Extending real-time collaborative editing systems with asynchronous communication

Claudia-Lavinia Ignat and Moira C. Norrie
Institute for Information Systems, ETH Zurich
{ignat,norrie}@inf.ethz.ch

Abstract

The integration of synchronous and asynchronous modes of communication and the possibility of alternating them along the stages of a project is an important issue for the customization of the collaborative work in the engineering domain. In this paper we present the synchronous and asynchronous modes of collaboration for two main classes of documents, namely, textual and graphical. We describe how the real-time collaborative systems that we have developed can be extended to support also asynchronous functionality. We present two approaches for the asynchronous communication: synchronizing the private workspace against a central repository and against another private workspace.

1. Introduction

Collaborative editing systems have been developed 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.

Most existing systems implement in isolation either the synchronous mode of communication, such as [2], [11] and [12], or the asynchronous mode of communication such as [10] and [8]. Since not all user groups have the same conventions and not all tasks have the same requirements, in software engineering as well as in any engineering domain, means of customizing the collaborative work is required. An integrated system supporting both synchronous and asynchronous collaboration is needed because these two modes of communication can be alternatively used in different stages of project development and under different circumstances. The *real-time* feature is needed when the users in the team want to frequently interact to achieve a common goal. The *non-real-time* feature is required if the users do not want to coordinate interactively with each other. Also, the asynchronous mode of communication is useful in the case that real-time collaboration cannot be performed for some period of time due to some temporary failures, but can operate again afterwards.

In this paper we present the synchronous and asynchronous modes of collaboration for two main classes of documents, namely, textual and graphical. We describe how the real-time collaborative systems that we have developed can be extended to support also asynchronous functionality. The asynchronous functionality can be further customized, offering the user the possibility of synchronizing the work against a central repository or against the private workspace of another user. In our approach we modeled the documents, both text and graphical, using a hierarchical structure. The general structured model of the documents offers a set of enhanced features such as increased efficiency and improvements in the semantics compared to the approaches that use a linear model of the document such as [2], [11] and [12]. We adopt a flexible way of dealing with conflicts, by allowing the possibility to specify, for any application domain, the set of rules that define the conflicts. We also offer various policies for resolving the conflicts, such as a master/slave approach or allowing the users to manually choose among conflicting operations.

The paper is structured as follows. In the first 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., presenting the synchronization of a private workspace against a central repository and against another private workspace. 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 will give an overview of the algorithms for maintaining consistency that underly the

functionality of the collaborative text and graphical editors which form the basis of our multi-mode collaborative system.

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 opposed to turn-taking and locking protocols.

We have chosen a hierarchical representation for both text and graphical documents. We model the text document as consisting of paragraphs, with each paragraph consisting of sentences, each sentence consisting of words and each word consisting of letters. We also use a tree model for representing the scene of objects in a graphical document. Groups are represented as internal nodes, while simple objects are represented as leaves. A group can contain other groups or simple objects.

In what follows we describe in turn the techniques underlying the functionality of the collaborative text and graphical editors.

Operational transformation mechanisms have been applied for maintaining the consistency of the copies of the shared documents in the *real-time collaborative text editor* that we have developed. Operational 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.

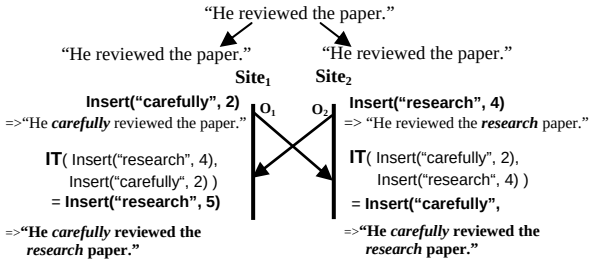

Figure 1. Operational Transformation

Figure 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"* into position 5, the result being *"He **carefully** reviewed the **research** paper"*, satisfying the intentions of both users. At $Site_2$, 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.

In [3] we proposed a consistency maintenance algorithm called treeOPT relying on a tree representation of documents. 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 operational transformation algorithms. Compared to the other operational transformation approaches that use a linear structure for representing the documents, the treeOPT 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 operational transformation algorithms recursively over the different document levels (paragraph, sentence, word and character) and it can use any of the operational transformation algorithms relying on linear representation. Please refer to [5] for a detailed description of our algorithm.

In the *object-based collaborative graphical editor* developed in our group, 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*,

changeColor, *changeBckColor*, *changePosition*, *changeSize*, *bringToFront*, *sendToBack*, *changeText*, *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. A serialization order of execution of these operations cannot be obtained: executing one operation will not make possible the execution of the other operation or will completely mask the execution of the other one. 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 combined effect of their intentions can be obtained by serializing 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.

