Concurrency Control and Awareness Support for Multi-synchronous Collaborative Editing

Mehdi Ahmed-Nacer, Pascal Urso
Université de Lorraine - INRIA, LORIA
Email: mehdi.ahmed-nacer@loria.fr,
pascal.urso@loria.fr
LORIA Campus scientifique, Nancy. France

Valter Balegas, Nuno Preguiça
CITI/FCT-Universidade Nova de Lisboa
Email: balegas@gmail.com,
nuno.preguica@di.fct.unl.pt
MONTE DA CAPARICA, PORTUGAL

***Abstract*—Collaborative editing tools have become increasingly popular in the last decade, with some systems being used by massive numbers of users. While traditionally collaborative editing systems would either target synchronous or asynchronous collaboration settings, some recent systems support both types of collaboration, even supporting disconnected work. In this paper we analyze the limitations of existing systems and propose a data management solution that overcomes such limitations. The proposed concurrency control algorithm, based on conflict-free data types, builds on the ideas previously developed for synchronous collaboration, extending them to support asynchronous collaboration. Our solution also includes the necessary information for providing comprehensive awareness information to users. The evaluation of our algorithm shows that comparing our solution with traditional solutions in collaborative editing, the conflict resolution strategy proposed in this paper leads to results closer to the ones expected by users.**

Keywords*—*Collaborative editing, multi-synchronous applications, move/update operations, concurrency control, CRDT, awareness information.

I. Introduction

In recent years a large numbers of massively used collaborative editing applications have been developed. These applications can be broadly classified as supporting asynchronous collaboration, such as wikis and version control systems; or synchronous collaboration, such as distributed real-time collaborative editors, collaborative design tools or cloud integrated development environment. Moreover, everyday mass market applications, such as note-taking tools or cloud drives – notably Google Drive [1] and Microsoft SkyDrive [2] – aim to provide collaboration and synchronization between multiple devices and to support offline work on mobile devices. At the heart of these systems, we can find the algorithms for concurrency control developed in the last decades by the groupware community, or adaptations of these algorithms.

In asynchronous collaborative applications, users modify shared documents independently without immediately observing changes being performed by other users. These applications usually use some kind of floor control [3] that allow a single user to modify each document (or document area), or allow concurrent updates to occur and combines them using either a three-way-merge tool [4] or patch-based techniques [5], [6]. In synchronous collaborative applications, each user immediately observes the changes performed by the other users. These applications often use algorithms based on operational transformation (OT) [7], [8] to merge concurrent updates while allowing each client to immediately observe her changes.

More recently, some of these synchronous collaborative applications, notably Google Drive [1], have started supporting disconnected operation for allowing users to modify documents during periods of both voluntary disconnection or connectivity problems. Using traditional groupware classification, we could say that these applications have become multi-synchronous. By relying on algorithms developed to synchronous settings (or simple adaptations of these algorithms), these applications exhibit unexpected behaviour to users when merging updates executed during disconnected periods, e.g. inserted text disappears without being explicitly removed by any user. Also, contrary to asynchronous systems, these systems include no or limited awareness mechanisms, making it hard for a user to understand what has happened during the merge of concurrent asynchronous modifications. The difficulty here is that these existing systems cannot be just corrected since traditional OT algorithms either do not scale or do not permit arbitrary topology of organization between users as allowed by distributed version control systems.

To address these issues, we argue that this new class of applications requires specialized solutions with integrated concurrency control and awareness mechanisms. We propose such a solution based on conflict-free data types (CRDT) [9]. CRDTs ensure eventual convergence of replicas, given that all updates are received in all replicas in causal order. CRDTs address the limitations of OT algorithms, while keeping the same behaviour for users. Unlike previous CRDT-based solutions [10], [11], our solution extends the traditional interface of documents with support for operations appropriate to an asynchronous setting - different granularity for *insert* and *delete* operations. We keep awareness informations that allow the system to show to the users the result of the merge of concurrent modifications.

Additionally, our solution offers new *update* and *move* operations that are necessary to offer fine-grain awareness and merge quality to the user. Even in these additional operations have been recognized important for the end user, few actual implementation exists due to the difficultly in ensuring consistency in presence of such operations and in identifying these subjective operations [12]. We propose a complete solution that allows to manage and to accurately extract these operation in the asynchronous context. We have

COLLABORATECOM 2013, October 20-23, Austin, United States

DOI 10.4108/icst.collaboratecom.2013.254113

implemented our algorithm and evaluated it by measuring the quality of merge of concurrent updates in documents from existing repositories. Results show that our approach improves the results of merge, by making it closer to the result expected by the users.

The remainder of this papers is organized as follows: Section *Requirements* discusses requirements for supporting multi-synchronous collaborative editing; we then present our solution for managing shared documents and providing awareness information to users in the following section; Section *Evaluation* provides an evaluation of our proposal in the context of existing repositories; Section *Related Work* discusses related work; and the following section concludes the paper with some final remarks.

II. Requirements

Collaborative editing applications allow users to collaborate for creating a shared document. To this end, users can typically edit any part of the shared document concurrently. An important aspect of collaboration is awareness [13], which allows a user to be aware of the actions of other users, making collaboration more effective and minimizing potential conflicts. Even with user awareness information, or due to latency to obtain this information, users can easily update an area of the document that is being edited by another user. To address this issue, collaborative editing systems include some form of concurrency control mechanism that handles concurrent actions from multiple users and merge their contributions.

In recent years, a new generation of web-based collaborative editing tools have been developed – e.g., Google Drive [1] and Microsoft Office at SkyDrive [2]. These systems allow users to edit shared documents both synchronously and asynchronously. In this section, we analyze the properties of theses systems in supporting collaborative editing and discuss a set of desired properties for such systems.

