CoDoc: Multi-mode Collaboration over Documents

Claudia-Lavinia Ignat and Moira C. Norrie

Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{ignat,norrie}@inf.ethz.ch

Abstract. In software engineering as well as in any engineering domain, a way of customizing the collaborative work to various modes of collaboration, i.e. synchronous and asynchronous, and the possibility of alternating these modes along the phases of a project is required. Our goal is to develop a universal information platform that can support collaboration in a range of application domains, the basic sharing unit being the document. Since not all user groups have the same conventions and not all tasks have the same requirements, this implies that it should be possible to customize the collaborative environment at the level of both communities and individual tasks. In this paper we present the consistency maintenance models underlying the synchronous and asynchronous modes of collaboration. We highlight the importance of choosing a general structured model of the document and particularly analyze the multi-mode collaboration for two main representative types of documents: textual and graphical.

1 Introduction

Within the CSCW field, 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. We have developed a real-time collaborative text editor [7] and a real-time collaborative graphical editor [8].

Asynchronous collaboration means that members of the group modify the copies of the documents in isolation, working in parallel and afterwards synchronizing their copies to reestablish a common view of the data. Version control systems are asynchronous systems used in group environments and merging plays a key role for achieving convergence in such systems.

In software engineering as well as in any engineering domain, the reduction of the product life cycle, i.e. fewer months between releases, together with the increase in the product complexity and the size of the team, requires a means of customizing the collaborative work to various modes of collaboration, i.e.

synchronous and asynchronous, and the possibility of alternating these modes along the phases of a project.

Our goal is to develop a universal information platform that can support collaboration in a range of application domains such as engineering design (CAD or CAAD) and collaborative writing (news agency, authoring of scientific papers or scientific annotations), the basic unit for collaboration being the document. Since not all user groups have the same conventions and not all tasks have the same requirements, this implies that it should be possible to customize the collaborative editor at the level of both communities and individual tasks.

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. In this way we can customize the collaborative editor and be able to support collaboration in a range of application domains. Choosing a general structured model offers a set of enhanced features such as increased efficiency and improvements in the semantics for both modes of collaboration. Moreover, the general model of the document allows a general consistency model to be found for multi-mode collaboration. An integrated system supporting both synchronous and asynchronous collaboration is needed in practice because these two modes of communication can be alternatively used in developing a project at different stages 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. For example, before a paper deadline when there is time pressure for the authors of the paper, the real-time feature is very helpful. Suppose the two authors of the paper agreed that one of them, the one that is a native english speaker, will go through the whole paper and check the english, while the second author is adding some new sections. The first author can go through the whole document and correct the english and then go on to revise the content and correct the english of the sections written in the meanwhile by the second author and distinguished by a different colour. Another application of real-time collaboration is in the case of editing computerized cooperative music notation both in orchestras and music schools, where the musicians may perform changes simultaneously on the same music score [1]. The cooperative distributed editor of music may be seen as a particular case of a collaborative graphical editor.

The *non-real-time* feature is required if the users do not want to coordinate interactively with each other. An application of this mode of collaboration is when users prefer working in their private workspaces, performing modifications and only afterwards publishing their work to the other members of the group. For example, co-authors of a book that collaboratively write different chapters will merge their work only after finishing a first draft. Another relevant example that requires the need of asynchronous communication is the following one: two users concurrently modelling a huge XML database have a basic DTD and want to work on enhancing the DTD, each one on some specific parts. In the meantime they fill in data into the XML document conforming to the DTD and use XSLT to produce transformations on the XML. In this example, uncoupled editing is desired: During their individual work, the two programmers need to repeatedly edit the DTD, fill in data into the XML and test the transformations, this fact requiring that the DTD is kept in separately consistent states. Also, the asynchronous mode of communication is useful in the case that a real-time collaboration cannot be performed for some period of time due to some temporary failures but can operate again afterwards. For instance, consider the case of a teleconference where the users share a common whiteboard that is interrupted by a communication failure. It should be possible that the members work in isolation on their local copies of the whiteboard during the failure, but be able to merge their work when the communications are restored.

The paper is structured as follows. In the first section we describe the main features of the real-time editing systems that we have developed by highlighting the principles we have used for maintaining consistency in the case of both text and graphical documents. In section 3 we go on to describe the asynchronous mode of collaboration, first giving an overview of the copy/modify/merge technique used in the asynchronous communication and then showing in detail how this paradigm can be implemented by applying the same basic principles as for real-time collaboration. In section 4 we compare our approach with some related work. Concluding remarks are presented in the last section.

