Introducing Selective Undo Features in a Collaborative Editor

Weihai Yu
Department of Computer Science
UIT - The Arctic University of Norway
Weihai.Yu@uit.no

Abstract

Undo is an important functionality of editors. Selective undo is widely regarded as an important feature for collaborative editing. However, even after nearly three decades of active research and development, there is still no practical support of selective undo for collaborative editing. This paper introduces the selective undo features that we have implemented as part of a collaborative editing subsystem in the GNU Emacs text editor.

1 Introduction

Undo is a key feature of editors. In a normal 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 meaningful linear sequence of operations. In the research community of collaborative editing, *selective undo* is widely regarded as an important feature [4, 7, 9, 11–14, 17]. With selective undo, a user can undo an earlier operation, regardless of when and where the operation was generated.

Selective undo can also be useful when there is no collaboration among multiple users, as shown, for instance, in [16].

Despite wide acceptance in the research community, selective undo is practically not supported in collaborative editors. For example, with Google Drive (`https://drive.google.com`), a user can only undo operations in the reverse chronological order as they were performed in the user's local machine.

An editor supporting selective undo should have the following features.

1. Operation granularity. Text editing is character-string based in nature. Operations on existing text may have arbitrary granularity. Undo or redo character by character would be very inconvenient. Furthermore, some operations may consist of a number of sub-operations. Examples include indentation of source-code blocks, global text substitutions, etc.

2. Operation dependencies. Undoing an arbitrary earlier operation may lead to undesirable effects due to operation dependencies. For example, a user first inserts

This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

a misspelled word and then makes a correction. The correction depends on the insertion of the misspelled word. It is undesirable to undo the insertion of the misspelled word alone and leave the correction part behind.

3. Operation selection. The user should be able to conveniently view, find and select an earlier operation in order to undo its effects.

4. Convenience. In most of the time, the user should be able to effortlessly undo a default operation. In the case of a single-user editor, it is often the last performed editing operation.

Existing research systems support only selective undo of operations with fixed granularity (e.g. characters or unbreakable lines). Consequently, users can typically only undo earlier operations character by character or line by line, even for operations such as copy-paste, find-replace or select-delete. To the best of our knowledge, there is no collaborative editing system that handles undesirable undo effects due operation dependencies, support for convenient operation selection, or accounts for default operations to undo. Only our recent work supports selective undo of string-wise operations with arbitrary granularity [17, 18] and handles operation dependencies for selective undo [18].

In this paper I introduce our support for all the above-mentioned features implemented in the widely used GNU Emacs text editor (`https://www.gnu.org/software/emacs/`). Our collaborative editing subsystem in Emacs is called Wyde. In [18], we presented the support of the first two features, namely selective undo of string-wise operations and dealing with undesirable effects of selective undo. In this paper, I will focus on the other features, i.e. operation selection and effortless undo and redo. I will also report my first experience of using the new system.

This paper is organized as the following. Section 2 presents related work and highlights the contributions of this paper. Section 3 briefly reviews our CRDT approach [18] to real-time collaborative editing. Section 4 presents the display and filtering of editing histories in Wyde, and how to navigate through the histories. Section 5 presents how to perform undo and redo effortlessly in Wyde. Section 6 reports my first experience using Wyde. Finally, Section 7 concludes.

2 Related Work

Most of the present collaborative editing work is based on operational transformation (OT) [3, 12, 13]. With OT, a local operation is performed immediately on the local replica of a document; a remote operation is transformed and integrated in the local site based on the positions of the existing and concurrent operations. The time complexity depends on the lengths of operation histories (linear at best). As a critical issue with OT, it is hard to design correct operation transformation functions that is generally applicable to various integration algorithms [5]. One common way to relax certain required conditions for transformation functions is to enforce a global total order in which operations are transformed and integrated at all sites [15]. As a consequence, OT approaches generally do not scale well and practically require the involvement of central servers. Furthermore, support for off-line (or asynchronous) editing while enforcing a global total order is non-trivial.