Although the model used for representing the text and, respectively, the graphic document is hierarchical and the same consistency model (causality preservation, convergence and intention preservation) has been applied to both text and graphical domains, the techniques used for achieving consistency are different. For maintaining consistency in the case of the object-based graphical documents, a serialization mechanism has been applied rather than the operation transformation principle as in the case of the text editor. Please refer to [4] for a detailed description of the algorithm.

3. Asynchronous collaborative 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 [13,1] 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. The most well-known algorithm for computing the difference is diff3 [7]. 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 [6, 10, 8] 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.

In what follows we will describe how the steps for synchronizing the private workspace with the repository can be implemented using the same basic mechanisms as for real-time collaboration. Further, we investigate the process of synchronizing a private workspace against another private workspace.

3.1. Synchronization against the repository

In this subsection we analyze the mechanism of synchronizing the copy of the document from the private workspace against the repository. We analyze the problems that arise at the committing stage when a user commits a copy from the private workspace into the repository and at the updating stage when the copy from the private workspace is updated by a version from the repository.

If a user wants to *commit* the working copy into the repository, he sends a request to the repository server. 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 delta 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 executes sequentially the operations from the log and updates the number of the latest version.

At the *updating* stage, updates made on a committed working copy by another user cannot simply be reapplied in the private working space. This is due to the fact that

the operations from the repository were generated in an initial context represented by a version of the document from the repository and this context has been changed by the execution of the set of operations generated in the private working space. Moreover, some of the operations from the repository are in conflict with some of the operations from the working space.

Our solution for dealing with conflicts in a flexible way follows the ideas described in [9]. 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.

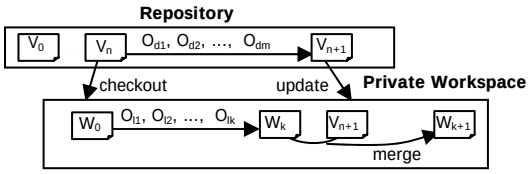

Figure 2. Updating stage

In Figure 2, we have sketched the updating stage of the Copy/Modify/Merge approach. In the repository, the difference between version V_{n+1} and version V_n is represented by the sequence of operations $DL=[O_{d1}, O_{d2}, ..., O_{dm}]$. In the private workspace of a site, after the checkout of version V_n from the repository, the local copy of the document W_0 has been updated by the sequence of operations $LL=[O_{l1}, O_{l2}, ..., O_{lk}]$ to the local copy W_k. For updating the local copy W_k with the version V_{n+1} from the repository, a merge between the sequence of operations from list *DL* and the operations from *LL* will be performed. Afterwards, the version W_{k+1}, the result of merging, can be commited to the repository.

From the list *DL* of operations, not all of them can be re-executed in the private workspace because some of these operations may be in conflict with some of the operations in the list *LL*.

A nonconflicting operation needs to be transformed against the log *LL* of operations in the same way a remote operation is integrated into the history buffer of a site in the case of the real-time mode of communication [3]. In the case that some conflicting operations precede a nonconflicting operation *O*, *O* needs to be transformed before its integration into *LL*. The transformation consists of excluding from *O* the effect of the conflicting operations (by applying the exclusion transformation) because the context of definition of *O* changed. The new form of *O* can then be integrated into the list *LL*.

In order to perform a commit, the difference between the local copy in the private workspace (W_{k+1}) and the last updated version from the repository (V_{n+1}) needs to be computed. The operations in *LL* need to be transformed according to the sequence of operations from *DL* that have been re-executed in the private workspace. The fact that some operations from *DL* have been in conflict with some of the operations in *LL* and could not be executed in the private workspace needs to be included into the delta as their inverse in order to cancel the effect of these operations. For instance, the inverse of the operation of inserting the word *"love"* in paragraph 2, sentence 3 into the 4th position, *InsertWord(2,3,4, "love")* is *DeleteWord(2,3,4, "love")*.

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: *"He like the book."* Suppose a conflict is defined between two operations concurrently inserting the same character in the same position in the document 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}*=InsertSentence("He loves reading.", 1,1)* and O_{12}*=InsertCharacter("d", 1,2,2,5)*. Operation O_{11} inserts the sentence *"He loves reading."* in the first paragraph as the first sentence, in order to obtain the version W_{11}*="He loves reading. He like the book."* Operation O_{12} inserts the character *"d"* in paragraph 1, sentence 2, word 2, as the last character (position 5) in order to obtain the version W_{12}*="He loves reading. He liked the book."* The second user performs the operation O_{21}, O_{21}*=InsertCharacter("d",1,1,2,5)* in order to obtain W_{21}*="He liked the book."*

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*="He loves reading. He liked the book."* and the list $DL_{1,0}$ representing the difference between V_1 and V_0 in the repository will be $[O_{11}, O_{12}]$.