2 Real-time Collaborative Editing Systems

In this section we are going to briefly describe 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 CoDoc.

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. Also, both systems are characterized by high concurrency, meaning that any number of 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 both for text and graphical documents. We model the text document as consisting of paragraphs, 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 the 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 the next subsections we describe in turn the techniques underlying the functionality of the collaborative text and graphical editors.

2.1 Real-time Collaborative Text Editor

The operational transformation approach has been identified as an appropriate approach for maintaining consistency of the copies of shared documents in real-time collaborative editing systems. It 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. Various operational transformation algorithms have been proposed: dOPT [5], adOPTed [14], GOT/GOTO [16], SOCT2, SOCT3 and SOCT4 [19].

Figure 1 illustrates a very simple example of the operation transformation mechanism. Suppose the shared document contains a single sentence *"We attended the receptions."* 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 *"all"* as the 3rd word into the sentence, in order to obtain *"We attended all the receptions."* Concurrently, $User_2$ performs operation O_2 of inserting the word *"conference"* as the 4th word into the sentence, in order to obtain *"We attended the conference receptions."* Let us analyse what happens at each site when the operation from the other site arrives. At $Site_1$, when operation O_2 arrives, if executed in its original form, the result would be *"We attended all conference the receptions."* which is not what the users intended. At $Site_2$, when operation O_1 arrives, if executed in its original form, the result, fortunately, would be a merge of the intentions of the two users, i.e. *"We attended all the conference receptions."* But, generally, executing the operations in their generation form at remote sites, will not ensure that the copies of the documents at $Site_1$ and $Site_2$ will converge. So, we see the need of transforming the operations when they arrive 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 needs to adapt the position of insertion, i.e. increase its position by 1. In this way the transformed operation O_2 will become an insert operation of the word *"conference"* into position 5, the result being *"We attended all the conference receptions."*, 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.

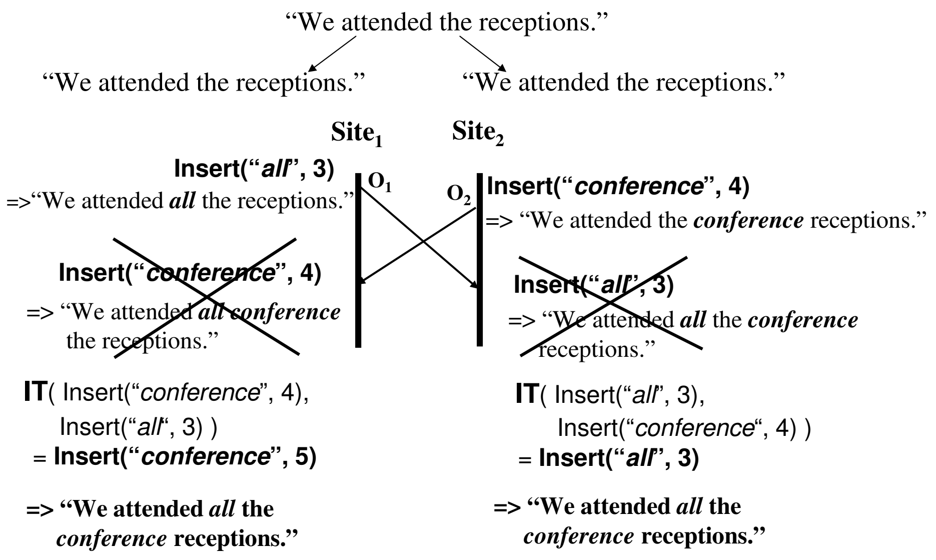
"We attended the receptions."
"We attended the receptions."
"We attended the receptions."
Site1
Site2
Insert("all", 3)
=>"We attended all the receptions."
O1
O2
Insert("conference", 4)
=> "We attended the conference receptions."
Insert("conference", 4)
=> "We attended all conference the receptions."
Insert("all", 3)
=> "We attended all the conference receptions."
IT(Insert("conference", 4), Insert("all", 3))
= Insert("conference", 5)
=> "We attended all the conference receptions."
IT(Insert("all", 3), Insert("conference", 4))
= Insert("all", 3)
=> "We attended all the conference receptions."

