*sort-key-embedding y*) |
sort-key-embedding (*Final i*) = [(*i*, *Suc 0*)]

lemma *sort-key-embedding-injective*:
sort-key-embedding x = *sort-key-embedding y* ⟹ *x* = *y*
apply (*induct x arbitrary*: *y*)
apply (*metis embed-dir-inj list.distinct*(*1*) *list.inject sort-key.exhaust*

sort-key-embedding.simps)
by (*metis fst-conv list.distinct*(*1*) *list.inject sort-key.exhaust*
sort-key-embedding.simps)

instantiation *sort-key* :: (*ord*) *ord*
begin
definition *sort-key-less-eq-def* [*simp*]:
(*x* :: ('*a* :: *ord*) *sort-key*) ≤ *y* ⟷
(*sort-key-embedding x* ≤ *sort-key-embedding y*)

definition *sort-key-less-def* [*simp*]:
(*x* :: ('*a* :: *ord*) *sort-key*) < *y* ⟷
(*sort-key-embedding x* < *sort-key-embedding y*)

instance ..
end

instantiation *sort-key* :: (*order*) *order*
begin
instance by (*intro-classes*, *simp-all add*: *less-le-not-le sort-key-embedding-injective*)
end

instantiation *sort-key* :: (*linorder*) *linorder*
begin
instance by (*intro-classes*, *meson less-imp-le not-le sort-key-less-eq-def*)
end

end

5.1 Definition of Ψ

theory *Psi*
imports *SortKeys HOL−Eisbach.Eisbach*
begin

fun *extended-size* :: ('*a sort-key*) *extended* ⇒ *nat*
where
extended-size ⟦*x*⟧ = *size x* |
extended-size - = *0*

lemma *extended-simps* [*simp*]:
(⊢ < *x*) = (*x* ≠ ⊢)
(⟦*x'*⟧ < ⟦*y'*⟧) = (*x'* < *y'*)
⟦*x'*⟧ < ⊣
¬(⟦*x'*⟧ < ⊢)
¬(⊣ < *x*)
⊢ ≤ *x*
(⟦*x'*⟧ ≤ ⟦*y'*⟧) = ((*x'* :: '*a* :: *linorder*) ≤ *y'*)
x ≤ ⊣

$\neg([\![x']\!] \leq \vdash)$
$(\dashv \leq x) = (x = \dashv)$
by (*case-tac* [!] *x*, *simp-all add*:*less-extended-def less-eq-extended-def le-less*)

fun *int-size* **where** *int-size* (*l*,*u*) = *max* (*extended-size l*) (*extended-size u*)

lemma *position-cases*:
assumes $\bigwedge$ *y z*. *x* = [\![*NonFinal* (*y*,*Left*) *z*]\!] $\Longrightarrow$ *p*
assumes $\bigwedge$ *y z*. *x* = [\![*NonFinal* (*y*,*Right*) *z*]\!] $\Longrightarrow$ *p*
assumes $\bigwedge$ *y*. *x* = [\![*Final y*]\!] $\Longrightarrow$ *p*
assumes *x* = $\vdash$ $\Longrightarrow$ *p*
assumes *x* = $\dashv$ $\Longrightarrow$ *p*
shows *p*
by (*metis assms embed-dir.cases extended-size.cases sort-key-embedding.cases*)

fun *derive-pos* ::
('*a* :: *linorder*) × *sort-dir* ⇒ '*a sort-key extended* ⇒ '*a sort-key extended*
where
derive-pos h [\![*NonFinal x y*]\!] =
(*if h* < *x then* $\dashv$ *else* (*if x* < *h then* $\vdash$ *else* [\![*y*]\!])) |
derive-pos h [\![*Final x*]\!] =
(*if fst h* < *x* ∨ *fst h* = *x* ∧ *snd h* = *Left then* $\dashv$ *else* $\vdash$) |
derive-pos - $\vdash$ = $\vdash$ |
derive-pos - $\dashv$ = $\dashv$

lemma *derive-pos-mono*: *x* ≤ *y* $\Longrightarrow$ *derive-pos h x* ≤ *derive-pos h y*
apply (*cases h*, *cases snd h*)
apply (*rule-tac* [!] *position-cases* [**where** *x*=*x*])
apply (*rule-tac* [!] *position-cases* [**where** *x*=*y*])
by (*simp-all*, *auto*)

fun γ :: ('*a* :: *linorder*) *position* ⇒ *sort-dir* ⇒ '*a* × *sort-dir*
where
γ [\![*NonFinal x y*]\!] - = *x* |
γ [\![*Final x*]\!] *d* = (*x*,*d*) |
γ $\vdash$ - = *undefined* |
γ $\dashv$ - = *undefined*

fun *derive-left* **where**
derive-left (*l*, *u*) = (*derive-pos* (γ *l Right*) *l*, *derive-pos* (γ *l Right*) *u*)

fun *derive-right* **where**
derive-right (*l*, *u*) = (*derive-pos* (γ *u Left*) *l*, *derive-pos* (γ *u Left*) *u*)

fun *is-interval* **where** *is-interval* (*l*,*u*) = (*l* < *u*)

fun *elem* **where** *elem x* (*l*,*u*) = (*l* < *x* ∧ *x* < *u*)

fun *subset* **where** *subset* (*l*,*u*) (*l'*,*u'*) = (*l'* ≤ *l* ∧ *u* ≤ *u'*)

method *interval-split* **for** $x :: ('a :: linorder)\ position \times 'a\ position =$
(*case-tac* [!] x,
rule-tac [!] *position-cases* [**where** x=*fst* x],
rule-tac [!] *position-cases* [**where** x=*snd* x])

lemma *derive-size*:
$[\![Final\ i]\!] \leq fst\ x \wedge is\text{-}interval\ x \Longrightarrow int\text{-}size\ (derive\text{-}left\ x) < int\text{-}size\ x$
$snd\ x \leq [\![Final\ i]\!] \wedge is\text{-}interval\ x \Longrightarrow int\text{-}size\ (derive\text{-}right\ x) < int\text{-}size\ x$
by (*interval-split* x, *simp-all add*:*less-SucI*)

lemma *derive-interval*:
$[\![Final\ i]\!] \leq fst\ x \Longrightarrow is\text{-}interval\ x \Longrightarrow is\text{-}interval\ (derive\text{-}left\ x)$
$snd\ x \leq [\![Final\ i]\!] \Longrightarrow is\text{-}interval\ x \Longrightarrow is\text{-}interval\ (derive\text{-}right\ x)$
by (*interval-split* x, *simp-all*, *auto*)

function $\Psi :: ('a :: linorder)\ position \times 'a\ position \Rightarrow 'a \Rightarrow 'a\ sort\text{-}key$
where
$\Psi\ (l,u)\ i = Final\ i$
if $l < [\![Final\ i]\!] \wedge [\![Final\ i]\!] < u$ |
$\Psi\ (l,u)\ i = NonFinal\ (\gamma\ l\ Right)\ (\Psi\ (derive\text{-}left\ (l,u))\ i)$
if $[\![Final\ i]\!] \leq l \wedge l < u$ |
$\Psi\ (l,u)\ i = NonFinal\ (\gamma\ u\ Left)\ (\Psi\ (derive\text{-}right\ (l,u))\ i)$
if $u \leq [\![Final\ i]\!] \wedge l < u$ |
$\Psi\ (l,u)\ i = undefined$ **if** $u \leq l$
by (*metis leI old.prod.exhaust*, *auto*)

termination
apply (*relation measure* $(\lambda(p,i).\ int\text{-}size\ p)$, *simp*)
using *derive-size* **by** *fastforce*+

proposition *psi-elem*: $is\text{-}interval\ x \Longrightarrow elem\ [\![\Psi\ x\ i]\!]\ x$
proof (*induct int-size x arbitrary*:x *rule*: *nat-less-induct*)
case *1*
consider (*a*) $[\![Final\ i]\!] \leq fst\ x$ | (*b*) $elem\ [\![Final\ i]\!]\ x$ | (*c*) $snd\ x \leq [\![Final\ i]\!]$
using *not-le* **by** (*metis elem.simps prod.collapse*)
then show *?case*
proof (*cases*)
case *a*
hence *elem* $[\![\Psi\ (derive\text{-}left\ x)\ i]\!]\ (derive\text{-}left\ x)$
by (*metis 1 derive-size*(*1*) *derive-interval*(*1*))
then show *?thesis* **using** *a 1*(*2*)
by (*interval-split* x, *simp-all del*:Ψ.*simps*, *auto*)
next
case *b*
then show *?thesis* **by** (*cases* x, *simp*)
next
case *c*
hence *elem* $[\![\Psi\ (derive\text{-}right\ x)\ i]\!]\ (derive\text{-}right\ x)$

```
    by (metis 1 derive-size(2) derive-interval(2))
  then show ?thesis using c 1(2)
    by (interval-split x, simp-all del:Ψ.simps, auto)
 qed
qed

proposition psi-mono:
 assumes i1 < i2
 shows is-interval x ⟹ Ψ x i1 < Ψ x i2
proof (induct int-size x arbitrary:x rule: nat-less-induct)
 case 1
 have a:⟦Final i1⟧ < ⟦Final i2⟧
  using assms by auto
 then consider
  (a) ⟦Final i1⟧ ≤ fst x ∧ ⟦Final i2⟧ ≤ fst x |
  (b) ⟦Final i1⟧ ≤ fst x ∧ elem ⟦Final i2⟧ x |
  (c) ⟦Final i1⟧ ≤ fst x ∧ snd x ≤ ⟦Final i2⟧ |
  (d) elem ⟦Final i1⟧ x ∧ elem ⟦Final i2⟧ x |
  (e) elem ⟦Final i1⟧ x ∧ snd x ≤ ⟦Final i2⟧ |
  (f) snd x ≤ ⟦Final i2⟧ ∧ snd x ≤ ⟦Final i1⟧
  using assms 1(2) apply (cases x)
  by (metis (mono-tags, opaque-lifting) dual-order.strict-trans elem.simps
      fst-conv leI snd-conv)
 then show ?case
 proof (cases)
  case a
  hence Ψ (derive-left x) i1 < Ψ (derive-left x) i2
   by (metis 1 derive-size(1) derive-interval(1))
  thus ?thesis using a 1(2) by (cases x, simp)
 next
  case b
  thus ?thesis using 1(2) apply (cases x, simp)
   by (rule-tac [!] position-cases [where x=fst x], simp-all)
 next
  case c
  show ?thesis
  proof (cases γ (fst x) Right = γ (snd x) Left)
   case True
   have e:is-interval (derive-left x) using c 1(2) derive-interval(1) by blast
   have f:derive-left x = derive-right x using True by (cases x, simp)
   have h:Ψ (derive-left x) i1 < Ψ (derive-right x) i2
    apply (cases x, simp only: f)
    by (metis 1.hyps 1.prems c derive-size(2) e f)
   show ?thesis using c 1(2) h True by (cases x, simp)
  next
   case False
   hence γ (fst x) Right < γ (snd x) Left using 1(2) c
   by (interval-split x, simp-all, auto)
   then show ?thesis using c 1(2) by (cases x, simp)
```

```
    qed
  next
    case d
    thus ?thesis using 1(2) a by (cases x, simp)
  next
    case e
    thus ?thesis using 1(2) apply (cases x, simp)
      by (rule-tac [!] position-cases [where x=snd x], simp-all del:Ψ.simps)
  next
    case f
    hence b:Ψ (derive-right x) i1 < Ψ (derive-right x) i2
      by (metis 1 derive-size(2) derive-interval(2))
    thus ?thesis using f 1(2) by (cases x, simp)
  qed
qed

proposition psi-narrow:
  elem ⟦Ψ x′ i⟧ x ⟹ subset x x′ ⟹ Ψ x′ i = Ψ x i
proof (induct int-size x′ arbitrary: x x′ rule: nat-less-induct)
  case 1
  have a: is-interval x using 1(2)
    by (metis dual-order.strict-trans elem.elims(2) is-interval.simps)
  have d: is-interval x′ using a 1(3) apply (cases x′, cases x, simp) by auto
  consider
    (before) ⟦Final i⟧ ≤ fst x′ |
    (between) elem ⟦Final i⟧ x′ |
    (after) snd x′ ≤ ⟦Final i⟧ using 1 apply simp
    by (metis elem.simps leI prod.collapse)
  then show ?case
  proof (cases)
    case before
    have b: ⟦Final i⟧ ≤ fst x using before 1 apply (cases x)
      by (metis dual-order.trans fst-conv subset.elims(2))
    obtain z where z-def: Ψ x′ i = NonFinal (γ (fst x′) Right) z
      using before d apply (cases x′) by simp
    have c:γ (fst x′) Right = γ (fst x) Right
      using 1(3) z-def 1(2) apply (cases x, cases x′, simp)
      apply (rule-tac [!] position-cases [where x=fst x])
      apply (rule-tac [!] position-cases [where x=fst x′])
      using before by (simp-all del:Ψ.simps, auto)
    have c1:subset (derive-left x) (derive-left x′)
      using c 1(3) by (cases x, cases x′, simp add:derive-pos-mono)
    have g:z = Ψ (derive-left x′) i using z-def before d by (cases x′, simp)
    have elem ⟦NonFinal (γ (fst x) Right) z⟧ x
      using 1(2) z-def by (simp add: c)
    hence elem ⟦z⟧ (derive-left x) using before b
      by (interval-split x, simp-all del:Ψ.simps, auto)
    hence Ψ (derive-left x′) i = Ψ (derive-left x) i
      using 1(1) before d c1 apply (simp only:g)
```

```
    by (metis (no-types) derive-size(1))
   thus ?thesis using before b a d c by (cases x, cases x', simp)
  next
   case between
   thus ?thesis using 1 by (cases x, cases x', auto)
  next
   case after
   have b: snd x ≤ ⟦Final i⟧ using after 1 apply (cases x)
    by (metis (mono-tags, opaque-lifting) dual-order.trans prod.exhaust-sel
        subset.simps)
   obtain z where z-def:Ψ x' i = NonFinal (γ (snd x') Left) z
    using after d by (cases x', simp)
   have c:γ (snd x') Left = γ (snd x) Left
    using 1(3) z-def 1(2) apply (simp, cases x, cases x')
    apply (rule-tac [!] position-cases [where x=snd x])
    apply (rule-tac [!] position-cases [where x=snd x']) using after
    by (simp-all del:Ψ.simps, auto)
   have c1:subset (derive-right x) (derive-right x')
    using c 1(3) by (cases x, cases x', simp add:derive-pos-mono)
   have g:z = Ψ (derive-right x') i using z-def after d by (cases x', simp)
   have elem ⟦NonFinal (γ (snd x) Left) z⟧ x
    using 1(2) z-def by (simp add: c)
   hence elem ⟦z⟧ (derive-right x) using after b
    by (interval-split x, simp-all del:Ψ.simps, auto)
   hence Ψ (derive-right x') i = Ψ (derive-right x) i
    using 1(1) after d c1 apply (simp only:g)
    by (metis (no-types) derive-size(2))
   thus ?thesis using after b a d c by (cases x, cases x', simp)
  qed
qed

definition preserve-order :: 'a :: linorder ⇒ 'a ⇒ 'b :: linorder ⇒ 'b ⇒ bool
  where preserve-order x y u v ≡ (x < y ⟶ u < v) ∧ (x > y ⟶ u > v)

proposition psi-preserve-order:
  fixes l l' u u' i i'
  assumes elem ⟦Ψ (l, u) i⟧ (l',u')
  assumes elem ⟦Ψ (l', u') i'⟧ (l, u)
  shows preserve-order i i' ⟦Ψ (l,u) i⟧ ⟦Ψ (l', u') i'⟧
proof −
  have l < u using assms(2) by auto
  hence a:elem ⟦Ψ (l, u) i⟧ (max l l', min u u')
   using assms(1) psi-elem by fastforce
  hence b:Ψ (l,u) i = Ψ (max l l', min u u') i
   by (simp add: psi-narrow)
  have l' < u' using assms(1) by auto
  hence elem ⟦Ψ (l',u') i'⟧ (max l l', min u u')
   using assms(2) psi-elem by fastforce
  hence c:Ψ (l',u') i' = Ψ (max l l', min u u') i'
```

by (*simp add*: *psi-narrow*)
hence *max l l′ < min u u′* **using** *a min-def* **by** *auto*
then show *?thesis* **apply** (*simp only*: *preserve-order-def b c*)
using *psi-mono extended-simps*(*2*) *is-interval.simps* **by** *blast*
qed

end

5.2 Sorting

Some preliminary lemmas about sorting.

