# Peritext: A CRDT for Collaborative Rich Text Editing

GEOFFREY LITT, MIT CSAIL, USA
SARAH LIM, UC Berkeley, USA
MARTIN KLEPPMANN, University of Cambridge, United Kingdom
PETER VAN HARDENBERG, Ink & Switch, USA

Conflict-Free Replicated Data Types (CRDTs) support decentralized collaborative editing of shared data, enabling peer-to-peer sharing and flexible branching and merging workflows. While there is extensive work on CRDTs for plain text, much less is known about CRDTs for rich text with formatting. No algorithms have been published, and existing open-source implementations do not always preserve user intent.

In this paper, we describe a model of intent preservation in rich text editing, developed through a series of concurrent editing scenarios. We then describe Peritext, a CRDT algorithm for rich text that satisfies the criteria of our model. The key idea is to store formatting spans alongside the plaintext character sequence, linked to a stable identifier for the first and last character of each span, and then to derive the final formatted text from these spans in a deterministic way that ensures concurrent operations commute.

We have prototyped our algorithm in TypeScript, validated it using randomized property-based testing, and integrated it with an editor UI. We also prove that our algorithm ensures convergence, and demonstrate its causality preservation and intention preservation properties.

531

CCS Concepts: • **Human-centered computing** → **Asynchronous editors**; • **Information systems** → Version management.

Additional Key Words and Phrases: collaborative editing, asynchronous collaboration, rich text, Conflict-free Replicated Data Types

**ACM Reference Format:**
Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 531 (November 2022), 35 pages. https://doi.org/10.1145/3555644

## 1 INTRODUCTION

Realtime collaborative rich text editors like Google Docs have become a critical tool in modern knowledge work. An important part of implementing these editors is the collaboration algorithm that determines how to merge edits from users concurrently editing a shared document.

Most commercial collaborative editors are based on the Operational Transform (OT) family of algorithms [12, 45]. While this approach has proven successful in practice, it has a drawback: all known OT algorithms for rich text require a central server to mediate edits. This limits scalability, precludes peer-to-peer decentralized sharing, and limits the flexibility of branching and merging workflows on a document.

Authors' addresses: Geoffrey Litt, MIT CSAIL, Cambridge, MA, USA, glitt@mit.edu; Sarah Lim, UC Berkeley, Berkeley, CA, USA, slimberly@berkeley.edu; Martin Kleppmann, University of Cambridge, Cambridge, United Kingdom, (mk428@cst.cam.ac.uk; Peter van Hardenberg, Ink & Switch, San Francisco, CA, USA, pvh@inkandswitch.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2022 Copyright held by the owner/author(s).
2573-0142/2022/11-ART531
https://doi.org/10.1145/3555644

Meanwhile, Conflict-free Replicated Data Types (CRDTs) [43, 44] are another class of algorithms that allow decentralized editing by modeling documents as data structures in which concurrent operations are commutative. There are many CRDT algorithms for plain text (see Section 2.3), but not much work has been done on CRDTs for rich text. We are not aware of any published algorithms; some open-source implementations have extended plaintext CRDTs to cover rich text, but these extensions cause editing anomalies that fail to preserve user intent.

In this paper, we present a novel CRDT called Peritext which supports collaborative editing of rich text in a decentralized setting. Specifically, we make the following four contributions:

- We demonstrate how naively extending existing plaintext or tree CRDTs does not accurately preserve user intent in the context of rich text editing. In Section 2 we highlight specific editing anomalies in several open source rich text editors.
- We propose a general model of intent preservation in collaborative rich text editing, using a series of example scenarios where two users concurrently edit the same formatted text. This provides a test suite that can be used to evaluate the intent preservation behavior of any algorithm for collaborative rich text editing (Section 3).
- We describe a novel CRDT for rich text called Peritext that satisfies all the criteria of our model for intent preservation. The key idea is to store an append-only set of formatting spans alongside the plaintext character sequence, where each span starts and ends on either side of some character in the sequence, addressed via a stable identifier. The final formatting visible in the editor is a deterministic function of the formatting spans that is independent of the order in which formatting operations arrived at a node, guaranteeing convergence across nodes. We describe a prototype implementation of our algorithm [31], written in TypeScript and integrated with an editor UI based on the ProseMirror library (Section 4).
- We show that Peritext complies with a widely-used consistency model for collaborative editing [45]: we prove in Appendix A that it converges, and we explain how it preserves causality and user intentions. We evaluate our prototype implementation using randomized property-based testing, generating concurrent editing sessions and checking that Peritext merges them correctly.

Peritext supports both real-time and asynchronous collaboration, allowing users to choose their preferred mode depending on the context. Moreover, Peritext provides a basis for local-first [27] rich text editing software, which allows users to continue working while their device is offline, and gives users greater privacy, ownership, and agency over the files they create.

Peritext is not a complete system for asynchronous collaboration: for example, it does not yet visualize differences between document versions. Moreover, in this paper we focus only on *inline formatting* such as bold, italic, font, text color, links, and comments, which can occur within a single paragraph of text. In a future paper we will extend our algorithm to support *block elements* such as headings, bullet points, block quotes, and tables.

## 2 RELATED WORK

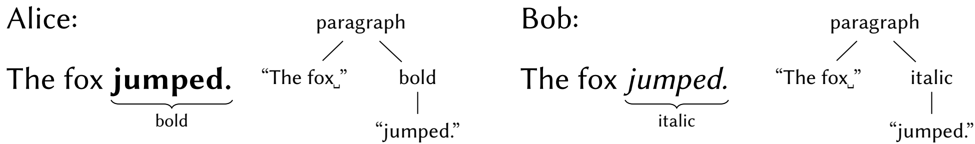
Collaboration on text documents has long been of interest to the CSCW community, although many early systems did not provide fine-grained merging of concurrent edits [24, 28, 40]. Algorithms for synchronous (real-time) collaborative editing allow different users to concurrently update the same document, without locking or any other restrictions, and automatically merge those documents into a state that preserves all users' updates. Such algorithms fall into two main families: the Operational Transformation (OT) approach [12, 45], and the Conflict-free Replicated Data Type (CRDT) approach [43, 44]. Implementing real-time collaborative editing of rich text is notoriously hard; the CKEditor team reports that it took them approximately 42 person-years [7].

## 2.1 Representing rich text as a tree

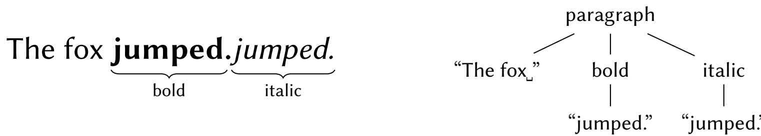
Rich text is often represented as a tree structure such as HTML, XML, or JSON. Both OT [10, 20, 21, 23] and CRDT algorithms [26, 33] have been proposed for handling concurrent updates to such a tree data structure. However, generic tree algorithms are not suitable for rich text collaboration because concurrent edits to the tree can result in anomalies.

For example, consider two users, Alice and Bob, who are editing a document that initially reads:

Alice applies bold formatting to the word "jumped" to obtain "The fox **jumped.**" Concurrently, Bob applies italic formatting to the same word.




In a generic tree structure, the formatting changes are expressed by deleting the word "jumped" from the text node, and adding a new bold/italic node containing the word "jumped." When such concurrent operations are merged, the aforementioned tree algorithms produce duplicated text:




We have observed this undesirable text duplication behavior in the integration of Convergence.io with the Froala text editor [9], which is based on an OT tree data structure [8, 32]. The problem is that the manipulations of the tree structure did not accurately capture the intent of the user: a formatting change was expressed in terms of deletion and insertion, giving the false impression that the user wanted to change the document text. We show in Section 4.4 how we can obtain a tree structure of this form while preserving user intentions.

## 2.2 Operational Transformation (OT) for rich text

Several widely-used commercial rich text editors use Operational Transformation for collaborative editing; in particular, Google Docs [11] is based on Jupiter [35]; Microsoft Word 365 and Dropbox Paper are also believed to use OT, although little is known publicly about their algorithms. Among open source rich text editors, CKEditor [7], ProseMirror [17], and Quill [6] all use OT.

Despite this widespread use, there are very few academic publications exploring OT algorithms for rich text. Ignat, André, and Oster [19] define a sophisticated OT algorithm for rich text, supporting text with style attributes, as well as a tree structure of nested elements with operations to split, merge, and move elements. It avoids the text duplication problem of Section 2.1 by defining operations that are specialized to the task of rich text editing. This algorithm defines eight operation types, and therefore requires 64 transformation functions for each pair of operations [37].

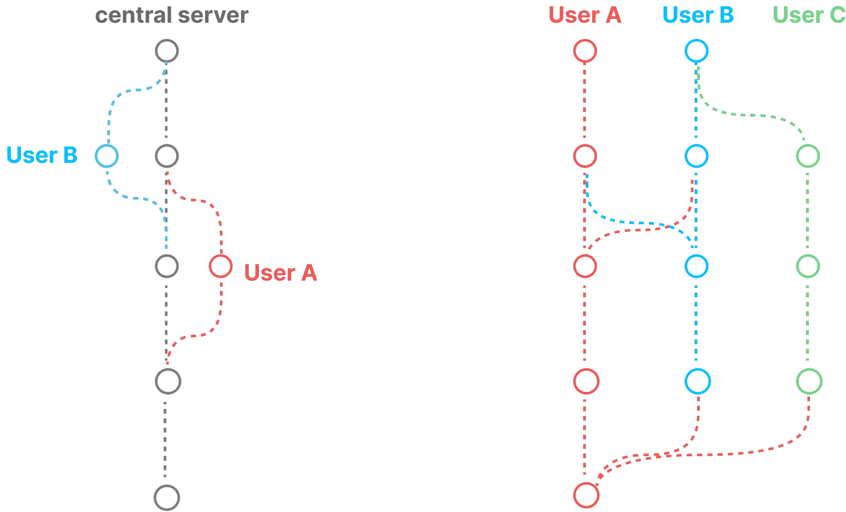



Fig. 1. In rich text OT algorithms, every update to a document is immediately merged into a linear timeline maintained by a server. CRDT algorithms can support a Git-like model where multiple versions (branches) of a document exist side-by-side, and users can choose to merge arbitrary versions.

CoWord (later named CodoxWord) pioneered OT on rich text [46, 48, 51]. It uses three types of operations: *insert*, *delete*, and *update*, where an update operation modifies an attribute of a span of characters. The published transformation functions for CoWord [48] are incorrect: when an update operation is concurrent with an insertion or deletion that falls within the span of the update, they do not transform the length of the update operation or the attributes of the insert operation. This algorithm is therefore not able to guarantee convergence.

While OT for rich text has been widely adopted, it also suffers from several downsides:

- It requires that all communication between collaborators flows via a central server, making it difficult to support peer-to-peer and decentralized architectures. Although peer-to-peer OT is possible in principle [38, 45, 47], all rich text OT algorithms we know of do in fact require a central server since they use a Jupiter-like [35] model.
- It is difficult to support offline editing and disconnected operation: when two users have been working offline and come back online, each of one user's operations have to be transformed with respect to all of the other user's operations, which is slow [30].
- OT algorithms are designed on the assumption of synchronous collaboration, where every user's edits are applied to the shared document as soon as possible. However, in some situations it is preferable to support branching and merging workflows, for example by allowing each user to work on their own version of a document for a while, and to let the users decide when (and whether) to merge versions later. For example, in Figure 1, user A merges an edit by user B while concurrently user B merges an edit by user A, and then

subsequently both users want to merge edits by user C: many OT algorithms do not support this usage pattern.

### 2.3 Conflict-free Replicated Data Types (CRDTs) for rich text

CRDTs are a family of algorithms that allow any number of replicas of a data structure to be maintained in a distributed system, such that each replica can be updated independently, and the states of replicas can be merged in arbitrary ways [41, 43, 44]. CRDTs avoid the problems of OT by making concurrent operations commutative, so that a replica can apply them in any order and converge to the same state as a replica that applied the same operations in a different order.

Many CRDTs for plain text have been proposed, including Treedoc [42], WOOT [39], RGA [43], Causal Trees [14], Logoot [50], LSEQ [34], and YATA [36]. Peritext builds upon RGA, which is similar to Causal Trees, although our algorithm could use any of these plain text CRDTs with minor adaptations. All of these algorithms work by assigning a unique identifier to each character.

To our knowledge, there is no prior published research on CRDTs for rich text, although there are some open source implementations: Ritzy [16], Yjs-richtext [22], and Papyrus [15].

*2.3.1 Ritzy.* Ritzy uses the Causal Trees CRDT [14] as the underlying data model for the sequence of characters. Each character is extended with an `attributes` property containing the formatting (e.g. bold, italic) of that character. When a user selects a span of text and changes its formatting, the CRDT internally updates the attributes of each individual character in the selected span. Concurrent attribute updates to the same character are resolved using a *last write wins* rule (the update with the higher timestamp takes priority).