Fig. 1. Operation transformation example

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 operational 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 operational 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 [7] 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 and, in this way, our algorithm can be seen as extending the existing operational transformation algorithms. An important advantage of the algorithm is related to its improved efficiency. In our representation of the document, the history of operations is not kept in a single buffer, but rather distributed throughout the whole tree, and, when a new operation is transformed, only the history distributed on a single path of the tree will be spanned. Moreover, when working on medium or large documents, operations will be localized in the areas currently being modified by each individual user and these may often be non-overlapping. In these cases, almost no transformations are needed, and therefore the response times and notification times are very good. Another important advantage is the possibility of performing, not only operations on characters, but also on other semantic units (words, sentences and paragraphs). The transformation functions used in the operational transformation mechanism are kept simple as in the case of character-wise transformations, not having the complexity of string-wise transformations. An insertion or a deletion of a whole paragraph can be done in a single operation. Therefore, the efficiency is further increased, because there are fewer operations to be transformed, and fewer to be transformed against. Moreover, the data is sent using larger chunks, thus the network communication is more efficient. Our approach also adds flexibility in using the editor, the users being able to select the level of granularity at which they prefer to work.

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 such as dOPT [5], adOPTed [14], GOT/GOTO [16], SOCT2, SOCT3 and SOCT4 [19] (see [6]).

2.2 Real-time Collaborative Graphical Editor

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* and *group/ungroup*.

In the case of conflicting operations, we have identified two types of conflict between the operations: real conflict and resolvable conflict.

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 prevent or completely mask the execution of the other operation. An example of real conflicting operations are two concurrent operations both targeting the same object and changing the colour of that object to different values.

The collaborative graphical editing system we have implemented is a customizable editor allowing groups of users to choose a policy for dealing with concurrency. Our system offers three policy modes in the case that a set of concurrent operations are conflicting: the null-effect based policy where none of the operations in conflict are executed, the priority based policy when only the operation issued by the user with the highest priority wins the conflict and the multi-versioning policy when the effects of all operations are maintained. The last of these has still to be implemented.

In the case of the null-effect based policy for dealing with conflicts, none of the concurrent real conflicting operations will be executed. In the case of the priority based policy, given a set of concurrent real conflicting operations, only the operation with the highest priority will be executed, the other operations being cancelled.

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 graphical 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. In what follows we will give an explanation of this difference. As we have seen, the text document has been modelled as a set of paragraphs, each paragraph as a set of sentences and so on. Each semantic unit (paragraph, sentence, word, character) can be uniquely identified by its position in the sequence of the children elements of its parent. In order to achieve consistency, the insertion and deletion operations on these elements may shift the positions of the elements in the sequence of the children elements of their parent for adapting to the effect of concurrent operations. In the case of graphical documents, the objects are not organized into sequences and identified by their position in the sequence. Rather, they are identified by unique identifiers which are immutable. So, there is no need to adapt the identifiers due to the concurrent operations. Graphical objects in the case of graphical documents have associated attributes which are subject to concurrent operations. The elements in the tree model of the text document have no associated attributes. In the case that we want to associate different fonts and styles to elements of the text document, we could represent those elements using object identifiers and attach attributes such as font size to the elements.

3 Asynchronous Collaborative Editing Systems

In this section, we first describe briefly the copy/modify/merge paradigm supported by configuration management tools. Afterwards, we investigate how each of the steps of this paradigm can be implemented using the same basic mechanisms as for real-time collaboration.

3.1 Copy/Modify/Merge Techniques

All configuration management tools support the Copy/Modify/Merge technique. This technique consists basically of three operations applied on a shared repository storing multiversioned objects: checkout, commit and update.

A *checkout* operation creates a local working copy of an object from the repository.

A *commit* operation creates in the repository a new version of the corresponding object by validating the modifications done on the local copy of the object. The condition of performing this operation is that the repository does not contain a more recent version of the object to be committed than the local copy of the object.

An *update* operation performs the merging of the local copy of the object with the last version of that object stored in the repository.

