A CRDT Supporting Selective Undo for Collaborative Text Editing

Weihai Yu, Luc André, Claudia-Lavinia Ignat

To cite this version:

Weihai Yu, Luc André, Claudia-Lavinia Ignat. A CRDT Supporting Selective Undo for Collaborative Text Editing. 15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2015, Grenoble, France. pp.193-206, 10.1007/978-3-319-19129-4_16 . hal-01246212

HAL Id: hal-01246212

https://hal.inria.fr/hal-01246212

Submitted on 18 Dec 2015

A CRDT Supporting Selective Undo for Collaborative Text Editing

Weihai Yu[1], Luc André[234], and Claudia-Lavinia Ignat[234]

[1] Department of Computer Science, UiT - The Arctic University of Norway
[2] Inria, Villers-lès-Nancy, F-54600, France
[3] Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France
[4] CNRS, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France

Abstract. Undo is an important feature of editors. However, even after over two decades of active research and development, support of undo for real-time collaborative editing is still very limited. We examine issues concerning undo in collaborative text editing and present an approach using a layered commutative replicated data type (CRDT). Our performance study shows that it provides sufficient responsiveness to the end users.

1 Introduction

Undo is a key feature of editors. In a single-user editor, a user can conveniently undo earlier editing operations in reverse chronological order. In a collaborative editor, however, users at different sites may generate operations concurrently. This means that a user cannot easily perceive a linear operation order. Some systems restrict what can be undone. For example, with Google Drive (`https://drive.google.com`), a user can only undo locally generated operations. User studies show that users indeed expect to be able to undo other users' operations when working on common tasks [1]. In the research community of collaborative editing, *selective undo* is widely regarded as an important feature [2–9]. With selective undo, a user can undo an earlier operation, regardless of when and where the operation was generated.

Current systems that support selective undo are subject to two main limitations. Firstly, they only support undo of operations on atomic objects (e.g. characters or unbreakable lines). In the case of string-wise operations such as copy-paste, find-replace or select-delete, users can typically only undo earlier operations character by character. Secondly, selective undo may lead to undesirable effects. For example, a user first inserts a misspelled word and then makes a correction. The correction depends on the first insertion of the word. It is undesirable to undo the insertion alone and leave the correction behind as a groundless modification.

In this paper we propose a novel approach to collaborative text editing that supports selective undo of string-wise operations. This is the first work that manages undesirable effects of undo.

2 Related Work

There are two general approaches to collaborative editing, based either on operation transformation (OT) [6, 7, 10] or on commutative replication data types (CRDT) [8, 9, 11–14]. With OT, a remote operation is transformed and integrated in the local site. The time complexity depends on the lengths of operation histories (linear at best). Furthermore, it is hard to design correct operation transformation functions [15]. One common way to relax certain required conditions for transformation functions is to restrict the order in which operations are transformed at all sites. Therefore OT approaches generally do not scale well and practically require the involvement of central servers. With CRDT, concurrent insertions are ordered based on the underlying data structure, so the time complexity may not depend on the lengths of operation histories. [16] reported that CRDT algorithms are better suited for large-scale distributed environments and outperform OT algorithms by orders of magnitudes.

Supporting string operations and selective undo requires obtaining at runtime relations among operations, such as whether a string is part of a larger insertion or whether an operation is an undo of another operation. Since strings might be split by subsequent operations and operations are executed concurrently, obtaining such relations can be complicated. Deriving such relations through operation transformation is particularly difficult. Currently, most related work can only apply undo to insertion and deletion of atomic objects [2–8]. To the best of our knowledge, only our previous work [9] supports selective undo of string operations. However, [9] does not account for possible undesirable effects of undo.

In this paper, we propose a novel CRDT that captures useful relations among operations. Our approach offers support for string-based undo and deals with undesirable effects of selective undo. Our current work is built on our previous work. The general view-model system structure is similar to the one described in [9]. The underlying scheme for character identifiers is similar to the one described in [11].

3 Undo Effects

Allowing undo of any operation without restriction might lead to undesirable effects.

Example 1 *The state after two insertions ins_1 (with string "`this is hard`") and ins_2 ("`not `") is "`this is not hard`". Undoing ins_1 results in state "`not `". If the text "`is hard`" is a single unit and the string "`not `" is part of it, then, without the text "`is hard`", the string "`not `" becomes groundless.*

When a user inserts a string *str* into an existing *unit string* str_0, str_0 is the *ground* of *str*. If str_0 had not existed, the user would not have inserted *str* and the existence of *str* is *groundless*.

The definition of unit strings depends on the types of documents. Without loss of generality, we define a unit string as being generated by a single operation, such as an insertion or the undo of a deletion. More specifically, if op_0 generates string str_0 and *ins* inserts string *str* into str_0, op_0 is the *ground operation* of *op* (or *op* is *built on* op_0)

and str_0 is the *ground string* of *str*. Furthermore, the built-on relation is transitive. That is, if op_2 is built on op_1 and op_1 is built on op_0, then op_2 is also built on op_0.