theory *Sorting*
imports *Main HOL.List HOL−Library.Sublist*
begin

lemma *insort*:
assumes *Suc l < length s*
assumes *s ! l < (v :: 'a :: linorder)*
assumes *s ! (l+1) > v*
assumes *sorted-wrt (<) s*
shows *sorted-wrt (<) ((take (Suc l) s)@v#(drop (Suc l) s))*
proof −
have *sorted-wrt (<) (take (Suc l) s@(drop (Suc l) s))*
using *assms*(*4*) **by** *simp*
moreover have
$\bigwedge x.\ x \in set\ (take\ (Suc\ l)\ s) = (\exists\, i.\ i < (Suc\ l) \wedge i < length\ s \wedge s\ !\ i = x)$
by (*metis in-set-conv-nth length-take min-less-iff-conj nth-take*)
hence $\bigwedge x.\ x \in set\ (take\ (Suc\ l)\ s) \Longrightarrow x < v$
using *assms* **apply** (*simp*)
using *less-Suc-eq sorted-wrt-nth-less* **by** *fastforce*
moreover have
$\bigwedge x.\ x \in set\ (drop\ (Suc\ l)\ s) = (\exists\, i.\ Suc\ l + i < length\ s \wedge s\ !\ (Suc\ l + i) = x)$
using *assms*(*1*) **by** (*simp add*:*in-set-conv-nth add.commute less-diff-conv*)
hence $\bigwedge x.\ x \in set\ (drop\ (Suc\ l)\ s) \Longrightarrow x > v$
using *assms* **apply** (*simp*)
by (*metis add.right-neutral add-diff-cancel-left′ diff-Suc-Suc diff-is-0-eq′*
leI le-less-trans less-imp-le sorted-wrt-iff-nth-less)
ultimately show *?thesis*
by (*simp add*:*sorted-wrt-append del*:*append-take-drop-id*)
qed

lemma *sorted-wrt-irrefl-distinct*:
assumes *irreflp r*
shows *sorted-wrt r xs ⟶ distinct xs*
using *assms* **by** (*induction xs*, *simp*, *simp*, *meson irreflp-def*)

lemma *sort-set-unique-h*:
assumes *irreflp r ∧ transp r*
assumes *set (x#xs) = set (y#ys)*

assumes $\forall z \in set\ xs.\ r\ x\ z$
assumes $\forall z \in set\ ys.\ r\ y\ z$
shows $x = y \wedge set\ xs = set\ ys$
by (*metis assms insert-eq-iff irreflp-def list.set-intros*(*1*) *list.simps*(*15*) *set-ConsD transpD*)

lemma *sort-set-unique-rel*:
assumes *irreflp r* $\wedge$ *transp r*
assumes *set x* = *set y*
assumes *sorted-wrt r x*
assumes *sorted-wrt r y*
shows *x* = *y*
proof −
have *length x* = *length y*
using *assms* **by** (*metis sorted-wrt-irrefl-distinct distinct-card*)
then show *?thesis* **using** *assms*
apply(*induct rule*:*list-induct2*, *simp*, *simp*)
by (*metis assms*(*1*) *list.simps*(*15*) *sort-set-unique-h*)
qed

lemma *sort-set-unique*:
assumes *set x* = *set y*
assumes *sorted-wrt* (<) (*map* (*f* :: ('*a* ⇒ ('*b* :: *linorder*))) *x*)
assumes *sorted-wrt* (<) (*map f y*)
shows *x* = *y*
using *assms* **apply** (*simp add*:*sorted-wrt-map*)
by (*metis* (*no-types*, *lifting*) *irreflp-def less-irrefl sort-set-unique-rel transpD transpI transp-less*)

If two sequences contain the same element and strictly increasing with respect.

lemma *subseq-imp-sorted*:
assumes *subseq s t*
assumes *sorted-wrt p t*
shows *sorted-wrt p s*
proof −
have *sorted-wrt p s* ∨ ¬ *sorted-wrt p t*
apply (*rule list-emb.induct*[**where** *P*=(=)])
using *list-emb-set assms* **by** *fastforce*+
thus *?thesis* **using** *assms* **by** *blast*
qed

If a sequence t is sorted with respect to a relation p then a subsequence will be as well.

fun *to-ord* **where** *to-ord r x y* = (¬(*r*** *y x*))

lemma *trancl-idemp*: $r^{++++}\ x\ y = r^{++}\ x\ y$
by (*metis r-into-rtranclp reflclp-tranclp rtranclp-idemp rtranclp-reflclp rtranclp-tranclp-tranclp tranclp.cases tranclp.r-into-trancl*)

lemma *top-sort*:
fixes *rp*
assumes *acyclicP r*
shows *finite s* $\longrightarrow$ ($\exists$ *l*. *set l* = *s* $\wedge$ *sorted-wrt* (*to-ord r*) *l* $\wedge$ *distinct l*)
proof (*induction card s arbitrary*:*s*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
hence *s* $\neq$ {} **by** *auto*
moreover
have *acyclicP* (r^{++}) **using** *assms*
by (*simp add*:*acyclic-def trancl-def trancl-idemp*)
hence *acyclic* ({(*x*,*y*). r^{++} *x y*} $\cap$ *s* $\times$ *s*)
by (*meson acyclic-subset inf-le1*)
hence *wf* ({(*x*,*y*). r^{++} *x y*} $\cap$ *s* $\times$ *s*) **using** *Suc*
by (*metis card.infinite finite-Int finite-SigmaI nat.distinct*(*1*)
wf-iff-acyclic-if-finite)
ultimately obtain *z* **where**
z $\in$ *s* $\wedge$ ($\forall$ *y*. (*y*, *z*) $\in$ ({(*x*,*y*). r^{++} *x y*} $\cap$ *s* $\times$ *s*) $\longrightarrow$ *y* $\notin$ *s*)
by (*metis ex-in-conv wf-eq-minimal*)
hence *z-def*: *z* $\in$ *s* $\wedge$ ($\forall$ *y*. r^{++} *y z* $\longrightarrow$ *y* $\notin$ *s*) **by** *blast*
hence *card* (*s* $-$ {*z*}) = *n*
by (*metis One-nat-def Suc.hyps*(*2*) *card-Diff-singleton-if card.infinite*
diff-Suc-Suc diff-zero nat.simps(*3*))
then obtain *l* **where** *l-def*:
set l = *s* $-$ {*z*} $\wedge$ *sorted-wrt* (*to-ord r*) *l* $\wedge$ *distinct l*
by (*metis Zero-not-Suc card.infinite finite-Diff Suc*)
hence *set* (*z*#*l*) = *s* **using** *z-def* **by** *auto*
moreover have $\forall$ *y* $\in$ *set l*. $\neg$(r^{**} *y z*) **using** *z-def l-def rtranclpD* **by** *force*
ultimately show *?case*
by (*metis distinct.simps*(*2*) *insert-absorb l-def list.simps*(*15*)
sorted-wrt.simps(*2*) *to-ord.elims*(*3*))
qed

lemma *top-sort-eff*:
assumes *irreflp* p^{++}
assumes *sorted-wrt* (*to-ord p*) *x*
assumes *i* < *length x*
assumes *j* < *length x*
assumes (p^{++} (*x* ! *i*) (*x* ! *j*))
shows *i* < *j*
using *assms* **apply** (*cases i* > *j*)
apply (*metis sorted-wrt-nth-less r-into-rtranclp reflclp-tranclp*
rtranclp-idemp rtranclp-reflclp to-ord.simps)
by (*metis irreflp-def nat-neq-iff*)

end

5.3 Consistency of sets of WOOT Messages

theory *Consistency*
imports *SortKeys Psi Sorting DistributedExecution*
begin

definition *insert-messages* :: ('$\mathcal{I}$, 'Σ) *message set* ⇒ ('$\mathcal{I}$, 'Σ) *insert-message set*
where *insert-messages M* = {*x*. *Insert x* ∈ *M*}

lemma *insert-insert-message*:
insert-messages (*M* ∪ {*Insert m*}) = *insert-messages M* ∪ {*m*}
by (*simp add*:*insert-messages-def*, *simp add*:*set-eq-iff*)

definition *delete-messages* :: ('*a*, '*s*) *message set* ⇒ '*a delete-message set*
where *delete-messages M* = {*x*. *Delete x* ∈ *M*}

fun *depends-on* **where** *depends-on M x y* = (*x* ∈ *M* ∧ *y* ∈ *M* ∧ *I x* ∈ *deps* (*Insert y*))

definition *a-conditions* ::
(('*a* :: *linorder*), '*s*) *insert-message set* ⇒ ('*a extended* ⇒ '*a position*) ⇒ *bool*
where *a-conditions M a* = (
a ⊢ < *a* ⊣ ∧
(∀ *m*. *m* ∈ *M* ⟶ *a* (*P m*) < *a* (*S m*) ∧
a ⟦*I m*⟧ = ⟦Ψ (*a* (*P m*), *a* (*S m*)) (*I m*)⟧))

definition *consistent* :: ('*a* :: *linorder*, '*s*) *message set* ⇒ *bool*
where *consistent M* ≡
inj-on I (*insert-messages M*) ∧
(⋃ (*deps* ‘ *M*) ⊆ (*I* ‘ *insert-messages M*)) ∧
wfP (*depends-on* (*insert-messages M*)) ∧
(∃ *a*. *a-conditions* (*insert-messages M*) *a*)

lemma *consistent-subset*:
assumes *consistent N*
assumes *M* ⊆ *N*
assumes ⋃ (*deps* ‘ *M*) ⊆ (*I* ‘ *insert-messages M*)
shows *consistent M*
proof −
have *a*:*insert-messages M* ⊆ *insert-messages N*
using *assms*(*2*) *insert-messages-def* **by** *blast*
hence *b*:*inj-on I* (*insert-messages M*)
using *assms*(*1*) *consistent-def inj-on-subset* **by** *blast*
have *wfP* (*depends-on* (*insert-messages N*))
using *assms*(*1*) *consistent-def* **by** *blast*
moreover have
depends-on (*insert-messages M*) ≤ *depends-on* (*insert-messages N*)
using *a* **by** *auto*
ultimately have *c*:*wfP* (*depends-on* (*insert-messages M*))
using *a wf-subset* [*to-pred*] **by** *blast*

```
  obtain a where a-conditions (insert-messages N) a
    using assms(1) consistent-def by blast
  hence a-conditions (insert-messages M) a
    by (meson a a-conditions-def subset-iff)
  thus ?thesis using b c assms(3) consistent-def by blast
qed

lemma pred-is-dep: P m = ⟦ i ⟧ ⟶ i ∈ deps (Insert m)
  by (metis Un-iff deps.simps(1) extended.set-intros extended.simps(27)
      extended-to-set.simps(1) insert-message.exhaust-sel)

lemma succ-is-dep: S m = ⟦ i ⟧ ⟶ i ∈ deps (Insert m)
  by (metis Un-insert-right deps.simps(1) extended-to-set.simps(1) insertI1
      insert-message.exhaust-sel)

lemma a-subset:
  fixes M N a
  assumes M ⊆ N
  assumes a-conditions (insert-messages N) a
  shows a-conditions (insert-messages M) a
  using assms by (simp add:a-conditions-def insert-messages-def, blast)

definition delete-maybe :: 'ℐ ⇒ ('ℐ, 'Σ) message set ⇒ 'Σ ⇒ 'Σ option where
  delete-maybe i D s = (if Delete (DeleteMessage i) ∈ D then None else Some s)

definition to-woot-character ::
  ('ℐ, 'Σ) message set ⇒ ('ℐ, 'Σ) insert-message ⇒ ('ℐ, 'Σ) woot-character
  where
    to-woot-character D m = (
      case m of
        (InsertMessage l i u s) ⇒ InsertMessage l i u (delete-maybe i D s))

lemma to-woot-character-keeps-i [simp]: I (to-woot-character M m) = I m
  by (cases m, simp add:to-woot-character-def)

lemma to-woot-character-keeps-i-lifted [simp]:
  I ‘ to-woot-character M ‘ X = I ‘ X
  by (metis (no-types, lifting) image-cong image-image to-woot-character-keeps-i)

lemma to-woot-character-keeps-P [simp]: P (to-woot-character M m) = P m
  by (cases m, simp add:to-woot-character-def)

lemma to-woot-character-keeps-S [simp]: S (to-woot-character M m) = S m
  by (cases m, simp add:to-woot-character-def)

lemma to-woot-character-insert-no-eff:
  to-woot-character (insert (Insert m) M) = to-woot-character M
 by (rule HOL.ext, simp add:delete-maybe-def to-woot-character-def insert-message.case-eq-if)
```

definition *is-associated-string* ::
(*'a*, *'s*) *message set* ⇒ (*'a* :: *linorder*, *'s*) *woot-character list* ⇒ *bool*
where *is-associated-string M s* ≡ (
consistent M ∧
set s = *to-woot-character M* ‘ (*insert-messages M*) ∧
(∀ *a*. *a-conditions* (*insert-messages M*) *a* ⟶
sorted-wrt (<) (*map a* (*ext-ids s*))))

fun *is-certified-associated-string* **where**
is-certified-associated-string M (*Inr v*) = *is-associated-string M v* |
is-certified-associated-string M (*Inl* -) = *False*

lemma *associated-string-unique*:
assumes *is-associated-string M s*
assumes *is-associated-string M t*
shows *s* = *t*
using *assms*
apply (*simp add*:*ext-ids-def is-associated-string-def consistent-def*
sorted-wrt-append)
by (*metis sort-set-unique*)

lemma *is-certified-associated-string-unique*:
assumes *is-certified-associated-string M s*
assumes *is-certified-associated-string M t*
shows *s* = *t*
using *assms* **by** (*case-tac s*, *case-tac* [!] *t*, (*simp add*:*associated-string-unique*)+)

lemma *empty-consistent*: *consistent* {}
proof −
have *a-conditions* {} (λ*x*. (*case x of* ⊢ ⇒ ⊢ | ⊣ ⇒ ⊣))
by (*simp add*: *a-conditions-def*)
hence ∃ *f*. *a-conditions* {} *f* **by** *blast*
moreover have *wfP* (*depends-on* {}) **by** (*simp add*: *wfP-eq-minimal*)
ultimately show *?thesis* **by** (*simp add*:*consistent-def insert-messages-def*)
qed

lemma *empty-associated*: *is-associated-string* {} []
by (*simp add*:*is-associated-string-def insert-messages-def empty-consistent*
ext-ids-def a-conditions-def)

