Diploma Thesis

# Asynchronous Collaborative Text Editing

Bogdan Dumitriu
bdumitriu@bdumitriu.ro

May 20th, 2004

Institute of Information Systems
Swiss Federal Institute of Technology (ETHZ)

Diploma Professor
Prof. Dr. Moira C. Norrie

Supervisor
Claudia Ignat

# Contents

# Chapter 1

# Introduction

## 1.1 Why do research in this area?

In this day and age it is becoming increasingly clear that all individuals that keep on working alone are destined to be easily overcome by groups of people working together. In a business environment that relies strongly on concurrency (as the one nowadays certainly is), assigning serious tasks to the individual (as opposed to groups) is no longer feasible. Every business that persists in doing so is destined to fail sooner or later in such an environment.

As one might expect, this exact same principle applies roughly identically in most other areas of human life as well, it is not limited to business environments alone. It is a well known fact that people work much more efficiently in groups than alone and that, consequently, there is a strong tendency in our society that involves the migration towards group work above all else. One can notice this tendency in virtually all present day organizations, may them be profit or non-profit.

Another ingredient which is essential to the argument "I'm trying to make" refers to the omnisciency of technology in day to day life. Every single area of work uses and benefits from the advances of technology in some way or another. Undoubtedly, the most widely occurring form of technology is materialized in computers. They are everywhere around us, all qualified personnel interacts with them daily and their penetration of the market is ever more deeper. Under such circumstances, it is only natural that group work itself, along with all other types of human activities, should eventually migrate towards this emerging digital world. Naturally, in order for such a migration to be possible, adequate support has to be made available to those that need it.

This slowly brings us to the point where this research comes into play, namely the point of thinking about the precise technological tools that members of a group require in order for them to truly benefit from the advantages that computers have to offer. These tools should ideally form the software foundation on which groups should be able to base their work. The common name currently used for the study and research directed towards the development of such tools is Computer-Supported Cooperative Work (CSCW). Essentially, this is a multi-disciplinary field which brings social and computer scientists together, presenting them with the common task of discovering how groups can best bene-

fit from technological advances of the moment and developing the right software for this to become a reality.

The particular subarea of Computer-Supported Cooperative Work that this paper tries to approach is called Collaborative (or Cooperative) Editing Systems. A rough description of this field can be found in [Sun et al. 1998] and implies that such systems are very useful groupware tools that allow physically dispersed people to edit a shared textual document, draw a shared graph structure, record ideas during a brainstorm meeting or hold a design meeting. Of course, these are just some basic examples; additional ones can easily be imagined. The motivation for developing such systems can be found by looking at the means used today by most groups in order to coordinate their work. A typical scenario involves a very tight group policy to which all members, with no exception, must conform. This policy would usually indicate precisely what parts of the common objective (may it be a document, a piece of graphics, an architectural design, etc.) each member of the group can modify, rendering all its other parts virtually inaccessible (or at least unchangeable) to this particular member. For two (or more) persons to actually work on the same part of the "document" extremely careful coordination must be employed, with geographical distance making this a difficult, if not impossible, task. Therefore, most "real" group work today is drastically limited by the need of physical proximity of the group members. Under these circumstances, the necessity for the kind of tools advertised by the field of Collaborative Editing Systems becomes obvious.

Open source software development is, for many, the most relevant example of collaborative work which involves members scattered all over the world. These people may be freelance programmers, developers working for various companies, researchers in universities or simply individuals who want to contribute with their own share to the advance of the various software tools. It is clear that each of these people do their part of the work from their own physical location (home, university, office or wherever) and somehow must be aided by technology in order for their joint efforts to actually materialize into something useful. This particular example has been brought into discussion with the purpose of extending the argument exposed so far with the concept of "asynchrony" that appears in the title of this paper alongside that of "collaborative editing systems". In this line of thought, the remark that needs to be made is that open source software development, besides being a relevant example of collaborative work, is also a typical case in which the members of the group don't need to keep so tight a synchronization among each other as in the case of various other group activities. The reason for this is that often during the process of software development several changes (some of which could take hours or even days to be completed) have to be joined before the master "document" can be updated in order for the others to become aware of the modifications. By contrast, there are many cases when a more frequent (up to the point of constant) synchronization must be kept among the participants. Such cases do not constitute the subject of this paper, even though reference to them will occasionally be made, when aspects thereof are relevant to the subject at hand.

This research work, as explained before, places itself within the area of Collaborative Editing Systems and comes as an addition to the already existing achievements in the field, narrowing its focus, however, into two major ways. The first confine is given by the fact that only asynchronous editing will be employed throughout this paper, while the second is related to the issue of deal-

ing with text editing only. This implies that the results achieved by the current work will be useful to those interested in editing various types of text documents (such as source code, technical reports, LaTeX documents, books and virtually anything that can be expressed in characters alone) in an asynchronous manner (i.e. working together on the same project(s), but only interested in reaching common versions only every once in a while). Despite the fact that serious research is being conducted in this field, there is still a lot of room for improvement in various areas, such as efficiency (both in running time and in storage capacity usage), closer adherence to the user's intentions, reliability or mere usability.

In light of all that has been discussed up to this point, hopefully both the subject and the motivation of this research have become sufficiently clear to the reader, thus enabling us to move on to a short description of the work that has been done.

## 1.2 Project outline

The idea behind this project was to create a completely functional system which would allow users to concurrently edit in isolation the same text document(s) and synchronize their work in order to obtain a common view of the edited documents. The synchronization process is to be done in a reliable and consistent way, such that none of the modifications made by any of the users are lost and that all users end up with exactly the same version of the document after the synchronization has taken place. Additionally, all the editing and network communication support has to be provided by the application.

The most challenging aspect of the project consisted in developing the robust algorithm to be used during the synchronization phase. For this purpose, the first part of the work involved a thorough survey of available algorithms both for real-time and asynchronous synchronization, including those requiring the use of a dedicated central server as well as those without such a requirement. This was done with the intention of identifying the most appropriate one to be adapted for the project's needs. Developing this algorithm was a thorough process, during which several very serious issues had to be overcome in order to obtain a correctly working version. The most notable of these issues arose from our idea of going even further than other researchers have gone and design this algorithm so that it could be recursively applied on a tree representation of the document. Choosing a general structured model of the document allows modelling of a larger class of documents such as XML documents compared to the linear model approach. Moreover, the hierarchical structure offers a set of enhanced features such as increased efficiency and improvements in the semantic.

An entire "infrastructure" had then to be built around this algorithm in order to present the user with a usable and fully functional product. This included the development of an appropriate repository where all versions of the document(s) could be stored and offered, on request, to any interested client. Aside from this, the repository also had to provide its clients with the possibility of committing their changes to the document in order for others to have access to them. Another part of the "infrastructure" consisted in the implementation of a graphical user interface to allow the clients to retrieve, modify and send their modifications back to the repository (only to mention the basic functions).

Direct user synchronization is yet another feature that we wanted to provide our users with. This had implications both at the algorithmic level (since we needed to further ensure consistency and user intention preservation in the presence of this new type of synchronization) and at the "infrastructure" level (since all clients needed to be able to behave not only as editors of documents, but also as providers of documents). Efforts were made in this direction because we realized that such an addition to the functions of the application would be a great benefit to the users that prefer working in smaller groups within the large groups.

The entire project was implemented using exclusively Java technologies, thus making it possible for users with various platforms and operating systems to easily work together with the exact same software.

## 1.3 Overview

This paper begins by looking at the efforts of other researchers which have worked in this field and showing their achievements as well as their weak points, and suggesting the new ideas we have pursued in order to fill some of the gaps others have left empty. We also try to explain why we think our work is worth while by presenting some real life possible uses of asynchronous text editing.

The next chapter goes into more detail regarding various aspects, concepts and ideas peculiar to asynchronous editing in general and asynchronous text editing in particular, concepts which are necessary for the understanding of the rest of this paper.

Chapter four presents some (mostly theoretical) issues concerning operation-based editing. The most important part of the chapter tries to explain what the inclusion and exclusion transformations are as these are fundamental aspects which are important for the actual implementation. Other concepts such as consistency models and operation context are introduced as well.

The fifth chapter represents the most important part of this paper since it is the chapter in which our actual work is presented. We begin by describing some of the data strctures we use, the employed operation model and the basic functions that work with the model. Then we go on to describing the most important algorithms we have developed as well as some of the extra features we have introduced. The chapter ends with an informal efficiency analysis of the most relevant algorithms.

Our paper ends by suggesting a few ways in which we believe our work can be continued by others and drawing some conclusions based on our results.

# Chapter 2

# Background and motivation

We shall begin this paper by first looking into the already existing developments in the area of asynchronous collaborative editing at the moment of our contact with the field. This chapter presents the achievements and ideas of other researchers, with occasional comments regarding advantages or disadvantages of the analyzed systems. In the end, we shall explain why we believed work in this area was necessary and set as our objective to develop a new asynchronous editing system.

## 2.1 Related work

In this section we try to briefly introduce the reader to ongoing efforts in the area of asynchronous collaborative editing and try to show the basis on which our work had to be built. We cover state- as well as operation-based systems (see section 3.5.2 to find out more about the distinction between the two types of systems) and discuss the ideas introduced in each of them.

### 2.1.1 State-based version control systems

The systems we present in this section are both widely-used, production systems and therefore we relate more distantly to them than to the ones in the next section. However, they basically have the same objectives that we do and therefore should be mentioned as related work.

#### CVS

CVS (Concurrent Versioning System) is the traditional versioning system used by programmers as aid for collaborative source code development. Even though CVS can be used for collaborative editing of any kind of text file, probably more than 90% of the cases it is used for source code development. Virtually everybody in the programming business has at least heard of CVS and most of us have also used it. Therefore, we shall only give a brief description here as probably anything more than that would be common knowledge. The information below comes from [CVS].

The Concurrent Versioning System implements a version control system: it keeps track of all work and all changes in a set of documents, typically the

implementation of a software project, and allows several (potentially widely separated) developers to collaborate. CVS has become popular in the free software world. Its developers release the system under the GNU General Public License.

CVS uses a client-server architecture: a server stores the current version(s) of the project and its history, and clients connect to the server in order to check-out a complete copy of the project, work on this copy and then later check-in their changes. Typically, client and server connect over a LAN or over the Internet, but client and server may both run on the same machine if CVS has the task of keeping track of the version history of a project with only local developers. The server software normally runs on Unix, while CVS clients may run on any major operating-system platform.

Several clients may edit copies of the project concurrently. When they later check-in their changes, the server attempts to merge them. If this fails, for instance because two clients attempted to change the same line in a certain file, then the server denies the second check-in operation and informs the client about the conflict, which the user will need to resolve by hand. If the check-in operation succeeds, then the version numbers of all files involved automatically increment, and the CVS server writes a user-supplied description line, the date and the author's name to its log files.

Clients can also compare different versions of files, request a complete history of changes, or check-out a historical snapshot of the project as of a given date or as of a revision number. Many open-source projects allow "anonymous read access", meaning that the clients may check-out and compare versions without a password; only the check-in of changes requires a password in these scenarios.

Clients can also use the "update" command in order to bring their local copies up-to-date with the newest version on the server. This eliminates the need for repeated downloading of the whole project.

CVS can also maintain different "branches" of a project. For instance, a released version of the software project may form one branch, used for bug fixes, while a version under current development, with major changes and new features, forms a separate branch.

The main advantages of CVS (which determined its wide-spread use from today) relate to the number of facilities it offers to its users, such as version management, branch management, log management, project structure management, security policies and many others which make it fit for use in production. If it had not been for CVS, many of the open-source projects that exist today would not have been possible to develop due to geographical separation of the developers. The open-source world owes a lot to CVS.

However, the system also has several drawbacks, such as problems when renaming files, moving files to a different directory, lack of proper support for binary files, limited security in sending passwords over the network and so on. Other emerging systems (e.g. Subversion, BitKeeper) are trying to deal with these problems and eventually build a better CVS.

The issue we are interested in from the perspective of our project is that CVS is a state-based versioning control system, which bases its file comparison on the diff3 algorithm. The building block CVS works with is the line. This means that the changes of two users are in conflict if they refer to the same line. Concurrent changes that do not 'conflict', or touch the same lines of a file in different ways, are merged automatically. Changes that do conflict are noted in the working files and sent back to the user for manual resolution. It can,

consequently, become very frustrating for the user, when merging of changes cannot be done automatically, to manually analyze each conflict and decide what needs to be kept and what needs to be removed. The fact that the only way to solve conflicts is by user intervention is actually the problem with all state-based systems that exist today, not only CVS.

### Subversion

The home page of Subversion (see [Subversion Home Page]) gives the clearest comparison between it and CVS, which we reproduce here in order to justify why this newer system is a step forward from CVS. Even though some of the improvements are somewhat beyond the area of our discussion, we kept them for the sake of completeness.

- *Most current CVS features.*

  Subversion is meant to be a better CVS, so it has most of CVS's features. Generally, Subversion's interface to a particular feature is similar to CVS's, except where there's a compelling reason to do otherwise.

- *Directories, renames, and file meta-data are versioned.*

  Lack of these features is one of the most common complaints against CVS. Subversion versions not only file contents and file existence, but also directories, copies, and renames. It also allows arbitrary metadata ("properties") to be versioned along with any file or directory, and provides a mechanism for versioning the 'execute' permission flag on files.

- *Commits are truly atomic.*

  No part of a commit takes effect until the entire commit has succeeded. Revision numbers are per-commit, not per-file; log messages are attached to the revision, not stored redundantly as in CVS.

- *Apache network server option, with WebDAV/DeltaV protocol.*

  Subversion can use the HTTP-based WebDAV/DeltaV protocol for network communications, and the Apache web server to provide repository-side network service. This gives Subversion an advantage over CVS in interoperability, and provides various key features for free: authentication, path-based authorization, wire compression, and basic repository browsing.

- *Stand-alone server option.*

  Subversion also offers a stand-alone server option using a custom protocol (not everyone wants to run Apache 2.x). The stand-alone server can run as an inetd service, or in daemon mode, and offers basic authentication and authorization. It can also be tunneled over ssh.

- *Branching and tagging are cheap (constant time) operations.*

  There is no reason for these operations to be expensive, so they aren't.

  Branches and tags are both implemented in terms of an underlying "copy" operation. A copy takes up a small, constant amount of space. Any copy is a tag; and if you start committing on a copy, then it's a branch as

well. (This does away with CVS's "branch-point tagging", by removing the distinction that made branch-point tags necessary in the first place.)

- *Natively client/server, layered library design.*

  Subversion is designed to be client/server from the beginning; thus avoiding some of the maintenance problems which have plagued CVS. The code is structured as a set of modules with well-defined interfaces, designed to be called by other applications.

- *Client/server protocol sends diffs in both directions.*

  The network protocol uses bandwidth efficiently by transmitting diffs in both directions whenever possible (CVS sends diffs from server to client, but not client to server).

- *Costs are proportional to change size, not data size.*

  In general, the time required for an Subversion operation is proportional to the size of the changes resulting from that operation, not to the absolute size of the project in which the changes are taking place. This is a property of the Subversion repository model.

- *Efficient handling of binary files.*

  Subversion is equally efficient on binary as on text files, because it uses a binary diffing algorithm to transmit and store successive revisions.

- *Parsable output.*

  All output of the Subversion command-line client is carefully designed to be both human readable and automatically parsable; scriptability is a high priority.

The improvements we are mostly interested in are the fact that commits are truly atomic, that the client/server protocol sends diffs in both directions and that costs are proportional to change size, not to data size. Most of the rest are facilities our system does not deal with, since it is only a prototype, not a production system (like CVS and Subversion are). This means we were only interested in efficient, user-friendly, merging of versions and efficient bandwidth use. The basic idea is that our system provides all of the three above mentioned as well (truly atomic commits, diffs in both directions and costs proportional to changes), which means it is, at least in these area, an improvement over CVS in the same way Subversion is.

The last thing to mention, however, is that [Subversion Home Page] states that, currently, their merge support is essentially the same as CVS's. This is where we tried to improve over Subversion as well and create a merge support which is much more usable by the user.

Beside CVS and Subversion, there are various other state-based systems which basically behave more or less in the same way. Therefore, there is no point in talking about them anymore as we would just be saying the same things all over again. Two examples of such other systems are BitKeeper (see [Bitkeeper Home Page]) and Microsoft Visual SourceSafe (see [VSS Home Page]).

### 2.1.2 Operation-based version control systems

The second category of systems we are going to discuss about are operation-based version control systems. These are all still research systems (i.e. they still lack most of the facilities which real users need in order to utilize them). However, they enclose very interesting ideas and, should they be fit for commercial use, they would most likely find a lot of adopters. We relate more closely to these projects both because our system is operation-based as well and because we have also only developed a prototype system, not a fully-fledged production one.

#### Safe Generic Data Synchronizer

In [Molli et al. 2003], the authors propose using the operational transformation approach to define a general algorithm for synchronizing a file system and file contents. The protocol they have developed allows the use the same algorithm for synchronizing both the files of the file system and the contents of those files. In order to achieve that, they propose a transformational approach which is somewhat close to the one we used in our project.

The model of transformational approach considers $n$ sites. Each site has a copy of the shared objects. When an object is modified on one site, the operation is executed immediately and sent to others sites to be executed again. So every operation is processed in four steps: (a) generation on one site, (b) broadcast to others sites, (c) reception by others sites, (d) execution on other sites. The execution context of a received operation $op_i$ may be different from its generation context. In this case, the integration of $op_i$ by others sites may leads to inconsistencies between replicas.

In the operational transformation approach, received operations are transformed according to local concurrent operations and then executed. This transformation is done by calling transformation functions. A transformation function $T$ takes two concurrent operations $op_1$ and $op_2$ defined on the same state $s$ and returns $op'_1$. $op'_1$ is equivalent to $op_1$ but defined on a state where $op_2$ has been applied.

The transformational approach defines two main components: *the integration algorithm* and *the transformation functions*. The integration algorithm is responsible of receiving, broadcasting and executing operations. It is independent of the type of shared data, it calls transformation functions when needed. The transformation functions are responsible for merging two concurrent operations defined on the same state. They are specific to the type of shared data.

We are less interested in the generic algorithm they propose, since it is fit more closely for real-time editing than for asynchronous editing. The idea that we want to underline, however, is that of using transformation functions for integrating remote operations executed in different context. We will apply the exact same idea in our project, with the distinction that we shall not address the matter of file system synchronization, but only that of file content synchronization.

The main difference between the two approaches is that they use a fixed working unit, the so called block while we use more flexible working units. Their operations are of the type addblock, deleteblock and moveblock. The transfor-

mation functions are defined accordingly for each pair of operations (addblock-addblock, addblock-delblock, addblock-moveblock, etc.). Our approach, on the other hand, allows for far more granularity in what the working unit is concerned. This has been on of our goals from the very beginning. We can work at paragraph, sentence, word or character level as the blocks of our text documents. Moreover, our transformation functions do not make any distinction between the levels of granularity. They work the same no matter whether they deal with paragraphs, sentences, words or characters.

An earlier work of the same main authors ([Molli et al. 2002]) relates in another way to our project, namely in that it introduces transformation functions that work on trees. Their interest, however, no longer lies in text editing, but in XML and CRC (Class, Responsibility, Collaboration) cards editing (both of which types of documents are viewed as tree structures). The operations they deal with here are CreateNode, DeleteNode, CreateAttribute, DeleteAttribute and ChangeAttribute. Therefore, they need to define 25 transformation functions (for each pair of operations). The paper describes, in essence, the same operation transformation mechanism as in the one mentioned before. The distinction lies in the fact that the prototype described here also allows asynchronous editing (however, details are not very abundant).

As we were saying, our interest with this paper comes from the use of tree operations. These are fairly close to the ones we use in our model. The distinction comes from the fact that in [Molli et al. 2002] the logs of local operations are not distributed throughout the tree as in our case. This means that the operations do determine changes to the tree structure, but they themselves are kept separately from the structure of the tree, which means that still all remote operations have to be transformed against all local operation, resulting in a performance loss.

A novel concept which is derived from using operations that work on trees is that the transformation of an operation might lead to its cancellation (we refer to this as turning the operation into a NOP (see 5.2.1)). We have used this concept in our project extensively as well.

### FORCE

[Shen and Sun 2002] proposes a flexible merging framework in which semantic merging policies are separated from the syntactic merging mechanism for asynchronous collaborative systems. In this framework, semantic merging policies are not restricted by the merging algorithms used in the syntactic merging mechanism, and the syntactic merging mechanism is flexible to support a wide range of semantic merging policies. This framework is illustrated in figure 2.1.

*The policy component* creates various semantic merging policies. These policies are application-dependent. For example, if the shared document is an article, the semantic merging policies could specify a set of spell and grammar checking rules. If the shared document is a computer program, the semantic policies could specify the programming language's syntax parsing rules. The essential observation is that if the work is properly divided among participants in the way that different participants play different roles, it is possible that their concurrent updates made to the shared document do not semantically conflict with each other, and therefore a new document could be generated to integrate all the updates made by different participants.

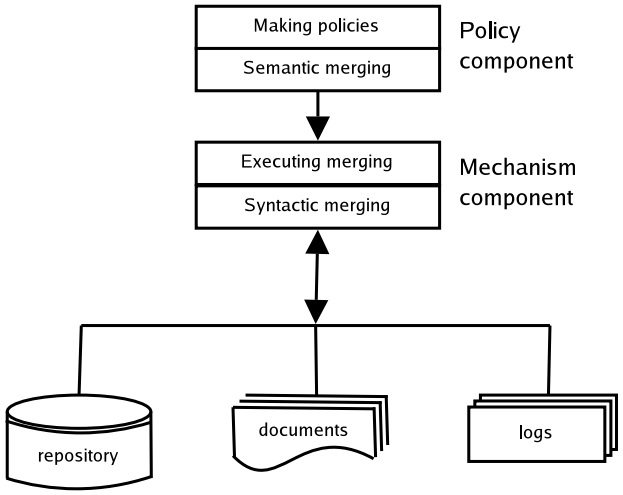


Figure 2.1: The FORCE flexible merging framework

*The mechanism component* executes syntactic merging according to the policy specified in the policy component. A remote update will be applied to a local working copy only if it does not semantically conflict with any update performed by the local user according to the specified policy. The mechanism component also includes some data structures: repository for storing versions, logs for storing updates and document working copies. These data structures are needed for the mechanism component to execute syntactic merging.

This is a very strong model which we have adopted ourselves. The policy component was kept fairly simple and is represented by the semantic conflict function described in section 5.3.2. We have also essentially adopted FORCE's basic merging algorithm (actually, adapted it to our tree representation of the text document and, more importantly, distributed log). It best suited our requirements since it uses operation-based merging and semantic rules for resolving conflicts. However, in [Shen and Sun 2002], the approach is described only for the linear representations 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 editing semantics.

The way we used the algorithm described in FORCE will be explained in detail in later chapters.

### dARB

Other researchers have also looked at tree representation of documents in the case of collaborative editing. The dARB algorithm [Ionescu and Marsic 2000] also uses a tree structure for representing the document, 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. They use an arbitration scheme for resolving the conflicts in the case of 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 create/delete word (this is actually also an insert character operation, but the inserted character is a space) are assigned a greater priority than the operation that modifies a character in a 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 the other user concurrently performs some modification in the original sentence, the user performing the split will win the arbitration and need to send the state of the whole paragraph that contains the sentence to all other users.

By allowing the customization of specifying the conflicts for various semantic units, our approach is more general than the one defined in [Ionescu and Marsic 2000] that considers that any concurrent operations performed on the same vertex of a tree are conflicting. Moreover, their approach defines operations (delete, insert) only at the character level in this algorithm, i.e. sending only one character at a time, thus the number of packages through the network increases greatly. 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).

**Work in our group**

Last, but not least, we strongly relate our work to that done in our group in the previous years, described in [Ignat 2002], [Ignat 2003], [Ignat 2004a], [Ignat 2004b] and [Nedevschi 02]. This work has been directed towards the development of a real-time collaborative text editing based on a tree structure similar to the one we used in the current project. One of the goals of our group is to integrate the two (now separate) editors into a single one where the user can choose at any time if (s)he wants to work in an asynchronous manner or in a synchronous one.

The purpose of the project developed two years ago was to develop an efficient and reliable concurrency control algorithm, and to implement a simple collaborative real-time editor relying on this new algorithm.

