Enhancing rich content wikis with real-time collaboration

Claudia-Lavinia Ignat, Luc André, Gérald Oster

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Enhancing rich content wikis with real-time collaboration

Claudia-Lavinia Ignat,* Luc André and Gérald Oster

Inria, F-54600
Université de Lorraine, F-54000
CNRS, F-54500

SUMMARY

Wikis are one of the most important tools of Web 2.0 allowing users to easily edit shared data. However, wikis offer limited support for merging concurrent contributions on the same pages. Users have to manually merge concurrent changes and there is no support for an automatic merging. Real-time collaborative editing reduces conflicts as the time frame for concurrent work is very short. In this paper, we propose extending wiki systems with real-time collaboration. We propose an automatic merging solution adapted for rich content wikis. Our solution relies on an operational transformation algorithm defined for high level operations that capture user actions such as move, merge and split.

KEY WORDS: CSCW; collaborative editing; operational transformation; wiki

1. INTRODUCTION

The Web 2.0 era is associated with the growing success of participatory collaborative tools for public users. Enterprises have integrated a part of these tools in their information systems to improve their productivity and to facilitate the emergence of a form of collective intelligence.

Among these tools, wikis efficiently share and structure a large amount of knowledge. Wikis are interlinked pages of rich text. Any user can view and edit these pages. For instance, a software development team can dedicate a wiki page to the minutes of their meetings, another one to their bug reports and a last one to the progress of their development. This way, everybody can keep an eye on when is the next meeting, who will attend, the minutes of a previous meeting, what are the bugs found or fixed, and the completed work on the project. Moreover, a wiki allows fast updates and everybody can edit the content without any restriction. A wiki offers all these features in a single tool.

However, most existing wiki systems such as MediaWiki do not automatically merge parallel modifications on the same page. This becomes critical when changes have to be performed very quickly. A typical example is breaking news, when it is common that hundreds of people contribute to the same related Wikipedia pages in a very short amount of time. If two users are concurrently editing the same article, when they try to save their changes, the changes of only one user are published. The other user will be presented with two versions of the wiki page: the one that the user tried to publish and the recently published version. Conflicts, i.e. concurrent changes on the same article, have to be manually resolved by users by retyping or copying and pasting the changes they performed in the last version that was published. Moreover, users are unaware of other concurrent updates on the same article until they try to publish their own changes. Resolving conflicts might become very tedious, and this can be critical when a change has to be published almost instantly.

*Correspondence to: Inria Centre Nancy - Grand Est, 615 rue du Jardin Botanique, 54600 Villers-lès-Nancy, FR. Email: claudia.ignat@inria.fr

Ignat et al. [1, 2, 3, 4] highlight the fact that conflicts are reduced if users see modifications of other users in real-time, i.e. immediately, and that conflicts are increasing with the delay users see modifications of other users. Instead of requiring that users check periodically for updates, real-time collaboration relies on pushing modifications to users immediately after they are available. That is, updates are not submitted at the end of the edit on a wiki page, but each update is immediately sent at its generation and integrated at its reception. Real-time collaboration is useful for publishing breaking news, brainstorming sessions, taking live meeting minutes, live discussion on a bug report or when editing an article when a deadline approaches. Real-time collaborative editing has gained in popularity since the wide availability of free services such as Google Docs.

For confidentiality reasons, enterprises cannot use tools for real-time collaborative editing provided by big service providers such as Google Docs. Documents produced during collaborative editing are stored by these service providers who have therefore control over them. This is a perceived threat of privacy. Enterprises would prefer having control over the storage space and as they use wiki systems they would prefer that they do not need to switch using other tools.

To overcome limitations of existing wikis concerning merging issues and reduce conflicts in the case of concurrent contributions to the same wiki page, in this paper we propose enhancing wikis with real-time collaboration. We propose supporting real-time collaboration additionally to the traditional asynchronous mode of collaboration where users can work in isolation on the wiki page and merge later their changes. Users can switch from one collaboration mode to the other when they wish.

Operational transformation approach initially proposed by Ellis and Gibbs [5] and then further improved by other researchers such as Sun et al. [6] was identified as a suitable approach for maintaining document copies consistency in real-time collaborative editing. In this approach user changes are modeled as document operations. Local operations generated by a user are executed immediately on their own document copy. Remote operations, i.e. operations received from other users, are transformed against concurrently executed operations. The transformations should be performed in such a manner that the intentions of the users are preserved and, at the end, the copies of the documents converge.

Wikis can be edited either as plain text documents with a special markup-based syntax or using a WYSIWIG (What You See Is What You Get) editor which allows content (text and graphics) of the wiki page to be displayed onscreen during editing in a form closely corresponding to the end result.

Simply applying existing operational transformation approaches with a set of transformations designed for plain text documents such as proposed by Imine et al. [7] would fail to preserve the special wiki syntax after integration of concurrent operations. This might also break the underlying hierarchical structure of the wiki.

Existing software such as Google Docs[†] or CoWord proposed by Xia et al. [8] are WYSIWYG editors but they do not take advantage of the underlying structure of the document. Basically, they see the document as a linear sequence of elements. High level operations are translated into primitive insert, delete and update operations on basic elements. This approach leads to simple algorithms, but semantics of user operations is lost. For instance, move of a block of text is obtained by deletion of the block character by character followed by re-insertion of each character of the block. If a user modifies a block of text that is concurrently moved, then his changes are either completely lost or they are placed outside the context of the moved block of text. In this case, user intention of modifying the paragraph is lost.

In this paper, we propose an operational transformation mechanism defined for high level operations that capture user actions such as move, merge and split. We define a document model and its associated grammar as well as high level operations for the proposed document. We specify the combined effects of concurrent operations. This paper is the first one that proposes an operational transformation approach for high level operations, existing approaches being limited to low level operations (insert, delete and update) on textual documents. We describe an architecture of the wiki system that allows both real-time and asynchronous collaboration and switching from one mode of

[†]https://docs.google.com/

DOI: 10.1002/cpe

collaboration to the other one. This architecture also allows users to edit a wiki page either using a wiki syntax or a WYSIWYG editor.

As a validation of our proposed contributions we very briefly describe the integration of the proposed operational transformation approach and of the architecture that allows both real-time and asynchronous editing in the context of a wiki system called XWiki[‡]. This integration resulted into XWiki being the first system that allows collaborative editing of wiki pages where user changes are merged immediately after they are performed.

In this paper, we first review existing approaches offering support to real-time collaboration over rich text documents. We then provide a general architecture for our approach, describe and justify our document model, our operations, the merging algorithm and the associated operation transformations. Further, we briefly discuss the benefits and the limitations of managing operations which capture high-level user intentions. Finally, we provide concluding remarks and some directions for future work.

2. RELATED WORK

To our knowledge no wiki system that offers real-time collaborative editing exists. Confluence, one of the most popular wikis in corporate environments, provides collaborative editing awareness, in which editor icons are shown on the sidebar to indicate whether somebody is concurrently editing the same page. However, user concurrent changes on the same wiki page are not automatically merged in real-time.

In the rest of this section we present existing real-time collaborative editors for rich text content by focusing on their synchronisation mechanism.

CoWord [8] is a plug-in that supports collaborative editing of Microsoft Word documents. It uses Transparent Adaptation (TA) approach for converting each high-level operation to a sequence of primitive operations. Operational Transformation (OT) technique is applied to this sequence of primitive operations for maintaining consistency among document copies. In CoWord, the set of primitive operations is *Insert*, *Delete* and *Update*. The editor supports creating any Word (rich text) document, but the transformation of high-level operations into primitive ones erases all information about the original high-level operation. For instance, the move of a character is transformed into one *Delete* operation followed by one *Insert*. Concurrently moving a sequence of ten characters while inserting a new character between the fourth and the fifth one will result in the ten characters moved (deleted then inserted), and the new single character inserted outside of its context. The expected result is to have the new character between the others, as the user intended to. Information about the move intention is discarded.

