Available online at www.sciencedirect.com
ScienceDirect
Computers in Industry 57 (2006) 758–767
COMPUTERS IN INDUSTRY
www.elsevier.com/locate/compind

Customisable collaborative editing supporting the work processes of organisations

Claudia-Lavinia Ignat *, Moira C. Norrie

Institute for Information Systems, ETH Zurich, CH-8092 Zurich, Switzerland

Accepted 27 April 2006
Available online 30 June 2006

Abstract

The development of customisable working environments that not only manage information and communication, but also support the work processes of organisations is very important. In this paper we describe a collaborative approach that offers customisation in terms of the modes of collaboration that can be used alternatively in different stages of a project, i.e. synchronous and asynchronous, for two main classes of documents, textual and graphical. Moreover, by using a hierarchical structure for representing the documents, we offer a flexibility of working at different granularity levels. We highlight the fact that the real-time and asynchronous modes of communication use the same basic mechanisms for maintaining consistency in the collaborative environments.

Keywords: Customisable collaborative editing; Synchronous/asynchronous communication; Consistency maintenance; Operation transformation; Operation serialisation

1. Introduction

Collaboration is an important aspect of any team activity and hence of importance to any organisation – be it business, science, education, administration, political or social. Increasingly, collaborative work activities are centred on a virtual space that manages the relevant information and communication between the members of the team. Wide geographical distribution of individuals, possibly across a wide range of time zones, together with the mobility of individuals, increases the problems of collaboration. Therefore the development of collaborative working environments that not only manage information and communication, but also support the actual work processes of organisations is very important.

Collaborative editing systems is a subfield of Computer-Supported Cooperative Work (CSCW) investigating ways to support a group of people editing documents collaboratively over a computer network. The collaboration between users can be synchronous or asynchronous. Synchronous collaboration means that members of the group work at the same time on the same documents and modifications are seen in real-time by the other members of the group. Asynchronous collaboration means that members of the group modify the copies of the documents in isolation, afterwards synchronizing their copies to reestablish a common view of the data.

In an engineering domain not all users of a group have the same convention and not all tasks have the same requirements. Our goal is to offer a customisable system that can support collaboration in a range of application domains such as engineering design (architectural design, product data management design) and collaborative writing (news agency, authoring of scientific papers or annotations). We offer customisation in terms of the types of the documents forming the basic unit for collaboration, i.e. graphical and textual, in terms of the modes of collaboration, i.e. synchronous and asynchronous and in terms of the flexibility of working at different levels of granularity, such as paragraph, sentence, word or character in the case of the text documents.

For instance, in the case of collaborative architectural design, support for collaboration needs to be offered both for graphical and text documents. Graphical documents are needed for performing brainstorming sessions consisting of a graphical sketching of the issues to be discussed, the assignments of the tasks, as well as for performing the architectural design itself.

* Corresponding author.
E-mail address: ignat@inf.ethz.ch (C.-L. Ignat).

0166-3615/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.compind.2006.04.005

Text documents are needed for the collaborative writing of the documentation. The support for the two modes of collaboration – synchronous and asynchronous – and the possibility of switching from one mode of collaboration to the other corresponding to the different stages of a project is very important for supporting a work process. In the case of the architectural design, the brainstorming should be performed in real-time by the members of the group because rapid feedback is required while decisions are being taken to create the to-do list or the task assignment. But, in the actual designing phase, the asynchronous mode is required to allow different parts of the architectural design to be developed in isolation. In a later phase, after the design parts are assembled, synchronous and asynchronous modes may be inter-mixed. For example, synchronous communication can be used when real-time collaboration between the members of the group is required to collectively modify the design. On the other hand, in some situations, an expert may want to review the design in isolation and merge any modifications that they make at a later time. In such situations, an asynchronous mode of the collaboration is required.

In our approach we modeled the documents, both text and graphical, using a hierarchical structure. The text documents have been modeled as consisting of paragraphs, each paragraph consisting of sentences, each sentence consisting of words and each word consisting of characters. We also use a tree model for representing the scene of objects in the graphical documents, where groups are represented as internal nodes, while simple objects are represented as leaves, a group being able to contain other groups or simple objects.

The tree structure of the document offers not only increased efficiency, but also flexibility of working at different levels of granularity. To illustrate this feature, we will consider the case of asynchronous collaborative text editing, making a comparison with the well known versioning systems such as RCS [19] and CVS [1]. The basic unit of conflict used by the above mentioned versioning systems is fixed, being the line, meaning that the changes performed by the two users are in conflict if they refer to the same line. Concurrent changes on different lines are merged automatically. In our approach, we work with different semantic units, such as paragraphs, sentences, words or characters, in this way offering a more flexible approach for defining and resolving the conflicts. Rules for defining the conflict such as the one stating that concurrent changes performed in the same sentence are conflicting can be easily specified in our approach.

In this paper we present the synchronous and asynchronous modes of collaboration for two main classes of documents, namely, textual and graphical as a flexible way of supporting collaboration in the work processes of organisations. We show that the same basic ideas for maintaining consistency that have been used for real-time communication can also be used to support the asynchronous mode of collaboration. Throughout the paper, we highlight the improvements that are achieved by using a structured model of the document in comparison with the linear model in terms of efficiency and flexibility of working at different levels of the document, enabling conflicts to be defined and resolved in an easy way.

The paper is structured as follows. In the next section we give an overview of the main principles for consistency maintenance in the case of real-time collaborative text and graphical editors. In Section 3 we go on to describe the asynchronous mode of collaboration for both text and graphical editing. In Section 4 we compare our work with some related work. Concluding remarks are presented in the last section.

2. Real-time collaborative editing

In this section we give an overview of the techniques for maintaining consistency that underly the functionality of the collaborative text and graphical editors that we have implemented and illustrate how they have been applied for the real-time communication.