The empty set of messages is consistent and the associated string is the empty string.

end

5.4 Create Consistent

theory *CreateConsistent*
imports *CreateAlgorithms Consistency*
begin

lemma *nth-visible-inc'*:
assumes *sorted-wrt* (<) (*map a* (*ext-ids s*))
assumes *nth-visible s n* = *Inr i*
assumes *nth-visible s* (*Suc n*) = *Inr j*
shows *a i* < *a j*
proof −
have *subseq* (*ext-ids* (*filter is-visible s*)) (*ext-ids s*)
by (*simp add*: *ext-ids-def subseq-map*)
hence *sorted-wrt* (<) (*map a* (*ext-ids* (*filter is-visible s*)))
using *assms*(*1*) *subseq-imp-sorted sorted-wrt-map* **by** *blast*
moreover have *a*:*Suc n* < *length* (*ext-ids* (*filter is-visible s*))
apply (*rule classical*) **using** *assms*(*3*) **by** *simp*
ultimately show *?thesis* **using** *assms*(*2*) *assms*(*3*) **apply** (*simp*)
using *sorted-wrt-nth-less* **by** *fastforce*
qed

lemma *nth-visible-eff*:
assumes *nth-visible s n* = *Inr i*
shows *extended-to-set i* ⊆ *I* ‘ *set s*
proof −
have *i* ∈ *set* (*ext-ids* (*filter is-visible s*))
apply (*cases n* < *length* (*ext-ids* (*filter is-visible s*)))
using *assms* **by** *auto*
thus *?thesis*
apply (*simp add*: *ext-ids-def*)
using *extended.inject* **by** *auto*
qed

lemma *subset-mono*:
assumes *N* ⊆ *M*
shows *I* ‘ *insert-messages N* ⊆ *I* ‘ *insert-messages M*
proof −
have *insert-messages N* ⊆ *insert-messages M* **using** *assms*
by (*metis* (*no-types*, *lifting*) *Collect-mono-iff insert-messages-def subsetCE*)
thus *?thesis* **by** (*simp add*: *image-mono*)
qed

lemma *deps-insert*:
assumes ⋃ (*deps* ‘ *M*) ⊆ (*I* ‘ *insert-messages M*)
assumes *deps m* ⊆ *I* ‘ *insert-messages M*
shows ⋃ (*deps* ‘ (*M* ∪ {*m*})) ⊆ (*I* ‘ *insert-messages* (*M* ∪ {*m*}))
proof −
have *deps m* ⊆ *I* ‘ *insert-messages* (*M* ∪ {*m*}) **using** *assms*(*2*) *subset-mono*
by (*metis Un-upper1 order-trans*)
thus *?thesis* **using** *assms*(*1*) **apply** (*simp*)
by (*meson rev-subsetD subsetI subset-insertI subset-mono*)
qed

lemma *wf-add*:
fixes $m :: ('a,'b)$ *insert-message*
assumes *wfP* (*depends-on M*)
assumes $\bigwedge n.\ n \in (M \cup \{m\}) \Longrightarrow I\ m \notin deps\ (Insert\ n)$
assumes $m \notin M$
shows *wfP* (*depends-on* $(M \cup \{m\})$)
proof −
have $\bigwedge Q.\ Q \neq \{\} \Longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y \in M \cup \{m\}) \wedge (z \in M \cup \{m\}) \wedge$
$I\ y \in deps\ (Insert\ z) \longrightarrow y \notin Q)$
proof −
fix $Q :: ('a,\ 'b)$ *insert-message set*
assume $b{:}Q \neq \{\}$
show $\exists\, z{\in}Q.\ \forall\, y.\ (y \in M \cup \{m\}) \wedge (z \in M \cup \{m\}) \wedge I\ y \in deps\ (Insert\ z)$
$\longrightarrow y \notin Q$
proof (*cases* $\exists\, x.\ x \in Q - \{m\}$)
case *True*
hence $\exists\, z{\in}\ Q - \{m\}.\ \forall\, y.\ (y \in M) \wedge (z \in M) \wedge I\ y \in deps\ (Insert\ z)$
$\longrightarrow y \notin Q - \{m\}$
by (*metis depends-on.simps assms*(*1*) *wfP-eq-minimal*)
then show *?thesis* **using** *assms*(*2*) *DiffD2* **by** *auto*
next
case *False*
hence $Q = \{m\}$ **using** *b* **by** *blast*
thus *?thesis* **using** *assms*(*2*) **by** *blast*
qed
qed
thus *?thesis* **by** (*simp add*:*wfP-eq-minimal*, *blast*)
qed

lemma *create-insert-p-s-ordered*:
assumes *is-associated-string N s*
assumes *a-conditions* (*insert-messages N*) *a*
assumes *Inr* (*Insert m*) = *create-insert s n σ new-id*
shows *a* (*P m*) < *a* (*S m*)
proof −
obtain *p q* **where** *pq-def*:
create-insert s n σ new-id = *Inr* (*Insert* (*InsertMessage p new-id q σ*))
by (*metis* (*no-types*, *lifting*) *One-nat-def add.right-neutral add-Suc-right*
create-insert.elims sum.case-eq-if sum.simps(*4*) *assms*(*3*) *bind-def*)
have *Inr p* = *nth-visible s n* **using** *pq-def Error-Monad.bindE* **by** *fastforce*
moreover have *Inr q* = *nth-visible s* (*Suc n*)
using *pq-def Error-Monad.bindE* **by** *fastforce*
ultimately have *a p* < *a q*
using *assms* **by** (*metis is-associated-string-def nth-visible-inc*′)
moreover have *m* = *InsertMessage p new-id q σ*
using *assms*(*3*) *pq-def* **by** *auto*
ultimately show *?thesis* **by** (*simp add*: *pq-def*)
qed

lemma *create-insert-consistent*:
assumes *consistent M*
assumes *is-associated-string N s*
assumes $N \subseteq M$
assumes *Inr m = create-insert s n σ new-id*
assumes *new-id ∉ I ‘ insert-messages M*
shows *consistent* $(M \cup \{m\})$
proof −
obtain *p q* **where** *pq-def*:
create-insert s n σ new-id = Inr (*Insert* (*InsertMessage p new-id q σ*))
by (*metis* (*no-types*, *lifting*) *One-nat-def add.right-neutral add-Suc-right create-insert.elims assms*(*4*) *sum.case-eq-if sum.simps*(*4*) *bind-def*)
define m' **where** $m' =$ *InsertMessage p new-id q σ*
hence *a*:$m =$ *Insert* m' **using** *pq-def assms*(*4*) **by** *auto*
hence *d*: *create-insert s n σ new-id = Inr* (*Insert* m')
using *pq-def assms* **by** *simp*
have *b*:$I\ m' =$ *new-id* **using** m'*-def* **by** (*simp add*:*I-def*)
hence *inj-on I* (*insert-messages M* $\cup \{m'\}$) **using** *assms*(*5*) *assms*(*1*)
using *consistent-def* **by** *fastforce*
hence *inj-on I* (*insert-messages* $(M \cup \{m\})$) **using** *assms*(*4*) *pq-def* m'*-def*
by (*metis Inr-inject insert-insert-message*)
moreover
have *p*:*extended-to-set p* $\subseteq$ *I ‘ set s* **using** *pq-def nth-visible-eff* **by** *fastforce*
have *q*: *extended-to-set q* $\subseteq$ *I ‘ set s*
using *pq-def* **apply** (*simp add*:*bind-def del*:*nth-visible.simps*)
apply (*cases nth-visible s n*, *simp*)
by (*cases nth-visible s* (*Suc n*), *simp*, *simp add*: *nth-visible-eff*)
have *extended-to-set p* $\cup$ *extended-to-set q* $\subseteq$ *I ‘ set s* **using** *p q* **by** *simp*
hence *extended-to-set p* $\cup$ *extended-to-set q* $\subseteq$ *I ‘ insert-messages N*
by (*metis assms*(*2*) *is-associated-string-def to-woot-character-keeps-i-lifted*)
hence *extended-to-set p* $\cup$ *extended-to-set q* $\subseteq$ *I ‘ insert-messages M*
using *assms*(*3*) *subset-mono* **by** *blast*
hence *c*:*deps m* $\subseteq$ *I ‘ insert-messages M* **using** *pq-def assms*(*4*) **by** *auto*
hence $\bigcup$ (*deps ‘* $(M \cup \{m\})$) $\subseteq$ (*I ‘ insert-messages* $(M \cup \{m\})$)
by (*metis consistent-def assms*(*1*) *deps-insert*)
moreover have *w*:
$\forall\, n \in$ *insert-messages* $M \cup \{m'\}$. *deps* (*Insert n*) $\subseteq$ *I ‘ insert-messages M*
by (*metis a c consistent-def assms*(*1*) *Sup-le-iff imageI insert-iff insert-is-Un insert-messages-def mem-Collect-eq sup.commute*)
hence $\forall\, n \in$ *insert-messages* $M \cup \{m'\}$. *I* $m' \notin$ *deps* (*Insert n*)
using *b assms*(*5*) **by** *blast*
hence *wfP* (*depends-on* (*insert-messages* $M \cup \{m'\}$))
by (*metis Un-insert-right insert-absorb wf-add assms*(*1*) *consistent-def sup-bot.right-neutral*)
moreover obtain *a* **where** *a-def*: *a-conditions* (*insert-messages M*) *a*
using *consistent-def assms*(*1*) **by** *blast*
define a' **where**
$a' = (\lambda i.$ *if* $i =$ ⟦ *new-id* ⟧ *then* ⟦Ψ (*a* (*P* m'), *a*(*S* m')) *new-id*⟧ *else a i*)
hence *a-conditions* (*insert-messages* $(M \cup \{m\})$) a'

```
  proof −
    have a' ⊢ < a' ⊣ using a'-def a-conditions-def a-def by auto
    moreover have
      ⋀m''. m'' ∈ (insert-messages M ∪ {m'}) ⟶
          a'(P m'') < a'(S m'')  ∧
          a' ⟦I m''⟧ = ⟦Ψ (a'(P m''), a'(S m'')) (I m'')⟧
    proof
      fix m''
      assume e: m'' ∈ (insert-messages M ∪ {m'})
      show a'(P m'') < a'(S m'') ∧ a' ⟦ I m''⟧ =
           ⟦Ψ (a'(P m''), a'(S m'')) (I m'')⟧
      proof (cases m'' ∈ insert-messages M)
        case True
        moreover have deps (Insert m'') ⊆ I ‘ insert-messages M
          using e w by blast
        hence P m'' ≠ ⟦ new-id ⟧ ∧ S m'' ≠ ⟦ new-id ⟧
          by (meson assms(5) contra-subsetD pred-is-dep succ-is-dep)
        moreover have I m'' ≠ new-id
          using assms(5) True by blast
        ultimately show ?thesis using a-def True
          by (simp add: a-conditions-def a'-def)
      next
        case False
        moreover have I m'' = new-id using False b e by blast
        moreover have deps (Insert m'') ⊆ I ‘ insert-messages M
          using False a c e by blast
        hence P m'' ≠ ⟦ new-id ⟧ ∧ S m'' ≠ ⟦ new-id ⟧
          by (meson assms(5) contra-subsetD pred-is-dep succ-is-dep)
        moreover have a-conditions (insert-messages N) a
          using a-def a-subset assms is-associated-string-def by blast
        hence a (P m') < a (S m')
          by (metis assms(2) d create-insert-p-s-ordered)
        hence a' (P m'') < a' (S m'') using calculation a'-def False e by auto
        ultimately show ?thesis using e a'-def by auto
      qed
    qed
    ultimately show ?thesis using a-conditions-def
      by (metis a insert-insert-message)
  qed
  ultimately show ?thesis using consistent-def a by (metis insert-insert-message)
qed

lemma bind-simp: (x ≫= (λl. y l) = Inr r) ⟹ (y (projr x) = Inr r)
  using isOK-I by force

lemma create-delete-consistent:
  assumes consistent M
  assumes is-associated-string N s
  assumes N ⊆ M
```

assumes *Inr m = create-delete s n*
shows *consistent (M ∪ {m})*
proof −
obtain *i* **where** *pq-def*: *create-delete s n = Inr (Delete (DeleteMessage i))*
by (*metis* (*no-types*, *lifting*) *Error-Monad.bindE create-delete.simps assms*(*4*))
hence *a*: *m = Delete (DeleteMessage i)* **using** *assms*(*4*) **by** *auto*
hence *b*: *insert-messages (M ∪ {m}) = insert-messages M*
by (*simp add*:*insert-messages-def*)
have *n ≠ 0* **apply** (*rule classical*) **using** *pq-def* **by** (*simp add*:*bind-def ext-ids-def*)

then obtain *u* **where** *n = Suc u* **using** *not0-implies-Suc* **by** *blast*
then have *i ∈ I ‘ set s* **using** *pq-def*
apply (*cases u < length* (*filter is-visible s*))
apply (*simp add*:*bind-simp ext-ids-def nth-append*)
apply (*meson filter-is-subset imageI in-set-conv-nth subset-code*(*1*))
apply (*cases u = length* (*filter is-visible s*))
by (*simp add*:*bind-def ext-ids-def nth-append*)+
hence *i ∈ I ‘ insert-messages N* **using** *assms*
by (*metis is-associated-string-def to-woot-character-keeps-i-lifted*)
hence *c*:*deps m ⊆ I ‘ insert-messages M* **using** *a*
by (*metis assms*(*3*) *deps.simps*(*2*) *singletonD subsetCE subsetI subset-mono*)
then show *?thesis* **using** *assms*(*1*) *b* **by** (*simp add*:*consistent-def*)
qed

end

5.5 Termination Proof for *integrate-insert*

theory *IntegrateInsertCommute*
imports *IntegrateAlgorithm Consistency CreateConsistent*
begin