Lately, there appear a new family of approaches to collaborative editing based on commutative replication data types (CRDT) [2, 6, 8, 10, 14, 17]. 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. Furthermore, at different sites, operations can be integrated in different orders. Ahmed-Nacer et al. reported in [1] 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 an operation 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 fixed objects (characters or unbreakable lines) [4, 7, 9, 11–14]. Only our previous work [17, 18] supports selective undo of operations with dynamic and appropriate granularity. Furthermore, [18] is the first work that accounts for possible undesirable effects of undo due to operation dependencies.

The work presented in this paper is built on [18], with the following new features:

- undo or redo of a selected previous *group operations* like multiple substitutions and block indentation,
- display of the editing history,
- filtering of displayed history, such as according to selected regions in the document or using a regular expression, or both,
- navigation through the operation history,
- selection of operation and undo or redo,
- effortless undo of default operations.

These features are unique even for single-user editors. This is due to our CRDT data structure that materializes the editing operations, which makes the implementation of all these features pretty straightforward. In contrast, the implementations of undo in existing single-user editors are based on operational transformation. For example, Emacs supports regional undo (a restricted form of selective undo) as a built-in feature. Emacs maintains a list of data elements representing the last performed operations. To find the operations in a region, Emacs walks backward through the list and transforms the operation positions in the data elements with respect to the current document state. This means that in principle we have to walk through the entire list to find all operations in the region. For a long list, this can be an expensive operation.

Azurite [16] is an Eclipse (`http://www.eclipse.org`) plug-in where every editing operation is associated with some meta-data, called dynamic segment, containing information like the position of the operation in the document and the character string of the operation. Performing an editing operation involves updating all existing dynamic segments. In addition, it has to deal with complicated relations among overlapping operations.

3 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 (Figure 1).

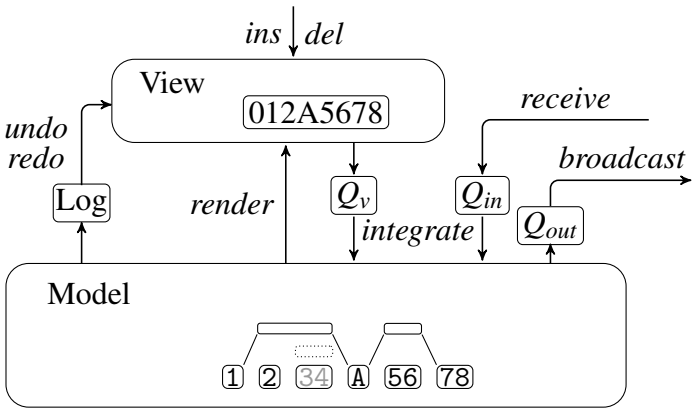


Figure 1: View, model and operations

A peer concurrently receives local operations generated by the local 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.

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.

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*. Operation op_g is a ground operation of *op*, if *op*'s existence depends on the existence of op_g. The parent-child relationship defines dependencies among operations.

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

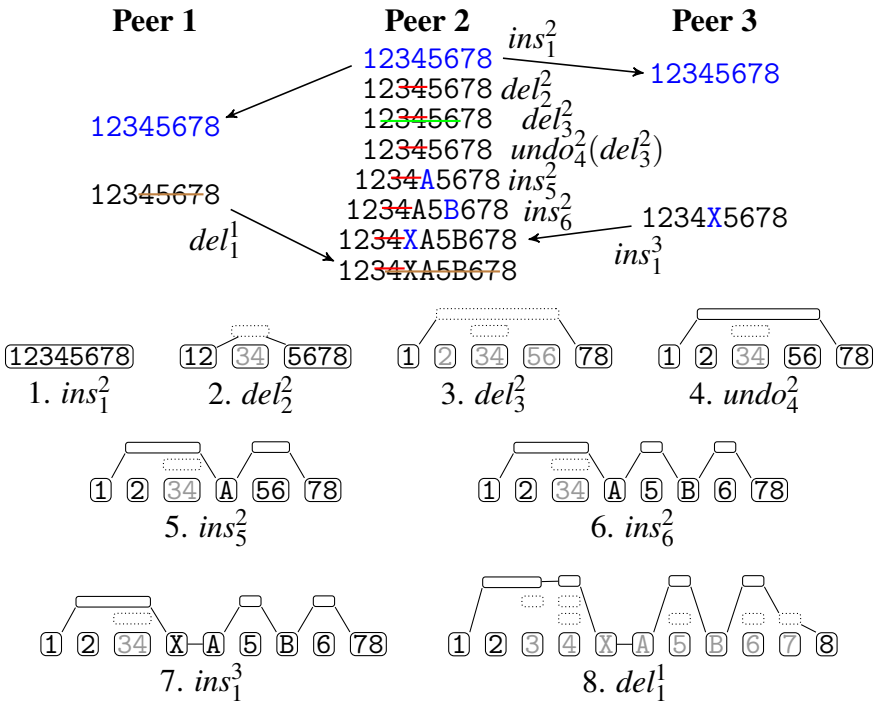


Figure 2: Examples of model updates

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 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 2 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.