The effect of undoing an operation *op* should be as if *op* and all operations built on *op* had never occurred. More specifically, suppose $H^s = H_0 \cdot op \cdot H_1 \cdot undo(op) \cdot H_2$ is the history of operations at site *s*, where H_0 represents the sequence of operations executed before *op*, H_1 the sequence of operations after *op* and H_2 the sequence of operations after the undo of *op*. If we denote by $H^{\widetilde{op}}$ the sequence of operations as the result of removal from *H* of all operations built on *op*, then H^s and $H_0 \cdot H_1^{\widetilde{op}} \cdot H_2^{\widetilde{op}}$ should produce the same strings. Notice that *op* and $undo(op)$ may be generated from different sites. Also, although the operations in H_2 occur after the undo of *op* at site *s*, H_2 may still contain operations built on *op*, due to concurrent operations.

Our definition of ground operations might be too general to the user. In practice, the user may not agree that string *str* is built on str_0 (or *str* is useful outside the context of str_0). In such situations, the user should be able to decide which operations are not built on the operation being undone (or to manually select which groundless strings, detected by the editor, should remain after the undo). Thus, when a user tries to perform an undo that results in groundless strings, the editor should warn the user, so that the user is able to determine the final effect of the undo, or to simply give up the undo.

However, due to concurrent operations, a collaborative editor is not always able to warn the user of possible groundless strings in time. In Example 1, when a user at a remote site undoes ins_1 before ins_2 arrives, the undo does not cause any groundless string. In such cases, all sites should unanimously (without user intervention) eliminate the groundless string "`not `" when they receive both $undo(ins_1)$ and ins_2.

Example 2 *In Example 1, another site first executes* ins_1 *and then executes* del_1 *("* `is hard`*") concurrently with* ins_2*. The string "*`not `*" inserted by* ins_2 *becomes groundless after a site executes both* ins_2 *and* del_1*.*

A concurrent deletion may also cause groundless strings. Notice that a deletion never causes groundless strings locally. Hence the sites should always unanimously eliminate groundless strings caused by remote deletions.

Our work ensures that there is no groundless effect of local undo (unless the user explicitly wants the effect) and there is no groundless effect of remote undo or deletion. Furthermore, it ensures the traditional correctness criteria convergence and intention preservation [6] as discussed in Section 6.

4 View and Model

With a collaborative editor, a document is concurrently updated from a number of peers at different sites. Every peer consists of a view of the document, a model, a log of operation history and several queues.

A peer concurrently receives local operations generated by the user and remote operations sent from other peers. Local operations take immediate effect in the view. The peer stores executed local operations and received remote operations in queues. During a synchronization cycle, it integrates the stored operations in the model and shows the effects of integrated remote operations in the view. The peer also records integrated operations in the log. Later, it broadcasts integrated local operations to other peers. At any time, the user may undo an operation selected from the log.

Every peer has a unique peer identifier *pid*. An operation originated at a peer has a peer update number *pun* that is incremented with every integrated local operation. Therefore, we can uniquely identify an operation with the pair (pid, pun). In what follows, we use op^{pid}_{pun} to denote an operation *op* identified with (pid, pun).

A view is mainly a string of characters. A user at a peer can insert or delete a substring at a position in the view, and undo an earlier integrated local or remote operation selected from the log.

A model materializes editing operations and relations among them. It consists of layers of linked nodes that encapsulate characters. Conceptually, characters have unique *identifiers* that are totally ordered (though not every identifier is explicitly represented in the model). For two characters c_l and c_r, if $c_l.id < c_r.id$, then c_l appears to the left of c_r. A character identifier is represented as a sequence of integers. For $c_l.id = p_0 \ldots p_{k-1} p^l_k \ldots$, $c_r.id = p_0 \ldots p_{k-1} p^r_k \ldots$ and $p^l_k < p^r_k$, the two identifiers start to differ at the $(k+1)$-th integer. Suppose we insert a string of characters $c_0 \ldots c_n$ between c_l and c_r. The identifier of character c_i $(0 \leq i \leq n)$ in the string is $p_0 \ldots p_{k-1} p_k p_{k+1} (p_{k+2} + i)$, where $p^l_k < p_k < p^r_k$ and p_{k+1} is a function of *pid*. If another peer inserts a string $c'_0 \ldots c'_m$ at the same place and generates p'_k, p'_{k+1} and p'_{k+2}, the two strings are ordered according to p_k and p'_k. If $p_k = p'_k$, the two strings are then ordered according to p_{k+1} and p'_{k+1} (i.e. according to *pid*s). We refer the interested readers to [11] for a more complete description of the generation of character identifiers.

Nodes at the lowest layer of a model represent insertions and contain inserted characters. Nodes at higher layers represent deletions. That is, a higher-layer node (outer node) deletes the characters in the lower-layer nodes (inner nodes) it contains.

A node contains the identifier cid_l of its leftmost character and cid_r of its rightmost character. The identifiers of the other characters (i.e. not at the edges of the node) are not explicitly represented in the model. An insertion node also contains a string *str* of characters.