Studying the existing algorithms constituted the first part of the project. Extensive testing which include implementation of some of the algorithms was employed. This lead to the discovery of faults in some of the algorithms. The entire process is described in detail in [Nedevschi 02]. The second part of the project was concerned with the development and implementation of the algorithm itself, aiming at maintaining not only the syntactic consistency, but also the semantic consistency. This work was based on an already existing linear model, which, unfortunately, proved to be faulty in some special cases. This, however, was only discovered towards the end of the project and the issue has not yet been solved (neither by our group nor by other researchers).

The work presented in this paper comes as complementary to the one done two years ago, aiming to contribute to achievement of the final goal of the group, namely 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 [Ignat 2004b].

## 2.2 Motivation

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 either implement the synchronous mode of communication or the asynchronous mode of communication, but not both of them. Since different user groups have different conventions and requirements, software engineering as well other engineering domains require means of customizing the collaborative work. An integrated system supporting both synchronous and asynchronous collaboration is needed because these two modes of communication can be alternately 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 line of thought, in order to contribute to the group's work towards achieving such a dual system, we have taken upon ourselves the task of developing the module that deals with the asynchronous mode of communication. What asynchronous editing means and the concepts involved will be discussed in detail in chapter 3, so we shall not cover them here. What we shall try to do here instead is explain why the asynchronous editing mode is necessary and illustrate this with some typical use cases.

#### Hiding draft work from others

One can easily imagine situations when users want to obtain a certain final version of their work before showing it to others. There could be several reasons for wanting this, among which we could mention:

- intermediate work might be incorrect. Think about a programmer working on an algorithm. It is obvious that most of the times the initial code (s)he writes is incorrect and should not be seen by the others. Under such circumstances, the author will probably prefer to develop and test the code first and only when (s)he thinks it is correct submit it to the repository.

- intermediate work might be not make sense until it is complete. Think about the case when someone writes some paragraphs in a mixed order and decides to only rearrange them in the correct order at the end, when (s)he has written all of them. Until (s)he does that, someone else reading the paragraphs will be taken aback by the lack of logic. It is probably desirable to avoid that.

- intermediate work might be just an attempt. If someone is not sure about adding something to a document, but starts writing it though with the intention of deciding whether to keep it or not later on (depending on the outcome), it would probably be best for others not to see this work until a decision to keep it has been made by its author.

By using asynchronous editing mode, authors can simply keep changing the document without the other group members being able to see any of the modification and only commit it (make it available for the other to see) only when it is complete.

**Increased efficiency**

In a vast majority of cases group members working together on a document each modify different parts of it. The most obvious example is that of two or more authors working together on a book. Almost always each author will work on his/her own chapters and not interfere with what the other authors are writing. In such cases, it is common sense that it would make no sense to keep constantly synchronizing the versions of the document when it is most likely that the modification would not even appear in the part of the document the user is looking at/working on (only so much of a document can appear on the computer screen at one time).

In such cases, it makes no sense to waste bandwidth and processing resources in order to achieve something nobody really needs. Whenever one author needs to read something the other authors have written, (s)he can simply update his/her version then and obtain the modifications. Aside from bandwidth and processing power, this is also more efficient from the following point of view: when somebody is writing something, it is very common that entire sections get deleted, modified, moved around, rewritten and so on. If we were to use synchronous communication, all these actions would be mirrored in all the documents on all sites. If, however, asynchronous communication is used, a log compression method can be employed (see section 5.4.1) in order to cancel out pairs of insert/delete operations that first insert and then delete the same thing. In this way, a minimal set of operations which achieve the same effect can be sent over the network.

**Offline use**

Another reason for supporting asynchronous communication is for coming to the aid of dial-up users (which, if we analyze statistics, represent way over 50% of Internet users nowadays). When using synchronous communication, there is no way in which users can work together without being connected to the network, since communication between sites occurs constantly. This is no longer the case

with asynchronous editing, because the user him/herself gets to decide when communication is to take place.

This way, the user can simply log to the Internet when (s)he wants to start working, check out the current version of the document and then log off from the Internet and keep working without being connected. After a certain period of time, the same user can re-log and commit his/her changes. Asynchronous communication makes this possible. And there is certainly a lot of potential in this given the large number of users which still use dial-up. Aside from that, there are also cases when network failures might actually prevent all users from synchronizing with each other. This would effectively render a synchronous editor unusable. However, asynchronous editing can go on as if nothing had happened. The only problems that might arise would be delays in the transmission of the changes, but there is no way to avoid this as long as the network is down.

Such a benefit is of even greater importance to the users who actually do not need to see what other users are doing during the time in which they are working, as it would distract them. Take programmers, for example: if someone is developing a stand-alone module which is to be used by other then that someone will certainly not be interested in changes that others are making to other functions, for instance. (S)he simply wants to focus on his/her work and submit it when it is ready. Asynchronous communication is ideal for such style of work.

#### Availability of changes history

Asynchronous editing generally follows a client-server model. In this case, the server is known as a repository, because it is where all the documents are kept. The essential difference between a database server or a file server and a document repository of an asynchronous editing system is that the repository stores not just the last version of the document, but actually all the versions starting from the very first one. The advantage of such a system is that at any time any version of the document can be retrieved and reviewed.

This is not the case with synchronous editors since in such systems there is usually no central repository that stores all changes ever made to the document. The changes are only broadcast to all sites that work together at one time and, at the end, just one final version is stored (overwriting the old one). There are numerous cases, however, when old versions of the document are needed, such as:

- in cases errors appear, older working versions represent a safety net one can fall on. This is especially common in software development where certain modifications can render parts of the (or even the entire) system unusable.
- someone might want to find out (in an architecture project, for example) who was responsible for some ill decisions which might have lead to current problems.
- coming back to software projects, it is often necessary to retrieve an older version of a program for example in order to install it on an older, less capable system.

- there are cases when some parts of a document are deleted and the authors realize later that it would have been better to keep them. Since all intermediate versions are still available, it is fairly easy to find those deleted bits and reinsert them in the document.
- version history is also useful for statistical purposes. Project managers frequently need to have an overview over the development of the project in order to make further decisions. Having an automated version history, it is relatively easy to obtain such an image of the evolution of the project.
- various other cases when having access to older versions can be valuable.

It is hopefully clear that this is a great benefit which asynchronous systems alone can provide.

Given what we have spoken about above, we hope that there is no further doubt in the mind of the reader that efforts towards developing an asynchronous system are definitely not in vain. Moreover, in addition to simply developing yet another asynchronous editing system, we have strived (by analyzing what other before us have done, building upon their achievements and learning from their mistakes) to actually come up with a system which would be valuable to users also in terms of performance, not just in terms of what we have mentioned in this section.

# Chapter 3

# Basic Concepts in Asynchronous Editing

The purpose of this chapter is to briefly introduce several basic concepts which are strongly related to the field of asynchronous editing. All these concepts will be widely used throughout the remaining chapters and, therefore, need to be properly (and, if appropriate) rigorously defined before being actually used. The goal is to sufficiently accustom the reader with notions like commit, update or check out (as well as many others) in order for him/her to have the necessary basis to understand the rest of this paper.

## 3.1 Version

The most basic concept which one comes across when dealing with asynchronous text editing is known as the *version* of a document. A version represents the state of the document at a very precise moment in time. Generally, the version of the document changes every time a user "tells" the system that (s)he has brought one or more changes to the document and desires to make the system (and, through the system, the other users) aware of his/her changes. Each version of the document is given a number and a timestamp, both of which uniquely identify it. Version numbers are generated in increasing order. The *current version* of the document refers to the version with the greatest number from the set of all available versions.

As a side note, various existing version control systems (see 3.2) also introduce more advanced concepts, such as those of *subversion* or *revision* in addition to that of version. Since such concepts are not used anywhere in this paper they will be silently ignored.

## 3.2 Version control systems

In [Shen and Sun 2002] *version control systems* are introduced as "widely used asynchronous collaborative systems in team-working environment, where document merging is a key function." Essentially, the purpose of a version control system is to provide its users with a history of document versions which dates

back to the beginning of the editing of a document, allow them to retrieve any of these versions, modify them (ideally, in a concurrent manner) and commit their changes so that others can become aware of them as well. As mentioned before, versions are usually identified by their date or by a version number. Generally, the system should allow the retrieval of any of the two identifiers provided that the other one is known.

For the particular case of software development, a version control system allows programmers involved in the project to keep track of all the changes they have made to their files during the development process. This is especially useful when, after several changes, certain functionalities ceased to behave as expected or certain bugs seemed to have appeared. In such cases, a version control system acts as an supplemental insurance that the already achieved goals will never be lost.

A rather interesting description of *version control* itself can be found in [Collins 2004] and reads:

> Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing those changes the next day. But the usefulness of version control software extends far beyond the bounds of the software development world. Anywhere you can find people using computers to manage information that changes often, there is room for version control.

The basic configuration of a version control system can be illustrated by Figure 3.1. As shown, the main actors in a version control system are the *repository* and the *client*s, the functions of which will be detailed in the subsequent sections. Essentially, the repository is a container of all versions of the document, while the clients connect to the repository through a network (may it be a LAN, the Internet or any other type of network) in order for them to retrieve and submit document changes (and thus work together as a group). Clients hold local reflections of some version of the document from the repository (also known as *working copies*) on which they make all their modifications.

## 3.3 Repository

A rough definition of a *repository* could be:

**Definition 3.1** *A* repository *is a store of items that typically are fetched in order to perform some task. Items in a repository (such as a document) would be retrieved in order to be used in their own right. In contrast, data in a database might be used to compute statistics, or to verify access, or retrieve information associated with a triggering event, rather than used as an artifact in their own right.*

Particularizing this definition, we could refer to a version control system repository as the component of the version control systems which holds all the versioned data. It is, in essence, a *sort of* file server which, in addition to basic file server functions, also memorizes all the intermediate versions of the files it handles. A repository should provide at least the two following services for its clients:

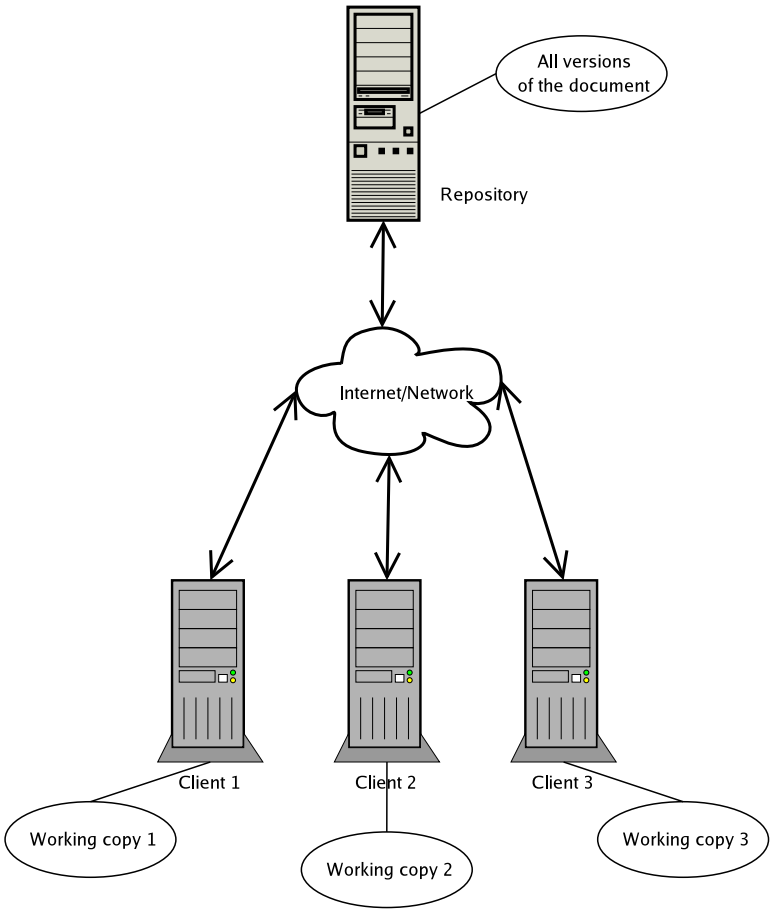


Figure 3.1: Configuration of a basic version control system

- retrieval of any version of any of the files it stores
- submission of changes to any of the files it stores from any of its clients and record these changes as a new version of the document

Apart from these two basic functions, a repository can additionally offer a number of optional ones, such as:

- retrieval of the version number of a document based on the date at which it was created
- retrieval of the date at which a certain version of a document was created
- support for directories (in addition to files)
- storage of the name of the person who has brought certain changes to a file
- etc.

## 3.4 Repository client

The *repository client*s are the actual users which connect to the repository and make use of the functions it provides. Technically, any number of clients can connect to the repository and then read or write the stored documents. A client has to read data from the repository in order to find out about the changes performed by other clients on the document (this operation is referred to as an "update") and write data to the repository in order to "publish" their own changes (known as the "commit" operation).

## 3.5 Versioning models

Note: the ideas presented in this section are a slightly modified form of those described in [Collins 2004].

Virtually all version control systems are faced with the same fundamental question: how can they allow all users to access the same document and still ensure that the changes made by all users are preserved. In other words, how can one make sure that the changes one user makes will not disappear in the moment another user *simultaneously* makes other changes? Such cases frequently appear when two or more users work on the same document at the same time.

If no special care is taken we could be faced with a situation as the following: two users work concurrently on the same document, each of them making his/her own changes. One of the users will obviously send his/her changes to the repository first. Later, the other user will commit (different modifications, naturally) as well. At this point the changes performed by the first user will not be included in the current version since the second user was not aware of them (meaning that the copy of the document he had been working on did not contain the changes made by the first user). The first user's changes would appear in the history of the document (should one be interested enough to look for them), but still, they would be missing from the latest version. This is obviously not the desired behavior of such a system.

### 3.5.1 The Lock-Modify-Unlock Solution

The most immediate solution to the shared document problem discussed above that comes to mind is to simply not allow more than one user to make modifications to a document at any given time. This can be easily achieved by employing a locking policy. This would essentially mean that when a user wants to edit a document (s)he is only allowed to do so if the document is not currently locked. If the document is not locked, this user will lock it (and thus make it impossible for any other user to gain access to it until it is unlocked by the same user who locked it in the first place) and then be able to modify to it. Once (s)he has finished working with the document, (s)he will unlock it so that other users can access it as well. If the document is locked at the time some user wants to access it, the user will simply have to wait until the user currently editing the document decides to unlock it.

The *Lock-Modify-Unlock* policy certainly has several severe drawbacks, easily noticeable even to someone only slightly accustomed to systems based on this

policy. Here is a short list enumerating just some of them:

- only one user can work on a document at a given moment in time. This is a serious limitation, especially for cases when users want to work on totally different parts of the document, so they would actually not interfere in any way.
- the user holding the lock could simply forget to release it once he is finished doing his/her modifications, thus blocking all other users from accessing the document. This could lead to serious delays for groups working together. Also, network failures could also lead to the impossibility of a user releasing the lock with the same undesirable consequences.
- there is no way to allow access to documents based on priorities if the user who has locked the document goes offline. Suppose the group policy states that the group leader should have priority access to the documents on which the group works, but a certain member of the group has locked on ore more of the documents and then went offline. There is no way in which the system can forcibly take the lock away (supposedly using a mechanism similar to preemption) from the user holding it and give it to the group leader.

In light of such limitations, it is clear that a new model had to be employed.

### 3.5.2 The Copy-Modify-Merge Solution

Most modern version control systems are built to work in the manner of *Copy-Modify-Merge* as an alternative to locking: each user checks out a separate working copy from the repository by making a local image of the original and does all his/her modifications on this copy. This is done independently of all other users which might be working on their own working copies of the same original version. When editing is finished, two situations can arise. If only one user has worked on the document, then his version is simply stored on the repository as the new version. If, however, two or more users worked concurrently on the document, their modifications are merged together (either completely automatically or partially automatically and partially by hand) into a new version (actually, a different version is created on the repository for each user, but each successive version integrates the modifications introduced by all versions previous to it regardless of whether they were made concurrently or not).

As mentioned in [Ignat 2004b], the *Copy-Modify-Merge* technique consists basically of three operations applied on a shared repository storing multiversioned objects: checkout, commit and update. These three fundamental operations can be defined as follows:

**Definition 3.2** *A* checkout *operation is the operation by which a client creates a local working copy of a user-specified version of an object (more specifically, a document) from the repository.*

**Definition 3.3** *A* commit *operation is the operation by which the user submits a modified version of his/her working copy of an object from the repository back to the repository, with the intention of making it the current version. The*

*commit operation is successful only if the repository does not contain a more recent version of the object than the one the original working copy of the client trying to do the commit was a replica of.*

**Definition 3.4** *An* update *operation performs the merging of the local working copy of the object with the latest version of that object stored in the repository.*

The most essential, and at the same time difficult, function of asynchronous collaborative systems, the one which actually differentiates one version control system from another, is the *merging* done during the update phase. This is the process of integrating two documents (which are usually modified versions of the same original document) in order to generate a new document that contains as many of the changes that appear in both documents as possible.

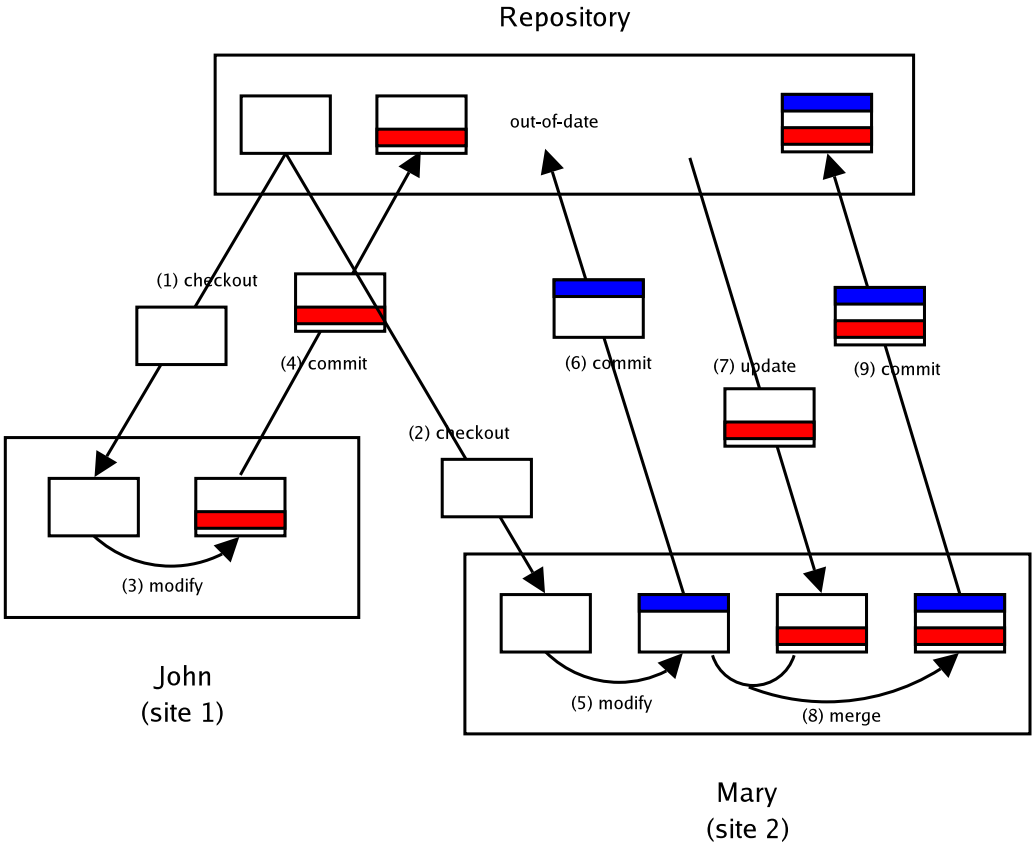


Figure 3.2: A Copy-Modify-Merge scenario

In order to clarify the inner workings of the *Copy-Modify-Merge* mechanism, we illustrate a typical usage scenario in Figure 3.2. In this example, John (the user at site 1) and Mary (the user at site 2) both checkout the same original version of a document from the repository roughly at the same time, thus creating two, initially identical, working copies, each in his/her own private workspace.

These are denoted as steps (1) and (2) in the figure. Next, John does his modifications on his working copy (operation (3) in the figure) and obtains a modified version of his working copy. When he considers that his work is finished, he goes on to performing operation (4), namely commit his modified version of the document to the repository. As the current version on the repository is the same one as the one John's original working copy was replicating, the commit step ends successfully (see Definition 3.3) and the current version on the repository changes to John's local working copy. In parallel with John, Mary does her own modifications to her working copy, obviously oblivious of the ones John has made (step (5) in the figure). After finishing her work she tries to commit her own changes back to the repository (operation (6)). Her attempt to commit is unsuccessful because, as the repository puts it, her version is out-of-date. This translates to the fact that the version she modified is no longer the latest version on the repository. Therefore, as Definition 3.3 stipulates, her commit is bound to be unsuccessful. In order to overcome this problem and actually be able to commit her changes, Mary will first have to do an update (step (7) in the figure). By doing so, she becomes aware of the modifications John has made (and successfully committed) and merges (operation (8)) them into her own local working copy (actually, merges her working copy with the latest version of the document on the repository and obtains a new document which becomes her current working copy). Thereafter, she retries to commit her changes (illustrated as step (9)) and, as the working copy she is now considered (by the repository) to have started working from is a replica of the latest version of the document on the repository, she succeeds. As one can see, the current version of the document on the repository after both John and Mary have successfully committed contains both of their modifications, which is what we wanted in the first place.

As mentioned before as well, the essential operation in this model is the *merging* operation. Two distinct ways of doing a merge between two documents (usually versions of the same original document) are effectively employed in production and research systems today: state-based merging and operation-based merging.

*State-based merging* is a way of merging two documents that only uses information about the states of the documents, without requiring any information whatsoever about how one state evolved into another. When a user edits some version of a document, all that he will have in the end will be the final version (or state) of the document, without the list of changes that have transformed the initial state into the current one. In order to merge the two states of the document (in the case of an update, for instance) a comparison algorithm (such as diff, for example) is used to compute a delta which encapsulates the difference between the two states. This delta is then applied on one of the document's states in order to generate the document state that represents the result of the merging.

*Operation-based merging*, on the other hand, works in a completely different manner. It essentially stores the information about the evolution of one state of the document into another. This information is kept under the form of the list of operations that have been performed in order to transform the initial version of the document into the current one. Merging, in this case, is done by applying an algorithm that computes the form in which the set of operations associated with one version of the document have to be applied on the second

version of the document in order to achieve the same effect as if their original form had been applied on the original version of the document (the original version from which both versions we are now trying to merge emerged). Once this form is obtained, the operations (in their modified form) are simply applied on the second document in order to generate the version representing the result of the merging.

Merging happens at two stages throughout the model, at the committing stage and at the update stage ([Shen and Sun 2002]). Merging at the committing stage takes place when a user wants to commit his changes and the repository must modify the latest version of the document so that it reflects the modifications made by the user currently committing. This kind of merge is actually a very trivial, particular case of the general merge in which all that we have to do is reapply the operations executed locally by the user on the document version on the repository (if dealing with operation-based merging) or simply add the new state of the document (as committed by the user) as the latest version (if dealing with state-based merging). Merging at the update stage takes place when a commit is unsuccessful due to the out-of-date problem. Therefore, the user trying to commit has to do an update first in order to be then able to recommit. As part of this update operation, the user has to merge the version received from the repository with his/her own working copy. This kind of merging (at the update stage) is no longer trivial since the two documents that have to be merged both brought some modifications to a common original version and both sets of modifications have to appear in the final version. The way to do this has been intuitively described above (both in the state-based and in the operation-based approach). A more detailed explanation of how this is actually done in the case of operation-based merging is in fact the subject of a large part of this paper.

# Chapter 4

# Issues Regarding Operation-Based Merging

As we mentioned in section 3.5.2, *operation-based merging* is the technique of merging two document versions into one by working on two logs of operations (each of which describes the set of operations an original version of the document has gone through in order to transform itself into its current form). In this chapter we take a closer look to what *operation-based merging* is, the specific issues associated with it and what its advantages over its "competitor", *state-based merging*, are. We will also introduce the notions of inclusion and exclusion transformation in the context of *operation-based merging*, explain why they are necessary and define both their actual implementation and some functions based on these transformations that will be used in our algorithm.

