Using conflict-free replicated data types to support block editing

Quentin Lee[a], Martin Li[a], Cas van Rijn[a], Wang Hao Wang[a], Bert Willems[b], Stef Busking[b], Martin Middel[b], Bart Gerritsen[a]

[a]TU Delft
[b]Fonto

Abstract

Recent times once more informed us on the relevance of capable online collaborative tools. For our online collaborative XML editor, we have looked into technologies for constrained block editing which, obeying schemas such as with XML, permit on- and off-line users or agents to add, delete, copy, move, split and merge blocks of text. To that end, we studied the current state of Operational Transformations (OT) and Conflict-free Replicated Data Types (CRDT). Furthermore, after selection of the best-ranking enabling technology, we studied existing CRDT implementations for unstructured texts, and extended a Logoot-based CRDT to implement on-and offline split and merge block support. We designed a generic concept and created a scientific prototype to test the concept on its correctness. For now, we neglected undo and redo operations. Within these limitations, we show that our prototype converges under most circumstances. We verify causality and assess the experience of user editing intent. Finally, we give an outlook and design recommendations for production implementations, and suggestions for tackling the problem of cyclic references in block mergers.

Keywords: Collaborative online editing; online-offline collaborative tools; constrained block editing; structured text editing; conflict free replication; splitting merging blocks; logoot-based CRDT

1. Introduction

Today, the vast majority of document editing is done with the use of feature-rich software environments supporting all basic tasks required for rich-text editing, revision, publishing and archiving, complete with document history. Principal examples are Microsoft Word or Google Docs. Some of these software even offer users the ability to collaborate with multiple distributed editors co-editing the same document simultaneously, on- and offline. This can significantly reduce the time for documents to be produced and converged into a final form. Collaborative editing can be among humans, but also among other agents, such as reporting tools, loggers, or web feeds. Automated and human editing may meet in for instance template-endowed documents, supporting the inlay of structured blocks of data in a streaming fashion. Multi-agent, multi-source editing applications are found in automated-online contracting and trading, code base management systems, and many other domains.

1.1 Block editors

Block-based editors are feature-rich software environments which allow for the editing of structured content without the need of having knowledge about the underlying data structure. In feature rich text-editing the user can make any change at any time. While this gives more freedom it also limits the amount of structure a document has. Structured content aims to organize content according to some predefined syntactic schema. One of the advantages of structured content is easier identification of elements, by humans as well as agents, Khare and Rifkin, 1997. Also, content can be validated against a formalized and configuration maintained schema that can be shared or exchanged, allowing for easy exchange among different editors.

Fonto is a company developing products for authoring and reviewing structured content. The current version of Fonto's editor supports a limited form of collaborative editing, in which a user needs to acquire a 'lock' to gain editing rights for a document. This prohibits users from editing the document for which they do not have the lock. The idea is to investigate whether current editors can be enhanced with on- or offline, distributed collaborative editing possibilities, while preserving block-editing.

The goal of this paper is to research the possibility of collaborative block-editing and create a prototype incorporating this. Main challenges are to support a variety of operations types, such as operations on atomic text elements, as well as block-based operations like insert, delete, move, merge and split blocks, thereby guaranteeing convergence and preserving user editing intent.

Several research efforts have already been reported on the possibilities of collaborative block-editing, Oster et al., 2006, Ignat et al., 2017, Martin et al., 2010, Davis and Ian, 2010. Ignat et al., or other types of collaborative editing. Our research strategy is to explore existing concepts of collaborative editing, extend selected existing ideas with block editing, and provide a unified prototype.

1.2 Organization of this paper

This paper is organized as follows: after this introduction, in section 2 we zoom in on the current state of online collaborative editing concepts, their data structures, operations, their limitations, and merit their potential for the research task at hand. In section 3, we outline the design of our concept, its data structures and its operations. Section 4 presents the construction of the created prototype. We will discuss and demonstrate how to reach overall convergence in finite time. We will discuss how we have tested the prototype and how we have verified the correctness of our prototype. Section 5 describes and discusses results as they materialize from automated and manual testing. In section 6 we evaluate these results, conclude on the achievements and validate conclusions. Finally, we will present our outlook on further progress we see fit and possible.

2 Background

2.1 OT and CRDT

Two of the most frequently used techniques to enable users to work collaboratively are
Operational Transformations (OT) and Conflict-free Replicated Data Types (CRDT), Nicolaesecu et al., 2016. OT is a technique that transforms the index of operations to take into account the effects of concurrent operations and assures replica convergence. OT is widely used for rich-text editors such as Google Docs and Etherpad, Ahmed-Nacer et al., 2014. OT can be both centralized (i.e. running on a server) and decentralized, Ahmed-Nacer et al., 2011. In OT operations need to be transformed to repair inconsistencies, Santosh and Khunteta, 2010. In order too repair those inconsistencies OT uses a history buffer and/or an operation vector, Ahmed-Nacer, 2011.

The use of CRDT is more recent than OT and while CRDT removes some of the drawbacks of OT, it also introduces new challenges. Being relatively new, CRDT supports currently only two canonical (or: atomic) data types: plain text and arbitrary tree structures such JSON and XML, but only in a rudimentary way. Like OT, CRDT seeks to reach eventual consistency over multiple coexisting instances (copies), Ahmed-Nacer, M. (2011). Operations on CRDTs are concurrent and commutative, exploiting abstract data types such as lists and ordered trees. CRDT requires no history of operations, and no detection of concurrency in order to ensure consistency. CRDT has the potential to outperform OTs in terms of time as well as space complexity, Ahmed-Nacer, 2011.

2.2 OT versus CRDT

After the above course analysis, we found CRDT to offer the best prospects for our extension, because:

1. it breaks down editing instructions in highly granular activities
2. it thus offers the best options for complex operations involving blocks
3. it scales better with growing number of editing agents (human editors and automated editing streams)
4. it is intrinsically less vulnerable to delays and events that happen offline

In the sequel of this research, we will focus on CRDT. There are various variants and implementations of CRDT. We will discuss them further, below.

2.3 CRDT correctness

In earlier work on CRDT, considerable attention has gone to the definition of correctness of CRDT: for a resulting CRDT to be correct, it has to respect the CCI criteria, Weiss et al., 2008, Sun et al., 1998:

1. Causality: All operations have been ordered by a precedence relation
2. Convergence: The system has converged; all replicas are identical when the system is in idle state
3. Intention: The expected effect (the user editing intent) of an editing operation can be observed on all replicas.

For block editing, as of yet, it is unclear as to whether these CCI criteria are also sufficient, or additional criteria will have to be added. In the below analysis, using qualified reasoning, we seek to identify arguments to sharpen the research questions we will be outlining further down.

2.3.1 Causality and block editing

Causality implies order-able sequences of operations, and imposing and maintaining an order by means of a precedence relation of all operations. Like for current-state CRDT, a precedence relation will also be needed for block editing. The precedence relation must however be extended such that it also supports block operations. Important question: can we do so independent from block content?

In this paper, we might first seek to stay close to the CCI criteria and try to design a generic concept, providing a framework to which the handling all block-specific internals can be added later on, if needed; a framework that facilitates the handling ordered sequences of operations generically. We do not want to go into the details of every structure, languages and syntax, but rather take a generic approach to block editing. While examining literature, we observed that the notion of a block is still largely missing. We define a block in this paper as a container, demarcated by the block markers (begin-of-block, end-of-block), empty or filled with arbitrary text content of any structure, language or syntax such as other blocks. In any implementation, deletion of the block requires both block-markers, along with the content in between, to be removed. Blocks containing

other (nested) blocks are delimited by the outermost block markers, as indicated in the user editing intent. Blocks inside are assumed to be part of the block content. If a block contains multiple pairs of block markers, the first end-of-block marker at the same level level of nesting as the begin-of-block marker, terminates the block.

2.3.2 Convergence and block editing

Convergence is at the very essence of online collaborative editing. The notion of convergence is evident in the context of CRDT, Weihai and Sigbjørn, 2020, and for block editing this notion will not be any different. The time frame in which editing operations evolve and complete may be elongated. This is not believed to disqualify or invalidate this criterion for block editing. There is no reason either, to do anything extra with respect to convergence, for block editing.

2.3.3 Intention and block editing

Obviously, for block editing to be relevant, the user editing intent must be converted into an end result that can be achieved at reasonable cost, defined as: with a acceptable number of instructions. Again, no clue as to this aspect bearing relevance for block editing could be found in literature. We may reason, however, that working with a compound of editing operations in a block, with possible delays, may make the assurance of user editing intent to be correctly reflected in the end result, at best tedious. In all cases, user intention must comply to the schema of the embedding context, i.e., structure, language and syntax. In case of a violation, users must be able to derive what has gone wrong, and how to correct from the current state.