Subsequent operations may split existing nodes. Nodes of the same operation share an *op* element as the operation's descriptor. The descriptor contains the identifier and type of the operation, a set $\mathscr{P}$ (for parents) of references to the descriptors of *op*'s ground operations and a set $\mathscr{C}$ (for children) of operations built on *op*. The descriptor also has an *undo* element that contains a set $\mathscr{U}$ of identifiers of its *undo* operations (there might be more than one, as multiple peers might concurrently undo the same operation). An *undo* element may itself have its own *undo* element (e.g. when the original operation is redone). Thus the *undo* elements of an operation form a chain. The operation is *effectively undone* if the length of the chain is an odd number.

An insertion is *self-visible* if it is not effectively undone. A deletion is *self-visible* if it is effectively undone. An operation is *visible* if it is self-visible and all its ground operations are visible. A character is *visible* if all operations on it are visible.

There are three types of links among nodes: *l-r* links maintain the left-right character order; op_l-op_r links connect nodes of the same operations; *i-o* links maintain the inner-outer relations. The outermost nodes and the nodes inside the same outer node are linked

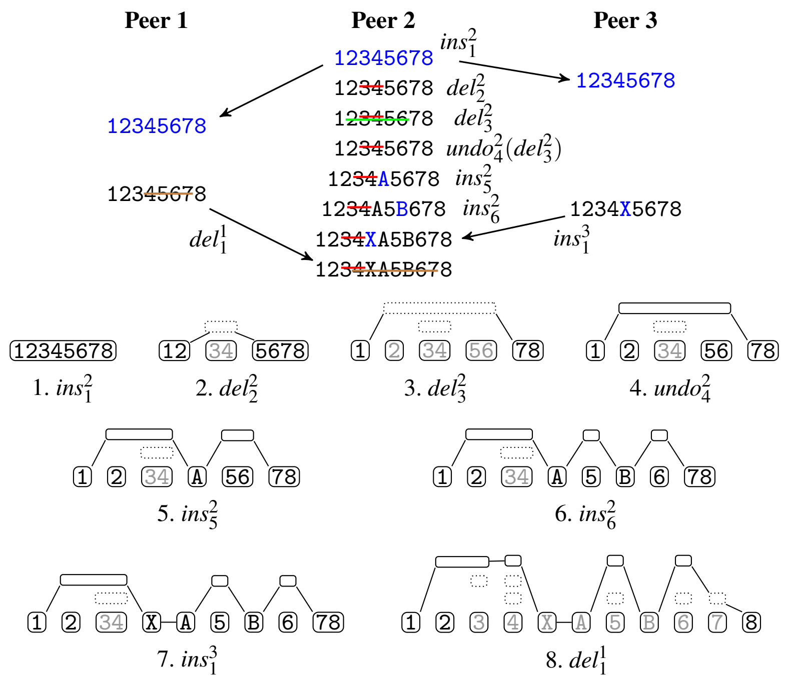


Fig. 1. Examples of model updates

with *l*-*r* links. When the view and the model are synchronized, the view equals to the concatenation of all visible characters of the outermost nodes through the *l*-*r* links.

Figure 1 shows an example with three peers. The upper part shows a number of operations generated at the peers. The lower part shows the model snapshots at Peer 2. Nodes of the same deletion are aligned horizontally. Nodes with dotted border are self-invisible. Characters in light gray are invisible. We describe how to update the model in the following section.

5 Operations and Undo

A user may execute the following normal view operations: (i) $ins(pos, str)$ inserts string *str* at position *pos*. (ii) $del(pos, len)$ deletes *len* characters right to position *pos*. In addition to the normal operations, the user can undo any operation selected from the log.

A peer stores executed normal view operations in a queue. It may aggregate consecutive operations, for instance, to form string operations from character operations. During a synchronization cycle, the peer turns view operations into model operations before integration.

To avoid traversing a large number of nodes for every local operation, a model maintains a *current position*, (v_{curr}, p_{curr}), where v_{curr} is the current node and p_{curr} is the offset to the left edge of v_{curr}. Because a user typically focuses on a small region at a time, the distances between consecutive operations are often short.

There are three normal model operations: (i) $move(m)$ moves the current position a distance of m visible characters (leftwards when negative). (ii) $ins(str)$ inserts string str at the current position. (iii) $del(len)$ deletes len characters right to the current position.

A peer processes a local undo operation in the order opposite to normal operations. It first integrates the undo in the model and then synchronizes it to the view.

For each integrated local operation, a peer broadcasts a representation of the *model update* to remote peers. A node is uniquely identified by $(op.pid, op.pun, cid_l)$, where $(op.pid, op.pun)$ is the identifier of its operation and cid_l is the identifier of its leftmost character. The peer uses the identifiers of the involved nodes, offsets to the leftmost characters etc. to describe the update, so that remote peers can unambiguously locate the referent nodes and split boundaries. Each peer maintains a hash table of nodes using their identifiers, so locating a referent node takes near-constant time.