A. Limitations of Current Solutions

Google Drive and Microsoft Office at SkyDrive present two different strategies for supporting collaborative editing discussed hereafter.

Google Drive: Google Drive (formally named Google Docs) [1] allows users to edit a document synchronously when they are online at the same time. It also support editing of text documents during disconnected periods, with the reintegration of updates occurring when the user becomes online again.

When merging updates executed during disconnected periods with the current version, the resulting state may be unexpected for users, either because it becomes semantically incorrect or because it violates users' intentions [14]. Some of the problems that can occur are the following:

- *Concurrent updates on the same sentence:* Two users can edit the same sentence simultaneously without being aware of the other users' interaction, which may leave the sentence semantically inconsistent [15];
- *Typographic errors:* When users correct the same typographic error by inserting a character, two characters may show up in the document;
- *Cursor position:* The cursor position of a user may change unexpectedly due to the arrival of new updates;
- *Updates loss:* In some situations, when users update the same block of text offline, insertions from one users can be deleted even if no user explicitly deleted the inserted text. For example, if one user inserts one line in the middle of some other lines and some other user concurrently removes those line, the inserted text will be removed.

Some of these phenomena are the direct result of allowing users to concurrently access any part of the document without coordination and thus it is unlikely that it is possible to avoid them to happen in multi-synchronous settings without restricting users' action. We believe the problem with Google Docs solution is that it lacks a good mechanism for providing awareness information about the result of merges and in particular of situations where potential conflicting actions have occurred.

Google Docs provides only support to recover previous revisions of the documents. This can help in some situations where the merge policy has hidden content produced by some user. However, we found out that the past revisions are not able to accurately reproduce the executed operation. In the example of the update loss we described, where two users concurrently update the same block of text offline, if the first client to get back online is the one that deleted the block, then the revision will not keep the updates from the other user. Independently of whether this is a bug or just the way the merge procedure works, we argue that the current awareness support on this system falls short for asynchronous editing.

Microsoft SkyDrive: Microsoft SkyDrive [2] uses a more conservative collaboration support. In this system, users are forced to synchronize the document explicitly. When a user saves the document, the document is sent to the server, and if there were conflicting updates, he is asked to solve conflicts manually. Other online users will not see the merged document (and updates) until they save the document again. Meanwhile, new conflicts can be generated as users continue modifying older versions of the document.

Additionally, while editing a document, there is awareness information that presents the area other users are editing (but not the contents of theirs edits). This solution provides weak awareness and no synchronous update support, making collaboration hard.

B. Properties of Desired Solution

We now discuss the properties of a solution that supports both synchronous and asynchronous collaborative editing, addressing the limitations of the solution analyzed before.

a) Synchronous mode: While editing a document in synchronous mode, two basic solutions can be adopted: immediately propagate and integrate edits or propagate edits at some user-defined synchronization point. The first solution, adopted by Google Docs, allows users to immediately observe changes performed by other users. The second solution, adopted in Microsoft SkyDrive, allows users to control the visibility of their changes.

The second approach seems the most appropriate when observing partial changes from some other user may pose problems to a user's work. For example, while editing a document with structural requirements, such as the source code of a program, observing partial changes may make it impossible for a user to execute some action, such as compiling the code. In most other cases immediately applying updates from other users seems appropriate. In any case, obtaining awareness that concurrent modifications occur without applying them can help [16].

In our approach we support both solutions: users can immediately observe all changes in synchronous mode while they can control the visibility of their changes by working in asynchronous mode for some periods.

Regarding awareness information, there are two types of awareness information that can be presented. First, the position of users' cursors, for allowing a user to be aware of which area of the document other users are working at. Second, information about other users recent edits, for allowing a user to check other users updates easily. This can be presented as a log of changes (as in Google Docs) or by coloring each users recent edits with different colors.

While an important component for providing this awareness information is the user interface, it is also necessary to guarantee that the necessary information is available: the position of each user's cursor and the identification of the author of each update.

b) Asynchronous mode: While editing a document in asynchronous mode, integrating edits at the granularity of the character may lead to unexpected behaviors. Thus, it seems appropriate that edits are integrated at a much larger granularity using coarse update operations. When concurrent updates to the same element exist, users must be notified to resolve the conflict or at least check that the choices made by the merge algorithm are correct.

Unlike the solution adopted in Microsoft SkyDrive and in most version control systems, it might be interesting to keep the document with multiple versions of some element for some time before resolving the conflict, while allowing additional changes to be performed. This approach would allow new updates to be immediately integrated and conflicts to be resolved at some later time by the most appropriate users. For example, multiple versions of some element may represent alternative options that users may want to decide at a later point in time.

When editing a document, users sometimes move an element from one position to another: for example, a user editing a scientific paper moves the position of a paragraph, or a section; a user editing a source code may move the position of a function in a class, etc. When collaboratively editing a document, problems arise if the element is concurrently updated and moved - the expected result would be to move the updated version of the element to the new position, but this is not supported by typical solutions. To support this semantics, it is important to explicitly support the move operation. Such scenario may also occurs during synchronous editing where these operations are usually executed by cutting and pasting the moved element. Users must be informed about the result of the merge whenever any conflict has been solved.

III. Solution for Managing Shared Document

In this section we present the algorithm to manage the document state. A document is defined as a conflict-free replicated data type. We start by introducing the essential properties of conflict-free data types.

A. Conflict-free Replicated Data Types

Conflict-free replicated data types (CRDT) [9] are a class of distributed data types that allow replicas to be modified without coordination while guaranteeing that replicas converge to the same correct value after all updates are propagated and executed in all replicas.