Different peers can update the model concurrently. The model data structure is a CRDT that has the following convergence property: when all peers have applied the same set of updates, the states of the model at all peers converge.

We refer the interested readers to [18] for more details on the data structure and algorithms on the model.

Materialization of editing operations in the model makes a number of tasks easier, including support for selective undo and handling of operation dependencies [18], as well as user friendliness features like display of operation history and selection of operations to undo or redo, as described in the following sections.

4 Viewing and selecting operations

In a single-user editor without selective undo, undo or redo is relatively easy. The user simply undoes the latest editing operations in reverse chronological order. Sometimes it

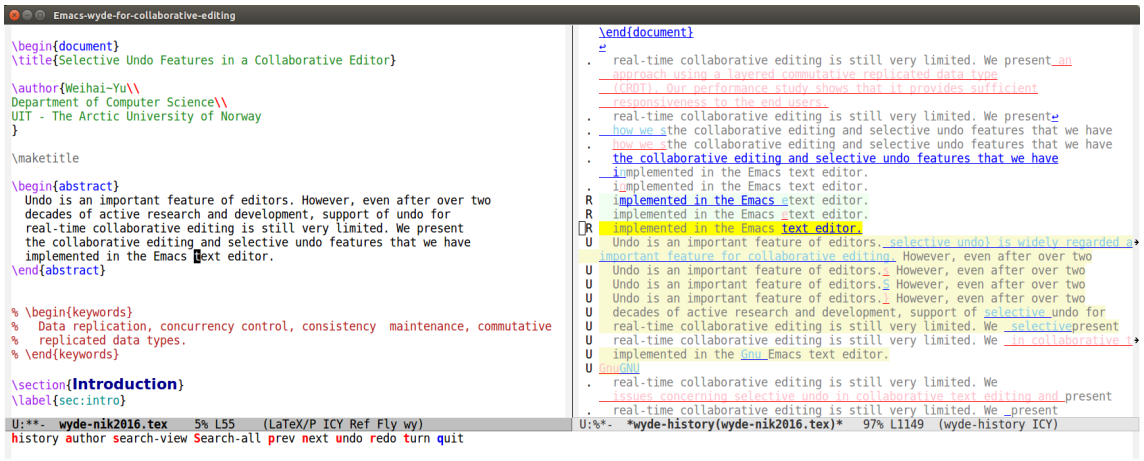


Figure 3: Operation history view

is not easy to know when to stop undoing if there are more than a handful of operations to undo. Typically, the user keeps undoing until it has gone too far, and then corrects this with a number of redos.

To perform a selective undo, the user must be able to view previous operations in a comprehensible way. Figure 3 shows a document (this current paper) under editing in Wyde and a history view to its right side.

In the history view, every operation is preceded with a leading character for its current state: "U" indicates that the operation is undone; "R" indicates that the operation has been first undone and is finally redone; a dot "." indicates that nothing has happened with the operation.

The character string of the operation is underlined. The string of an insertion is displayed in blue; the string of a deletion is in red. If the string is not visible in the document, it is displayed in dimmed color. The operation strings are displayed with the surrounding text (in light gray).

The operations in the history view are in the chronological order in which they were originated. If the user performs operation op_1 prior to operation op_2, op_1 is displayed before op_2. Subsequent undo or redo of the operations does not change the displaying order.

Wyde maintains the complete operation history of a document, even persistently across editing sessions. The number of operations can thus be overwhelmingly large for the user to navigate through. Wyde provides a number ways to display the relevant subset of operations: operations in a given region, operations whose text matches a regular express and operations generated by a particular user.