A model-view synchronization does the following tasks sequentially: (1) integrating local operations, (2) integrating remote operations, and (3) updating the view (with a *render* procedure). This ensures that, when a model integrates a local operation, there is no concurrent remote operation in the model.

Procedure *localIns*(*pid*, *pun*, *str*)

```
1 $(v_l, v_r) \leftarrow split(nextVisible(v_{curr}, p_{curr}, 0))$
2 $v_{ins} \leftarrow Node(cidsBetween(v_l.cid_r, v_r.cid_l, pid, str.len), Op(pid, pun), str)$
3 $setInsGroundOps(v_{ins}.op, v_l, v_r)$
4 $insertBetween(v_{ins}, v_l, v_r)$
5 $v_{curr}, p_{curr} \leftarrow v_r, 0$
```

Procedure *localIns* integrates a local insertion. It places the new inserted string to the right of all invisible characters at the current position. Procedure $nextVisible(v, p, n)$, called from *localIns* (line 1) and *localDel* (lines 1 and 2), returns the position of the n-th visible character right to position (v, p). In Fig. 1-5, Peer 2 inserts "`A`" of ins^2_5 to the right of the invisible "`34`".

If the insertion position is inside an existing node, *localIns* splits the node (line 1). Procedure *split* returns either the new nodes after the split, or two existing nodes if the split position is at the edge of an existing node. It also splits the corresponding inner nodes, recursively down to an insertion node at the lowest layer. This way, it exposes the character identifiers at the position of the split. In Fig. 1-6, when inserting "`B`", Peer 2 splits the "`56`" nodes of both del^2_3 and ins^2_1.

Next, *localIns* creates a new insertion node (line 2). Procedure *cidsBetween* generates the character identifiers using the ones at the insertion position.

Procedure *setInsGroundOps* updates the $\mathscr{P}$ and $\mathscr{C}$ sets of the insertion and its ground operations (line 3). If v_l and v_r are of the same operation, then this operation is a ground operation of the new insertion. The procedure goes on with v_l's rightmost inner node and v_r's leftmost inner node, downward until the lowest layer. In Fig. 1-6, both del^2_3 and ins^2_1 are ground operations of ins^2_6.

Finally, *localIns* connects the new insertion node with the neighboring nodes (line 4) and moves the current position to the right end of the inserted string (line 5).

Procedure *localDel*(*pid*, *pun*, *len*)

```
1 $(v_l, v_r) \leftarrow split(nextVisible(v_{curr}, p_{curr}, 0))$
2 $(v'_l, v'_r) \leftarrow split(nextVisible(v_r, 0, len))$
3 $v_{del} \leftarrow Node(v_r.cid_l, v'_l.cid_r, Op(pid, pun))$
4 $insertInners(v_{del}, [v_r..v'_l])$
5 $insertBetween(v_{del}, v_l, v'_r)$
```

Procedure *localDel* splits existing nodes at the deletion boundaries (lines 1 and 2), inserts a new node for the deletion at the outermost layer (lines 3 and 5) and associates to it the corresponding inner nodes (line 4). Notice that a deletion may contain invisible characters inside the deleted string. For example, del^2_3 in Fig. 1-3 contains "`34`".

A model integrates a remote update only when the update is *ready for integration*, i.e., when all nodes and elements which the update refers to exist in the model (possibly after some split). For example, ins^3_1 in Fig. 1 is ready for integration in models in which a node of ins^2_1 exists. The ready-for-integration condition is less strict than the general "happen-before" condition in the literature (such as [6]), because only the nodes and elements which the update *directly* refers to must exist in the model.

Procedure *remoteIns*(*pid*, *pun*, *cid*, *str*, $\mathscr{G}$, v, *p*)

```
1 $v_{ins} \leftarrow Node(cid, stringRightEndCid(cid, str.len), Op(pid, pun), str)$
2 $setGroundOps(v_{ins}.op, \mathscr{G})$
3 $(v_l, v_r) \leftarrow insNarrow(cid, split(v, p))$
4 $extendInsGroundOps(v_{ins}.op, v_l, v_r)$
5 $insertBetween(v_{ins}, top(v_l), top(v_r))$
```

A remote insertion specifies the inserted string *str*, the identifier of the leftmost character *cid*, ground operations $\mathscr{G}$ of the insertion, and the insertion position (v, p). Procedure *remoteIns* re-generates the insertion node v_{ins} (line 1) and updates the $\mathscr{P}$ and $\mathscr{C}$ sets of $v_{ins}.op$ and operations in $\mathscr{G}$ (line 2).

Next, *remoteIns* splits (if necessary) the nodes at the insertion position and narrows down the position among the concurrent insertions using character identifiers (line 3). In Fig. 1-7, there is already a concurrent insertion "`A`" at the position of ins^3_1. When the identifier of "`X`" is smaller than that of "`A`", Peer 2 inserts "`X`" between "`4`" and "`A`".