Two types of CRDTs have been defined: *operation-based CRDTs*, where modifications are propagated as operations (or patches) and executed on every replica; and *state-based CRDTs*, where modifications are propagated as states and merged on every replica.

In this work, we focus on a solution based on *operation-based CRDTs*, as this solution is responsive enough for synchronous collaboration and requires less communication to propagate each update, making it more appropriate to a setting where supporting low latency is important. As such, a shared document is an *operation-based CRDT* (or CRDT when no confusion can arise), with an interface that includes read-only functions, producing no side-effect on the document state, and updates, that produce side-effects.

When a user modifies the document producing an update, the following process occurs: *(i)* the replica computes from this update a downstream operation that can be applied on any replica, *(ii)* the replica applies locally this operation, *(iii)* the replica disseminates the operation to other replicas, *(iv)* the other replicas apply this operation when they want : as soon as received in synchronous mode, when user decides in asynchronous mode.

In the replica where an update is generated, the computation and the execution of the downstream operation are executed atomically in sequence. The downstream operation must be propagated to other replicas for execution, but this propagation can be delayed as long as required – e.g. until the replica reconnects to the network.

It has been proven that if all concurrent downstream operations of a CRDT commute, all replicas converge to the same state after executing all downstream operations in any order that respects causal order [9].

B. System Model

Our solution assumes a set of distributed nodes, S, each one maintaining a replica of the shared document CRDT[1]. Each node communicates with a subset of nodes of S. During this communication, each node s_i may send operations originated in s_i or in some other node. We make no assumption on the topology of the network of nodes, and no assumption on the communication latency. However, as usual in operation-based replication, our algorithm requires that all updates are

[1]In the description, we consider without lack of generality that a single document exists in the system. When multiple documents exist, each document has its own set of nodes that replicate the document.

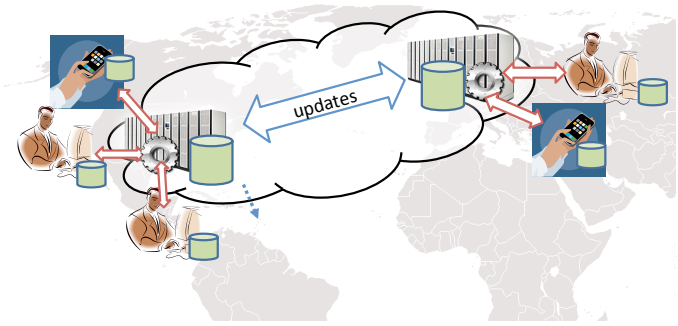


Figure. 1: Possible cloud-based system architecture.

propagated to all replica nodes to work correctly. These requirements can be achieved by using a large number of different protocols [17], [18], [19].

1) Causal Order Requirement: In our solution, we use the fact that updates are delivered in causal order to avoid maintaining information about deleted elements for which the create operation has not been received yet. This requirement could be easily dropped by using techniques similar to the ones proposed in Logoot-Undo [20].

We decided to maintain the causal delivery requirement to guarantee that the state of the document is always causally consistent in the sense that if a user has made some update after observing some prior update, this prior update should always be executed before the more recent one. This property is commonly assumed as a correctness criteria in collaborative editing [14].

2) Possible Deployments: As massively used distributed version control systems (DVCS) – e.g. *git* – our algorithm can be deployed in any architecture, from a peer-to-peer to a centralized solution, such as used in many cloud-based hosting service for software development.

For example, a typical cloud deployment, depicted in Fig. 1, works as follows. The data management service is provided by servers located in a set of data centers located around the world. Each server maintains a replica of the document CRDT. Users access the shared documents in their devices by executing a collaborative editing application. The application manages a local copy of the document CRDT and interacts with a single data center. Updates executed by the user are propagated to the data center. Updates executed synchronously by other users are received sequentially from the data center. Each data center propagates updates received from clients to other clients and to all data ceners. Each data center propagates updates received from a data center to all clients and to other data centers (that have not received them yet).

In such deployment, causal delivery can be achieved very efficiently if each data center propagates operations to other data centers and clients in the same order it has received them.

C. Document CRDT

In our solution, we use a simple document structure, where each document is composed by a sequence of elements. We expect each element to be a semantic unit of the document and be composed by a sequence of more elementary editing elements – for example, an element can be paragraph composed by a sequence of characters. The document CRDT can maintain multiple versions for each element, as a result of concurrent updates.

TABLE I: Handling of concurrent updates to the same element and associated awareness solution.

	insert	update	delete	move
insert	keep two elements	not possible	not possible	not possible
	highlight new elems.	-	-	-
update	-	create versions	delete element	move the updated version
	-	show both versions	show del. element	highlight
delete	-	-	delete	delete element
	-	-	nothing needed	show del. element
move	-	-	-	create clones
	-	-	-	highlight clones

The interface of our document CRDT includes four operations: *insert* an element; *delete* an element; *update* an element, by replacing all existing versions by a new version; *move* an element to a new position.

1) Synchronous Editing of an Element: The operations defined in the document CRDT are coarse-grain, and appropriate for an asynchronous editing setting. When the document is being edited in a synchronous setting, the document CRDT is used as follows.

Collaborative editing in an element version – e.g. adding or removing a character in a line – must be handled by an existing algorithm, such as a CRDT based solution [10], [11] or operational transformation [14] (in our implementation we use TreeDoc CRDT). These solutions are appropriate for synchronous editing as users immediately observe changes executed by other users and can solve any conflict that arises. The state of the collaborative editing session is integrated in the document CRDT by issuing an *update* operation whenever the document is saved and before any document CRDT operation on the element is executed in the local copy.