2.3.4 Conclusion criteria for block editing

CRDT employs the CCI criteria for correctness. Although we found no evidence whatsoever in literature, by qualified reasoning, it appears that the CCI criteria are equally apt for correctness in the case of block editing. We therefore presume we may rely on the CCI criteria, until our research reveals this turns out to be a wrong.

2.4 Commutative and convergent CRDT

Implementations of CRDT have two distinct approaches: commutative Cai et al., 2022, and convergent, Shapiro et al., 2011. Commutative CRDT are primarily operation-based; they maintain information about the start, progress, and completion of operations. The ordering of operations is preserved. Convergent CRDT are state-based; they are defining and keep track of the CRDT state. More about this below.

2.4.1 Commutative CRDT

Commutative CRDT are basically operation-based, meaning that they copy their local operations to other CRDT, so other CRDT can perform the same operation, to synchronize. The forwarded editing changes may be received by targeted CRDT in a different order than sent, as a result of routing and transmission characteristics. Consequently, one challenge to overcome when using commutative CRDTs, is to handle operations that may have arrive in a different order than sent, Baquero et al., 2017. Solving this problem can be obtained by using a clock and timestamps. Currently, different such clock implementations are known Demibiras et al., 2017. The advantage of commutative CRDT is its simplicity of implementation and its low networking work load.

2.4.2 Convergent CRDT

Convergent CRDT are state-based, meaning that they send their whole state vector with every operation they communicate to other CRDT. Peer CRDT receive this state and update their own state to mirror exactly the received state. The most important part of the convergent CRDT is the merge method, which essentially takes two corresponding replicas of the same logical entity, resolving any conflicts, and produce an updated state as an output, Sypytkowski., 2017. The merge method must conform to three properties:

1. The commutative property: when state vectors are merged, no matter whichstate is taken as the source and which as the target the result must converge to a similar state and object
2. The associative property: the final impact of an array of to-be-merged statevectors must be the same irrespective of the order of arrival or processing
3. Idempotent property: the result must be exactly the same object (i.e, the sameblock or blocks in our case), even if operations are carried multiple times on the same object. The idempotent property states that it is not needed to care about potential duplicates

2.4.3 Last-Writer-Wins-Element-Set

Some CRDT concepts we encountered in literature, solve a specific problem or focus on a specific aspect. The Last-Writer-Wins-Element-Set CRDT LWW for short is based on the use of timestamps. All operations and elements in the CRDT are assigned a timestamp, and all operations will be executed in the order of these timestamps, Shapiro et al., 2011 Whenever a conflict arises, the operation with the most recent timestamp is the one that will be executed. This takes out the problem of operations instructions arriving for synchronization of duplicates in another order than sent. Time stamping instructions is a design aspect we will address later on.

2.4.4 Sequence CRDTs

Sequence CRDT are sequences, arrays or ordered sets of CRDT objects. These CRDT can be used to build a collaborative text-editor. Literature reports on different such implementations. The general idea is mostly the same. Each element gets a unique position in the document and this position is saved somewhere. When adding an element, it gains a unique position and is added to the data structure in which the positions are saved. This approach has been introduced earlier in OT environments, such as Google Docs. In a sense, it copies the technique of dynamically embedding intelligent objects in a text. When deleting the object, it is either a delete-andreorganization of the data structure, or the position is kept in the data structure and the element is marked as a tombstone, i.e. deleted and/or make inactive or invisible to keep the structure intact. Updating an element is not supported by all sequence CRDT, most implementations only support deletion and insertion of atomic text elements, Ahmed-Nacer et al., 2014.

Sequence CRDT algorithms share the convergence property. The difference among implementations of this sort is mostly in flexibility; some versions only support deletion and insertion while others also support the update operations of embedded elements. There is also a difference in time-complexity for these algorithms. The same holds true for space-complexity, Ahmed-Nacer et al., 2014, Briot et al., 2016.

2.5 Evaluating CRDT for block editing

Kleppmann and Beresford described using CRDT for concurrent editing in JSON, which uses structured blocks and a syntax for encoding and could therefore be useful for our research, Klepeman and Beresford,. 2017. The problems solved in the paper by Kleppmann and Beresford are however focusing mostly on conflicting insertions and deletions in a JSON object. Kleppmann and Beresford do not consider moving,

splitting, and merging blocks of text, involving a sending and a receiving context, we believe are essential operations for our research. For the case of XML-encoded texts, conflict resolution is obviously dependent on the structured language definition, like XML, XSL Stylesheets, (X)HTML, etc. To move an XML block from a sending environment to a receiving block requires an XML check in the receiving environment, to keep the XML tree sane. Both any preparatory actions in the receiving context, as well as getting the sending context in the state of the point of sending, requires proper control over precedence relations. Yet, causality must be strong enough to transparently materialize the user editing intent. We will defer this problem to our design of the extended concept in section 4.

Even more simple use cases cannot be neglected: one of the reasons that a standard CRDT would not work immediately for block editing without modification of most of the concepts, is because in block editing, blocks can be embedded in a sequence in a larger context. If three blocks are nested that way, deleting the middle block requires the third block to be moved and placed in the vacant second position in the CRDT, so that it is directly following the first block. Otherwise, in most CRDT concepts we found in literature, the bottom block would typically be seen as a part of the middle block and also be deleted. If not processed as a single operation, it might render CRDT copies incorrect. The use of our block definition with block markers (see above) may already improve opportunities to resolve this problem, but is in itself not sufficient. The precedence relation also has to be updated accordingly.

2.6 Research questions

The main research question that has been specified in section 1:

Can we extend (an existing)on- and offline collaborative editing concept to support block-based insert,delete, move, split and merge operations?

can now be broken down into:

1. How to ultimately define a block in the context of the proposed extended CRDTconcept?
2. How to deal with block structure, language and syntax generically?
3. How to handle anticipate and generically handle conflict of embedding contextand block?
4. How to use preserve and exploit the precedence relation in our extended concept?

Testing and validation will focus on assuring the CCI criteria are upheld and sufficient.

3 Design block-based editing concept

3.1 From analysis to design

After this elaborate analysis, we need to outline how to proceed with the design of the extended concept. Knowing the research questions, we can now select the best-fitting concept from literature to depart from. To this end, we will explore repositories on GitHub and Gitlab. Repositories will be chosen based on extend-ability, flexibility, and the degree of needed operations it already contains. More in particular:

- Does the concept present the present state we found in literature?
- Is its code base in the repository well documented?
- Can the concept and code base (and its tool suite) potentially be extended to the projected extended concept we describe in this research?

Overlooking, evaluating and weighing the detailed research questions formulated, we decided to use commutative CRDT, offering easier updates of the current state, and allowing easier addition of different operations. By carefully inspecting the GitHub repositories, the Logoot implementation by T-Mullen (Github(https://github.com/tmullen/logoot-crdt) has been selected, as the best candidate. This Logoot

implementation is completely based on the Logoot paper by Weiss et al, 2009. In section 3.2.1 the workings of this logoot implementation will be explained in more detail.

We evaluated the way Mullen has implemented the CRDT for inserting and deleting characters. It is our prediction that most of the extensions needed for block editing could be implemented by merely expanding the code base. This will serve as the concept prototype implementation to depart from. Its correctness will be verified and validated. To create a research prototype for the extended concept, we design and implement the above extensions in an extended research prototype.

3.2 Design block-based editing concept

3.2.1 Basic Logoot implementation

The Mullen Logoot uses a tree structure to support the insertion and deletion of text. Initializing a Logoot instance, creates a basic tree consisting of one root node with two child nodes, which serves as a begin and end node of the document. The child nodes of the starting root node always have the Id 0 and base which define the lower and upperbound of the ids. The value of base should be decided based on the use case Nedelec et al, 2013. A high base for example is not very efficient if there are not many insertions. For the examples in this paper, we use a base of 256. An example of such tree is shown in figure 1. Every node has a position, which consists of a sequence of identifiers. In this sequence, the last item is the identifier of this node, whilst all other identifiers specify the path to follow from the root to get to this node. An identifier consists of an Id, a site, and a clock. The Id is an integer that is mainly used for determining the position. The site is a string used to identify which user performed the operation. The clock is an integer which represents the count of operations performed by that particular user. All three components are used for determining the placement for the node. So when the Ids are the same, the site is the next value to compare with. Similarly, when the sites are identical, the clock will be compared. Since the clock is the operation count, if all three are the same, it means that it is the exact same operation. However, to simplify this in this report, when an Id is mentioned, the Id of the identifier of that particular node is meant.