Google Docs and Etherpad[§] work in a similar way. All edits are transformed into three basic types of changes: inserting text, deleting a range of text and applying styles to a range of text. Transformations are provided for all pairs of these types of changes. However, these operations are not enough for capturing user intention. For instance, the previously provided scenario leads to the deletion of the moved sequence of characters together with the new inserted character and the re-insertion of moved sequence of characters at the new position. The new inserted character disappears.

Davis et al. [9] proposed an approach for real-time collaboration over SGML (Standard General Markup Language) documents. The operations that can be performed on the tree are the insertion of a subtree as a child of a specified node, the deletion of a subtree and the modification of the content of a node. As authors mention, this approach does not deal with high-level operations such as move. Indeed, dealing with special cases of concurrent moves is not trivial and the paper does not provide enough details on how this case can be managed.

[‡] http://www.xwiki.com/
[§] http://etherpad.org/

DOI: 10.1002/cpe

Docx2Go proposed by Puttaswamy et al. [10] is a framework for collaborative editing over XML documents on mobile devices where not all devices have the complete document. It adapts the Logoot approach proposed by Weiss et al. [11] for XML documents. XML elements possess unique identifiers, the set of identifiers being ordered and dense. Docx2Go supports four types of operations at the element level: *Insert* adds a new element at a specific location in the relative order; *Delete* removes a specified element; *Update* changes the internal contents of an element; and *Move* changes the relative order of a specified element with respect to other elements. When one element is concurrently edited, the generated conflict can be resolved manually or automatically. In the case of a manual resolution the owner of the document is in charge of resolving the conflict. However, very often in collaborative editing multiple users edit the document and there is no explicit owner of the document. In the case of automatic resolution, when concurrent updates are performed on the same element, the element will be duplicated, each version of the element including the individual changes performed. For instance, if the element is a paragraph, the document will contain as many versions of the paragraph as the number of concurrent changes. The paper presents no awareness mechanism that would inform users about the different versions of the elements that were concurrently changed.

As we can see, there is no suitable mechanism that offers an automatic resolution of conflicts that closely reflects user intentions. The previously highlighted issues are not simple bugs that can be corrected. The only way that an approach preserves user intentions is that it deals with a rich set of operations for which intentions are precisely specified. After providing in the next section a description of the various working modes, we will describe the model of the document, the set of operations reflecting user intentions and the combined effects of pairs of these operations.

3. WORKING MODES

The wiki system architecture consists of a wiki server that stores all wiki pages and a set of clients. Each time a user edits a wiki page, a copy of the edited page is created at the client side by downloading its last revision from the server.

Usually, wiki users edit pages in isolation by performing changes on their local copy of the wiki page and then publishing them at a later time. In this collaboration mode users are not informed about concurrent changes of other users on the same page. Users will notice concurrent changes only when they save their changes. Consider that two users start editing in parallel the same wiki page. Suppose one user published his changes. When the second user tries to save his modifications, his save request is aborted. He is notified that the content he has edited is not anymore based on the last published revision of the page. Generally, the wiki user interface presents him the two revisions of the page: the old revision he modified that includes his changes, and the last revision that is available on the server. He has to manually merge both revisions before being able to save the page. This process of resolving conflicts can be very tedious and often can lead to lost updates (modifications present in one of the two revisions are unintentionally discarded). Figure 1 illustrates the architecture for XWiki system which is very similar to other existing wiki systems with an asynchronous collaboration. The system is composed of a XWiki server and several XWiki web clients. An XWiki web client fetches wiki pages from the server, updates them and saves them back to the server. Merging at the level of the client can be manually done by the client or it can be an optional automatic local merge.

To overcome the issues generated by the isolated mode, we envisaged a wiki system with a real-time collaboration mode where a change of one user is immediately integrated in the copies of the document maintained by the other users. In this real-time collaboration mode, as soon as a user performs a change, this change is sent to the server that processes it by potentially performing some transformations. The server then sends it to the other user clients which at their turn might transform it according to the local changes before execution.

First, we investigated whether such a collaboration mode would be of interest to the clients of the XWiki system. We have done some short interviews with representatives of the companies clients of the XWiki system. We explained them the functionality of the real-time collaboration mode and

DOI: 10.1002/cpe

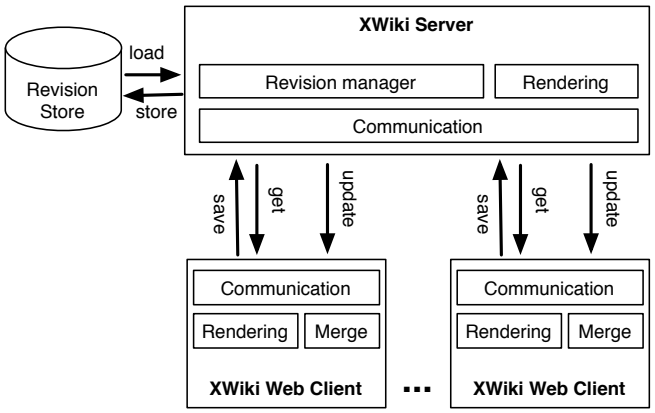


Figure 1. Architecture for asynchronous wiki editing.

we asked them whether such functionality would be helpful for their daily work. We also asked them to provide us with scenarios where this collaboration mode could be used. Interviews revealed that this collaboration mode would be of great interest in some particular cases of use such as taking notes collaboratively during meetings or presentations and when the company has to deliver with very short deadlines documentations that require contributions of several employees. We also investigated forums of users of Confluence and noticed that they also manifested a high interest in this collaboration mode ¶.

We concluded that we would enhance the existing XWiki system with the real-time collaboration, but we would keep available the asynchronous collaboration mode and allow users to switch from one collaboration mode to the other when they want. Users can start several collaborative editing sessions in parallel on the same page and they can edit the wiki page by using the wiki syntax or the WYSIWYG editor.

To support all these features, some modifications on the classical architecture of the wiki system are required: (i) a session manager that is able to manage editing sessions (start / end of isolated-mode editing sessions, start / end of real-time editing sessions, user joins / departures from real-time sessions, etc.) (ii) a communication mechanism that allows to stream changes between clients via the wiki server, (iii) a merging mechanism that is able to merge changes from the different users with the local changes. The new architecture for the XWiki system featuring real-time editing is illustrated in Figure 2.

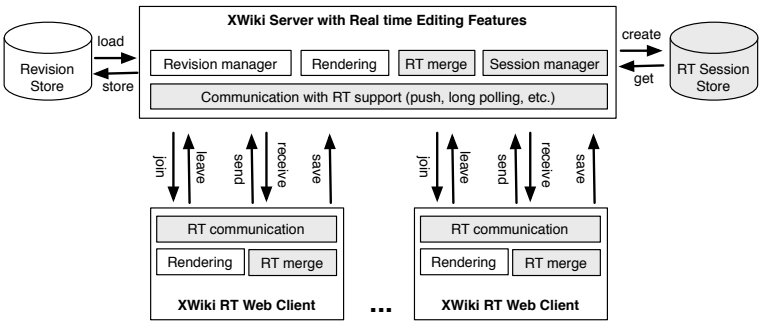


Figure 2. Architecture for real-time (RT) wiki editing.

We integrated our approach in the XWiki system. The solution that we proposed is available as an extension to XWiki. XWiki is a java-based wiki that runs on any servlet container such as Tomcat

¶https://jira.atlassian.com/browse/CONF-8333

DOI: 10.1002/cpe

or Jetty. The Jupiter algorithm proposed by Nichols et al. [12] required that the same algorithm is running on the server side and on the client side. Server side code is developed in Java while client side is based on standard web technology such as HTML and Javascript. In our implementation, we decided to use the Google Web Toolkit (GWT) that allows to write the client side code in Java and cross compile it into Javascript. This ensures that exactly the same algorithms (Jupiter + transformation functions) are running on the client and the server sides.