If the user selects a region in the document, the history view only displays the operations restricted to the region. Displaying operations in a region is often the most effective way to zoom into the relevant operations. Some operations, for instance global text substitution, are widely spread in the document. Regular expression based filtering can be more effective in this case. Filtering based on users provide another option, useful when the user knows who performed the operations.

When the user activates a *select-and-undo* sub-session, a history view is displayed and a sub-menu appears in the echo area at the bottom of the editor, to help the user perform a number of tasks with typing a single character:

- `h` to re-display a fresh new **h**istory,
- `a` to search for operations performed by a particular **a**uthor,
- `s` to **s**earch for operations in the view containing substrings matching a regular expression,
- `S` to **s**earch for operations in the entire history with a regular expression,
- `p` to move to the **p**revious operation,
- `n` to move to the **n**ext operation,
- `u` to perform an **u**ndo,
- `r` to perform a **r**edo,
- `t` to **t**urn the direction of the current undo or redo chunk,
- `q` to **q**uit the current select-and-undo sub-session.

Searching for operations matching a regular expression is different from the traditional way of searching in the document. The text in the current document state matching a regular expression might be involved with multiple editing operations. On the other hand, searching for operations will find operations *containing* the matching text, regardless of whether the text is currently visible in the document or whether it is has been split by other operations.

The user can navigate up and down through the operations, and expand the number of displayed operations when the navigation goes beyond the displaying boundary. The current selected operation during the navigate is highlighted in yellow. During the navigation, the cursor in the document moves to the operation boundary accordingly. This not only provides a nice way of navigating through the document, but also presents a rich context of the operation.

5 Undo and redo behavior

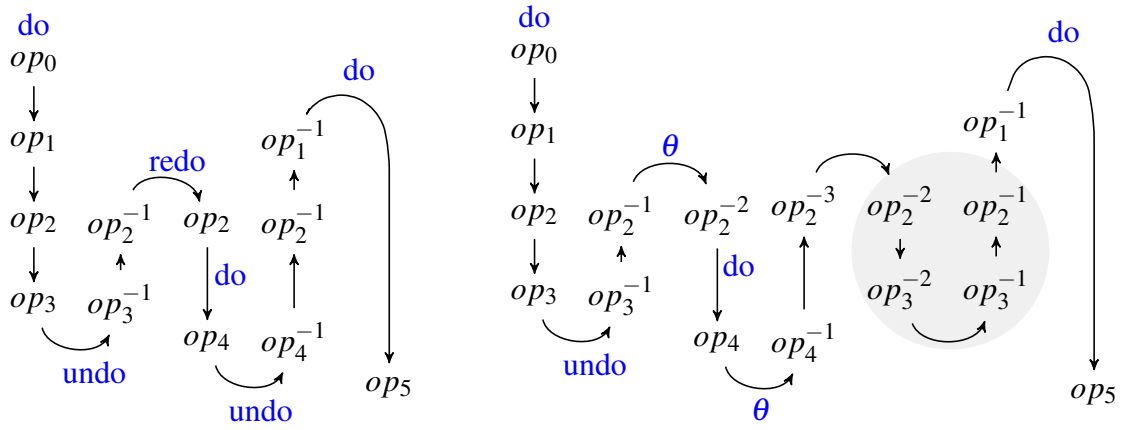
Selective undo is a powerful feature, but it can be complicated to use. We will show that performing undo and redo in Wyde is at least as convenient as in a single-user editor.

Undo and redo in single-user editors

In a single-user editor without selective undo, it is usually more convenient to perform undo in most use cases.

A single-user editor typically has *undo* and *redo* as two distinct operations (Figure 4-a). If the last editing operation was a normal update or redo operation, an *undo* operation undoes the last operation. If the last operation was an undo, and *undo* operation undoes one more operation and expands the undo sequence. During the undo process, a previously undone operation is regarded as non-existent. For example, op_2^{-1} in Figure 4-a is performed immediately after op_4^{-1}, as if op_3 were non-existent. The *redo* operation works in a similar way. Notice that an undo or redo is always related to the corresponding original operation.