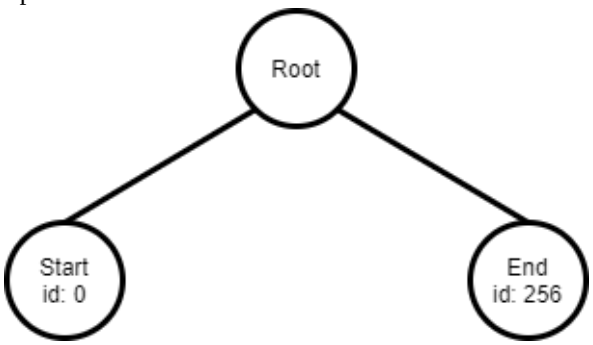

Fig. 1 Basic tree for the Mullen Logoot

3.2.2 Insert and delete nodes

Every insert operation must have an index, such that the CRDT knows where to insert the value. The index will be used to generate a node in the correct position in the tree by making use of the nodes which will become neighbors of the to-be-inserted value. The neighbor nodes in the tree are the current node at the given index and the node at the given index+1. The index will be limited to the amount of child nodes in the tree - 2, to ensure that this operation is bounded correctly. For consistent and efficient Id generation between the two

neighbor nodes, we use the LSEQ algorithm, Nedelec et al., 2013, which was already implemented in the Mullen Logoot.

Algorithm 1: Insertion of value on given index

Result: Updated tree
Parameter : value
Parameter : index
Parameter : *depthFirstSearch*(*start*, *index*) return node at given *index* in depth-first manner starting from node *start*
root ← root node of the tree;
left_neighbor ← *depthFirstSearch*(*root*, *index*);
right_neighbor ← *depthFirstSearch*(*root*, *index* + 1);
position ← *LSEQIdGenerator*(*left_neighbor*, *right_neighbor*);
node ← Insert node at *position* and assign *value* to it;

When a node is inserted in the CRDT from one client, it sends out a message containing the position and the content of the node to all remote CRDT instances. The receiving CRDT instances receive this message and build on the exact same position a node with the same content, if there exists no node on that path already. When there is already a node on the exact position, the operation will be aborted. We can safely do so, since the Id, site and clock all must be equal to be on the exact position. Since the clock can never be the same for two different operations, we can safely say that the operation has already been received and processed.

Algorithm 2: Receiving an insertion operation from remote CRDT

Result: Updated tree
Parameter : value
Parameter : position
if *position* already exists then
  Return
end
node ← Insert node at *position* and assign *value* to it;

Similar to the insert operation, the delete operation must have an index. The index will lead to a node in depth-first manner, which should be removed from the tree. However, when this node contains (non-empty) child nodes, we will only set the empty flag of this node to true, which will convert this node to a tombstone. When the node does not have children, it can directly be deleted from the tree. The CRDT will also recursively go over the parent nodes of this node to check if these can be removed from the tree as well. We can remove this parent if the parent is a tombstone and has no (non-empty) children. This process is called trimming. Just like the insert operation, a message is sent out containing the position of the deleted node, such that the receiving CRDT can delete the exact same node from the tree.

Algorithm 3: Deletion of node on given index

Result: Updated tree
Parameter : index
root ← root node of the tree;
node ← *depthFirstSearch*(*root*, *index*);
Set empty attribute of *node* to true;
Trim the tree starting from *node*;

Algorithm 4: Receiving a delete operation from remote CRDT

Result: Updated tree
Parameter : position
root ← root node of the tree;
node ← *getNodeFromPosition*(*root*, *position*);
Set empty attribute of *node* to true Trim the tree starting from *node*

Both insert and the delete operations will send messages to other CRDT with their respective path of the inserted or deleted node. This implies that all the CRDT will contain the exact same tree structure with the same content since all nodes will be inserted or deleted on the same path. Because all CRDT contain the same tree structure, convergence is guaranteed.

3.3 Design objectives

For the extended concept based on the Mullen logoot, we now identify the following design goals:

1. Add inserting and deleting atomic text elements
2. Implement the notion of a block and extend the CRDT to insert and delete blocks
3. Implement moving blocks to a given position in (the structure, syntax of) thedocument
4. Extend the CRDT to perform a split of a block into two valid blocks
5. Extend the CRDT to merge two blocks into one block
6. Extend the CRDT to converge with block editing operations in arbitrary order,such its precedence order can be reconstructed

The first design objective is already answered with this current CRDT implementation, as explained above. The last design goal is currently only answered for the insert and delete operation, but other operations are still missing.

3.4 Design and implementation of the block support

To add block support (design objective 2) to this CRDT, the current Logoot tree will be altered to support block operations. The main idea behind this new design is that we will have one main Logoot instance and recursively use Logoot instances within this main Logoot instance to support blocks, as also discussed by Ahmed-Nacer, 2011. This introduces the idea of 'block nodes', which differentiates itself from the known nodes for atomic text elements. Every block node will contain its own Logoot instance in the CRDT, resulting in nested tree structures. The block nodes are inserted in exactly the same fashion as the basic Logoot instance for characters (atomic text elements). With this approach, if we want to insert atomic text in a block, we can simply search for the corresponding block node and perform exactly the same insert operation described earlier for atomic text. If we manage to extend the concept this way, we will have reached design objective 2.

Each block node is identified with a blockId and contains a Logoot instance. The blockId is independent of the main Logoot and is a unique randomized string, allowing the main Logoot to search for the node based on blockId. Introducing this notion, all atomic text operations should include a blockId along with the operation, specifying the block on which the operation should be performed on. This allows the CRDT to delegate the operation to the corresponding block even if the block gets moved, merged or split. Inserting block nodes (insertBlock) can be done in exactly the same manner as inserting atomic text. The only difference is that we will create a block node instead of a node containing atomic text. For deleting block nodes (deleteBlock), we can also use the same approach. The only difference is that we will search for the block node based on blockId instead of index.

3.4.1 Block operations

As for the block operations (block-move, block-merge, and block-split; design objectives 3-5), new approaches had to be thought of in order to guarantee convergence, since CRDT should also support operations which are executed offline. The latter requirement drastically increases the complexity of the code, since the order of the operations can differ per CRDT replica. Every operation should be able to converge when the order of execution is different.

CRDTs converging when operations are executed offline are difficult to handle. For example, insert operations are performed before someone splits a block offline, so the order of operations is flipped for this user. This means that at one replica the insert operations are split correctly, while the other replica inserts the insertion in the wrong block (fig. 2). Our designs for the block-move, block-merge and block-split operation should take such scenarios into account.

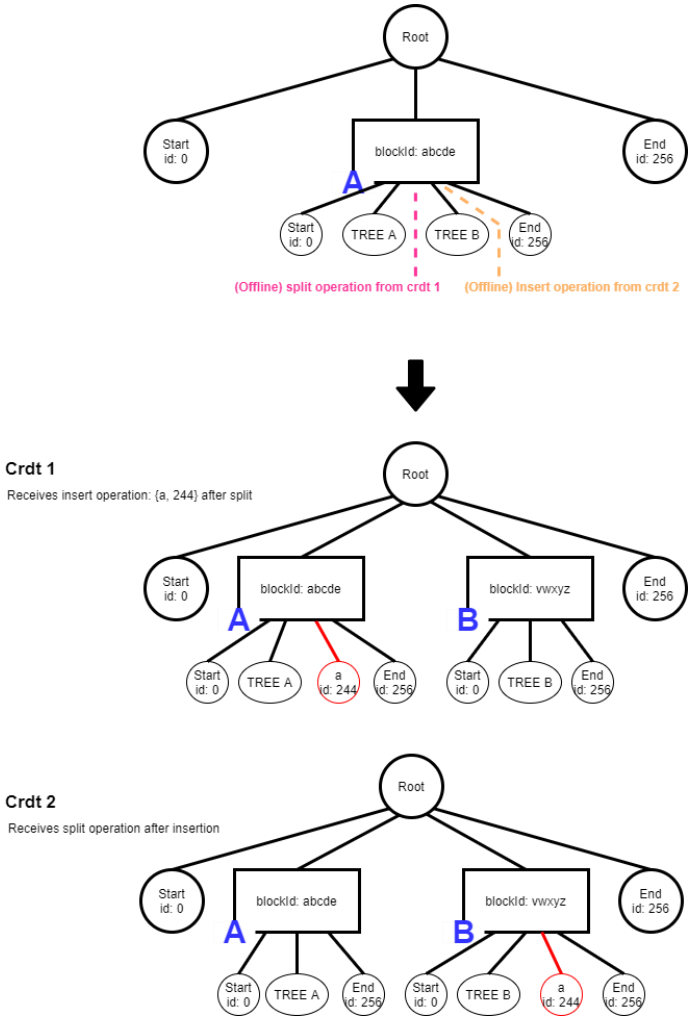

Fig. 2 Two CRDT diverge when both CRDT are working offline using the split and insert operation

3.4.2 Move block

The suggested approach for block-move (design objective 3) is as follows. Moving block A requires an index, which is used for determining the new position. When the move operation is called, we will simply insert a new block B at the index we want to move the block to. Then we will transfer the Logoot of block A to block B and eventually remove block A. Block B will be assigned the same blockId as the deleted block A, to ensure that all other to-be-performed (offline) operations will be performed in block B. This remedies the lack of convergence, without the need to puzzle the precedence relation (fig. 3).