## 4.1 Operations

In an operation-based model, the *operation* represents the fundamental concept the whole system revolves around. Simply put, an operation is an action the user can perform in order to modify a document (to change it from one state to another). Generally, by means of an operation a certain part of the document is changed, either by adding something to it or by deleting something from it or by changing its properties. Depending on the type of editor we are dealing with, an operation can take very different forms. In a text editor, for instance, example of possible operations would be *insert character* and *remove character*. If, on the other hand, we are working with a graphical editor, example of possible operations are *create figure*, *change foreground color* or *delete figure*. Operations can also be composed in order to create higher level operations (a series of *insert character*'s, for example, could be combined to form a single *insert word* operation).

The principle behind operation-based merging is that the delta between two versions of a document is stored as a list of operations which have been executed on the original version of the document in order to obtain its current version. In this context, we preferred using the operational approach instead of the state-based one in this project because operations are more finely grained than the deltas generated by state comparison (as it is common to do in state-based

merging). This is because their effect region can begin at any character in a document and contain an arbitrary number of characters (instead of only being able to refer entire lines, for example). Moreover, by using the operation-based approach, the activity of the users can be tracked, because the operations they perform are stored in a log, thus maintaining information about the evolution of the document from one state into another.

In what text editing is concerned, there are two general types of primitive operations: *insert* and *delete*, denoted (in a generic manner) as *Insert*/*Delete*[p, s] (inserting/deleting string s at/from position p in the document). We will use this notation later on in this chapter when discussing the inclusion and exclusion transformations.

Another very important operation-related concept (in what cooperative editing is concerned) is known as the *intention* of an operation. The following definition of the term is taken from [Sun et al. 1998]:

**Definition 4.1** *The* intention *of an operation O is the execution effect which can be achieved by applying O on the document state from which O was generated.*

For the sake of clarity, we mention that the effect of an operation exclusively refers to its syntactic effect (meaning the insertion and deletion of characters at specific positions in the document) and not to its semantic effect (for example, changing a verb from present to past or vice-versa). What the concept of the intention of an operation is needed for will become obvious in the following sections.

## 4.2 A consistency model

Previous theoretical approaches that have looked into the specific issues of cooperative editing systems (see [Sun et al. 1996]) have shown that there are several properties that such a system has to maintain in order for it to be considered consistent. However, before actually discussing these properties, let us first try to explain how a cooperative editing system can become inconsistent if special care is not taken.

### Divergence

The problem of divergence especially appears in real-time cooperative systems, but we can easily show that asynchronous systems have to deal with it as well. The divergence issue usually arises in real-time cooperative systems when operations arrive at different sites in different orders, resulting in different final results. For instance, if users at two sites are working on the same document and they each just execute a single operation and send it to the other one, the results will probably be different (unless the operations are commutative, but this is usually not the case). This is because at site 1 operation $O_1$ (the operation executed at site 1) is executed first and operation $O_2$ (the one from site 2) second, while at site 2 the order of execution is reversed ($O_2$ first and then $O_1$).

A similar situation can occur in an asynchronous configuration as well. Take, for instance, the case when two users (John and Mary) edit the same document and John commits before Mary (this is *always* the case since no two - or more -

users can commit at the same time). Before the Mary can commit herself, she will have to do an update (see section 3.5.2) in order to integrate the changes John has made. But Mary has obviously made her own changes to the document (i.e. has executed a set of operations on the document). Now, let us suppose that she simply executes John's operations on her working copy and then commits it to the repository. John, on the other hand, wanting himself to update his own version (so that it contains Mary's changes), will, say, take the current version from the repository (the one containing Mary's changes) and execute the operations therein on his working copy. We have come to the exact same case as with real-time editing, when both users have executed the same operations, but their versions of the document differ. This is because they have executed the operations in different order and the operations are *not* commutative.

The divergence problem is usually solved in real-time collaborative editing systems by means of a serialization protocol, so that operations are sure to be executed in the exact same order on all sites. This is, however, not possible in the case of an asynchronous system because executing operations in the same order on all sites usually implies an undo-execute-redo scheme ("undo" of (local) operations already executed, "execute" of operations that should precede them in the serial order, and "redo" of the undone operations; this scheme is necessary because of the need to ensure good local responsiveness, which implies executing the local operations as soon as they are generated, without waiting for those remote ones which might precede them in the serial order). The undo-execute-redo scheme is impossible to apply in asynchronous systems because operations on the repository cannot be undone and interleaved with others since this would lead to further inconsistencies between clients (some might have updated with one set of operations and some with another, modified, set of operations and all would think that they have the same version of the document when, in fact, this wouldn't be true). Different techniques thus have to be found in order to ensure convergence of document versions when dealing with asynchronous editing. The way our system solves the divergence problem will be detailed in chapter 5.

**Causality-violation**

Due to the nondeterministic communication latency, operations may arrive to a site and be executed out of their natural *cause-effect* order. For instance, consider the case of two operations, $O_1$ (generated on site 1) and $O_2$ (generated on site 2), such that $O_2$ is dependent on $O_1$ (i.e. site 2 had already received $O_1$ when $O_2$ was generated). Suppose that, due to different communication delays over the network, $O_2$ arrives at, say, site 3 before $O_1$. This is a typical case of causality-violation, because $O_2$ might need some effect that $O_1$ produces in order for it to make sense (let us assume that $O_1$ is the insertion of a sentence in a document and $O_2$ is the insertion of a word inside that sentence; it is obvious that if the sentence does not exist, it makes no sense to try inserting a word in it).

This type of inconsistency is particular to real-time collaborative editing systems only and thus does not appear in the case of asynchronous systems. The reason for this is that communication in the asynchronous case takes place only between two end points (the client and the repository). And when multicast does not appear, a situation like the one depicted above is impossible to appear. When only unicast is involved, operations that one site (the client/the repos-

itory) sends to the other (the repository/the client) are sure to arrive at the destination in the same order in which they were issued by the source (*Note: this statement no longer holds if the transport protocol used for communication does not guarantee in order delivery of messages*).

Given the fact that this paper is dealing with asynchronous editing systems, we shall pursue the matter of causality-violation no further, referring the reader to [Sun et al. 1996] for additional information.

**Intention-violation**

Due to the fact that operations are generated concurrently, the effect of an operation when executed on a site may be different from the intended effect of the operation at the time of its generation. This means that the user who generated the operation might have wanted to insert a letter in a certain word (say), while the *actual* effect of the execution of that operation (on a different site) might be the insertion of the letter in a different word. This violation of user intention comes from the fact that the context in which the operation was generated and the context in which it is executed are effectively different (here, we use context as an equivalent of document state). For example, if an operation $O_1$ is generated on a document in state $S_1$ and later executed (in the exact same form) on the same document, but in state $S_2$ (which is obtained, for instance, by executing operation $O_2$ in state $S_1$), it will possibly achieve a different effect than intended because the position argument of $O_1$ (remember that we consider an operation in a text editing system to be of the form Insert/Delete[p, s], with p the position and s the string to insert/delete) might no longer refer the same logical position in the document. More concrete examples of this can be found later in this chapter, when inclusion and exclusion transformations are introduced.

Although a clear distinction between divergence and intention-violation might be a bit difficult to see, still one does exist. The essential difference between the two comes from the fact that divergence can always be avoided by using a serialization protocol while this does not hold in the case of intention-violation (i.e. as long as operations are executed in their original form there is no serialization protocol which can ensure intention-preservation).

In light of the problems described above we can finally return to defining the consistency model mentioned in the title of this section. The original form of the definition (as it appears in [Sun et al. 1996]) is appropriate for real-time systems. We shall firstly present it in this form and then make some notes regarding aspects peculiar to asynchronous systems.

**Definition 4.2** *A cooperative editing system is said to be consistent if it always maintain the following properties:*

1. ***Convergence**: when the same set of operations have been executed at all sites, all copies of the shared document are identical.*
2. ***Causality-preservation**: for any pair of operations $O_a$ and $O_b$, if $O_a \rightarrow O_b$, then $O_a$ is executed before $O_b$ on all sites.*

3. ***Intention-preservation****: for any operation $O$, the effects of executing $O$ at all sites are the same as the intention of $O$, and the effect of executing $O$ does not change the effects of independent operations.*

*Note: "$\rightarrow$" denotes the* causal ordering relation*, the exact definition of which is beyond the scope of this paper. For an intuitive explanation of the concept, please refer to the example given in the subsection called "Causality-violation".*

In a nutshell, the *convergence* property ensures that users on all sites will eventually have the exact same final result once the editing session is finished; the *causality-preservation* property guarantees an ordering of the execution of operations which takes causality into account; while the *intention-preservation* says that the effect achieved by the execution of an operation at a remote site is identical to the one achieved by its execution at the originating site and also that the effects of independent operations do not overlap. The consistency model is both a promise to the user and a contract for the developers of cooperative systems.

As we have shown previously, in the case of asynchronous systems the problem of causality-preservation no longer appears due to the nature of communication between the end-points in the system. In what divergence and intention-violation are concerned, however, these problems do arise in the asynchronous case as well. The way we have solved these issues in this project will be discussed in detail in chapter 5. In what follows we will only try to explain it in just a few words for the sake of completeness. During this explanation, the concept of inclusion is used, so you might want to jump to section 4.3, read that first and then return back here and read on.

The basic idea is that we do not use a serialization protocol in order to apply operations in the same order on all sites, but instead we work with transformations of the sets of operations. The situation we always have to deal with in the asynchronous system is that of a client doing an update from the repository. Thus, we always have two sets of operations we have to merge (the local set of operations and the remote set of operations). The local operations have already been executed locally and they will not be undone. What we do is we transform both sets of operations (the local and the remote set) so that each of them properly includes the effects of the operations in the other one, thus obtaining two modified sets. For purposes of this discussion, we shall call these two modified sets the *transformed local log* and the *execution form of the remote log*. Once this transformation of operations is completed, the execution form of the remote log is executed locally in order to update the local working copy with the changes from the repository. Since the remote operations have been modified in order to include the effect of the local operations, intention-preservation is ensured (see next section). On the other hand, the transformed local log is sent back (committed) to the repository. This log contains the operations that had been executed locally, but modified so that they include the effect of the operations from the repository. By doing this we ensure that the intention of the local operations will be preserved as well when they will be executed by another client (again, see next section).

By applying this technique we not only achieve intention-preservation, but convergence as well. In order to understand this fact, consider the cas of two users working in parallel (John and Mary). Suppose they both start working from the same version of the document. John does his changes, Mary does hers

and then they both want to update in order to integrate the changes of the other. Let us suppose Mary commits her changes first. John will have to start by updating his working copy in order for it to include Mary's changes and then commit his own changes to the repository. When updating, the algorithm will transform Mary's operations so that they include the effect of John's. But in the same time, it will change John's operations so that they include the effect of Mary's. John will then execute the execution form of Mary's operations on his working copy and commit his own transformed local log. When Mary finally updates, the repository will send her John's transformed local log, which she will execute on her own working copy of the document. At this point, they both have the same version of the document. The convergence of the two copies is ensured by means of the merging algorithm, which guarantees that the sets of operations will be transformed in such a way that the same effect will be achieved on both sites. Why this is indeed true will become clearer once you read the description of the merging algorithm from chapter 5.

## 4.3 Achieving intention-preservation

Achieving intention-preservation is much harder than achieving convergence (and causality-preservation). This is because the intention-violation problem is not related to the execution order of operations and cannot be resolved by just rescheduling operations, as in the case of achieving convergence and causality-preservation. To achieve intention-preservation, an operation has to be transformed before its execution in order to compensate the changes made to the document state by other executed operations [Sun et al. 1996]. To better understand why this is so, let us begin this section by offering two real life examples of intention-violation.

### 4.3.1 The "inclusion" problem

Suppose that the initial version of a document is:

| |
|---|
| We dance and the music dies.<br>We run through the stars.<br>We are without excuse. |

Also, suppose that two users start working on this document in the same time, creating the following two different version:

| John | Mary |
|---|---|
| We dance and the music slowly dies.<br>We run through the stars.<br>We are without excuse. | We dance and music dies.<br>We run through the stars.<br>We are without excuse. |

which, in terms of operations, would be translated to:

| John | Mary |
|---|---|
| Insert[23, 'slowly '] | Delete[13, 'the '] |

Suppose now that John commits his changes to the repository, i.e. sends a list of operations containing just one entry: {Insert[23, 'slowly ']}. Mary, wanting

to integrate John's changes, does an update and receives from the repository the exact same list of operations that John previously committed, i.e. {Insert[23, 'slowly ']}. Say Mary would simply execute the operations from the list (in this case, the single operation) in the form in which she received them. The result she would obtain would be:

We dance and music diesslowly .
We run through the stars.
We are without excuse.

instead of:

We dance and music slowly dies.
We run through the stars.
We are without excuse.

This is because the *context* in which Mary executes John's operation is different from the *context* in which the operation was originally created. The context has changed because Mary has executed an operation of her own (i.e. the Delete[13, 'the '] operation), thus changing the state of the document. Therefore, position 23 from John's operation no longer refers the same *logical* position in the document as John intended. The actual effect of this is the insertion of the string 'slowly ' on position 23 in Mary's context, which is right after the word 'dies', resulting in the wrong sentence. This is a typical case of user intention-violation. The obvious solution to this problem is to modify John's operation so that it *includes* the effect of that of Mary's. In this case, what the system has to do is subtract the value 4 (the length of the deleted string 'the ') from position 23 and thus change the insert operation to Insert[19, 'slowly ']. Applying this operation in Mary's context would lead to the correct version of the document.

This is essentially an example of the "inclusion" problem.

### 4.3.2 The "exclusion" problem

This time, let us focus on what happens on just one site, since in the case of the "exclusion" problem this will suffice. Again, let us start our discussion from the following initial state of a document:

We dance and the music dies.
We run through the stars.
We are without excuse.

The user deletes the word 'the' in order to obtain:

We dance and music dies.
We run through the stars.
We are without excuse.

and then inserts the word 'slowly' to reach this form:

We dance and music slowly dies.
We run through the stars.
We are without excuse.

or, in terms of operations:

| Delete[13, 'the ']<br>Insert[19, 'slowly '] |
|---|

Now, for reasons independent of our user, something happens and his first operation, Delete[13, 'the '] is found to be conflicting with some other user's operation and has to be canceled (reasons why this might happen will be given in chapter 5; for now, let us just assume that it has to happen). Under these circumstances, if the second operation, Insert[19, 'slowly '], which was not conflicting, and thus did not have to be canceled, is kept unmodified, a re-execution of the local log, currently of the form {Insert[19, 'slowly ']}, on the initial state of the document, would yield:

| We dance and the muslowly sic dies.<br>We run through the stars.<br>We are without excuse. |
|---|

which is obviously wrong. However, this is exactly what a different user (virtually) executing this list of operations (a list, in this case, containing just one operation) would obtain. The reason why this happens is, once again, the fact that the context in which the operation was generated and the context in which it is executed are essentially different. The resolution, however, is different here, since it no longer involves including the effect of another operation into the current one, but *excluding* it. In our case, the effect of the operation Delete[13, 'the '] has to be *excluded* from the operation Insert[19, 'slowly ']. The latter would then become Insert[23, 'slowly '] (we have added 4 - the length of the deleted string - to the value 19 and obtained position 23). Apply this transformed operation effectively preserves the user intention:

| We dance and the music slowly dies.<br>We run through the stars.<br>We are without excuse. |
|---|

This second example illustrated the typical case of the "exclusion" problem.

### 4.3.3 Context of operations

Now that we have outlined a rough idea of why inclusion and exclusion transformations are necessary we shall next try to approach the problem from a more theoretical perspective and properly define the above mentioned transformations this way.

The first term that appeared in the examples above was that of *operation context*. Conceptually, an operation $O$ is associated with a *context*, denoted as $CT_O$, which is the list of operations that need to be executed to bring the document from its initial state to the state on which $O$ is defined. The importance of a context is that the effect of an operation can only be interpreted correctly when it is executed in the context in which it was defined. More rigorously, [Shen and Sun 2002] defines this concept as:

**Definition 4.3** *Given an operation $O$, its* context*, denoted as $CT_O$, is the document state on which $O$'s parameters are defined.*

Given an initial document state, $S_0$, and a list of operations $L = [O_1, \ldots, O_n]$ performed on $S_0$, the current document state is denoted as $S_0 \circ L = S_0 \circ [O_1, \ldots, O_n]$. So $CT_O = S_0$, $CT_{O_i} = CT_{O_{i-1}} \circ [O_{i-1}]$ $(1 < i \leq n)$. The context of an operation can be changed by explicitly applying the inclusion and exclusion transformations (see 4.3.4).

A concept related to that of operation context, and one which will allow us to specify the pre- and post-conditions of transformation functions, is that of *context equivalence*:

**Definition 4.4** *Given two operations, $O_a$ and $O_b$, $O_a$ is* context equivalent *$O_b$, denoted as $O_a \sqcup O_b$, iff $CT_{O_a} = CT_{O_b}$.*

It is important to remember, for the part of this paper where we discuss the merging algorithm, that an essential precondition for analyzing whether two operations are conflicting or not is that they have to be context equivalent. This analysis of the two operations would be done by a conflict-detection function. If this precondition is not ensured, then the conflict-detection function will most likely return an erroneous result, since it would have no way of knowing that the two operations actually have different contexts. To illustrate how this could happen, consider the following example. For its purposes, let us just assume that our conflict-detection function returns *true* (i.e. that the two operations received as parameters are conflicting) if both operations represent changes of the same word.

Assume the initial state of the document is:

| We dance and the music dies.<br>We run through the stars.<br>We are without excuse. |
|---|

One user successively generates the following two operations:

| $O_{11}$ = Insert[3, 'all ']<br>$O_{12}$ = Insert[12, 'd'] |
|---|

in order to change the document to:

| We all danced and the music dies.<br>We run through the stars.<br>We are without excuse. |
|---|

while the other user simply deletes the letters 'n' and 'd' from the word 'and' by performing:

| $O_{21}$ = Delete[10, 'nd'] |
|---|

In this situation, the conflict-detection function, when analyzing $O_{12}$ and $O_{21}$ in order to detect whether they are conflicting or not, will reason that positions 12 and 10 (the positions of the two operations) refer to the same word (namely the word 'and'). Since both operations refer to the same word they are in conflict by definition so the function will return true. As you might have noticed, this answer would be wrong because, in fact, the two operations did not refer to the same word ($O_{12}$ referred to the word 'dance' and $O_{21}$ referred

to the word 'and'). The erroneous judgment is due to the fact that it does not hold that $O_{12} \sqcup O_{21}$, which is exactly the point we were trying to make.

Generally speaking, if we have two lists of operations $L_1 = [O_1^1, \ldots, O_m^1]$ and $L_2 = [O_1^2, \ldots, O_n^2]$ performed on the same base version, it is true that $O_1^1 \sqcup O_1^2$, but it is not true that $O_i^1 \sqcup O_j^2$ $(1 < i \leq m, 1 \leq j \leq n)$. Since this is the case we will usually have to deal with when doing the merging phase of the algorithm, special care will have to be taken that we always transform operations in the two lists in such a way that whenever we compare two of them it will hold that they are context equivalent, thus avoiding problems as the one illustrated above.

The last concept which will help us define the inclusion and exclusion transformations is that of *context preceding relation*.

**Definition 4.5** *Given two operations $O_a$ and $O_b$, $O_a$ is* context preceding $O_b$, *denoted as $O_a \mapsto O_b$, iff $CT_{O_b} = CT_{O_a} \circ [O_a]$.*

Given a log of operations $L = [O_1, \ldots . O_n]$, since all these operations have been executed sequentially by the same user (and thus each of them including the effects of all the ones preceding it), it is necessarily true that $O_{i-1} \mapsto O_i$ $(1 < i \leq n)$. Suppose that an operation $O_k$ cannot be re-executed and has to be excluded from the log. We cannot simply take it out and execute all the operations following it as if nothing had changed. If we think of $O_{k+1}$, for example, it was defined on the document state after the execution of $O_k$ and therefore cannot be executed as is on the document state before the execution $O_k$. In order to omit $O_k$, operations $O_{k+1}, \ldots, O_n$ must be transformed to achieve $O_{k-1} \mapsto O_{k+1}$ and $O_j \mapsto O_{j+1}$ $(k+1 \leq j < n)$ and only then can they be correctly executed.

### 4.3.4 Inclusion and exclusion transformations

We have presented several problems throughout this chapter that all basically reduce to the same issue, namely the necessity of employing a way to transform operations so that they can be executed in a different context than the one in which they were generated and achieve the same effect as originally intended by the user who performed them. The exact same transformations have to be used in order to ensure the aforementioned necessary precondition for correctly reasoning about possibly conflicting operations.

The two types of transformations we need to introduce are the *inclusion* and the *exclusion* transformations. These are defined in [Sun et al. 1996] as follows:

**Definition 4.6** *To include operation $O_b$ into the context of $O_a$, the* inclusion transformation *function $IT(O_a, O_b)$ is called to produce $O'_a$, as specified below:*

*Specification. $IT(O_a, O_b) : O'_a$*

1. *Precondition for input parameters: $O_a \sqcup O_b$.*
2. *Postcondition for output: $O_b \mapsto O'_a$, where $O'_a$'s execution effect in the context of $CT_{O'_a}$ is the same as $O_a$'s execution effect in the context of $CT_{O_a}$.*

**Definition 4.7** *To exclude operation $O_b$ from the context of $O_a$, the* exclusion transformation *function $ET(O_a, O_b)$ is called to produce $O'_a$, as specified below:*

*Specification.* $ET(O_a, O_b) : O'_a$

1. *Precondition for input parameters:* $O_b \mapsto O_a$.
2. *Postcondition for output:* $O_b \sqcup O'_a$, *where $O'_a$'s execution effect in the context of $CT_{O'_a}$ is the same as $O_a$'s execution effect in the context of $CT_{O_a}$.*

In a simpler form, the inclusion transformation function transforms operation $O_a$ against $O_b$ in such a way that the impact of $O_b$ is effectively included in the parameters of the output operation $O'_a$ if $O_a \sqcup O_b$, while the exclusion transformation function transforms $O_a$ against $O_b$ in such a way that the impact of $O_b$ is effectively excluded from the parameters of the output operation $O'_a$ if $O_a \mapsto O_b$.

The form in which we actually encode the inclusion and exclusion transformations in our algorithm will be presented in chapter 5. In the same chapter we will also show how these transformations can be combined in order to achieve more complex effects which our algorithm requires.

# Chapter 5

# Asynchronous Editing Algorithm

The way the concepts presented in the previous chapters were put to work together in order to develop our novel asynchronous text editing system is described in this chapter. We start by presenting the structure of the text document we work with and mention how it enables us to achieve better performance, both in terms of algorithm efficiency and bandwidth consumption. Then we carry on by discussing the operation types that exist in our system and also present our implementation of basic and advanced operation transformations. The most important part of this chapter focuses on the description of the three stages of the copy-modify-merge asynchronous system (the commit, the update and the checkout stage). In the end we cover some additional facilities our system offers and several performance considerations.

## 5.1 Tree representation of the text document

The vast majority of collaborative text editing systems that exist nowadays start with the assumption that a text document is represented as a sequence of characters and simply take it from there. None of the research work we have studied (concerning asynchronous collaborative text editing systems) even mentioned the possibility of using any representation other than the classical one, let alone think about the advantages that could be drawn from an alternative representation. Using a linear view of the document usually implies that the only operations the system ever works with are insertion and deletion of characters. Even though there are variants of systems which allow the insertion and deletion of more than one character with a single insert/delete operation, still the essence is that the document is viewed as a series of characters with no additional structure. This brings several serious disadvantages to the system as a whole, disadvantages which we shall try to outline below.

All algorithms (regardless of how they work) relying on operation transformation keep a log of already executed operations in order to compute the proper execution form of new operations. In systems using a linear representation of the document all past operations are stored into a single common buffer (the per document buffer). When a new set of operations (performed by some other

user) from the repository has to be merged into the local document, all operations from the set have to be compared to every single operation from the local log, regardless of the fact that they might be referring to completely different logical sections of the document. There is simply no way to avoid this when linear representations are used. A great loss of efficiency arises from this since the numbers used to compute the running time of the algorithms tend to be much higher (i.e. the $n$ that appears in the efficiency formula of the algorithm, the $O(f(n))$ would be significantly higher - directly proportional to the length of the editing session - when all operations have to be compared one to another as opposed to when massive operation separation can be employed; it is enough to think of $O(n^2)$ algorithms in order to realize the improvements in running time that can be obtained if operations can be separated in very many smaller groups and compared separately). This is evermore worse when we realize that, statistically speaking, most of the time users working together on a document work on totally different parts of it. It is very seldom that two users actually work on the same section of a document. This means that not only would we have a theoretical efficiency improvement, but we would have an improvement which would appear very often in practice.