For the real-time communication component, we used HTTP streaming technology. In this approach, the browser opens a single persistent HTTP connection to the server for all events, which is handled incrementally on the browser side. Each time the server sends a new event, the browser receives and interprets it, but neither side closes the connection. The socket connection between the client and the server remains open until the last page/response has been sent. This solution pushes changes to other clients as soon as they are generated.

4. DOCUMENT MODEL

The source code of a wiki is called wikitext and is a combination of macros, meta information for a page and HTML. There is no one-to-one correlation between wikitext and HTML as certain features are present in one but not the other. Therefore, we cannot consider the DOM (Document Object Model) as the underlying representation of the document. There are also two other reasons why DOM cannot be considered as the basic document structure. Firstly, translation of user changes on the underlying DOM representation of the document is browser dependent and varies from one browser to the other. Secondly, users should be able to switch from editing the wiki page from a wiki syntax to WYSIWYG editor.

Representing the document model by using a linear structure is not feasible. For instance, styles and paragraphs contain text which in a linear structure would be represented by means of special characters for begin and end of the text. Correctly updating these characters when operations and their transformations are performed and guaranteeing that the document is well formed would be very complex.

The solution that we adopted was to design a specific hierarchical wiki structure for the underlying document model. This allows wikitext to be represented in a sufficiently abstract manner such that it can be modified and rendered back into wikitext without loss of information as well as into a variety of HTML formats or no-HTML formats such as plain text. Adopting a wiki specific model as the underlying structure was also the solution chosen by MediaWiki visual editor [‖].

The structure of the wikitext is described using elements, some of which contain other elements while others contain content, but never both. The elements composing the wikitext are listed below:

- **Paragraph** consisting of a series of lines of content.
- **Heading** consisting of a single line of content and a heading level.
- **List** consisting of a series of items, each containing a single line of content, depth and style information.
- **Table** consisting of a series of rows, each containing a series of cells composed of a series of elements.
- **Template** consisting of an application controlled content with any number of parameters composed of content/elements.

The meaning or appearance of text can be defined by annotating the previously described elements to obtain the following features:

- **Formatting**: Bold, italic, internal and external links, etc.

[‖] https://www.mediawiki.org/wiki/VisualEditor

DOI: 10.1002/cpe

- **Rendering**: Images and templates.

The XWiki WYSIWYG editor has a hierarchical structure summarized by the following grammar:

```
document  =  element+
element   =  paragraph | heading | item | tableRow
paragraph =  p(content+)
heading   =  h[1-6](content+)
item      =  (ol|ul)@depth(content+)
tableRow  =  R(content+) cell+
cell      =  C(content+)
content   =  text | inline | link | image
inline    =  span @style (text)
link      =  a @href @style (text)
image     =  img @src (text)
```

The *document* is seen as a sequence of elements that can be paragraphs, headings, items of a list or rows of a table. A *paragraph* node, with node name `p` contains a sequence of contents whose structure will be detailed in the next paragraph. *Heading* represents section titles of the document. Depending on the relative importance of each section, the names of those nodes are `h1`, `h2`, `h3`, `h4`, `h5` and `h6`. `h1` corresponds to the major section of the document, while `h6` corresponds to the least important section of the document. A heading has the same structure as a paragraph, i.e. it contains a sequence of contents. An *item* node of a list has also the same structure as a paragraph containing a sequence of contents. The node name can be either `ol` or `ul` with `depth` attribute. `ul` denotes an unordered list and `ol` denotes an ordered list. `depth` denotes the level of imbrication of the elements of the list. A table is composed of *tableRows*. Each row of the table is composed of a sequence of cells. The first cell in the row is denoted by `R`, while the other cells are denoted by `C`. A *cell* has the same structure as a paragraph, i.e. containing a sequence of contents. Note that rather than defining an element list or table, we define only item list elements with a level of imbrication and table rows. The reason is that we want to have as far as possible elements with the same structure such that we can define generic operations on these elements.

A *content* element can be:

- a *text* node containing a sequence of characters
- an *inline* node whose node name is `span` and that has a `style` attribute. An *inline* node is composed of a single text node.
- a *link* or anchor node whose node name is `a` and that has `href` and `style` attributes. The `href` attribute specifies the URL of the page the link refers to, i.e. the link's destination. The `style` attribute specifies the style of the link.
- an *image* node whose node name is `img` and that has a `src` attribute.

For instance, for the wiki page given in Figure 3b, the wikitext in the wiki syntax of XWiki is presented in Figure 3a and the underlying structure of the document is given in Figure 3c.

5. OPERATIONS

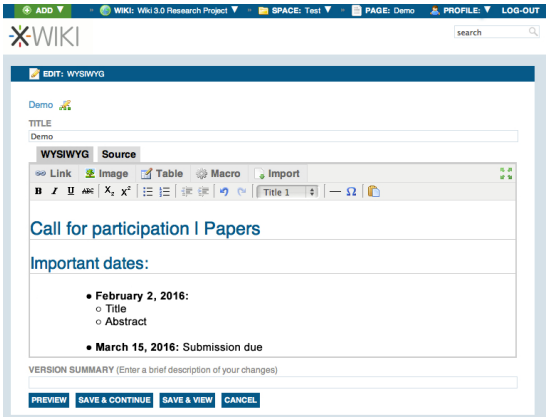
Before describing the list of operations, we first provide an explanation about the identification of the elements or nodes targeted by the operations. An *element* in the document structure presented in the previous section, i.e. in the first level of the document, directly under the root, is identified by an *index* in the list of elements. For instance, in Figure 3c, the element `<h2>` is identified by the index 1. Other interior nodes are identified by their *path* in the tree. For instance, in Figure 3c, the span node `<span style='bold'>` with the content `''February 2, 2016''` is identified by `[2,0]`. A character in a text node is identified by the *path* to the text node and a position *pos* inside the text node. For instance, the character `'e'` from the text node `''February 2, 2016''` is identified by the *path* `[2,0]` to the text node and the position 1.

DOI: 10.1002/cpe

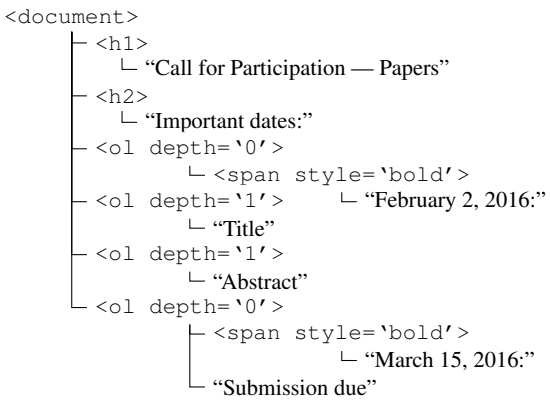
```
= Call for participation | Papers =
== Important dates: ==

* **February 2, 2016:**
** Title
** Abstract
* **March 15, 2016:** Submission due
```

(a) Wikitext of the wiki page.



(b) Wiki page being edited in XWiki.



(c) Document model.

Figure 3. The different representations of a wiki page.

In the following, we will use `<b></b>` to refer to `<span style='bold'>` and so on. We remind that this kind of explicit structure is not visible to users neither in the wiki syntax nor in the WYSIWYG mode. Users see this explicit structure only in the case of an export to HTML format. Wiki syntax has special mark-ups that replace the begin and end tags for certain styles and structures.