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, a merge between the list of operations $DL_{1,0}$ and the local list $[O_{21}]$ is performed. The operations belonging to $DL_{1,0}$, O_{11} and O_{21}, need to be integrated in turn into the local list $[O_{21}]$. There is no conflict between O_{11} and O_{21}. Moreover, O_{21} is not an operation of sentence insertion, so, according to the treeOPT algorithm, the execution form of O_{11} is the same as its original form, i.e. O'_{11}*=InsertSentence("He loves reading.", 1,1)*. Operation O_{12} will not be executed in the private workspace because O_{12} and O_{21} are conflicting operations and the local operation O_{21} wins the conflict according to the assumed policy of merging.

As a result of merging, the current state of the document in the private workspace of $User_2$ is *"He loves reading. He liked the book."* After updating the local workspace against the repository, $User_2$ can commit his local changes. For committing the local changes, the difference $DL_{2,1}$ between the new version V_2 and V_1 in the repository needs to be computed. The difference $DL_{2,1}$ is represented by the execution form of O_{21} transformed against the list $DL_{1,0}$. Because O_{11} inserts a sentence before the target sentence of operation O_{21}, the transformed operation O_{21} against O_{11} is *InsertCharacter("s", 1,2,2,5)*. Between O_{21} and O_{12} there is a conflict, so O_{21} does not need to be transformed against O_{12}. Therefore, the execution form of O_{21} is O'_{21}=*InsertCharacter("s", 1,2,2,5)*. Moreover, to illustrate the fact that the effect of O_{12} has been cancelled from the repository, inverse(O_{12}) has to be included into $DL_{2,1}$. So, $DL_{2,1}$=[inverse(O_{12}), O'_{21}] and V_2="*He loves reading. He liked the book."*

When $User_1$ updates the local workspace, the operations in $DL_{2,1}$ need to be executed in their form, because no other concurrent operations have been executed in the local workspace of $User_1$ since the creation of version V_1. So, the local copy of the document is "*He loves reading. He liked the book."*

We have applied the same principles of operation transformation as for real-time communication both in the case of the integration of an operation from the repository into the local workspace and in the case of computing the deltas between the versions of the documents into the repository. 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.

The same extension from the synchronous to the asynchronous mode of communication can be achieved in the case of the graphical editor. Rather than using the operational transformation approach as in the case of the text document, the serialization approach together with the undo/redo scheme that has been used for real-time communication can be adapted for the merging process of the asynchronous communication.

3.2. Synchronization between local workspaces

Asynchronous collaboration can be further extended to allow not only the synchronization of the documents between a private working space and the repository, but also between two private working spaces. In the case of collaboratively writing a paper, often an author of the paper wants to synchronize his/her work with the modifications performed by only one of the other authors before committing to the repository. In this subsection we are going to describe the mechanism of synchronizing two local workspaces.

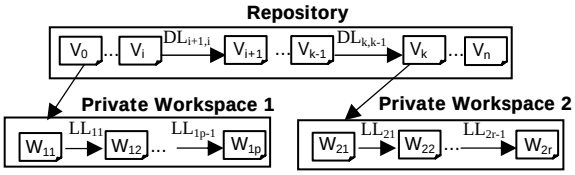

Figure 3. Workspace Synchronization

In Figure 3 we illustrated the case of two users concurrently editing two different versions of the document, version V_i and the more recent version V_k, respectively. Suppose $User_1$ wants to synchronize with the changes performed by $User_2$. The set of operations constituting the difference between version V_k and version V_i, $DL_{k,i}=DL_{i+1,i} \cup \ldots \cup DL_{k,k-1}$, followed by the set of operations representing the local modifications of $User_2$, $L_2=LL_{21} \cup \ldots \cup LL_{2r-1}$, need to be integrated into the set of operations performed by $User_1$, $L_1= LL_{11} \cup \ldots \cup LL_{1p-1}$. After the synchronization with the private workspace of $User_2$, $User_1$ can further perform new modifications on the local copy of the document, synchronize the work with other private workspaces or commit the changes into the repository. Note that the changes performed by $User_2$ and merged with the changes of $User_1$ will be stored in the repository in the case that $User_1$ commits the changes performed in the local workspace before $User_2$. The synchronization of a private workspace can be performed only against another private workspace that has the base version equal or more recent than the current one.

The conflicts are handled in the same way as described in the case of the synchronization against the repository. We offer to the user the possibility to choose among variants of solutions for the conflicts or to adopt a master/slave approach.