In Fig. 2 a scenario is illustrated in order to show the functionality of the Copy/Modify/Merge paradigm. $User_1$ and $User_2$ checkout the document from the repository and create local copies in their private workspaces (operations 1 and 2, respectively). $User_1$ modifies the document (operation 3) and afterwards commits the changes (operation 4). $User_2$ modifies in parallel with $User_1$ the local copy of the document (operation 5). Afterwards, $User_2$ attempts to commit his changes (operation 6). But, at this stage, $User_2$ is not up-to-date and therefore cannot commit his changes on the document. $User_2$ needs to synchronize his version with the last version, so he downloads the last version of the document from the repository (operation 7). A merge algorithm will be performed in order to merge the changes performed in parallel by $User_1$ and $User_2$ (operation 8). Afterwards, $User_2$ can commit his changes to the repository (operation 9).

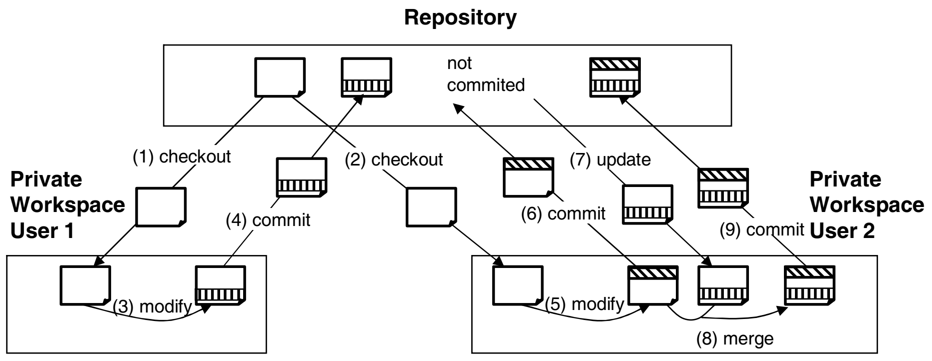

Fig. 2. Copy/Modify/Merge paradigm

Two different ways of performing the merging operations have been devised: state-based merging and operation-based merging.

State-based merging [2, 17] uses only information about the states of the documents and no information about the evolution of one state into another. In the merging process the two states of the document are compared and a delta is generated (the most well-known algorithm used is diff3 [11]). Afterwards, this delta is applied on one of the states of the document to generate the document state representing the result of merging.

Operation-based merging [10, 15] keeps the information about the evolution of one state of the document into another in a buffer containing the operations performed between the two states of the document. The merging is done by executing the operations performed on a copy of the document onto the other copy of the document to be merged.

In what follows 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.

3.2 Committing Stage

In the case that 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, in the case that 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.

3.3 Updating Stage

Merging at the updating stage is more difficult to achieve than the merging at the committing stage. Updates made on a committed working copy by another user cannot be simply reapplied in the private working space, because they were generated in another context. In the updating stage, a set of conflicts among operations occurs.

Our aim is to find a unifying approach for collaborative text and graphical editors working in both synchronous and asynchronous modes.

In the case of the real-time collaborative text editor application, the semantics are somehow already incorporated in the way of resolving the conflicts. Always there is a way of combining the individual intentions of the users to obtain a group intention (although this does not completely satisfy the individual intentions). For example, consider a shared document consisting of the sentence: *"He has discovered that we do not like his company."* Suppose two users concurrently edit the copies of this document, the first one inserting the word *"finally"* in order to obtain *"He has finally discovered that we do not like his company."* and the second one deleting *"He has discovered that"* in order to obtain *"we do not like his company."* After performing both operations the result will be *"finally we do not like his company."* which is not what either of the users expected. However, the use of different colours provides awareness of concurrent changes made by other users and the users can further edit the result.

As we have already seen, in the case of the real conflicting operations for the graphical editor, the individual intentions of the users cannot be respected. For instance, consider the case of two users concurrently moving the same object to different positions. The only way of respecting the intentions of the two users is by creating versions of the object, which might not be the preferred solution. As alternative solutions, the collaborative graphical editor offers a null-effect based policy, i.e. none of the two operations will be executed, or the priority based policy where the operation of the user that has the highest priority is executed.

Our solution for dealing with conflicts in a flexible way is to allow the possibility to specify, both for the text and graphical editor, a set of rules that define the conflicts as well as a function showing if there is a conflict between any two operations in a certain context. The conflicts can therefore be defined specifically for any application domain. In this way, we make a distinction between the syntactic and semantic consistency. *Syntactic consistency* means to reconcile the divergent copies by using either operational transformation or a serialization mechanism. It ensures that all sites have the same view, even if that view has no meaning in the context of the application. On the other hand, *semantic consistency* is defined specifically for an application, in our approach being modelled by the set of rules that specify the conflicts. For resolving the conflicts, either an automatic solution can be provided or human intervention can be required. Different merge policies in the context of various collaborative activities have been analysed in [13].