We defined the following operations that integrate user action semantics and that are applied on the document model presented in the previous section:

- `NewElement(int index, String content, int siteId)` that creates a new element with an empty text node. The operation takes as parameters the index `index` of the new element, the type of the element `content` such as `p` for paragraphs, `h1`, `h2`, `h3`, `h4`, `h5`, `h6` for headings, `ol`, `ul` for list items and `R`, `C` for cells of a table and the site identifier `siteId` that will be used for defining a priority in the case of concurrent insertions at the same index. For instance, if two operations `NewElement(index1, content1, siteId1)` and `NewElement(index2, content2, siteId2)` are generated concurrently where `index1=index2` and `siteId1 < siteId2`, operation `NewElement(index1, content1, siteId1)` will have priority over `NewElement(index2, content2, siteId2)`. This means that operation `NewElement(index1, content1, siteId1)` will be executed in its original form and an element with content `content1` will be added at the index `index1`. Operation `NewElement(index2, content2, siteId2)` will be transformed and the element with content `content2` will be added at `index2+1`. In addition to creating a new element, `NewElement` operation creates a leaf text node as a child of the new element. The `NewElement` operation does not insert text in the document model. In order to insert text in a new added element at position `index`, the operation `InsertText(0, [index,0],...)` has to be called.

DOI: 10.1002/cpe

- `UpdateElement(int index, String content, int siteId)` updates the type of the element at index `index` with the type specified in `content`. The site identifier `siteId` is used for defining a priority in the case of concurrent updates at the same index.

- `MergeElement(int index, String content, int leftChNb)` merges two adjacent elements. The parameters of this operation are the index `index` of the right element to be merged, the type `content` of the left element and the number of children `leftChNb` of the left element to be merged. The operation appends the children of the right element after the children of the left element. The `leftChNb` and `content` parameter are not used for the execution of the operation but during the process of transformation of operations.

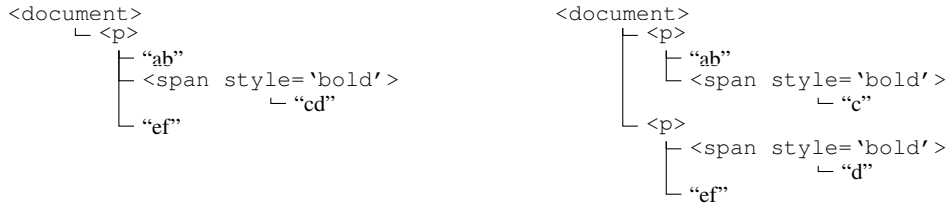
- `Split(int pos, int[] path, boolean splitLeaf)` splits one node in two. This method takes as parameters the path `path` to the text node to be split and the position `pos` inside the text node where the split will occur.

  If the split position is between two nodes, `pos` will be equal to `0` and `path` references the right node. In this case, the method creates a new element after the element to be split and moves into the new element the children of the initial element located after the split position. In this case the parameter `splitLeaf` that is used during operation transformations is set to `false` meaning that the children were moved without being themselves split. For instance, Figure 4 illustrates that the split of `<p>ab<b>cd</b>ef</p>` before `c` leads to `<p>ab</p><p><b>cd</b>ef</p>`.

(a) initial document. (b) after split execution.

Figure 4. Document model before and after a split before character 'c'.

  In the other case, the split position is inside a text leaf node. In this case the leaf is divided in two parts. If the leaf has as parent a node with a `style` attribute, this parent will be duplicated such that both divided parts have their own parent. As in the previous case a new element is created after the element to be split and the children of the initial element situated after the split position are moved into the new element. In this case the parameter `splitLeaf` that is used during operation transformations is set to `true` as a leaf was divided into two parts. For instance, Figure 5 illustrates that the split of `<p>ab<b>cd</b>ef</p>` after `c` leads to `<p>ab<b>c</b></p><p><b>d</b>ef</p>`.



(a) initial document. (b) after split execution.

Figure 5. Document model before and after a split after character 'c'.

- `MoveElement(int origin, int dest, int siteId)` moves one element child of the root document located at index `origin` to the new index `dest`. The parameter

DOI: 10.1002/cpe

`siteId` is used for the case when two different elements are concurrently moved to the same location or when a same element is concurrently moved to two different locations. The index `dest` is computed with respect to the document model containing the element at the original position. For instance, in a document that contains five paragraphs (p_0, p_1, p_2, p_3, p_4), the operation `MoveElement(1,3,9)` executed by $Site_9$ moves the paragraph p_1 between p_2 and p_3, resulting into (p_0, p_2, p_1, p_3, p_4).

- `InsertText(int pos, int[] path, char c, int siteId)` inserts one character in a leaf text node. It takes as parameters the `path` of the leaf node, the position `pos` where insertion will take place inside the leaf, the character `c` that is inserted into the leaf and the site identifier `siteId` of the site that generated the operation. The parameter `siteId` is used to define the priority in the case of concurrent operations at the same position.

- `DeleteText(int pos, int[] path)` deletes one character in a text node. It takes as parameters the `path` of the leaf node and the position `pos` of the character inside the leaf node that will be deleted.

- `Style(int start, int end, int[] path, String param, String value, int siteId, boolean splitStart, boolean splitEnd)` adds/deletes a style to some text (bold, underlined, link, ...) inside a leaf text node. The parameters `start` and `end` define the region inside the text node where the style will be applied and `path` defines the path to the text node. The style is defined in the form of a pair composed of the name `param` of the style and its value `value`. For instance, for making bold a part of the text we can set `param=bold` and `value=true`, while for removing the bold style of the text we can set `param=bold` and `value=false`. The parameter `siteId` is used for the case of concurrent modifications of the same attribute. The parameter `splitStart` is set to `true` if `start` has a value different from `0` at the generation of the operation and to `false` otherwise. The parameter `splitEnd` is set to `true` if `end` has a value different from the length of the text of the leaf at the generation of the operation and to `false` otherwise. The parameters `splitStart` and `splitEnd` will be used during the transformations and show whether the style was initially applied for the begin, end, middle or the whole part of the text. When this operation is applied the leaf node is either kept unite or is divided into two or three parts depending on the values of the parameters `start` and `end`. If the leaf has a parent that has a `style` attribute, this parent is replicated for each part of the leaf. Otherwise, a parent node is created between the root and the new created parts of the leaf. Afterwards, the parameter `param` of the parent node is created if it did not exist before and the value `value` is associated to it. Figure 6 shows an example of an update of the document model after $Site_1$ applies an operation `Style` to make bold a character.

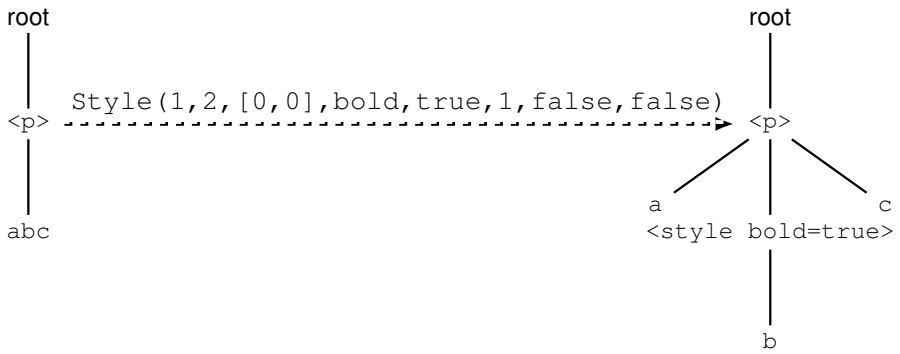


Figure 6. Illustration of a Style operation applied to a single character within an existing paragraph.

DOI: 10.1002/cpe

For deleting an element the characters of the element are successively deleted and the empty element is kept in the document.

Table I summarizes all operations and their parameters.