In the following we show that *integrate-insert* terminates. Note that, this does not yet imply that the return value will not be an error state.

lemma *substr-simp* [*simp*]: *substr s l u = nths s {k. l < Suc k ∧ Suc k < u}*
proof (*cases l ≤ length s*)
case *True*
have *set* (*nths* (*take l s*) {*k*. *l < Suc k ∧ Suc k < u*}) = {}
by (*simp add*:*set-nths*)
hence *nths* (*take l s*) {*k*. *l < Suc k ∧ Suc k < u*} = [] **by** *blast*
moreover have {*j*. *Suc* (*j* + *l*) < *u*} = {..<(*u*−*Suc l*)} **by** *auto*
moreover have *min* (*length s*) *l* = *l* **using** *True* **by** *auto*
ultimately
have *nths* (*take l s* @ *drop l s*) {*k*. *l < Suc k ∧ Suc k < u*} = *substr s l u*
by (*simp add*:*nths-append del*:*append-take-drop-id*)
then show *?thesis* **by** *simp*
next
case *False*
hence *set* (*nths s* {*k*. *l < Suc k ∧ Suc k < u*}) = {}

by (*simp add*:*set-nths*)
hence *nths s* {*k*. *l* < *Suc k* ∧ *Suc k* < *u*} = [] **by** *blast*
thus *?thesis* **using** *False* **by** *simp*
qed

declare *substr.simps* [*simp del*]

Instead of simplifying *substr* with its definition we use *substr-simp* as a simplification rule. The right hand side of *substr-simp* is a better representation within proofs. However, we cannot directly define *substr* using the right hand side as it is not constructible term for Isabelle.

lemma *int-ins-loop-term-1*:
assumes *isOK* (*mapM* (*concurrent w l u*) *t*)
assumes *x* ∈ *set* (*concat* (*projr* (*mapM* (*concurrent w l u*) *t*)))
shows *x* ∈ (*InString* ∘ *I*) ‘ (*set t*)
using *assms*
by (*induction t*, *simp*, *simp add*: *bind-simp del*:*idx.simps set-concat*, *blast*)

lemma *fromSingleton-simp*: (*fromSingleton xs* = *Inr x*) = ([*x*] = *xs*)
by (*cases xs rule*: *fromSingleton.cases*, *auto*)

lemma *filt-simp*: ([*b*] = *filter p* [*0*..<*n*]) =
(*p b* ∧ *b* < *n* ∧ (∀ *y* < *n*. *p y* ⟶ *b* = *y*))
apply (*induction n*, *simp*, *simp*)
by (*metis atLeast-upt cancel-comm-monoid-add-class.diff-cancel*
filter-empty-conv lessThan-iff less-Suc-eq neq0-conv zero-less-diff)

lemma *substr-eff*:
assumes *x* ∈ (*InString* ∘ *I*) ‘ *set* (*substr w l u*)
assumes *isOK* (*idx w x*)
shows *l* < (*projr* (*idx w x*)) ∧ (*projr* (*idx w x*)) < *u*
proof −
obtain *i* **where** *i-def*: *idx w x* = *Inr i* **using** *assms*(*2*) **by** *blast*
then have *l* < *i* ∧ *i* < *u* **using** *assms*(*1*)
apply (*simp add*: *set-nths image-iff fromSingleton-simp filt-simp*)
apply (*simp add*:*ext-ids-def*)
by (*metis* (*no-types*, *lifting*) *Suc-mono length-map less-SucI list-update-id*
list-update-same-conv map-update nth-Cons-Suc nth-append)
thus *?thesis* **using** *i-def* **by** *auto*
qed

lemma *find-zip*:
assumes *find* (*cond* ∘ *snd*) (*zip* (*p*#*v*) (*v*@[*s*])) = *Some* (*x*,*y*)
assumes *v* ≠ []
shows
cond y
x ∈ *set v* ∨ *y* ∈ *set v*
x = *p* ∨ (*x* ∈ *set v* ∧ ¬(*cond x*))
y = *s* ∨ (*y* ∈ *set v*)

```
proof −
  obtain i where i-def:
    i < Suc (length v)
    (zip (p#v) (v@[s])) ! i = (x,y)
    cond y
    ∀j. j < i ⟶ ¬(cond ((v@[s])!j))
    using assms apply (simp add:find-Some-iff) by force
  show cond y using i-def by auto
  show x ∈ set v ∨ y ∈ set v using assms(2) i-def(1,2)
   by (metis fst-conv in-set-conv-nth length-0-conv length-Cons length-append-singleton
       less-Suc-eq less-Suc-eq-0-disj nth-Cons-Suc nth-append nth-zip snd-conv)
  show x = p ∨ (x ∈ set v ∧ (¬(cond x)))
    apply (cases i)
    using i-def(2) apply auto[1]
    by (metis Suc-less-eq fst-conv i-def(1,2,4) length-Cons
       length-append-singleton lessI nth-Cons-Suc nth-append nth-mem nth-zip)
  show y = s ∨ y ∈ set v
    by (metis diff-is-0-eq' i-def(1,2) in-set-conv-nth length-Cons
      length-append-singleton less-Suc-eq-le nth-Cons-0 nth-append nth-zip snd-conv)
qed

fun int-ins-measure'
  where
    int-ins-measure' (m,w,p,s) = (
      do {
        l ← idx w p;
        u ← idx w s;
        assert (l < u);
        return (u − l)
      })

fun int-ins-measure
  where
    int-ins-measure (m,w,p,s) = case-sum (λe. 0) id (int-ins-measure' (m,w,p,s))
```

We show that during the iteration of *integrate-insert*, the arguments are decreasing with respect to *int-ins-measure*. Note, this means that the distance between the W-characters with identifiers p (resp. s) is decreasing.

```
lemma int-ins-loop-term:
  assumes idx w p = Inr l
  assumes idx w s = Inr u
  assumes mapM (concurrent w l u) (substr w l u) = Inr d
  assumes concat d ≠ []
  assumes find ((λx.⟦I m⟧ < x ∨ x = s) ∘ snd)
    (zip (p#concat d) (concat d@[s])) = Some r
  shows int-ins-measure (m, w, r) < u − l
proof −
  have a: ⋀x y. x ∈ set (concat d) ⟹ idx w x = Inr y ⟹ l < y ∧ y < u
    using int-ins-loop-term-1 substr-eff assms(3) by (metis isOK-I sum.sel(2))
```

```
  hence b: l < u using assms
    by (metis concat.simps(1) diff-is-0-eq less-imp-le-nat
        mapM.simps(1) not-less-eq substr.simps sum.sel(2) take0)
  obtain p' s' where ps-def: r = (p', s') by (cases r, simp+)
  show ?thesis
  proof (cases int-ins-measure' (m, w, r))
    case (Inl a)
    then show ?thesis using b by (simp add:ps-def)
  next
    case (Inr b)
    then obtain l' u' where ps'-def: idx w p' = Inr l' idx w s' = Inr u'
      using ps-def apply (simp add:bind-simp del:idx.simps) by blast
    then have l' ≥ l ∧ l' < u ∧ u' > l ∧ u' ≤ u ∧ (l' > l ∨ u' < u)
      using a b ps-def find-zip(2,3,4) assms(1,2,4,5)
      by (metis (no-types, lifting) Inr-inject order.order-iff-strict)
    thus ?thesis using ps-def ps'-def apply (simp add:bind-simp del:idx.simps)
      by (cases l' < u', simp del:idx.simps, linarith, simp del:idx.simps)
  qed
qed

lemma assert-ok-simp [simp]: (assert p = Inr z) = p by (cases p, simp+)

termination integrate-insert
  apply (relation measure int-ins-measure, simp)
  using int-ins-loop-term by (simp del:idx.simps, blast)
```

5.6 Integrate Commutes

```
locale integrate-insert-commute =
  fixes M :: ('a :: linorder, 's) message set
  fixes a :: 'a extended ⇒ 'a position
  fixes s :: ('a, 's) woot-character list
  assumes associated-string-assm: is-associated-string M s
  assumes a-conditions-assm: a-conditions (insert-messages M) a
begin

lemma dist-ext-ids: distinct (ext-ids s)
  using associated-string-assm a-conditions-assm
  apply (simp add:is-associated-string-def sorted-wrt-map)
  by (metis (mono-tags) irreflp-def le-less not-le sorted-wrt-irrefl-distinct)

lemma I-inj-on-S:
  l < length s ∧ u < length s ∧ I(s ! l) = I(s ! u) ⟹ l = u
  using dist-ext-ids apply (simp add:ext-ids-def)
  using nth-eq-iff-index-eq by fastforce

lemma idx-find:
  assumes x < length (ext-ids s)
  assumes ext-ids s ! x = i
```

shows *idx s i = Inr x*
using *assms dist-ext-ids nth-eq-iff-index-eq*
by (*simp add:filt-simp fromSingleton-simp*, *blast*)

lemma *obtain-idx*:
assumes $x \in$ *set* (*ext-ids s*)
shows $\exists\, i.$ *idx s x = Inr i*
using *idx-find assms* **by** (*metis in-set-conv-nth*)

lemma *sorted-a*:
assumes *idx s x = Inr l*
assumes *idx s y = Inr u*
shows $(l \leq u) = (a\ x \leq a\ y)$
proof −
have *sorted-wrt* (<) (*map a* (*ext-ids s*))
using *associated-string-assm a-conditions-assm is-associated-string-def* **by** *blast*
then show *?thesis*
using *assms* **apply** (*simp add:filt-simp fromSingleton-simp*)
by (*metis leD leI le-less length-map nth-map sorted-wrt-nth-less*)
qed

lemma *sorted-a-le*: *idx s x = Inr l* $\Longrightarrow$ *idx s y = Inr u* $\Longrightarrow$ $(l < u) = (a\ x < a\ y)$
by (*meson sorted-a not-le*)

lemma *idx-intro-ext*: $i <$ *length* (*ext-ids s*) $\Longrightarrow$ *idx s* (*ext-ids s* ! *i*) = *Inr i*
using *dist-ext-ids* **by** (*simp add:fromSingleton-simp filt-simp nth-eq-iff-index-eq*)

lemma *idx-intro*:
assumes $i <$ *length s*
shows *idx s* $[\![I\ (s\ !\ i)]\!]$ = *Inr* (*Suc i*)
proof −
have *ext-ids s* ! (*Suc i*) = $[\![I\ (s\ !\ i)]\!] \wedge$ *Suc i* < *length* (*ext-ids s*)
using *assms* **by** (*simp add:ext-ids-def nth-append*)
thus *?thesis* **using** *idx-intro-ext* **by** *force*
qed

end

locale *integrate-insert-commute-insert = integrate-insert-commute* +
fixes *m*
assumes *consistent-assm*: *consistent* ($M \cup \{$*Insert m*$\}$)
assumes *insert-assm*: *Insert m* $\notin M$
assumes *a-conditions-assm-2*:
a-conditions (*insert-messages* ($M \cup \{$*Insert m*$\}$)) *a*
begin