The procedure then updates the information about ground operations with respect to the concurrent operations (line 4): if the neighboring node v_l (or v_r) is a concurrent insertion, its ground operations become also the ground operations of the new insertion;

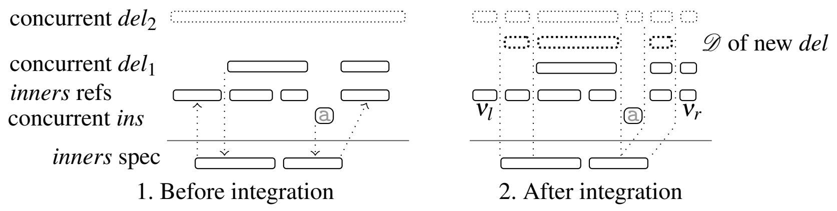


Fig. 2. Integrating a remote deletion

if a concurrent deletion contains both v_l and v_r, the deletion becomes a ground operation of the new insertion. The visibility of the remote insertion is therefore dependent on the visibility of the containing concurrent deletions. This addresses the issue illustrated in Example 2. In Fig. 1-7, del_3^2, which is a ground operation of ins_5^2 (“A”), becomes a ground operation of the new ins_1^3 (“X”).

Finally, *remoteIns* connects the nodes at the outermost layer (line 5). In Fig. 1-7, the “X” node of ins_1^3 connects to “234” of del_3^2 and “A” of ins_5^2.

A remote deletion specifies the inner nodes of the deletion at the time of its creation. The referent inner nodes at the current peer may differ from the specified ones in two ways: (1) the remote peer might have split the inner nodes at deletion boundaries (as shown with the upward arrows in Fig. 2-1); (2) the current peer might have split the inner nodes when integrating concurrent operations (as shown with the downward arrows in Fig. 2-1). In the figure, del_1 is undone and the insertion of “a” *sees* the restored characters of del_1. del_2 sees both “a” and the restored characters of del_1.

Procedure *remoteDel*(*pid*, *pun*, *inners*)

```
1 $(inners, v_l, v_r) \leftarrow prepareInners(inners)$
2 $del \leftarrow Op(pid, pun)$; $\mathscr{D} \leftarrow makeDels(pid, pun, del, inners)$
3 for $v \in \mathscr{D}$ do $placeDel(v, overlappingDels(v))$
4 for $v$ between $(v_l, v_r), v.op.type = ins \land \neg overlapping(v, \mathscr{D})$ do
5     $setGroundOps(v.op, \{del\})$
6 $connectTopNodes(\mathscr{D})$
```

The *prepareInners* procedure (line 1 of Procedure *remoteDel*) uses the specified inner nodes to split the existing nodes at deletion boundaries and returns the new referent inner nodes and their left and right neighbors v_l and v_r (as shown in Fig. 2-2). The *makeDels* procedure (line 2) generates a set $\mathscr{D}$ of nodes for the remote deletion based on the referent inner nodes and concurrent operations. Procedure *placeDel* (line 3) places the generated deletion nodes against the nodes of the overlapping concurrent deletions. For example, a deletion with a larger *pid* is placed above a concurrent deletion with a smaller *pid*. The deletion becomes a ground operation of the concurrent insertions inside the *inners* nodes (lines 4 and 5). For the new deletion nodes at the outermost layer, *connectTopNodes* connects them with the neighboring nodes (line 6).

In Fig. 1-8, the del_1^1 update specifies a single inner node "4567" of ins_1^2. Procedure *prepareInners* splits the "34" nodes of ins_1^2, del_2^2 and del_3^2. Procedure *makeDels* generates the nodes for del_1^1. Procedure *placeDel* places the del_1^1 nodes below those of del_2^2 and del_3^2. del_1^1 becomes a ground operation of the concurrent insertions ins_5^2, ins_6^2 and ins_1^3, which makes characters "X", "A" and "B" invisible. Finally, Procedure *connectTopNodes* connects node "7" of del_1^1 with neighboring nodes "6" of del_3^2 and "8" of ins_1^2.

When a user tries to undo an operation and makes the operation invisible, any operation built on the undone operation becomes groundless (and therefore also invisible). In Fig. 1-7, undo of $undo_4^2$, or redo of del_3^2, would make the insertions ins_5^2, ins_6^2 and ins_1^3 (that are contained in $del_3^2.op.\mathscr{C}$) groundless. If there were other operations built on these insertions, they would also become groundless.

The user may selectively keep the effects of operations built on the operation being undone. In Fig. 1-7, if the user decides to redo del_3^2 and keep the visible effect of ins_6^2, the model then removes ins_6^2 from $del_3^2.op.\mathscr{C}$ and del_3^2 from $ins_6^2.op.\mathscr{P}$.

The execution of a local *undo* starts in the model, with the following steps: (1) integrate local and remote operations in the queues; (2) integrate the undo with the *undo* procedure; (3) move the current position to the edge of the undo; (4) synchronize the model with the view so that the user sees the effects of the undo.