Other editing actions executed in a synchronous session lead to the execution of a document CRDT operation: a cut & paste leads to a *move* operation; the delete of an element leads to a *delete* operation; and the creation of a new element leads to an *insert* operation.

2) Policy For Handling Conflicting Operations: Table I presents a table that shows how our solution handles concurrent operations on the same element. Because we get more precise awareness information on the operations done by the users, we do not consider every couple of modification occurring at the same position as conflicting, contrary to existing DVCS. For instance, a DVCS considers that the insertion of two new and unrelated methods at the same position in a source code is a

conflict. We claim that users must be aware of such an event but without blocking the editing process.

We now describe our conflict resolution decisions. As just mentioned, when two users insert an element at the same position, i.e., between the same two existing elements, our solution will keep both elements. After merging two versions of the document, we rely on the standard operation awareness to show each user the updates of the other user, allowing them to verify if the automatic result is acceptable.

On two concurrent updates, our solution keeps both versions, as it is not possible to decide which update is the best. After the merge, the editor will show the users this situation.

On concurrent update and delete, our solution deletes the element. However, it keeps the new version of the element that can be shown to the user by the editor. The same approach is used on a concurrent delete and move.

On concurrent update and move, the updated version is moved, as we expect that the move to be changing the place of the element, independently of its value. In this case, the moved version is only highlighted.

On two concurrent moves, the same element becomes visible in two places of the document. The editor must highlight this situation and ask the user to solve it. If a user executes an operation on an element visible in two different places, a copy of the element is created, as we assume that in such case the user wants a copy of the element to be present in each place.

Next we describe how this approach is implemented in the document CRDT specification and how an editor can obtain the necessary information to present the specified awareness information.

D. Document Specification

The code of the document CRDT is presented in Fig. 2. The model of the algorithm is composed of two sets. The set POS associates an element identifier to a position identifier. An element can be cloned to several positions due to concurrent move operations, but a position correspond to a unique element. Positions identifier are them-self unique and *generated* using a CRDT algorithm such as Logoot or TreeDoc. These identifiers are unique and totally ordered and can be generated without coordination [10], [11]. The set VAL associates identifier elements to values through unique timestamp. Again, an element can have several values (versions) due to concurrent update operations, each one with its own timestamp.

The document presented to the user is obtained using the *ordered* list of visible positions. A position is visible if it is associated to an element identifier which has at least one value. For each position, the value presented to the user by default is the value with the greatest timestamp. For positions that have multiple values, editors must convey that information to users using some user-interface technique.

We consider the four operations a user can produce on a document. These operations are *insert*, *update*, *delete* and *move*. Each operations is transformed into a 4-tuple, $(oldPosition, oldValues, newPosition, newValue)$, which are respectively the sets of position and values to remove and position and value to add to the model. These sets are empty by default. The $insert$ operation generates a new element with a new position. When inserting an element between two existing elements, the new position identifier computed using a CRDT identifier will be ordered between the identifiers of those two existing elements. This guarantees that the element will always appear at the same correct relative position in respect to those two existing elements. The $update$ operation generates a new value for an element and removes the existing ones. The $delete$ operation removes the position and all the existing values. The $move$ operation creates a new position for the element and remove the old one.

The $move$ operation can produce clones, i.e. an element appearing at several different positions. Editors should present awareness information to users to make them know that they are editing a clone. If the user decides to edit a clone, we assume that the user wants to affect only the element at the position she is editing. To support this, $update$, $delete$ and $move$ operations that affect a clone remove only the targeted position and create, if required, a new element, thus detaching this position from the original element.

Due to lack of space, we can only give the intuition of the correctness of our document CRDT specification. Eventual consistency of the model's sets is ensured by the *uniqueness* of identifiers. Each element inserted in the sets has a unique identifier - *pos* for the set POS and the pair (id, ts) for the set VAL. These identifiers can only be produced once and thus can only be inserted in the corresponding set once. The identifiers can be deleted by more than one operation, but after being deleted they will never reappear. An identifier is only deleted after being added due to causal delivery. This guarantees that after executing all operation, an identifier is in the set if it has been added and never deleted. Consistency of the view, i.e. the guarantee that all users eventually view the same document, is ensured by the total order of position identifiers and of timestamps.

For efficiency purposes, the sets POS and VAL can be implemented using maps. Our actual implementation contains two maps. The first is a Java TreeMap[2] $pos \rightarrow id$ that allows to compute $positionAt(i)$ and $elementAt(i).id$ efficiently in $O(log(n))$. The second is a hash table $id \rightarrow (\{pos\}, ts \rightarrow val)$ that allows to compute $elementAt(i).positions$ and the value with the greatest timestamp efficiently.

E. Information for Providing Awareness Information

Our document CRDT creates new unique identifiers whenever a new element version is created or updated. The unique identifiers we are using include the identifier of the node where the operation originated and a counter of the operations originated in that node. These identifiers can be used by an editor to provide awareness information about the updates recently executed in other nodes - for example, by highlighting those elements. This is the typical awareness synchronous editors already provide, by highlighting the changes of other users.