Aside from the running time of algorithms, we can also sketch another disadvantage of linear representation, namely the increased bandwidth usage. In order to understand this, think of the two following ways of expressing the creation of the sentence *"We dance and the music dies"*. The way a linear system does this is by generating a separate operation for the insertion of each character, something like [Insert(0,'W'), Insert(1,'e'), Insert(3,'␣'), Insert(4,'d'), ...]. On the other hand, a system using a more sophisticated way of representing documents could simply achieve this with one operation only: [InsertSentence(0,'We dance and the music dies')]. Even though the characters themselves occupy the same amount of space, the overhead (position parameter, for example) of the second way of representation is infinitely smaller than that of the first way. And this is just one sentence we are talking about. Extrapolate this to the case of a full-fledged document and you will easily understand why the use of bandwidth capacity is way more efficient when using a tree representation.

The final issue we would like to mention is that of granularity level. When working with linear representations, the user is constrained to one single level of granularity, namely that of character granularity. The impact of this is that, even though syntactic consistency is ensured, there is simply nothing that can be done in what semantic consistency is concerned. To understand this, think of two users working with the sentence "We danc and the music dies" (notice the missing 'e' from the word 'dance'). Say the first user adds an 'e' at the end of 'danc' in order to obtain "We dance and the music dies". The second user, noticing the problem as well, decides to solve it by changing 'danc' to 'wait', thus transforming the sentence to "We wait and the music dies". A traditional system based on linear representation of documents will at best be able to merge the two changes in order to obtain "We waite and the music dies", which is obviously not what any of the users wanted. This is because there is no way for the users to specify what the basic semantic unit they want to be working with should be. If, for instance, it would be possible to say that changes made to the same word are *always* conflicting (i.e. work at the word granularity level instead of the character granularity level) the problem above would be solved. However, the exact same situation could happen when using word granularity

level if two users insert two different words in a sentence in order to correct it (each of which, taken separately, rendering the sentence correct), but the system puts both words in the sentence, generating a nonsense sentence all over again (since the two words put together make no sense). Why not work at the sentence granularity level then (i.e. two operations making changes to the same sentence are *always* conflicting)? The idea is that different granularity levels are appropriate for different projects and it should be the user who decides what semantic unit (s)he should be using. However, linear representation systems do and can not offer this option to the user because they are unaware of document structure.

Given all these disadvantages, it is clear that there are lot of immediate improvements one can think about. And this is exactly what we have done. We thought of improvements. In order to address the issues mentioned above, the first (and most important) step was to design a way of representing the text document structure which would then allow the use of improved algorithms. The remaining of this section will describe this structure.

The model we propose (based on the one developed in our group and described in [Ignat 2002] and [Ignat 2003]) is a hierarchical one which views the document as a collection of paragraphs, the paragraph as a collection of sentences, the sentence as a collection of words and the words as a collection of characters. Each of this represents a different level of granularity the user can work at. These levels correspond to common semantic elements used in the natural language. Even though our current implementation works with a fixed height tree, the employed algorithms were designed and written in such a way that the actual significance of the level is not taken into consideration, such that they can be applied in their current form on virtually any other kind of tree representation of a document (regardless of the height of the tree). For instance, in the case of writing a book, the hierarchical structure of the document would consist of the following units of granularity: book, chapter, section, paragraph, etc. However, in what follows we are going to focus only on the text documents corresponding to the levels of abstraction presented above. We assigned each of the abstraction units a different numeric level:

- document level = 0
- paragraph level = 1
- sentence level = 2
- word level = 3
- character level = 4

*Note: given that the numbers increase as we go down in the tree, let us mention, for the sake of clarity, that whenever we say lower/higher level of the tree, we see the document level as the highest level and the character level as the lowest level.*

The best way of rigorously defining what our document tree means is to start with the formal definition of the generic node of our tree:

**Definition 5.1** *A* node *is a structure of the form* $N = \{level, length, content, children, log\}$*, where*

- level *represents the level in the tree of the node; level* $\in \{0, 1, 2, 3, 4\}$.
- length *represents the length of the node,*

$$length = \begin{cases} 1 & , \text{ if } level = 4; \\ \sum_{i=1}^{n} length(child_i) & , \text{ otherwise.} \end{cases}$$

$$f(x) = \begin{cases} x - 1, & x < 10 \\ x + 1, & x \geq 10 \end{cases}$$

- content *represents the content of the node, defined only for leaf nodes*

$$content = \begin{cases} undefined & , \text{ if } level < 4; \\ a\ character & , \text{ if } level = 4; \end{cases}$$

- children *represents an ordered list of the nodes that represent the children of the current node* $\{child_1, \ldots, child_n\}$.
- log *represents an ordered list of already executed operations that all refer to the children of this node* $\{operation_1, \ldots, operation_m\}$.

The *level* of each node has a value between 0 and 4, meaning that each node represents one of the following: a document, a paragraph, a sentence, a word or a character. The *length* of a node actually represents the number of leaves in the subtree having the node as its root (and, since each leaf represents a single character, it also represents the number of characters in the semantic unit encoded by the node). The *content* of the nodes, even though unspecified for all non-leaf nodes, is actually generated recursively by concatenating the *content* of its children. Finally, the *log* associated with the node keeps track of all insertions and deletions of children of that node so that (by combining the logs of all the nodes in the tree) the entire sequence of operations that the user generated can be recreated.

In this context, we can define a text document as follows:

**Definition 5.2** *A* document *is a node having level* $= 0$.

Figure 5.1 illustrates how the following document (formed of two paragraphs) is represented in our structure:

> We dance and the music dies. We run through the stars.
> We are without excuse.

*Note: You might have noticed a difference from the model we have described previously, namely that there are no nodes to represent characters. This is because in our implementation we tried to save a bit of memory and not store each character as a separate object. We wanted to do this because the character node itself does not store any additional information other than the character itself (what we really mean is that there is no log associated with a virtual character node because there are no operations which can be done at the character level, i.e. the operations with the finest granularity in our model are the insertion and deletion of single characters, and these operations pertain to the word level). Therefore, by simply making a few adjustments we were able to save quite a bit of memory without any loss in generality. The fields whose interpretation*

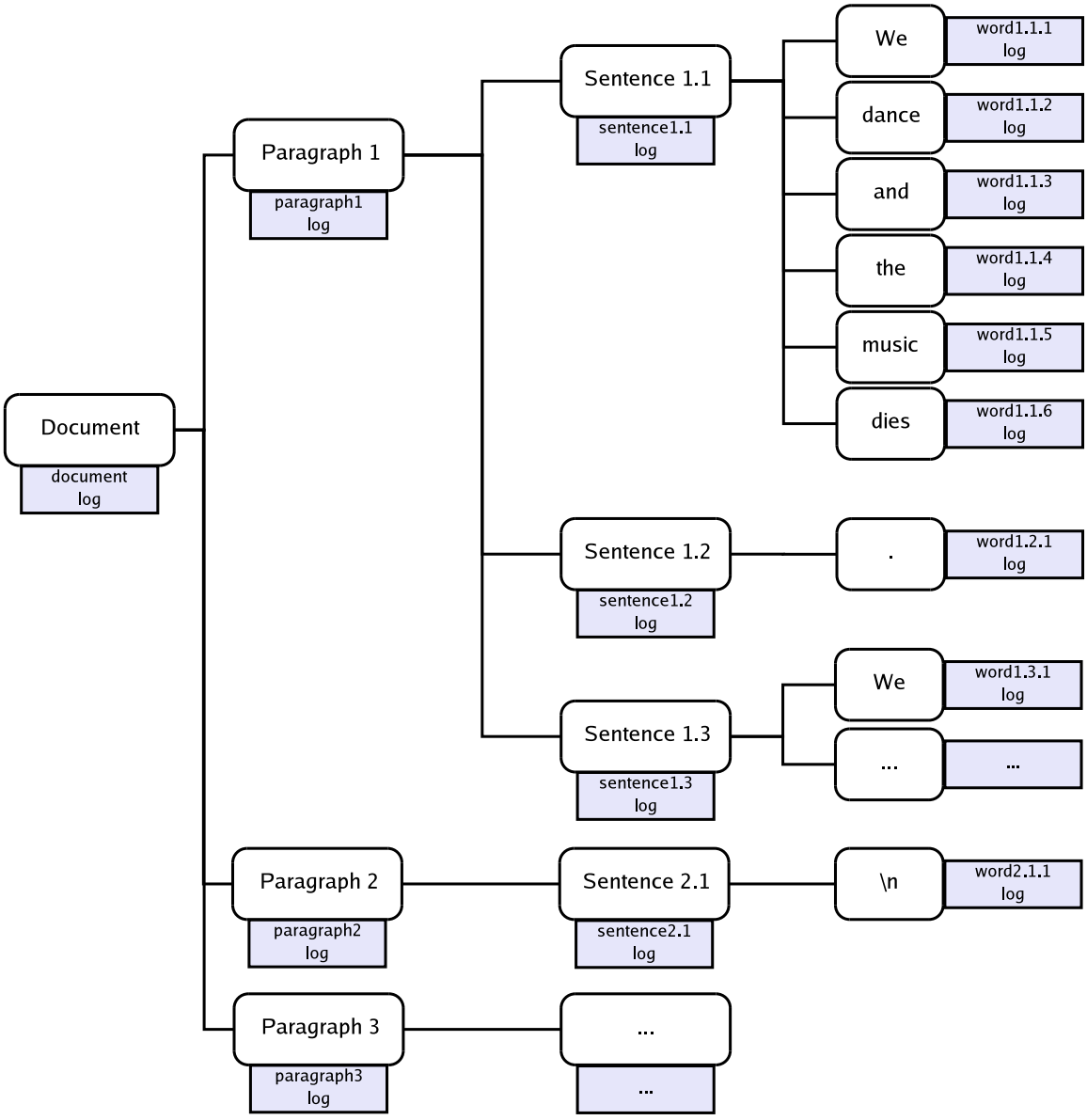


Figure 5.1: Example of a document representation

*we had to change in order to achieve this were the* content *and* length. *Since level 4 nodes no longer exist, the* content *of level 3 - word - nodes was defined to contain a string which represents all the characters in the node. The* length *then also had to be adapted, so that the length of level 3 nodes is now the length of the content string.*

Aside from that, figure 5.1 shows the hierarchical organization of the document and highlights the log associated with each node. You can see that the root of the tree represents the document itself. The document's children are the three paragraphs, each paragraph has the sentence that it is formed of as children and so on. As you can see in the figure as well, the punctuation marks are represented as nodes of their own. Each punctuation mark will be represented as a separate paragraph, sentence or word (depending on the type of separator it is). This means that paragraph separators will be represented as separate paragraphs, sentence separators (such as the '.' you can see in the figure) will be represented as separate sentences and word separators will be represented as separate words. In the case of paragraph and sentence separators, the entire structure down to the word will be created (if you take the '.' in the figure, for example, you can see that it forms a sentence on its own containing just one word; that word represents the punctuation mark itself; another example of this is the paragraph separator, '\n', from the figure - notice how the entire structure down to the word is created).

This is the tree structure we shall be using to represent the text documents used in our project.

## 5.2 Operations and operation transformation

### 5.2.1 Operation definition

Once the tree structure was defined, the next step was to devise a way in which operations are represented in the system. Our model only uses two types of operations: insertion and deletion. However, as we wanted to overcome some of the disadvantages mentioned in the previous section, the operations allowed in our system are not only the classic insertion and deletion of characters, but an extended set of these, consisting of the following operations:

- InsertChar and DeleteChar
- InsertWord and DeleteWord
- InsertSentence and DeleteSentence
- InsertParagraph and DeleteParagraph

Besides these, a special type of operation was also added in order to specify a null effect:

- NOP

Formally speaking, all the nine operations listed above can be defined in the following manner:

**Definition 5.3** *An* operation *is a structure of the form* $op = \{type, level, index, content\}$*, where:*

- type *represents the type of the operation,* $type \in \{insert, delete, nop\}$.
- level *represents the level in the tree to which the operation pertains,* $level \in \{0, 1, 2, 3\}$.
- index *represents an array of indices, with* $index[i]$ *= the index for the* $i^{th}$ *level of the tree,* $i \in \{0, \ldots, level\}$.
- content *represents the text content of the operation, i.e. the text inserted or deleted by the operation.*

The *type* property defines whether the operation at hand is an insert, a delete or a NOP. If the operation is a NOP none of the other fields are of importance. The *level* tells us if the operations is at the document (0) level (i.e. an InsertParagraph of DeleteParagraph), at the paragraph (1) level (i.e. an InsertSentence or DeleteSentence),. at the sentence (2) level (i.e. an InsertWord or DeleteWord) or at the word (3) level (i.e. an InsertChar or DeleteChar). The *index* gives us the position in the document where we have to insert that particular unit of text. For instance, in the case of a InsertChar, index will contain 4 entries with the following significance: index[0] gives us the paragraph in which we want to insert the character, index[1] gives us the sentence within the paragraph in which we want to insert the character, index[2] the word within the sentence and index[3] the position in the word at which to insert the character. Please notice that index contains exactly *level* entries for each operations, where *level* is the actual level parameter of the operation. Finally, the *content* holds the inserted/deleted text (a character for InsertChar/DeleteChar, a word for InsertWord/DeleteWord and so on).

For simplicity, in the explanations that follow, we shall denote operations by specifying the type and the level in the operation name (i.e. InsertWord, DeleteParagraph, etc.), the index as separate parameters (whose number depends on the level of the operation) and the content. An example would be **DeleteWord(2, 1, 5, "hello")**. In this case, the operation type is insert, the operation level is 2 (sentence), the index is the array [2, 1, 5] and the content is "hello". In order to avoid misunderstandings, please notice that a DeleteWord operation is an operation at the sentence level (i.e. level 2), not at the word level. This is because we insert and delete words into a particular sentence and, therefore, the operations will appear in the log at the sentence level. This is why we say they pertain to the sentence level. The same holds for all other types of operations as well (InsertChar and DeleteChar pertain to the word level and so on).

### 5.2.2 Applying operations

The first thing that needs to be explained about operations is the way they are applied on the document structure in order to update it. The applying of an operation is necessary in very different circumstances, ranging from executing local operations to integrating remote ones. As it is very important to understand how the document structure changes as an effect of applying operation, we shall try to explain in this section how this is achieved in our project.

Listing 5.1 reproduces a stripped down version of the code that implements this. We shall try to walk through it line by line and explain the actions that are performed. The first thing to notice is the parameters of the method. These are the operation that has to be executed and a tree node. The method is always called from the outside using the root of the document tree as the node. We say "from the outside" because, as the method is recursive, when it calls itself recursively, values other than the root of the document tree will be used.

The first thing to do is to check whether we are dealing with a NOP. If so, the method simply returns as no changes have to be made to the document tree. If this is not the case, the variable *childNo* is assigned with a value depending on the level in the tree where the *node* parameter is. The idea is that we want *childNo* to hold the number of the child of the current node that the operation index indicates. This is used to recursively go down the tree until the "work node" is reached. By "work node" we refer to the node where the changes actually have to take place. If the operation is an InsertWord, for example, the changes have to take place when we reach the sentence level in the tree (i.e. reach the node that represents the sentence in which the word has to be inserted). The assignment is done in the lines 6-13. Next, we check to see whether we have already reached the level in the tree where we have to make the changes. If we have not, a recursive call to applyOperation is made in order to go one level down the tree (this is achieved by calling the method using the appropriate child of the current *node* as the second parameter).

Lines 19 through 35 show how an insert operation is executed. In a nutshell, if we are dealing with an InsertChar (i.e. the level of the operation is the word level) all we have to do is modify the content of the word node in order for it to contain the inserted character in the position specified by *index[3]*. If the operation pertains to a level higher than the word level, we resort to one of the *parseWord()*, *parseSentence()* or *parseParagraph()* methods, which will build the entire subtree that correctly models the content to insert and return its root. This root we add as a new child of the current *node* in the position indicated by *childNo* (if you think about it, this is actually analogous to inserting a single character at the appropriate position in the word for operations at the word level).

Finally, lines 39 through 48 illustrate how a delete operation is handled. A delete at the word level is executed by simply removing the indicated character from the appropriate word (in a similar manner to the way an insert operation adds it), while a delete at a higher level in the tree simply removes the child indexed by *childNo* from the current *node*'s list of children (line 47 in the code). Lines 44 and 45 are necessary for local operations only in order for the content of the locally generated operation to be correctly stored (even thought the inversion is executed for remote operations as well, its effect is irrelevant in this case because the remote operations, once executed locally, are discarded; even if we change the content of the operation in line 45, since the operation is discarded anyway, this has no further implications; moreover, since the operation is a delete, any changes that are generated by the inverse method are lost anyway when the whole subtree is deleted in 47). The *inverse()* method called in line 44 executes the inverse of all operations (insert instead of delete and vice-versa) from the subtree it receives as parameter in the reverse order than that in which they have been stored in the log. The purpose of doing this is to ensure that whatever changes might have been done locally to the subtree to be deleted

```
applyOperation(Operation op, DocumentTreeNode node)
{
    if (op.getType() == OperationType.nop)
        return;

    if (node.getLevel() == TreeLevel.document)
        childNo = op.index[0];
    else if (node.getLevel() == TreeLevel.paragraph)
        childNo = op.index[1];
    else if (node.getLevel() == TreeLevel.sentence)
        childNo = op.index[2];
    else if (node.getLevel() == TreeLevel.word)
        childNo = op.index[3];

    if (op.getLevel() != node.getLevel())
        applyOperation(op, node.getChildAt(childNo));
    else
    {
        if (op.getType() == OperationType.insert)
        {
            if (op.getLevel() == TreeLevel.word)
            {
                node.setContent(node.getContent().substring(0, childNo) +
                    op.getContent() +
                    node.getContent().substring(childNo));
            }
            else if (op.getLevel() == TreeLevel.sentence)
                node.addChildAt(childNo,
                    Parser.parseWord(op.getContent()));
            else if (op.getLevel() == TreeLevel.paragraph)
                node.addChildAt(childNo,
                    Parser.parseSentence(op.getContent()));
            else if (op.getLevel() == TreeLevel.document)
                node.addChildAt(childNo,
                    Parser.parseParagraph(op.getContent()));
        }
        else if (op.getType() == OperationType.delete)
        {
            if (op.getLevel() == TreeLevel.word)
                node.setContent(node.getContent().substring(0, childno) +
                    node.getContent().substring(childNo + 1));
            else
            {
                inverse(node.getChildAt(childNo));
                op.setContent(node.getChildAt(childNo).getContent());

                node.removeChild(childNo);
            }
        }
    }
}
```

Listing 5.1: apply operation

are reversed. The reason for this is that these changes will be lost once the subtree is deleted in line 47, but the content stored in the delete operation would still reflect them, were we not to call the *inverse()* operation. This would lead to inconsistencies once the operation would be sent to other sites. Once the *inverse()* method is called, however, we can safely store the content of the operation in line 45.

Now that the way operations are applied is clear, let us advance to the next important operation-related functionality, namely the two basic transformations: the inclusion and the exclusion transformation.

### 5.2.3 Inclusion and exclusion transformation

As you probably recall, the previous chapter introduced and explained the whole concept of inclusion and exclusion transformations in the theoretical context in which they originally appeared. In this section we assume that the reader is already accustomed with the concepts already described in chapter 4 and go on to describing the way we put these concepts into practice in our project.

We have based our development of the transformation functions on the close of analysis of [Imine et al. 2003], which verifies the correctness of state-of-art transformation functions defined on strings using an automated theorem prover, in order to avoid the common mistakes others have made when writing such functions.

*Note: For the remainder of this section we make the following naming convention: from the two operations involved in an inclusion/exclusion transformation we denote the operation that has to be transformed as the* current operation *and the operation against which the current operation has to be transformed as the* against operation.

We start our discussion by mentioning that we had to define the transformations in a way that takes into account the fact that the operations are no longer applied on a linear structure, but on a hierarchical one. Several implications follow from here, such as the fact that the inclusion/exclusion of an against operation that refers to lower level of the tree than that the current operation refers to has no effect whatsoever. This is also the case with two operations on the same level, but referring to units which are not the children of the same parent. However, an against operation at a higher level in the tree than the current operation might need to change the appropriate index value in the current operation (when the two operation refer to the same subtree).

This approach is rather different than that in use by linear systems, where any against operation that refers to a position before that of the current operation is sure to change the index of the current operation. Herein lies one of the advantages of using a tree-based representation: significantly fewer operation will have to be changed.

The inclusion and exclusion transformations we explain in this section represent the building blocks of higher level functions the algorithm uses, therefore their understanding is essential for a good comprehension of the algorithm later on.

Both the inclusion and the exclusion methods are part of the Operation class, therefore all the methods that are not used within are themselves members of the Operation class.

### The *include* method

We shall start with the *include()* method which is displayed in listing 5.2.

Let us walk through the code one step at a time and see what happens. The first thing to do is clone the current operation in order to obtain a version of it (initially identical) which we can then modify. A check is then made in order to find out whether or not either of the two operations in question (the current or the against operation) is a NOP. If this is the case, we can simply return since an operation does not change by including a NOP and a NOP does not change by including an operation, whatever it is.

Once this is out of the way, the next thing to do is obtain the level at which each of the two operations refer and store it in *levelCurrent* and *levelAgainst*. The level an operation refers to is given by its type. An InsertParagraph, for instance, acts at level 0 in the tree, while a DeleteChar acts at level 3. Next, a few checks are made in order to see if the against operation should have any effect on current operation. First, if its level is lower than that of the current operation, we can safely return from the method without making any changes to the current operation since, for example, an InsertWord cannot influence a DeleteSentence in any way, even if it is in that exact sentence. This is because the only index an InsertWord would influence would be the word index, but a DeleteSentence does not have a word index in the first place, so there is nothing to influence. Then, we want to see whether the two operations refer to the same subtree (*Note: two operations refer to the same subtree if the nodes they work on to are either identical (i.e. they refer to the same node) or if one is the ancestor of the other*). This is achieved in lines 17-19. The idea here is that if the against operation is an InsertSentence and the current operation is a DeleteWord that InsertSentence could only influence the sentence index of the DeleteWord *if* the word is deleted from a sentence that is in the same paragraph as the one to which the InsertSentence is referring.

At this point of execution we know that the against operation might actually have an effect on the current operation. If the against operation is a *delete* there are two cases in which the current operation needs to be changed.

The first case is actually a combination of conditions, the first of which requires that the lowest available index of the against operation (i.e. the index which corresponds to the actual level of the tree to which the against operation refers) is equal to the corresponding index in the current operation. These means that both operations work on the same node or that the current operation works on a node that is the subtree the root of which is the node the against operation works on. Further on, we need to consider the two cases separately.

- when the current operation is of a lower level than the against operation, no additional condition has to be met. This translates to the fact that the against operation is actually deleting the entire subtree in which the node the current operation works on is located. Obviously, the object of the current operation ceases its existence once the against operation is executed, so the current operation no longer has a purpose, therefore it needs to be canceled.

- when the two operations refer to the same level, the current operation also has to be a delete. This translates to the fact that both operations are effectively deleting the exact same semantic unit (may it be paragraph,

```
1  Operation include(Operation opAgainst)
2  {
3      Operation result = clone();
4
5      if (opAgainst.getType() == OperationType.nop ||
6          getType() == OperationType.nop)
7          return result;
8
9      // the level of the current operation
10     int levelCurrent = getLevel();
11     // the level of the against operation
12     int levelAgainst = opAgainst.getLevel();
13
14     if (levelAgainst > levelCurrent)
15         return result;
16
17     for (int i = 0; i < levelAgainst; i++)
18         if (index[i] != opAgainst.index[i])
19             return result;
20
21     if (opAgainst.getType() == OperationType.delete)
22     {
23         if (index[levelAgainst] == opAgainst.index[levelAgainst])
24             if (levelCurrent != levelAgainst ||
25                 levelCurrent == levelAgainst &&
26                 getType() == OperationType.delete)
27                 result.setType(OperationType.nop);
28
29         if (index[levelAgainst] > opAgainst.index[levelAgainst])
30             result.index[levelAgainst]--;
31     }
32     else if (opAgainst.getType() == OperationType.insert)
33     {
34         if (index[levelAgainst] == opAgainst.index[levelAgainst] &&
35             getType() == OperationType.insert &&
36             levelCurrent == levelAgainst)
37
38         {
39             if (current and against ops generated at different sites)
40             {
41                 if (getContent() > opAgainst.getContent())
42                     result.index[levelAgainst]++;
43             }
44
45             if (current and against ops are the same operation)
46                 result.setType(OperationType.nop);
47         }
48         else if (index[levelAgainst] >= opAgainst.index[levelAgainst])
49             result.index[levelAgainst]++;
50     }
51
52     return result;
53 }
```

Listing 5.2: include operation

a sentence, a word or a character). It is clear that in order to preserve the intention of both users the deletion only has to be performed once. The solution is again to cancel the current operation, since the against operation already did its job.