definition *invariant* **where**
invariant pm sm = (*pm* $\in$ *set* (*ext-ids s*) $\wedge$ *sm* $\in$ *set* (*ext-ids s*) $\wedge$
subset (*a pm*, *a sm*) (*a* (*P m*), *a* (*S m*)) $\wedge$

```
   elem (a ⟦I m⟧) (a pm, a sm))

fun is-concurrent where
  is-concurrent pm sm x = (x ∈ set s ∧
   subset (a pm, a sm) (a (P x), a (S x)) ∧
   elem (a ⟦I x⟧) (a pm, a sm))

lemma no-id-collision: I m ∉ I ‘ insert-messages M
proof −
  have inj-on I (insert-messages (M ∪ {Insert m}))
    using consistent-def consistent-assm by fastforce
  hence I m ∈ I ‘ insert-messages M ⟶ Insert m ∈ M
    by (simp add: image-iff inj-on-eq-iff insert-messages-def)
  thus ?thesis using insert-assm by blast
qed

lemma not-deleted: to-woot-char m = to-woot-character M m
proof −
  have Delete (DeleteMessage (I m)) ∉ M
  proof
    assume Delete (DeleteMessage (I m)) ∈ M
    hence deps (Delete (DeleteMessage (I m))) ⊆ I ‘ insert-messages M
      using consistent-assm associated-string-assm
      apply (simp add:consistent-def is-associated-string-def)
      using image-subset-iff by fastforce
    thus False using no-id-collision by simp
  qed
  thus to-woot-char m = to-woot-character M m
    by (cases m, simp add:to-woot-character-def delete-maybe-def)
qed

lemma invariant-imp-sorted:
  assumes Suc l < length (ext-ids s)
  assumes a(ext-ids s ! l) < a ⟦I m⟧ ∧ a ⟦I m⟧ < a(ext-ids s ! (l+1))
  shows sorted-wrt (<) (map a (ext-ids ((take l s)@to-woot-char m#drop l s)))
proof −
  have l ≤ length s using assms(1) by (simp add:ext-ids-def)
  hence ext-ids (take l s@to-woot-char m#drop l s) =
        (take (Suc l) (ext-ids s))@⟦I m⟧#(drop (Suc l) (ext-ids s))
    by (cases m, simp add:ext-ids-def take-map drop-map)
  thus ?thesis
    using assms associated-string-assm is-associated-string-def a-conditions-assm
    apply (simp flip:take-map drop-map)
    by (rule insort, simp+, blast)
qed

lemma no-self-dep: ¬ depends-on (insert-messages M ∪ {m}) m m
proof −
  have wfP (depends-on (insert-messages M ∪ {m}))
```

```
    using consistent-assm
    apply (simp add:consistent-def)
    by (metis Un-insert-right insert-insert-message sup-bot.right-neutral)
  thus ?thesis
    by (metis mem-Collect-eq wfP-eq-minimal)
qed

lemma pred-succ-order:
  m' ∈ (insert-messages M ∪ {m}) ⟹ a(P m') < a ⟦I m'⟧ ∧ a(S m') > a ⟦I m'⟧
  by (metis elem.simps is-interval.simps psi-elem a-conditions-def
      a-conditions-assm-2 insert-insert-message)

lemma find-dep:
  assumes Insert m' ∈ (M ∪ {Insert m})
  assumes i ∈ deps (Insert m')
  shows ⟦i⟧ ∈ set (ext-ids s)
proof −
  have i ∈ I ‘ insert-messages M
  proof (cases m' = m)
    case True
    hence i ∈ I ‘ insert-messages (M ∪ {Insert m})
      using assms consistent-assm
      by (simp add:consistent-def, blast)
    moreover have i ≠ I m using assms True no-self-dep by auto
    ultimately show ?thesis
      by (metis (no-types, lifting) UnE image-Un image-empty image-insert
          insert-insert-message singletonD)
  next
    case False
    hence Insert m' ∈ M using assms by simp
    then show i ∈ I ‘ insert-messages M
      using assms is-associated-string-def associated-string-assm consistent-def
      by (metis (no-types, opaque-lifting) Union-iff contra-subsetD image-iff)
  qed
  hence i ∈ I ‘ (set s)
    using associated-string-assm by (simp add:is-associated-string-def)
  thus ⟦i⟧ ∈ set (ext-ids s)
    by (simp add:ext-ids-def image-iff)
qed

lemma find-pred:
  m' ∈ (insert-messages M ∪ {m}) ⟹ P m' ∈ set (ext-ids s)
 using find-dep by (cases P m', (simp add:ext-ids-def insert-messages-def pred-is-dep)+)

lemma find-succ:
  m' ∈ (insert-messages M ∪ {m}) ⟹ S m' ∈ set (ext-ids s)
 using find-dep by (cases S m', (simp add:ext-ids-def insert-messages-def succ-is-dep)+)

fun is-certified-associated-string' where
```

```
  is-certified-associated-string' (Inr v) = (
    set v = to-woot-character (M ∪ {Insert m}) ‘
      (insert-messages (M ∪ {Insert m})) ∧
    sorted-wrt (<) (map a (ext-ids v))) |
  is-certified-associated-string' (Inl -) = False

lemma integrate-insert-final-step:
  assumes invariant pm sm
  assumes idx s pm = Inr l
  assumes idx s sm = Inr (Suc l)
  shows is-certified-associated-string' (Inr (take l s@(to-woot-char m)#drop l s))
proof −
  define t where t = (take l s@(to-woot-char m)#drop l s)
  hence set t = set s ∪ {to-woot-char m}
    by (metis Un-insert-right append-take-drop-id list.simps(15)
        set-append sup-bot.right-neutral)
  hence
    set t = to-woot-character M ‘ insert-messages M ∪ {to-woot-character M m}
    using not-deleted by (metis associated-string-assm is-associated-string-def)
  hence
     set t = to-woot-character (M ∪ {Insert m}) ‘ insert-messages (M ∪ {Insert
m})
    apply (simp add: to-woot-character-insert-no-eff)
    using insert-insert-message by fastforce
 moreover have sorted-wrt (<) (map a (ext-ids t)) using assms invariant-imp-sorted
    by (simp add:invariant-def fromSingleton-simp filt-simp t-def)
  ultimately show ?thesis
    using t-def associated-string-assm by (simp add:is-associated-string-def)
qed

lemma concurrent-eff:
  assumes idx s pm = Inr l
  assumes idx s sm = Inr u
  obtains d where mapM (concurrent s l u) (substr s l u) = Inr d ∧
    set (concat d) = InString ‘ I ‘ {x. is-concurrent pm sm x}
proof −
  define t where t = substr s l u
  have set t ⊆ set s ⟹ (isOK (mapM (concurrent s l u) t) ∧
    set (concat (projr (mapM (concurrent s l u) t))) =
    InString ‘ I ‘ {x. x ∈ set t ∧ a (P x) ≤ a pm ∧ a (S x) ≥ a sm})
  proof (induction t)
    case Nil
    then show ?case by simp
  next
    case (Cons th tt)
    hence th ∈ to-woot-character M ‘ insert-messages M
      using associated-string-assm by (simp add: is-associated-string-def)
    then obtain th' where th'-def:
      th' ∈ insert-messages M ∧ P th' = P th ∧ S th' = S th
```

```
    by (metis image-iff to-woot-character-keeps-P to-woot-character-keeps-S)
  obtain l' where l'-def: idx s (P th) = Inr l'
    using th'-def find-pred obtain-idx by fastforce
  obtain u' where u'-def: idx s (S th) = Inr u'
    using th'-def find-succ obtain-idx by fastforce
  have {x. x = ⟦I th⟧ ∧ l' ≤ l ∧ u ≤ u'} =
    InString ‘ I ‘ {x. x = th ∧ a (P x) ≤ a pm ∧ a sm ≤ a (S x)}
    using sorted-a l'-def u'-def assms
    by (rule-tac set-eqI, simp add:image-iff, blast)
  then show ?case
    using Cons
    by (simp add:bind-simp l'-def u'-def
        concurrent.simps[where w=th] del:idx.simps, auto)
 qed
 moreover have
  ⋀x. (x ∈ set (substr s l u)) = (x ∈ set s ∧ a pm < a ⟦I x⟧ ∧ a ⟦I x⟧ < a sm)
  apply (simp add:set-nths in-set-conv-nth)
  using sorted-a-le idx-intro assms by blast
 ultimately have
  isOK (mapM (concurrent s l u) (substr s l u)) ∧
  set (concat (projr (mapM (concurrent s l u) (substr s l u)))) =
    InString ‘ I ‘ {x. is-concurrent pm sm x}
  by (simp only:t-def, fastforce)
 thus ?thesis using that by auto
qed

lemma concurrent-eff-2:
 assumes invariant pm sm
 assumes is-concurrent pm sm x
 shows preserve-order ⟦I x⟧ ⟦I m⟧ (a ⟦I x⟧) (a ⟦I m⟧)
proof −
 have x ∈ to-woot-character M ‘ insert-messages M
  using assms(2) associated-string-assm is-associated-string-def
    is-concurrent.elims(2) by blast
 then obtain x' where x'-def: I x = I x' ∧ P x = P x' ∧ S x = S x' ∧ x' ∈
insert-messages M
  using to-woot-character-keeps-P to-woot-character-keeps-S
    to-woot-character-keeps-i by fastforce
 have elem (a ⟦I x⟧) (a (P m), a (S m))
  using assms by (simp add: invariant-def, auto)
 moreover have elem (a ⟦I m⟧) (a (P x), a (S x))
  using assms by (simp add: invariant-def, auto)
 moreover have a-conditions (insert-messages M ∪ {m}) a
  by (metis insert-insert-message a-conditions-assm-2)
 ultimately have preserve-order (I x) (I m) (a ⟦I x⟧) (a ⟦I m⟧)
  by (simp add: a-conditions-def psi-preserve-order x'-def)
 thus ?thesis by (simp add: preserve-order-def)
qed
```

lemma *concurrent-eff-3*:
assumes *idx s pm* = *Inr l*
assumes *idx s sm* = *Inr u*
assumes *Suc l* < *u*
shows {*x*. *is-concurrent pm sm x*} ≠ {}
proof −
define *H* **where**
H = {*x*. *x* ∈ *insert-messages M* ∧ *a pm* < *a* ⟦*I x*⟧ ∧ *a* ⟦*I x*⟧ < *a sm*}
have *wfP* (*depends-on* (*insert-messages M*))
using *associated-string-assm* **by** (*simp add*: *consistent-def is-associated-string-def*)
moreover have *f*:*H* ⊆ *insert-messages M* **using** *H-def* **by** *blast*
hence *depends-on H* ≤ *depends-on* (*insert-messages M*) **by** *auto*
ultimately have *wfP* (*depends-on H*) **using** *wf-subset* [*to-pred*] **by** *blast*
moreover
have *u*: *l* < *length s* **using** *assms*(*2*) *assms*(*3*)
by (*simp add*:*fromSingleton-simp filt-simp*, *simp add*:*ext-ids-def*)
hence *v*:*a pm* < *a* ⟦*I*(*s* ! *l*)⟧ ∧ *a* ⟦*I*(*s* ! *l*)⟧ < *a sm*
using *sorted-a-le assms u idx-intro* **by** *blast*
have *I* (*s* ! *l*) ∈ *I* ‘ *insert-messages M*
by (*metis image-eqI associated-string-assm is-associated-string-def nth-mem*
to-woot-character-keeps-i-lifted u)
hence ∃ *x*. *x* ∈ *H* **using** *v H-def* **by** *auto*
ultimately obtain *z* **where** *z-def*: *z* ∈ *H* ⋀ *y*. *depends-on H y z* ⟹ *y* ∉ *H*
by (*metis wfP-eq-minimal*)
have *a*:⋀*x*. *x* ∈ *deps* (*Insert z*) ⟹ ¬(*a pm* < *a* ⟦*x*⟧ ∧ *a* ⟦*x*⟧ < *a sm*)
proof −
fix *x*
assume *a*:*x* ∈ *deps* (*Insert z*)
hence *x* ∈ *I* ‘ *insert-messages M*
using *insert-messages-def associated-string-assm*
apply (*simp add*:*consistent-def is-associated-string-def*)
using *H-def z-def*(*1*) **by** *blast*
then obtain *x*′ **where** *x*′*-def*: *x*′ ∈ *insert-messages M* ∧ *x* = *I x*′ **by** *blast*
hence *x*′ ∉ *H* **using** *z-def*
using *a depends-on.simps* **by** *blast*
thus ¬(*a pm* < *a* ⟦*x*⟧ ∧ *a* ⟦*x*⟧ < *a sm*) **using** *H-def x*′*-def* **by** *blast*
qed
have *ext-ids s* ! *0* = ⊢ ∧ *0* < *length* (*ext-ids s*) **by** (*simp add*:*ext-ids-def*)
hence *b*:¬(*a pm* < *a* ⊢)
by (*metis not-less-zero sorted-a-le assms*(*1*) *idx-intro-ext*)
have *ext-ids s* ! (*Suc* (*length s*)) = ⊣ ∧ *Suc* (*length s*) < *length* (*ext-ids s*)
by (*simp add*:*nth-append ext-ids-def*)
moreover have ¬(*Suc* (*length s*) < *u*) **using** *assms*(*2*)
by (*simp add*:*fromSingleton-simp filt-simp*, *simp add*:*ext-ids-def*)
ultimately have *c*:¬(*a* ⊣ < *a sm*) **by** (*metis sorted-a-le assms*(*2*) *idx-intro-ext*)
have *d*:*a* (*P z*) ≤ *a pm*
using *a b c pred-is-dep pred-succ-order H-def z-def*(*1*) **by** (*cases P z*, *fastforce*+)
have *e*:*a* (*S z*) ≥ *a sm*
using *a b c succ-is-dep pred-succ-order H-def z-def*(*1*) **by** (*cases S z*, *fastforce*+)

```
  have to-woot-character M z ∈ set s
    using f associated-string-assm is-associated-string-def z-def(1) by fastforce
  hence is-concurrent pm sm (to-woot-character M z)
    using H-def z-def(1) d e by simp
  thus ?thesis by blast
qed

lemma integrate-insert-result-helper:
  invariant pm sm ⟹ m' = m ⟹ s' = s ⟹
  is-certified-associated-string' (integrate-insert m' s' pm sm)
proof (induction m' s' pm sm rule:integrate-insert.induct)
  case (1 m' s' pm sm)
  obtain l where l-def: idx s pm = Inr l
    using 1(2) invariant-def obtain-idx by blast
  obtain u where u-def: idx s sm = Inr u
    using 1(2) invariant-def obtain-idx by blast
  show ?case
  proof (cases Suc l = u)
    case True
    then show ?thesis
    apply (simp add:l-def u-def 1 del:idx.simps is-certified-associated-string'.simps)
      using 1(2) l-def u-def integrate-insert-final-step by blast
  next
    case False
    have a pm < a sm using invariant-def 1(2) by auto
    hence a:l < u using sorted-a-le  l-def u-def by blast
    obtain d where d-def: mapM (concurrent s l u) (substr s l u) = Inr d ∧
      set (concat d) = InString ‘ I ‘ {x. is-concurrent pm sm x}
      by (metis concurrent-eff l-def u-def)
    have b:concat d ≠ []
      by (metis Suc-lessI concurrent-eff-3 False l-def u-def
          a d-def empty-set image-is-empty)
    have c:⋀x. x ∈ set (concat d) ⟹
      preserve-order x ⟦I m⟧ (a x) (a ⟦I m⟧) ∧ x ∈ set (ext-ids s) ∧
       a pm < a x ∧ a x < a sm
      using 1(2) d-def concurrent-eff-2
      by (simp del:set-concat add:ext-ids-def, blast)
    obtain pm' sm' where ps'-def: find ((λx.⟦I m⟧ < x ∨ x = sm) ∘ snd)
        (zip (pm # concat d) (concat d @ [sm])) = Some (pm',sm')
      (is ?lhs = ?rhs)
      apply (cases ?lhs)
      apply (simp add:find-None-iff)
       apply (metis in-set-conv-decomp in-set-impl-in-set-zip2 length-Cons
            length-append-singleton)
      by fastforce
    have d:pm' = pm ∨ pm' ∈ set (concat d) using ps'-def b
      by (metis (full-types) find-zip(3))
    hence pm' ∈ set (ext-ids s) using c 1(2) invariant-def by auto
    hence pm' ∈ InString ‘ I ‘ insert-messages M ∨ pm' = ⊢ ∨ pm' = ⊣
```

apply (*simp add*:*ext-ids-def*)
by (*metis image-image associated-string-assm is-associated-string-def to-woot-character-keeps-i-lifted*)
hence $pm' \neq [\![I\ m]\!]$ **using** *no-id-collision* **by** *blast*
hence $(pm' = pm \vee pm' < [\![I\ m]\!]) \wedge (sm' = sm \vee sm' > [\![I\ m]\!] \wedge sm' \in set\ (concat\ d))$
by (*metis* (*mono-tags*, *lifting*) *ps'-def b find-zip*(*1*) *find-zip*(*3*) *find-zip*(*4*) *less-linear*)
hence *e*:*invariant pm' sm'*
using *1*(*2*) *c d* **apply** (*simp add*:*invariant-def del*:*set-concat*)
by (*meson dual-order.strict-trans leD leI preserve-order-def*)
show *?thesis* **apply** (*subst integrate-insert.simps*)
using *a b e ps'-def 1 d-def False l-def u-def*
by (*simp add*:*1 del*:*idx.simps integrate-insert.simps*)
qed
qed

lemma *integrate-insert-result*:
is-certified-associated-string' (*integrate-insert m s* (*P m*) (*S m*))
proof −
have *invariant* (*P m*) (*S m*)
using *find-pred find-succ pred-succ-order* **by** (*simp add*:*invariant-def*)
thus *?thesis* **using** *integrate-insert-result-helper* **by** *blast*
qed
end