In Table II we present how it is possible to detect that conflicting operations have occurred by inspecting the local state of the document CRDT. For presenting the awareness

[2]A navigable map based on a red-black tree.

```
1  local set POS // Set of pairs (id, pos)
2  local set VAL // Set of pairs ((id, ts), value)

4  function positionAt( int index)  // Returns the position identifier of the element visible at a given position
5    let visible = {pos : ∃(id, pos) ∈ POS ∧ ∃((id, ts), v) ∈ VAL}
6    let ordered = order(visible) // Obtains an array of ordered elements
7    return ordered[index]

9  function elementAt(int index)  // Returns the identifier of the element visible at a given position with its clones and its value with the greatest
   timestamp
10   let identifier = choose id : ∃(id, positionAt(index)) ∈ POS
11   let positions = {pos : ∃(identifier, pos) ∈ POS}
12   let value = {val : ∃((identifier, ts), val) ∈ VAL ∧ ∀t : ((identifier, t), x) ∈ VAL ⟹ ts > t})
13   return (identifier, positions, value)

15 operation insert( int index, V v)  // Inserts a new element at a given position
16   let id = unique()
17   newPosition = {(id, generate(positionAt(index), positionAt(index + 1)))}
18   newValue = {((id, unique()), v)}

20 operation update(int index, V v)  // Sets a value at a given position
21   let id = elementAt(index).identifier
22   if |elementAt(index).positions| > 1 // Removes a clone and creates a new element
23     let newId = unique()
24     oldPosition = {(id, positionAt(index))}
25     newPosition = {(newId, generate(positionAt(index − 1), positionAt(index + 1)))}
26     newValue = {((newId, unique()), v)}
27   else  // Removes all values and sets a new value
28     oldValues = {((id, ts), x) : ((id, ts), x) ∈ VAL}
29     newValue = {((id, unique()), v)}
30   endif

32 operation delete(int index)  // Removes an object at a given position
33   let id = elementAt(index).identifier
34   oldPosition = {(id, positionAt(index))}
35   if |elementAt(index).positions| ≤ 1 // Removes a complete element
36     oldValues = {((id, ts), x) : ((id, ts), x) ∈ VAL}
37   endif

39 operation move(int index, int destination)  // Moves an element
40   let id = elementAt(index).identifier
41   let pos = generate(positionAt(destination), positionAt(destination + 1))
42   oldPosition = {(id, positionAt(index))}
43   oldValues = {}
44   if |elementAt(index).positions| > 1 // Removes a clone and creates a new element
45     let newId = unique()
46     newPosition = {(newId, pos)}
47     newValue = {((newId, unique()), elementAt(index).value)}
48   else  // Moves a complete element
49     newPosition = {(id, pos)}
50   endif

52 effect(oldPosition, oldValues, newPosition, newValue)
53   VAL = VAL \ oldValues ∪ newValue
54   POS = POS \ oldPosition ∪ newPosition
```

Figure. 2: Document CRDT with update and move operations.

TABLE II: Conditions for triggering the presentation of awareness information on conflicts.

	insert	update	delete	move
insert	standard highlight	-	-	-
update	-	$\exists ts_1, ts_2, v_1, v_2 : ((id, ts_1), v_1) \in VAL \wedge ((id, ts_2), v_2) \in VAL$	$\exists ts, v : (id, ts), v) \in VAL \wedge \nexists p : (id, p) \in POS$	standard highlight
delete	-	-	nothing to do	$\exists p : (id, p) \in POS \wedge \nexists ts, v : ((id, ts), v) \in VAL$
move	-	-	-	$\exists p_1, p_2 : (id, p_1) \in POS \wedge (id, p_2) \in POS$

information suggested in Table I, in some cases it might be necessary to access some deleted information - e.g. on the concurrent update and delete of an element, the information about the position of the element is not kept in the document CRDT. This information can be maintained in the document CRDT by keeping sets with tombstones containing recently deleted information - we omit this part from the specification for simplicity.

IV. Evaluation

To evaluate the interest of our approach, we need to know if potential users are satisfied by the merge result obtained by our solution. Such user satisfaction metrics can be obtained using user studies. However such studies are limited in scale and require a lot of effort if every possible configuration should be evaluated. Instead, we use an open-sourced benchmark framework [21] able to evaluate collaborative editing algorithms in the context of real usage.

A. Framework

The benchmark framework transforms a git software state-based history of collaboration to an operation-based history. Then, it can replay any implemented operation-based algorithm against this history trace. The framework can measure the performance of the algorithm in term of computing time, memory requirements, bandwidth consumption and merge quality estimation.

To estimate merge quality, the framework follows a procedure similar to the git merging. When a user merges branches in git, it first obtains a best-effort automatic merge result done by diff3 utility. If this automatic merge does not produce conflicts in the git meaning – i.e. two modifications at the same position – the result is automatically committed. If the automatic merge produces conflicts, the user must resolve them before committing. Thus the merge commit present in the history represent the user expectation.

When replaying a git history, the framework computes the automatic merge obtained by the evaluated algorithm and compares it, using the diff tool, to the result committed by the actual user. The framework measure the number and the size of operations obtain by diff to estimate the difference between a collaborative editing merge result and the "ideal" result committed in the history.

Indeed, developers have collaborated to produce the sources present in the studied git repositories. They have made a lot of effort to merge the concurrently produced versions of the source code. If the difference between the merge obtain by a tool and their intended result is shorter, they will spend less effort and we assume that they will be more satisfied by the tool.

B. Operation detection

In order to evaluate our algorithm we need operation-based histories that contains *update* and *move* operations. The used benchmark framework is originally based of the *insert*, *delete* and *replace* operations produced by the standard unix diff utility. These operations manipulate block of lines. A replace operation correspond to the deletion and the insertion of possibly non-related blocks of text at the same position. Starting from these original operations we modified the open-source framework to produce update and move operations.

The problem is that update and move operations are subjective. They are difficult to accurately detect, especially in asynchronous collaboration, were we do not capture each user's keystroke. For instance, considering any deletion followed by an insertion at the same position as an update may not lead to an adequate result since the inserted text may not be related to the delete one. Also, detecting move operation requires to identify non-trivial text that reappears in an other place in the document, but such text may be updated afterward before committing a new version.