Procedure *undo*(*pid*, *pun*, *op*)

```
1 $push((pid,pun),op.undo.\mathscr{U})$
```

Procedure *undo* integrates both local and remote undo of an operation, which is either a normal operation or an undo of another operation. Procedure *undo* can receive either an *op* or an *undo* element as the argument of the *op* parameter. The procedure simply inserts the identifier of the undo into the corresponding $\mathscr{U}$ set.

For a remote undo, if there has been a concurrent identical undo and the $\mathscr{U}$ set was not empty, inserting a new identifier does not change the visibility of the operation and there is therefore no effect in the view. For a local undo, the real overhead is the move of the current position and the synchronization with the view.

6 Correctness

We consider two traditional correctness criteria, convergence and intention preservation, as defined in [6]. A formal proof is outside the scope of this paper.

Convergence requires that, all peers have the same view when they have integrated and synchronized the same set of operations. Our approach guarantees convergence by enforcing the following properties: (a) models of all peers have the same set of characters; (b) the characters have the same left-right order; (c) the characters have the same visibility.

Intention preservation requires that, for any operation *op*, (a) the effects of executing *op* at all peers are the same as the intention of *op*, and (b) the effect of executing *op* does not change the intention of independent operations.

Intention is not formally defined in [6] and is open to different interpretations. Generally, the *intention* of an operation is decided at the view of the originating peer. More specifically, an insertion is between two specific characters; a deletion removes a string of characters from the view; undo of an insertion removes the inserted characters from the view; undo of a deletion makes the removed characters re-appear in the view and the positions of the re-appeared characters must preserve the intentions of the corresponding insertions.

In our approach, there is also *induced intention* due to concurrent operations and selective undo. More specifically, the intention of an operation is preserved only when the intentions of all its ground operations are preserved. When the effect of an operation disappears (e.g. due to an undo or a deletion), the effects of all operations built on it should also disappear. The algorithms take care that every operation has the same induced intention at all peers.

Notice that undoing a deletion brings the deleted characters back in the view only when the insertions of the corresponding characters are not undone and the characters are not deleted by any concurrent overlapping deletion. That is, undoing a deletion does not change the intention of undoing any insertion or the intention of any other deletions. This is in contrast with related work that defines the effect of concurrent deletions of the same character as a single deletion: undoing a deletion thus changes the intention of all concurrent deletions of the same character. For example, in Fig. 1-8., if undoing del_1^1 makes the entire "`4567`" visible, the intentions of del_2^2 and del_3^2 are not preserved. On the other hand, concurrent undos of the same operation are regarded as a single undo, because they are always unambiguously defined on the same operation.

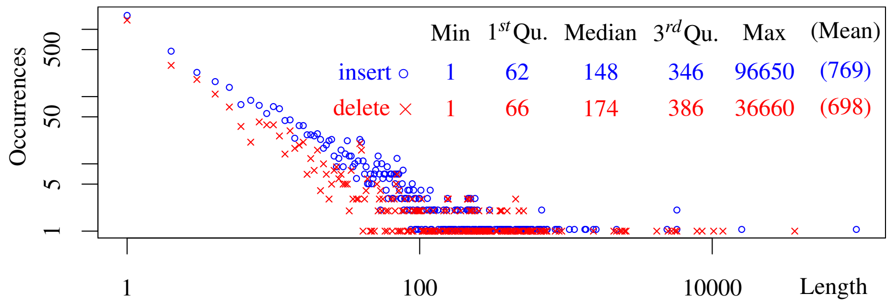
7 Performance

The response time of view operations is an important part of an editor's responsiveness to local user operations. Except selective undo, all view operations are executed completely in the view. Their performance therefore are nearly the same as a single-user editor. However, local view operations are executed only when system resources (CPU, memory etc.) are available, so responsiveness is dependent on the overall performance of the editor, including the more expensive model operations.

Table 1. Time complexity of procedures

split	$O(hn+l)$	*move*	$O(m)$
render	$O(m+rh)$	*undo*	$O(1)$
local *ins*	$O(s+hn+l)$	local *del*	$O(s+hn+l)$
remote *ins*	$O(k_i l+k_d+hn)$	remote *del*	$O(k_d(s+hn+l)+k_i)$

Table 1 summarizes the time complexity of the different procedures. In the table, parameter *m* is the distance of a move, i.e. the number of nodes at the outermost layer



	Min	1^{st}Qu.	Median	3^{rd}Qu.	Max	(Mean)
insert ○	1	62	148	346	96650	(769)
delete ×	1	66	174	386	36660	(698)

Fig. 3. Occurrences of operations with different lengths

a move traverses. l is the length of a character identifier. s is the span of an operation, i.e. the number of nodes between the leftmost and rightmost nodes of the operation, including those not belonging to the operation. h is the height of a node, i.e. the number of layers in the outer-inner structure. n is the number of a node's inner nodes. k_i is the number of concurrent conflicting insertions and k_d is the number of concurrent overlapping deletions. r is the size of the region to be rendered, i.e. the number of nodes in the region where new updates should be synchronized to the view.