Storing the formatting separately on each character has an important consequence for collaborative editing behavior. Starting with the sentence "The fox jumped" as before, imagine that Alice bolds the text, while Bob concurrently inserts some new text:

> Alice:
>
> **The fox jumped.**
> (bold)
>
> Bob:
>
> The brown fox jumped.
> ("brown": inserted)

In Ritzy, when the two users merge their states, the final outcome is:

> **The** brown **fox jumped.**
> (bold) (bold)

That is, the word "brown" retains the formatting it had at the time Bob inserted it. Whether this is desirable or not is perhaps a matter of taste; as explained in Section 3.1, Peritext handles this situation differently.

A bigger problem arises if the attribute that Alice attaches to the sentence is not bold formatting, but rather a comment. In most rich text editing software, a comment is attached to a single contiguous span of text, and it would be surprising for the comment to become split into two parts due to concurrent insertion of text in the middle of the comment span. It would be possible to render the comment as a single span nevertheless, but doing this requires care in Ritzy's per-character-attributes model. The overhead of storing attributes individually on each character is also a concern with this model.

*2.3.2 Yjs-richtext.* Yjs implements rich text collaboration [22] on top of YATA [36], a plain text CRDT. It adds formatting by embedding hidden control characters in the sequence of characters to denote the beginning and end of formatting spans, for example "start bold" or "end bold."

This approach avoids the problem from Section 2.3.1, since any characters inserted between start-bold and end-bold control characters become bold. However, this approach has different problems. Consider the following scenario, in which Alice bolds the first two words, and Bob concurrently bolds the last two words:

Alice:

**The fox** jumped.
bold

```
<b>The fox</b> jumped.
```

Bob:

The **fox jumped.**
bold

```
The <b>fox jumped.</b>
```

We represent the control characters as <b> and </b>, and use different colors to distinguish the characters inserted by Alice from those inserted by Bob. After merging we get:

```
<b>The <b>fox</b> jumped.</b>
```

Alice Bob Alice Bob

Yjs renders this document by iterating left to right through the text, remembering at every given point whether the current status is bold or non-bold. When it encounters a <b> character, it makes the following text bold, and when it encounters a </b> character, it makes the following text non-bold. With this strategy, the rendered result is:

**The fox** jumped.
bold

In our opinion, it would be better to make the entire sentence bold, because Alice and Bob have collectively bolded every character. It would be possible to improve the algorithm's behavior in this particular example (e.g. by counting the number of "start" and "end" markers), but further edge cases exist, as shown in Section 2.3.3.

Another example further illustrates the problems with the Yjs approach. Say Alice and Bob both start with the document:

**The fox jumped** over the dog.
bold

```
<b>The fox jumped</b> over the dog.
```

If Alice wants to unbold the text, Yjs implements this by removing the control characters. On the other hand, if Bob wants to unbold only the word "fox," Yjs adds two additional control characters:

Alice:

The fox jumped over the dog.

```
The fox jumped over the dog.
```

Bob:

**The** fox **jumped** over the dog.
bold bold

```
<b>The </b>fox<b> jumped</b> over the dog.
```

When Yjs merges Alice and Bob's edits, both Alice's deletions and Bob's insertions of control characters take effect, so the result is:

> The fox **jumped over the dog.**
> (bold: "jumped over the dog.")
>
> `The </b>fox<b> jumped over the dog.`

Note that the bold formatting now extends even to the words "over the dog," even though those words were never bold in any document version that a user created. If there are no more bold/non-bold control characters later in the text, this anomaly would result in the entire rest of the document becoming bold, which is not justified by the operations performed by the users.

#### 2.3.3 *Counting control characters.*

We attempted to modify the Yjs algorithm to fix the aforementioned problems. For example, instead of toggling between bold and non-bold, we could maintain a count of the number of start-bold and end-bold control characters we have seen, and format text bold if it is preceded by more starts than ends. This approach would fix the two examples in Section 2.3.2, but it exhibits problematic behavior in other edge cases.

For example, assume Alice and Bob both start with the following document:

> **The fox jumped** over the dog.
> (bold: "The fox jumped")
>
> `<b>The fox jumped</b> over the dog.`

Alice unbolds "jumped," while concurrently Bob unbolds "fox jumped" and bolds "dog":

> Alice:
>
> **The fox** jumped over the dog.
> (bold: "The fox")
>
> `<b>The fox</b> jumped over the dog.`
>
> Bob:
>
> **The** fox jumped over the **dog.**
> (bold: "The"; bold: "dog.")
>
> `<b>The</b> fox jumped over the <b>dog.</b>`

As these edits are merged, the bolding of "dog" is lost, because the text "dog" is preceded by the same number of start-bold and end-bold markers:

> **The** fox jumped over the dog.
> (bold: "The")
>
> `<b>The</b> fox</b> jumped over the <b>dog.</b>`

If we changed the algorithm to duplicate the initial start-bold marker, making it `<b><b>The</b> fox</b> jumped over the <b>dog.</b>` then the word "dog" would be bold, but now the unbolding of the word "fox" would be lost. No matter how many layers of fixes we try to apply to this algorithm, we have not found a variant that is able to accurately preserve the users' intentions.

The general problem with control characters it that they make it difficult to represent changes of formatting over time. When partially overlapping spans of text are toggled between bold and non-bold several times, control characters tell us where a span begins or ends, but they do not tell us which formatting is older and which is newer. When several users concurrently modify control characters, it is difficult to determine whether the merged result is intention-preserving.

*2.3.4 Papyrus.* After we finished the implementation of Peritext and released it as open source, we learned of the existence of Papyrus, a rich text CRDT that takes a similar approach to Peritext. We had not found it previously since it had no documentation and was not described in any publication. The author of Papyrus subsequently wrote a brief article comparing Papyrus to Peritext [15].

In summary, Papyrus is similar to Peritext in that it stores formatting annotations outside of the document text by referencing the unique identifier of the first and last character of the formatted span. However, unlike Peritext, it does not support different policies for text insertion at span boundaries, which we explain in Section 3.3.

## 3 CRITERIA FOR INTENT PRESERVATION

In order to reason about the correctness of an algorithm for merging rich text edits, we need a specification for the desired behavior. In the prior sections, we have demonstrated some undesirable behaviors that occur in existing algorithms; in this section we generalize this analysis and develop a model of intent-preserving merge behavior for text with inline formatting.

Such a model is necessarily subjective; although our model is informed by studying the behavior of popular text editors, we do not claim that it is the only possible specification for correct behavior. Still, we think specifying a model is a useful contribution because it allows us (and other researchers) to evaluate behavioral specifications separately from solution implementations.

Furthermore, as with plain text CRDTs, this model only preserves low-level *syntactic* intent, and manual intervention will often be necessary to preserve *semantic* intent based on a human understanding of the text. However, maximally preserving low-level intent is still helpful for supporting easy manual fixes and minimizing work for the user. Intent preservation is also particularly important in asynchronous editing scenarios, where writers cannot react in realtime to edits being made by others.

### 3.1 Concurrent formatting and insertion

**Example 1.** In the example shown in Section 2.3.1 (Alice makes the entire text bold while Bob inserts the word “brown” in the middle), we believe the following outcome is the most desirable:

> **The brown fox jumped.**
> bold

This example suggests a general rule: when formatting is added to the document, it applies to any text inside a range between two characters, even if that text was not present when the formatting was applied.

### 3.2 Overlapping formatting

What should happen when both users apply formatting at the same time to overlapping regions?

**Example 2.** Let’s say Alice bolds the first two words while Bob bolds the last two words:

> Alice:
> **The fox** jumped.
> bold
>
> Bob:
> The **fox jumped.**
> bold

When we merge these two edits, we observe that the word “fox” was set to bold by both users. In this case there is only one reasonable outcome — the whole text should be bold:

> **The fox jumped.**
> bold

**Example 3.** What should happen if Alice bolds some text while Bob makes an overlapping span italic?

> Alice:
>
> **The fox** jumped.
> bold
>
> Bob:
>
> The *fox jumped.*
> italic

It seems clear that **"The"** should be bolded, and *"jumped"* should be italicized. But what formatting should apply to "fox," where both users changed the formatting? Because bold and italic can coexist on the same word, we think it is logical to make "fox" both bold and italic:

> **The** ***fox*** *jumped.*
> bold b+i italic

*3.2.1 Conflicting overlaps.* So far, we have seen merge results that seem to preserve both users' intent. However, not all operations merge so cleanly.

**Example 4.** Consider assigning colored highlighting to some text. Alice applies red coloring to "The fox," and Bob applies blue coloring to "fox jumped":

> Alice:
>
> The fox jumped.
> red
>
> Bob:
>
> The fox jumped.
> blue

What should happen when we merge these two edits? Unlike the previous examples, there is no way to preserve the intent of both users — the word "fox" must be either red or blue, but it cannot be both. As a result, this is a *conflict* that may require some manual intervention to resolve.

One strategy might be to entirely eliminate one user's edit because the two cannot coexist, but to us this seems unreasonably restrictive. Another option might be to blend the two colors together on the word "fox," but then we would be creating a new color that was used by neither Alice nor Bob. In our opinion, the most reasonable behavior is: in just the region where the two formatting ranges overlap, we arbitrarily choose either Alice's color or Bob's color.

> The fox jumped.
> red blue
>
> or:
>
> The fox jumped.
> red blue

It is important that the same color is chosen for everyone who views the document, so the choice needs to be deterministic. And if somebody subsequently changes the color again, then the latest color-change operation determines the final color. This conflict resolution policy is known as *last write wins* or *Thomas write rule* [49].

An alternative to making an arbitrary automated choice would be to expose the conflicting operations in the user interface — for example, the editor could show an annotation noting that a

conflict had occurred, asking a user to review the merged result. However, requiring the user to manually resolve every conflict could become tedious; we prefer to merge documents automatically and then allow the user to subsequently correct any aspects that were not merged to their liking.

**Example 5.** Conflicts occur not only with colors; even simple bold formatting operations can produce conflicts. For example, Alice first makes the entire text bold, and then updates "fox jumped" to be non-bold, while Bob marks only the word "jumped" as bold:

> Alice:
>
> **The fox jumped.** (bold)
>
> **The** fox jumped. (non-bold)
>
> Bob:
>
> The fox **jumped.** (bold)

The word "The" was set to bold by Alice and not changed by Bob, so it should be bold. The word "fox" was set to non-bold by Alice and not changed by Bob, so it should be non-bold. But the word "jumped" was set to non-bold by Alice, and to bold by Bob. In this case we have a conflict on the word "jumped," because the word cannot be both bold and non-bold at the same time. We therefore have to make an arbitrary deterministic choice, just as in the previous example:

> **The** fox jumped. (bold) or: **The** fox **jumped.** (bold) (bold)

A user could even toggle some text back and forth between bold and non-bold several times. In this case, we say that the latest state of Alice's document conflicts with Bob's latest state, but we do not consider earlier states to be part of the conflict.

#### 3.2.2 *Multiple instances of the same mark.*

**Example 6.** There is one more case to consider for handling overlapping marks of the same type. Consider the case where Alice and Bob both leave comments on overlapping parts of the text:

> Alice:
>
> The fox jumped. (Alice's comment on "The fox")
>
> Bob:
>
> The fox jumped. (Bob's comment on "fox jumped.")

The two comments are likely to have different content, and therefore we cannot merge them into a single mark. Moreover, comments behave differently from the colored text in Example 4: although a single character cannot be both red and blue, a single character in the text *can* have multiple associated multiple comments. We can render this in the editor by showing the two highlight regions overlapping:

> The fox jumped. (Alice's comment) (Bob's comment)

### 3.3 Text insertion at span boundaries

Another case we need to consider is: when a user types new text somewhere in the document, what formatting should those new characters have? We argued in Example 1 that if text is inserted into the middle of a bold span, then that new text should also be bold. But it's less clear what should happen when text is inserted at the boundary between differently formatted portions of text.

**Example 7.** Let's say we start with a document where the span "fox jumped" is bold, and the rest is non-bold:

> The **fox jumped**.
> bold

Now Alice inserts "quick␣" before the bold span, and "␣over the dog" before the final period. In all major rich text editors we tested (Microsoft Word, Google Docs, Apple Pages), the result is:

> The quick **fox jumped over the dog**.
> bold

That is, the text inserted before the bold span becomes non-bold, and the text inserted after the bold span becomes bold. The general rule here is that an inserted character inherits the bold/non-bold status of the preceding character. The same applies to most types of character formatting, including italic, underline, font, font size, and text color. However, if text is inserted at the start of a paragraph, where there is no preceding character in the same paragraph, then it inherits the formatting of the following character.