lemma *integrate-insert-result*:
assumes *consistent* (*M* ∪ {*Insert m*})
assumes *Insert m* ∉ *M*
assumes *is-associated-string M s*
shows *is-certified-associated-string* (*M* ∪ {*Insert m*}) (*integrate-insert m s* (*P m*) (*S m*))
proof −
obtain *t* **where** *t-def*: (*integrate-insert m s* (*P m*) (*S m*)) = *Inr t* ∧
set t = *to-woot-character* (*M* ∪ {*Insert m*}) ‘ (*insert-messages* (*M* ∪ {*Insert m*}))
proof −
fix *tt*
assume *a*:(⋀*t*. (*integrate-insert m s* (*P m*) (*S m*)) = *Inr t* ∧
set t = *to-woot-character* (*M* ∪ {*Insert m*}) ‘ *insert-messages* (*M* ∪ {*Insert m*}) ⟹
tt)
obtain *a* **where** *a-def*: *a-conditions* (*insert-messages* (*M* ∪ {*Insert m*})) *a*
using *consistent-def assms* **by** *blast*
moreover have *a-conditions* (*insert-messages M*) *a*
using *assms a-subset is-associated-string-def a-def* **by** *blast*
ultimately interpret *integrate-insert-commute-insert M a s m*
using *assms* **by** (*simp add*: *integrate-insert-commute-insert-def integrate-insert-commute-def integrate-insert-commute-insert-axioms.intro*)

```
    show tt using a integrate-insert-result
      apply (cases integrate-insert m s (P m) (S m)) by auto
  qed
  have b:⋀a. a-conditions (insert-messages (M ∪ {Insert m})) a ⟹
    sorted-wrt (<) (map a (ext-ids t))
  proof −
    fix a
    assume c:a-conditions (insert-messages (M ∪ {Insert m})) a
    moreover have a-conditions (insert-messages M) a
      using assms a-subset is-associated-string-def c by blast
    ultimately interpret integrate-insert-commute-insert M a s m
    using assms by (simp add: integrate-insert-commute-insert-def integrate-insert-commute-def
integrate-insert-commute-insert-axioms.intro)
    show sorted-wrt (<) (map a (ext-ids t))
      using integrate-insert-result t-def by simp
  qed
  show ?thesis using b t-def assms(1) by (simp add:is-associated-string-def)
qed

locale integrate-insert-commute-delete = integrate-insert-commute +
  fixes m
  assumes consistent-assm: consistent (M ∪ {Delete m})
begin

fun delete :: ('a, 's) woot-character ⇒ ('a, 's) woot-character
  where delete (InsertMessage p i u -) = InsertMessage p i u None

definition delete-only-m :: ('a, 's) woot-character ⇒ ('a, 's) woot-character
  where delete-only-m x = (if DeleteMessage (I x) = m then delete x else x)

lemma set-s: set s = to-woot-character M ‘ insert-messages M
  using associated-string-assm by (simp add:is-associated-string-def)

lemma delete-only-m-effect:
  delete-only-m (to-woot-character M x) = to-woot-character (M ∪ {Delete m}) x
  apply (cases x, simp add:to-woot-character-def delete-maybe-def)
  by (metis delete-only-m-def insert-message.sel(2) delete.simps)

lemma integrate-delete-result:
  is-certified-associated-string (M ∪ {Delete m}) (integrate-delete m s)
proof (cases m)
  case (DeleteMessage i)
  have deps (Delete m) ⊆ I ‘ insert-messages (M ∪ {Delete m})
    using consistent-assm by (simp add:consistent-def DeleteMessage)
  hence i ∈ I ‘ insert-messages (M ∪ {Delete m}) using DeleteMessage by auto
  hence i ∈ I ‘ set s using set-s by (simp add:insert-messages-def)
  then obtain k where k-def: I (s ! k) = i ∧ k < length s
    by (metis imageE in-set-conv-nth)
  hence ext-ids s ! (Suc k) = ⟦i⟧ ∧ Suc k < length (ext-ids s)
```

by (*simp add*:*ext-ids-def nth-append*)
hence *g*:*idx s* ⟦*i*⟧ = *Inr* (*Suc k*) **apply** (*simp add*:*fromSingleton-simp filt-simp*)
using *dist-ext-ids nth-eq-iff-index-eq* **by** *fastforce*
moreover define *t* **where** *t* = *List.list-update s k* (*delete* (*s* ! *k*))
ultimately have *a*: *integrate-delete m s* = *Inr t*
using *k-def DeleteMessage* **by** (*cases s* ! *k*, *simp*)
have ⋀*j*. *j* < *length s* ⟹ (*DeleteMessage* (*I*(*s* ! *j*)) = *m*) = (*j* = *k*)
apply (*simp add*: *DeleteMessage*) **using** *I-inj-on-S k-def* **by** *blast*
hence *List.list-update s k* (*delete* (*s* ! *k*)) = *map delete-only-m s*
by (*rule-tac nth-equalityI*, (*simp add*:*k-def delete-only-m-def*)+)
hence *set t* = *delete-only-m* ‘ *set s* **using** *t-def* **by** *auto*
also have ... = *to-woot-character* (*M* ∪ {*Delete m*}) ‘ (*insert-messages M*)
using *set-s delete-only-m-effect image-cong* **by** (*metis* (*no-types*, *lifting*) *image-image*)
finally have *b*:
set t = *to-woot-character* (*M* ∪ {*Delete m*}) ‘ (*insert-messages* (*M* ∪ {*Delete m*}))
by (*simp add*: *insert-messages-def*)
have *ext-ids s* = *ext-ids t*
apply (*cases s* ! *k*, *simp add*:*t-def ext-ids-def*)
by (*metis* (*no-types*, *lifting*) *insert-message.sel*(*2*) *list-update-id map-update*)
moreover have ⋀*a*. *a-conditions* (*insert-messages M*) *a* ⟹ *sorted-wrt* (<) (*map a* (*ext-ids s*))
using *associated-string-assm is-associated-string-def* **by** *blast*
ultimately have *c*: ⋀*a*. *a-conditions* (*insert-messages* (*M* ∪ {*Delete m*})) *a*
⟹ *sorted-wrt* (<) (*map a* (*ext-ids t*))
by (*simp add*:*insert-messages-def*)
show *?thesis*
apply (*simp add*:*a is-associated-string-def b c*)
using *consistent-assm* **by** *fastforce*
qed
end

lemma *integrate-delete-result*:
assumes *consistent* (*M* ∪ {*Delete m*})
assumes *is-associated-string M s*
shows *is-certified-associated-string* (*M* ∪ {*Delete m*}) (*integrate-delete m s*)
proof −
obtain *a* **where** *a-def*: *a-conditions* (*insert-messages* (*M* ∪ {*Delete m*})) *a*
using *consistent-def assms* **by** *blast*
moreover have *a-conditions* (*insert-messages M*) *a*
using *assms a-subset is-associated-string-def a-def* **by** *blast*
ultimately interpret *integrate-insert-commute-delete M a s m*
using *assms* **by** (*simp add*: *integrate-insert-commute-def integrate-insert-commute-delete.intro*
integrate-insert-commute-delete-axioms.intro)
show *?thesis* **using** *integrate-delete-result* **by** *blast*
qed

fun *is-delete* :: (('*a*, '*s*) *message*) ⇒ *bool*

where
is-delete (*Insert m*) = *False* |
is-delete (*Delete m*) = *True*

proposition *integrate-insert-commute*:
assumes *consistent* (*M* ∪ {*m*})
assumes *is-delete m* ∨ *m* ∉ *M*
assumes *is-associated-string M s*
shows *is-certified-associated-string* (*M* ∪ {*m*}) (*integrate s m*)
using *assms integrate-insert-result integrate-delete-result* **by** (*cases m*, *fastforce*+)

end

5.7 Strong Convergence

theory *StrongConvergence*
imports *IntegrateInsertCommute CreateConsistent HOL.Finite-Set DistributedExecution*
begin

lemma (**in** *dist-execution*) *happened-before-same*:
assumes *i* < *j*
assumes *j* < *length* (*events k*)
shows (*happened-immediately-before*)$^{++}$ (*k*,*i*) (*k*,*j*)
proof −
obtain *v* **where** *v-def*: *j* = *Suc i* + *v* **using** *assms*(*1*) *less-iff-Suc-add* **by** *auto*
have *is-valid-event-id* (*k*, *Suc i* + *v*) ⟶ (*happened-immediately-before*)$^{++}$ (*k*,*i*) (*k*, *Suc i* + *v*)
apply (*induction v*, *simp add*: *tranclp.r-into-trancl*)
by (*metis Suc-lessD add-Suc-right fst-conv happened-immediately-before.elims*(*3*)
is-valid-event-id.simps snd-conv tranclp.simps)
then show *?thesis*
using *is-valid-event-id.simps v-def assms* **by** *blast*
qed

definition *make-set* **where** *make-set* (*k* :: *nat*) *p* = {*x*. ∃*j*. *p j x* ∧ *j* < *k*}

lemma *make-set-nil* [*simp*]: *make-set 0 p* = {} **by** (*simp add*:*make-set-def*)

lemma *make-set-suc* [*simp*]: *make-set* (*Suc k*) *p* = *make-set k p* ∪ {*x*. *p k x*}
using *less-Suc-eq* **by** (*simp add*:*make-set-def*, *blast*)

lemma (**in** *dist-execution*) *received-messages-eff*:
assumes *is-valid-state-id* (*i*,*j*)
shows *set* (*received-messages* (*i*,*j*)) = *make-set j* (λ*k x*. (∃ *s*. *event-at* (*i*, *k*) (*Receive s x*)))
using *assms* **by** (*induction j*, *simp add*:*make-set-def*, *simp add*: *take-Suc-conv-app-nth*)

lemma (**in** *dist-execution*) *finite-valid-event-ids*:
finite {*i*. *is-valid-event-id i*}
proof −
define *X* **where** *X* = {*p*. *events p* = *events p*}
have *finite X* ⟹ ∃ *m*. (∀ *p* ∈ *X*. (*length* (*events p*)) < *m*)
apply (*induction rule*:*finite-induct*, *simp*)
by (*metis gt-ex insert-iff le-less-trans less-imp-not-less not-le-imp-less*)
then obtain *m* **where** *m-def*: ⋀*p*. *length* (*events p*) < *m* **using** *X-def fin-peers*
by *auto*
have {(*i*,*j*). *is-valid-event-id* (*i*,*j*)} ⊆ {(*i*,*j*). *j* < *m*}
using *m-def* **by** (*simp add*: *Collect-mono-iff less-trans*)
also have ... ⊆ *X* × {*j*. *j* < *m*} **using** *X-def* **by** *blast*
finally have {*i*. *is-valid-event-id i*} ⊆ *X* × {*j*. *j* < *m*}
by (*simp add*: *subset-iff*)
thus *?thesis*
using *fin-peers finite-Collect-less-nat finite-cartesian-product*
infinite-super subset-eq
by (*metis UNIV-I*)
qed

lemma (**in** *dist-execution*) *send-insert-id-1*:
state i ⤜ (λ*s*. *create-insert s n σ i*) = *Inr* (*Insert m*) ⟹ *I m* = *i*
by *fastforce*

lemma (**in** *dist-execution*) *send-insert-id-2*:
state i ⤜ (λ*s*. *create-delete s n*) = *Inr* (*Insert m*) ⟹ *False*
by *fastforce*

lemma (**in** *dist-execution*) *send-insert-id*:
event-at i (*Send* (*Insert m*)) ⟹ *I m* = *i*
using *send-correct send-insert-id-1 send-insert-id-2* **by** *metis*

lemma (**in** *dist-execution*) *recv-insert-once*:
event-at (*i*,*j*) (*Receive s* (*Insert m*)) ⟹ *event-at* (*i*,*k*) (*Receive t* (*Insert m*)) ⟹
j = *k*
using *no-data-corruption send-insert-id at-most-once*
by (*simp*, *metis* (*mono-tags*) *Pair-inject event-pred.simps fst-conv is-valid-event-id.simps*)

proposition *integrate-insert-commute′*:
fixes *M m s*
assumes *consistent M*
assumes *is-delete m* ∨ *m* ∉ *T*
assumes *m* ∈ *M*
assumes *T* ⊆ *M*
assumes *deps m* ⊆ *I* ‘ *insert-messages T*
assumes *is-certified-associated-string T s*
shows *is-certified-associated-string* (*T* ∪ {*m*}) (*s* ⤜ (λ*t*. *integrate t m*))
proof (*cases s*)
case (*Inl a*)

then show *?thesis* **using** *assms* **by** *simp*
next
case (*Inr b*)
have $T \cup \{m\} \subseteq M$ **using** *assms*(*3*) *assms*(*4*) **by** *simp*
moreover have $\bigcup$ (*deps* ‘ ($T \cup \{m\}$)) $\subseteq$ *I* ‘ *insert-messages* ($T \cup \{m\}$)
using *assms*(*5*) *assms*(*6*) *Inr* **apply** (*simp add*:*is-associated-string-def consistent-def*)
by (*meson dual-order.trans subset-insertI subset-mono*)
ultimately have *consistent* ($T \cup \{m\}$)
using *assms consistent-subset* **by** *force*
then show *?thesis* **using** *integrate-insert-commute assms*(*2*) *assms*(*6*) *Inr* **by** *auto*
qed

lemma *foldM-rev*: *foldM f s* (*li*@[*ll*]) = *foldM f s li* $\gg\!=$ (λ*t*. *f t ll*)
by (*induction li arbitrary*:*s*, *simp*+)

lemma (**in** *dist-execution*) *state-is-associated-string′*:
fixes *i M*
assumes *j* $\leq$ *length* (*events i*)
assumes *consistent M*
assumes *make-set j* (λ*k m*. $\exists$ *s*. *event-at* (*i*,*k*) (*Receive s m*)) $\subseteq$ *M*
shows *is-certified-associated-string* (*make-set j* (λ*k m*. $\exists$ *s*. *event-at* (*i*,*k*) (*Receive s m*))) (*state* (*i*,*j*))
using *assms*
proof (*induction j*)
case *0*
then show *?case* **by** (*simp add*: *empty-associated*)
next
case (*Suc j*)
have *b*:*j* < *length* (*events i*) **using** *Suc* **by** *auto*
show *?case*
proof (*cases events i* ! *j*)
case (*Send m*)
then show *?thesis* **using** *Suc* **by** (*simp add*: *take-Suc-conv-app-nth*)
next
case (*Receive s m*)
moreover have *is-delete m* $\vee$ *m* $\notin$ (*make-set j* (λ*k m*. $\exists$ *s*. *event-at* (*i*,*k*) (*Receive s m*)))
apply (*cases m*) **using** *recv-insert-once Receive b* **apply** (*simp add*: *make-set-def*)