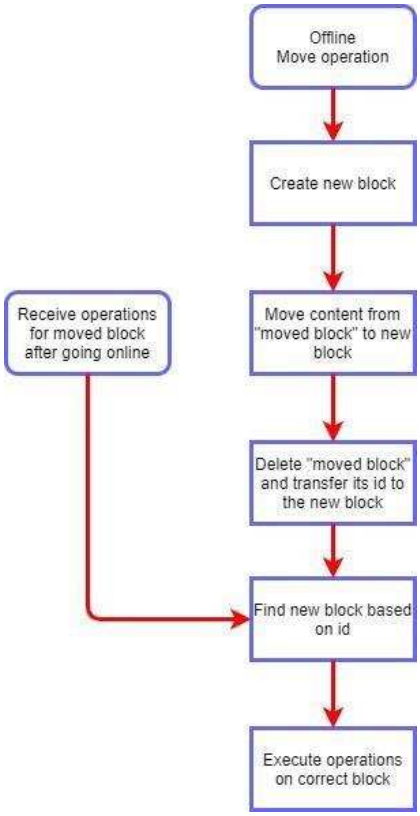

Fig. 3 Precedence relation scheme for offline block-move with insert operations.

Algorithm 5: Moving a block to given index

Result: Updated tree
Parameter : blockId
Parameter : index
root ← root node of the tree;
block ← *searchBlock*(*root*, *blockId*)
newBlock ← *insertBlock*(*root*, *index*, *blockId*)
transfer *block.logoot* to *newBlock.logoot*
deleteBlock(*block*)

Algorithm 6: Receiving a block-move operation from remote CRDT

Result: Updated tree
Parameter : blockId
Parameter : position
root ← root node of the tree;
block ← *searchBlock*(*root*, *blockId*)
newBlock ← *insertBlock*(*root*, *position*, *blockId*)
transfer *block.logoot* to *newBlock.logoot*
deleteBlock(*block*)

3.4.3 Split block

The approach for splitting a block (design objective 4) is as follows. Splitting a block requires an index, such that the CRDT can determine where to split the block. For this, we will introduce the 'SplitNode', which will represent a virtual split in the block. When block A is split, block A will insert a 'SplitNode' according to the index, just like the normal insertion. A new block B, with a new blockId, will be created right next to block A and copies the exact same tree structure of block A into block B. For block A, the blockId will stay the same and all nodes after the SplitNode will be removed. For block B, all nodes before and including the SplitNode will be removed (fig. 4). When offline operations happen at the same time as the split operation, it might happen to be an operation which should be performed on a index greater than the SplitNode in block A. In these cases, the operation should be delegated to block B. Since every operation is sent individually, it can easily be checked whether it should be delegated since the left neighbor of the to-be-inserted node should be the SplitNode in these cases. Delegating the operation to block B would still converge, since the tree structure of block A and block B are exactly the same. Therefore, the position that is emitted through the message is still valid for block B.

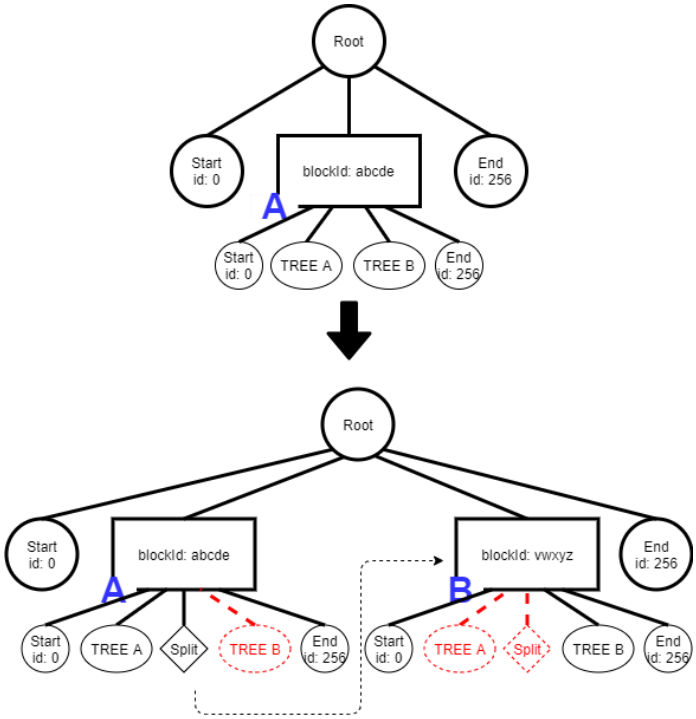

Fig. 4 Splitting block A using references

Algorithm 7: Splitting a block on the given index

Result: Updated tree
Parameter : blockId
Parameter : index
root ← root node of the tree;
block, *blockIndex* ← *searchBlock*(*root*, *blockId*)
splitNode, *splitIndex* ← *insertSplitNode*(*block.root*, *index*)
newBlock ← *insertBlock*(*root*, *blockIndex*)
copy *block.logoot* to *newBlock.logoot*
Remove all nodes from index [0, *splitIndex*] from *block*
Remove all nodes from from index [*splitIndex* + 1, end] from *newBlock*

Algorithm 8: Receiving a block-split operation from remote CRDT

Result: Updated tree
Parameter : blockId
Parameter : newBlockId
Parameter : newBlockPosition
Parameter : splitPosition
root ← root node of the tree;
block, *blockIndex* ← *searchBlock*(*root*, *blockId*)
splitNode, *splitIndex* ← *insertSplitNode*(*block.root*, *splitPosition*)
newBlock ← *insertBlock*(*root*, *newBlockPosition*, *newBlockId*)
copy *block.logoot* to *newBlock.logoot*
Remove all nodes from index [0, *splitIndex*] from *block*
Remove all nodes from from index [*splitIndex* + 1, end] from *newBlock*

3.4.4 Merge blocks

For the block-merge operation (design objective 5), the approach changes the end node of block A into a merge node. So when block A and block B are merged, block A will change its end node to a merge node, which contains a reference to block B. This guarantees the state of the block where the merge node will be put at the end of the block (fig. 5). Since in this approach, block B stays intact, operations executed on block B from another user will still converge, even after a merge (fig. 6). Merging a block which already has a merge node, would result in delegation of the block-merge operation to the referenced block specified in the merge node. As can be imagined, different sequences or (offline) merge operations can lead to divergence of the states. To ensure that all block-merge operations will result in consistent states on the receiving end, we will give all merge operation a timestamp upon inserting. When receiving a merge operation, we will remove all merge nodes from the two block nodes which should be merged. After that, we will re-insert the merge nodes including the newly added merge node in chronological order. If there is a conflict (the CRDT wants to insert a merge node in a block, but it already exists in the block), we delegate the insertMergeNode operation to the reference node in the found merge node. This ensures that all replicas have the same order of merge operations.