In Fig. 3, we have sketched the updating stage of the merging. 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. Afterwards, the local copy W_k needs to be updated with the version V_{n+1} from the repository. So, a merge between the sequence of operations from list DL and the operations from LL will be performed in order to obtain the local copy W_{k+1}. Afterwards a commit can be performed to the repository. But, 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}) should be computed.

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

A nonconflicting operation needs to be integrated into 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, as described in [7].

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.

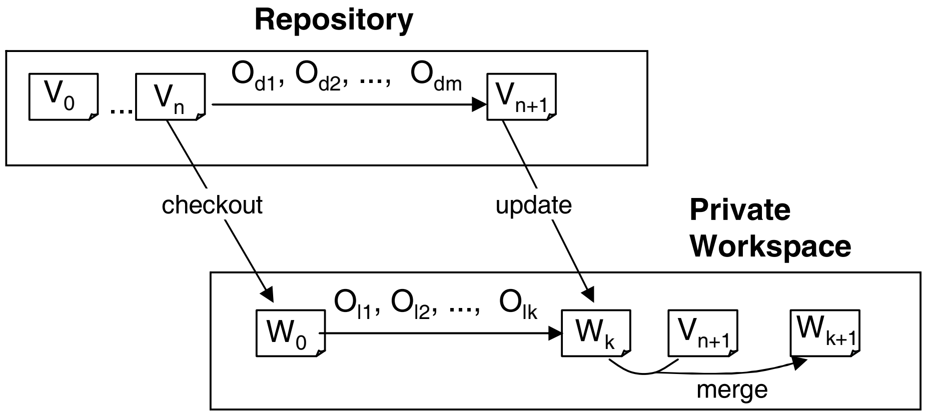

Fig. 3. The updating stage of the Copy/Modify/Merge paradigm

In order to compute the difference between the local copy in the private workspace (W_{k+1}) and the last updated version from the repository (V_{n+1}), 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. Moreover, 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 those operations. For instance, the inverse of the operation of inserting the word *"love"* in paragraph 2, sentence 3, as the 4th word, *InsertWord(2,3,4, "love")*, is *DeleteWord(2,3,4, "love")*.

Let us give an example to illustrate the asynchronous communication. Suppose the repository contains as version V_0 the document consisting of only one paragraph with one sentence: *"He enjoy going to cinemas."* Suppose two operations concurrently inserting the same character in the same position in the document are conflicting. Further, suppose two users check out version V_0 from the repository into their private workspaces and have as first version in their private workspace $W_{10} = V_0$ and $W_{20} = V_0$ respectively. The first user performs operations O_{11} and O_{12}, where O_{11}=*InsertSentence("He loves movies.", 1,1)* and O_{12}=*InsertCharacter("s", 1,2,2,6)*. Operation O_{11} inserts the sentence *"He loves movies."* in the first paragraph as the first sentence, in order to obtain the version W_{11}= *"He loves movies. He enjoy going to cinemas."* Operation O_{12} inserts the character *"s"* in paragraph 1, sentence 2, word 2, as the last character (position 6) in order to obtain the version W_{12}= *"He loves movies. He enjoys going to cinemas."* The second user performs the operations O_{21} and O_{22}, O_{21}=*InsertCharacter("s", 1,1,2,6)* and O_{22}=*DeleteCharacter(1,1,5,7)*. Operation O_{21} inserts character *"s"* to correct the spelling of the word *"enjoy"* in order to obtain W_{21}= *"He enjoys going to cinemas."* Operation O_{22} deletes the last character from the word *"cinemas"* in order to obtain W_{21}= *"He enjoys going to cinema."*

Suppose that, after performing these operations, 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 movies. He enjoys going to cinemas."* and the list DL_{01} representing the difference between V_1 and V_0 in the repository will be [O_{11}, O_{12}].