apply (*metis nat-neq-iff*)
by (*simp*)
moreover have *deps m* $\subseteq$ *I* ‘ *insert-messages* (*make-set j* (λ*k m*. $\exists$ *s*. *event-at* (*i*,*k*) (*Receive s m*)))
apply (*rule subsetI*)
using *semantic-causal-delivery Receive b* **apply** (*simp add*:*insert-messages-def image-iff make-set-def*) **by** *metis*
ultimately show *?thesis*

```
    using Suc apply (cases s, simp add:take-Suc-conv-app-nth foldM-rev)
    using integrate-insert-commute' by fastforce
  qed
qed

lemma (in dist-execution) sent-before-recv:
  assumes event-at (i,k) (Receive s m)
  assumes j < length (events i)
  assumes k < j
  shows event-at s (Send m) ∧ happened-immediately-before^++ s (i,j)
proof −
  have a:event-at s (Send m)
    using assms no-data-corruption by blast
  hence happened-immediately-before s (i,k)
    using assms by (cases s, simp, metis (mono-tags, lifting) event.simps(6))
  hence (happened-immediately-before)^++ s (i,j) using happened-before-same
    by (meson assms(2) assms(3) tranclp-into-tranclp2)
  thus ?thesis using a by blast
qed

lemma (in dist-execution) irrefl-p: irreflp (happened-immediately-before^++)
  by (meson acyclic-def dist-execution.acyclic-happened-before
      dist-execution-axioms irreflpI tranclp-unfold)

lemma (in dist-execution) new-messages-keep-consistency:
  assumes consistent M
  assumes event-at i (Send m)
  assumes set (received-messages i) ⊆ M
  assumes i ∉ I ‘ insert-messages M
  shows consistent (insert m M)
proof −
  have a:is-valid-state-id i using assms(2) by (cases i, simp)
  consider
    (1) (∃ n σ. Inr m = (state i) ⤜ (λs. create-insert s n σ i)) |
    (2) (∃ n.   Inr m = (state i) ⤜ (λs. create-delete s n))
    by (metis (full-types) send-correct assms(2))
  then show ?thesis
    proof (cases)
    case 1
    then obtain s n' σ where s-def:
      Inr s = state i Inr m = create-insert s n' σ i
      by (cases state i, simp, simp add:bind-def, blast)
    moreover have is-associated-string (set (received-messages i)) s
      using a assms(1) assms(3) apply (cases i, simp only:received-messages-eff)
      using s-def(1) state-is-associated-string'
      by (simp, metis (mono-tags, lifting) is-certified-associated-string.simps(1))
    ultimately show ?thesis using create-insert-consistent s-def assms
      by (metis Un-insert-right sup-bot.right-neutral)
  next
```

```
    case 2
    then obtain s n' where s-def:
      Inr s = state i Inr m = create-delete s n'
      by (cases state i, simp, simp add:bind-def, blast)
    moreover have is-associated-string (set (received-messages i)) s
      using a assms(1) assms(3) apply (cases i, simp only:received-messages-eff)
      using s-def(1) state-is-associated-string'
      by (simp, metis (mono-tags, lifting) is-certified-associated-string.simps(1))
    ultimately show ?thesis using create-delete-consistent s-def assms
      by (metis Un-insert-right sup-bot.right-neutral)
  qed
qed

lemma (in dist-execution) sent-messages-consistent:
  consistent {m. (∃ i. event-at i (Send m))}
proof −
  obtain ids where ids-def: set ids = {i. is-valid-event-id i} ∧
    sorted-wrt (to-ord (happened-immediately-before)) ids ∧ distinct ids
    using top-sort finite-valid-event-ids  by (metis acyclic-happened-before)
  have ⋀x y. happened-immediately-before^++ x y ⟹ x ∈ set ids ∧ y ∈ set ids
    using converse-tranclpE ids-def tranclp.cases by fastforce
  hence a:⋀x y. happened-immediately-before^++ x y ⟹
    (∃ i j. i < j ∧ j < length ids ∧ ids ! i = x ∧ ids ! j = y)
    by (metis top-sort-eff ids-def  distinct-Ex1 irrefl-p)
  define n where n = length ids
  have n ≤ length ids ⟹ consistent (make-set n (λk x. event-at (ids ! k) (Send
x)))
  proof (induction n)
    case 0
    then show ?case using empty-consistent by simp
  next
    case (Suc n)
    moreover obtain i j where ij-def:
      ids ! n = (i,j) n < length ids
      is-valid-event-id (i,j) is-valid-state-id (i,j)
    by (metis Suc.prems Suc-le-lessD ids-def is-valid-event-id.elims(2) is-valid-state-id.simps
        le-eq-less-or-eq mem-Collect-eq nth-mem)
   moreover have set (received-messages (i,j)) ⊆ make-set n (λk x. event-at (ids
! k) (Send x))
    using ij-def apply (simp add:received-messages-eff del:received-messages.simps,
rule-tac subsetI)
      using sent-before-recv a apply (simp add:make-set-def)
      by (metis (no-types, opaque-lifting) distinct-Ex1 ids-def in-set-conv-nth)
    moreover have (i,j) ∉ I ‘ insert-messages (make-set n (λk x. event-at (ids !
k) (Send x)))
     apply (simp add:insert-messages-def image-iff make-set-def del:event-at.simps)

       using ids-def le-eq-less-or-eq nth-eq-iff-index-eq send-insert-id
       by (metis dual-order.strict-trans1 ij-def(1) ij-def(2) less-not-refl)
```

ultimately show *?case* **using** *Suc*
apply (*cases events i ! j*)
using *new-messages-keep-consistency* [**where** $i = (i,j)$] **by** *simp+*
qed
moreover have *make-set n* ($\lambda k\ x$. *event-at* (*ids ! k*) (*Send x*)) = {x. ($\exists\, i$. *event-at i* (*Send x*))}
apply (*simp add*:*make-set-def n-def*, *rule set-eqI*, *subst surjective-pairing*, *simp only*:*event-pred.simps*)
using *ids-def* **apply** *simp*
by (*metis fst-conv in-set-conv-nth is-valid-event-id.simps mem-Collect-eq prod.exhaust-sel snd-conv*)
ultimately show *?thesis* **using** *ids-def n-def* **by** *simp*
qed

lemma (**in** *dist-execution*) *received-messages-were-sent*:
assumes *is-valid-state-id* (i,j)
shows *make-set j* ($\lambda k\ m$. ($\exists\, s$. *event-at* (i, k) (*Receive s m*))) $\subseteq$ {m. $\exists\, i$. *event-at i* (*Send m*)}
using *no-data-corruption* **by** (*simp add*:*make-set-def*, *rule-tac subsetI*, *fastforce*)

lemma (**in** *dist-execution*) *state-is-associated-string*:
assumes *is-valid-state-id i*
shows *is-certified-associated-string* (*set* (*received-messages i*)) (*state i*)
using *state-is-associated-string′ received-messages-eff*
sent-messages-consistent received-messages-were-sent assms **by** (*cases i*, *simp*)

end

6 Strong Eventual Consistency

theory *SEC*
imports *StrongConvergence*
begin

In the following theorem we establish that all reached states are successful. This implies with the unconditional termination property (Section 5.5) of it that the integration algorithm never fails.

theorem (**in** *dist-execution*) *no-failure*:
fixes *i*
assumes *is-valid-state-id i*
shows *isOK* (*state i*)
apply (*cases state i*)
by (*metis assms state-is-associated-string is-certified-associated-string.simps*(*2*), *simp*)

The following theorem establishes that any pair of peers having received the same set of updates, will be in the same state.

theorem (**in** *dist-execution*) *strong-convergence*:

assumes *is-valid-state-id i*
assumes *is-valid-state-id j*
assumes *set* (*received-messages i*) = *set* (*received-messages j*)
shows *state i* = *state j*
using *state-is-associated-string is-certified-associated-string-unique* **by** (*metis assms*)

As we noted in Section 4.7, we have not assumed eventual delivery, but a corollary of this theorem with the eventual delivery assumption implies eventual consistency. Since finally all peer would have received all messages, i.e., an equal set.

7 Code generation

export-code *integrate create-insert create-delete* **in** *Haskell*
module-name *WOOT* **file-prefix** *code*

8 Proof Outline

In this section we outline and motivate the approach we took to prove the strong eventual consistency of WOOT.

While introducing operation-based CRDTs Shapiro et al. also establish [24][Theorem 2.2]. If the following two conditions are met:

- Concurrent operations commute, i.e., if a pair of operations m_1, m_2 is concurrent with respect to the order induced by the happened-before relation, and they are both applicable to a state s, then the message m_1 (resp. m_2) is still applicable on the state reached by applying m_2 (resp. m_1) on s and the resulting states are equal.
- Assuming causal delivery, the messages are applicable.

Then the CRDT has strong convergence. The same authors extend the above result in [23, Proposition 2.2] to more general delivery orders $\xrightarrow{d}$ (weaker than the one induced by the happened-before relation), i.e., two messages may be causally dependent but concurrent with respect to $\xrightarrow{d}$. Assuming operations that are concurrent with respect to $\xrightarrow{d}$ commute, and messages are applicable, when the delivery order respects $\xrightarrow{d}$ then again the CRDT has strong convergence.

A key difficulty of the consistency proof of the WOOT framework is that the applicability condition for the WOOT framework has three constraints:

1. Dependencies must be met.
2. Identifiers must be distinct.

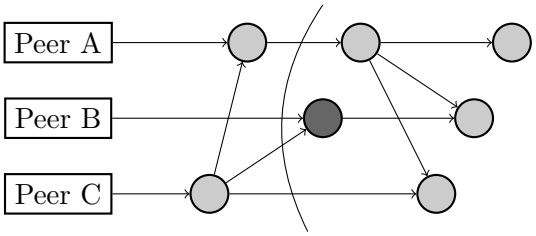


Figure 3: Example state graph, where the consistency is established left of the bend curve.

3. The order must be consistent, i.e. the predecessor W-character must appear before the successor W-character in the state an insert message is being integrated.

The first constraint is a direct consequence of the semantic causal delivery order. The uniqueness of identifiers can be directly established by analyzing the implementation of the message creation algorithms. Alternatively, Gomes et al. [7] use an axiomatic approach, where they require the underlying network protocol to deliver messages with unique identifiers. They provide a formal framework in Isabelle/HOL that can be used to show consistency of arbitrary CRDTs. Their results could be used to establish constraints 1 and 2.

The last constraint is the most intricate one, and forces us to use a different method to establish the strong eventual consistency. The fact that the order constraint is fulfilled is a consequence of the consistency property. But the current fundamental lemmas require applicability of the operations in the first place to establish consistency, which would result in a circular argument.

Zeller et. al. actually predict the above circumstance in the context of state-based CRDTs [27]:

> In theory it could even be the case that there are two reachable states for which the merge operation does not yield the correct result, but where the two states can never be reached in the same execution.

Because of the above, we treat WOOT as a distributed message passing algorithm and show convergence by establishing a global invariant, which is maintained during the execution of the framework. The invariant captures that the W-characters appear in the same order on all peers. It has strong convergence as a consequence, in the special case, when peers have received the same set of updates. It also implies that the generated messages will be applicable.

In Figure 3, we exemplify an induction step in a proof over the execution of the framework. The invariant is established for all states left of the dashed

lines, and we show that it remains true if we include the state, drawn in dark gray. Note that induction proceeds in an order consistent with the happened-before relation.

The technique we are using is to define a relation *is-associated-string* from a set of messages to the final state their application leads to. Crucially, that relation can be defined in a message-order independent way. We show that it correctly models the behaviour of Algorithm *integrate* by establishing that applying the integration algorithm to the associated string of a set M leads to the associated string of the set $M \cup \{m\}$ in Proposition *integrate-insert-commute*.

We also show that at most one s fulfills *is-associated-string* M s, which automatically implies commutativity (cf. Lemma *associated-string-unique*).

Note that the domain of the relation *is-associated-string* consists of the sets of messages that we call *consistent*. We show that, in every state of a peer, the set of received messages will be consistent. The main ingredient required for the definition of a consistent set of messages as the relation *is-associated-string* are *sort keys* associated to the W-characters, which we will explain in the following Section.

8.1 Sort Keys

There is an implicit sort key, which is deterministically computable, using the immutable data associated to a W-character and the data of the W-characters it (transitively) depends on.

We show that Algorithm *integrate* effectively maintains the W-characters ordered with respect to that sort key, which is the reason we can construct the mapping *is-associated-string* in a message-order independent way. An alternative viewpoint would be to see Algorithm *integrate-insert* as an optimized version of a more mundane algorithm, that just inserts the W-characters using this implicit sort key.

Since the sort key is deterministically computable using the immutable data associated to a W-character and the data of the W-characters it (transitively) depends on, all peers could perform this computation independently, which leads to the conclusion that the W-characters will be ordered consistently across all peers.

The construction relies on a combinator Ψ that computes the sort key for a W-character, and which requires as input:

- The unique identifier associated to a W-character.
- The sort keys of the predecessor/successor W-characters.

Its values are elements of a totally ordered space.

Note that the predecessor (resp. successor) W-character of a W-character is the W-character that was immediately before (resp. after) it at the time it was inserted. Like its unique identifier, it is immutable data associated with that W-character. Sometimes a W-character is inserted at the beginning (resp. end) of the string. For those W-characters, we use the special smallest (resp. largest) sort keys, denoted by $\vdash$ (resp. $\dashv$) as predecessor (resp. successor). These keys themselves are never associated to a W-character.

We will write $\Psi\ (l,\ u)\ i$ for the value computed by the combinator for a W-character with identifier i, assuming the sort key of its predecessor (resp. successor) is l (resp. u).

For example, the sort key for a W-character with identifier i inserted in an empty string (hence its predecessor is $\vdash$ and its successor is $\dashv$) will be $\Psi\ (\vdash,\ \dashv)\ i$. A W-character inserted between that character and the end of the string, with identifier j, would be assigned the sort key $\Psi\ (\llbracket\Psi\ (\vdash,\ \dashv)\ i\rrbracket,\ \dashv)\ j$.

The sort key needs to fulfill a couple of properties, to be useful:

There should never be a pair of W-characters with the same sort key. Note, if this happens, even if those W-characters were equal or ordered consistently, we would not be able to insert a new W-character between those W-characters.

Since the W-characters have themselves unique identifiers, a method to insure the above property is to require that Ψ be injective with respect to the identifier of the W-character it computes a sort key for, i.e., $\Psi\ (l,\ u)\ i = \Psi\ (l',\ u')\ i' \Longrightarrow i = i'$.

Another essential property is that the W-characters with predecessor having the sort key l and successor having the sort key u should have a sort key that is between l and u, such that the W-character is inserted between the preceding and succeeding W-character, i.e., $l < \Psi\ (l,u)\ i < u$.

This latter property ensures intention preservation, i.e. the inserted W-character will be placed at the place the user intended.

If we review function *concurrent*, then we see that the algorithm compares W-characters by identifier, in the special case, when the inserted W-character is compared to a W-character whose predecessor and successor are outside of the range it is to be inserted in. A careful investigation, leads to the conclusion that:

If $l \leq l' < \Psi\ (l,u)\ i < u' \leq u$ then $\Psi(l,u)\ i$ can be compared with $\Psi(l',u')\ i'$ by comparing i with i', i.e.:

- $i < i' \Longrightarrow \Psi\ (l,u)\ i < \Psi(l',u')\ i'$

In Section 5.1 we show that a combinator Ψ with the above properties can be constructed (cf. Propositions *psi-narrow psi-mono psi-elem*). Using the sort keys we can define the notion of a consistent set of messages as well

as the relation *is-associated-string* in a message-order independent way.