**Example 8.** There are some exceptions to the rule in Example 7: if we insert text at the start or end of a link, or the start or end of a comment, then the major rich text editors place the new text outside of the link/comment span. For example, if "fox jumped" is a link:

> The fox jumped.
> link

After Alice inserts text like in Example 7, the result is:

> The quick fox jumped over the dog.
> link

That is, while a bold or italic span grows to include text inserted at the end of the span, a link or comment span does not grow in the same way. Whether this behavior is desirable is perhaps up for debate, but we note that popular rich text editors are remarkably consistent in this regard, suggesting that this behavior is a deliberate design choice.

If the end of a link and the end of a bold span fall on the same character, and text is inserted after that character, what should happen? The most consistent behavior would be for that text to be bold but not linked. Microsoft Word behaves this way, whereas Google Docs and Apple Pages make the new characters neither bold nor linked.

Table 1. Dimensions of behavior characterizing mark types

| Can marks overlap? | Do marks expand? | Examples |
|---|---|---|
| No | Yes | Bold, italic, colored text |
| No | No | Links |
| Yes | No | Comments |

### 3.4 Generalizing to other mark types

The above examples represent a test suite characterizing reasonable merging behaviors that preserve user intent in a variety of common editing scenarios. While we have used specific formatting types like bold and links to illustrate the behavior, these rules can apply more generally to other kinds of formatting as well.

The formatting types we have described can be categorized along two axes:

- *Can marks overlap?* Is it possible for the same character to have more than one associated mark of this type?
- *Do marks expand?* If a user types at the end of a mark, does it expand to contain the new text?

Table 1 shows how our example formatting types fit into this taxonomy. We believe that most inline formatting in rich text documents can fit into one of these categories; for example, underline obeys the rules for bold and italic. Of course, the developers of a specific text editor could also choose to change the behavior of a specific mark within this framework. For example, a developer might decide that colored text marks should be allowed to overlap, with the overlap region rendering a blend of the colors. From perspective of a collaboration algorithm, each mark type is just some configuration of these parameters, and rendering is left as a concern for the UI layer.

## 4 PERITEXT: A RICH TEXT CRDT

We now introduce the approach we have taken for rich text collaboration in Peritext. We describe our algorithm in four parts:

- Representing the textual content of a rich text document using an existing plain text CRDT
- Generating CRDT operations representing formatting changes
- Applying these operations to produce an internal document state
- Deriving a document suitable for a text editor, based on the internal state

Our approach in designing the algorithm was to capture the user input, and hence the user intent, as closely as possible:

- `insert` operations are generated when the user types or pastes new characters somewhere in the text;
- `remove` operations are generated when the user hits the backspace or delete key somewhere in the text, or overwrites selected text;
- an `addMark` or `removeMark` operation is generated when the user selects some text and chooses a formatting option from the menu or by keyboard shortcut. A “mark” is any property that is applied to a contiguous substring of the text: for example, a word highlighted in bold, or a sentence annotated with an inline comment.

### 4.1 The underlying plain text CRDT

Our implementation uses RGA [43] / Causal Trees [14] as its basis, although in principle it could extend any plain text CRDT.

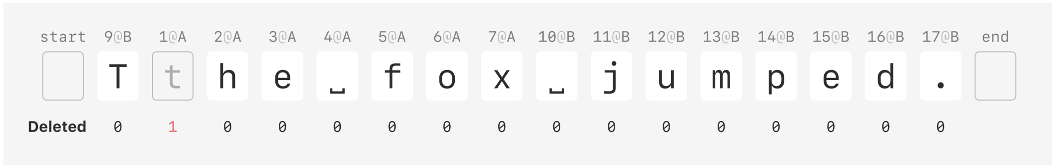



Fig. 2. Our document model stores a Boolean flag alongside each character, which becomes True when the character is deleted. Deleted characters remain in the document model as *tombstones*, which are needed to handle concurrent edits.

Every operation that modifies the state of the document is given a unique, immutable identifier called `opId` (operation ID). An opId is a Lamport timestamp [29]; we write it as a string of the form `counter@nodeId` where `counter` is an integer and `nodeId` is a unique ID (e.g. UUID) of the client that generated the operation. Whenever we make a new operation, we give it a `counter` that is one greater than the greatest counter value of any existing operation in the document, from any client. It is possible for two opIds to have the same counter value if different nodes generate operations concurrently, but since a given client never uses the same `counter` value twice, the combination of `counter` and `nodeId` is globally unique.

opIds are ordered as follows: `counter1@node1 < counter2@node2` if `counter1 < counter2` (using a numeric comparison); if `counter1 == counter2` we break ties using a string comparison of `node1` and `node2`.

*4.1.1 Inserting and deleting plain text.* The key idea of most text CRDTs is to represent the text as a sequence of characters, and to give each character a unique identifier. In our case, the ID of a character is the opId of the operation that inserted the character. To insert a character into a document, we generate an insert operation of the following form:

```
{ action: "insert", opId: "2@alice", afterId: "1@alice", character: "x" }
```

This operation inserts the character "x" after the existing character whose ID is `1@alice`. To determine the position where a character is inserted, we always reference the ID of the existing character after which we want to insert, because these IDs remain stable over time. To insert at the beginning of the document we use `afterId: null`. If two users concurrently insert at the same position (i.e. with the same `afterId`), we order the insertions by their `opId` to ensure both users converge towards the same sequence of characters.

To delete a character from a document, we generate a `remove` operation of the following form:

```
{ action: "remove", opId: "5@alice", removedId: "2@alice" }
```

This operation removes the existing character whose ID is `2@alice` (i.e. the "x" we inserted above). As before, we identify the deleted character by its ID than by its index. When we process a remove operation, the character is not actually deleted from the document entirely, but we just mark it as deleted, leaving behind a *tombstone*. That way, if another insert operation references the deleted character's ID in its `afterId`, we still know where to place the inserted character.

As shown in Figure 2, the state of the plain text document then consists of three things for each character: the opId of the operation that inserted it, the actual character, and a flag to tell us whether it has been deleted. In Figure 2, Alice (nodeId `A`) first typed the text "the fox," generating opIds `1@A` to `7@A`. Then Bob (nodeId `B`) deleted the initial lowercase "t" (the deletion has opId `8@B`) and replaced it with an uppercase "T" (opId `9@B`), and finally Bob typed the remaining text "␣jumped." (opIds `10@B` to `17@B`). The final text therefore reads "The fox jumped." but the lowercase "t" is still in the sequence, marked as deleted.

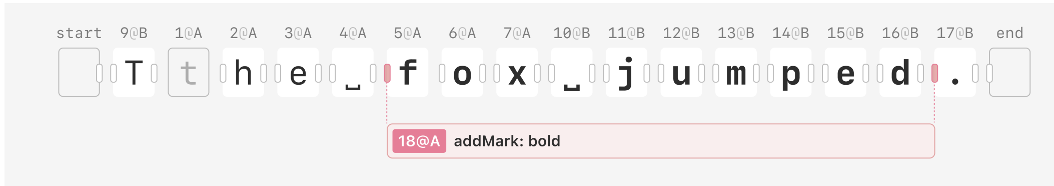



Fig. 3. Conceptually, each character has two formatting anchor positions: one before the character, and one after. Anchor positions determine whether a formatting operation will be extended when new text is inserted on the boundary. For simplicity, subsequent figures will only show anchor positions with attached mark operations.

That is all we need to implement collaborative editing of plain text: inserting a single character, and removing a single character. Larger operations, such as cutting or pasting an entire paragraph, turn into many single-character operations. This can be optimized to be more efficient (as discussed in Section 4.7), but for now, single-character insertions and deletions are sufficient.

### 4.2 Generating inline formatting operations

The next step is to allow the formatting of the text to be changed. Every time the user modifies the formatting of the document, we generate either an `addMark` or a `removeMark` operation. For example, if Alice selects the text “fox jumped” and hits Ctrl+B to make it bold, we generate the operation below, shown in Figure 3:

```
{
  action: "addMark",
  opId: "18@A",
  start: { type: "before", opId: "5@A" },
  end:   { type: "before", opId: "17@B" },
  markType: "bold"
}
```

This operation has an opId of `18@A`. It takes the span of text starting with the character whose ID is `5@A` (i.e. the “f” of “fox”), and ending with the character whose ID is `17@B` (the final period). All characters in this span (including the start character, but excluding the end character) become bold.

The start and end positions of the span are denoted using *anchor positions* that refer to characters by their IDs. Each character in the text has two anchor positions, before and after the character, where the start and end of a formatting operation can be attached.

*4.2.1 Removing a mark.* If the user changes their mind and decides that the text should not be bold or linked after all, we do not remove the `addMark` operation. In fact, we never remove an operation, we only ever generate new operations (we show in Section 4.7 how to weaken this assumption). Instead, we generate a `removeMark` operation that undoes the effect of the earlier `addMark` and sets a sequence of characters back to be non-bold. For example, the following operation, shown in Figure 4, makes “␣jumped” non-bold:

```
{
  action: "removeMark",
  opId: "20@A",
  start: { type: "before", opId: "10@A" },
  end:   { type: "before", opId: "17@B" },
```

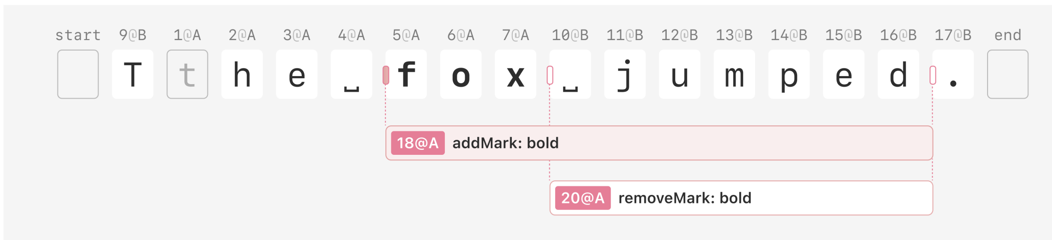



Fig. 4. We can undo any part of a mark by applying the corresponding `removeMark` operation. Here, we unbold the characters ␣jumped, while ensuring that subsequent insertions after `fox` remain bold.

```
  markType: "bold"
}
```

To remove a prior mark entirely, the `start` and `end` of the `removeMark` operation should be the same as for the `addMark` that should be overwritten. In general, a `removeMark` can start and end on any character, regardless of its current formatting, and the effect is to set that span to be non-bold/non-linked/non-italic/etc.

*4.2.2 Inserting text at span boundaries.* A formatting change always occurs in the gap between two characters. The `type: "before"` properties in the example `addMark` operation indicate that the bold span starts in the gap immediately before the "f" (i.e. the "f" is bold, but the preceding space is not), and the bold span ends in the gap immediately before the "." (i.e. the "d" of "jumped" is bold, but the period is not). We could also choose `type: "after"` if we wanted a span to start or end on the gap immediately after a particular character. Moreover, the `start` property could be `"startOfText"` if we want the span to always start right at the beginning of the document, and the `end` property could be `"endOfText"` if we want it to end after the last character.

If another user inserts text within that span, like in Example 1 in Section 3.1, those new characters still fall within the range defined by the `addMark` operation, and so they are also formatted bold. And when text is inserted at the boundaries of the bold span, it behaves like in Example 7: text inserted before the "f" is non-bold, whereas text inserted between the "d" and the period is bold, because it still falls within the span before the period.

The reason that each character has two distinct anchor positions is that this allows us to customize how marks expand at span boundaries. For example, unlike bold formatting, links should not grow when text is inserted at the end, as shown in Example 8. To achieve this behavior, we model the `addMark` operation for a link so that it ends in the gap immediately after the the last character of the mark, as in this operation (shown in Figure 5):

```
{
  action: "addMark",
  opId: "19@A",
  start: { type: "before", opId: "5@A" },
  end:   { type: "after",  opId: "16@B" },
  markType: "link",
  url: "https://www.google.com/search?q=jumping+fox"
}
```

The result is that the "d" of "jumped" is still part of the link, but any text that is inserted after that character will not be part of the link.

Fig. 5. Mark types can include additional metadata, such as a hyperlink. Note that unlike bold marks, which end on the character *after* the span, links end on the *last* character of the span. This approach prevents the link from growing when new text is appended.

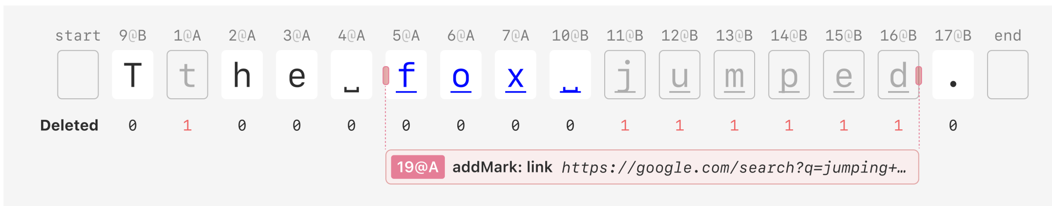



Fig. 6. When a character is marked as deleted, we preserve any attached operations. To insert before the period without growing the link, the insertion should be placed after the tombstone `16@B`.