A replace operation may not delete as many lines as it inserts and we must detect which lines are updated or inserted or deleted. So, we produce update operations using a dynamic programming algorithm. This algorithm first computes $0 \leq \delta_{i,j} \leq 1$, the Levenshtein distance divided by the size of the lines between each couple of lines in a replace operation, and then find the minimum edit script between the deleted and inserted block of text. To avoid false positive updates, a couple $ins(i)/del(j)$ is considered as an update only if the corresponding $\delta_{i,j}$ is lower than a threshold Tu.

For instance, the following replace operation :

```
- % test if x is greater than 0
- int a;
- Object toto;
- if (x > 0)
==================
+ % file procedure
+ % useful for stuff
+ % test if x is greater or equal than 0
+ int a=0;
+ File f;
+ if (x >= 0)
```

is transformed with a threshold of $Tu = 0.3$ into five operations : one insert, one update, one insert, one delete, and one update:

```
+ % file procedure
+ % useful for stuff

- % test if x is greater than 0
- int a;
==================
+ % test if x is greater or equal than 0
+ int a=0;

- Object toto;

+ File f;

- if (x > 0)
==================
+ if (x >= 0)
```

Since moved blocks can also be updated, to produce move operations, we apply the same dynamic programming algorithm on the remaining delete and insert operations. The move detection has a different threshold Tm and produces only block containing at least two lines in order to avoid obvious

false positives such as a single closing brace "}" deleted and inserted at different positions.

C. Results

To evaluate the quality of the merge obtained by our algorithm we selected nine git repository among the most popular projects available on GitHub web site[3].

In Table III we present the characteristics of git repositories that we used in our experiments, showing for each one: (i) the head commit sha1 used to run our experiments; (ii) the number of files that contains at least one merge operation; (iii) the total number of merges performed by users; (iv) the number of lines edited corresponding to number of lines added and deleted; (v) the number of operations is the number of block added, removed, moved and updated; and (vi) the number of updated and moved operations. We observe that the number of operations correlate well with the number of update and move operations. However, the number of update operations is approximately one order of magnitude higher than the number of move operations.

Using each repository history, we apply the replication benchmark on our algorithm and the original TreeDoc [10] CRDTs for collaborative editing. TreeDoc CRDT supports the typical *insert*, *delete* interface and can be taken as a representative of the results that would be obtained with such interface, including most CRDT and operational transformation solutions. We evaluate two configuration of our algorithm : our algorithm using only *update* operations, and our algorithm using *update* and *move* operations. Since repository histories now contain update and move operations, every evaluated algorithm must handle such operations. The default behavior is to produce the corresponding *delete* and *insert* operation to obtain the same local effect.

1) Thresholds: Before we evaluate our proposed merge algorithm itself, we need to estimate the most adequate values for thresholds Tu and Tm. t this end, we apply the benchmark framework on the repository with more commits – git/git – using different possible values for the couple (Tu, Tm). We vary both Tu and Tm from 0 to 1 in steps of 0.1. For each couple, we measure the three configurations : TreeDoc as a reference, and our algorithm with *update* operations, and with both *update* and *move* operations. Fig. 3 presents the obtained results. The horizontal axis represent the Tu value, the depth axis the Tm value, and the vertical axis the number of lines in the difference between the user committed merges and automated merges computed by the algorithms (less is better).

First, we note that our update and move detection has a small but observable impact on the result of TreeDoc. The reason is that the granularity of the *insert* and *delete* operations is modified by this detection. The maximum impact is 1.92%. In the same way, the move detection also sightly impacts our algorithm with only update operations. The maximum impact is 0.52%.

Second, we observe that our algorithm outperforms the TreeDoc algorithm for any threshold values except for $Tu = 0$ and $Tm = 0$, i.e. without *update* and *move* operations. This is not surprisingly, since in this case our algorithm is the same as TreeDoc.[4]

Finally, the overall best performance is obtained by our algorithm with *update* and *move* operations and threshold values $Tu = 0.9$ and $Tm = 0.2$. With such values the gain obtained against the original TreeDoc is 17.97%.

To understand the real effect of move operation in our framework, we compute the difference between the result of the update only and move and update variation. This is presented Fig. 4. This figure allows to verify that the best threshold value for move detection is around 0.2 and 0.3. We also observe false positive *update* operations do not affect more the quality of the merge than move but false positive *move* operations are more problematic. Indeed, the *move* operation effect is counter-productive (negative result) for values higher or equal to $Tm = 0.9$. Also, despite being observable, the impact of *move* operation on the result quality is low. This is due to the much lower number of *move* operations compared to *update* ones. For instance, we detect 55,483 *update* operations and 4,857 *move* operations with the same thresholds, $Tu = 0.5$ and $Tm = 0.5$. This may also be due to the merge mechanism of git that do not take into account move operations.

2) Repository result: We present the result obtained on the different source code git repositories in Table IV. We use threshold values of $Tu = 0.9$ and $Tm = 0.2$.

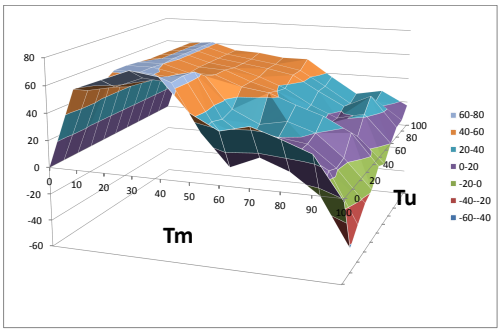


Figure. 4: Move effect

TABLE IV: Evaluation of merge quality in different repositories (results for our solution are the improvement over TreeDoc).