A replicated architecture, where users work on local copies of the shared documents and instructions are exchanged by message passing, has been used in both cases of the text and graphical collaborative editors. The users are able to concurrently edit any part of the shared document. As previously mentioned, both the textual and graphical documents have been modeled using a hierarchical structure.

2.1. Text editing

Operation transformation mechanisms [3] have been applied to maintain the consistency of copies of shared documents in the real-time collaborative text editor that we have developed.

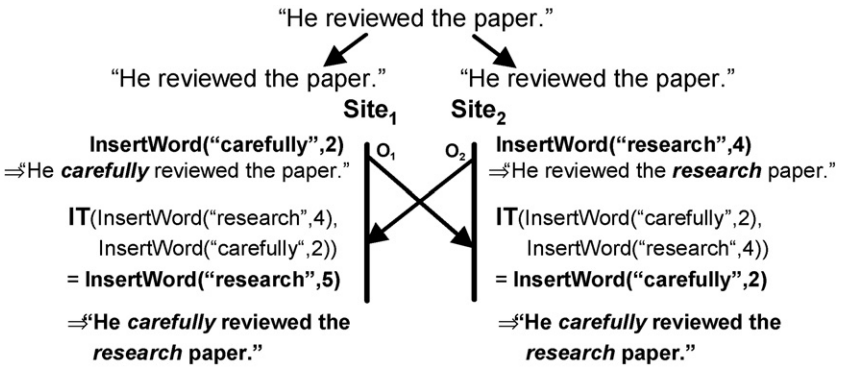

Fig. 1. Operation transformation.

Operation transformation allows local operations to be executed immediately after their generation and remote operations need to be transformed against the other operations. The transformations are performed in such a manner that the intentions of the users are preserved and, at the end, the copies of the documents converge.

Fig. 1 illustrates a very simple example of the operation transformation mechanism. Suppose the shared document contains a single sentence "*He reviewed the paper.*" Two users, at $Site_1$ and $Site_2$, respectively, concurrently perform some operations on their local replicas of the document. $User_1$ performs operation O_1 of inserting the word "*carefully*" as the 2nd word into the sentence, in order to obtain "*He carefully reviewed the paper.*" Concurrently, $User_2$ performs operation O_2 of inserting the word "*research*" as the 4th word into the sentence, in order to obtain "*He reviewed the research paper.*" Executing the operations in their generated form at remote sites will not preserve the intentions of the users and the copies of the documents will not converge. The operations need to be transformed when they are executed at a remote site.

The simplest form of operation transformation is the *Inclusion Transformation* – $IT(O_a, O_b)$, which transforms operation O_a against a concurrent operation O_b such that the impact of O_b is included in O_a.

In the previous example, at $Site_1$, when operation O_2 arrives, it needs to be transformed against operation O_1 to include the effect of this operation. Because the concurrent operation O_1 inserted a word before the word to be inserted by operation O_2, operation O_2 will become an insert operation of the word "*research*" at position 5, the result being "*He carefully reviewed the research paper*", satisfying the intentions of both users. At $Site_1$ and $Site_2$, in the same way, operation O_1 needs to be transformed against O_2 in order to include the effect of O_2. The position of insertion of O_1 does not need to be modified in this case because operation O_2 inserted a word to the right of the insertion position of O_1. Therefore, the transformed operation O_1 has the same form as the original operation O_1. We see that the result obtained at $Site_2$ respects the intentions of the two users and, moreover, the two replicas at the two sites converge.

Another form of operation transformation called *Exclusion Transformation* – $ET(O_a, O_b)$ transforms O_a against an operation O_b that precedes O_a in causal order such that the impact of O_b is excluded from O_a.

Most real-time collaborative editors relying on existing operation transformation algorithms for consistency maintenance use a linear representation for the document, such as a sequence of characters in the case of text documents. All existing operation transformation algorithms keep a single history of operations already executed in order to compute the proper execution form of new operations. When a new remote operation is received, the whole history needs to be scanned and transformations need to be performed, even though different users might work on completely different sections of the document and do not interfere with each other. Keeping the history of all operations in a single buffer decreases the efficiency.

In [6] we proposed a consistency maintenance algorithm called treeOPT relying on a tree representation of documents where the document has different levels of granularity: document, paragraph, sentence, word and character, the highest level being the document. The hierarchical representation of a document is a generalisation of the linear representation (such as a sequence of characters in the case of text documents) used by most real-time collaborative editors relying on existing operation transformation algorithms. Each node in the tree representation of the document will keep a history of insertion and deletion operations associated with its child nodes. An example for the tree structure of the document is illustrated in Fig. 2.

An operation is specified by its type, the positions for the upper levels, starting with the paragraph level and ending with the level of the operation, and the content of the operation. For

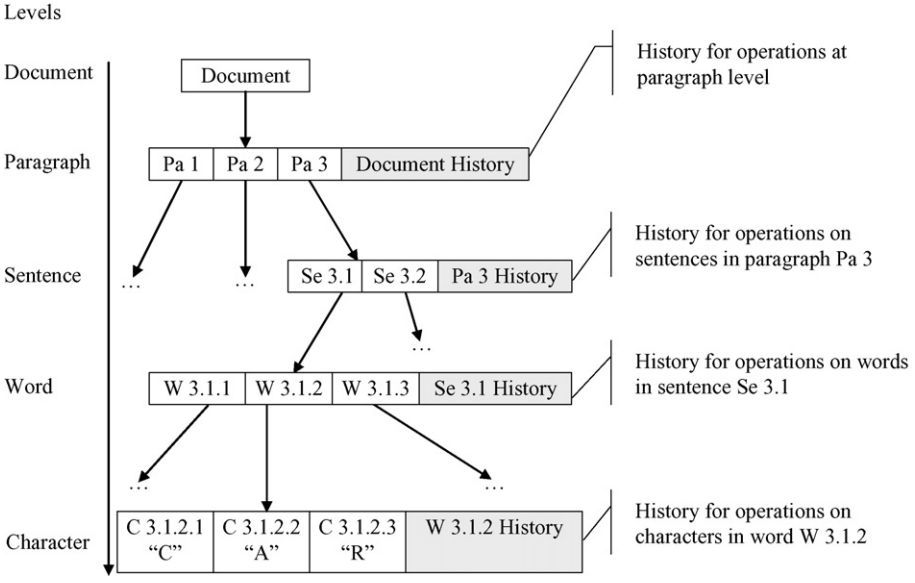

Fig. 2. Tree structure of the document.