So, in both of these two cases, the current operation needs to be changed to a NOP, which is exactly what line 27 does.

The second case to consider is more straightforward and it represents the typical inclusion case, and it is analogous to that used in linear systems. If the index of the current operation corresponding to the level of the against operation is greater than the one of the against operation, then it is clear that it has to be decreased by one since the against operation deleted a node. Therefore, the node the current operation index was originally referring to is now one position more to the left. This case is covered in lines 29-30.

Now let us see what happens when the against operation is an *insert* operation. Again, we have two cases to deal with.

The first case is when the lowest available index of the against operation (that corresponding to the actual level of the tree to which the against operations refers) is equal to the corresponding index in the current operation *and* both operations refer to the same level of the tree *and* the current operation is also an insert. As before, this means that both operations insert a semantic unit (a character, a word, etc.) at the same index of the parent node. The discussion here is a bit more difficult. On one hand we have the case when the two operations were generated by different sites (see line 39). In this case it is irrelevant what order the two semantic units are inserted in (since no priority exists in what the two users who generated the two operations are concerned, any order of insertion is as good as the other), but it is essential that they are inserted in the same order on all sites. In order to achieve this, a simple trick was employed, namely to alphabetically compare the two contents to be inserted and always insert the content which is alphabetically "lower" before the content that is alphabetically "higher". This means that if the against operation inserts the "lower" content, then the current operation has to have its index increased by 1 so that the insertion it makes takes place in the position immediately after the one on which the content of the against operation was inserted. It would have been equally correct to increase the index had the opposite case been true (i.e. if the against operation inserted the "higher" content). On the other hand, we have the case when the two operations were generated by the same site. In this case no reordering has to be made since the two operations were generated by the same user and the original order has to be kept.

An additional check is then made to see if the two operations have the same unique identifier (line 45). If it is true, then the current operation is canceled (i.e. turned to NOP). This case can appear when direct user synchronization is used. We shall return to it when we talk about this facility of our application.

Again, the second case is the "classical" one, analogous to that used in linear systems. If the index of the current operation corresponding to the level of the against operation is greater than or equal to that of the against operation, it has to be increased by one since the against operation inserted a new node. Therefore, the node the current operation index was originally referring to is now one position more to the right. This case is covered in lines 48-49.

At the end, the transformed version of the current operation is returned.

### The *exclude* method

Now, we go on to explaining the way in which the exclusion of an operation is achieved in our algorithm. You can find the method illustrated in listing 5.3.

The *exclude* method is slightly more straightforward than the include one. The first lines are roughly identical. Here, as in the case of the include method, excluding NOP from an operation or excluding an operation from NOP has no effect whatsoever, so we can simply return. Next, we also need to retrieve the level at which each of the two operations works and store it into two variables which we have named the same for convenience.

Once again, if the against operation refers to a lower level than the current operation or if the two operations refer to different subtrees (i.e. the nodes they refer to are neither the same nor is any of them the ancestor of the other), there is nothing that we have to do and can simply return. For a more elaborate discussion about this, please refer to the previous subsection where we explain the include method.

Next, let us see what happens when the against operation is an insert. As with include, we have two cases. The first one arises when the indices of the two operations that indicate the position at the level at which the against operation has effect coincide. This means that they either refer to the same node of the document tree or that the current operation refers to a node that is in the subtree the root of which is the node the against operation acts on. If the former is the case and the current operation is itself a delete operation, it means that the current operation deletes the exact same node the against operation inserts. Excluding the effect of the against operation is just another way of saying that we need to see what happens with the current operation when the against operation is no longer executed. Obviously, if the node is no longer inserted, it cannot be deleted by the current operation, so the operation has to be canceled altogether. In the latter case (of the two cases described above), the current operation wants to insert or delete a node in/from the subtree that has been inserted by the against operation. But if the insertion no longer takes place, the current operation (regardless of whether it is an insert or a delete) does not makes sense anymore, so again it has to be canceled. This is what we do in line 28.

The second case appears when the index of the current operation that gives the positioning in the level the against operation refers to is greater than its counterpart in the against operation. We need to decrease the mentioned current operation index by one since the node it indicates moves one position to the left if the against operation (which, if you remember, was an insert) is no longer executed. For the sake of symmetry, let us mention that this is the "classical" case.

Finally, when the against operation is a delete, we only have to deal with the "classical" case and increase the index of the current operation that indicates the position in the level the against operation refers to by one if the condition in line 36 is met (this is the same condition that we have described over and over in the previous paragraphs, so chances are that you are already familiar with it). The reason we have to add 1 to the index is that if the delete operation is no longer executed, the node the index indicates moves one position to the right.

```
1  public Operation exclude(Operation opAgainst)
2  {
3      Operation result clone();
4
5      if (opAgainst.getType() == OperationType.nop ||
6          getType() == OperationType.nop)
7          return result;
8
9      // the level of the current operation
10     int levelCurrent = getLevel();
11     // the level of the against operation
12     int levelAgainst = opAgainst.getLevel();
13
14     if (levelAgainst > levelCurrent)
15         return result;
16
17     for (int i = 0; i < levelAgainst; i++)
18         if (index[i] != opAgainst.index[i])
19             return result;
20
21     if (opAgainst.getType() == OperationType.insert)
22     {
23         if (index[levelAgainst] == opAgainst.index[levelAgainst])
24         {
25             if (levelCurrent != levelAgainst ||
26                 levelCurrent == levelAgainst &&
27                 getType() == OperationType.delete)
28                 result.setType(OperationType.nop);
29         }
30
31         if (index[levelAgainst] > opAgainst.index[levelAgainst])
32             result.index[levelAgainst]--;
33     }
34     else if (opAgainst.getType() == OperationType.delete)
35     {
36         if (index[levelAgainst] >= opAgainst.index[levelAgainst])
37             result.index[levelAgainst]++;
38     }
39
40     return result;
41 }
```

Listing 5.3: exclude operation

As in the case of the include method, we terminate the method by returning the modified version of the current operation.

### 5.2.4 Transpose and symmetric inclusion transformation

The final step before introducing the merging algorithm itself is to define two additional functions which "wrap" the inclusion and exclusion transformation in order to obtain the form in which the algorithm actually uses them. These two functions are *transpose* and *symmetric inclusion transformation*.

The *transpose* operation is defined to transpose $O_a$ and $O_b$ where $O_a \mapsto O_b$ in order to achieve $O_b \mapsto O_a$ (see definition 4.5 if you have forgotten what "$\mapsto$" stands for). Therefore, if $O_{k-1} \mapsto O_k \mapsto O_{k+1}$, after *transpose*$(O_k, O_{k+1})$, it becomes $O_{k-1} \mapsto O_{k+1} \mapsto O_k$. As a result, $O_{k+1}$ can be executed right after $O_{k-1}$. This is exactly what we shall use transpose for: take operation $O_k$ from the log and move it (transpose it) successively to the right in order for it to become the last operation of the log. But we shall talk about this later.

For now, we need to explain why a simple inversion of $O_a$ and $O_b$ would not suffice. The problem is that if we reverse the order of execution of the two operations there are some changes that we have to make to the operations in order for user intention to be preserved. These changes are rather intuitive. First we have to make sure that operation $O_b$ excludes the effect of operation $O_a$ since if we are moving it before $O_a$ it means that the effect of $O_a$ (which is initially included in $O_b$) has to be canceled. Second, we have to make sure that the effect of $O_b$ is included into $O_a$. This is because the transpose implies that $O_a$ will be executed after $O_b$, so it should become aware of its effect (since it initially was not as it was executed before it). Putting this into code, what we obtain is:

```
Transpose(O_a, O_b)
{
    O := ET(O_b, O_a);
    O_b := IT(O_a, O);
    O_a := O;
}
```

where by $IT$ we denote the *inclusion transformation* and by $ET$ we denote the *exclusion transformation*. The actual implementation is slightly more complicated because several special cases have to be taken into consideration.

One of them is the case when $O_a$ is an insert on a specific position and $O_b$ is a delete from the same position. If you think of how the include and exclude operations are implemented, you realize that if we simply execute the transpose in the above form we would first exclude the effect of $O_a$ (the insert) from $O_b$ (the delete). This would yield a NOP since otherwise $O_b$ would end up trying to delete something that doesn't exist yet. Transforming $O_b$ into a NOP would be perfectly fine. The problem comes at the next step when we include the effect of the NOP in $O_a$. Including $O_a$ against a NOP yields an unmodified $O_a$ as a result. This is obviously wrong since the effect of applying the modified $O_b$ and $O_a$ (in this order) after the transpose should be the same as applying the original $O_a$ and $O_b$ (in this order). Clearly, this does not hold if $O_b$ becomes NOP and $O_a$ remains unchanged. In order to find a solution for this particular case we had to handle it separately and transform both $O_a$ and $O_b$ to NOP.

Another special case appears when $O_a$ and $O$ (the version of $O_b$ which no longer contains the effect of $O_a$) are both Insert's in the same position. For example, consider the case when $O_a = InsertChar(3, 1, 2, 0, \text{'a'})$ and $O_b = InsertChar(3, 1, 2, 1, \text{'b'})$. The final string that would be obtained is "ab". After excluding the effect of $O_a$ from $O_b$ (in line 1 of the transpose), the new version of $O_b$ would be $O = InsertChar(3, 1, 2, 0, \text{'b'})$. If now we were to simply perform the inclusion of $O_a$ against $O$ (the second line of the transpose operation), $O_b$ would be assigned $O_b = InsertChar(3, 1, 2, 1, \text{'a'})$. The final result would be wrong because the effect of applying the modified $O_b$ and then the modified $O_a$ (which is what we would do after the call to transpose) would be the obtaining of the string "ba" instead of the initial string, "ab". In order to solve this special case we need to replace line 2 of the transpose function (for this case only) with $O_b := O_a$ (meaning that we are assigning $O_a$ to $O_b$ as is, without the inclusion of the effect of $O$). The result of this in the case we were discussing above is that we would eventually obtain $O_b = InsertChar(3, 1, 2, 0, \text{'a'})$, which means that by executing the modified form of $O_b$ and then of $O_a$ we would obtain the correct string, "ab".

However, once all these special cases are dealt with, the procedure behaves correctly and achieves what we wanted in the first place: given two operations, $O_a$ and $O_b$ such that $O_a \mapsto O_b$ the transpose function changes them so that afterwards $O_b \mapsto O_a$. More than that, if the new $O_b$ and $O_a$ are executed in this order, they will achieve the same effect as the execution of the original $O_a$ and $O_b$ (in this order).

The second operation we were mentioning at the beginning of this section was the *symmetric inclusion transformation*. Let us start with the following example in order to clarify why the symmetric inclusion transformation is needed. Imagine the simple scenario of two users (John and Mary) on two different sites working on the same version of a document. Assume each of them performs just one operation (John performs operation $O_J$ and Mary operation $O_M$) on his/her working copy. Then, Mary commits her operation to the repository. John has to update his local copy of the document before being able to commit himself. Let us analyze what happens during this update. John will receive a single operation from the repository, $O_M$, which has to be integrated into his version. The operation cannot be executed 'as is' because the context John currently has is different from the context in which $O_M$ was generated. Therefore, $O_M$ has to include the effect of $O_J$ for a correct execution. Once this is accomplished, $O_M$ can be safely executed on John's local copy. The question that remains unanswered is: what will John commit to the repository? He cannot simply commit $O_J$ as it is because the version on the repository has changed (meaning that the context on the repository is now different from the one in which $O_J$ was generated). The solution is to make $O_J$ include the effect of the operation on the repository (namely, the effect of $O_M$). Once this is done, the modified version of $O_J$ can be safely committed to the repository since now its context corresponds to that on the repository.

If we take a second and think back about the operations we have done, we shall realize that they were two include operations. First, we included the effect of $O_J$ in $O_M$ and then we included the effect of $O_M$ into $O_J$. As you have probably figured out on your own already, this is a *symmetric inclusion*. Even though the real case will be slightly more complicated than the one from above, still the symmetric inclusion transformation will have to be used in that exact

same way on individual pairs of local and remote operations. The procedure itself, as you might imagine, is very straightforward:

```
SymmetricInclusion(O_a, O_b)
{
    O := IT(O_a, O_b);
    O_b := IT(O_b, O_a);
    O_a := O;
}
```

This being said, we now have all the background laid out and are finally ready to advance to the description of the commit, update and checkout procedures themselves.

## 5.3 The stages of the collaborative editing algorithm

As we explained in section 3.5.2, the main operations of an asynchronous editing system are the commit, update and checkout operations. As ours is precisely such a system, we also had to offer an implementation of the three basic operations. This was actually the most consistent and challenging part of the project. In this section we shall describe the way in which we have implemented each of the above operations.

### 5.3.1 The commit stage

To commit a working copy in the repository, a certain course of actions has to be taken in order to ensure the consistency of the system. We describe this course of actions as a sequence of steps the system executes when the user requests that the document (s)he has changed locally be committed to the repository.

The first action the system performs is send a message to the repository including the version number of the document which is the base version for the editing session (i.e. the latest version of the document that is integrated in the local working copy of this user). This is done in order to find out whether the current version on the repository is identical to the base version of the current user. The repository does this check and sends a positive or negative answer back to the user. Let us analyze what each of these cases means.

If the current version on the repository is identical to the base version of the user, it means that no changes have been committed to the repository that this user is not aware of. Therefore, the user can simply send his log to be stored on the repository as is, without any modifications. If, however, the repository answers that its current version is different from the base version of the user, it means that somebody else has committed some changes that this user is not aware of. Consequently, the user cannot simply send his unmodified local log to the repository. This is because the operations would end up to be executed in a different context than the one in which they were generated, leading to erroneous results.

To better understand this, think of the following concrete case: when the user starts working the version on the repository is $V_k$. This will become his

base version. Suppose he would only execute one operation, $O$. The context of this operation is version $V_k$ of the document. In the meantime, the repository's version changes (by other users committing their own changes) from $V_k$ to $V_{k+n}$. And now assume that $O$ was committed as is on the repository. Later, when another user would check out the document from the repository, (s)he would receive a set of operations which includes $O$, which (s)he would execute in the order in which (s)he receives them. This means that $O$ would end up executed in the context of version $V_{k+n}$ which is different from the one it was generated in ($V_k$). The effect is that the user intention would probably be not preserved.

In light of these explanations, it is hopefully clear why we cannot simply commit an unchanged log to the repository when the base version on the local site and the current version of the repository differ. What happens in this case is that the system will warn the user that his/her version of the document needs to be updated before (s)he can successfully commit and then stop the commit stage. In order to update the document, the update procedure has to be called. What this procedure does is described in the next section. Once the update is complete, the user can try to commit once again by recalling the commit procedure, thus re-initiating the same sequence of actions from above.

Now, let us see what happens when the repository sends back a positive answer and the user can carry on with the commit. The user should send the local log to be stored on the repository. What we need to detail here is what the local log actually is in our case, given the fact that we have a log distributed throughout the tree that represents the document. Figure 5.2 gives you an overall image of how this is done. What you see in this figure is a sample document tree in which the (possibly empty) log of each node is shown.

The figure depicts how the serialized log that is sent to the repository is composed. The first operations will be those in the document log, then those in the paragraph logs (with paragraph 1 log first, paragraph 2 log second and so on), then those in the sentence logs (ordered corresponding to the numbering in the figure) and, finally, those in the word logs. In fewer words, the commit log is composed by doing a breadth-first traversal of the document tree and orderly adding the log associated with each node to the commit log. What we obtain is a log in which all document level operations come first, all paragraph level operations (ordered by paragraph number) come second, all sentence level operations (ordered by sentence number) come third and all word level operations (ordered by word number) come last.

This ordering is not random. The reason for which we have ordered the operations this way will become more obvious when we explain the update stage. A short explanation would be that when, during the update stage, we merge the log from the repository with the local log, we need to progressively update the local tree in a top-down manner, in order for the level $k$ to already be updated when we merge operations on level $k+1$ and below. If it were not so, we would have no way of knowing what *local* $k$-level node operations acting on the $k+1$ and lower levels from the *remote* log refer to. In other words, if a remote operation's $k$-level index is $m$ (while the operation itself refers to level $k+1$ or below), if we do not update the *local* level $k$ of the tree before executing the operation, we do not know which *local* node on level $k$ is actually the $m^{th}$, so we basically do not know which $k$-level logs to merge.

Once this commit log is generated, all that remains to be done is to send it to the repository. Of course, it could be the case that between the initial

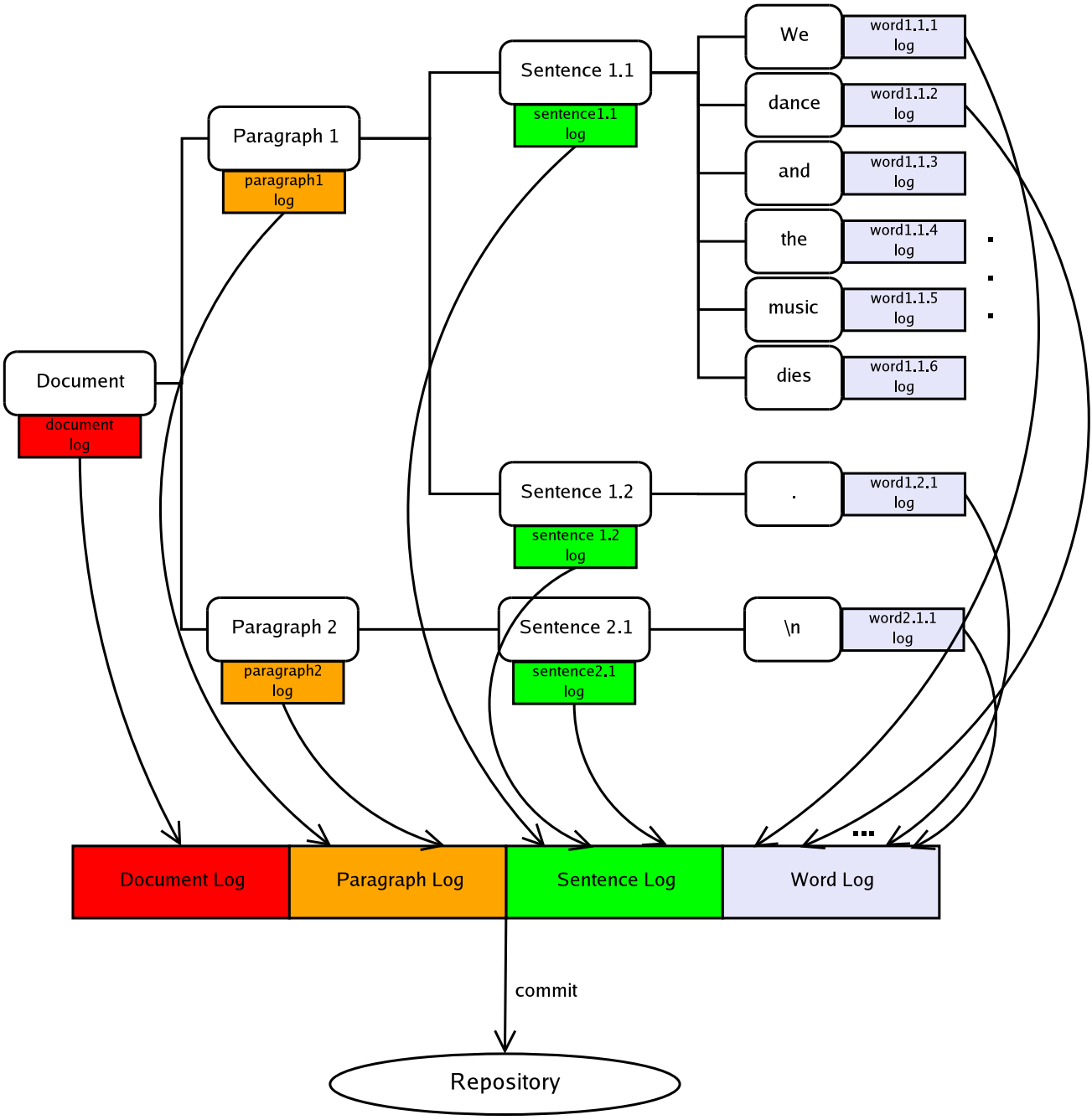


Figure 5.2: The committed log

comparison of the local base version and the current version on the repository, the situation has changed and the two versions might no longer be the same when the commit is actually attempted. In order to ensure that a faulty commit does not take place, the base version is once again sent along with the commit log, thus allowing the repository to atomically redo the comparison and, if successful, to commit the changes. If, however, the version on the repository has changed, a failure message will be sent back to the client telling it that it needs to update its version before committing. The client would then have to do an update and then restart the whole commit procedure. If the version has not changed, then the changes are stored on the repository and the current version number is increased by one.

All the client has to do in the case that it receives a positive response from the repository is to empty the local log (since the operations it contained are now on the repository). This is done by traversing the whole document tree and deleting all the logs associated with the tree nodes. Also, the base version on the client is increased by one (the client is aware of the changes which comprise the new version on the repository since it itself has generated and committed those changes).

### 5.3.2 The update stage

We have finally reached the most complex part of our algorithm, the update stage. In this section we shall try to explain it as clearly as possible since it represents an essential part of our work at this project. A necessary precondition for the thorough understanding of this section is that you have worked your way through this entire chapter as all the concepts and functions described previously are used in the implementation and we shall assume that you are already familiar with them.

We shall break the description of the update method in two distinct parts and explain them separately. These two parts are the basic merge algorithm (which takes two lists of operations, merges them according to a series of rules and returns two transformed lists) and the actual update algorithm which uses the merge algorithm internally in order to update the local document tree with a log taken from the repository.

#### The basic merge algorithm

The basic merge algorithm we describe here is an adaptation of the one presented in [Shen and Sun 2002]. In linear systems, this algorithm can be used 'as is' as the entire update stage since what it does is merge two list of operations: the local log and the log from the repository. In a liner system all we have is these two lists which contain all the operations. However, in our case it will need to be integrated into a larger scale algorithm that traverses the local tree and the remote log of operations in parallel and decides what sets of operations have to be compared and merged.

The reasons for which we need to merge the two lists should be already clear from the previous sections, so we shall only lightly address them here. Basically, the local user has started working from version $V_k$ on the repository. This is the last version which (s)he is aware of (i.e. any operation that appears in versions higher than $V_k$ are unknown to this user). When (s)he finishes editing and

wants to commit the changes to the repository, the commit can be performed only if the context of the local operations is modified so that it no longer is version $V_k$, but version $V_{k+n}$ (the current version on the repository). In order to modify this context of the local operations we need to take the operations that represent the delta between version $V_k$ and version $V_{k+n}$ on the repository (in what follows we shall call these operations the *remote log*) and perform two actions:

- change the form of all the local operations to make them include the effects of the operations in the remote log (in other words, change the context of all local operations from version $V_k$ to version $V_{k+n}$)
- apply the operations from the remote log on the local copy of this user in order to update it to version $V_{k+n}$. The operations, however, cannot be executed in the form in which they are received from the repository because the local context differs from the remote context in which these operations were generated. Consequently, they have to be transformed in order to include the effect of all the local operations before they can be executed

These two actions are achieved by the merge algorithm. Before advancing to the algorithm itself, let us introduce two functions that we shall use. The first of them is called *semantic conflict*. In [Dourish 1996], the conflicts are classified as being either syntactic or semantic. Syntactic conflicts occur at the system infrastructure level, while semantic conflicts are inconsistencies from the perspective of the application domain. In the case of the multi-user text editing, consistency from the users' perspective is often not the same as consistency from that of the system's. Generally, the operational transformation algorithms solve the syntactic inconsistency problems, but they do not enforce semantic consistency. In [Munson 1994] and [Shen and Sun 2002] an additional abstraction layer which would allow algorithm writers to completely separate the syntactic merging rules form the semantic merging ones is thus introduced by means of the semantic conflict function.