We have implemented the core algorithms in Emacs Lisp, aiming at supporting collaborative editing in a widely used open-source editor. We ran two experiments for performance study. The first one is based on a trace of operations for editing a paper. This experiment can be considered to be representative for real-life editing sessions. It is nonetheless based on the trace of a single-user editor. The second experiment is based on generated operation traces that force a large number of conflicting concurrent operations. The measurement was taken under GNU Emacs 24.3.1 running in 32-bit Linux 3.14.2-ARCH on an old ThinkPad T61p (2007 model) with 2.2GHz Intel Core2 Duo CPU T7500 and 2GB RAM.

In the first experiment, we captured the trace of operations for editing a technical paper in a two-week period. The paper is based on the templates and even contents of other papers. Therefore the editing involves a number of copy, paste and deletion of relatively large text. This trace forms the view operations. We then aggregated and converted the view operations into model operations. Figure 3 shows the number of model operations and their lengths (numbers of characters) obtained from the trace.

We ran the trace with two peers. To make sure that operations are valid (i.e. with valid positions and lengths), the peers behave in the following way. For each operation, each peer generates and integrates a local operation, and sends the encoded representation of the update to the other peer. It then receives and integrates the identical update from the other peer, and undoes immediately the last identical operation of the second peer. Therefore, only the effects of the operations originated from the first peer remain. Finally, each peer sends the encoded representation of the undo to the other peer, and integrates the identical concurrent undo from the other peer.

Figure 4 shows the execution time of different procedures. The y-axis represents the execution time in milliseconds (ms). The x-axis represents the time at which the procedures are called.

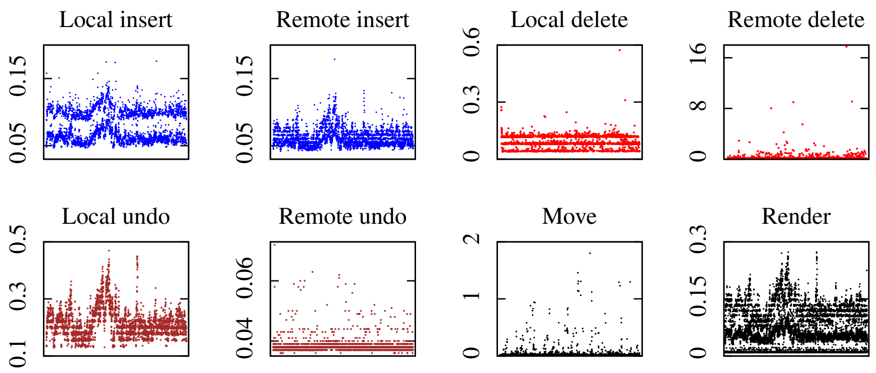


Fig. 4. Execution time (ms) of different procedures

The first important observation is that the execution time of all procedures stays pretty stable and is generally independent of the length of the operation history. This is mostly due to the use of hashing to locate nodes.

Integrating local and remote insertions takes around 0.05 $\sim$ 0.1 ms. Note that the sizes of inserted strings vary from one character to nearly 100K characters, but still the time for integration varies with very small margins. The reason is that character strings are mainly generated by view and networking procedures. Furthermore, string and buffer management in Emacs is efficient.

Integrating local and remote deletions takes around 0.2 ms. There are cases where integrating a local and remote deletion can take up to 0.6 ms and 17 ms respectively. In these cases, a deletion involves a relatively large number of nodes. That is, the s and n in Table 1 are relatively large.

Integrating a local undo takes around 0.2 ms. This includes checking for groundless strings, moving the current position to the undo, and synchronizing the view with the model. Integrating a remote undo takes only 0.04 ms.

Procedure *move* takes less than 0.2 ms the vast majority of times, because editing operations often focus on a small region for a period of time. Even in the occasions where the move distances are long, it takes less than 2 ms.

Procedure *render* takes around 0.1 ms. In the experiment, the model and view are synchronized after the integration of every remote update or local undo. Therefore, the execution time of Procedure *render* does not vary much. It should be pointed out that in the figure, the time of *render* is included in the integration of local undo but not in the other operations.

With respect to memory usage, at the peek, Emacs used an additional 10 MiB of main memory during the experiment. Totally, 20 MiB was allocated for the experiment, including the part that has been freed. This memory consumption is shared by two peers.

The first experiment simulates how the algorithms work with a real-life session. However, it does not reveal how they work when a document is simultaneously edited by a large number of users, because there are only two peers and conflicts of concurrent

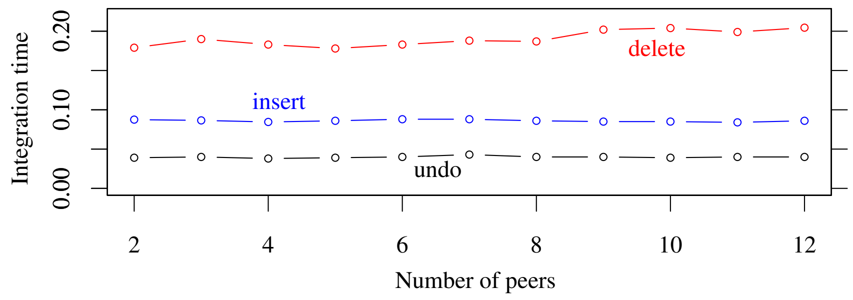


Fig. 5. Execution time (ms) with conflicting operations

operations follow exactly the same patterns. In the second experiment, we study the performance of our work when there are varying number of conflicting or overlapping concurrent operations. In what follows, we use conflicting operation to mean either conflicting insertions or overlapping insertions and deletions.