InsertText	pos	insertion position into the leaf
	path	path from the root to the leaf
	char	inserted character
	siteId	unique site identifier
DeleteText	pos	deletion position into the leaf
	path	path from the root to the leaf
NewElement	index	index of the new element
	content	type of the new element (h1,p...)
	siteId	unique site identifier
UpdateElement	index	index of the updated element
	content	type of the updated element (h1,p...)
	siteId	unique site identifier
MoveElement	origin	index of the element to be moved
	dest	final index of the element
	siteId	unique site identifier
MergeElement	index	right element index
	content	type of the left element
	leftChNb	number of children in the left element
Split	pos	split position into the leaf
	path	path from the root to the leaf
	splitLeaf	true if a leaf needs to be split
Style	start	start position of the range into the leaf
	end	end position of the range into the leaf
	path	path from the root to the leaf
	param	name of the style
	value	value of the style
	siteId	unique site identifier
	splitStart	true if the range does not include the start of the leaf
	splitEnd	true if the range does not include the end of the leaf

Table I. Summary of operations and their parameters.

6. SYNCHRONISATION MECHANISM

In this section we describe our synchronisation approach. In the first subsection we give an overview of the architecture for the synchronisation of the document models. We next describe our choice for an existing merging algorithm. We further describe an overview of our proposed transformation functions for the operations we defined on the document model. Last subsection details the split and merge operations as well as the transformation functions involving these operations. This work is the first one that considers split and merge on a hierarchical structure.

6.1. Overview architecture for real-time synchronisation

In Figure 7 we show the architecture for the real-time collaboration involving two clients that can edit a wiki page either using the wikitext syntax or using the WYSIWYG editor. When a user is modifying the wiki page by using the WYSIWYG editor his actions are caught by the browser and reflected in the DOM model. When the user is modifying the wikipage by using the text editor with the wiki syntax, changes are reflected on the wikitext model. Changes on the DOM or wikitext model are translated to the tree model. Each client maintains locally a wikitext model,

DOI: 10.1002/cpe

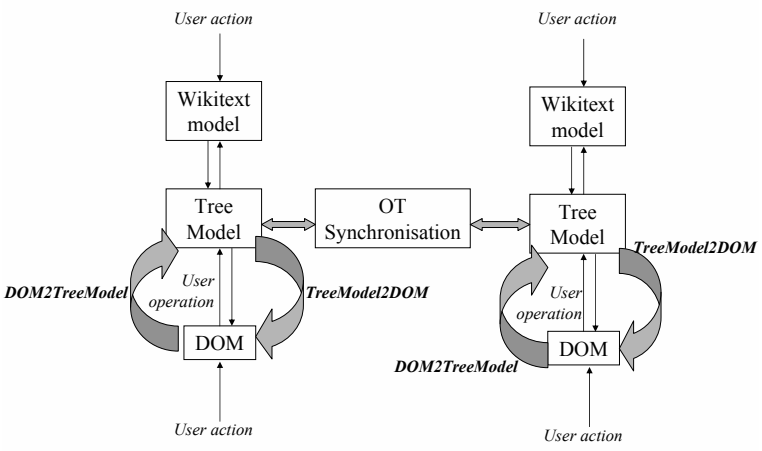


Figure 7. Architecture for real-time synchronisation.

a DOM representation and the tree model representation of the wiki page. Concurrent changes done on copies of the tree model are merged by the OT synchronisation module (that uses the OT merging approach) and therefore copies of the tree models converge. Modifications done on the tree model due to modifications of remote users are translated to changes on the wikitext model and on the DOM. Tree model copies are synchronised resulting into synchronised wikitext models. DOM representations are not synchronised as they are browser specific.

6.2. Traditional merging algorithm

As a merging approach we use operational transformation mechanism that is well known for its suitability for the real-time collaboration.

An operational transformation approach is composed of an integration algorithm and a set of transformation functions. Any integration algorithm such as Jupiter proposed by Nichols et al. [12] or SOCT4 proposed by Vidot et al. [13] can be used. In our implementation we used Jupiter algorithm. In what follows we provide a brief description of this algorithm.

Jupiter uses a 2-way synchronisation approach that allows a client to synchronise with a server. Shared documents are replicated at all cooperating client sites and the central server. Consequently, only client-server communication is needed. Both clients and servers keep operations in a log that for ease of explanation of the underlying approach can be seen as a 2-dimensional state space graph. The nodes of the state space graph denote application states and are represented by means of state vectors. As synchronisation is performed between two sites only, i.e. the server and a client, the state vector contains two elements: the first one represents the number of operations locally generated and the second one represents the number of operations received from the other site. The edges of the graph represent either the original user requests or the result of transformations of operations. The transformation function involving two operations o_1 and o_2 returns as a result a pair of operations (o_1', o_2') where o_1' is obtained by transforming o_1 against o_2 and o_2' is obtained by transforming o_2 against o_1. A transformation can be performed only when the two operations have been generated from the same state. A transformation of an operation against another operation is obtained by translating the first operation along the vector representing the second operation. For instance, Figure 8 illustrates the fact that the transformation of vector s_1 against c produces s_1'. But, s_2 cannot be transformed against c as s_2 and c were not generated from the same state, c being generated in the state $(0, 0)$ and s_2 being generated in the state $(0, 1)$. By translating c from the state $(0, 0)$ to the state $(0, 1)$, i.e. transforming c against s_1, the resulting operation c' will have the same generation state as s_2. Therefore, s_2 can be transformed against c', the result being s_2'.

A local operation is executed immediately and then propagated to the server. The server transforms the operation, if necessary, then executes the transformed operation on its copy of the

DOI: 10.1002/cpe

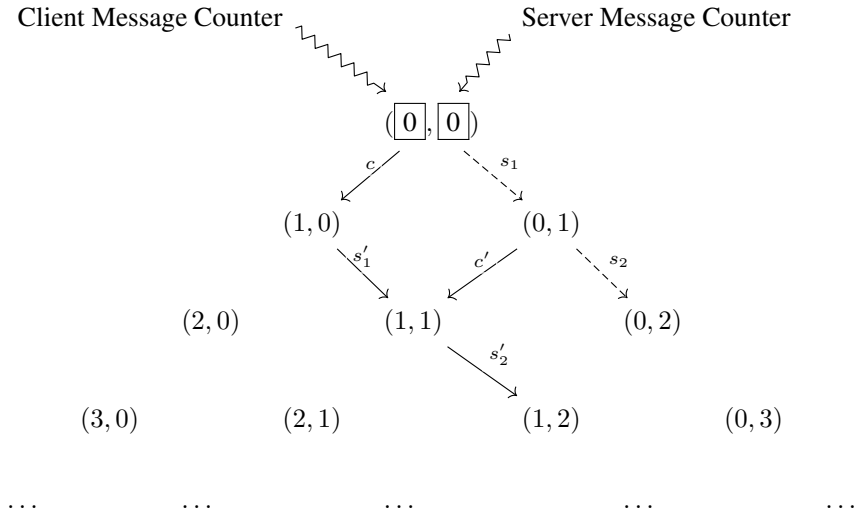


Figure 8. The state graph in the Jupiter approach

document and broadcasts the transformed operation to all other client sites. When an operation sent by the central server is received at a client site, it has to be transformed before it is executed on the local copy of the document.

We now explain the main idea of the execution of a remote operation at a client site. Suppose that the last known state of the server at a client site was (x, y). Further, suppose that the state at the client site is $(x + k, y)$, i.e. the client generated k messages since the state (x, y), as illustrated in Figure 9. These k messages are kept in the state space graph as they are used in the process of transformation of the incoming server operations. The next incoming message from the server must originate from one of the states between (x, y) and $(x + k, y)$, i.e. the server must have processed some of the k messages generated by the client. Suppose that the server message originates in the state $(x + i, y)$ and therefore the current state of the server is $(x + i, y + 1)$. The saved operations locally generated between the states (x, y) and $(x + i, y)$ are discarded since they are no longer needed, the server having already processed them. Remote operations generated by the server have to be sequentially transformed against the client operations that are not included in their generation state. These operations are generated between the states $(x + i, y)$ and $(x + k, y)$. As a result of this transformation, the edge of the graph originating at $(x + i, y)$ and ending at $(x + i, y + 1)$ is translated to the edge starting at $(x + k, y)$ and ending at $(x + k, y + 1)$. Meanwhile, the operations saved at the client have to be transformed in order to include the effect of the remote operation from the server, such that a new remote operation that arrives at the client site is correctly transformed against the saved operations. Therefore, the sequence of saved operations ranges between the server states $(x + i, y + 1)$ and $(x + k, y + 1)$.