As a result of the merging of the private workspaces, some uncommitted operations are replicated in various workspaces. There is the need to avoid committing the same operations more than once. Suppose $User_1$ merges the changes performed by $User_2$ represented by the list L of operations in *Private Workspace 2*. The nonconflictual operations from L will be integrated into *Private Workspace 1*. Further, suppose that $User_1$ performs some

modifications and commits the changes to the repository. At a later time, $User_2$ tries to commit, too. The operations in *L* that have been integrated into *Private Workspace 1* and have been committed to the repository by $User_1$ should not be integrated again into the repository. Moreover, users are allowed to repeatedly synchronize their workspaces against a certain workspace. The system should avoid that operations that have already been integrated into the local workspace are reintegrated at a later synchronization. To tackle this problem, operations are uniquely identified. For instance, the identifier of an operation can be the pair *(sid, seq)* where *sid* is the site identifier and *seq* is the number of the operation generated at that site. In this way an operation having the same identifier as an operation belonging to the log into which it needs to be integrated will not be included into the log.

4. Related Work

Synchronization has been investigated in different research areas such as database systems, distributed systems and configuration management. Related work for the real-time approach adopted in our systems can be found in [3]. Our operation-based merging approach used for the asynchronous communication is closely related to the works described in [10] and [8].

As in FORCE [10], our approach uses semantic rules on top of syntactic merging. However, in [10] the approach is described only for the linear representation of text documents, whereas the hierarchical representation used in our approach yields a set of advantages such as increased efficiency and improvements in the semantics. Moreover, we have described the synchronous and asynchronous modes of communications, not only for the case of text documents, but also for graphical documents. Our goal is to build a general information platform supporting multi-mode collaboration over general types of documents. So far, we have considered text and graphical documents as representative for our research.

In [8], the authors proposed using the operational transformation approach to define a general algorithm for synchronizing a file system and file contents. The main difference between our approach and the one in [8] is that, in our approach, semantic conflicts among operations can be specifically defined for any application domain, while the synchronizer proposed in [8] automatically finds a solution in the case of conflict.

Both works consider only the synchronization of a private workspace with the repository and do not investigate the synchronization between two workspaces.

5. Conclusions

We have presented a customizable approach to collaborative editing offering both synchronous and asynchronous solutions, for both textual and graphical documents. We have shown how the algorithms used for the asynchronous communication reuse the same techniques underlying real-time collaborative systems. We have described two approaches for the synchronization of the local working space: against the repository and against another working space.

6. References

[1] B. Berliner, "CVS II: Parallelizing software development", *Proc. of USENIX*, Washington D.C., 1990.

[2] C.A. Ellis and S.J. Gibbs, "Concurrency control in groupware systems", *Proc. of the ACM SIGMOD Conf. on Management of Data*, May 1989, pp. 399-407.

[3] C.L. Ignat and M.C. Norrie, "Customizable Collaborative Editor Relying on treeOPT Algorithm", *Proc. of the 8th ECSCW*, Helsinki, Finland, Sept. 2003, pp. 315-334.

[4] C.L. Ignat and M.C. Norrie, "Grouping/Ungrouping in Graphical Collaborative Editing Systems", *IEEE Distributed Systems online*, The 5th Intl. Workshop on Collaborative Editing, ECSCW'03, Helsinki, Finland, Sept. 2003.

[5] C. Ignat and M.C. Norrie, "Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems", *IEEE Distributed Systems online*, The 4th Intl. Workshop on Collaborative Editing, CSCW 2002, New Orleans, Louisiana, Nov. 2002

[6] E. Lippe and N. van Oosterom, "Operation-based merging", *Proc. of the 5th ACM SIGSOFT Symposium on Software development environments*, 1992, pp. 78-87.

[7] W. Miller and E.W. Myers, "A file comparison program", *Software-Practice and Experience 15(1)*, 1990, pp. 1025-1040.

[8] P. Molli, G. Oster, H. Skaf-Molli and A. Imine, "Using the transformational approach to build a safe and generic data synchronizer", *Group 2003 Conference*, Nov. 2003.

[9] J.P. Munson and P. Dewan, "A flexible object merging framework", *Proc. of ACM conference on Computer Supported Cooperative Work*, 1994, pp. 231-242.

[10] H. Shen, C. Sun, "Flexible Merging for Asynchronous Collaborative Systems", *Proc. of CoopIS/DOA/ODBASE 2002*, pp. 304-321.

[11] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms and achievements", *Proc. ACM Intl. Conf. on CSCW*, Seattle, Nov. 1998, pp. 59-68.

[12] N. Vidot, M. Cart, J. Ferrié and M. Suleiman, "Copies convergence in a distributed real-time collaborative environment", *Proc. of the ACM Conf. on CSCW*, Philadelphia, USA, Dec. 2000, pp. 171-180.

[13] W.F. Tichy, "RCS- A system for version control", *Software-Practice and Experience, 15(7)*, 1985, pp.637-654