The built-in undo mechanism of Emacs is different. There is only a single *undo* operation (Figure 4-b), which either starts a fresh new undo sequence and undoes the immediate previous update operation, or continues with the current undo sequence. When an *undo* operation follows immediately another *undo* operation, it will undo one more operation; otherwise, it undoes the last update operation, which could be a normal update or an undo operation. Thus, if an *undo* operation follows an operation that does not modify the document, like a cursor movement, it will start a fresh new undo sequence and undo the immediate previous update operation, regardless of whether it is an undo



(a) distinct undo and redo (b) undo only

Figure 4: Undo and redo in single-user editors

or not. If that previous update operation is an *undo*, this effectively achieves a redo. In Figure 4-b, θ is a non-update operation. An *undo* operation following op_2^{-1} and θ undoes op_2^{-1}, depicted as op_2^{-2}, which has the same effect as op_2.

Notice that in the undo-only approach, an *undo* operation undoes a previous operation in history without relating to its original operation. Consequently, when continuing an undo sequence, a previously undone operation is not regarded as non-existent. In Figure 4-b, because op_2^{-1} precedes op_2^{-2}, op_2^{-2} is performed after op_2^{-3}. To achieve the same undo effect as in Figure 4-a, the user has to walk through the extra undo subsequence as shown in the shaded circle in Figure 4-b.

Since human users normally relate undos and redos with original update operations, the distinct undo-redo model is usually easier to use. On the other hand, the undo-only model is more suitable for handling concurrent undos in a collaborative editor, because an undo specifies explicitly which "version" of an operation it undoes. Therefore we adopt the undo-only model in the model and support the distinct undo-redo model in the view.

It is easy to perform an undo in a single user editor, mainly because the effect of an undo is unambiguously defined. The user does not have to make any choice. Therefor the key to support the same level of convenience in a collaborative editor with selective undo is to make useful default selections, so that the user does not have to make any choice most of the time.

Undo and redo in Wyde

When a user is undoing or redoing a sequence of operations, the last undone or redone operations form a "chunk". In the history view in Wyde, an undo chunk is highlighted in light yellow and a redo chunk is highlighted in light green (as shown in Figure 3).

In a select-and-undo sub-session, when the operation at point is not currently undone, typing "u" undoes the operation and starts a new undo chunk. If the user types a second "u", the chunk expands. During the expansion of a chunk, the already undone operations are regarded as non-existent and will be skipped over (so there may be "holes" in a chunk). The expansion also skips operations that were not originated by the same user, as these operations are regarded as belonging to different "input flows". If a user has started an undo chunk with an operation too early in the history, she may *turn* the direction of the undo, so the chunk expands in the other direction.

A redo works in a similar way.

In Wyde, which operation to undo or redo depends either on the selected operation (i.e. the operation at point when the select-and-undo sub-session is active), or on the last update operation (when the sub-session is not active), as shown in the Table 1.

operation in question	operation to undo	operation to redo
op	***op***	no effect
undo(*op*)	***undoUp***(), *undoDown*()	***op***
redo(*op*)	***op***	*redoUp*(), ***redoDown***()

Table 1: Operation to undo or redo

In the table, "operation in question" stands for the selected operation or the last update operation. *op* stands for a normal update operation. If the operation in question is a normal update or redo, an *undo* operation undoes the corresponding update. If the operation in question is an undo and the undo is inside the current undo chunk, it expands the current undo chunk and continues undoing. If, however, the undo is not inside the current chunk, Wyde will re-establish an undo chunk around the undo in question before the expansion. *undoUp*() expands the undo chunk upward and returns the new uppermost operation. *undoDown*() expands the chunk downwards. Similarly, *redoDown*() and *redoDown*() expand the redo chunk.

Note that if the user does not explicitly select an operation or turn the direction of undo or redo, the undo and redo behavior is shown in bold in the table. This is exactly the same as in the distinct undo-redo model of a single-user editor. Therefore the undo and redo functionality in Wyde is at least as convenient to the user as in a single-user editor.

6 My first experience

I have been writing this paper as a LaTeX document in GNU Emacs with Wyde features turned on. The main purpose of doing this include, (1) to manually test the software (in addition to pretty comprehensive unit tests), (2) to spot potentials of improvements, and (3) to be able to report my first experience to the interested readers. This first experience does not involve concurrent collaboration, but I think it might still be of interest, as the introduced features are unique even for single-user editors.