Algorithm 9: Merging two blocks on their given blockIds

Result: Updated tree
Parameter : leftBlockId
Parameter : rightBlockId
$root \leftarrow$ root node of the tree;
$leftBlock \leftarrow searchBlock(root, leftBlockId)$
$rightBlock \leftarrow searchBlock(root, rightBlockId)$
$mergeNode \leftarrow insertMergeNode(leftBlock, rightBlockId)$
$rightBlock.isMerged \leftarrow true$;

Algorithm 10: Receiving a block-merge operation from remote CRDT

Result: Updated tree
Parameter : leftBlockId
Parameter : rightBlockId
Parameter : mergeReference
$root \leftarrow$ root node of the tree;
$leftBlock \leftarrow searchBlock(root, leftBlockId)$
$rightBlock \leftarrow searchBlock(root, rightBlockId)$
$mergeReferences \leftarrow$
$set\{getMergeReferences(leftBlock), getMergeReferences(rightBlock), mergeReference\}$
sort *mergeReferences* on timeStamp
for $reference \in mergeReferences$ do
  $insertMergeNode(reference.from, reference.to, reference.timestamp)$ set according nodes to *merged*
end

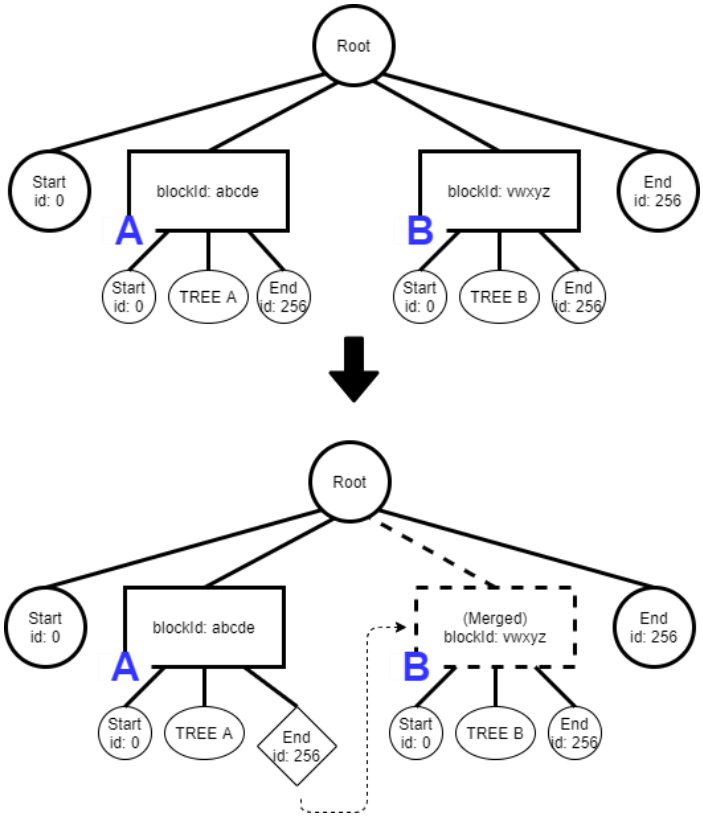

Fig. 5 A valid state of the blocks when block A and block B are merged

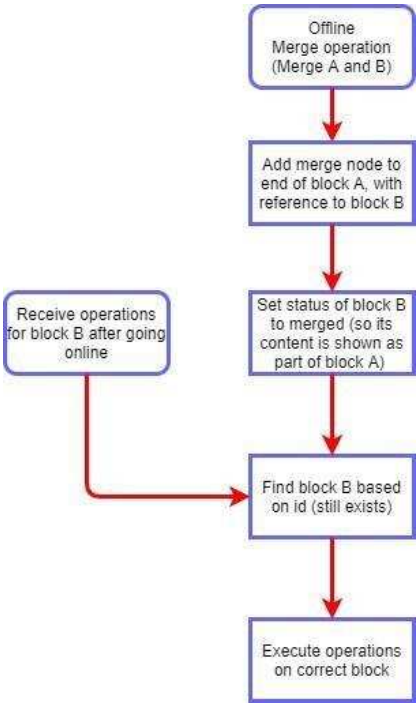

Fig. 6 Precedence relation scheme for offline block-merge with insert operations.

4 Experiments

After designing and implementing the extended CRDT, we will perform two experiments to test the correctness of our extended CRDT. The first experiment is in the form of automated tests, while the second one is in the form of a manual test.

4.1 Enumeration tool for automated tests

To run automated tests, a testing framework will be developed in which tests are generated using a script. The generation of tests will require a starting state, a set of operations, and a sequence of operations to be performed. The framework will then enumerate through all possible scenarios (exhaustive testing), starting from the starting state, with the set of operations. These scenarios are tested in an offline as well as an online setting. This will form a tree, where each child node is a result of a performed operation on the parent node. The depth of the tree is determined by the amount of steps. Since asserting all possible scenarios will be exponentially large, there is a possibility of pruning each layer of the generated tree included in the testing framework. This way, still an effective variety of test cases can be tested within a reasonable amount of time. Figure 7 will depict what this pruning process looks like for each level. The states that are executed will be tested for convergence by comparing the output of the CRDTs.

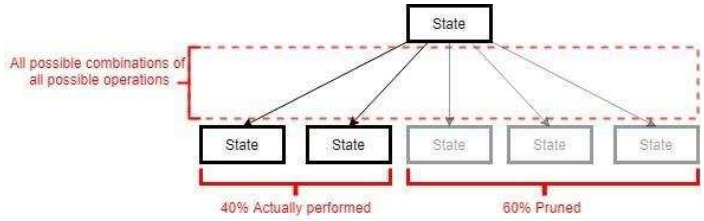

Fig. 7 A pruning percentage of 60% is uphold for each level in the tree for all possible combinations of operations

4.1.1 Automated test the Mullen Logoot

As a first step, we verify the Mullen Logoot code base to produce the correct results. To do so, we are going to use the automated testing framework described in the previous section. The initial Logoot was already tested for correctness by Mullen, but the model will once more be tested on correctness with randomized operations within our testing framework to ensure a valid basis for our extended concept. The initial Logoot only consists of insert and remove operations. The framework will start with a starting state, containing three character nodes and generates tests for all possible insert and delete operations whilst being online and offline. Since the correctness was already tested by Mullen, only a relatively small selection of possible operations will be taken into consideration for each level of the tree by using pruning. The states that are not pruned will be further explored while the pruned states are deleted from the tree. The settings that were used for the test generation is shown in table 1. Next to the automated tests, we will also write some unit tests to supplement the automated tests.

Table 1 The test suite used to test combinations of the insert and delete operations

	Text		Settings	
Test	Insert	Delete	# Steps.	Prune (%)
1	X	X	4	97

4.1.2 Automated testing of the extended CRDT

To test the extended CRDT for correctness, we are going to make use of a combination between unit tests, integration tests and generated tests using the automated testing framework. Combinatorics involved render exhaustive testing infeasible, and therefore, we consider combinations of operations having high risk to conflict with each other, such as splitting a block while moving the same block. These cases are included as our edge cases. A list of all combinations tested (set of operations, pruning percentage and amount of operations) are given in table 2. As table 2 reveals, (combinations of) operations differ in pruning percentage. The pruning percentage results from experiments done with manual testing, to reveal combinations that are particularly error prone. Merge and split with other operations are an example of this. Therefore we have many test suites with merge and/or split to search for these bugs.

Table 2 The test suites used to test combinations of operations. The pruning percentage is the percentage of test cases removed from the tree.

Test	Text in block		Block					Settings	
	Insert	Delete	Insert	Delete	Move	Split	Merge	#Steps.	Prune (%)
1					X	X		2	95
2								2	90
3		X		X				2	90
4				X			X	3	0
5			X		X		X	2	97
6	X	X					X	2	99
7	X					X	X	2	96
8						X	X	2	0
9							X	4	0
10				X	X			2	90
11	X	X			X			2	99
12					X			2	50
13		X				X		2	97
14						X		2	0
15	X	X	X	X	X	X	X	1	0

4.2 Editor design

User editing intent lacks a concise validation criterion: it is not possible to use our testing framework for that purpose. To adequately test whether the CRDT respects user editing intent, a special purpose editor will be developed to collect user experiences while manually testing user editing intent. In our block-based editor, we use begin-of-block and end-of-block identifiers to identify blocks of text. In this approach, it would be difficult to visualise merge, move and split operations. Therefore, we designed this block-based editor to contain multiple text areas, where each text area represents a block of text. Using this approach, it will be visually clear what a block is and how the document structure changes after specific operations have been applied. The merge, split and move operation will be executed using a button click instead of a hotkey.

To send messages to different online editors, WebSockets will be used. Further implementation details are deemed irrelevant for the discussion. WebSockets make it possible for this editor to show real-time collaborative editing on different devices. CRDT do not need a central server (section 2). However, for the ease of the implementation of the editor, we have created a central server architecture for our editor implementation. This central server is used to store data in memory which can be useful if we want to reproduce specific scenarios.