instance, *InsertWord*(3,1,2,"*love*") denotes an operation that inserts the word "*love*" into paragraph 3, sentence 1, at word position 2. In what follows we are going to shortly present the principles of functionality of the treeOPT algorithm, i.e. the integration of a remote operation into the history buffer distributed throughout the tree.

A remote operation is integrated into its corresponding history buffer by recursively applying the operation transformation mechanism starting with the document level and continuing with the other lower levels of the document till the level that is equal with the level of the operation is reached. At the beginning the operation is transformed against the document history buffer containing the operations that insert or delete whole paragraphs by considering the paragraph number the operation is referring to. The result of this transformation is the position of the paragraph in whose subtree the operation belongs. In the case that the level of the operation is the paragraph, the integration procedure stops. Otherwise, the procedure is recursively applied by considering the history buffer of the paragraph whose number was found in the previous step and which contains the operations that insert or delete whole sentences. The process is repeated till the current level in the integration process becomes equal to the level of the operation. At the end of the integration procedure the right positions for the semantic units the operation is referring to (paragraph, sentence, etc.) are obtained and the operation is stored in the corresponding history buffer.

Compared to the other operation transformation approaches that use a linear structure for representing the documents, the tree OPT algorithm offers better efficiency and flexibility of the granularity, the users being able to work at paragraph, sentence, word or character level. The algorithm applies the same basic mechanisms as existing operation transformation algorithms recursively over the different document levels (paragraph, sentence, word and character) and it can use any of the operation transformation algorithms relying on linear representation.

2.2. Graphical editing

In the object-based collaborative graphical editor developed in our group [4], the shared objects subject to concurrent accesses are the graphic primitives such as lines, rectangles, circles and text boxes. The operations operating on these primitives are *create, delete*, *setColor*, *setBckColor*, *translate*, *scale*, *setZ* (setting the depth of the target object/group), *setText*, *group* and *ungroup*.

Two types of conflicting operations have been identified: real conflicting and resolvable conflicting operations.

Real conflicting operations are those conflicting operations for which a combined effect of their intentions cannot be established. We have defined that a pair of operations is real conflicting in the case that a serialisation order of execution of these operations cannot be obtained to preserve the intentions of the operations. By executing the two operations in any order, either the effect of the second executed operation will completely make invisible the effect of the first executed operation or the execution of the first operation will not make possible the execution of the second operation. An example of real conflicting operations is the case of two concurrent operations both targeting the same object and changing the colour of that object to different values.

Resolvable conflicting operations are those conflicting operations for which a partial combined effect of their intentions can be obtained by serialising those operations. Consequently, ordering relations can be defined between any two concurrent operations. Any two resolvable conflicting operations can be defined as being in right order or in reverse order. An example of operations that are in a resolvable conflict are the concurrent operations of changing the colour of a group and the ungroup operation applied on that group. A combination of the intentions of the two concurrent operations can be obtained by executing first the change colour operation and afterwards the ungroup operation. In the case that the ungroup operation is executed first, the change colour operation will refer to a non-existent identifier.

For maintaining consistency in the case of the object-based graphical documents, a serialisation mechanism has been applied rather than the operation transformation principle as in the case of the text editor. In what follows we briefly describe the principle of operation serialisation. When a remote operation arrives at a site, all the operations from the local log starting with the first operation that is concurrent and in conflict with the remote operation and ending with the last operation in the log will be undone. The idea of operation serialisation is the re-execution of the undone operations in such an order that achieves a partial combined effect of the intentions of the users. The operations from the local log that are in a real conflict with the remote operation, as well as the operations that are in a right order resolvable conflict and the operations that are in a reverse order resolvable conflict are saved in separate lists. In the case that there are no operations in real conflict with the remote operation, the undone operations are re-executed in such a way that the operations that are in a right order resolvable conflict with the remote operation are re-executed before the remote operation and the operations that are in a reverse order resolvable conflict are re-executed after the execution of the remote operation. In the case that there are operations in a real conflict relation with the remote operation, either the remote operation or all the local real conflicting operations will be executed, depending on the policy chosen by the application. For instance, in the case of conflict between two operations, the application might choose to execute none of the conflicting operations or it might choose to execute one of the operations based on some criteria, such as priorities for the users. In the case that the remote operation has to be cancelled, all the undone

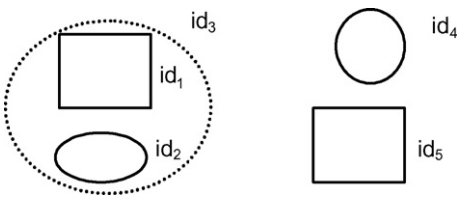

Fig. 3. The initial scene of objects.

operations are re-executed in their order from the log. In the case that the local conflicting operations have to be cancelled, but the remote operation has to be executed, all the undone operations from the log except the real conflicting operations are re-executed such that the operations that are in a right order resolvable conflict are executed before the remote operation and the operations that are in a reverse order resolvable conflict are executed after the remote operation.