Generally speaking, a semantic merging policy is specified as a set of Semantic Merging Rules (SMR) with which the *Semantic-Conflict(SMR,*$O_r$*,*$O_l$*,CT)* determines whether two concurrent updates, $O_r$ and $O_l$, are semantically conflicting according to the context $CT$ on which they were generated. On one extreme, the function could automatically return *true/false* without human intervention if the *SMR* has been well formulated. Automatic detection of conflicts is the most desirable way. However, this is not at all easy to achieve. On the other extreme, the detection of conflicts could be completely manual if the *SMR* is unable to be formulated in any way. As a result, it is completely up to the user to determine what updates made by another user should be merged into his/her working copy, possibly with consultation with that user. Manual detection of conflicts is the most general way for all applications, although it is the least desirable way. In most cases, the detection of conflicts is the combination of automatic and manual detections. In other words, some conflicts can be automatically detected while others have to be detected by humans [Shen and Sun 2002].

In our project, we took a rather simple approach to this issue and defined conflicts in the following way: two operations are conflicting if they change

the same semantic unit, where the semantic unit is indicated by the working granularity level chosen by the user. The user can decide if (s)he wants to work at the word, sentence or paragraph level. If the user chooses, say, to work at the word level, it means that two concurrent operations changing the same word will *always* be conflicting. In order to achieve this, we provide the following implementation of the *semantic conflict* function:

```
1  boolean semanticConflict(Operation op1, Operation op2)
2  {
3      if (op1.getType() == OperationType.nop ||
4          op2.getType() == OperationType.nop)
5          return false;
6
7      if (level <= op1.getLevel())
8      {
9          for (int i = 0; i < level; i++)
10             if (op1.getIndex()[i] != op2.getIndex()[i])
11                 return false;
12
13         return true;
14     }
15     else
16         return false;
17 }
```

The two operations received as parameters are the two concurrent operations about which we need to say whether they are conflicting or not. The *level* variable that appears in the code is the global variable set by the user. This is how the user makes his/her choice of semantic unit (s)he wants to work on. A value of 4 for the level variable, for example, indicates that the user wants to work at the word level. A precondition for calling this function is that the two operations have to have the same level. Since it is only called from the merge method where only operations of the same level are compared, this precondition is always met. Another required precondition is that the context of the two operations is the same. This is also ensured by the merge method.

The first thing the function does is check if any of the operations is a NOP. If this is the case, it simply returns false since a NOP is never in conflict with any other operation. Further on, the level the user chose to work at is compared with the level of the operations (remember one of the preconditions said that the two operations need to be of the same level). If the operations are of higher level than the level of conflict then false is returned (line 16) since they are not conflicting (for example, if the level is set to word level and the two operations are insertion and deletion of, say, sentences, they are obviously not conflicting). If, however, they are of the same level or of lower level than the level of conflict set by the user, we need to check whether all their indices down to the level of conflict coincide. If so, we conclude they are conflicting and return true (line 13). Otherwise, if any of the indices down to the defined conflict level are different, false is returned (line 11) meaning that the two are not conflicting (they refer to different semantic units).

This is the *semantic conflict* implementation we shall use in the merge algorithm.

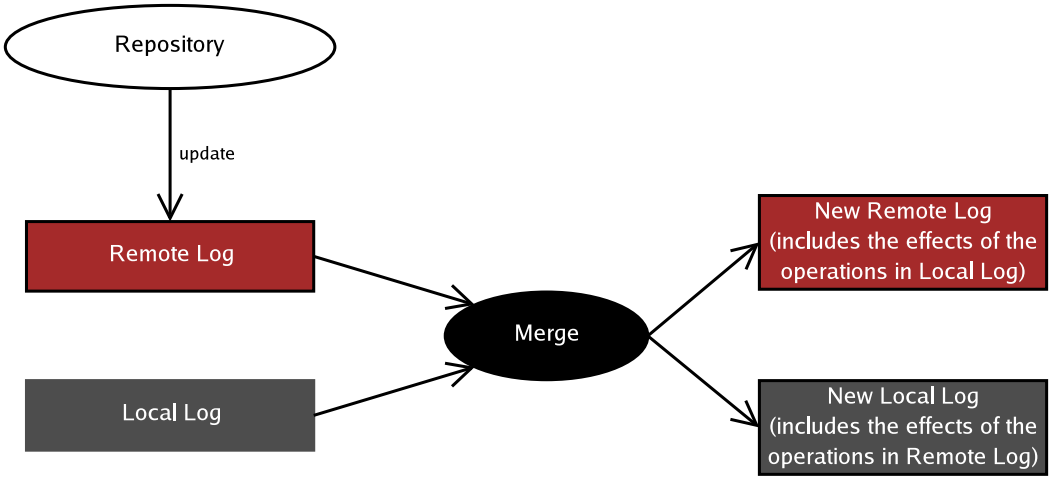


Figure 5.3: Merging two logs

The second function we need to define is the *remove operation* function. Its purpose is to move an operation from position $i$ in a log on the last position in the log by transposing (see 5.2.4) it one step at a time over operations $i+1$ through $n$. This function will be used when an operation from the remote log is conflicting with one from the local log and needs to be removed from the log. It cannot be simply removed since its effect has to be excluded from all the operations that come after it in the log. Therefore, we need this special *remove operation* function which achieves this. Its implementation is rather straightforward and it looks like this:

```
1 Operation removeOperation(int k, ArrayList log)
2 {
3     for (int i = k; i < log.size() - 1; i++)
4         Operation.transpose(log.get(i), log.get(i+1));
5
6     return log.remove(log.size()-1);
7 }
```

Lines 3-4 achieve the transposing of the operation at the end of the log, while line 6 removes the transposed operation from the log and returns it.

Now that these two functions are clarified, we can go on to presenting the basic merge algorithm. Figure 5.3 shows what the merge operation is supposed to do. This is the case when we would be dealing with a linear representation of the document. In our case, the situation (as we mentioned) is slightly different, meaning that we would not compare the two logs directly, but split them taking the tree structure into account and repetitively call the merge function.

As the figure shows, the input parameters of the merge operation are two logs, a remote and a local one, and the output parameters are also two logs, the modified local and the modified remote log, each of them changed in order to include the effects of the operations in the other one. The merge procedure itself is illustrated in listing 5.4.

```
1  public void merge(ArrayList remoteLog, ArrayList localLog,
2      ArrayList newRemoteLog, ArrayList newLocalLog)
3  {
4      for (int i = 0; i < remoteLog.size(); i++)
5      {
6          // save local log into CLL
7          ArrayList CLL = localLog.clone();
8
9          // save operation on position i in remote log into CRLi
10         Operation CRLi = remoteLog.get(i).clone();
11
12         for (int j = 0; j < localLog.size(); j++)
13         {
14             if (semanticConflict(remoteLog.get(i), localLog.get(j)))
15             {
16                 // restore remoteLog[i] from CRLi
17                 remoteLog.set(i, CRLi);
18
19                 // restore localLog from CLL
20                 localLog = CLL;
21
22                 // remove operation i from the remote log
23                 Operation o = removeOperation(i, remoteLog);
24                 i--;
25
26                 // add the inverse of the removed
27                 // operation to the local log
28                 newLocalLog.add(o.invert());
29
30                 // end the for loop
31                 break;
32             }
33             else
34             {
35                 symInclusion(remoteLog.get(i), localLog.get(j));
36             }
37         }
38
39         if (j == localLog.size())
40         {
41             newRemoteLog.add(remoteLog.get(i));
42         }
43     }
44
45     for (int i = 0; i < localLog.size(); i++)
46     {
47         newLocalLog.add(localLog.get(i));
48     }
49 }
```

Listing 5.4: the merge procedure

In the way in which we apply the general update algorithm, whenever we call the merge function there will be a current node which we shall have reached in our traversal of the tree. All operations from the remote log and from the local log (the ones we use as actual parameters when calling the merge function) will refer to that particular node. Thus, the *remoteLog* contains a list of remote operations that pertain to the current node (i.e. if the current node is a sentence, then the remote log will contain any operations that insert or delete words in that particular sentence and nothing else - neither of higher nor of lower levels). The *localLog* contains the local operations that refer to that node in the tree. The *newRemoteLog* is the list of operations that should be executed sequentially on the current document state of the working copy in order to update it. It will contain most or all the operations from the original remote log, modified in order to include the effects of those in the local one. We say that it might not actually contain all of the operations in the original remote log because some of them might be in conflict with the local ones and would then have to be removed. The *newLocalLog* will store the list of operations which will have to replace the local log pertaining to the current node. They will represent most or all the operations in the local log transformed in order to include the ones in the remote log. Additionally, it might also include the inverse of some of the operations in the remote log. Why this might be will become clear soon.

The largest part of the function takes place within the big for loop (lines 4-43) which iterates over the operations in the remote log. Thus, for each operation in the remote log we need to take the following series of actions:

- save a copy of the local log as it is at the beginning of the iteration, in case we need to restore it later (in line 20).
- save a copy of the current operation in the remote log (the one we have reached with our iteration over the remote log) in case it needs to be restored later (in line 17)
- iterate over all the operations in the local log and compare them one by one with the operation in the remote log by calling the *semantic conflict* procedure we described earlier. Here we have two cases:
    1. the two operations are not in conflict: what we do in this case is make sure that each one includes the effect of the other (the remote operation will be transformed to include the effect of the local one and vice-versa) by calling the *symmetric inclusion* function described in section 5.2.4. If you look at the big picture, if the remote operation will not be in conflict with any of the local operations, by the end of the inner iteration (the one over the local log) it will have orderly included the effect of each of the local operations and each of the local operations will have included its effect. In this form, the remote operation can be properly executed in the local context (since by means of the successive inclusions we have effectively changed its context to the local one). If you look at the even bigger picture, by the time all the operations in the remote log will have been iterated by the outer for loop, each of the operations in the local log will have included the effect of each of the operations in the remote log. Therefore, we shall be able to add them to the new local log, as their

context is now updated in order to include all the operations from the repository.

2. the two operations are in conflict: in this case there are several things that we can do. The one described in the algorithm is the simplest of them, namely to automatically choose the local operation as the winner of the conflict and cancel the remote one. We have also implemented different other policies for dealing with conflict, such as manual conflict resolution, in our project, but for the sake of clarity, we have not included them in the "cosmetized" code included here as it would have uselessly complicated things. Therefore, in what follows we shall simply assume that the local operation is always chosen to be kept.

   Returning to the algorithm, if the two operations are in conflict and the local one has to be kept, what we need to do is eliminate the remote operation from the remote log and change all things accordingly. The first thing to do is to reset the remote operation itself to its original form (line 17). This needs to be done because we have already changed it to include the effect of all the local operations up to the current one (with which it is conflicting) and, in order for us to be able to correctly remove it, we need to have it in its original form. Then, we also have to restore the local log to its form before making some of the local operations include the effect of the remote operation we are now removing. This is because once the remote operation gets removed, its effect will no longer have to be included in the local operations. The next thing to do is remove the remote operation from the remote log. The way to do this is by calling the *removeOperation* procedure we defined above in order to exclude the effect of the remote operation from all the operations in the remote log that come after it and also in order to make it include the effect of all these operations. Please refer back to the description of *removeOperation* to see how this effect is achieved. The reason why we also need to make the remote operation include the effect of those that come after it is that we need to append its inverse to the new local log and in order to generate that inverse in the current context we need to change the context of the original operation itself. It is just as if the local site had been aware of the operation and had willingly canceled it. Its inverse is added to the beginning of the *newLocalLog* so that we do not have to make it aware of the effect of the local operations (this would not be needed). The local operations themselves do not need to be aware of its effect (even though one might think so, given that it is added before them in the new local log) because what it does is cancel the effect of another operation (its original, uninverted, version from the repository) which the local operations have not been made aware of either. So, what happens is that the two effects which would have to be included into the local operations cancel each other out. In the end we add the inverse of the removed remote operation to the new local log (as we said earlier), decrease the value of the $i$ index by one (since one operation has been removed and, if we did not do this, an operation would be skipped in the outer

loop) and forcibly end the inner loop by the break instruction in line 31.

- add the modified remote operation to the new remote log if it has not been in conflict with any of the operations in the local log. The idea is that if the remote operation can be executed (i.e. it is not in conflict with any local operations) we need to add it to the new remote log so that it ends up being applied on the local document (in order to update it to the version from the repository).

Once the outer for is executed, all that is left to do is add all the (now transformed) operations from the local log into the new local log. At this point, we have properly populated the new remote log and the new local log and can end the execution of the merge function.

**The update algorithm**

This algorithm is the highest level algorithm which achieves the actual update of the entire local version of the document (represented as a tree) with the changes that have been committed by others to the repository. Therefore, one of its inputs is the entire remote log which the repository sends to this client when the client indicates that it wants to perform an update of the local version. The remote log represents the delta between two consecutive versions on the repository ($V_k$ and $V_{k+1}$). If the client needs to update from version $V_k$ to version $V_{k+n}$, what it will have to do is repetitively call the update method we describe below in order to successively update in one-version increments (i.e. from version $V_k$ to version $V_{k+1}$ first, from version $V_{k+1}$ to version $V_{k+2}$ second, and so on, with the last update being from version $V_{k+n-1}$ to version $V_{k+n}$; this is obviously transparent to the user). We would like to remind you that the operations in such a remote log are ordered according to their level (i.e. operations working on the highest level first and operations working on the lowest level last). In order to become more familiar with the structure of this remote log, please refer to section 5.3.1. The local log that the client commits in the commit stage (which is what section 5.3.1 describes) is the exact remote log which we receive here, when trying to perform an update. The remote log therefore contains a linearization of the logs that were initially part of a tree document structure.

The objective of the update method is to achieve the two effects we described above when talking about the basic merge algorithm at the level of the entire document tree (replace the local log associated with each node with a new one which includes the effects of all relevant operations in the remote log *and* execute 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 algorithm that achieves this is presented in listing 5.5. This is a very stripped down version of the actual algorithm. The main parts that have been taken out of it are those which allowed several types of conflict resolution policies. In the version we present here, the implicit policy is that the local version of the operations is always the one that is kept when conflicts arise. You can read more about conflict resolution policies in section 5.4.3.

Aside from the remote log, the other parameter of the update method is the *current node*, as we call it. The idea is that the method will be called recursively

```
1  void update(DocumentTreeNode currentNode, ArrayList remoteLog)
2  {
3      ArrayList remoteLevelLog = new ArrayList();
4      ArrayList localLevelLog = currentNode.getLog();
5
6      ArrayList newRemoteLog = new ArrayList();
7      ArrayList newLocalLog = new ArrayList();
8
9      int borderIndex = remoteLog.size();
10     for (int i = 0; i < remoteLog.size(); i++)
11     {
12         Operation op = remoteLog.get(i);
13         if (op.getLevel() == currentNode.getLevel())
14         {
15             remoteLevelLog.add(op.clone());
16         }
17         else
18         {
19             borderIndex = i;
20             break;
21         }
22     }
23
24     updateOperationIndices(localLevelLog, currentNode.getIndices());
25
26     merge(remoteLevelLog, localLevelLog, newRemoteLog, newLocalLog);
27
28     for (int i = 0; i < newRemoteLog.size(); i++)
29     {
30         applyOperation(newRemoteLog.get(i));
31     }
32     currentNode.setLog(newLocalLog);
33
34     int nrChildren = currentNode.getNoChildren();
35     ArrayList[] childRemoteLog = new ArrayList[nrChildren];
36     for (int i = 0; i < currentNode.getNoChildren(); i++)
37     {
38         childRemoteLog[i] = new ArrayList();
39     }
40
41     for (int i = borderIndex; i < remoteLog.size(); i++)
42     {
43         Operation op = remoteLog.get(i);
44
45         for (int j = 0; j < newLocalLog.size(); j++)
46         {
47             op = op.include(newLocalLog.get(j));
48         }
49
50         childRemoteLog[op.getIndex(currentNode.getLevel())].add(op);
51     }
52
53     for (int i = 0; i < currentNode.getNoChildren(); i++)
54     {
55         update(currentNode.getChildAt(i), childRemoteLog[i]);
56     }
57 }
```

Listing 5.5: the update procedure

in order to traverse the entire document tree. The *current node* represents the node we have reached in the traversal. Obviously, the method will initially be called using the root of the document tree as a parameter. Now let us walk through the method step by step and see what the main actions that it performs are.

The first few lines initialize the four lists we are going to be working with, *localLevelLog*, *remoteLevelLog*, *newLocalLog* and *newRemoteLog*. These lists serve the exact same purpose as they did in the basic merge algorithm. We are using different names for the remote and local log here in order to mark the fact that we are actually only working with parts of the remote and local node at a time, namely those parts that contain operations which refer to the current node. Actually, this is also true for the new remote and new local log as well, but the variable names would have become too long, so we just kept them as they were. The only list of the four which is initialized from the very beginning with something other than the empty list is the *localLevelLog*. As you can see in line 4, it is initialized with the log of the current node. The rest of the lists will be populated as we advance in the method.

Lines 9 through 22 have the purpose of adding all the remote operations pertaining to the current node to the *remoteLevelLog*. The way to do this is iterate over the remote log as long as we have operations whose level is identical to the level of the current node. As soon as we reach the first operation which is of lower level than the level of the current node, we exit the loop. This implementation is correct because at any time the remote log will be ordered in decreasing operation level order. We have explained above why this is true for the first call of the update method (the call from outside the method). This is also valid when we recursively call the method because the remote logs we use as parameters are always kept in this order. In addition to populating the *remoteLevelLog* these part of the code also saves the index of the first operation that refers to a lower level than the level of the current node in the *borderIndex* variable.

The next thing to do is change the indices of all the operations in the *localLevelLog* so that they correspond to the current position in the tree of the node whose log they belong to. What happens is that during the update algorithm nodes might get inserted or deleted from the tree, as we apply the modified remote operations on the local version of the tree in lines 28-31. This implies that the tree structure changes dynamically as we traverse it. As the positions of the nodes change, it is clear that all operations that appear in the log of the nodes whose position has changed will no longer have valid indices. For example, if we had a DeleteChar(1,3,4,5,'d') and paragraph 1 has been shifted two positions to the left by the insertion of two new paragraphs before it, we have to change the operation to DeleteChar(3,3,4,5,'d') in order for it to be correct. So this is what procedure *updateOperationIndices* does: it takes all the operations in the *localLevelLog* and updates their indices so that they correspond to the current position in the tree of the node they belong to. An alternative to this would have been to make all operations in all the logs in the tree include the effect of all executed remote operations. We hope it is clear without further explanations that this would have been by far less efficient than the variant presented here.

Next, the basic merge algorithm described in the previous section is called with the four lists as parameters. What the merge algorithm achieves should already be clear from above, so we shall no longer dwell on this.

Once the merge of the *remoteLevelLog* and the *localLevelLog* is finished and the two new logs (*newRemoteLog* and *newLocalLog*) are obtained, the next thing we need to do is apply the operations from the *newRemoteLog* on the local copy of the document in order to update it. This is achieved in lines 28-31. The local log of the current node is then replaced with the *newLocalLog* (see the explanations on the basic merge algorithm to understand why this is done).

The last thing we need to do is divide the remaining part of the remote log (the part containing the operations that were not merged at this level, i.e. the ones from *borderIndex* on) among the children of the current node and call the update method recursively. If you think about it, the initial remote log is formed of four parts (the insert/delete paragraph operations, the insert/delete sentence operations, the insert/delete word operations and the insert/delete character operations). The first call to update will process all the insert/delete paragraph operations. What we do afterwards is divide all the remaining operations (from the second, third and fourth part alike) among the paragraphs in the tree (i.e. all operations operating on paragraph 1 in one list, all operating on paragraph 2 in another and so on). Then we recursively call the update method with all of these separate lists in turn. Every recursive call will thus have a remote log as parameter which contains three parts (insert/delete sentence operations, insert/delete word operations and insert/delete character operations). The process is repeated, i.e. in every call all the insert/delete sentence operations will be processed while the rest will be divided among the children of the current node and the update method will be called again. This goes on until the last level is reached.

Now let us see how the actual division is done. Lines 34-39 build the lists in which the operations will be stored. We have called these lists *childRemoteLog*. The for loop in lines 41-51 performs two actions:

1. make each operation in the remote log (starting with the one on position *borderIndex*) include the effects of all the operations in the *newLocalLog*. This is necessary as operations in the new local log are of higher level than the ones remaining in the remote log and thus can influence the context of the remote operations. To clarify things, let us consider the case of a remote InsertWord operation. The operation has four indices: the paragraph, sentence, word and character index. As this is a fourth level operation, it will only get processed after having gone through four successive update calls. At the first call, lines 45-48 will modify its paragraph index in order to include the effects of the insert/delete paragraphs in the new local log. At the second call, the same lines will modify its sentence index and at the third call its word index. The character index is going to be modified inside the basic merge algorithm by means of symmetric inclusion. By the time the operation is included into the new remote log by the merge algorithm and ends up being executed it will have been properly modified in order for its execution to be correct.

2. add the remote operation into the corresponding *childRemoteLog*. We choose the *childRemoteLog* to put it into by looking at the (already modified by the previous action) index corresponding to the level of the current node.

By the end of the for loop, all remote operations will have been transformed

and placed in the correct list. The last thing to be done (lines 53-56) is to recursively call the update method with each of the previously created lists of operations as remote logs.

### 5.3.3 The checkout stage

The third operation that an asynchronous system needs provide is the checkout of a version of the document from the repository. The user usually identifies the version (s)he wants to check out by means of a version number. The system we have built also allows users to identify the version number closest to a given date and time. If I, as a user, know that the version I am interested in was committed (either by me or by somebody else) sometime around the $20^{th}$ of May 2004, 14:53 PM I will just feed this information to the system and obtain the version number that was committed first after this moment in time. Once the version number is found out, it can simply be used to request the version it identifies.

The way to check out version, say, $V_k$ is to ask the repository for all the operations which represent the delta between version $V_0$ and version $V_k$. The repository will concatenate the operations which transform version $V_0$ in $V_1$, those that transform version $V_1$ to $V_2$ and so on until those which transform version $V_{k-1}$ to $V_k$ into a single list and send it back to the client. All the client has to do then is to iterate through this list of operations and apply them on his/her initial empty version of the document as below:

```
1  void checkout(int versionNr)
2  {
3      // obtain the operations which represent the delta between
4      // version 0 and version versionNr from the repository
5      ArrayList remoteOperations = repository.getOperations(0, versionNr);
6
7      for (int i = 0; i < remoteOperations.size(); i++)
8      {
9          applyOp(remoteOperations.get(i);
10     }
11 }
```

## 5.4 Additional functionalities

In this section we shall address all the remaining functionalities of the system which are complementary to the basic ones described above. These are additional facilities we have created for our users and which add either efficiency or extra functionality to the system as a whole. Even though we are presenting all of them under the same heading, there is no dependency among them whatsoever. Each of them is self-contained and can function independently in the absence of the others. The facilities we are going to refer to are *log compression*, *direct user synchronization* and *conflict resolution policies*.

### 5.4.1 Log compression

Log compression is the mechanism by which we reduce the size of the local log by means of transforming several lower level operations into a single higher level operation which achieves the same effect as the combined effect of the initial operations. An example would be compressing several InsertChar operations which insert characters in the same word into one single InsertWord operation which inserts the word which is formed of the characters which the initial InsertChar operations inserted. In the same way we can combine several InsertWord operations into a single InsertSentence and several InsertSentence operations into a single InsertParagraph. In this way we ensure the obtaining of an optimal efficiency when merging versions of the document because instead of having to analyze tens of separate lower level operations we can simply analyze just one or two higher level operations.

The log compression procedure is called before an update is made, in order to compress the local log (which we need to merge with the remote one) to a minimum. It is also called before a commit is performed in order to send a reduced form of the local log to the repository for bandwidth preservation (both for the commit which is about to take place and for subsequent checkouts performed by other users) and for storage space preservation on the repository. Also, the procedure is called before a direct user synchronization procedure is enacted, both on the side of the user who requests the synchronization and on the side of the user who accepts it (see section 5.4.2 to find out more about direct user synchronization).