Note also that an operation with `markType: "link"` has an additional `url` attribute. To change the URL that a link points to, we simply generate another `addMark` operation with a new URL and the same `start` and `end`. For a comment, we include only a unique comment ID in the actual `addMark` operation, and we store the text, author, timestamp, and other details of the comment in a separate CRDT.

A transition from bold to not-bold could happen not only because of the end of an `addMark`, but also because of the start of a `removeMark` (and vice versa for transition from not-bold to bold). To maintain the rule that an inserted character is formatted the same as the preceding character, `removeMark` operations use `type: "before"` on both `start` and `end` for bold and similar types of marks. For example, if we are in the state shown in Figure 4 and then insert "␣suddenly" immediately after the "x," it will be bold, because it falls within the `addMark` span but not within the `removeMark` span.

Inserting a character at the start of a paragraph requires a special case: we have to first insert the character, and then (if necessary) generate additional formatting operations so that the inserted character is given the same formatting as its successor.

For marks that should not grow at the end, such as links and comments (cf. Table 1), `addMark` uses `type: "before"` for `start` and `type: "after"` for `end`. For `removeMark` the roles are reversed: these operations use `type: "after"` for `start` and `type: "before"` for `end`. This ensures that when text is inserted at the end of a link, that text is not part of the link, regardless of whether the link is ending because it's the end of the `addMark` that created it, or the beginning of a `removeMark` that removed the link property from the subsequent characters.

One more detail is required to ensure that insertions at the end of a link behave correctly: it could happen that the character to which the link end is anchored is deleted (a tombstone). In the example in Figure 6, "fox jumped" was linked, and then the word "jumped" was deleted.

Now assume the user wants to replace "jumped" with "frolicked," so they insert that word after the space `10@B`. This position is at the end of the link (the next visible character, the period, is not part of the link), so we expect the new character to be non-linked because links don't grow. However,

when there are tombstones at the position where a new character is inserted, our plain text CRDT by default places the inserted character before the tombstones. This would place "frolicked" before the tombstones of "jumped," making it part of the link, which is undesirable.

To fix this issue, if we need to insert a character at a position where there are tombstones, we scan the list of tombstones at this position. If there are any tombstones whose "after" anchor is the start or end of any formatting operation, we insert the character after the last such tombstone. Otherwise we insert before the tombstones, as usual. In the example above, the after-anchor of character "d" (`16@B`) is the end of `addMark` operation `19@A`, and so we place the new word after that character. This ensures that insertions at the end of a link are placed outside of the link.

If the list of tombstones contains anchors for the start or end of several formatting operations, it is possible that no ideal insertion position exists. In this situation, the inserted text can be placed arbitrarily relative to the tombstones, and the worst-case outcome is that the text is formatted differently from what was desired. We believe that this situation is rare enough that it is acceptable for the user to manually adjust the formatting in this case.

### 4.3 Applying operations

We now show how our algorithm applies operations (from either the local author or a remote collaborator) to update the internal CRDT state. We must ensure that applying concurrent operations is commutative: when two operations were generated concurrently, we can apply those operations in any order, and the resulting document must be the same.

Inserting and deleting characters is straightforward; we use the RGA plain text CRDT logic described in Section 4.1. But applying an `addMark` or `removeMark` operation requires a new approach and is one of the key steps in our algorithm.

As described above, each character has two associated *anchor positions* for formatting operations, one before and one after the character. At each of these anchor positions, we may store a set of operations, which we call an *op-set*. The op-sets may also be absent; when a character is first inserted, its op-sets on both anchor positions are absent. An op-set that is absent is different from an op-set that is present but empty; in our prototype, an absent op-set is represented as `null`.

If an op-set $S$ is present at an anchor position $p$, it has the following meaning:

- $S$ contains exactly the set of `addMark` and `removeMark` operations which *overlap* with that anchor position. In other words, each op in $S$ starts at or before $p$, and ends after $p$.
- The same op-set $S$ also applies to a contiguous span of all positions following $p$ where the op-set is absent; op-sets are only present at positions where the formatting may change. This is a kind of compression: we could redundantly store the same op-set on many anchors, but it saves memory to only store op-sets at boundaries of formatting spans.

Algorithm 1 shows pseudocode for the process of applying an `addMark` or `removeMark` operation to the CRDT state. In summary: let *opsAt* be a mutable list containing op-sets, indexed by anchor positions. We write $opsAt[p] = \perp$ to denote that the op-set is absent at anchor position $p$. We start by updating the op-set at the start position of the new operation. We then update any existing op-sets within the span of the operation. Finally, we add an op-set at the end position of the operation which does *not* contain the operation, to mark that the active operations have changed at that point in the sequence.

We can demonstrate this process with an example. Imagine we start with the text "The fox jumped." with no formatting, and Alice wants to bold the first two words. She generates an `addMark` operation that starts before the "T" (`9@B`) and ends before the space following "x" (`10@B`). Alice applies this operation to her own document data structure by setting the op-set to the left of the

**Algorithm 1** Apply operation *op* of type addMark or removeMark

```
function ApplyOp(op: AddMarkOp | RemoveMarkOp)
    start ← starting anchor position of op
    end ← ending anchor position of op
    if opsAt[start] = ⊥ then                        ▷ Create/update op-set at the start position
        opsAt[start] ← FindPrevious(start) ∪ {op}
    else
        opsAt[start] ← opsAt[start] ∪ {op}
    end if

    for pos such that start < pos < end do          ▷ Iterate rightwards; update op-sets within op
        if opsAt[pos] ≠ ⊥ then
            opsAt[pos] ← opsAt[pos] ∪ {op}
        end if
    end for

    if opsAt[end] = ⊥ then                          ▷ Update the op-set at the end position
        opsAt[end] ← FindPrevious(end) − {op}
    end if
end function

function FindPrevious(pos: AnchorPosition)          ▷ Find nearest op-set to left of given anchor
    while pos ≠ ⊥ do
        if opsAt[pos] ≠ ⊥ then
            return opsAt[pos]
        end if
        pos ← pos.prev                              ▷ Iterate towards beginning of document
    end while
    return ∅                                        ▷ If no op-set found, return empty set
end function
```

Fig. 7. The state of Alice's document after applying a local bold operation.

first character (`9@B`) to be a set containing that `addMark` operation, and setting the op-set to the left of the second space character (`10@B`) to be the empty set. The result is shown in Figure 7.

Concurrently, Bob italicizes the last two words and the period, as in Example 3 in Section 3.2, and illustrated in Figure 8. He generates an `addMark` operation for this span, which starts before

Fig. 8. The state of Bob's document after applying a local italic operation.

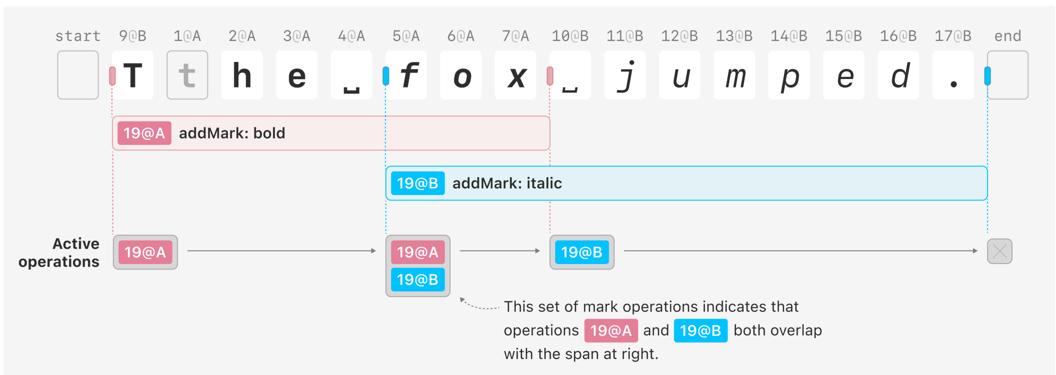



Fig. 9. The state of Alice's document after applying a remote operation from Bob.

the "f" and runs until the end of the text (i.e. after the last character). Bob has not yet seen Alice's bold operation when he generates an italic operation.

Next, we consider what happens when Alice locally applies Bob's operation (Figure 9). The first step is to update the op-set at the start position. In the example, the op-set to the left of the "f" character is absent, so we copy the op-set before the "T" character and add the italic operation to it. Next, we run the loop that iterates through all anchor positions within the span of the operation. We encounter an existing op-set to the left of the second space character: this was previously the empty set, and we add the italic operation to it. Finally, we update the op-set for the end position, because that position now marks a change in the formatting. In the example, the italic operation ends at `endOfText`, and we initialize it to be the empty set. Figure 9 shows the final state of the document after applying the operation. Our invariant has been preserved; each op-set contains exactly the formatting operations that pertain to the subsequent span of characters.

We prove in Appendix A that this algorithm is commutative: no matter in which order the formatting operations are applied, we end up in the same final state. Moreover, it is efficient: we only need to scan over the part of the document that is affected by the formatting operation, not the whole document.[1]

[1]When the first formatting operation is added to a document, FindPrevious will have to go all the way to the beginning of the document, which might be slow on a large document. To speed this up, we can use the following trick: when inserting characters, on some characters (say one in a thousand, chosen randomly) we initialize their op-set to be a copy of the closest preceding op-set (or an empty set if there is none). This has no effect on the correctness of the algorithm, but it ensures that when searching for the closest preceding op-set, we will find one that is present after scanning backwards for only about 1,000 characters (on average), thereby avoiding having to scan the entire document.

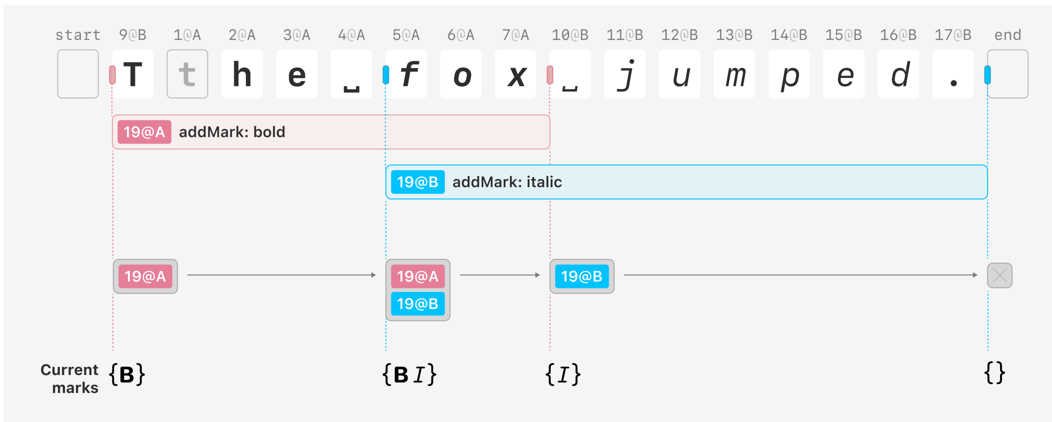



Fig. 10. For each span in the text, we must convert the set of all historical mark operations into a *current* formatting state for that span. In the case of the example from above, we compute that the word fox is both bold and italic.

### 4.4 Producing a final document

The algorithm in the previous section subdivided the document into spans, each associated with a set of mark operations. However, this is not enough to display a formatted document. Each op-set may contain many `addMark` and `removeMark` operations referring to the same formatting attributes; we need to convert this op-set into a *current* formatting state, using an algorithm that is deterministic and independent of the order that operations were received.

To produce a final document, we can iterate over the spans in the document, and for each one convert the associated op-set into a description of the current formatting, like this:

```
[
  { text: "The ",     format: { bold: true } },
  { text: "fox",      format: { bold: true, italic: true } },
  { text: " jumped.", format: { italic: true } }
]
```

Because each operation affects only the aspect indicated by its `markType` (bold, italic, etc.), we can consider the operations for each `markType` separately. In the example from the previous section, we can easily see that the word "fox" should be both bold and italic, because the bold and italic operations overlap with that word (Figure 10).

However, in other cases, the conversion process might be more complicated. There may be multiple operations with the same `markType` for the same span: for example, some text may be bolded and unbolded again several times. With most mark types, the values indicated by different operations are mutually exclusive: a character must be either bold or non-bold; a character cannot have more than one text color. For these mark types, we use a last-write-wins conflict resolution policy, choosing the formatting operation with the maximum operation ID as the winner.