When $User_2$ gets access to the repository he will receive a message to update the local copy. At this moment, a merge between the list of operations DL_{01} and the local list [O_{21}, O_{22}] is performed, i.e. according to the semantic rules, O_{11} and O_{12} will be integrated into the local list. There is no semantic conflict between O_{11} and either of the operations O_{21} and O_{22}. Moreover, neither of the operations O_{21} and O_{22} are operations 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 movies.", 1,1)*. Operation O_{12} will not be executed because O_{12} and O_{21} are conflicting operations. As a result of merging, the current state of the document is: *"He loves movies. He enjoys going to cinema."* In order to compute the difference DL_{21} between the current copy of the document of $User_2$ and version V_1 in the repository, the operations in the local list [O_{21}, O_{22}] need to be transformed according to the nonconflicting operations from DL_{01}, i.e. operation O_{11}. Because O_{11} inserts a sentence before the target sentence of operations O_{21} and O_{22}, operations O_{21} and O_{22}, will have their execution form: O'_{21}=*InsertCharacter("s", 1,2,2,6)* and O'_{22}=*DeleteCharacter(1,2,5,7)*. Since O_{12} was a conflicting operation, DL_{21}=[inverse(O_{12}), O'_{21}, O'_{22}].

When $User_1$ will update the local workspace, the operations in DL_{21} 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 will be *"He loves movies. He enjoys going to cinema."*

We can see that, for the collaborative text editor, the same principle of operation transformation for integrating an operation into a log containing concurrent operations applies both for asynchronous communication as well as for real-time communication. As in the case of the real-time mode of communication, also in the case of the asynchronous mode of collaboration, a set of improvements can be obtained because of the way of structuring the document. Only a 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. For example, two operations inserting words in two different paragraphs will not need to be transformed because they do not interfere with each other. 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 could 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. In the case of the updating stage of the asynchronous communication, and more specifically in the case of merging, instead of using the operational transformation approach as in the case of the text document, the serialization approach together with the undo/redo scheme can be used in the same way it was used for real-time communication. The set of conflicting operations can be expressed in the function for defining the semantic conflicts between the operations. For integrating an operation into a log containing other concurrent operations, an undo/redo scheme is performed taking into account a set of serialization rules as explained in [8].

4 Related Work

We have already presented the advantages of our approach that uses a hierarchical structure over the other approaches that use a linear structure for the representation of the document.

In the case of the asynchronous communication, we have adopted the same merging mechanism as in FORCE [15]. It is best suited to our requirements since it uses operation-based merging and semantic rules for resolving conflicts. However, in [15], the approach is described only for the linear representation of text documents. The hierarchical representation that we have adopted in our approach yields a set of advantages such as an increased efficiency and improvements in the semantics. Moreover, we have described the synchronous/ asynchronous modes of communication, 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.

Other research works looked at the tree representation of documents in the case of the collaborative editing. In [18] an XML diff algorithm has been proposed. The approach uses a state-based merging involving a complex mechanism to compute the difference between the two tree structures representing the initial and the final document, respectively. This difference is represented as a set of operations that transforms the initial document into the final one. Our approach is operation-based, the operations being kept in a history buffer during the editing process, and therefore the comparison of tree structures is unnecessary.

The dARB algorithm [9] uses a tree structure for representing the document and an arbitration scheme for resolving the conflicts in the case of the real-time collaboration. The arbitration scheme decides to keep the intentions of only one user in the case that some users perform concurrent operations that access the same node in the tree. The arbitration is done according to priorities assigned to operation types. For instance, the operations to create/delete words are assigned a greater priority than the operations that modify a character in the word. There are cases when one site wins the arbitration and it needs to send, not only the state of the vertex itself, but maybe also the state of the parent or grandparent of the vertex. For example, if one user performs a split of a sentence, while another user concurrently performs some modification in the original sentence, the user performing the split will win the arbitration and need to send to all other users the state of the whole paragraph that contains the sentence. By allowing the customization of specifying the conflicts for various application domains, our approach is more general than the one defined in [9] that considers any concurrent operations performed on a vertex of a tree to be conflicting. By using operational transformation, we try to accommodate the intentions of all the users and we do not need the retransmission of whole sentences, paragraphs or, indeed, the whole document in order to maintain the same tree structure of the document at all sites.

In [3], the operational transformation was applied to documents written in dialects of SGML such as XML and HTML. However, the approach was applied only for the real-time communication of these types of documents.