The generalisation of the 2-way synchronisation described above to a multi-way synchronisation has been explained in Zafer et al. [14]. All clients maintain a local copy of the shared document and a state-space graph to keep track of the operations that have been executed locally. The server maintains a copy of the document and a state-space graph for each client. Each client-server pair synchronises their copies in the same way as was done by the 2-way communication in Jupiter. In the n-way synchronisation when a message generated by a client is received by its corresponding server, it is transmitted to the other servers. At their turn, servers synchronise with the corresponding clients, as if that message had been generated locally. Forwarding a message from a server to other servers has to be atomically performed before any other message is processed by any of the servers. Concurrent messages are processed in the order in which they arrive at the server. A client processes

DOI: 10.1002/cpe

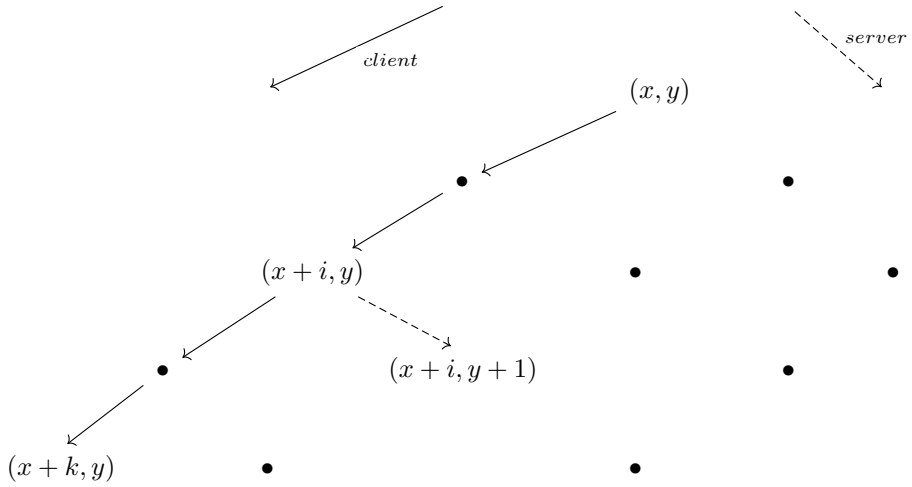


Figure 9. The state space graph at the execution of a server operation at a client side in the Jupiter algorithm

operation	structure modification			
	element index	element type	inner tree structure	leaf content
InsertText				×
DeleteText				×
NewElement	×			
UpdateElement		×		
MoveElement	×			
MergeElement	×	×	×	
Split	×		×	×
Style			×	×

Table II. Classification of operations.

immediately locally generated operations and then remote operations are processed in the order in which they have been processed at the server side.

After describing the main algorithm for integration of operations, in the next subsection we provide a brief overview of the transformation functions for the defined set of operations.

6.3. Overview of the proposed transformation functions

Table II summarizes the set of operations described in section 5 and their effect on the document structure: some of them modify the index of the targeted elements, while others modify the element type, the inner structure of the tree or the content of the leaf text nodes.

For the eight operations defined earlier, we must define sixty four transformations. Here we only provide a brief overview of our transformations. Interested reader can refer to the full set of transformation functions available at https://gist.github.com/oster/04ca4fc1aaea7de58700.

Table III summarizes the combined effect of the different classes of operations: the ones that modify elements index, elements type, the inner structure of the tree or the content of the leaf text nodes. Columns indicate the classes of operations to be transformed, while lines indicate the classes of operations against which the transformations are performed.

DOI: 10.1002/cpe

structure modification of the operation to be transformed according to	structure modification performed by the operation to be transformed			
	element index	element type	inner tree structure	leaf content
element index	update of the element index	.	.	.
element type	update of the element index	.	.	.
inner tree structure	update of the element index	.	update of the path to the leaf	.
leaf content	update of the element index	.	update of the path to the leaf	update of the position in the leaf

Table III. Summary of the transformation functions.

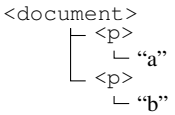
In what follows we briefly explain all transformation functions with a focus on the complex ones that required a careful analysis for the definition of combined effects and those that result not into a single operation to be executed, but into a sequence of operations.

6.3.1. Transformation of NewElement When a `NewElement` operation is transformed, the insertion index of the element is updated if some elements (e.g. paragraphs, headings, etc.) were concurrently moved, merged, split or inserted. The parameter *`siteId`* is used to break ties. This behavior is the same as the one in a classical OT editor (with linear representation, and insert, delete, update operations only). A corner case occurs when the new element is inserted between two concurrently merged elements. Here the choices are to insert the new element between the merged ones or to append the new element after the merged ones. We have chosen the second choice that respects the intention of merging two elements as well as insertion of the new element at a position that changed due to concurrent operations. An example illustrating our choice is provided in Figure 10.

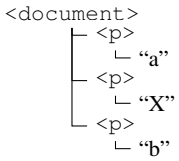
6.3.2. Transformation of MergeElement `MergeElement` operations are transformed according to the same principles. However there are two additional cases to deal with: If one element was inserted between the merged ones and if one of the merged elements was concurrently moved. In the first case, when a new element was inserted between two merged elements, the two initial elements are merged and the inserted new element is appended after the merged elements. This case is illustrated in Figure 10. In the second case, if one of the merged elements was concurrently moved to a certain location, the other element to be merged is moved as well to that location and merged to the initial element. This case is illustrated in Figure 11.

6.3.3. Transformation of Split `Split` takes as parameters the path to the text node to be split and the position inside the text node. Difficult transformations of `Split` are those against operations that modify path and position. For instance, in Figure 12, two concurrent operations are executed: `Split(1, [0,0])`, i.e. split after “a”, and `Style(0, 2, [0,0], underlined, true)`, i.e. underline “abc”. The `Style` operation splits the text node of the first paragraph into two subtrees. The first subtree contains the style node `<span style='underlined'>` with the text node “abc” as a child. The second subtree contains the text node “de”. The transformation of the operation `Split(1, [0,0])` against `Style(0, 2, [0,0], underlined, true)` leads to `Split(1, [0,0,0])` as the node to be split by the original operation `Split(1, [0,0])` is in fact the node `[0,0,0]`. For the sake of clarity, the

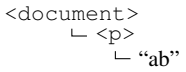
DOI: 10.1002/cpe



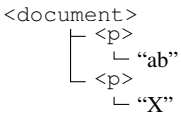
(a) initial document.



(b) concurrent insertion of a new paragraph.

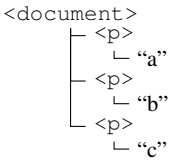


(c) concurrent merge of the two initial paragraphs.

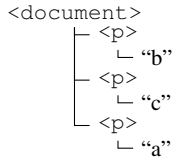


(d) result after reconciliation of the two concurrent operations.

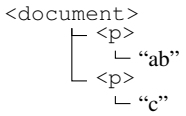
Figure 10. Document model for two concurrent operations NewElement and MergeElement.



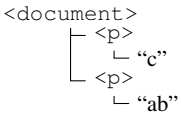
(a) initial document.



(b) concurrent move of the first paragraph at the end of the document.



(c) concurrent merge of the two initial paragraphs.



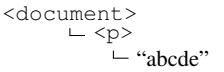
(d) result after reconciliation of the two concurrent operations.

Figure 11. Document model for two concurrent operations MergeElement and MoveElement.

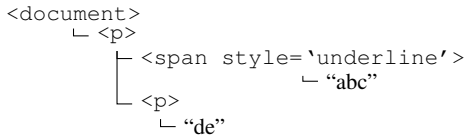
last parameter in the definition of `Split` as well as the last three parameters in the definition of `Style` were omitted.