For example, take a span to which two operations apply: an `addMark` operation with ID `19@A`, and a `removeMark` operation with ID `23@B`, both with `markType: "bold"`. We determine the winning operation by comparing their opIds using the ordering defined in Section 4.1. The `removeMark` operation takes precedence because `23@B > 19@A`, and therefore this span is non-bold.

Recall that whenever we make a new operation, we give it an opId with a `counter` that is one greater than the greatest counter value of any existing operation in the document. This ensures

that if the user changes their mind about formatting several times — for example, if they toggle the same word between bold and non-bold several times — then the final formatting operation will have the greatest counter and therefore take precedence over all the earlier operations.

In fact, for the purposes of determining the current formatting of a document, it is not strictly necessary to store the sets of all historical formatting operations: it would be sufficient to store the latest value for each mark type and each span, along with the opId that set that value. However, storing the set of formatting operations is useful if we want to compute what a document looked like sometime in the past, for the purpose of visualizing the differences between versions of a document. Although the current version of Peritext does not support this feature, we believe it is important for asynchronous collaboration, and we plan to add it in the future.

Not all mark types are mutually exclusive—in the case of comments, it is possible for several comments to be associated with the same span, and thus we retain all of the comments for which there is no corresponding `removeMark` operation that deletes the comment.

As further optimizations, we can avoid computing the current formatting for any spans that only contain deleted characters (tombstones), since they do not show up in a WYSIWYG editor, and we can concatenate any adjacent spans whose current formatting is identical.

### 4.5 Incremental patches

We have described a "from-scratch" method for iterating over all the spans in a document and producing a current document state. This is simple to understand, and is a reasonable approach when a document is first loaded, but it is not ideal for ongoing editing. One problem is performance: the from-scratch approach would need to iterate through the whole document on every keystroke, which can become slow in large documents. It also does not match the conceptual model of most text editing UI libraries—a text editor is typically implemented as a stateful object, so each edit should describe what *changed* in the document, not provide a whole new document state.

To address these issues, we have implemented an *incremental* approach to sending updates to the text editor UI. The key idea is to augment the process of applying an operation: in addition to updating the internal op-sets, we also produce *patches* that describe the effects of that operation on the publicly visible formatted document. This helps with performance, because the entire process of incorporating a new operation and updating the editor UI only needs to work in the local region of the document affected by that operation.

To process an `insert` operation, we first use the existing RGA algorithm to determine the insertion position for the new character, and compute its index. To compute the index of the character, we can count the number of non-deleted characters that precede the insertion position. There are data structures [4] that can perform this index computation efficiently, without having to scan the whole document, but they go beyond the scope of this paper.

In addition, we need to determine what formatting to apply to the inserted character. We do this by searching backwards in the array of characters, starting from the insertion position, until we find an op-set of formatting marks. From this set of operations we then compute the marks for the inserted character using the usual last-write-wins logic, and then we construct a patch that inserts a character with that formatting at the index we computed.

For example, a patch to insert the letter "x" at index 6 with bold formatting might look like this:

```
{ type: "insert", char: "x", index: 6, format: { bold: true } }
```

Note that patches use indexes, whereas operations use opIds to identify positions in the text. This is fine because patches are only used to propagate updates from the CRDT to the text editing UI; by running the CRDT logic on the same thread as the UI, we can avoid having to handle concurrency

between these two components. Operations that are sent from one user to another need to use opIds since they need to handle concurrent updates.

A `remove` operation is the simplest to process: we find the character with the appropriate opId; if it is already deleted, we do nothing; if it is not yet deleted, we mark it as deleted and compute the index of the deleted character. We then construct a patch that asks the editor to delete the character at that index.

When applying an `addMark` or `removeMark` operation, we don't need to change any text in the editor, but we may need to update the formatting of several spans of text. As described in Section 4.3, we iterate through contiguous spans in the document, adding the new operation to the op-set for that span. For each span, we then consider whether or not to emit a patch expressing a change in formatting. We use the last-writer-wins logic to compute the formatting for both the old op-set and the newly updated op-set, and compare them. If the formatting has changed, then we emit a patch that updates the formatting for that span.

One interesting consequence of this approach is that applying a formatting operation will sometimes only emit patches for some parts of the operation's span. For example, an operation to bold the entire document might emit patches to only bold certain sections of the document, if there are other concurrent operations that negate its effects on the other sections.

The exact details of emitting a patch are somewhat subtle, because we are forced to entangle the logic for applying an operation with the logic for computing its effects. However, it is straightforward to check the correctness of an implementation, by accumulating incremental patches and then comparing the result to the simpler from-scratch algorithm described in the previous section. We refer the reader to the code in our supplemental material for a full implementation of emitting patches.

### 4.6 Prototype implementation

We have implemented a working prototype of the Peritext CRDT in TypeScript [31]. Our code is based on a simplified version of the Automerge CRDT library [25], and we hope to integrate our algorithm into Automerge in the future. The implementation contains unit tests of many specific scenarios described in this work, as well as a randomized property-based testing suite that checks for convergence. We have tested random edit traces of over 5,000 operations being exchanged between three peers without violating convergence. This gives us confidence that the implementation is correct, because we were able to find several coding bugs in earlier versions of our implementation with much smaller edit traces. A formal proof that our algorithm converges is presented in Appendix A.

For the editor UI, we chose to build on ProseMirror [18], a popular library that is already used in many collaborative editing contexts (usually with OT). Its modular design gave us the necessary flexibility to intervene in the editor's dataflow at appropriate points. We also expect that Peritext would integrate well with other editor UIs since we did not specialize the design to ProseMirror in particular.

Currently, our implementation is somewhat specialized to the small set of marks shown in this paper: bold, italic, links, and comments. However, we intend these to be a representative set of formatting marks, and expect that their behavior would extend to other kinds of user-configurable marks as well.

### 4.7 Performance and efficiency

Our algorithm is designed with performance in mind: in particular, updates only operate locally on the text they touch, and do not require scanning or recomputing the entire document. However, we have prioritized simplicity over performance in some areas: we store each character as a separate

object (which uses a lot of memory); we remember all tombstones and the history of all formatting operations; and the process of finding the character with a particular opId, and computing its index, is not optimized.

These areas in which we have prioritized simplicity are not fundamental to the algorithm, and can be addressed with some implementation effort. For example, Automerge [25] and other CRDTs that support plain text collaboration use a compressed representation [1] of the character sequence, in which characters with consecutive opIds are represented as a simple string rather than an object per character. They also feature data structures that make it efficient to convert an opId into a character index and vice versa, which is needed for integration with editors such as ProseMirror.

We have described Peritext in a simple system model where every operation is stored forever, and it is based on the RGA plain text CRDT, which needs to remember deleted characters as tombstones. This design leads to unbounded storage growth, but it can be optimized in several ways:

- Instead of storing an op-set of `addMark`/`removeMark` operations at an anchor position, it is sufficient to store the value (e.g. bold or non-bold) and highest opId for each mark type. For example, if a word is toggled between bold and non-bold several times, we only need to retain the latest state, and we can discard formatting operations whose effect has been overwritten.
- Mechanisms for garbage-collecting tombstones in RGA [43] apply also to Peritext. When a tombstone is purged, any marks attached to that tombstone need to be moved to the closest preceding or following character that is not being purged; when all characters within a formatting span have been purged, the span can be deleted as well.
- It is also possible to use a tombstone-free plain text CRDT, such as Logoot [50], instead of RGA. Since it is also based on a unique ID per character, the principle of attaching formatting to character identifiers applies equally. If all characters within some formatting span have been deleted, we still need to retain the formatting span, because another user may concurrently insert a character into that span and thus "resurrect" it. A span can be deleted once causal stability [2] has ensured that there will be no more concurrent insertions into that span.
- Even without garbage collection, storing all of the operations forever is not as expensive as it may seem: Automerge's compressed file format can store every single keystroke in the editing history of a text document at a cost of less than one byte per operation. This optimization can also be applied to Peritext.

## 5 CONCLUSION

In this work, we have analyzed the problem of merging concurrent edits to a rich text document. We have shown that extending a plain text document with per-character formatting attributes, or with control characters to indicate start and end of formatting spans, leads to situations in which the authors' intent is not preserved. In contrast, our approach of storing formatting annotations outside of the text, and attaching them to characters via unique IDs, satisfies all of our intent preservation scenarios and guarantees convergence. On this basis we have developed a rich text CRDT that supports overlapping inline formatting, and shown how to implement it efficiently.

Peritext is only the first step towards a system for collaboration on rich text: it simply allows two versions of a rich text document to be merged automatically. A full solution for collaboration (especially asynchronous collaboration) will require further work on visualizing editing history and changes, highlighting conflicts for manual resolution, and other features. Since Peritext works by capturing a document's edit history as a log of operations, it provides a good basis for implementing those further features in the future.

Inline formatting is sufficient for short blocks of text, but longer documents often rely on more sophisticated block elements or hierarchical formats, such as nested bullet points, which Peritext

currently does not model. Further work is required to ensure edits to block structures like bulleted lists can be merged while preserving author intent. Hierarchical formatting constructs raise new questions around intent preservation — for example, what should happen when users concurrently split, join, and move block elements?

Another area for future exploration is moving and duplicating text within a document. If two people concurrently cut-paste the same text to different places in a document, and then further modify the pasted text, what is the most sensible outcome?

We hope that Peritext and other CRDTs for rich text will enable new collaboration workflows which are not possible to build on top of existing data structures for storing text documents. Users could try out their own divergent long-running branches and easily merge them back together with powerful comparison views. People could choose to work in private, or to block out distracting changes being made by others. Rather than seeing document history as a linear sequence of versions, we could see it as a multitude of projected views on top of a database of granular changes.

## DATA ACCESS STATEMENT

Our prototype implementation of Peritext is made freely available under the MIT License at https://github.com/inkandswitch/peritext and at https://doi.org/10.17863/CAM.87326.

## ACKNOWLEDGMENTS

Thanks to Notion for sponsoring Slim Lim's contributions to this project; to Blaine Cook and Tim Evans at Condé Nast for their input throughout the project; to Marijn Haverbeke for ProseMirror and for other guidance; to Kevin Jahns and Seph Gentle for valuable feedback on our algorithm; to Sam Broner, Daniel Jackson, Rae McKelvey, Ivan Reese, and Adam Wiggins for feedback on the paper, and to the many editors, journalists, and writers who showed us how they work and shared their insight and experience with us. Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, Prisma, RelationalAI, SoftwareMill, and Adam Wiggins.

## REFERENCES

[1] Luc Andre, Stephane Martin, Gerald Oster, and Claudia-Lavinia Ignat. 2013. Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing. In *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. ICST, Austin, United States. https://doi.org/10.4108/icst.collaboratecom.2013.254123

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. *Pure Operation-Based Replicated Data Types*. arXiv:1710.04469 https://arxiv.org/abs/1710.04469

[3] Kenneth P Birman, André Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 9, 3 (Aug. 1991), 272–314. https://doi.org/10.1145/128738.128742

[4] Loïck Briot, Pascal Urso, and Marc Shapiro. 2016. High Responsiveness for Group Editing CRDTs. In *Proceedings of the 19th International Conference on Supporting Group Work*. ACM, Sanibel Island Florida USA, 51–60. https://doi.org/10.1145/2957276.2957300

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (second ed.). Springer. https://doi.org/10.1007/978-3-642-15260-3

[6] Jason Chen, Zihua Li, and David Greenspan. [n.d.]. *Quill Delta.* https://github.com/quilljs/delta

[7] Szymon Cofalik and Anna Tomanek. 2018. *Lessons learned from creating a rich-text editor with real-time collaboration.* https://ckeditor.com/blog/Lessons-learned-from-creating-a-rich-text-editor-with-real-time-collaboration/

[8] Convergence Labs, Inc. [n.d.]. *Convergence Developer Guide: Real Time Models.* https://docs.convergence.io/guide/models/real-time-models/

[9] Convergence Labs, Inc. [n.d.]. *Convergence Examples: Froala.* https://examples.convergence.io/examples/froala/

[10] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. 2002. Generalizing Operational Transformation to the Standard General Markup Language. In *ACM Conference on Computer Supported Cooperative Work (CSCW 2002)*. ACM,