4.2.1 Editor features

The editor contains the designed functionality laid down in the design objectives. The operations include:

1. Inserting text in a block
2. Deleting text in a block
3. Inserting a block
4. Deleting a block
5. Splitting a block at a given index

6. Merging two blocks
7. Moving a block to a given index

In addition to these features, offline support and some extra visualisations are added to simulate situations where one or multiple users work offline. In addition, this gives users a better view of what happens with the CRDT when editing blocks.

Offline support in the editor

The first additional feature that will be added, is an offline working mode toggle. This feature allows us to simulate arbitrary networking delay. We have merely added this feature to the editor for analysis and verification purposes. Offline support does not introduce any additional functionality to the CRDT itself. In the editor, there will be a button which can be used to toggle the editor to online or offline mode. If the editor is offline, all received editing requests will be postponed until the editor goes online. Likewise, the outgoing editing requests will be buffered until the editor returns to online mode.

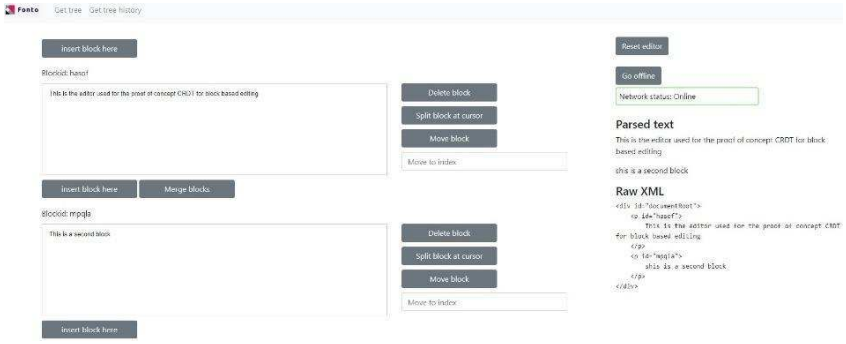

Fig. 8 The editor created as a proof of concept

4.3 Manual testing

Manual testing is mainly used for verifying the user intent. However, the editor can also be used to visualize and replay scenarios from the automated tests. Users can use all kinds of CRDT operations and inspect for undesired or unexpected behavior, and poor user editing intent. The history of operations will be kept to be able to reproduce unexpected behavior, if encountered. To decide which scenarios we are going to test manually, we are going to look at the result excerpted from the automated tests and reproduce/modify tests which failed earlier in the development process.

5 Results

This section presents the results of the tests described in the previous section. We discuss results in view of the research questions stated in section 2.

5.1 Test results

5.1.1 Automated tests

The automated tests that were generated to verify the correctness for the initial Mullen Logoot repository brought up no issues. One minor bug was found in the Mullen Logoot: a small fraction of the state is not parsed correctly, resulting in run time errors. After correcting this bug, the test suite showed convergence for all

possible combinations. We generated exhaustive cases up to four operations per CRDT. Pruning is used to verify cases with more operations per CRDT replica. The pruning rate was approx. 0.97 with only 3% being iterated further on. To test CRDT to be consistent, we ran the test suite 1000 times, none failed.

For our extended concept CRDT, various combinations of operations have been tested quasi-exhaustively, as explained. All combinations of tested operations as shown in table 2 resulted in test cases passing, except for the merge operation. This merge operation caused an infinite loop when two offline CRDTs created a circular merge. This is the first concept flaw. It will be discussed in greater detail in section 5.2.

5.1.2 Manual tests

The manual tests showed promising results in the overall performance: it showed that our extended concept is quite robust with stable convergence. Only a small fraction of specific combinations in sequential order resulted in unexpected behavior, though still converging with other CRDT. Manual analysis also informed us about a second concept flaw: content loss due to a specific way of combination of split and merge operations. Further discussion on this concept flaw follows in 5.2 too.

5.2 Discussion concept flaws

The first concept flaw is the circular merge. What happens is that two CRDT seek to merge the same set of blocks in opposite order, while being offline (and only then). So if the CRDT have block 1,2 and 3 and one CRDT merges block 1 and 2 and block 1 and 3 sequentially, while simultaneously the other CRDT merges block 3 and 2 and block 3 and 1 sequentially, the eventual CRDT will have blocks containing circular references, see figure 9 for an illustration.

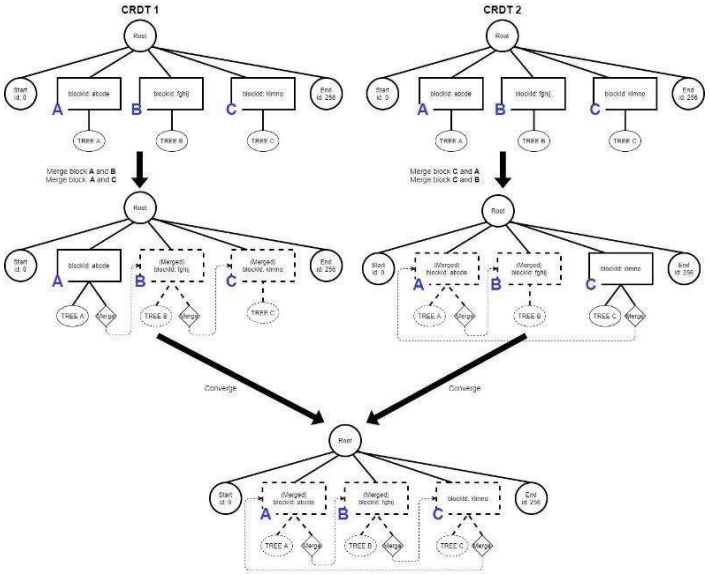

Fig. 9 A situation in which a circular merge occurs

One way to solve this, and how we would propose to work around this problem, is to assign one principal CRDT to each collaborative edit session, where the principal CRDT is the only CRDT which is allowed to execute the block-merge. But still any CRDT can issue a block-merge request. This will result in a single-point of control, where the principal CRDT needs to verify all requests for possible circular references. Another solution strategy to solve this issue is using timestamps. Each merge operation gets an assigned timestamp,

and after each merge operation, all merge operations will be reverted and reordered based on their timestamp. All merge operations will be re-executed in this revised order. Before applying all merge operations, the algorithm will check if there are circular references. It will check if for example, block 1 has a reference to block 2 and block 2 has a reference to block 1. If a circular reference is found, the timestamps will be compared and the merge operation with the highest timestamp will be ignored, thereby effectively removing the circular reference. The problem with this approach is that in our concept, it is not known whether all users who had used the merge operation have been online and exchanged their operations. When all the CRDTs have the same state, removing the same merge reference could solve the problem. We did not further go into this issue.

The second concept flaw found in the current implementation of the extended concept is that when splitting a block and merging the two resulting blocks from the split, the content belonging to the 'right-side' part of the block could be lost on all replicas. Analyzing the case revealed that the cause is in the sequence of a MergeNode after a SplitNode. Since every insertion and deletion after the SplitNode will be redirected to the reference block, the MergeNode will be inserted in the reference block, causing the content loss.

We found the following solution to solve this: adding a mechanism which would allow the CRDT to distinguish whether an operation has been executed before or after the insertion of a given SplitNode. Then, the CRDT should only redirect the operation if it was executed before the operation of the SplitNode. This would make sure that the SplitNode only does what it is designed for: redirect nodes which were inserted before the split in block x, but should be in block y after the split. Implementing such an algorithm would be complex, but feasible. When encountering a SplitNode, the receiving CRDT should be able to communicate if the sending CRDT did this operation before receiving the SplitNode or after.

With these modifications, all tests pass and adequate user editing intent is obtained for block editing.

6 Conclusion

This section will evaluate and conclude on the results shown in the previous section. Next, we review the research questions set out and discuss what the answer on the research questions are based on these results. Finally, this section will discuss what the next steps are for this CRDT to extends it functionality or improve its performance.

6.1 Higher-level interpretation of the results

We have shown that a present state CRDT can be extended straightforwardly to support block editing. Inserting atomic text elements such as characters in blocks, deleting characters in blocks, inserting blocks (creating new blocks), deleting blocks, moving whole blocks, splitting blocks, and both on- and offline support for these operations. A block in our work, is simply defined as a collection of text elements, possibly nested blocks, bounded by a begin-of-block and an end-of-block marker. A block marker can be a dedicated token of choice, including a line break, depending on the structure, language or syntax involved. This clear and explicit demarcation of a block is the only common denominator that separates a block from its embedding context. Its embedding context can be another block, so that a block can be moved into another block. The merge operation implemented needed an additional provision to avoid circular merges in offline mode. We have given a solution for this. Another case deserving further attention is the potential content loss when performing multiple split and merge operations on the same block. we also gave a solution for this. With these modifications, all tests (automated and manual) pass.