For a better understanding of the algorithm, we illustrate its functionality by means of an example. Consider the scene of objects shown in Fig. 3. It consists of a group having the identifier id_3 composed of two objects with the identifiers id_1 and, respectively, id_2. The scene of objects contains also other two objects with the identifiers id_4 and id_5.

Suppose three users concurrently edit this graphical document, as illustrated in Fig. 4. The user at $Site_1$ groups the group with the identifier id_3 with the object having the identifier id_5 and changes the colour of the newly created group into *red*. Concurrently, the user at $Site_2$ wants to ungroup the group id_3 and to change the colour of the object id_1 to *blue*. At the same time, the user at $Site_3$ performs a grouping of group id_3 with the object id_4. In the case of real conflicting operations we consider that the operation generated from a site with the highest priority wins the conflict. Let us consider that $Site_1$ has priority 3, $Site_2$ has priority 2 and $Site_3$ has priority 1. Let us analyse the application of the algorithm at each of the three sites.

At $Site_1$, after O_1 and O_2 are executed, HB = [O_1, O_2]. When the causally ready operation O_3 is received, the algorithm detects that O_3 is concurrent and in conflict with O_1, so O_2 and O_1 are undone. The conflict between O_1 and O_3 is a right order resolvable conflict. So, O_1 and O_2 are redone and afterwards O_3 is executed. So, HB = [O_1, O_2, O_3]. When the causally ready operation O_4 is received, the algorithm detects that O_4 is concurrent and in conflict with O_2, so O_3 and O_2 are undone. The conflict between O_2 and O_4 is a right order resolvable conflict, there are no operations in a reverse order with O_4, so, O_2 and O_3 are redone and O_4 executed. After the execution of O_4, HB = [O_1, O_2, O_3, O_4]. When O_5 is received, the real conflict between O_5 and O_1 and the reverse order resolvable conflict between O_3 and O_5 are detected. All operations from HB are undone. Because the priority of O_1 is higher than the priority of O_5, O_5 becomes NOP

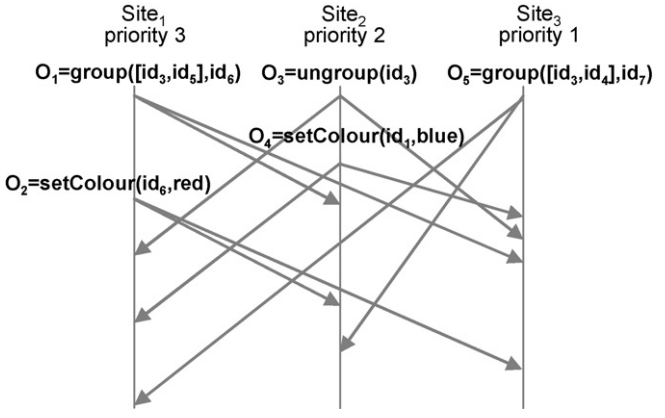

Fig. 4. Scenario – the functionality of the algorithm.

operation, i.e. it is cancelled, the operations from HB are redone and O_5 will be added at the end of the history buffer. So, finally HB = [O_1, O_2, O_3, O_4, O_5 = NOP].

At $Site_2$, after O_3 and O_4 are performed, HB = [O_3, O_4]. When O_1 is received, the algorithm detects that O_3 is in a reverse order resolvable conflict with O_1, so O_4 and O_3 are undone. O_3 is in a reverse order conflict with O_1, so O_4 is redone, O_1 executed and O_3 redone. So, HB = [O_4, O_1, O_3]. When O_2 is received, the reverse order resolvable conflict between O_4 and O_2 is detected, O_3, O_1 and O_4 are undone. O_1 and O_3 are redone, O_2 executed and O_4 redone, the history buffer becoming HB = [O_1, O_3, O_2, O_4]. When O_5 is received, the real conflict between O_5 and O_1 is detected and all operations from HB are undone. Because the priority of O_1 is higher than the priority of O_5, O_5 becomes NOP, the operations from HB are redone and O_5 will be added at the end of the history buffer. So, finally HB = [O_1, O_3, O_2, O_4, O_5 = NOP].

At $Site_3$, after the execution of O_5, HB = [O_5]. When O_4 is received, its execution will be delayed, because it is not a causally ready operation, i.e. there are other operations that have been generated before the generation of O_4 at $Site_2$. When O_3 is received, the algorithm detects that O_5 and O_3 are in a right order conflict, O_5 is undone, then redone, followed by the execution of O_3. Afterwards, O_4 becomes ready for execution. No conflict between O_4 and the operations in the history buffer is detected, so O_4 is added at the end of the history buffer. So, HB = [O_5, O_3, O_4]. When O_1 is received, the real conflict between O_1 and O_5 is detected, as well as the reverse order resolvable conflict between O_3 and O_1. The operations in the history buffer are undone, O_5 becomes NOP, O_4 is redone, O_1 is executed, followed by the re-execution of O_3. So, HB = [O_5 = NOP, O_4, O_1, O_3]. When O_2 is received, the reverse order resolvable conflict between O_2 and O_4 is detected and therefore, O_3, O_1 and O_4 are undone. Afterwards, O_1 and O_3 are redone, O_2 is executed and O_4 is redone. So, HB = [O_5 = NOP, O_1, O_3, O_2, O_4].

3. Asynchronous collaborative editing

In this section we are going to analyse in turn the techniques underlying the asynchronous communication in the case of the collaborative text and graphical editing.

All configuration management tools support the Copy/Modify/Merge technique. This technique consists basically of three operations applied on a shared repository storing multi-versioned documents: *checkout*, *commit* and *update*. A *checkout* operation creates a local working copy of the document from the repository. A *commit* operation creates a new version of the corresponding document in the repository by validating the modifications done on the local copy of the document. This operation is performed only if the local document is up-to-date, i.e. the repository does not contain a new version committed by other users. An *update* operation performs the merging of the local copy with the last version of that document stored in the repository.