Listing 5.6 illustrates the compression procedure. It is a recursive procedure having as parameter the *current node* for which the compression is performed. Naturally, the outside call to it will use the root of the document tree as the current node. Lines 3-7 in the procedure represent the recursive call and the termination condition. Thus, we stop executing the procedure when we reach the leaves in the tree (i.e. nodes representing the word level). Compression is achieved in a bottom-up fashion, hence the recursive calls take place before the processing of the current node. In this way, when we get to actually process the log of the current node we are assured that all lower level logs have already been compressed.

As for the processing of the log of the current node itself, this is achieved in lines 9-50 of the algorithm. The situation from which we start working is the one in which the insertion of a word, for example, is distributed at the sentence and the word level. To make this clear, think of a user typing in the word “hello”. This will generate an InsertWord(...,’h’) at the sentence level and four InsertChar’s at the word level (InsertChar(...,’e’), InsertChar(...,’l’), InsertChar(...,’l’), InsertChar(...,’o’)). What we want to do is combine these operations into a single InsertWord(...,’hello’). The same happens in the case of sentences as well. What we need to do is iterate over the log of operations associated with the current node (remember: these are all operations which affect the children of the current node, not the current node itself) and, if it is indeed an insert operation, achieve the effect we described above.

The first thing we do is skip over delete operations and any operation (insert or delete) that is locked (operations get locked when direct user synchronization is used; you can read about this in the following section). The reason we skip over delete operations is that they have already been applied on the local

```
1  void compressLog(DocumentTreeNode currentNode)
2  {
3      if (currentNode.getLevel() == TreeLevel.word)
4          return;
5
6      for (int i = 0; i < currentNode.getNoChildren(); i++)
7          compressLog(currentNode.getChildAt(i));
8
9      ArrayList log = currentNode.getLog();
10     for (int i = 0; i < log.size(); i++)
11     {
12         Operation op = log.get(i);
13
14         if (op.getType() == OperationType.delete ||
15             op.isLocked())
16             continue;
17
18         int levelIndex = op.getIndex()[currentNode.getLevel()];
19
20         boolean unitDeleted = false;
21         for (int j = i + 1; j < log.size(); j++)
22         {
23             Operation opj = log.get(j);
24             int opjLevelIndex = opj.getIndex()[currentNode.getLevel()];
25
26             if (opj.getType() == OperationType.insert)
27             {
28                 if (opjLevelIndex <= levelIndex)
29                     levelIndex++;
30             }
31             else if (opj.getType() == OperationType.delete)
32             {
33                 if (opjLevelIndex < levelIndex)
34                     levelIndex--;
35                 else if (opjLevelIndex == levelIndex)
36                 {
37                     unitDeleted = true;
38                     break;
39                 }
40             }
41         }
42
43         if (unitDeleted)
44             continue;
45
46         op.setContent(currentNode.getChildAt(levelIndex).toString());
47
48         emptyLog(currentNode.getChildAt(levelIndex));
49     }
50 }
```

Listing 5.6: the compress procedure

tree, which means that the node they refer to no longer exists in the tree so there is actually nothing left to compress. Actually, the compression of several DeleteChar's into a DeleteWord is achieved "on the spot", at the moment when the last character of the word is deleted, so we do not have to deal with it here anymore. The reason we skip over locked operations is that the whole point to locking them in the first place is that they do not get compressed. Why this is necessary will become clear when you read about direct user synchronization. For now, just accept the fact that locked operations have to stay as they are, they cannot be compressed.

Now we are at the point where we know that the operation at hand (*op*, in the code) is an insert operation. This operation refers to a certain child of the current node (it actually represents the insertion of that child). However, the content of the operation is (if you think of the example we explained above) not the current content of the child node (since all operations other than the insertion of the first character/word/sentence are currently stored one level lower). Therefore, what we need to do is replace the content of the operation with the current content of that child and empty the entire child's log. If we come back to the example with InsertWord(...,'hello'), this translates to the fact that we want to change the content of the InsertWord(...,'h') operation from the current 'h' to 'hello' and thus transform it to InsertWord(...,'hello'). Next we want to delete the four InsertChar operations (in other words, empty the log of the node which represents the 'hello' word) since their effect is now contained in the InsertWord. Therefore, lines 46 and 48 illustrate the changing of the content of the current operation and emptying of the child's log, respectively.

We are left, however, with the problem of determining the *current* position of the child whose content we need to use. We cannot simply refer to the appropriate index of the current operation (*op*) in order to determine this child because subsequent operations (which were executed after the current operation) might have changed the position of that child. Lines 18 through 41 are there to solve this problem. The idea is fairly simple: we read the position of the child at the moment of insertion (given by the current operation index which refers to the level of the tree equal to that of the current node) and store it in the *levelIndex* variable and we look at all the operations that come after the current one in the log of the current node to see if they insert or delete children on a position lower than *levelIndex*. If this is the case we increase or decrease the *levelIndex* by one in order to track these changes. Aside from this, there is another case which needs to be handled, namely that when one of the subsequent operations actually deletes the semantic unit which the current operation inserted. If this case arises, the content of the insert operation no longer has to be modified because the delete operation is sure to have as content the same content which is currently in the insert operation (i.e. if we inserted the word 'hello' - which means that we have the InsertWord(...,'h') at the sentence level and the four InsertChar operations at the word level - and then we deleted it, the content of the delete operation will be 'h', not 'hello', meaning that the pair of insert/delete operations is consistent; as a side note, the fact that the delete operation indeed has that content is ensured in lines 44-45 of the *applyOperation* function, which was described in section 5.2.2). We avoid changing the content of the current operation in this case by marking that fact that the node it inserted is deleted later in line 37.

Once the *levelIndex* has been properly modified we can now use it to read the

content of the child we are interested in and replace the content of the current operation with it in line 46. Finally, as we were saying, the log of the child is emptied.

When the compression algorithm finishes its execution, all operations which can be combined in higher level operations will have been combined and the distributed log is now in its smallest form (at least in what the number of operations is concerned).

### 5.4.2 Direct user synchronization

A very useful (at least in our opinion) functionality our system offers is direct user synchronization. This means that any pair of users of the system can engage in a synchronization process without passing the operations they exchange through the repository. This, we believe, is useful when two or more users work together as a smaller group within the big group and would like to make sure that the versions they offer to the rest of the group are fairly correct. For example, think of the case of two students and a professor working together at a project. The professor is the group leader and usually supervises the work of the students. The two students keep working at a part of the project, but they might not want the professor to waste his/her time with checking things the two have not yet agreed on. Therefore, they can use direct user synchronization to work together and, only when they believe they have reached a state which they want the professor to see, actually commit the changes to the repository and thus make them available for the professor. Naturally, there are hundreds of imaginable scenarios where such a facility could be used.

Direct user synchronization, however, makes things much more difficult for the system because it has to ensure the consistency of the repository in the presence of such a synchronization method. What happens is that when a a user synchronizes with another, (s)he will receive the operations that the other executed at his/her site and integrate them in his/her local working copy. Next, if the user who integrated the changes of the other commits to the repository, (s)he will commit his/her own changes as well as those (s)he received by synchronization. No problem so far. However, when the user whose changes have been committed by the other user wants to commit him/herself we are faced with a problem: some of his/her operations have already been committed by somebody else. If the system were to commit those same operations all over again, user intention would obviously no longer be preserved. And this is just the easiest of cases. You can imagine the problems that arise when user 1 takes the changes of user 2 and then user 3 takes the changes of user 1 (which also include those of user 2). Also, users could synchronize repetitively with one another which makes the problem ever more difficult. The basic idea is that we had to find a way in which the document remains consistent with the user intentions regardless of how many users synchronize and how many times they do it.

The solution was to assign an unique identifier to each operation and use this identifier in order to find duplicate operations. In order to ensure that the same operation does not get committed to the repository more than once, what we did was add an extra condition to the include method which says that if the two operations (one of which is including the effect of the other) have the same unique identifier then the one including the effect of the other is transformed

into NOP (lines 45-46 in listing 5.2). Let us now see how this solves the problem of duplicate operations. We return to the case from above in order to show this. In order to make this discussion easier, imagine that the two users are called John and Mary and that John wants to synchronize with Mary (synchronization is a one-way process, therefore when we say John wants to synchronize with Mary what we mean is that John wants to become aware of Mary's operations; Mary, on the other hand, will not become aware of John's operations unless she initiates a synchronization of her own). In order for two users to be able to synchronize they both have to have the same base version (i.e. the version they started working from). Assume this version is $V_k$. In order to make things simple, suppose both users have performed one single operation ($O_J$ and $O_M$ respectively). Now, John synchronizes with Mary and obtains $O_M$ from her. He will transform it and execute it locally (read further on to find out how this is done). Then John commits his changes to the repository. This means he sends $O_J$ as well as the modified version of $O_M$ to the repository. Mary waits for another while (maybe she performs some more operations, but this is irrelevant for our discussion) and then tries to commit herself. As somebody else (in this case, John) has committed before her, she will first have to perform an update in order to be able to commit. When she updates, the repository sends her the two operations, $O_J$ and the modified $O_M$. Her local log is formed of the original $O_M$ only. If you think of how the merge is done, you will remember that, at some point, a symmetric inclusion of the modified $O_M$ and the original $O_M$ will be performed. Given what we have explained earlier about the modification of the include method, the effect of this is that both operations will become a NOP (when the modified $O_M$ includes the original $O_M$ the modified $O_M$ will be transformed to NOP and when the original $O_M$ includes the modified $O_M$ the original $O_M$ will be transformed to NOP). This means that the new local log will only contain a NOP and the new remote log will only contain a modified version of $O_J$. This is indeed correct as:

- the remote operation $O_M$ will no longer be executed locally (it would have been wrong if it had been executed because its effect is already achieved by the fact that the original $O_M$ was already executed locally when it was generated).
- the local operation $O_M$ will have been modified to a NOP and thus, after the log compression, will disappear from the local log. Therefore, it will not be re-committed to the repository (it would have been wrong if it had been re-committed because it is already there once - committed by John).

Even though this is just the simplest example one can imagine, the truth is that the exact same process takes place with more complicated situations as well: all operations which have already been committed to the repository (either by their original generator or by somebody who has taken them by means of synchronization) will not end up committed more than once because anybody else who would want to commit them would have to do an update first and, during that update, the duplicate operations will be transformed to NOP. Also, an operation already executed on a site will not be re-executed because the remote operation will be transformed to NOP during the symmetric inclusion. This is how the problem described above is effectively solved.

There is another aspect of the problem which is worth mentioning, namely the locking of operations. By default, as we described in the previous section, the compress log algorithm will compress lower level operations into higher level ones. This becomes a problem when direct user synchronization is used because some operations which have been taken by others might end up compressed into a higher level operation and, in this case, their unique identifier will be lost. The effect of this would be that we would no longer be able to correctly detect all duplicate operations resulting in possible re-executions (violating user intention). The solution to this is to *lock* all operations exchanged during a synchronization session and by this stop both the originating site and the receiving site from compressing them. In this way we are sure the original unique identifier is never lost and, consequently, the system will not fail in finding all the duplicate operations.

Finally, let us now see how the synchronization is actually done. The idea is fairly simple, given that we can reuse most of the code from the update algorithm in order to achieve this. Therefore, what we do is use the same code which is used when performing an update with some minor differences which we shall now point out. Before actually initializing the synchronization procedure, the remote user will need to give permission to the local user to take his/her operations. Once this is accomplished the remote user will behave exactly as a repository in what the local user is concerned, i.e. it will send a list of operations which will represent the equivalent of the remote log. These operations are exactly the same ones the remote user would send to the repository when performing a commit (meaning that they are serialized in the same manner: higher level operations first, lower level operations last). As far as the local user is concerned, the local log which is used is the same one as in the case of an update (the log distributed in the document tree).

Actually, the only difference from the update procedure given in listing 5.5 is that we have to remove line 32 from the algorithm. This is the line that changes the log of the *current node* to the new local log obtained after calling the merge method. This is no longer correct in the case of a synchronization because here we want to keep the local operations as they were (i.e. in an unmodified form) and simply append the modified form of the remote ones (the ones obtained by means of synchronization) to the already existing local log, which is achieved in the for loop in lines 28-31 (as the *applyOperation* method also adds the operation it applies to the log). Apart from this minor difference the rest of the procedure remains unchanged.

### 5.4.3 Conflict resolution policies

The final thing we believe is worth mentioning in what the facilities of our system are concerned is represented by the possibilities we offer to our users for resolving conflicts. By conflict resolution policy we refer to the way in which one operation of two conflicting operations is chosen. There are several conflict resolution policies our system allows and we are going to describe them below.

The basic distinction between conflict resolution policies is given by whether it is done automatically or manually. Basically, automatic resolution of conflicts means that the user will not be prompted for a decision regarding any kind of conflict, even though the result might not be exactly what (s)he wanted. Manual resolution, on the other hand, means that, if conflicts among operations arise,

the user will be asked to manually make a decision (i.e. choose one version or the other).

Further on, each of these types of conflicts has subtypes. If the user chooses automatic resolution, the default behavior is that the local operations will always be the ones to be kept in case of conflict. For example, if the conflict unit is set to the word level and if the local user inserted an 'f' in a word, while the operation on the repository represents the insertion of an 'n', then automatic conflict resolution will always decide to keep the insertion of the 'f' and cancel that of the 'n'. Although the interface of our system currently does not allow this, it is algorithmically possible (with minor modifications) to allow the user to choose the opposite behavior, i.e. set things up so that automatic conflict resolution always chooses the remote operation as the one to keep (and cancel the local one).

Automatic conflict resolution policies can also be used in direct user synchronization (as conflicts can arise during such synchronization as well). In this case the user has two choices: synchronize as master or synchronize as slave. The former case implies that the local operations will be chosen as the winners of conflicts, while the latter implies that the operations of the other user will get to be kept should conflicts appear. This master/slave policy thus allows the user to decide whether him/herself or the one (s)he is synchronizing with is the one who should get to have 'the final say'.

Turning to manual resolution policies now, the user again has two choices here. One of the choices is to use operation comparison as manual resolution policy. This means that whenever two operations are in conflict, we shall present the user with the effects of both and prompt him/her to decide which of the effects should be preserved. This is, however, highly user-unfriendly. Just take the simple case when each of the two users inserts three characters in the same word. This will yield three InsertChar operations on each site. Since the word is the lowest conflict unit possible, all operations from one site will surely be in conflict with all operations on the other. If this operation comparison policy is employed the user will be prompted to choose between all pairs of conflicting operations just to decide what (s)he wants the final word to be. You can imagine that this is not a very quick method of solving conflicts when we have possibly hundreds of words to decide about. Besides, it is also somewhat unuseful because it is pretty clear that if one user modified a word in a certain way (by adding a few characters to it) and the other user in a different way, the user performing the merging will probably want either one word or the other, not a combination of them (which translates to the fact that (s)he will probably want to choose either all local operations modifying the initial word or all remote operations modifying the word and not a combination of the two). Consequently, chances are that he will not want to choose the fact that he wants to keep one set of operations or the other three times (once for each InsertChar) but just once. Therefore, this technique is more of a debugging conflict resolution policy than one we would actually see a real user use.

The other manual conflict resolution policy we provide is the one we call conflict unit comparison. With this policy we are trying to fill the gap which the previous technique left open, namely to collect all local operations that affect the selected conflict unit (word, sentence or paragraph) on one hand and all remote operations that affect it on the other and prompt the user to choose one set of operations or the other. In order to do this, we present the user with

the two different effects achieved by applying all the local operations pertaining to the conflict unit where the conflict appeared and by applying all the remote operations pertaining to it. The user gets to analyze the two different effects and choose the one that suits him/her. By changing the conflict unit (s)he uses (s)he can decide just how many choices (s)he wants to make when performing an update or synchronize (the higher the conflict unit, the fewer the choices).

Both manual conflict resolution policies can be used with normal updates as well as with direct user synchronizations.

## 5.5 Algorithm performance

In this final section of the chapter we shall try to address the matter of the efficiency of our algorithm. This analysis has to be made since one of the most important goal we had was to obtain an algorithm which would run faster than existing ones. However, we shall not make a very formal efficiency analysis since this would be rather difficult (given the complexity of the algorithms used). We shall just try to give a rough outline of how efficient certain parts of the system run.

#### The update algorithm

The most important algorithm in the system is the update algorithm. Therefore this is the one we are going to focus most of our attention on. The description of the algorithm can be found in section 5.3.2 (more specifically, in listing 5.5).

The first thing we need to mention is that the worst case efficiency is, theoretically speaking, the same as in the case of linear algorithms, namely $O(n^2)$. The case when this happens is when all operations in the two lists refer to the same semantic unit. In such a case, it is clear that the actions of the linear and the tree algorithm would have to do the same thing, i.e. process each pair of operations (with one operation of the pair from the remote log and the other one from the local log), therefore obtaining a $O(n^2)$ efficiency. However, this case appears extremely seldom in practice because of two reasons:

- if the semantic unit is any other unit than the word, then even though the efficiency is $O(n^2)$ both for the linear and for the tree algorithm, still the number of operations which achieve the same effect is always smaller in the case of the tree representation. For example, if the semantic unit is the sentence (meaning we have operations all taking place in the same sentence) then for every InsertWord from the tree version we would need anywhere between 1 and the maximum number of characters in a word InsertChar operations in the linear version. And the numbers only get higher as we go up in the tree. Therefore, only when both the local and remote operations refer to the same word do we have the actual same efficiency of the two algorithms.

- the second reason is that if is very very seldom, in practice, to have all operations (both remote and local) refer to the same semantic unit. Statistically speaking, in 99.99% of the cases users change more than just one word/sentence/paragraph, so the actual worse case scenario appears very infrequently.

That being said, we can now start the analysis of what happens in most cases. Lines 9-22 of the algorithm simply iterate over the first $k$ operations of the remote log, so we clearly have linear efficiency there. The *updateOperationsIndices* iterates over the operations in the local log of the current node so, after a complete traversal of the tree, each operation in the local log will eventually have its indices corrected (if required), meaning that we shall have processed all operations in the local log, i.e. $O(n)$ efficiency. So far, so good.

Next, the merge algorithm is called, which takes two lists of operations as parameters. Say the $m$ is the number of operations in the local log and $n$ the number of operations in the remote log. Then the merge algorithm has an efficiency of $O(n * m)$. However, it should be noted that the values of $n$ and $m$ are just small numbers usually, since they count only the operations (both local and remote) that refer to the current node. In a linear system, the entire update procedure consists of calling the same merge algorithm with the entire local and remote logs as parameters. This means it obtains an efficiency of $O(n * m)$ on the whole, where $n$ and $m$ are the full sizes of the two logs. With our algorithm, however, only small groups of operations get analyzed together, which greatly improves the overall performance of the algorithm. If we consider a typical case when we have, say, 100 operations remotely and 100 operations locally, it is reasonable to think that of those 100 roughly 10 are insert/delete paragraph operations, 20 are insert/delete sentence operations, 50 are insert/delete word operations and 20 are insert/delete character operations. Additionally, it is also reasonable to assume that the 20 insert/delete sentence operations are divided among 4 paragraphs, the 50 insert/delete word operations are divided among 10 sentences and the 20 insert/delete character operations are divided among 10 words. Remember, this is just an example. Assuming the worst case scenario when both the local and the remote operations refer to the exact same paragraphs/sentences/words what we would get is: $O(10*10)$ for the insert/delete paragraphs, $O(4 * 5 * 5)$ (4 executions of the merge algorithm, all with a 5-operation local log and a 5-operation remote log) for the insert/delete sentences, $O(10 * 5 * 5)$ for the insert/delete words and $O(10 * 2 * 2)$ for the insert/delete characters. This adds up to a total of $O(100 + 100 + 250 + 40) = O(490)$. If we compare that with the $O(100 * 100) = O(10,000)$ which we obtain in the case of a linear algorithm, it is clear that the performance gain is tremendous. And think that this was the worst case scenario, when local and remote operations all refer to the exact same semantic units. The real case is usually much better. However, we have to keep in mind that the overhead of traversing the tree and dividing the remote list of operations adds a bit to the tree version of the algorithm, making its performance slightly worse than merely that of the merge calls which we were discussing above.

Applying the modified remote operations (lines 28-31) is again an $O(n)$ procedure. Lines 41-51, however, are a bit trickier. The fact that we have to include the effect of all the operations in the new local log into the operations in the remote log has the following effect: each operation from the remote log will have to include all the operations from the local logs that belong to nodes which are on the path from the root of the tree to the node to which the operation refers. However, if you think about it, the progress from the linear version is still great. This is because if in the linear case we have all operations including each other, here we have each remote operation including at most $p + \frac{s}{sc} + \frac{w}{sc*wc} + \frac{c}{sc*wc*cc}$ operations, where $p$, $s$, $w$ and $c$ represent the number of insert/delete paragraph,

insert/delete sentence, insert/delete word and insert/delete character operations respectively, *sc* represents the average number of sentences in a paragraph which are modified locally, *wc* the average number of words in a sentence that are modified locally and *cc* the average number of characters in a word that are modified locally. However, this formula only applies for insert/delete word operations. In the case of higher level operations the last one, two or three terms of the sum do not appear (the best case is that of insert/delete paragraphs where the number of included operations is just $p$, the first term). A concrete efficiency formula is hard to derive for this inclusion procedure, but it is clear that it is a very small fraction of $m * n$.

In conclusion, even if both the linear algorithm and the tree algorithm have the same theoretical efficiency of $O(n*m)$, the performance of the tree algorithm is significantly better because the coefficients hidden by the efficiency formula of the tree algorithm are much smaller than the ones hidden by the efficiency formula of the linear algorithm.

**The compression algorithm**

Another algorithm which is interesting in what its efficiency is concerned is the compression algorithm (see section 5.4.1). This is also a recursive algorithm. Each execution of it takes $O(n^2)$ in the number of operations in the log of the current node. If we assume an even distribution of the operations in the entire tree, we would end up with an overall efficiency of $O(nodes * (\frac{n}{nodes})^2)$ or $O(\frac{n^2}{nodes})$. Given that a node's logs contains a number of k operations in average (with k a constant depending on the time between two consecutive commits) it means that $nodes = \frac{n}{k}$. Therefore the overall efficiency of the compress algorithm is $O(\frac{n^2}{\frac{n}{k}}) = O(n)$, i.e. the compress algorithm is a linear algorithm.

The previous computation of efficiency holds in the initial assumption of an even distribution of operations in the tree. This, however, is usually not the case. Instead, in most cases, we have operations confined to a certain subtree of the entire tree, because users work on certain paragraphs at a time, leaving the others alone. The effect of this is that the $k$ constant from above will have higher values. However, they will still be constant values, therefore the efficiency itself will remain linear. The running times, on the other hand will usually be somewhat higher, given that the hidden constant is not as small as in the case of an even distribution.

All other algorithms used in the system are fairly fast, most of them having obvious linear efficiencies, so there is no need to discuss them here any further. The final conclusion is that the system as a whole has either a linear (for the commit and checkout stages) or an $O(n * m)$ (for the the update stage) efficiency, which is the same as linear systems have. The performance, however, if compared to that of linear systems, is significantly superior given the fact that the hidden constants that appear in the case of our algorithm are much smaller. This means that our work was indeed successful and that the system we have created does bring an improvement over existing implementations.

# Chapter 6

# Design issues

In order to build the entire project around the basic algorithms, there were several parts of the system we had to develop. We dedicate this chapter to explaining the specific design approaches used for building each of these parts. We shall describe each of the different components separately first and then show how they are put together in order to obtain the final system.

The main parts of our application are the core system, the network communication module, the parser module and the client-side graphical user interface.

## 6.1 The core system

The core system represents the largest part of our system as it is the one that integrates all the algorithms presented in the previous chapter. The entire core system resides on the client side of the application (which makes it a so-called "fat client"). This means that the logic of all operations described previously (such as commit, update and so on) are handled by the client, while the repository is merely a container with support for various simple requests. The client is the one where the processing actually takes place.

Most of the algorithms are enclosed within the methods of the Document class, while a certain amount of them reside in the Operation class. Figure 6.1 tries to show this distribution of algorithms by presenting a class diagram of the main classes that compose the core of the application.