Implementing the merge functionality was the most time consuming operation to implement, because merging blocks in combination with moving and splitting blocks and offline support has a huge amount of edge cases requiring careful attention. It has been shown that if only one (principal) CRDT can merge, it will always result in convergence. Ultimately, preventing or lifting the infinite loop in the circular merge permitting

multiple CRDTs performing a merge operations at the same time, may be the ultimate solution. The potential content loss after a specific split-merge operation can be remedied too, as described in the previous section. Our implementation has been designed for research purposes in the first place. Both of the above issues were studied, but not yet well patched in our implementation, and believed to be easily tackled in a more refined and mature implementation.

As a concrete block editing application, we implemented operations on an XML block. Given the simple yet concise block definition, we had no difficulties in making the XML block editing work. Although still a bit premature, this leads us to believe that, indeed, this might be a path towards a generic and easy to adapt concept. The more so, because we narrowly followed the Mullen design to extend the concept, whenever we could. By taking a block as a container, possibly containing nested blocks, all block-internal operations can be postponed and left to confined postprocessing, provided block operations, such as creating, deleting, moving, splitting and merging a block are accommodated. This is the case in our extended concept. The user editing intent is provided for by crisp causality, control over the precedence relation in all forwarding, buffering and processing of editing instructions. We found a fitting scheme for id-assignments to blocks, based on the original Mullen implementation.

Regarding the run time efficiency (linear time) of all implemented operations, it is too early to conclude on the performance for practical purposes. Further tests with different input sizes are needed to indicate whether they are efficient enough for practical usage. Currently, no benchmarks have been performed to check for which file size or even tree size operations become inefficient. This could be a next step into further research.

6.2 Strengths and weaknesses analysis

One strength of the Mullen CRDT is how blocks have been implemented: basically each block has its own CRDT for the content in the block as a Logoot field. Each CRDT already has the basic infrastructure needed for the creation of blocks. Therefore, it should not be difficult to implement nested blocks in this CRDT. Depending on further algorithmic details, this may increase the complexity of the move, merge and split operations.

Another strength of this CRDT implementation is that new types of nodes could easily be added to the CRDT. If special nodes were to be added added in support of structure, language or syntax, these could easily be added to the current set of nodes, rendering this CRDT implementation easily extendable.

Finally, this CRDT implementation uses JSON objects to send messages to other replicas. This means that there is a lot of freedom to decide how to implement the network layer. This could be done the same way it is implemented in the editor using WebSockets, or any other way of sending strings over the internet.

One of the drawbacks the Mullen concept and its base code have, is that each character is a node by itself and therefore the tree size can become extremely large when editing large documents. Large trees will most likely result in slower insertions and deletions of characters. For practical usage, it is important, that the tree size and its implementation will be optimised.

Another weakness, also in the field of tree size, is that deleting nodes might create many tombstones. A tombstone will be created if the node the user is trying to delete has child nodes or if a user tries to delete a block. For small documents this does not matter, but once users start deleting lots of text, an abundant amount of tombstones may be spawn. This might result in unnecessary slower insertions and deletions of text, since the CRDT might have to traverse all the tombstones to find a node.

A further aspect of the Mullen approach that may take rethinking, is that the searchBlock method uses a BFS algorithm. This is not the most efficient algorithm to find a block as quick as possible and when the amount of blocks in the document increases, this algorithm will also get slower.

We discussed the merging of blocks in the previous sections, and we outlined a method to keep away from deadlock situations. When merging a large amount of blocks in one block, the complexity of this approach

might increase considerably, as every previous merge operation related with the blocks will be removed and inserted again after every merge operation.

In the split block approach, another recommendation for improvement is that for every split operation, the whole block is copied to the newly inserted block. When splitting extremely large amounts of content, every node has to be created again and this might take some time if the size of the block is large. This is unwanted behavior, since with real-time collaborative editing, users expect operations to be visible instantly.

6.3 Achievements and conclusions

The main research question we posed in the beginning of this paper addressed the possibility to extend an existing state-of-the-art CRDT concept to support block editing. We have designed a concept realizing this. Our achievement shows that the research question can be answered as follows. We found a fitting implementation due to Mullen, that served as a fitting departure for our research. It could be proven to comply to the CCI-criteria generally accepted for CRDT; our null hypothesis. We achieved to extend this concept and make it support block-editing operations in a way so that the extended concept again showed to comply to the CCI criteria (extended hypothesis). We thereby;

- gave an elementary yet appropriate definition for a block (research question1). Literature did not provide such a definition. Its simplicity reflects its ease of implementation and its further options for concise containment of block-internal aspects (research question 3)
- managed to extend the Mullen identification mechanism to make it support blockids (research question 2), so handling and processing it could also fit in the existing processing and handling. This is important to achieve; contained-yeteasy-to-extend block editing operations, completely independent from the block structure, language, and syntax (research question 3 and 4). We demonstrated this by using XML-blocks in our proofing. For full fledged implementation of XML, there will be no need to dig deep down into the handling of block; a overarching XML-tree parsing can be implemented to keep the XML tree of the whole document sane
- showed that the precedence relation can be maintained and preserved so thatin all cases (research question 5), we obtain (eventual) convergence in a stable manner, provided we implement provisions to avoid circular reference while merging block. Causality is strong and user editing intent is intuitive and stable (correctness criteria). Nothing has come up suggesting that the CCI criteria are insufficient for block editing. We managed to embed block editing within the Mullen concept in such a way that the CCI criteria were in jeopardy, or needed to be augmented somehow

We claim that the flawed aspects we signalled, can either be removed in a more robust production design and implementation with the pointer we gave. Based on these achievements, our conclusion is therefore that we have shown that the current state of the art CRDT support block editing in a way compliant to the CCI criteria, in onand offline situations, for any amount of CRDT, without central server orchestration.

6.4 Pointers for further research

In this section we intend to give further recommendations to further improve the concept, based on our experiences. The first optimisation is regarding the tree size. As mentioned before, the tree size could extremely grow once the document size improves. The tree size could drastically be reduced by storing a string in a node instead of a separate character in each node. However, when a user wants to insert in that string, the string still needs to be split in two nodes. Research has already been done about this in the following paper, Yu, 2012.

Another way to reduce the tree size, is to perform cleanups once no user is editing the document. Such clean up could function as a garbage collector and could remove tombstones from the tree, remove split

references and remove all merge references from the CRDT by adding them to the block the refer to. Cleaning up the merge references will also mitigates the previous described weakness regarding merges, since after every clean up, all merged blocks will be reduced to one block with no merge references. Cleanups will also optimise the split operation, since tree size will be reduced, less content has to be copied to the new block.

To overcome the circular references in the merge problem, assigning additional governance to a principal CRDT (master copy) can remove the problem, as we have shown. It may also be removed by some additional agent, orchestrating the references during operations; a block may provisionally be represented by a projected block while offline, being executed while online again. In the latter option, one has to sacrifice user editing intent to overcome the merge-block issues, which may not be desirable. Another path to improvement is to immediately remove the edge which has the highest timestamp in the circle. However, this approach is rather ineffective for solving more than one circular reference, since multiple CRDTs of multiple replicas can form different cycles, which will then converge to different circular references. An approach which might work is to save all merge references and construct a directed graph which contains all the references. The only problem left to solve, is to commonly find the same path with all CRDTs, including all the connected nodes (if possible). Since each replica contains the same references, if a common path can be found, then all replicas would merge the same blocks and therefore the replicas would converge.

To optimise the searchBlock algorithm, a different algorithm could be used to find the blocks. Instead of using BFS to find the block, a CRDT could keep track of a map containing the blockId and the path to the block or the block itself. This would optimise the searchBlock algorithm to O(1). To make this algorithm work however, one should ensure that the map will be updated correctly after every block operation.

Finally, regarding a combination with the merge and split operation: as described in the previous section, the current issue with the merge and split operation, is that some merge nodes may be redirected after being inserted due to a split node. This could incur undesired behavior in some cases, and sought behavior in other cases. When a block is already split, the merge with that block would be redirected to the new block created by the split operation. This is undesired since the block was already split and the expected behaviour would be to merge that block. To solve this problem however, some distinction should be made as to whether something was done before the split operation or after. When this distinction can be made, this problem can be solved.