State-based merging and operation based merging have been identified as two different approaches for performing the merging. *State-based merging* [19,1] uses only information

about the states of the documents and no information about the evolution of one state into another. The merging process involves computing the difference between the two states of the document by comparing their states. This difference is then applied to one of the documents in order to generate the document that represents the result of merging. *Operation-based merging* [11,16] keeps a log of operations performed between the initial state and the current state of the document as a representation of the difference between the two states of the document. Afterwards, the operations performed on the working copy of the document and recorded in the log are re-executed on the other copy.

3.1. Text editing

In this subsection we are going to briefly present the merging algorithms applied for the linear representation of documents as described in [16] and then present a generalisation of the merging algorithms for the hierarchical structure of the documents. We are going to highlight the fact that the same basic mechanisms used for the real-time editing have been used for the asynchronous editing and the advantages of using the hierarchical structure versus the linear structure for the asynchronous editing are the same as for the real-time editing.

3.1.1. Linear-based merging

We start by analysing the synchronisation of the copy of the document from the private workspace against the repository in the case of a linear representation of the document.

If a user wants to commit the working copy into the repository, he sends a request to the repository. In the case that the process of another concurrent committing request is under current development, the request is queued in a waiting list. Otherwise, if the base version (the last updated version from the repository) in the private workspace is not the same as the last version in the repository, the repository sends to the site a negative answer that the site needs to update its working copy before committing. In the case that the base version from the working space is the same as the last version in the repository, the site will receive a positive answer. Afterwards, the site sends to the repository the log of operations representing the difference between the last version in the repository and the current copy from the workspace and increases the base version number. Upon receiving the list of operations, the repository stores the operations from the log and updates the number of the latest version.

In the checkout phase, the local workspace is emptied and all the operations from the repository representing the delta between the version of the document the user wants to work on and the initial version of the document are executed into the local workspace of the user.

In the updating phase, the merging algorithm has to be performed for the following scenario. The local user started working from version V_k on the repository but cannot commit the changes because meanwhile the version from the repository has been updated to version V_{k+n}. Let us denote by *LL* the list of operations executed by the user in their local workspace and by *DL* the list of operations representing the delta between versions V_{k+n} and V_k. The commit can be performed only if the context of the local operations is modified so that it is no longer version V_k, but version V_{k+n}.

Two basic steps have to be performed. The first step consists of applying the operations from *DL* on the local copy of the user in order to update the document to version V_{k+n}. The operations from the repository, however, cannot be executed in their original form since the local context differs from the remote context in which these operations were generated. Consequently, the remote operations have to be transformed in order to include the effect of all the local operations before they can be executed in the user workspace. The second step consists of transforming the operations in *LL* in order to include the effects of the operations in *DL*, i.e. changing the context of the local operations from version V_k to version V_{k+n}. The list of the transformed local operations will represent the new delta into the repository.

Some of the operations from the list *DL* are in conflict with some of the operations in the list *LL*, so a way for defining and resolving the conflicts has to be specified. Our solution for dealing with conflicts in a flexible way follows the ideas described in [14]. According to a specific application domain, a set of rules define the type of conflicts between operations and a function shows if there is conflict between any two operations in a certain context. For resolving the conflicts, an automatic solution can be provided or human intervention can be required for choosing a solution for the conflict.

Consider that operation O_i belonging to the list *DL* is in conflict with an operation from *LL*. In this case O_i cannot be executed in the local workspace. All operations following it in the list *DL* need to exclude its effect from their context, realised by the transposition of O_i towards the end of the list *DL* following the same procedure as described in [18]. The fact that O_i is in conflict with an operation from *LL* and could not be executed in the private workspace needs to be included into the delta as its inverse in order to cancel the effect of O_i. For instance, the inverse of the operation of inserting the word "*love*" in paragraph 2, sentence 3 at the 4th position, *InsertWord*(2,3,4,"*love*") is *DeleteWord*(2,3,4,"*love*").

3.1.2. Hierarchical-based merging

We continue by presenting the generalisation of the merge algorithm described in the previous section for working on a hierarchical structure of the document.

The *commit* phase for the Commit/Modify/Merge paradigm in the case of the tree representation of the documents follows the same principles as in the case of the linear representation. The hierarchical representation of the history of the document is linearised by using a breadth-first traversal of the tree. In this way, the first operations in the log will be the ones belonging to the paragraph logs, followed by the operations belonging to the sentence logs and finally the operations belonging to the word logs. The *checkout* phase is identical to the one described for the linear representation of the documents.

The *update* procedure achieves the actual update of the local version of the hierarchical document with the changes that have been committed by other users to the repository and kept into the remote log. The remote log contains a linearisation of the

logs that were initially part of a tree document structure. The merging for the hierarchical structure of the documents is a generalisation of the linear merging for the levels of the document tree. It has as objective the computing of a new delta to be saved on the repository, i.e. the replacement of the local log associated with each node with a new one which includes the effects of all non conflicting operations from the remote log and the execution of a modified version of the remote log on the local version of the document in order to update it to the version on the repository.