In [12], the authors proposed the use of the operational transformation approach for defining a general algorithm for synchronizing a file system and file contents. The main difference between our approach and the one in [12] is that, in our approach, semantic conflicts among operations can be defined specifically for any application domain, while the synchronizer proposed in [12] automatically finds a solution in the case of conflict.

5 Concluding Remarks

Rather than adopting a single solution for supporting collaboration, we propose a customizable approach that can be adapted for various application domains and can offer a set of solutions for the different stages of the development of a common task. In this paper, we have described the solutions that we offer, both in terms of the types of documents that form the basic unit of collaboration, i.e. textual and graphical, and also in terms of the modes of collaboration, i.e. synchronous and asynchronous. We have shown how the algorithms necessary for supporting the asynchronous functionality extend the consistency maintenance algorithms that we previously developed for real-time collaboration.

We use a general structured model of the document that offers a set of enhanced features such as increased efficiency and improvements in the semantics and allows a general consistency model to be found for multi-mode collaboration.

We plan to evaluate our system according to some suggestions given in [4], mainly by performing user studies for testing the functionality of our systems and by measuring performance.

References

1. Bellini, P., Nesi, P., Spinu, M.B.: Cooperative visual manipulation of music notation. ACM Trans. on CHI, vol.9, no.3, Sept 2002, pp.194-237.

2. Berliner, B.: CVS II: Parallelizing software development. Proceedings of USENIX, Washington D.C., 1990.
3. Davis, A.H., Sun, C.: Generalizing operational transformation to the Standard General Markup Language. Proc. of CSCW, 2002, pp. 58-67.
4. Dewan, P.: An integrated approach to designing and evaluating collaborative applications and infrastructures. CSCW, no. 10, 2001, pp. 75-111.
5. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. Proc. of the ACM SIGMOD Conf. on Management of Data, May 1989, pp. 399-407.
6. Ignat, C.L., Norrie, M.C.: Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems. The 4th Intl. Workshop on Collaborative Editing, CSCW, New Orleans, USA, Nov. 2002.
7. Ignat, C.L., Norrie, M.C.: Customizable collaborative editor relying on treeOPT algorithm. Proc. of the 8th ECSCW, Helsinki, Finland, Sept. 2003, pp. 315-334.
8. Ignat, C.L., Norrie, M.C.: Grouping/ungrouping in graphical collaborative editing systems. IEEE Distributed Systems online, The 5th Intl. Workshop on Collaborative Editing, 8th ECSCW, Helsinki, Finland, Sept. 2003.
9. Ionescu, M., Marsic, I.: Tree-based concurrency control in distributed groupware. CSCW: The Journal of Collaborative Computing, vol. 12, no. 3, 2003.
10. Lippe, E., van Oosterom, N.: Operation-based merging. Proc. of the 5th ACM SIGSOFT Symposium on Software development environments, 1992, pp. 78-87.
11. Miller, W., Myers, E.W.: A file comparison program. Software - Practice and Experience, 15(1), 1990, pp. 1025-1040.
12. Molli, P., Oster, G., Skaf-Molli, H., and Imine, A.: Using the transformational approach to build a safe and generic data synchronizer, Group 2003 Conf., Nov. 2003.
13. Munson, J.P., Dewan, P.: A flexible object merging framework. Proc. of the ACM Conf. on CSCW, 1994, pp 231-242.
14. Ressel, M., Nitsche-Ruhland, D., Gunzenbauser, R.: An integrating, transformation-oriented approach to concurrency control and undo in group editors. Proc. of the ACM Conf. on CSCW, Nov. 1996, pp. 288-297.
15. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. Proc. of CoopIS/DOA/ODBASE 2002, pp. 304-321.
16. Sun, C., Ellis, C.: Operational transformation in real-time group editors: Issues, algorithms, and achievements. Proc. of the ACM Conf. on CSCW, Seattle, Nov. 1998, pp. 59-68.
17. Tichy, W.F.: RCS- A system for version control. Software - Practice and Experience, 15(7), Jul. 1985, pp. 637-654.
18. Torii, O., Kimura, T., Segawa, J.: The consistency control system of XML documents. Symposium on Applications and the Internet, Jan. 2003.
19. Vidot, N., Cart, M., Ferrié, J., Suleiman, M.: Copies convergence in a distributed real-time collaborative environment. Proc. of the ACM Conf. on CSCW, Philadelphia, USA, Dec. 2000, pp.171-180.