58–67. https://doi.org/10.1145/587078.587088
[11] John Day-Richter. 2010. *What's different about the new Google Docs: Making collaboration fast.* https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html
[12] Clarence A Ellis and Simon J Gibbs. 1989. Concurrency control in groupware systems. In *ACM International Conference on Management of Data (SIGMOD 1989)*. ACM, 399–407. https://doi.org/10.1145/67544.66963
[13] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 109 (Oct. 2017). https://doi.org/10.1145/3133933
[14] Victor Grishchenko. 2010. Deep hypertext with embedded revision control implemented in regular expressions. In *6th International Symposium on Wikis and Open Collaboration (WikiSym 2010)*. ACM. https://doi.org/10.1145/1832772.1832777
[15] Victor Grishchenko. 2021. *Papyrus: rich text CRDT from 2012.* https://github.com/gritzko/citrea-model/blob/master/story.md
[16] Raman Gupta. 2015. *Ritzy Editor.* https://github.com/ritzyed/ritzy
[17] Marijn Haverbeke. 2015. *Collaborative Editing in ProseMirror.* https://marijnhaverbeke.nl/blog/collaborative-editing.html
[18] Marijn Haverbeke. 2015. *ProseMirror: A toolkit for building rich-text editors on the web.* https://prosemirror.net/
[19] Claudia-Lavinia Ignat, Luc André, and Gérald Oster. 2017. Enhancing rich content wikis with real-time collaboration. *Concurrency and Computation: Practice and Experience* 33, 8 (March 2017). https://doi.org/10.1002/cpe.4110
[20] Claudia-Lavinia Ignat and Moira C Norrie. 2003. Customizable Collaborative Editor Relying on treeOPT Algorithm. In *8th European Conference on Computer-Supported Cooperative Work (ECSCW 2003)*. Springer, 315–334. https://doi.org/10.1007/978-94-010-0068-0_17
[21] Claudia-Lavinia Ignat and Moira C. Norrie. 2008. Multi-level Editing of Hierarchical Documents. *Computer Supported Cooperative Work* 17 (2008), 423—468. https://doi.org/10.1007/s10606-007-9071-2
[22] Kevin Jahns. 2016. *Rich Text type for Yjs.* https://github.com/y-js/y-richtext
[23] Tim Jungnickel and Tobias Herb. 2015. *TP1-valid Transformation Functions for Operations on ordered n-ary Trees.* https://arxiv.org/abs/1512.05949
[24] Leonard Kawell Jr., Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. 1988. Replicated document management in a group communication system. In *ACM Conference on Computer-Supported Cooperative Work (CSCW)*. ACM. https://doi.org/10.1145/62266.1024798
[25] Martin Kleppmann. 2022. *Automerge.* https://github.com/automerge/automerge
[26] Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (April 2017), 2733–2746. https://doi.org/10.1109/tpds.2017.2697382
[27] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2019*. ACM Press, Athens, Greece, 154–178. https://doi.org/10.1145/3359591.3359737
[28] Michael Koch. 1994. Design Issues and Model for a Distributed Multi-User Editor. *Computer Supported Cooperative Work* 3 (Sept. 1994), 359–378. https://doi.org/10.1007/BF00750746
[29] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563
[30] Du Li and Rui Li. 2006. A performance study of group editing algorithms. In *12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*. IEEE. https://doi.org/10.1109/icpads.2006.18
[31] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2021. Peritext prototype implementation. Available at https://github.com/inkandswitch/peritext. https://doi.org/10.17863/CAM.87326
[32] Michael MacFadden. [n.d.]. *Convergence JavaScript Client: Transformation Functions.* https://github.com/convergencelabs/convergence-client-javascript/tree/master/src/main/model/ot/xform
[33] Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML Collaborative Editing with Undo. In *On the Move to Meaningful Internet Systems (OTM 2010)*. Springer. https://doi.org/10.1007/978-3-642-16934-2_37
[34] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*. 37–46. https://doi.org/10.1145/2494266.2494278
[35] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *8th Annual ACM Symposium on User Interface and Software Technology (UIST 1995)*. ACM, 111–120. https://doi.org/10.1145/215585.215706
[36] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *19th International Conference on Supporting Group Work (GROUP 2016)*. ACM, 39–49.

https://doi.org/10.1145/2957276.2957310
[37] Gérald Oster. 2016. *wiki-transformation.md.* https://gist.github.com/oster/04ca4fc1aaea7de58700
[38] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *9th IEEE International Conference on Collaborative Computing (CollaborateCom 2006)*. IEEE. https://doi.org/10.1109/colcom.2006.361867
[39] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data Consistency for P2P Collaborative Editing. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. https://doi.org/10.1145/1180875.1180916
[40] François Pacull, Alain Sandoz, and André Schiper. 1994. Duplex: A Distributed Collaborative Editing Environment in Large Scale. In *ACM Conference on Computer Supported Cooperative Work (CSCW 1994)*. ACM, 165–173. https://doi.org/10.1145/192844.192900
[41] Nuno Preguiça. 2018. *Conflict-free Replicated Data Types: An Overview.* arXiv:1806.10254 https://arxiv.org/abs/1806.10254
[42] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS.2009.20
[43] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006
[44] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. ACM, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29
[45] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction* 5, 1 (March 1998), 63–108. https://doi.org/10.1145/274444.274447
[46] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. 2006. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Transactions on Computer-Human Interaction* 13, 4 (Dec. 2006), 531–582. https://doi.org/10.1145/1188816.1188821
[47] Chengzheng Sun, Yi Xu, and Agustina Ng. 2017. Exhaustive Search and Resolution of Puzzles in OT Systems Supporting String-Wise Operations. In *ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW 2017)*. ACM, 2504–2517. https://doi.org/10.1145/2998181.2998252
[48] David Sun, Steven Xia, Chengzheng Sun, and David Chen. 2004. Operational Transformation for Collaborative Word Processing. In *ACM Conference on Computer Supported Cooperative Work (CSCW 2004)*. ACM, 437–446. https://doi.org/10.1145/1031607.1031681
[49] Robert H Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4, 2 (June 1979), 180–209. https://doi.org/10.1145/320071.320076
[50] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 404–412. https://doi.org/10.1109/ICDCS.2009.75
[51] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. 2004. Leveraging Single-User Applications for Multi-User Collaboration: The CoWord Approach. In *ACM Conference on Computer Supported Cooperative Work (CSCW 2004)*. ACM, 162–171. https://doi.org/10.1145/1031607.1031635

# A CORRECTNESS OF PERITEXT

The most common definition of correctness in collaborative editing systems requires the algorithm to satisfy the following three properties [45, slightly paraphrased]:

**Convergence:** When the same set of operations have been applied at all sites, all copies of the shared document are identical.

**Causality preservation:** Let $O_b$ be an operation that is generated by site $s$, and let $O_a$ be an operation that had already been applied by $s$ at the time $O_b$ was generated. Then all sites apply $O_a$ before applying $O_b$.

**Intention preservation:** For any operation $O$, the effects of applying $O$ at all sites are the same as the intention of $O$, and the effect of applying $O$ does not change the effects of concurrent operations.

In this appendix we show that the Peritext algorithm satisfies these properties.

### A.1 Causality preservation

We begin with the causality preservation property. Peritext is based on a monotonically growing set of operations: even when text or formatting is deleted from a document, this edit takes the form of new `remove` or `removeMark` operations being generated and applied. There is no user action that causes the set of operations in a document to shrink.

A *version* of a document can be seen as the set of all operations that have ever been applied to it, starting from the empty initial document. Thanks to the convergence property, this set uniquely defines the resulting document state. To merge two document versions is then easy: we simply take the union of the two sets of operations. This union operation also preserves the property that the set of operations in a document grows monotonically over time.

To ensure causality preservation, we must additionally define a partial order $\rightarrow$ over the set of operations. For every operation there is a unique point in time when that operation was generated, and this event took place in the context of a particular document version (namely the document state in which the user input occurred). Let $O_b$ be an operation that was generated in the context of document version $V$, where $V$ is the set of all prior operations that had been applied when $O_b$ was generated. Then we say that $O_a \rightarrow O_b$ if and only if $O_a \in V$. Moreover, we say $O_b$ and $O_c$ are *concurrent* iff $O_c \nrightarrow O_b$ and $O_b \nrightarrow O_c$. It is not difficult to show that, as long as the set $V$ grows monotonically over time, the relation $\rightarrow$ is a strict partial order.

Causality preservation now requires ensuring that whenever $O_a \rightarrow O_b$, operation $O_a$ is always applied before $O_b$, whereas concurrent operations can be applied in any order. There are several possible approaches to achieve this:

- One option is to attach a vector clock to each operation, and to define $\rightarrow$ in terms of vector clock comparison, which is often the approach taken by causal broadcast protocols [3].
- Another approach is to maintain all operations for a particular document version in an ordered log, and to append new operations at the end when they are generated. To merge two document versions with logs $L_1$ and $L_2$ respectively, we scan over the operations in $L_1$, ignoring any operations that already exist in $L_2$; any operations that do not exist in $L_2$ are applied to the $L_1$ document and appended to $L_1$ in the order they appear in $L_2$. Thus, operations are applied in the same order they appear in the log, and that order is consistent with the $\rightarrow$ relation.

Various optimizations exist to improve the efficiency of these baseline algorithms. Since causality preservation mechanisms are well established, we refer to the literature [5] for further details.

### A.2 Intention preservation

The definition of intention preservation given above depends on the precise meaning of the term *intention*, which is difficult to formalize. Sun et al. [45] use the following, fairly loose definition: "The intention of an operation $O$ is the execution effect that can be achieved by applying $O$ on the document state from which $O$ was generated." This definition leaves the desired behavior unspecified in various situations:

- In the case of a new character $X$ originally inserted between two existing characters $A$ and $B$, that insertion should take place between the same two characters at all sites, even if those characters have moved to different document indexes in the meantime. If $A$ and/or $B$ have been concurrently deleted, or if another insertion has concurrently occurred between $A$ and $B$, it is unclear what it means for the operation to have the "same effect".
- In the case of a character $X$ being deleted from some original context, then the same character $X$ should be deleted from the same context at all sites, even if that character has moved to a different document index in the meantime. If the context has changed or if the same

character has been concurrently deleted by another user, it is again unclear what it means for the operation to have the "same effect".
- If one user marks a word as bold and another user marks the same word as non-bold, there is no final state that preserves both users' intentions and also ensures convergence, since a word cannot be both bold and non-bold at the same time. One option would be to allow the user to manually resolve the conflict, but it is unclear how manual resolution fits within the intention preservation framework.

We believe that the best approach to intention preservation is to ask: *"When two documents are merged, which outcome would be the least surprising to users?"* This is the question that led us to the exploration in Section 3, and we found it most productive to analyze the algorithm's behavior in terms of concrete examples. While we did not employ formal user testing to measure "surprisingness" of merge results, we hope that our reasoning in Section 3 about the most desirable outcome in each example will be plausible to readers and provide a basis for further discussion.

Peritext achieves the desired merge result in all of the examples in Section 3:

**Example 1 (insertion within the span of a concurrent formatting operation).** Peritext defines the start and end of a formatted span in terms of the IDs of the characters at both ends. The formatting then applies to all characters within that range, regardless of whether those characters existed at the time the formatting operation was generated, or whether they were inserted concurrently or later. This results in the desired behavior in the example.

**Example 2 (applying bold formatting to partially overlapping spans).** After both operations have been applied, Peritext breaks the document into three spans: the span where only Alice's bold operation applies, the span where both operations apply, and the span where only Bob's bold operation applies. In each of these spans, the current formatting is independently computed to be bold. As a result, the whole text is bold, as desired.

**Example 3 (partially overlapping bold and italic operations).** Like in Example 2, the text is broken into three spans. In the middle span, where the operations overlap, there are two operations with `markType` bold and italic, respectively. Since different `markTypes` are treated independently, the middle span is set to be both bold and italic. The merged result thus has a bold-only span, followed by a bold-italic span, followed by an italic-only span, as expected.

**Example 4 (partially overlapping highlights in different colors).** Like in the previous two examples, we end up with three spans. In the first and last span, there is only a single highlight operation that defines the color. In the middle span, where the highlights overlap, there are two operations with the same `markType`, and a last-write-wins policy chooses one of the colors arbitrarily but consistently, ensuring the same color is chosen at each replica.

**Example 5 (conflicting bold and non-bold operations on a span).** Like in Example 4, the span with the conflicting operations uses a last-write-wins policy to choose either bold or non-bold arbitrarily.

**Example 6 (partially overlapping comments).** In the span where the two comments overlap, there are two mark operations in the set. The comment mark type is configured so that marks are allowed to overlap, so rather than using the last-write-wins policy to choose one, we allow all of the comments to coexist in the final document.

For each comment operation we use a unique comment ID as the `markType`. The last-write-wins policy chooses one value per `markType` as the current value for that `markType`, but different `markTypes` are independent from each other. By using a different `markType` for each comment, the span where the comments overlap can have multiple associated comments without using last-write-wins to choose one of them.

**Example 7 (insertion at the boundaries of a bold span).** For bold operations, the span starts *before* the first bold character, and ends *before* the first non-bold character following the bold span. When text is inserted immediately before the first bold character, it falls outside of the bold span and is hence non-bold. When text is inserted immediately after the last bold character (i.e. before the following non-bold character), it falls within the bold span and is hence bold. This behavior is consistent with existing word processors.