The merging has to be achieved between the remote log containing the operations on the repository and the local logs associated with the nodes in the tree. The update procedure is repeatedly applied for each level of the document starting from the document level. The first step of the merging algorithm consists in the selection of those operations from the remote log that have the level identical with the level of the current node, composing the *remote level log*. Note that the level of an operation is equal to the level of the node in whose history the operation is kept. The operations belonging to the *remote level log* are eliminated from the remote log. Because during the update process applied for the previous upper levels of the document some nodes of the tree might get deleted or inserted, the operations in the local log of the current node need to adapt their positions to correspond to the current position in the tree. Afterwards, the basic merging procedure for linear structures is applied to merge the local log of the current node with the *remote level log*. As a result of the merging procedure, the *new remote log* representing the operations that need to be applied on the local document and the *new local log* representing the operations to be saved on the repository are computed. After the *new remote log* is applied locally, the operations from the initial remote log need to be divided among the children of the current node and the merging procedure recursively called for each child. But, before the distribution of the operations from the remote log among the child nodes, the operations in the remote log have to adapt their positions according to the operations in the *new local log* that are of a higher level. For instance, consider the case that the initial remote log is [*InsertParagraph*(3,"..."), *InsertSentence*(4,2,"...")] and as a result of the merging at the paragraph level the new local log contains the operation *InsertParagraph*(1,"..."). Because the *new local log* will be saved in the repository, the operation *InsertSentence*(4,2,"...") needs to change the position corresponding to the paragraph, such that the paragraph number becomes 5. After the operations from the remote log are transformed, they are added to the remote logs of the corresponding children of the current node and the procedure is recursively applied for each child and their corresponding remote log.

In what follows we will illustrate the asynchronous communication by means of an example. Suppose the repository contains as version V_0 the document consisting of only one paragraph with one sentence: "*Love mean for him admiration with heart*." Suppose a conflict is defined between two operations concurrently modifying the same word and the policy of merging is that, in the case of conflict, local modifications are kept automatically. Further, assume two users check out version V_0 from the repository into their private workspaces and have as first version in their private workspaces $W_{10} = V_0$ and $W_{20} = V_0$, respectively.

The first user performs the operations O_{11} and O_{12}, where O_{11} = *InsertChar*("*s*",1,1,2,5) and O_{12} = *InsertWord* ("*the*", 1,1,7). Operation O_{11} inserts the character "*s*" in the first paragraph, first sentence, second word in order to correct the misspelling of word "*mean*" for obtaining the version W_{11} = "*Love means for him admiration with heart*." Operation O_{12} inserts the word "*the*" into first paragraph, first sentence, as the 7th word in order to obtain the version W_{12} = "*Love means for him admiration with the heart*." The second user performs the operation O_{21}, O_{21} = *InsertSentence*("*He just admired her by loving with the mind*.",1,2) and O_{22} = *InsertChar*("*t*",1,1,2,5) in order to obtain W_{22} = "*Love meant for him admiration with heart. He just admired her by loving with the mind*."

Suppose that both users try to commit, but $User_1$ gets access to the repository first, while $User_2$'s request will be queued. After the commit operation of $User_1$, the last version in the repository will be V_1 = "*Love means for him admiration with the heart*." DL_{10} representing the difference between V_1 and V_0 in the repository is obtained as a result of the linearisation of the history buffer distributed throughout the tree, $DL_{10} = [O_{12}, O_{11}]$.

When $User_2$'s request is processed, the repository sends to $User_2$ a message to update the local copy. For updating the local copy of the document, the update procedure is applied. The local tree generated at the site of $User_2$ is traversed in a top-down manner. First we analyse the document level history. But there are no remote operations of paragraph level to be merged. The update is then applied for the paragraph level. The document consists of only one paragraph, so the history of paragraph 1 is analysed. There are no remote operations of sentence level, so the processing is applied for the sentence level. The local document contains two sentences, but there are no operations referring to sentence 2, so we will analyse the merging for sentence 1. Operation O_{12} is of word level, and because there are no local operations of word level, O_{12} will keep its original form. The update procedure will be recursively applied for each of the words belonging to sentence 1. We will analyse only the update applied for the second word of sentence 1, since the remote logs corresponding to the other words in the sentence are empty. The merge procedure will be applied between the list of operations consisting of O_{11} and the list consisting of O_{22}. O_{11} and O_{22} are conflicting and according to the assumed policy the local operation will be kept. As result of this merging, the *newLocalLog*, i.e. the list of operations to be transmitted to the repository, will be [$inv(O_{11})$, O_{22}] and the *newRemoteLog*, i.e. the list of operations to be applied on the local copy of the document will be empty. Therefore, the new local version of the document in the workspace of $User_2$ will be "*Love meant for him admiration with the heart. He just admired her by loving with the mind*." This will be also the new version V_2 of the document into the repository after $User_2$ commits. D_{21} will become $D_{21} = [O_{21}, inv(O_{11}), O_{22}]$.

When $User_1$ updates his local version of the document, the update procedure will be called in order to merge the history buffers distributed along the local tree with the corresponding operations from D_{21}. We are not going to describe the steps of the update procedure, but we just make the general remark that according to our algorithm, the operations of higher level granularity do not need to be transformed against the operations of lower level granularity. For instance, in our example, the operation O_{21} of sentence level does not need to be transformed against any of the local operations in the workspace of $User_1$.

We have applied the same principles of operation transformation as for real-time communication. Due to the hierarchical structure of the document, the same improvements as in the real-time communication feature in the asynchronous mode of collaboration. Only few transformations need to be performed when integrating an operation into a log as described above, because the operations in the log are distributed throughout the tree model of the document. Only those histories that are distributed along a certain path in the tree are spanned and not the whole log as in the case of a linear model of the document. Moreover, the function for testing if two operations are conflicting can be expressed more easily using the semantic units used for structuring the documents (paragraphs, sentences, words, characters) than in the case of a linear representation of the document. For example, rules interdicting the concurrent insertion of two different words into the same sentence can be very easily expressed and checked.

3.2. Graphical editing

The principles underlying the asynchronous communication for the collaborative graphical editing follow the Copy/Modify/Merge paradigm, in the same way it has been applied for the text editing. The checkout and commit procedures are identical with the corresponding procedures described for the text editing. In what follows we are going to describe the update procedure.