The editing history of a document edited in Wyde is kept persistent, along with the document itself. The user can actually undo or redo any operation ever performed on the document. Because Wyde is primarily for collaborative editing, operations are eventually sent to the remote peers that are concurrently editing the same document. To maintain the editing history, Wyde simply appends all outgoing messages to a local file.

When a user opens a document, Wyde re-plays the entire history, so that the model of the document is at exactly the same state as if the document had never been closed.

There are however many reasons things can go wrong and the result of a re-play might not be the same as the document last saved. The software may be buggy, or the user may have edited the document with another editor or with Emacs but with Wyde mode turned off. To get around the problem so that the user can continue editing the document, after re-playing the history, Wyde compares the resulting document with the last saved version. If there is any difference, Wyde applies the "diff" and generates an error report. The user may still continue with the editing. The software maintainer, i.e. I, can use the error report for software maintenance.

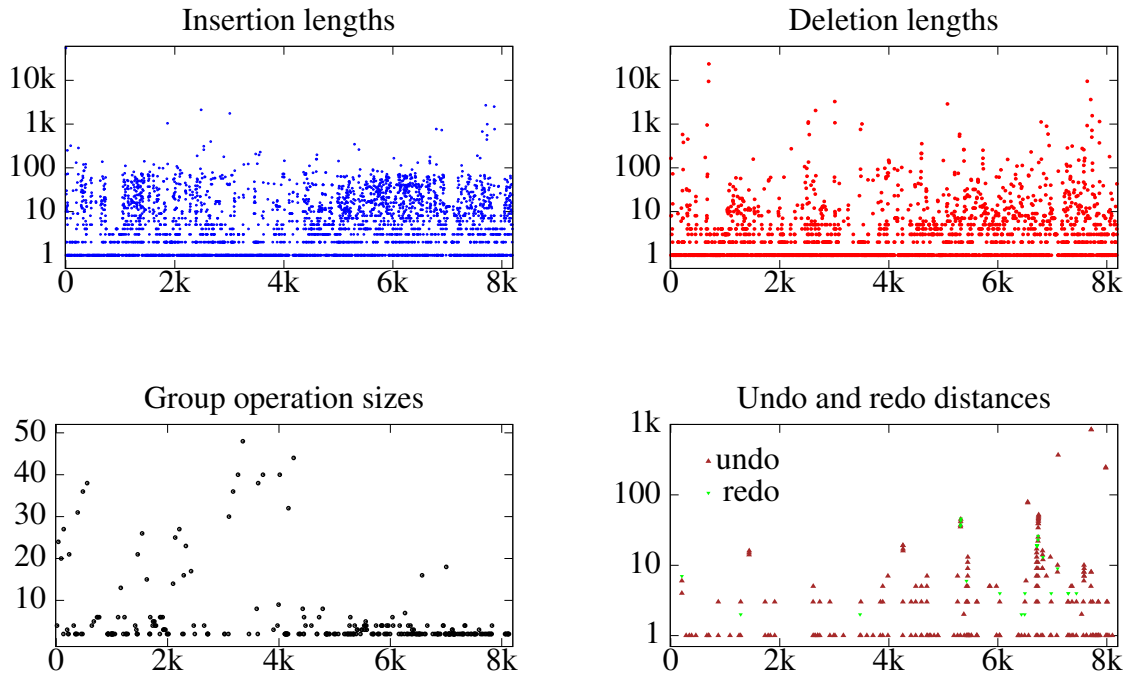


Figure 5: Usage of editing operations

Wyde has a debugging mode. When the debugging mode is turned on, the user may inspect various internal states of the software or even modify the software on the fly (which is very well supported in Emacs Lisp).

As up to this date, writing this paper has revealed three software bugs. Two of the bugs were buried deeply down in the model and could be non-trivial to hit in unit tests. Each bug took two to three days to fix.

In addition, the writing of this paper was halted for about a month due to re-writing of a component of the software and fine-adjustment of some features.

In general, the writing has been smooth, despite the bugs and the halt. I did have some moments with satisfaction after undoing earlier updates, which were impossible without selective undo.

I have collected some usage information about the editing operations that I have performed. I did this primarily out of curiosity, but the collected information might also be useful for me to figure out how useful particular features are and where the challenges could be. Collecting the information is pretty straightforward, when the entire history of editing the document is available.