We generate the operations for N peers as follows. First, we generate a random position p in the view. Then for every peer, we generate a random operation at a random position near p. For the random operations, 50% of them are insertions, 30% are deletions and 20% are undo of an earlier operation that contains position p. An insertion inserts 10 characters. A deletion deletes 7 characters. They are at random positions between $p-3$ and $p+3$, inclusive. After all peers have integrated all local and remote operations, we generate a new random position p, and the same process continues. We run this process until the execution time stabilizes. We vary the number of peers N from 2 to 12. For a reasonably sized document, the number of users that simultaneously edit a very small region, is normally only a very small fraction of the total number of users. So we believe the experiment is sufficient for the most challenging situations in real-world scenarios.

Figure 5 shows the time for integrating remote updates. The time for integrating local operations is not shown, because when a local operation is integrated, there are no concurrent remote operations integrated in the model. The results indicate that the increase of the number of conflicts does not have observable effect on delay.

8 Conclusion

Selective undo has long been regarded as a desirable feature of collaborative editors. However, support for selective undo has remained for two decades at the "necessary first step", namely for character-only operations without any regard of possible undesirable effects. In this work, we proposed support for selective undo in collaborative editing, including support for string operations and management of possible undesirable undo effects. Key to our approach is a layered CRDT that materializes operation relations essential for string operations and selective undo. We analyzed the complexity of the algorithms and presented experimental results. The results indicate that the approach provides sufficient responsiveness to end users.

There are still open issues to be addressed before end users can finally use this work. Our next tasks include a GUI for selection of operations to be undone and management of undo effects, and session management that supports dynamic groups, combination of synchronous and asynchronous operations, network partition, and so on.

Acknowledgments. The authors are grateful for sabbatical support from UiT - The Arctic University of Norway and The Research Council of Norway. This work is partially funded by the french national research program STREAMS (ANR-10-SEGI-010).

References

1. T. Seifried, C. Rendl, M. Haller, and S. D. Scott, “Regional undo/redo techniques for large interactive surfaces,” in *CHI*, 2012, pp. 2855–2864.
2. J. Ferrié, N. Vidot, and M. Cart, “Concurrent undo operations in collaborative environments using operational transformation,” in *CoopIS/DOA/ODBASE (1)*, 2004, pp. 155–173.
3. A. Prakash and M. J. Knister, “A framework for undoing actions in collaborative systems,” *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 4, pp. 295–330, 1994.
4. M. Ressel and R. Gunzenhäuser, “Reducing the problems of group undo,” in *GROUP*. ACM, 1999, pp. 131–139.
5. B. Shao, D. Li, and N. Gu, “An algorithm for selective undo of any operation in collaborative applications,” in *GROUP*. ACM, 2010, pp. 131–140.
6. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, “Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems,” *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.
7. D. Sun and C. Sun, “Context-based operational transformation in distributed collaborative editing systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 10, pp. 1454–1470, 2009.
8. S. Weiss, P. Urso, and P. Molli, “Logoot-undo: Distributed collaborative editing system on P2P networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1162–1174, 2010.
9. W. Yu, “Supporting string-wise operations andselective undo for peer-to-peer group editing,” in *GROUP*. ACM, 2014.
10. C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *SIGMOD*. ACM, 1989, pp. 399–407.
11. L. André, S. Martin, G. Oster, and C.-L. Ignat, “Supporting adaptable granularity of changes for massive-scale collaborative editing,” in *CollaborateCom*. IEEE, 2013.
12. G. Oster, P. Urso, P. Molli, and A. Imine, “Data consistency for P2P collaborative editing,” in *CSCW*. ACM, 2006, pp. 259–268.
13. N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, “A commutative replicated data type for cooperative editing,” in *ICDCS*. IEEE Computer Society, 2009, pp. 395–403.
14. H.-G. Roh, M. Jeon, J. Kim, and J. Lee, “Replicated abstract data types: Building blocks for collaborative applications,” *J. Parallel Distrib. Comput.*, vol. 71, no. 3, pp. 354–368, 2011.
15. A. Imine, P. Molli, G. Oster, and M. Rusinowitch, “Proving correctness of transformation functions functions in real-time groupware,” in *ECSCW*, 2003, pp. 277–293.
16. M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, “Evaluating CRDTs for real-time document editing,” in *DocEng*. ACM, 2011, pp. 103–112.