We constrained our research in that we neglected undo/redo operations. Now that we managed to minimized the intermingling of basic whole-block operations with blockinternal operations, a next step could be to investigate the possibility of doing the same for undo/redo operations: splitting whole-block undo/redo from block-internal undo/redo. Another limitation that our research had, like other research we consulted in literature, is that user editing intent is a weak notion to assess. Capturing user editing intent, formalizing it in a concept, and make it measurable, would be a great step forward. Finally, the issue of editing nested blocks within the CRDT deserves further attention. So far, we relied on the outer block markers to define what a block is. As explained, a block can be nested within another block, and a block can be moved into the embedding context of another block. For generic block editing, this suffices. Users can select the block they want to move, split, etc. within their editor. That is the block under concern. Perhaps, using a recursive approach, more elegant operations can be defined for nested blocks. This could again be a useful topic for further research.

7 Outlook

The extended concept as presented can be adapted for production purposes relatively easily. For YAML, JSON, XML, source code, and many other purposes, it can offer new collaborative editing scenarios. Further tailored implementations for blockinternals can provide further block editing comfort and extensions. Environments in which blocks are not yet exploited to the full extent or not employed at all, can now consider block operations. Asynchronous block-wise broadcasting, multi-casting, and other forms of content distribution or content co-creation may benefit from it.

Once our concept will be robust and resilient enough for use in multi-agent environments, networks of co-editing CRDT, fueled by artificial intelligence, are believed to be able to draft documents that may just need some supervision and a final human touch to finalize. This not only relieves humans from manual character-by-character editing, it also discloses much wider arenas of data and knowledge.

Acknowledgements

Authors would like to thank Fonto for making this research possible, and their valuable guidance and feedback.

Conflict of interest

Three of the authors team are working for Fonto company (fontoxml.com) . Their role in this research has been primarily to provide advice, guidance, and feedback on the results and assess its practical value for their activities.

Authors have no relevant financial or non-financial interests to disclose. Authors have no conflicts of interest to declare that are relevant to the content of this article. The authors have no financial or proprietary interests in any material discussed in this article.

References

R. Khare and A. Rifkin. Xml: a door to automated web applications. IEEE Internet Computing, 1(4):78–87, 1997.

Gérald Oster, Hala Skaf-Molli, Pascal Molli, and Hala Naja-Jazzar. Supporting Collaborative Writing of XML Documents. Research report, 2006.

Claudia-Lavinia Ignat, Luc André, and Gérald Oster. Enhancing rich content wikis with real-time collaboration. Concurrency and Computation: Practice and Experience, March 2017.

Stéphane Martin, Pascal Urso, and Stéphane Weiss. Scalable XML Collaborative Editing with Undo. Research Report RR-7362, INRIA, August 2010.

Stephen J Davis and Ian S Burnett. Collaborative editing using an xml protocol. In TENCON 2005 - 2005 IEEE Region 10 Conference, pages 1–5, 2005.

Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In Proceedings of the 19th International Conference on Supporting Group Work, GROUP '16, page 39–49, New York, NY, USA, 2016. Association for Computing Machinery.

Mehdi Ahmed-Nacer, Pascal Urso, Valter Balegas, and Nuno Preguiça. Merging OT and CRDT Algorithms. In 1st Workshop on Principles and Practice of Eventual Consistency (PaPEC), Amsterdam, Netherlands, April 2014.

Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In ACM, editor, 11th ACM Symposium on Document Engineering, pages 103–112, Mountain View, California, United States, September 2011.

Kumawat Santosh and Ajay Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements. International Journal of Computer Applications, 3, 07 2010.

Mehdi Ahmed-Nacer. Abstract unordered and ordered trees crdt, December 2011.

Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: a P2P collaborative editing system. Research Report RR-6713, INRIA, 2008.

Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact., 5(1):63–108, March 1998. CRDT for block editing: concept and implementation 31

Weihai Yu and Sigbjørn Rostad. 2020. A low-cost set CRDT based on causal lengths. In Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20). Association for Computing Machinery, New York, NY, USA, Article 5, 1–6. https://doi.org/10.1145/3380787.3393678

Cai, W., He, F. and Lv, X. Multi-core accelerated CRDT for large-scale and dynamic collaboration. J Supercomput 78, 10799–10828 (2022). https://doi.org/10.1007/s11227-022-04308-7

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, Inria – Centre Paris-Rocquencourt ; INRIA, August 2011.

Baquero, Carlos, Almeida, Paulo Sergio, and Ali. Pure operation-based replicated data types, Oct 2017.

Murat Demirbas, Marcelo Leone, Bharadwaj Avva, Deepak Madeppa, and Sandeep S. Kulkarni. Logical physical clocks and consistent snapshots in globally distributed databases, 2014.

Bartosz Sypytkowski. An introduction to state-based crdts, Dec 2017.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. IEEE Transactions on Parallel and Distributed Systems, 28(10):50, Jan 2011.

Loïck Briot, Pascal Urso, and Marc Shapiro. High Responsiveness for Group Editing CRDTs. In ACM International Conference on Supporting Group Work, Sanibel Island, FL, United States, November 2016.

M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. IEEE Transactions on Parallel and Distributed Systems, 28(10):2733–2746, 2017.

Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems, ICDCS '09, page 404–412, USA, 2009. IEEE Computer Society.

Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In 13th ACM Symposium on Document Engineering (DocEng), pages 37–46, Florence, Italy, September 2013. 10 pages.

Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq: An adaptive structure for sequences in distributed collaborative editing. In Proceedings of the 2013 ACM Symposium on Document Engineering, DocEng '13, page 37–46, New York, NY, USA, 2013. Association for Computing Machinery.

Weihai Yu. A string-wise crdt for group editing. In Proceedings of the 17th ACM International Conference on Supporting Group Work, GROUP '12, page 141–144, New York, NY, USA, 2012. Association for Computing Machinery.

Appendix. A Legend

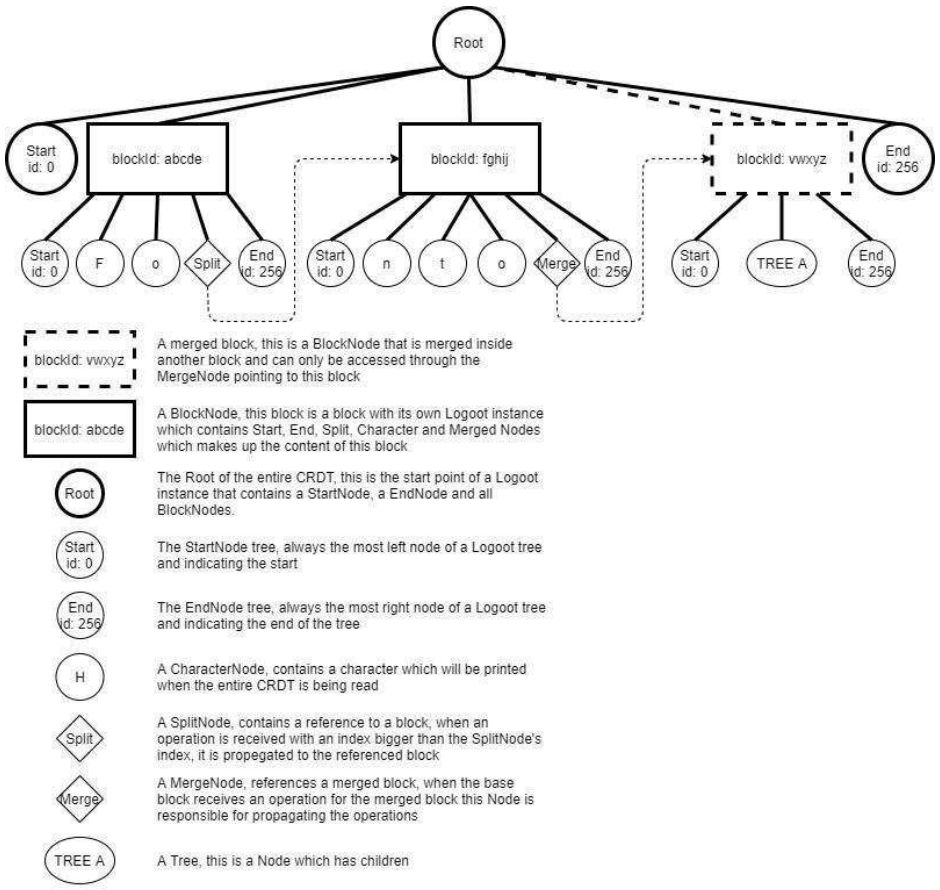

Fig. 10 Legend