6.3.4. Transformation of MoveElement Transformation of `MoveElement` operation is similar to that of `NewElement` and `MergeElement` operations. The corner case appears when merge or

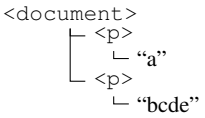
DOI: 10.1002/cpe



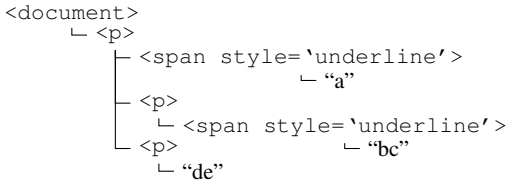
(a) initial document.



(b) concurrent underline of the text “abc”.



(c) concurrent split.



(d) result after reconciliation of the two concurrent operations.

Figure 12. Document model for two concurrent operations Split and Style.

move involving the same elements concurrently occurred with a `MoveElement` operation. This case was illustrated in Figure 11.

6.3.5. Transformation of InsertText and DeleteText `InsertText` operation is close to `Split` operation, as it carries a position `pos` and a path `path` parameters, but does not change the structure of the tree. To transform an insertion operation, the new path is computed according to concurrent `NewElement`, `MergeElement`, `Split`, `MoveElement` and `Style` operations and the new position is computed regarding concurrent `Split`, `InsertText`, `DeleteText` and `Style` operations. The same principle is applied for `DeleteText` operation except that a tie breaker parameter is not needed, since two concurrent deletions of the same character never conflict: the character is deleted anyway.

6.3.6. Transformation of Style `Style` is the most complex operation to manage. It can split a node into three others, while adding a child to one of them. For the sake of simplicity we only explain how transformation is performed when two concurrent `Style` operations occur. If they do not target the same element, no transformation is performed. Else, if they do not target the same child, the path of the second `Style` operation has to be updated if the first one splits the child into two or three children. Else, i.e. if they target the same child, the two operations may overlap, so the transformation of one `Style` operation may result into two or three `Style` operations. The transformation also has to check if the two styles are compatible. If they are not, `siteId` is used as the tie breaker. The discarded style is applied to take into account the structure modifications.

6.4. Split and Merge

One of the main contributions of this paper is the design of split and merge operations working on the tree structure as well as the definition of the combined effects of these operations with all the other operations in our model. To our knowledge this is the first work dealing with split and merge in collaborative editing. In this subsection we present in more details the `Split` and `MergeElement` operations and transformations involving these operations.

`Split` and `MergeElement` were defined for our needs and behave in a specific way as explained in what follows. Firstly, a `Split` operation splits an element at the deepest level in

DOI: 10.1002/cpe

the tree, and duplicates every upper level elements to create a second branch from the root. For instance, Figure 5 illustrates that the split of `<p>ab<b>cd</b>ef</p>` after `c` leads to `<p>ab<b>c</b></p><p><b>d</b>ef</p>`.

Secondly, we only allow the merge of elements at a depth of one (direct children of the root). It is sufficient in our context, since the aim of our model is to focus on the modifications of the structure of the wiki document. The structure is defined by the first level of the tree model, the other levels being related to content. In our context, a `Split` occurs when one user presses enter with the caret inside an element : the element (paragraph, item...) is divided into two. Similarly, a `MergeElement` occurs when backspace is pressed while being between two elements.

When a `Split` occurs, we need to know whether it initially occurred at the beginning of the element. The reason is that, if the split is at the beginning of the element, this element is not split but moved. If the `Split` position is inside an element, the element is split and then the right part is moved.

Without this information, an issue occurs when a `Split` is concurrent to a `DeleteText`, the deleted text being precisely the left part of the split. Since we do not remove any branches of the tree, the deletion of the left part of the split (after the split was executed) leads to an empty text node. On the other site (deletion first, then remote split received), the transformed operation splits the node at position zero. The default behaviour of a split at position zero is not to create a new node, so without a reminder that the node was split at its local site - and therefore has to be split again - the empty text node is not created and models diverge.

When an insertion concurrently occurs at the same position of a split, the text is inserted in the right part of the split node. The other choice could have been troublesome, since no left part is created if the position of the split is zero.

When merging two elements, the position of the right element to be merged (the position in the children list of the root) is provided and the children of this element are appended to the children of the left element of the merge (the right one is discarded). We store in the operation's attributes the initial number of children in the left node, before the merge. This parameter is needed to transform any operation that updates the right node. The path of the transformed operation has its first level reduced by one and its second one increased by the number of left children. For instance, the transformation of `InsertText(2,[1,1],a,3)` against `MergeElement(1,p,2)` results into `InsertText(2,[0,2],a,3)`.

The `Split` and `MergeElement` operations are a first step toward a generic insert, delete, split, merge, move set of operations and transformations for tree structures.

7. DISCUSSION

Usually editors provide lots of functionalities to their users, and need to work with a complex underlying structure. When adapting these editors for collaborative work, reducing everything to the simplest collaborative algorithm and mechanism seems inappropriate, as lot of the initial semantic will be lost during the process. We focused on wikis on this paper, the eight operations we extracted express (a subset of) the functionalities of a wiki, what a user really can do with a wiki. These operations offer support for adequate transformations that respect the semantic of the document. User intention is also better respected. To understand exactly what the intention of a user is seems impossible, but by enhancing the set of operations we offer users more ways to express their actions and thus more chance to find the automatic conflict resolution appropriate.

Our approach requires design of several complex transformations, losing generality. However, it offers a set of benefits including reduction in the number of operations needed to describe a task and therefore a reduction of the size of the log and an improvement in the computation of transformations. Indeed, the proposed operations replace a set of operations on characters as follows:

- `NewElement(int index, String content, int siteId)` operation replaces the set of operations corresponding to the insertion of each character describing the new created element. For instance, the new element `<h1>` is generated in our case by a single

DOI: 10.1002/cpe

operation, while in existing approaches working on characters is generated by using 4 operations corresponding to the characters <, h, 1 and >.

- `UpdateElement(int index, String content, int siteId)` operation replaces the set of operations corresponding to the deletion of each character composing the old content of the element and the insertion of each character of the new content of the element. For instance, replacing the old content of an element `<h1>` by `<p>` is generated by a single operation in our case, while in existing approaches on characters it would require in an optimal case 3 operations corresponding to the deletion of the characters `h`, `1` and an insertion of the character `p`.

- `MoveElement(int origin, int dest, int siteId)` operation replaces the set of operations corresponding to the deletion of each character composing the structure of the old element and the insertion of each of the deleted characters to the new position. For instance, the move of the element `<p><span style='bold'>ab</p>` is done by means of a single operation in our case and by means of 56 operations for deleting each character of the element description and reinserting these characters to a new specified position.

- `MergeElement(int index, String content, int leftChNb)` operation replaces the set of operations corresponding to the deletion of all space characters between the elements to be merged.

- `Split(int pos, int[] path, boolean splitLeaf)` operation replaces the operation of insertion of a space character that triggers the split of the element.

- `InsertText(int pos, int[] path, char c, int siteId)` and `DeleteText(int pos, int[] path)` operations replace their corresponding operations for insertion/deletion of the specified character.

- `Style(int start, int end, int[] path, String param, String value, int siteId, boolean splitStart, boolean splitEnd)` operation replaces the set of operations corresponding to the insertion of the characters describing the added style. If the target text for which the style is applied belongs to different nodes, the `span` element has to be duplicated to different locations and in this case the `Style` operation replaces the set of operations describing the duplicated added style by means of character insertions.

As we can see, in our approach we generate a much smaller number of operations than the number of operations generated by the character-based approaches. This reduces significantly the communication traffic.