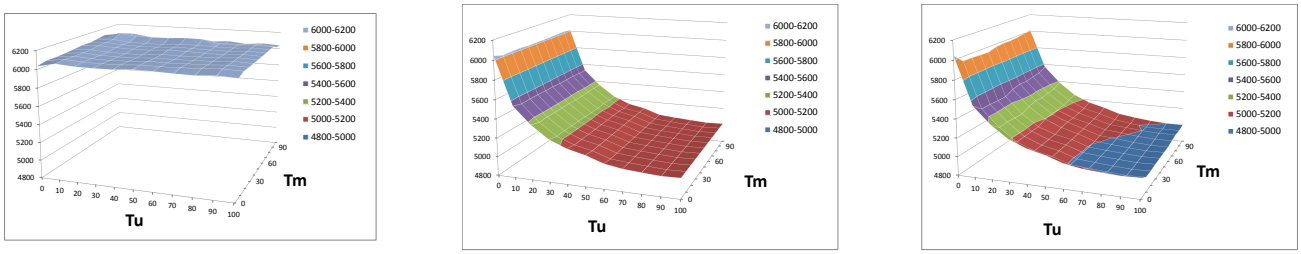
Project	TreeDoc		Updates only		Moves & Updates	
	blocks	lines	blocks	lines	blocks	lines
git	2221	6095	36.6%	17.8%	36.6%	18.1%
backbone	266	798	28.2%	11.7%	27.8%	11.7%
bootstrap	1409	6859	9.7%	6.1%	9.2%	6.0%
d3	711	3026	8.9 %	7.3 %	8.6 %	6.9 %
homebrew	14	25	28.6 %	24.0 %	28.6 %	24.0 %
html5-boilerplate	31	59	19.4 %	22.0 %	19.4 %	22.0 %
jquery	280	702	17.1 %	9.0 %	16.8 %	9.7%
node	532	2494	14.1 %	9.5 %	14.3 %	9.3 %
rails	774	2517	22.4 %	14.5 %	22.5%	15.0%

[3]https://github.com/popular/starred

[4]Except for very subtle implementation details.

TABLE III: Projects characteristics

Projects \ Feature	HEAD SHA1	FILES WITH MERGE	MERGE NUMBER	LINES EDITED	BLOCK OPERATIONS	UPDATES	MOVES
git	8c7a786b	557	5646	676486	141284	67433	3581
backbone	6ac7704c	10	274	56348	16434	8922	374
bootstrap	37d0a305	68	444	225596	38975	19033	2003
d3	d1d71e16	37	489	99093	18214	7963	1651
homebrew	911ded01	10	10	14377	4200	1714	88
html5-boilerplate	f27c2b73	3	34	4467	1577	671	29
jquery	2f2e045e	29	178	154745	40025	21265	1168
node	88333f7a	48	315	165835	39155	16839	1417
rails	36f7732e	351	1157	603528	129762	67507	3101



(a) TreeDoc (b) Update only (c) Update and move

Figure. 3: Threshold effect

We observe that our solution improves the result of reconciliation for all projects, although the degree of improvement varies. As the complexity of reconciliation depends on the use patterns, this variability is expectable.

When comparing results obtained by using block and line granularity, we observe that results for block granularity are usually better. This seems to suggest that line granularity used in most version control systems is not the best match for handling concurrent updates and solving conflicts. Our solution allows any granularity to be used in the operations.

When comparing results obtained with our solution using only *update* or using both *update* and *move* operations, we can observe that the difference is minimal. The reasons for such small difference have been already discussed in the previous sub-section. It also poses the question on whether supporting move, with all its complexity is worth. We continue to believe so, as our experience using (collaborative) editing tools suggest that move is an operation users execute while editing documents.

V. RELATED WORK

Existing approaches to build multi-synchronous collaborative tools or editors are designed upon operations-based techniques since state-based one are not responsive enough during synchronous editing phases.

Some approaches are based on operational transformation [7] (OT) for multi-synchronous collaborative editing. Advantages of OT are its flexibility in defining user operations and the ease of introducing awareness mechanism for concurrent operations. Indeed, every couple of concurrent operations must be managed explicitly by the concurrency control algorithm. The disadvantage of OT is its performance in an arbitrary distributed context. Purely peer-to-peer OT algorithms require non-scalable concurrency controls [22], [14], or dissemination mechanisms [23] that makes them suitable only for asynchronous collaboration. Additionally, most OT solution support only *insert* and *delete* operations.

Molli et al. propose a multi-synchronous [24] and an asynchronous [5] collaborative framework. Theses approaches provide awareness mechanisms such as state treemaps [25] or conflict blocks similar to version control systems. They do not provide update or move operations. The consistency is ensured by the centralized OT algorithm SOCT4 [26]. Preguica et al. [27] ensure consistency with the OT algorithm GOTO [14]. They provide update operations but not move ones. They claim to provide awareness but do not detail the mechanism at all.

Geyer et al. [28] propose a multi-synchronous system where consistency is ensured by a total order on operations and roll back/roll forward mechanism. Such a mechanism can be disturbing for the user, especially during synchronous editing. No awareness mechanism is discussed and introduction of update operations may cause lose of information.

Rahhal et al. [29] uses the Logoot [11] CRDT to ensure consistency of a multi-synchronous semantic wiki. However, they do not manage awareness on conflicting editions and they do not provide update or move operation.

Ignat et al. [16] provide an awareness mechanism in multi-synchronous collaboration. They do not deal with conflicts between concurrent operations, but propose a messaging system

to know that another user is modifying the document before he actually commits its modification. This system is independent from the consistency mechanism and can be also used with our approach. A similar awareness mechanism is provided in Microsoft SkyDrive [2].