8.2 Induction

We have a couple of criteria that define a consistent set of messages:

- Each insert message in the set has a unique identifier.
- If a message depends on another message identifier, a message with that identifier will be present. Note that for insert messages, these are the predecessor/successor W-characters if present. For delete messages it is the corresponding insert message.
- The dependencies form a well-order, i.e., there is no dependency cycle.
- It is possible to assign sort keys to each insert message, such that the assigned sort key for each insert message is equal to the value returned by the Ψ for it, using the associated sort keys of its predecessor and successors, i.e., $a\ (P\ m) < a\ (S\ m) \wedge a\ [\![I\ m]\!] = [\![\Psi\ (a\ (P\ m),\ a\ (S\ m))\ (I\ m)]\!]$. Note that we also require that sort key of the predecessor is smaller than the sort key of the successor.

The relation *is-associated-string* is then defined by ordering the insert messages according to the assigned sort keys above and marking W-characters, for which there are delete messages as deleted.

The induction proof (Lemma *dist-execution.sent-messages-consistent*) over the states of the framework is straight forward: Using Lemma *top-sort* we find a possible order of the states consistent with the happened before relation. The induction invariant is that the set of generated messages by all peers is consistent (independent of whether they have been received by all peers (yet)). The latter also implies that the subset a peer has received in any of those states is consistent, using the additional fact that each messages dependencies will be delivered before the message itself (see also Lemma *consistent-subset* and Proposition *integrate-insert-commute'*). For the induction step, we rely on the results from Section 5.4 that any additional created messages will keep the set of messages consistent and that the peers' states will be consistent with the (consistent subset of) messages they received (Lemma *dist-execution.state-is-associated-string'*).

end

9 Example

theory *Example*
imports *SEC*

begin

In this section we formalize the example from Figure 2 for a possible run of the WOOT framework with three peers, each performing an edit operation. We verify that it fulfills the conditions of the locale *dist-execution* and apply the theorems.

```
datatype example-peers
  = PeerA
  | PeerB
  | PeerC
derive linorder example-peers

fun example-events :: example-peers ⇒ (example-peers, char) event list where
  example-events PeerA = [
      Send (Insert (InsertMessage ⊢ (PeerA, 0) ⊣ CHR ''B'')),
      Receive (PeerA, 0) (Insert (InsertMessage ⊢ (PeerA, 0) ⊣ CHR ''B'')),
      Receive (PeerB, 0) (Insert (InsertMessage ⊢ (PeerB, 0) ⊣ CHR ''A'')),
      Receive (PeerC, 1) (Insert (InsertMessage ⟦(PeerA, 0)⟧ (PeerC, 1) ⊣ CHR
''R''))
  ] |
  example-events PeerB = [
      Send (Insert (InsertMessage ⊢ (PeerB, 0) ⊣ CHR ''A'')),
      Receive (PeerB, 0) (Insert (InsertMessage ⊢ (PeerB, 0) ⊣ CHR ''A'')),
      Receive (PeerA, 0) (Insert (InsertMessage ⊢ (PeerA, 0) ⊣ CHR ''B'')),
      Receive (PeerC, 1) (Insert (InsertMessage ⟦(PeerA, 0)⟧ (PeerC, 1) ⊣ CHR
''R''))
  ] |
  example-events PeerC = [
      Receive (PeerA, 0) (Insert (InsertMessage ⊢ (PeerA, 0) ⊣ CHR ''B'')),
      Send (Insert (InsertMessage ⟦(PeerA, 0)⟧ (PeerC, 1) ⊣ CHR ''R'')),
      Receive (PeerC, 1) (Insert (InsertMessage ⟦(PeerA, 0)⟧ (PeerC, 1) ⊣ CHR
''R'')),
      Receive (PeerB, 0) (Insert (InsertMessage ⊢ (PeerB, 0) ⊣ CHR ''A''))
  ]
```

The function *example-events* returns the sequence of events that each peer evaluates. We instantiate the preliminary context by showing that the set of peers is finite.

```
interpretation example: dist-execution-preliminary example-events
proof
  have a:UNIV = {PeerA, PeerB, PeerC}
    using example-events.cases by auto
  show finite (UNIV :: example-peers set) by (simp add:a)
qed
```

To prove that the *happened-before* relation is acyclic, we provide an order on the state that is consistent with it, i.e.:

- The assigned indicies for successive states of the same peer are increas-

ing.

- The assigned index of a state receiving a message is larger than the assigned index of the messages source state.

```
fun witness-acyclic-events :: example-peers event-id ⇒ nat
  where
    witness-acyclic-events (PeerA, 0) = 0 |
    witness-acyclic-events (PeerB, 0) = 1 |
    witness-acyclic-events (PeerA, (Suc 0)) = 2 |
    witness-acyclic-events (PeerB, (Suc 0)) = 3 |
    witness-acyclic-events (PeerC, 0) = 4 |
    witness-acyclic-events (PeerC, (Suc 0)) = 5 |
    witness-acyclic-events (PeerC, (Suc (Suc 0))) = 6 |
    witness-acyclic-events (PeerC, (Suc (Suc (Suc 0)))) = 7 |
    witness-acyclic-events (PeerA, (Suc (Suc 0))) = 8 |
    witness-acyclic-events (PeerA, (Suc (Suc (Suc 0)))) = 9 |
    witness-acyclic-events (PeerB, (Suc (Suc 0))) = 8 |
    witness-acyclic-events (PeerB, (Suc (Suc (Suc 0)))) = 9 |
    witness-acyclic-events (PeerA, (Suc (Suc (Suc (Suc n))))) = undefined |
    witness-acyclic-events (PeerB, (Suc (Suc (Suc (Suc n))))) = undefined |
    witness-acyclic-events (PeerC, (Suc (Suc (Suc (Suc n))))) = undefined
```

To prove that the created messages make sense, we provide the edit operation that results with it. The first function is the inserted letter and the second function is the position the letter was inserted.

```
fun witness-create-letter :: example-peers event-id ⇒ char
  where
    witness-create-letter (PeerA, 0) = CHR ''B'' |
    witness-create-letter (PeerB, 0) = CHR ''A'' |
    witness-create-letter (PeerC, Suc 0) = CHR ''R''

fun witness-create-position :: example-peers event-id ⇒ nat
  where
    witness-create-position (PeerA, 0) = 0 |
    witness-create-position (PeerB, 0) = 0 |
    witness-create-position (PeerC, Suc 0) = 1
```

To prove that dependencies of a message are received before a message, we provide the event id as well as the message, when the peer received a messages dependency.

```
fun witness-deps-received-at :: example-peers event-id ⇒ example-peers event-id ⇒
nat
  where
    witness-deps-received-at (PeerA, Suc (Suc (Suc 0))) (PeerA, 0) = 1 |
    witness-deps-received-at (PeerB, Suc (Suc (Suc 0))) (PeerA, 0) = 2 |
    witness-deps-received-at (PeerC, Suc (Suc 0)) (PeerA, 0) = 0
```

fun *witness-deps-received-is* :: *example-peers event-id* ⇒ *example-peers event-id* ⇒ (*example-peers event-id*, *char*) *insert-message*
where
witness-deps-received-is (*PeerA*, *Suc* (*Suc* (*Suc 0*))) (*PeerA*, *0*) = (*InsertMessage* ⊢ (*PeerA*, *0*) ⊣ *CHR ''B''*) |
witness-deps-received-is (*PeerB*, *Suc* (*Suc* (*Suc 0*))) (*PeerA*, *0*) = (*InsertMessage* ⊢ (*PeerA*, *0*) ⊣ *CHR ''B''*) |
witness-deps-received-is (*PeerC*, *Suc* (*Suc 0*)) (*PeerA*, *0*) = (*InsertMessage* ⊢ (*PeerA*, *0*) ⊣ *CHR ''B''*)

lemma *well-order-consistent*:
fixes *i j*
assumes *example.happened-immediately-before i j*
shows *witness-acyclic-events i* < *witness-acyclic-events j*
using *assms*
apply (*rule-tac* [!] *witness-acyclic-events.cases* [**where** *x*=*i*])
apply (*rule-tac* [!] *witness-acyclic-events.cases* [**where** *x*=*j*])
by *simp*+

Finally we show that the *example-events* meet the assumptions for the distributed execution context.

interpretation *example*: *dist-execution example-events*
proof
fix *i s m*
show
dist-execution-preliminary.event-at example-events i (*Receive s m*) ⟹
dist-execution-preliminary.event-at example-events s (*Send m*)
apply (*rule-tac* [!] *witness-acyclic-events.cases* [**where** *x*=*i*])
by *simp*+
next
fix *i j s* :: *example-peers event-id*
fix *m*
show *example.event-at i* (*Receive s m*) ⟹
example.event-at j (*Receive s m*) ⟹ *fst i* = *fst j* ⟹ *i* = *j*
apply (*rule-tac* [!] *witness-acyclic-events.cases* [**where** *x*=*i*])
apply (*rule-tac* [!] *witness-acyclic-events.cases* [**where** *x*=*j*])
by *simp*+
next
have *wf* (*inv-image* {(*x*,*y*). *x* < *y*} *witness-acyclic-events*)
by (*simp add*: *wf-less*)
moreover have {(*x*, *y*). *example.happened-immediately-before x y*} ≤
inv-image {(*x*,*y*). *x* < *y*} *witness-acyclic-events*
using *well-order-consistent* **by** *auto*
ultimately have *wfP example.happened-immediately-before*
using *well-order-consistent wfP-def wf-subset* **by** *blast*
thus *acyclicP example.happened-immediately-before*
using *wfP-acyclicP* **by** *blast*
next
fix *m s i j i'*

```
  have example.event-at (i, j) (Receive s m) ⟹
      i' ∈ deps m ⟹
    example.event-at (i, witness-deps-received-at (i, j) i') (Receive (I (witness-deps-received-is
(i, j) i')) (Insert (witness-deps-received-is (i, j) i'))) ∧ witness-deps-received-at (i,
j) i' < j ∧ I (witness-deps-received-is (i, j) i') = i'
    apply (rule-tac [!] witness-acyclic-events.cases [where x=(i,j)])
    by simp+
  thus example.event-at (i, j) (Receive s m) ⟹
      i' ∈ deps m ⟹
      ∃ s' j' m'.
        example.event-at (i, j') (Receive s' (Insert m')) ∧ j' < j ∧ I m' = i'
    by blast
next
  fix m i
  have example.event-at i (Send m) ⟹
       Inr m = example.state i ⤜ (λs. create-insert s (witness-create-position i)
(witness-create-letter i) i)
    apply (rule-tac [!] witness-acyclic-events.cases [where x=i])
    by (simp add:ext-ids-def)+
  thus example.event-at i (Send m) ⟹
         (∃ n σ. return m = example.state i ⤜ (λs. create-insert s n σ i)) ∨
         (∃ n. return m = example.state i ⤜ (λs. create-delete s n))
    by blast
qed
```

As expected all peers reach the same final state.

```
lemma
  example.state (PeerA, 4) = Inr [
    InsertMessage ⊢ (PeerA, 0) ⊣ (Some CHR ''B''),
    InsertMessage ⊢ (PeerB, 0) ⊣ (Some CHR ''A''),
    InsertMessage ⟦(PeerA, 0)⟧ (PeerC, 1) ⊣ (Some CHR ''R'')]
  example.state (PeerA, 4) = example.state (PeerB, 4)
  example.state (PeerB, 4) = example.state (PeerC, 4)
  by (simp del:substr-simp add:ext-ids-def substr.simps less-example-peers-def)+
```

We can also derive the equivalence of states using the strong convergence theorem. For example the set of received messages in the third state of peer A and B is equivalent, even though they were not received in the same order:

```
lemma
  example.state (PeerA, 3) = example.state (PeerB, 3)
proof −
  have example.is-valid-state-id (PeerA, 3) by auto
  moreover have example.is-valid-state-id (PeerB, 3) by auto
  moreover have
    set (example.received-messages (PeerA, 3)) =
     set (example.received-messages (PeerB, 3))
    by auto
  ultimately show ?thesis
    by (rule example.strong-convergence)
```

qed

Similarly we can conclude that reached states are successful.

lemma
isOK (*example.state* (*PeerC*, *4*))
proof −
have *example.is-valid-state-id* (*PeerC*, *4*) **by** *auto*
thus *?thesis* **by** (*rule example.no-failure*)
qed

end

References

[1] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating CRDTs for real-time document editing. In *Symposium on Document Engineering (DocEng)*, pages 103–112. ACM, 2011.

[2] L. Briot, P. Urso, and M. Shapiro. High responsiveness for group editing crdts. In *International Conference on Supporting Group Work (GROUP)*, pages 51–60. ACM, 2016.

[3] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[4] R. Dallaway. WOOT model for Scala and JavaScript via Scala.js. https://github.com/d6y/wootjs, 2016. Accessed: 2017-01-25.

[5] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Record*, volume 18, pages 399–407. ACM, 1989.

[6] V. Emanouilov. Collaborative rich text editor. https://github.com/kroky/woot, 2016. Accessed: 2017-01-25.

[7] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA), 2017.

[8] R. Kaplan. A real time collaboration toy project based on WOOT. https://github.com/ryankaplan/woot-collaborative-editor, 2016. Accessed: 2017-01-25.

[9] G. Klein, T. Nipkow, D. von Oheimb, C. Pusch, and M. Strecker. Java source and bytecode formalizations in isabelle: μjava.

[10] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[11] S. Kumawat and A. Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications*, 3(12):30–38, 2010.

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

[13] M. Letia, N. Preguiça, and M. Shapiro. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS Operating Systems Review*, 44(2):29–34, 2010.

[14] D. Li and R. Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work (CSCW)*, 19(1):1–43, 2010.

[15] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Symposium on Document Engineering (DocEng)*, pages 37–46. ACM, 2013.

[16] T. Olson. Real time group editor without operational transformation. https://github.com/TGOlson/woot-haskell, 2016. Accessed: 2017-01-25.

[17] G. Oster, P. Molli, P. Urso, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 1–10. IEEE, 2006.

[18] G. Oster, P. Urso, P. Molli, and A. Imine. Real time group editors without operational transformation. Technical Report RR-5580, INRIA, 2005.

[19] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *Conference on Computer Supported Cooperative Work (CSCW)*, pages 259–268. ACM, 2006.

[20] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 395–403. IEEE, 2009.

[21] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[22] H.-G. Roh, J.-S. Kim, J. Lee, and S. Maeng. Optimistic operations for replicated abstract data types. Technical report, 2009.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400. Springer-Verlag, 2011.

[25] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P wiki-based collaborative writing tool. In *International Conference on Web Information Systems Engineering*, pages 503–512. Springer, 2007.

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

[27] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. Formal specification and verification of crdts. In E. Ábrahám and C. Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.