The complexity of Jupiter algorithm that we use is `O(n)` where `n` is the number of concurrent operations. As in our case the number of operations is reduced compared to the approaches based on characters, we achieve a better complexity and therefore better performances. If the complexity of existing algorithms working on simple operations on characters is `O(`n_1`)`, where n_1 is the number of concurrent operations generated by the k sites, the complexity of our approach is `O(`n_2`)`, where n_2 is the number of concurrent complex operations generated by the k sites with the property that $n_2 << n_1$. Moreover, in the case of simple operations on characters, these operations which are usually insert and delete, modify characters index and their transformations have to compute the updated index. In the case of our approach, operations are classified into ones that modify elements index, elements type, the inner structure of the tree or the content of the leaf text nodes. As operations modify different document structures, most of transformations among these operations need no computation as they return the original operation. For instance, operations that modify uniquely the elements index such as `NewElement` and `MoveElement` do not interfere with operations that modify leaf content such as `InsertText` and `DeleteText`, operations that modify the element type such as `UpdateElement` and operations that modify the inner tree structure such as `Style`.

DOI: 10.1002/cpe

Delays in collaborative editing systems are due to physical communication technology be it copper wire, optical fiber or radio transmission, complexity of various algorithms for ensuring consistency and the type of architectures. While we claim no contribution regarding the physical communication and the client-server architecture where algorithm computations are done both on the client and the server side, our approach reduces the number of transformations that have to be performed by the underlying algorithm. By reducing the complexity of the algorithm, delays are reduced.

In this paper we did not discuss how operations are derived from user inputs. In non WYSIWYG collaborative editing systems which only use insert and delete operations, this mapping is straightforward. For instance, when someone presses the Return key an insertion of the character corresponding to a newline is generated. However, in a WYSIWYG editor, the same input could be interpreted as the insertion of a newline character, the creation of a new paragraph, or the split of a paragraph. The real operation can be in fact inferred from the context. For example, the default behaviour of the Return key is to insert a newline, but if there already exists a newline before this one and the insertion is in the middle of a paragraph, then a split proceeds instead, while if it is at the end of a paragraph a new element is inserted.

8. CONCLUSION AND FUTURE WORK

In this paper we proposed extending wiki systems with real-time collaboration. Users can switch from traditional asynchronous collaboration to the real-time collaboration and vice-versa when they want. We proposed an automatic merging solution adapted for rich content wikis. We presented the model of the wiki document and a set of high level operations that capture user intentions that are defined on this model. We provided an overview of the transformation functions needed for the synchronisation mechanism. Our proposed approach was integrated as an extension of the XWiki system. In this paper we presented our solution specifically for wiki pages, but it can be easily applied to other forms of rich text documents. In the future we plan performing user studies on the proposed synchronisation mechanism. More precisely, we plan to investigate user feedback on the conflict-resolution decisions made by the system.

REFERENCES

1. Ignat CL. Annotation of Concurrent Changes in Collaborative Software Development. *Proceedings of the IEEE International Conference on Intelligent Computer Communication and Processing - ICCP 2008*, Cluj-Napoca, Romania, 2008; 137–144.
2. Ignat CL, Oster G. Awareness of Concurrent Changes in Distributed Software Development. *Proceedings of the International Conference on Cooperative Information Systems - CoopIS'08*, Monterrey, Mexico, 2008; 456–464.
3. Ignat CL, Papadopoulou S, Oster G, Norrie MC. Providing Awareness in Multi-synchronous Collaboration Without Compromising Privacy. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2008*, ACM Press: San Diego, CA, USA, 2008; 659–668.
4. Ignat CL, Oster G, Fox O, Shalin VL, Charoy F. How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking. *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2015*, Springer-Verlag: Oslo, Norway, 2015.
5. Ellis CA, Gibbs SJ. Concurrency Control in Groupware Systems. *SIGMOD Record: Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD'89* May 1989; **18**(2):399–407, doi:10.1145/67544.66963.
6. Sun C, Jia X, Zhang Y, Yang Y, Chen D. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction* March 1998; **5**(1):63–108, doi:10.1145/274444.274447.
7. Imine A, Molli P, Oster G, Rusinowitch M. Proving Correctness of Transformation Functions in Real-Time Groupware. *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2003*, Springer Netherland: Helsinki, Finland, 2003; 277–293, doi:10.1007/978-94-010-0068-0_15.
8. Xia S, Sun D, Sun C, Chen D, Shen H. Leveraging Single-User Applications for Multi-User Collaboration: The CoWord Approach. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2004*, ACM Press: Chicago, IL, USA, 2004; 162–171, doi:10.1145/1031607.1031635.
9. Davis AH, Sun C, Lu J. Generalizing Operational Transformation to the Standard General Markup Language. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2002*, ACM Press: New Orleans, LA, USA, 2002; 58–67, doi:10.1145/587078.587088.

DOI: 10.1002/cpe

10. Puttaswamy KP, Marshall CC, Ramasubramanian V, Stuedi P, Terry DB, Wobber T. Docx2Go: Collaborative Editing of Fidelity Reduced Documents on Mobile Devices. *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys 2010*, ACM Press: San Francisco, CA, USA, 2010; 345–356, doi:10.1145/1814433.1814467.
11. Weiss S, Urso P, Molli P. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, IEEE Computer Society: Montreal, QC, Canada, 2009; 404–412, doi:10.1109/ICDCS.2009.75.
12. Nichols DA, Curtis P, Dixon M, Lamping J. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. *Proceedings of the 8th Annual ACM Symposium on User interface and Software Technology - UIST '95*, ACM Press: Pittsburgh, PA, USA, 1995; 111–120, doi:10.1145/215585.215706.
13. Vidot N, Cart M, Ferrié J, Suleiman M. Copies Convergence in a Distributed Real-Time Collaborative Environment. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2000*, ACM Press: Philadelphia, PA, USA, 2000; 171–180, doi:10.1145/358916.358988.
14. Zafer AA. NetEdit: A Collaborative Editor. Master's Thesis, Virginia Polytechnic Institute and State University, VA, USA 2001.

AUTHORS' BIOGRAPHIES

Claudia-Lavinia Ignat is permanent researcher at Inria. She obtained a PhD in Computer Science from ETH Zurich, Switzerland in 2006. Her research areas are distributed collaborative systems with a focus on consistency maintenance, group awareness, security and trust issues and more recently user studies. She developed several algorithms for maintaining consistency based on operational transformation and CRDT approaches for textual, hierarchical and graphical documents and wiki pages as well as awareness mechanisms in the context of software engineering, collaborative editing of textual documents and of web pages. She proposed a contract-based collaboration model as a support for a soft security mechanism in collaborative editing where access is given first to data without control but with restrictions that are verified a posteriori. In this collaboration model she proposed a mechanism for establishment of trusted logs relying on hash-chain based authenticators. She proposed an evaluation with users of real-time constraints in collaborative editing systems. She was involved in several research projects.

Gérald Oster is an Associate Professor at Telecom Nancy, University of Lorraine since 2006. He has an expertise in distributed collaborative systems with a focus on content replication mechanisms and their applicability. He received his Ph.D. in Computer Science from Nancy University in 2005. During his PhD, he worked on verification of correctness of a family of optimistic replication mechanisms (operational transformation) dedicated to collaborative editing. He proposed a framework based on an automated theorem prover and several sets of verified transformation functions for multiple data types. He worked on the design and the implementation of a universal file synchronizer. He is one of the father of the CRDT approach as he participated in the design of the WOOT algorithm which initiated researches on these distinctive data structures. He is currently investigating the limitations and the applicability in diverse domains of these novel replicated data structures. Gérald is or was involved in several research projects and participated in several technology transfer oriented projects.

Luc André received an Engineering degree in Software Engineering from ESIAL at the University of Lorraine in 2010. He is currently a Ph.D. student at the University of Lorraine in Inria Coast project-team. His main research interest is optimistic replication for distributed collaborative editing systems with a focus on user intentions preservation.