VI. Conclusions

In this paper we present a solution for supporting multi-synchronous collaborative editing. Our solution is based on conflict-free data types (CRDT) [9]. Unlike typical CRDT and OT solutions, our solution extends the traditional interface of documents with support for operation appropriate to asynchronous settings - different granularity for *insert* and *delete* operations, and *update* and *move* operations. These operations are essential to provide a better merge quality to the user, as shown in our evaluation.

Our solutions additionally keeps awareness information that allows applications to show to the users not only the updates recently executed by other users, but also to highlight the result of conflicting actions.

As future work we intend to further investigate the usefulness of providing move as a first-class operation for collaborative editing. We also plan to integrate our algorithm in a cloud-based web editing tool that supports geo-replication.

Acknowledgments

This work is partially supported by ANR project ConcoRDanT (ANR-10-BLAN 0208), and by Portuguese FCT/MCT project PEst-OE/EEI/UI0527/2011. Valter Balegas is supported by FCT/MCT Phd scholarship SFRH/BD/87540/2012.

References

[1] "Google drive," https://drive.google.com.

[2] "Microsoft skydrive," 2013, https://skydrive.live.com/.

[3] I. Greif, R. Seliger, and W. E. Weihl, "Atomic data abstractions in a distributed collaborative editing system," in *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '86. New York, NY, USA: ACM, 1986, pp. 160–172. [Online]. Available: http://doi.acm.org/10.1145/512644.512659

[4] F. S. F. GNU, "Diff3. Three way file comparison program," September 2005. [Online]. Available: http://www.gnu.org/software/diffutils/diffutils.html

[5] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine, "Using the transformational approach to build a safe and generic data synchronizer," in *GROUP 2003*. Sanibel Island, Florida, USA: ACM Press, November 2003, pp. 212–220.

[6] D. Roundy, "Darcs: distributed version management in haskell," in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, ser. Haskell '05. New York, NY, USA: ACM, 2005, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/1088348.1088349

[7] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems." in *SIGMOD Conference*, J. Clifford, B. G. Lindsay, and D. Maier, Eds. ACM Press, 1989, pp. 399–407.

[8] "What's different about the new google docs," 2010, http://googledocs.blogspot.fr/2010/09/whats-different-about-new-google-docs_22.html.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems (SSS)*, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976, Grenoble, France, October 2011, pp. 386–400.

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

[11] S. Weiss, P. Urso, and P. Molli, "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks," in *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. Montréal, Québec, Canada: IEEE Computer Society, jun. 2009, pp. 404 –412.

[12] M. K. Singley and J. R. Anderson, "A keystroke analysis of learning and transfer in text editing," *Human-Computer Interaction*, vol. 3, no. 3, pp. 223–274, 1987.

[13] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, ser. CSCW '92. New York, NY, USA: ACM, 1992, pp. 107–114.

[14] C. Sun and C. A. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements." in *CSCW'98*. New York, New York, États-Unis: ACM Press, November 1998, pp. 59–68.

[15] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *TOCHI*, vol. 5, no. 1, pp. 63–108, March 1998.

[16] C.-L. Ignat, S. Papadopoulou, G. Oster, and M. C. Norrie, "Providing awareness in multi-synchronous collaboration without compromising privacy." ACM, 2008, pp. 659–668.

[17] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: a flexible group communication system," *Commun. ACM*, vol. 39, no. 4, pp. 76–83, April 1996. [Online]. Available: http://doi.acm.org/10.1145/227210.227229

[18] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenkcr, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *PODC'87*. ACM Press, 1987, pp. 1–12.

[19] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *SOSP'97*. ACM Press, 1997, pp. 288–301.

[20] S. Weiss, P. Urso, and P. Molli, "Logoot-undo: Distributed collaborative editing system on p2p networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1162–1174, 2010.

[21] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating crdts for real-time document editing," ACM, Ed., San Francisco, CA, USA, september 2011, p. 10 pages.

[22] M. Suleiman, M. Cart, and J. Ferrié, "Serialization of concurrent operations in a distributed collaborative environment," ser. GROUP '97. New York, NY, USA: ACM, 1997, pp. 435–445.

[23] M. Cart and J. Ferrie, "Asynchronous reconciliation based on operational transformation for P2P collaborative environments." IEEE Computer Society, 2007, pp. 127–138.

[24] P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain, "Sams: Synchronous, asynchronous, multi-synchronous environments." IEEE, 2002, pp. 80–84.

[25] P. Molli, H. Skaf-Molli, and C. Bouthier, "State treemap: an awareness widget for multi-synchronous groupware," in *Groupware, 2001. Proceedings. Seventh International Workshop on*. IEEE, 2001, pp. 106–114.

[26] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman, "Copies convergence in a distributed real-time collaborative environment," in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, ser. CSCW '00. New York, NY, USA: ACM, 2000, pp. 171–180.

[27] N. Preguiça, J. L. Martins, H. Domingos, and S. Duarte, "Integrating synchronous and asynchronous interactions in groupware applications," in *Groupware: Design, Implementation, and Use*. Springer, 2005, pp. 89–104.

[28] W. Geyer, J. Vogel, L.-T. Cheng, and M. Muller, "Supporting activity-centric collaboration through peer-to-peer shared objects." ACM, 2003, pp. 115–124.

[29] C. Rahhal, H. Skaf-Molli, P. Molli, and S. Weiss, "Multi-synchronous collaborative semantic wikis," in *Web Information Systems Engineering-WISE 2009*. Springer, 2009, pp. 115–129.