Up till now, I have performed over 8000 editing operations. There are around 4000 insertions and 2500 deletions. (Wyde aggregates consecutive insertions and deletions of characters as much as possible until line boundaries.) Some of these insertions and deletions are stand-alone, while others are parts of around 1500 group operations (for instance, multiple substitutions as a single operation). I have undone over 200 times and redone over 30 times.

Figure 5 shows some additional information about the performed operations. The x-axis indecates the sequence numbers of operations. As we can see, it is quite often that the lengths of insertions and deletions are around 10 characters and above. There are even occasional cases where the lengths reached 10k characters. If only character-wise operations are supported, real-time collaborative editing would be very costly, both in

terms of run-time overhead and the number of messages over the network. For the end user, undoing operations character by character would be very painful. Unfortunately, most of the available systems support only character-wise operations (of course some operations, like initial insertion of a character string, can be easily optimized.). At present, only our work supports undo of string-wise operations.

Most of the group operations consist only of two sub-operations, such as auto-correction of miss-spelled words. There are occasional larger group operations, consisting of over 20 sub-operations. Undoing a group operation with a single undo is useful, but in our particular case, it is not convincingly indispensable. I have experienced in other cases, such as editing source code with IDE features, where there are much more frequent group operations and the sizes of them are often large.

The distance of an undo (or redo) is how far the undo (or redo) is from the original editing operation. In the figure, the undos are depicted in brown and redos in green. It shows that I have only performed a handful of selective undos with distances greater than 10. I performed almost all redos immediately following some undos, typically when I undid too much. In retrospect, I normally did not keep the history view constantly open and undoing a little too much is a way of getting the feedback of where to stop a sequence of undos. When I was writing new content, I mostly performed undo in the traditional way, as I was used to. I performed selective undo a little more when I was revising the document (and when I was more conscious about the availability of the features). Obviously, I myself was not yet used to making good use of the new selective undo features.

The resulting LaTeX file is around 49K bytes (excluding the BibTeX file and the images in Figures 3 and 5). The history file is around 1.9M bytes, about 40 times the size of the file being edited. This is also the total size of the messages that would have been sent if there were other people co-editing the document concurrently.

With respect of performance, all editing operations are spontaneous. I would like to refer the interested readers to [18] for the performance of the key operations in the CRDT data model.

7 Conclusion

Selective undo has long been regarded as a desirable feature of collaborative editors. However, support for selective undo has remained for over two decades as a "research feature". In this paper, I have introduced selective undo features implemented in GNU Emacs. I have also reported my own personal experience of using these features. The experience is still limited though, as the document was not edited by multiple authors concurrently. Still, I hope that I have convinced the readers of the fact that selective undo can indeed be practically usable.

References

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

[2] L. André, S. Martin, G. Oster, and C.-L. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *CollaborateCom*. IEEE, 2013.

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

[4] J. Ferrié, N. Vidot, and M. Cart. Concurrent undo operations in collaborative environments using operational transformation. In *CoopIS/DOA/ODBASE (1)*, pages 155–173, 2004.

[5] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *ECSCW*, pages 277–293, 2003.

[6] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *CSCW*, pages 259–268. ACM, 2006.

[7] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, 1994.

[8] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403. IEEE Computer Society, 2009.

[9] M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139. ACM, 1999.

[10] H.-G. Roh, M. Jeon, J. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011.

[11] B. Shao, D. Li, and N. Gu. An algorithm for selective undo of any operation in collaborative applications. In *GROUP*, pages 131–140. ACM, 2010.

[12] 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.*, 5(1):63–108, 1998.

[13] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, 2009.

[14] S. Weiss, P. Urso, and P. Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. Parallel Distrib. Syst.*, 21(8):1162–1174, 2010.

[15] Y. Xu and C. Sun. Conditions and patterns for achieving convergence in OT-based co-editors. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):695–709, 2016.

[16] Y. Yoon and B. A. Myers. Supporting selective undo in a code editor. In *ICSE*, pages 223–233. IEEE/ACM, 2015.

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

[18] W. Yu, L. André, and C. Ignat. A CRDT supporting selective undo for collaborative text editing. In *DAIS*, pages 193–206, 2015.