**Example 8 (insertion at the boundaries of a link).** For link operations, the span starts *before* the first linked character, and ends *after* the last linked character. When text is inserted immediately before the first linked character, or immediately after the last linked character, it falls outside of the link, making it non-linked in both cases. If the end anchor of the link operation falls on a character that has been marked as deleted (tombstone), the algorithm described in Section 4.2.2 comes into play. When a character is inserted immediately after the last non-tombstone character of the link, we scan over the tombstones starting from the insertion position, and place the inserted character after the end anchor of the link span, making it non-linked as required.

A finite list of examples can never serve as definitive proof that an algorithm is always "intention preserving" for all possible documents. However, we believe that the examples above, and their straightforward generalizations, provide us with sufficient confidence that the algorithm is as unsurprising as possible to users, and hence preserves the users' intentions.

### A.3 Convergence

While intention preservation is a somewhat subjective correctness property, convergence has a straightforward formal definition that lends itself to proofs. Convergence is also the key property of the *strong eventual consistency* model that defines the correctness for CRDTs [13, 44].

In this section we prove that the Peritext algorithm converges. Rather than requiring that the same set of operations has been applied at *all* sites, we prove a slightly stronger property:

Theorem A.1. *For any two versions of a Peritext document, if the same set of operations has been applied in both documents, then those documents are in the same state.*

Proof. Let $L_1$ be the log of operations in the order they were applied to one document, and $L_2$ the log for the other document. Because both documents contain the same set of operations, and because a given operation is not applied more than once, $L_2$ is a permutation of $L_1$. Moreover, because of the causality preservation property from Appendix A.1 we know that for any two operations $O_a$ and $O_b$ in the logs, if $O_a \rightarrow O_b$ then $O_a$ must precede $O_b$ in both logs.

We now start with $L_1$ and repeatedly swap the order of pairs of adjacent operations until the modified log is equal to $L_2$. By Lemma A.9 it is always possible to transform $L_1$ into $L_2$ in this way; moreover, we only ever need to swap operations that are concurrent. By Lemma A.2, concurrent operations commute; when we swap two adjacent concurrent operations in a log, the document state resulting from applying the operations in log order therefore remains unchanged at each swap. Therefore, the document resulting from applying the operations in $L_1$ is the same as that resulting from $L_2$, ensuring convergence. □

Lemma A.2. *Let $O_a$ and $O_b$ be two concurrent operations on a Peritext document, and let $D$ be any document version that contains all of the causal dependencies of $O_a$ and $O_b$ (that is, for any operation $O$ such that $O \rightarrow O_a$ or $O \rightarrow O_b$, $O$ has already been applied to $D$). Then $O_a$ and $O_b$ are commutative: that is, applying first $O_a$ and then $O_b$ to $D$ results in the same document state as applying first $O_b$ and then $O_a$ to $D$.*

PROOF. Peritext supports four types of operation: `insert`, `remove`, `addMark`, and `removeMark`. We consider the following cases:

- Both $O_a$ and $O_b$ are `insert` operations.
  Peritext relies on the RGA CRDT [43] to handle character insertions on the document text, and our additions to support formatting do not affect the way insertions are handled. The fact that two concurrent RGA insertion operations commute has been proven in earlier work [13, 26], so we elide the proof here.
- Both $O_a$ and $O_b$ are `remove` operations.
  Because $D$ contains all causally prior operations, the characters being deleted by $O_a$ and $O_b$ must exist in $D$ (possibly as tombstones). The effect of a remove operation is simply to mark the appropriate character as deleted. Regardless of whether $O_a$ and $O_b$ delete the same character or different characters, marking the characters as deleted has the same effect, no matter in which order the operations are applied.
- One of $O_a$ and $O_b$ is an `insert` operation, and the other is a `remove` operation.
  Without loss of generality, assume that $O_a$ is the `insert` and $O_b$ is the `remove`. If $O_b$ were to remove the character inserted by $O_a$, then we would have $O_a \rightarrow O_b$, but since $O_a$ and $O_b$ are concurrent, the character removed by $O_b$ must be different from the character inserted by $O_a$. Since insertion is not affected by whether another character is marked as deleted, the effect of $O_b$ (marking an existing character as deleted) and the effect of $O_a$ (inserting a new character) are unrelated and can take place in either order.
- Each of $O_a$ and $O_b$ is either a `addMark` or a `removeMark` operation.
  Then Lemma A.8 shows that the operations commute.
- One of $O_a$ and $O_b$ is a `remove` operation, and the other is an `addMark` or `removeMark` operation.
  The effect of the remove operation is simply to mark a character as deleted, and it is not affected by any formatting properties of the characters. Moreover, the effect of a formatting operation does not depend on whether a character is marked as deleted. Therefore, the effect of the two operations does not depend on the order in which they are applied.
- One of $O_a$ and $O_b$ is an `insert` operation, and the other is an `addMark` or `removeMark` operation.
  Without loss of generality, assume that $O_a$ is the `insert` and $O_b$ is the `addMark` or `removeMark`. Since $D$ contains the causal dependencies of $O_b$, the start and end characters used as anchors by $O_b$ must exist in $D$. Since $O_a$ and $O_b$ are concurrent, the character inserted by $O_a$ cannot be one of the anchors of $O_b$. Therefore the insertion position of $O_a$ must be either within the span of $O_b$ (after its start and before its end), or outside of it. If it is outside, the two operations do not interact, since a formatting operation only touches characters within its span, so the order of applying $O_a$ and $O_b$ makes no difference.
  If the insertion position is within the span of $O_b$, consider first the case where $O_b$ is applied before $O_a$. In this case, the formatting operation $O_b$ first updates its start and end character, and possibly also updates other characters within its span; thereafter $O_a$ inserts a character on which the op-sets are absent. An insertion operation is not affected by formatting information, and therefore the insertion has the same effect as if $O_b$ had not happened.
  Next consider the case where $O_a$ is applied before $O_b$. In this case, the insertion has the same effect as in the previous case. When the formatting operation $O_b$ is subsequently applied, it updates its start and end character, and also scans over the characters within its span, including the character inserted by $O_a$. However, applying a formatting operation considers each character independently, and for characters within its span it does not modify any op-sets that are absent. Since these properties are absent for the character just inserted by

$O_a$, applying $O_b$ does not modify the character inserted by $O_a$, and for all other characters within its span $O_b$ has the same effect as if $O_a$ had not been applied. Therefore the combined effect of $O_a$ and $O_b$ is the same, regardless of the order in which they are applied.

These cases cover all possible combinations of operation types for $O_a$ and $O_b$. □

*Definition A.3.* Define an *anchor position* in some Peritext document $D$ to be any value that can appear as the `start` or `end` property of a newly generated `addMark` or `removeMark` operation:

- `{type: "before", opId:` $c$`}` or `{type: "after", opId:` $c$`}`, where $c$ is the ID of an existing `insert` operation in $D$;
- `{type: "startOfText"}` or `{type: "endOfText"}`.

*Definition A.4.* Define a total order $\prec$ on anchor positions in a document $D$ such that:

- `{type: "before", opId:` $c$`}` $\prec$ `{type: "after", opId:` $c$`}` for any character ID $c$;
- `{type: "after", opId:` $c_1$`}` $\prec$ `{type: "before", opId:` $c_2$`}` whenever character $c_1$ precedes character $c_2$ in the document $D$ (other characters may appear between $c_1$ and $c_2$);
- `{type: "startOfText"}` $\prec$ `{type: "before", opId:` $c$`}` for any character ID $c$;
- `{type: "after", opId:` $c$`}` $\prec$ `{type: "endOfText"}` for any character ID $c$;
- $\prec$ is transitively closed, i.e. $\forall c_1, c_2, c_3.\ c_1 \prec c_2 \wedge c_2 \prec c_3 \Longrightarrow c_1 \prec c_3$.

*Definition A.5.* Define $opsAt(D, p)$ to be the set of mark operations at a particular position $p$ in the Peritext document $D$, that is:

- If $p$ = `{type: "before", opId:` $c$`}` for some character ID $c$, then $opsAt(D, p)$ is the op-set associated with the anchor point before the character with ID $c$;
- if $p$ = `{type: "after", opId:` $c$`}` for some character ID $c$, then $opsAt(D, p)$ is the op-set associated with the anchor point after the character with ID $c$.

We write $opsAt(D, p) = \bot$ to indicate that the op-set is absent.

*Definition A.6.* Define $opsAtOrBefore(D, p)$ to be:

- If $opsAt(D, p) \neq \bot$ then $opsAtOrBefore(D, p) = opsAt(D, p)$;
- If $opsAt(D, p) = \bot$ and $\exists p'.\ p' \prec p \wedge opsAt(D, p') \neq \bot$ then $opsAtOrBefore(D, p)$ is the non-absent op-set that most closely precedes $p$ in $D$;
- If $opsAt(D, p) = \bot$ and $\nexists p'.\ p' \prec p \wedge opsAt(D, p') \neq \bot$ then $opsAtOrBefore(D, p) = \{\}$.

*Definition A.7.* Let $D$ be a Peritext document, and $O_a$ an operation. Then define $D \circ O_a$ as the result of applying $O_a$ to the document $D$.

Lemma A.8. *Let $O_a$ and $O_b$ be two concurrent operations on a Peritext document such that each is either a* `addMark` *or a* `removeMark` *operation. Let $D$ be any document version that contains all of the causal dependencies of $O_a$ and $O_b$. Applying first $O_a$ and then $O_b$ to $D$ results in the same document state as applying first $O_b$ and then $O_a$ to $D$.*

Proof. An `addMark` or `removeMark` operation reads and modifies the op-sets of characters within its span. We therefore show that all of the op-sets in the document after applying $O_a, O_b$ are the same as after applying $O_b, O_a$. The current formatting of the document is then simply a deterministic function of those op-sets (i.e. the last-write-wins logic for choosing the most recent value for each `markType`); whenever the op-sets are identical in two documents, their formatting must also be identical.

Let $start_a$ and $end_a$ be the start and end anchors of the span of $O_a$ respectively, and similarly let $start_b$ and $end_b$ be the anchors of $O_b$. Since $D$ contains the causal dependencies of the operations it must contain $start_a$, $end_a$, $start_b$, and $end_b$. Formatting operations generated by Peritext always

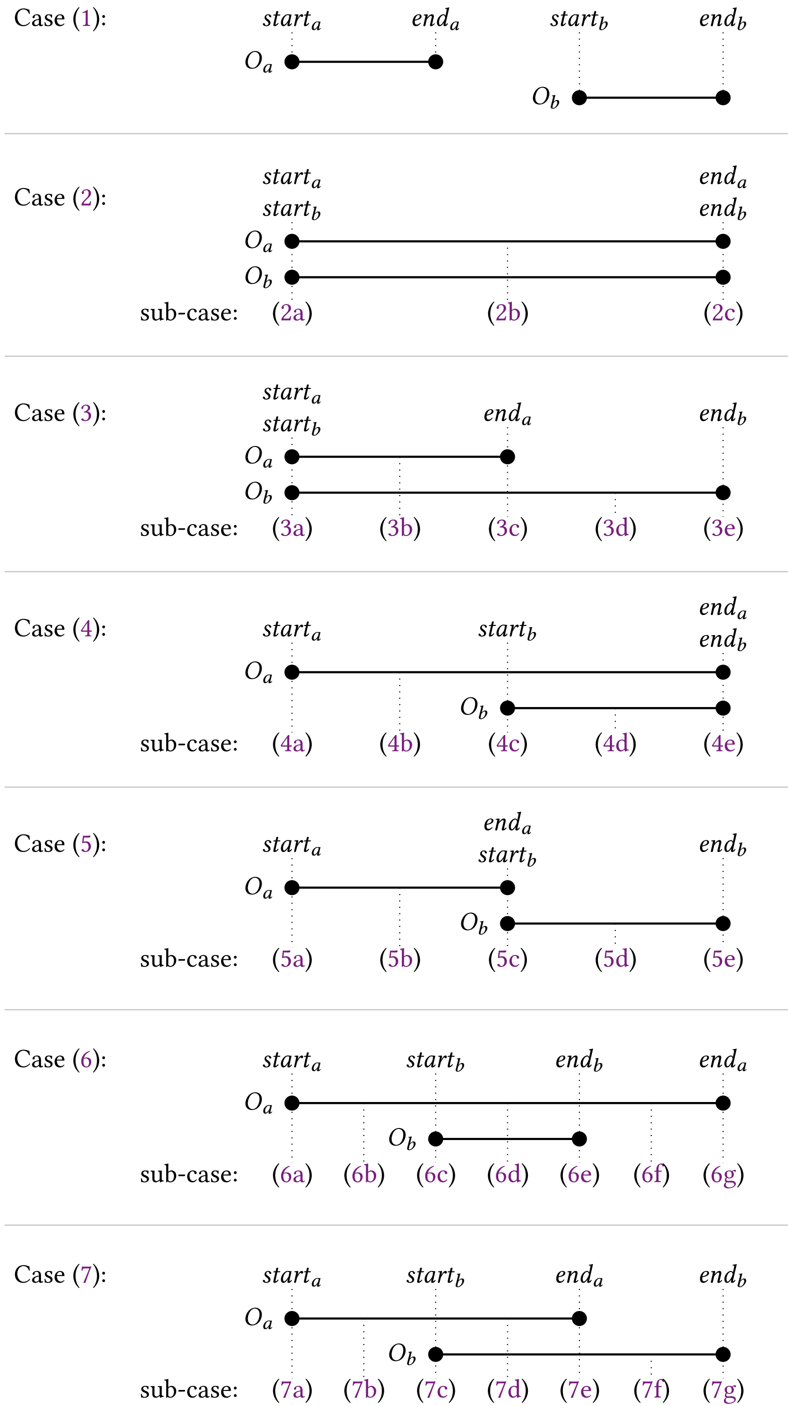



Fig. 11. Visualization of the possible interactions between two formatting operations. These cases and sub-cases appear in the proof of Lemma A.8.