On the repository the lists of operations representing the difference between the versions of the document are stored. On the client side the following structures are stored: *doc* – the local document, *lastSynchDoc* – the last version from the repository that was integrated into doc, *log* – the list of operations executed locally and not yet committed, representing the difference between *doc* and *lastSynchDoc*. The list of operations received from the repository is merged with the list of operations from the local log by integrating one by one the operations from the repository into the local log. The integration is realised by using the serialisation mechanism applied for the real-time graphical editing. The local document *doc* is modified to include the effects of the operations from the repository, *lastSynchDoc* is updated to the new version of the document from the repository. *log* is the list of operations that is sent to the repository in the case of a commit and represents the difference between the new version of the document that will be committed to the repository and the last committed version. *log* will include first the operations that cancel the effects of the operations from the repository, followed by the operations that represent the merging between the operations from the repository and the local log. A more detailed description of the asynchronous graphical editing approach can be found in [5].

4. Related work

In this paper we have described our approaches for achieving collaborative editing in the case of textual and graphical documents, both for the synchronous and asynchronous modes of communication. We have used the same basic principles for consistency maintenance both for the real-time and asynchronous communication: operation transformation for text editing and operation serialisation for graphical editing. We have modeled both textual and graphical documents using a hierarchical structure, which, as we have seen, yields a set of advantages. All the implemented collaborative tools for textual and graphical editing, both for synchronous and asynchronous communication have been integrated into a customisable collaborative tool for supporting the work processes of organisations.

Each of the approaches that we used for the development of the particular implementation of the synchronous or asynchronous text and graphical editor systems can be related to other works. But, to our knowledge, there is no system that offers customisation for the synchronous and asynchronous collaborative writing for more classes of documents. Therefore, in what follows we are going to compare only our approaches for the particular implementations of text and graphical editors for real-time and asynchronous communications with other existing approaches to these specific problems.

Concerning the real-time collaborative text editing, various algorithms for maintaining the consistency using the operational transformation principle have been used, such as dOPT [3], adOPTed [15], GOTO [17] and SOCT2, SOCT3 and SOCT4 [18] and the work described in [10]. All of these algorithms are based on a linear representation of the document whereas our algorithm uses a tree representation of the document and applies the same basic mechanisms as these algorithms recursively over the different document levels.

Other recent research has also looked at tree representations of documents for the real-time collaborative writing. The dARB [7] algorithm also uses a tree model for document representation, however it is not able to automatically resolve all concurrent accesses to documents and, in some cases, must resort to asking the users to manually resolve inconsistencies. The dARB algorithm may also use special arbitration procedures to automatically resolve inconsistencies; for example, using priorities to discard certain operations, thereby preserving the intentions of only one user. In our approach, we preserve the intentions of all users, even if, in some cases, the result is a strange combination of all intentions. However, the use of different colours provides awareness of concurrent changes made by other users and the main thing is that no changes are lost.

Moreover, in dARB, operations (delete, insert) are defined only at the character level in this algorithm, i.e. sending only one character at a time, and therefore the number of

communications through the network increases greatly. We tried to reduce the number of communications and transformations as much as possible, thereby reducing the notification time, which is a very important factor in groupware. For this purpose, our algorithm is not a character-wise algorithm, but an element-wise one, i.e. it performs insertions/deletions of different granularity levels (paragraphs, sentences, words and characters).

Other projects have used operational transformation applied to documents written in dialects of SGML (Standard General Markup Language) such as XML and HTML [2,12] or to CRC (Class, Responsibility, Collaboration) cards [12]. In these approaches the transformation functions had to be extended, while in our approach a transformational mechanism working for linear structures is recursively applied over the document levels. Moreover, in our approach the history is distributed throughout the tree which yields an increased efficiency.

Concerning the asynchronous text editing we can compare our approach with some versioning systems dealing with text documents such as RCS [19] and CVS [1] that use the state-based approach for merging. We used operation-based merging which offers a better solution for conflict resolution and also for tracking the user activity. The basic conflict unit used by the above mentioned versioning systems is the line, meaning that the changes performed by the two users are in conflict if they refer to the same line. Concurrent changes on different lines are merged automatically. Conflictual changes require manual intervention. In our approach we offer not only manual resolution for conflicts, but also other automatical resolution policies. Moreover, we work with different semantic units, such as paragraph, sentence, word or character, in this way offering a more flexible approach for defining and resolving the conflicts.

In FORCE [16] an operation-based merging approach has been proposed that uses semantic rules for resolving the conflicts. However, this approach can be applied only for a linear representation of the documents.

Other research works looked at the tree representation of documents for the asynchronous collaborative editing. In [20] an XML diff algorithm has been proposed, the approach relying on a state-based merging. Our approach is operation-based, offering better support for conflict resolution and for tracking of user activity.

In [13], the authors propose using the operational transformation approach to define a general algorithm for synchronizing file systems and file contents. The file systems have a hierarchical structure, however, for the merging of the text documents the authors proposed using a fix working unit, i.e. the block unit consisting of several lines of text. The defined operations that work on the block units are: *addblock*, *deleteblock* and *moveblock*. The transformation functions are defined for each pair of operations. In our approach we deal with semantic working units (paragraph, sentence, word, character) and we use one generic transformation function that is applied for all semantic units.

Concerning the graphical editing, to our knowledge, there are no other collaborative asynchronous editing systems. The real-time collaborative editing systems that use the serialisation mechanism such as Group Design [9] and LICRA [8] do not deal with grouping/ungrouping operations as in our approach.

Some other research groups have also investigated synchronous and asynchronous communications using the same basic techniques. In GOTO [17] and FORCE [16], the authors describe how the operation transformation mechanism has been applied for real-time and asynchronous collaborative text editing, respectively. As we have already pointed out, the difference between their approach and ours is that they have used a linear representation of text documents, while we have used a hierarchical representation for the text documents, yielding to an increased efficiency and the flexibility of working at different levels of granularity.