have the property that $start_a \prec end_a$ and $start_b \prec end_b$ according to the order in Definition A.4. We now consider the following cases, which are illustrated in Figure 11:

(1) $start_a \prec end_a \prec start_b \prec end_b$ or $start_b \prec end_b \prec start_a \prec end_a$: the spans of the two operations do not overlap. Since a formatting operation touches only characters within its span, the two operations do not affect each other and thus trivially commute.
There is just one edge case: if $opsAt(D, start_b) = \bot$, and if $O_a$ is applied before $O_b$, then when applying $O_b$, the closest preceding op-set $opsAtOrBefore(apply(D, O_a), start_b)$ might be the set at position $end_a$, which was updated by $O_a$. On the other hand, if $O_b$ is applied before $O_a$, then we might have $opsAt(D, end_a) = \bot$, making the effect of $O_b$ dependent on whether $O_a$ was applied first.
However, if $O_a$ is applied first and $opsAt(D, end_a) = \bot$, $O_a$ initializes it to equal its closest preceding op-set. This is the same set as $O_b$ would observe as its closest preceding op-set if $O_a$ had not been applied beforehand. Therefore, the effect of $O_b$ is the same, regardless of whether $O_a$ is applied first.

(2) $start_a = start_b \wedge end_a = end_b$: the spans of the two operations are identical. We consider each of the anchor positions within the operations' span:
   (a) Let $m = opsAtOrBefore(D, start_a)$. After applying $O_a$ and $O_b$ in either order, the op-set at $start_a$ equals $m \cup \{O_a, O_b\}$.
   (b) Let $m = opsAt(D, p)$ at any position $p$ with $start_a \prec p \prec end_a$. If $m = \bot$ is absent, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$ is present, the op-set at $p$ equals $m \cup \{O_a, O_b\}$ after applying $O_a$ and $O_b$ in either order.
   (c) Let $m = opsAtOrBefore(D, end_a)$. After applying $O_a$ and $O_b$ in either order, the op-set at $end_a$ equals $m$.

(3) $start_a = start_b \wedge end_a \neq end_b$: the two spans start at the same position, but one ends earlier than the other. Without loss of generality, assume $end_a \prec end_b$. We consider each of the anchor positions within the larger span:
   (a) Identical to case (2a).
   (b) Identical to case (2b).
   (c) Let $m = opsAtOrBefore(D, end_a)$. If $O_a$ is applied first, the op-set at $end_a$ is first set to $m$, and then updated to $m \cup \{O_b\}$ after applying $O_b$. If $O_b$ is applied first and $opsAt(D, end_a) \neq \bot$, then $O_b$ updates it to $m \cup \{O_b\}$ and $O_a$ leaves it unchanged. If $O_b$ is applied first and $opsAt(D, end_a) = \bot$, then $O_b$ leaves it absent; when $O_a$ is applied next, it initializes the op-set to be $opsAtOrBefore(apply(D, O_b), end_a)$. This set must equal $m \cup \{O_b\}$ according to one of the previous two sub-cases (applying $O_b$ in case (3a) always results in a op-set at position $start_a$ that is present and contains $O_b$). Thus, after applying $O_a$ and $O_b$ in either order, the op-set at $end_a$ equals $m \cup \{O_b\}$.
   (d) Let $m = opsAt(D, p)$ at any position $p$ with $end_a \prec p \prec end_b$. If $m = \bot$ is absent, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$ is present, the op-set at $p$ equals $m \cup \{O_b\}$ after applying $O_a$ and $O_b$ in either order.
   (e) Let $m = opsAtOrBefore(D, end_b)$. After applying $O_a$ and $O_b$ in either order, the op-set at $end_b$ equals $m$.

(4) $start_a \neq start_b \wedge end_a = end_b$: the two spans end at the same position, but one starts earlier than the other. Without loss of generality, assume $start_a \prec start_b$. We consider each of the anchor positions within the larger span:
   (a) Let $m = opsAtOrBefore(D, start_a)$. After applying $O_a$ and $O_b$ in either order, the op-set at $start_a$ equals $m \cup \{O_a\}$.

(b) Let $m = opsAt(D, p)$ at any position $p$ with $start_a \prec p \prec start_b$. If $m = \bot$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$, the op-set at $p$ equals $m \cup \{O_a\}$ after applying $O_a$ and $O_b$ in either order.

(c) Let $m = opsAtOrBefore(D, start_b)$. If $O_b$ is applied first, the op-set at $start_b$ is first set to $m \cup \{O_b\}$, and then updated to $m \cup \{O_a, O_b\}$ after applying $O_a$. If $O_a$ is applied first and $opsAt(D, start_b) \neq \bot$, then $O_a$ updates it to $m \cup \{O_a\}$, and $O_b$ subsequently updates it to $m \cup \{O_a, O_b\}$. If $O_a$ is applied first and the op-set in $D$ at $start_b$ is absent, then $O_a$ leaves it absent; when $O_b$ is applied next, it initializes the op-set to be $opsAtOrBefore(apply(D, O_a), start_b)$. This set must equal $m \cup \{O_a\}$ according to one of the previous two sub-cases (applying $O_a$ in case (4a) always results in a op-set at position $start_a$ that is present and contains $O_a$). $O_b$ then adds itself to this set, resulting in $m \cup \{O_a, O_b\}$. Thus, after applying $O_a$ and $O_b$ in either order, the op-set at $start_b$ equals $m \cup \{O_a, O_b\}$.

(d) Let $m = opsAt(D, p)$ at any position $p$ with $start_b \prec p \prec end_b$. If $m = \bot$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$, the op-set at $p$ equals $m \cup \{O_a, O_b\}$ after applying $O_a$ and $O_b$ in either order.

(e) Identical to case (2c).

(5) $start_b = end_a$ or $start_a = end_b$: one span begins exactly where the other span ends. Without loss of generality, assume $start_a \prec end_a = start_b \prec end_b$. We consider each of the anchor positions within either of the spans:

(a) Identical to case (4a).

(b) Let $m = opsAt(D, p)$ at any position $p$ with $start_a \prec p \prec end_a$. If $m = \bot$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$, the op-set at $p$ equals $m \cup \{O_a\}$ after applying $O_a$ and $O_b$ in either order.

(c) Let $m = opsAtOrBefore(D, start_b)$. After applying $O_a$ and $O_b$ in either order, the op-set at $start_b$ equals $m \cup \{O_b\}$.

(d) Let $m = opsAt(D, p)$ at any position $p$ with $start_b \prec p \prec end_b$. If $m = \bot$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$ is present, the op-set at $p$ equals $m \cup \{O_b\}$ after applying $O_a$ and $O_b$ in either order.

(e) Identical to case (3e).

(6) $start_a \prec start_b \prec end_b \prec end_a$ or $start_b \prec start_a \prec end_a \prec end_b$: one span falls entirely within the other. Without loss of generality, assume $start_a \prec start_b \prec end_b \prec end_a$. We consider each of the anchor positions within the larger span:

(a) Identical to case (4a).

(b) Identical to case (4b).

(c) Identical to case (4c).

(d) Identical to case (4d).

(e) Let $m = opsAtOrBefore(D, end_b)$. By a similar argument to case (4c), but without adding $O_b$ to the set, the op-set at $end_b$ equals $m \cup \{O_a\}$ after applying $O_a$ and $O_b$ in either order.

(f) Let $m = opsAt(D, p)$ at any position $p$ with $end_b \prec p \prec end_a$. If $m = \bot$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \bot$, the op-set at $p$ equals $m \cup \{O_a\}$ after applying $O_a$ and $O_b$ in either order.

(g) Let $m = opsAtOrBefore(D, end_a)$. After applying $O_a$ and $O_b$ in either order, the op-set at $end_b$ equals $m$.

(7) $start_a \prec start_b \prec end_a \prec end_b$ or $start_b \prec start_a \prec end_b \prec end_a$: the spans partially overlap. Without loss of generality, assume $start_a \prec start_b \prec end_a \prec end_b$. We consider each of the anchor positions within either of the spans:

(a) Identical to case (4a).

(b) Identical to case (4b).

(c) Identical to case (4c).
(d) Let $m = opsAt(D, p)$ at any position $p$ with $start_b \prec p \prec end_a$. If $m = \perp$, it is still absent after applying $O_a$ and $O_b$ in either order. If $m \neq \perp$, the op-set at $p$ equals $m \cup \{O_a, O_b\}$ after applying $O_a$ and $O_b$ in either order.
(e) Identical to case (3c).
(f) Identical to case (3d).
(g) Identical to case (3e).

These cases cover all possible relative orderings of $start_a$, $end_a$, $start_b$, and $end_b$. □

Lemma A.9. *Let $L_1$ and $L_2$ be two sequences of operations that are permutations of each other, with no duplicates. Assume that both sequences are in a causal order, i.e. if $O_a \rightarrow O_b$, then $O_a$ precedes $O_b$ in both sequences. Then $L_1$ can always be transformed into $L_2$ by repeatedly swapping adjacent, concurrent operations.*

Proof. By induction. The base case, where $L_1$ and $L_2$ contain no operations, is trivial. For the inductive step, let $L_1$ be any sequence of operations in causal order with no duplicates. Let $O$ be any operation that could be appended to $L_1$ by the Peritext algorithm, and let $L_1' = L_1 + O$ be the sequence $L_1$ with $O$ appended (the + operator stands for concatenation). Since every operation generated by Peritext has a unique ID, $O$ cannot appear in $L_1$, so $L_1'$ has no duplicates. Moreover, since Peritext preserves causality of operations, $L_1'$ must also be causally ordered; therefore, there can be no operation $O'$ in $L_1$ such that $O \rightarrow O'$.

Let $L_2'$ be any causally ordered permutation of $L_1'$. Since $O$ must appear once in $L_2'$, we can write $L_2' = L_{\text{prefix}} + O + L_{\text{suffix}}$ for some $L_{\text{prefix}}$ and $L_{\text{suffix}}$. Let $L_2 = L_{\text{prefix}} + L_{\text{suffix}}$; since this sequence contains all of the operations besides $O$, $L_2$ must be a permutation of $L_1$. Moreover, because there is no operation $O'$ in $L_1$ such that $O \rightarrow O'$, and the operations in $L_{\text{suffix}}$ are a subset of those in $L_1$, there cannot be any operations in $L_{\text{suffix}}$ that causally depend on $O$ ($O \nrightarrow O'$ for every $O'$ in $L_{\text{suffix}}$). Since $L_2'$ is causally ordered, $L_2$ with $O$ removed must also be causally ordered. Further, since $L_2'$ is causally ordered, $O' \nrightarrow O$ for every operation $O'$ in $L_{\text{suffix}}$, and therefore $O$ must be concurrent to every operation in $L_{\text{suffix}}$.

Since $L_1$ and $L_2$ are causally ordered permutations of each other, by the inductive hypothesis, $L_1$ can be transformed into $L_2$ by repeatedly swapping adjacent, concurrent operations. Therefore, $L_1' = L_1 + O$ can be transformed into $L_2 + O = L_{\text{prefix}} + L_{\text{suffix}} + O$ by repeated swapping. Finally, $L_{\text{prefix}} + L_{\text{suffix}} + O$ can be transformed into $L_2' = L_{\text{prefix}} + O + L_{\text{suffix}}$ by repeated swapping (swapping $O$ with the preceding operation, one element of $L_{\text{suffix}}$ at a time, until the right position is reached). Since $O$ is concurrent to every operation in $L_{\text{suffix}}$, this shows that $L_1'$ can be transformed into $L_2'$ by repeatedly swapping adjacent, concurrent operations. □

Received January 2022; revised April 2022; accepted August 2022