In [12] the authors propose an environment that allows the CRC cards and HTML-based collaborative editing in real-time or asynchronous mode of communication. Operation transformation is the basic mechanism underlying the two modes of communication. A main distinction between their approach and ours is that in our approach the history buffer is distributed throughout the tree, rather than being linear, resulting in an increased performance.

Not only have we developed novel algorithms for maintaining the consistency in a more efficient way, but we have integrated all of our implemented collaborative tools into one information platform offering a customisable collaboration for different ranges of applications. The collaborative textual and graphical editors have been implemented to test the theoretical aspects of the collaboration. We are currently enhancing our collaborative tools for real application domains. For instance, we are enhancing the collaborative graphical editor with new primitives such as polylines, free forms, bitmaps, facilities of zooming, creating annotations and working with layers in order to be used for collaborative product and architectural design.

5. Conclusions

We have presented a customisable approach to collaborative editing offering both synchronous and asynchronous solutions, for both textual and graphical documents. We have highlighted the fact that using a hierarchical structure of the textual document yields a set of improvements compared to the linear representation, such as an increased efficiency and the possibility of working at different granularity levels. In the case of the graphical documents, the hierarchical structure of the scene of objects allows an easy way of implementing the grouping/ungrouping operations that have not been implemented in any other collaborative graphical editor. We have shown how the algorithms used for the asynchronous communication reuse the same techniques underlying real-time collaborative systems. All the mechanisms described in this paper for maintaining the consistency both for the real-time and asynchronous communication have been implemented in our collaborative systems that allow the editing of textual and graphical documents. We plan to extend the collaborative editing also for XML files, both for real-time and asynchronous communication.

References

[1] B. Berliner, CVS II: parallelizing software development, in: Proceedings of the USENIX, Washington, DC, 1990.

[2] A.H. Davis, C. Sun, Generalizing operational transformation to the standard general markup language, in: Proceedings of the CSCW'02, 2002, pp. 58–67.

[3] C.A. Ellis, S.J. Gibbs, Concurrency control in groupware systems, in: Proceedings of the ACM SIGMOD Conference on Management of Data, May 1989, pp. 399–407.

[4] C.L. Ignat, M.C. Norrie, Grouping in collaborative graphical editors, in: Proceedings of the CSCW'04, Chicago, IL, November 2004.

[5] C.L. Ignat, M.C. Norrie, Operation-based versus state-based merging in asynchronous graphical collaborative editing, in: The 6th International Workshop on Collaborative Editing, CSCW'04, Chicago, IL, November 2004.

[6] C.L. Ignat, M.C. Norrie, Customizable collaborative editor relying on treeOPT algorithm, in: Proceedings of the 8th ECSCW, Helsinki, Finland, (September 2003), pp. 315–334.

[7] M. Ionescu, I. Marsic, An arbitration scheme for concurrency control in distributed groupware, in: The Second International Workshop on Collaborative Editing Systems, CSCW 2000, December 2000.

[8] R. Kanwati, LICRA: a replicated-data management algorithm for distributed synchronous groupware application, Parallel Computing 22 (1992).

[9] A. Karsenty, M. Beaudouin-Lafon, An algorithm for distributed groupware applications, in: Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993, pp. 195–202.

[10] D. Li, R. Li, Ensuring Content and Intention Consistency in Real-Time Group Editors, in: Proceedings of the International Conference on Distributed Computing Systems (ICDCS), Japan, March 2004.

[11] E. Lippe, N. van Oosterom, Operation-based merging, in: Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments, 1992, pp. 78–87.

[12] P. Molli, H. Skaf-Molli, G. Oster, S. Jourdain, SAMS: synchronous, asynchronous, multi-synchronous environments, in: Proceedings of the 7th International Conference on CSCW in Design, Rio de Janeiro, Brazil, September 2002.

[13] P. Molli, G. Oster, H. Skaf-Molli, A. Imine, Using the transformational approach to build a safe and generic data synchronizer, in: Group 2003 Conference, November 2003.

[14] J.P. Munson, P. Dewan, A flexible object merging framework, in: Proceedings of ACM Conference on CSCW, 1994, pp. 231–242.

[15] M. Ressel, D. Nitsche-Ruhland, R. Gunzenbauser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, in: Proceedings of ACM Conference on CSCW, November 1996, pp. 288–297.

[16] H. Shen, C. Sun, Flexible Merging for Asynchronous Collaborative Systems, in: Proceedings of the CoopIS/DOA/ODBASE, 2002, pp. 304–321.

[17] C. Sun, C. Ellis, Operational transformation in real-time group editors: issues, algorithms and achievements, in: Proceedings of the ACM International Conference on CSCW, Seattle, (November 1998), pp. 59–68.

[18] N. Vidot, M. Cart, J. Ferrié, M. Suleiman, Copies convergence in a distributed real-time collaborative environment, in: Proceedings of the ACM Conference on CSCW, Philadelphia, USA, (December 2000), pp. 171–180.

[19] W.F. Tichy, RCS – A system for version control, Software-Practice Experience 15 (7) (1985) 637–654.

[20] O. Torii, T. Kimura, J. Segawa, The consistency control system of XML documents, in: Symposium on Applications and the Internet, 2003.

Claudia-Lavinia Ignat received a BS in Computer Science at the Technical University of Cluj-Napoca, Romania. She is currently doing her PhD at the Institute for Information Systems at ETH Zurich, Switzerland. Her research topic is collaborative editing systems, with a focus on consistency maintenance algorithms.

Moira Norrie is a Professor at ETH Zurich where she is Head of the Institute for Information Systems. She is also a member of the Swiss National Research Council. Her research interests include object-oriented models and systems for data management, web engineering, mobile and collaborative information systems and interactive paper as a medium for integrating printed and digital information sources.