The arrows in the figure illustrate that the Document class references a DocuementTreeNode object as the root of the document tree and the DocumentTreeNode references a list of Operation objects (which form the log associated with each tree node) and also another DocumentTreeNode object which represents the parent of the current node.

You can see that the operation transformation methods (include, exclude, transpose, symmetric inclusion etc.) all belong to the Operation class. Also, this class provides accessors to all of an operation's fields such as type, level, global unique identifier, content, indices and so on.

Each DocumentTreeNode object (as the name suggests) stands for a node in the tree that represents the document. It is mostly a container class which additionally provides a means of managing child nodes. The rest of the methods (both accessor and selector) are self-explaining.

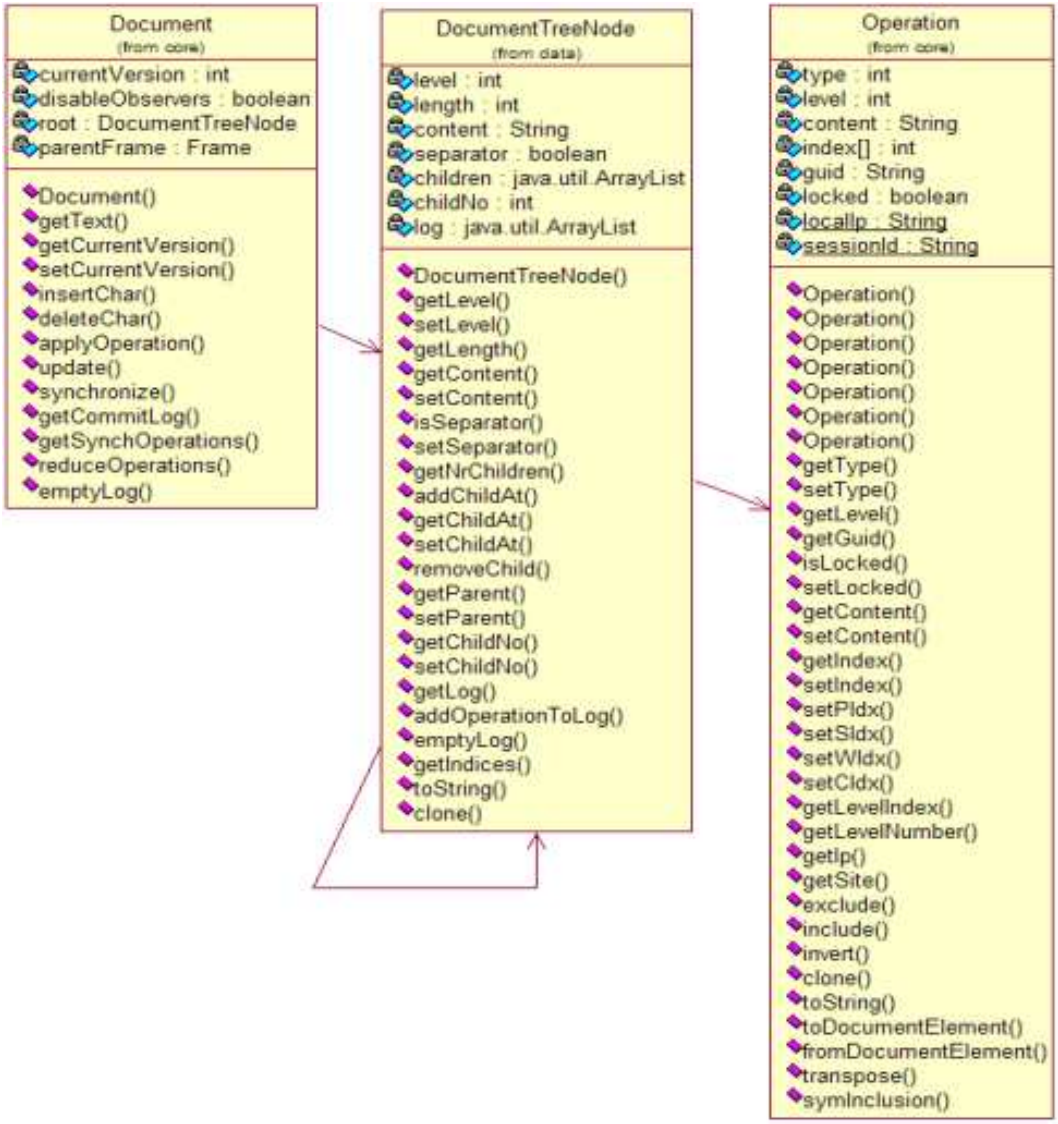


Figure 6.1: The core system class diagram

A text document is represented by an instance of the Document class. Such an object provides methods for inserting and deleting characters in and from the text document, applying operations, updating and synchronizing this document with remote changes, obtaining the complete log of operations which were executed on the document and clearing this log. The way all these methods are implemented has been described in the previous chapter.

Even though the core system represents the essence of the entire application, we shall not discuss it here any further since it has been extensively covered in chapter 5. We have only mentioned it in this chapter as well in order to clarify the general framework in which the operations already described appear.

### 6.1.1 The XML representation of operations

There are two methods we have not mentioned so far which appear within the Operation class and which we would like to discuss here, namely the *toDocumentElement* and the *fromDocumentElement* methods. These methods are used for creating an XML representation of the Operation object they are called on and for recreating an Operation object from its XML representation, respectively. The actual output (and input, respectively) of the two methods are, in fact, DOM elements, not clear text. The DOM element generated by *toDocumentElement* will be combined with others and eventually end up written to an XML file, while the DOM element from which an operation is recreated (by means of *fromDocumentElement*) is obtained as a subtree of a larger DOM representation of an XML file.

The Document Type Definition (DTD) which describes how an operation is represented in XML is reproduced below:

```
<!ELEMENT operation (guid,((pIdx)|(pIdx,sIdx)|(pIdx,sIdx,wIdx)|
  (pIdx,sIdx,wIdx,cIdx)),content)>
<!ATTLIST operation type (0|1|2) #REQUIRED>
<!ATTLIST operation level (0|1|2|3) #REQUIRED>
<!ELEMENT guid (#CDATA)>
<!ELEMENT pIdx (#CDATA)>
<!ELEMENT sIdx (#CDATA)>
<!ELEMENT wIdx (#CDATA)>
<!ELEMENT cIdx (#CDATA)>
<!ELEMENT content (#CDATA)>
```

Each operation has to have its type and level represented as attributes, while the rest of the components of the operation appear as elements (the global unique identifier, the content and the indices). As described by the DTD, an operation can have either one, two, three or four indices, depending on its level (document level, paragraph level, sentence level or word level).

An example of a representation of an operation could be the following:

```
<operation type="0" level="3">
 <guid>192.168.0.1\%26e9f9:fcf46ff876:-7ffa\%26e9f9:fcf46ff876:-7ff7</guid>
 <pIdx>0</pIdx>
 <sIdx>0</sIdx>
 <wIdx>8</wIdx>
 <cIdx>6</cIdx>
```

```
 <content>s</content>
</operation>
```

The operation denoted by this representation is InsertChar(0,0,8,6,'s') with the specified global unique identifier.

## 6.2 The network communication module

There were several communication requirements which we wanted to meet with our application. The most of important of these was the need for reliable communication between the clients and the repository as well as between the clients themselves. The necessity of this requirement should be obvious to the reader. Imagine the simple case when a single packet containing an operation is lost. This can have dramatic effects on the final result since all the following operations might no longer be properly defined, leading to inconsistencies either in the repository (if the packet was a commit packet) or in the client documents (if the packet was a checkout or update packet). Secondly, we also aimed for a communication mechanism which would be usable with very heterogenous environments (including firewall protected hosts, masquaraded hosts and so on) since clients of an asynchronous system could technically gain access to the network in very different ways.

The natural solution to these requirements was the use of the robust network communication mechanism made available by the standard Java implementation: Remote Method Invocation. We assume the reader is already familiar with RMI and the way it is used, so we shall only try to describe the specific way in which we based our network module on it. Figure 6.2 illustrates the RMI servers we implemented in our application.

As the figure shows, there are two distinct servers that appear in the system. One of them is the repository server and the other is the direct user synchronization server. The repository server is instantiated by the server side of the system and provides all the functionality associated with the system repository. A direct user synchronization server is instantiated on each client and has the purpose of allowing other clients to connect to the client running the server and retrieve its operations by means of the direct user synchronization mechanism.

The repository offers some standard functionality (represented by the *getOperations*, *getCurrentVersion* and *commit* methods) and a more "special" method which allows the clients to identify the version number closest to the a certain moment in time (*getVersionByTimestamp*). The basic functionality of the repository is to allow its clients to store operations in the document file (by remotely calling the *commit* method) and retrieve the operations which represent the delta between two consecutive versions (by remotely calling the *getOperations* method).

Since when two or more commit operations are tried concurrently, only one should be successful (while the others have to fail), we had to employ a mutual exclusion mechanism which would allow us to achieve this. The solution (given the Java implementation) was to mark the *commit* operation as *synchronized*, which ensured that only one client would be able to call it at a certain moment in time. Naturally, the *getCurrentVersion* method is also *synchronized* since the *commit* method changes the version number as well and it would be wrong to return an outdated version number to the clients.

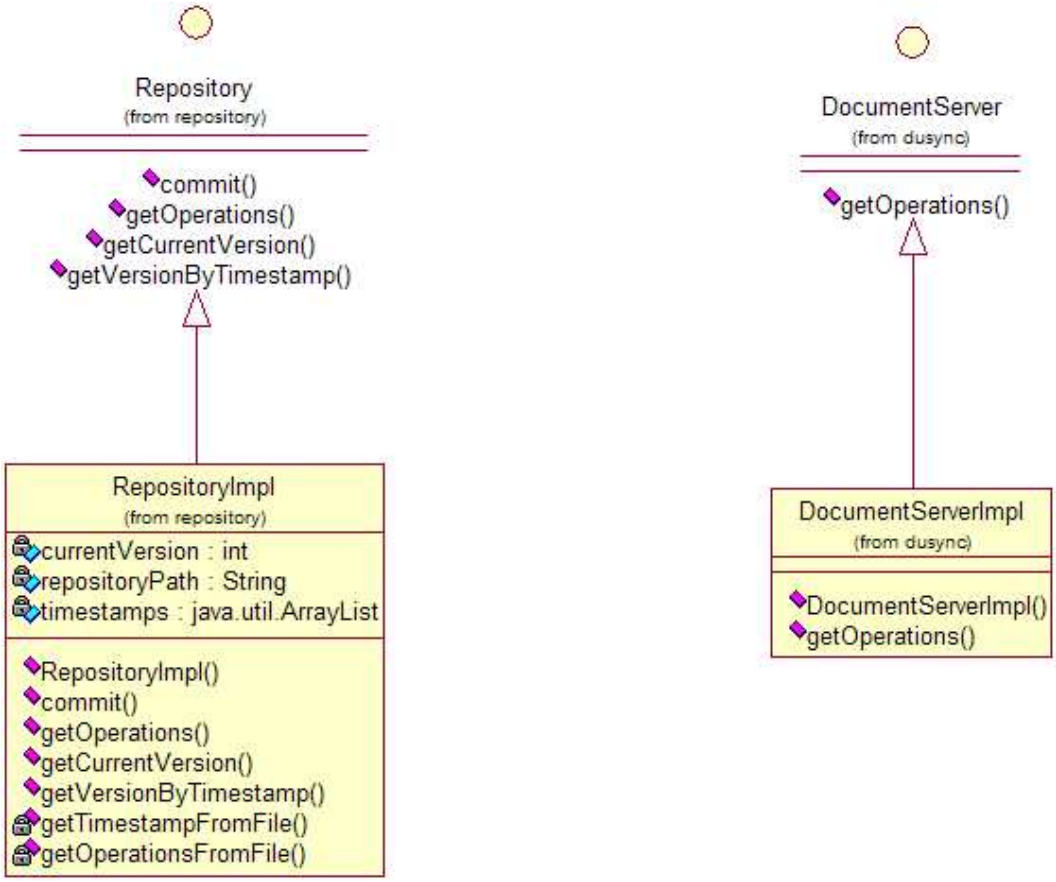


Figure 6.2: The network communication module

As for retrieving version numbers based on timestamps, this is a fairly simple operation. Each version is timestamped at the time it is committed with a date and time. This timestamp is recorded in the version file along with the list of operations which comprise that particular version. When a client wants to retrieve a certain version, (s)he usually knows the approximate time at which it was committed. (S)he will then submit this approximate timestamp to the repository and will expect a version number in return. The repository will traverse the list of timestamps and will return the version number associated with the timestamp that is closest to the one submitted by the client. Once the version number is obtained, the client can simply perform a checkout of that particular version in order to analyze (or even use) it.

The direct user synchronization server is slightly more simple, as the only operation it has to support is the retrieval of the local operations. This is achieved through the *getOperations* method. The method returns all the operations that are found in the local log of the client on which it runs (after previously locking them all - see section 5.4.2 for further details).

The rest of the network communication module is found on the clients. This

part simply connects to the RMI registry on the repository server (or on the client it wants to synchronize with directly) and retrieves the appropriate registered remote object. Once this is accomplished, RMI allows clients to call the methods of the remote object as if they were local.

The configuration of the RMI server (i.e., its IP address and port) are manageable by the client by means of the graphical user interface.

### 6.2.1 XML representation of document versions

Returning to the repository server, the final aspect to mention is the XML representation used to store version files. Each set of operations which comprise the change between two consecutive versions is stored in a separate XML file which is described by the following DTD:

```
<!DOCTYPE version-update [
  <!ELEMENT version-update (timestamp,operation+)>
  <!ELEMENT timestamp (year,month,day,hour,minute,second,mili)>
  <!ELEMENT year (#CDATA)>
  <!ELEMENT month (#CDATA)>
  <!ELEMENT day (#CDATA)>
  <!ELEMENT hour (#CDATA)>
  <!ELEMENT minute (#CDATA)>
  <!ELEMENT second (#CDATA)>
  <!ELEMENT mili (#CDATA)>
  <!ELEMENT operation ...>
]>
```

The "operation" element has already been described in section 6.2.1 and therefore has not been repeated here. As you can see, a version update consists of a single timestamp identifying the moment in time at which it has been committed followed by a list of one or more operations. An example of such a version update could be the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<version-update>
  <timestamp>
    <year>2004</year><month>6</month><day>5</day>
    <hour>14</hour><minute>2</minute><second>43</second><milli>265</milli>
  </timestamp>
  <operation type="1" level="0">
    <guid>192.168.0.1\%cafb56:fcf4795cd1:-7ffa\%cafb56:fcf4795cd1:-7ff0</guid>
    <pIdx>7</pIdx>
    <content>This is a test.</content>
  </operation>
  <operation type="1" level="0">
    <guid>192.168.0.1%cafb56:fcf4795cd1:-7ffa%cafb56:fcf4795cd1:-7fef</guid>
    <pIdx>5</pIdx>
    <content>This is another test.</content>
  </operation>
</version-update>
```

This represents a commit consisting of two operations, both of the type InsertParagraph. The version number itself is stored in the name of the XML file. This is a trick used in order to be able to quickly retrieve the version changes particular to a specific version without having to traverse a (possibly) large number of files. By using the version number as part of the file name of the XML file, we can simply go directly to the correct file when a request is made by a client.

## 6.3 The parser module

One of the other important parts of our application resides in the module which deals with the generation of a tree document representation of a random text (transforming a linear structure - the text stream - into a hierarchical one - the tree representation of the document using DocumentTreeNode objects as nodes in the tree). This is what we call the parser module. Its entire functionality resides in a single class which is illustrated in figure 6.3.

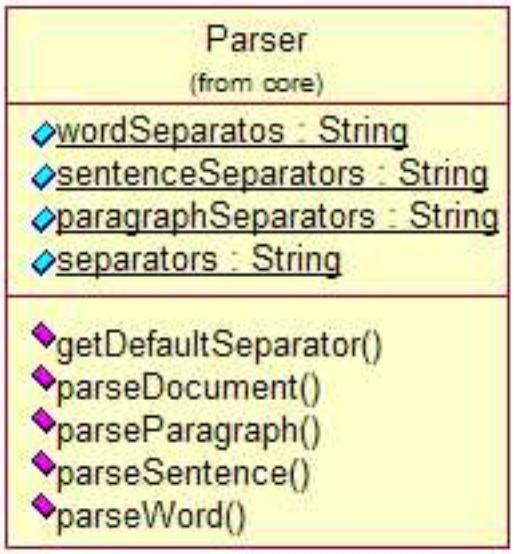


Figure 6.3: The parser class

The function of this class is to parse either an entire document, a paragraph, a sentence or merely a single word from a text stream. If we speak of a document, the parsing is done until the end of the stream is reached. If, on the other hand, we speak of lower level semantic units, the parsing is done either until a separator specific for that semantic unit is reached or the end of the stream is reached.

The functionality of the parser is based on the assumption that all texts are strings in the language defined by the following grammar (described in BNF):

```
document ::= paragraph document |
             ""
```

```
paragraph ::= actual_paragraph |
              paragraph_separator

actual_paragraph ::= sentence actual_paragraph |
                     sentence

paragraph_separator ::= "\n" | "\r"

sentence ::= actual_sentence |
             sentence_separator

actual_sentence ::= word actual_sentence |
                    word

sentence_separator ::= "." | "!" | "?"

word ::= actual_word |
         word_separator

actual_word ::= character actual_word |
                character

word_separator ::= " " | "," | ";" | ":"

character ::= any_charcter_other_than_separators
```

The main aspect we would like to underline here is that semantic unit separators (paragraph separators, sentence separators and word separators) are treated similarly to the semantic units they separate, i.e. each paragraph separator is viewed as a separate paragraph, each sentence separator is viewed as a separate sentence and each word separator is viewed as a separate word.

Based on this grammar, the *parseDocument* method parses an entire text document generating paragraph, sentence, word and character nodes which correspond to the actual structure of the text. But this way of generating the document tree representation is less used in our project. Instead, as the document is usually generated by successively applying the operations received from the repository and as these operations insert at most full paragraphs, the document tree representation is built progressively by creating entire subtrees which represent either a paragraph, a sentence or a word using the appropriate *parseXXX* method of the Parser class and adding them (the subtrees) in the correct position in the tree.

Apart from the parser, two other methods relevant to the management of the document tree representation are the *insertChar* and *deleteChar* functions from the Document class. These deal with the local insertions and deletions of single characters (as opposed to modifying the document by means of applying remotely generated operations). Each insertion may simply modify the content of a word node (if the user inserts a character in an existing word), create a new word node (if the user inserts a word separator or the first character of a new word), create a new sentence node (if the user inserts a sentence separator or the first character of a new sentence) or create a new paragraph node (if the

user inserts a paragraph separator or the first character of a new paragraph). In the case of inserting the first character of a new paragraph or sentence there are actually three (or two nodes, respectively) that are being built, not just one. This is because when a new paragraph is created the first sentence and first word of that paragraph are also created and when a new sentence is created the first word of that sentence is also created. Deletion takes place in the same way: when a character is deleted, depending on whether it is the last character of a word/sentecence/paragraph or not, the appropriate operations are generated and the appropriate node(s) is(are) deleted.

## 6.4 The Graphical User Interface

Finally, the last piece of the puzzle consists of the client side graphical user interface which allows the user to perform a series of actions, among which the most important are:

- edit a text document
- commit the changes made to a text document
- update the local working copy with the changes committed by others
- check out a particular version of a document from the repository
- choose the granularity level to work at
- choose the conflict resolution policy to be used
- manage a list of users with whom direct synchronization can be employed
- synchronize with any user from the list mentioned above

All these should be performed in a simple, intuitive manner which is exactly what we tried to accomplish with our graphical user interface. Figure 6.4 is a representation of the class diagram which contains the main GUI components. Aside from the ones represented in the figure, the system also contains some additional GUI-related classes which have not been represented due to lack of space.

The figure also shows the Document class once again in order to illustrate that this is actually the so-called "data model" of the graphical user interface, as the entire functionality of the GUI strongly relates and interacts with an instance of the Document class. Every change or action the user performs by means of the GUI is actually reflected in the state of the Document instance used. The GUI basically provides a way to control the parameters of the Document instance and to trigger the various methods which determine changes in the document.

Aside from the main GUI, the figure also shows a few classes called *XXXDialog*. These all represent separate screens which allow the user to perform various actions such as choosing the repository server to use, managing the list of users (s)he wishes to directly synchronize with, choosing between document variants in case of conflict and so on.

All in all, the GUI is a fairly simple one which allows the user to perform all the actions allowed by the system. We have also attached a screen shot of our graphical user interface (figure 6.5) so that the reader can also visualize this component of the system put to work.

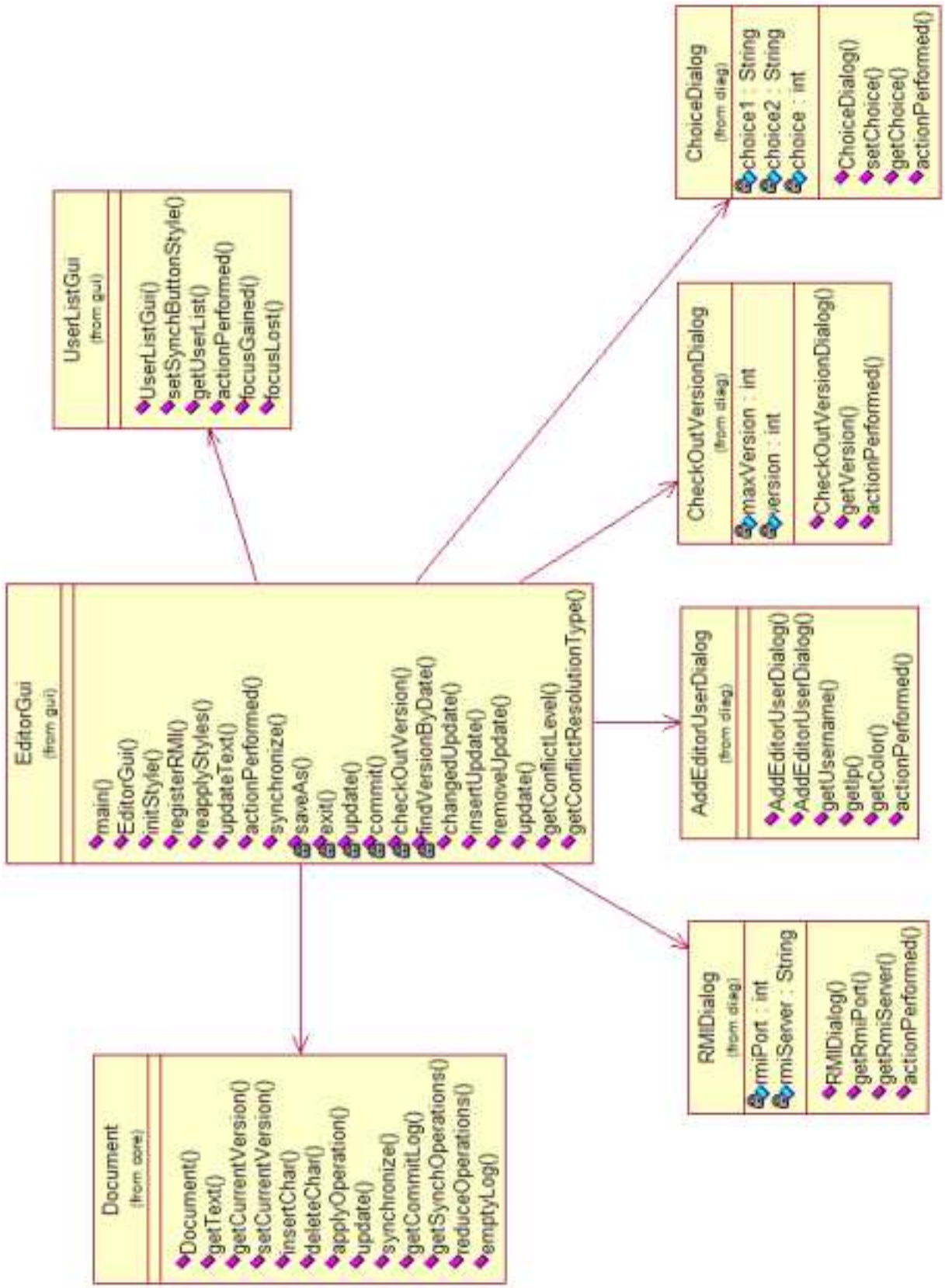


Figure 6.4: The graphical user interface

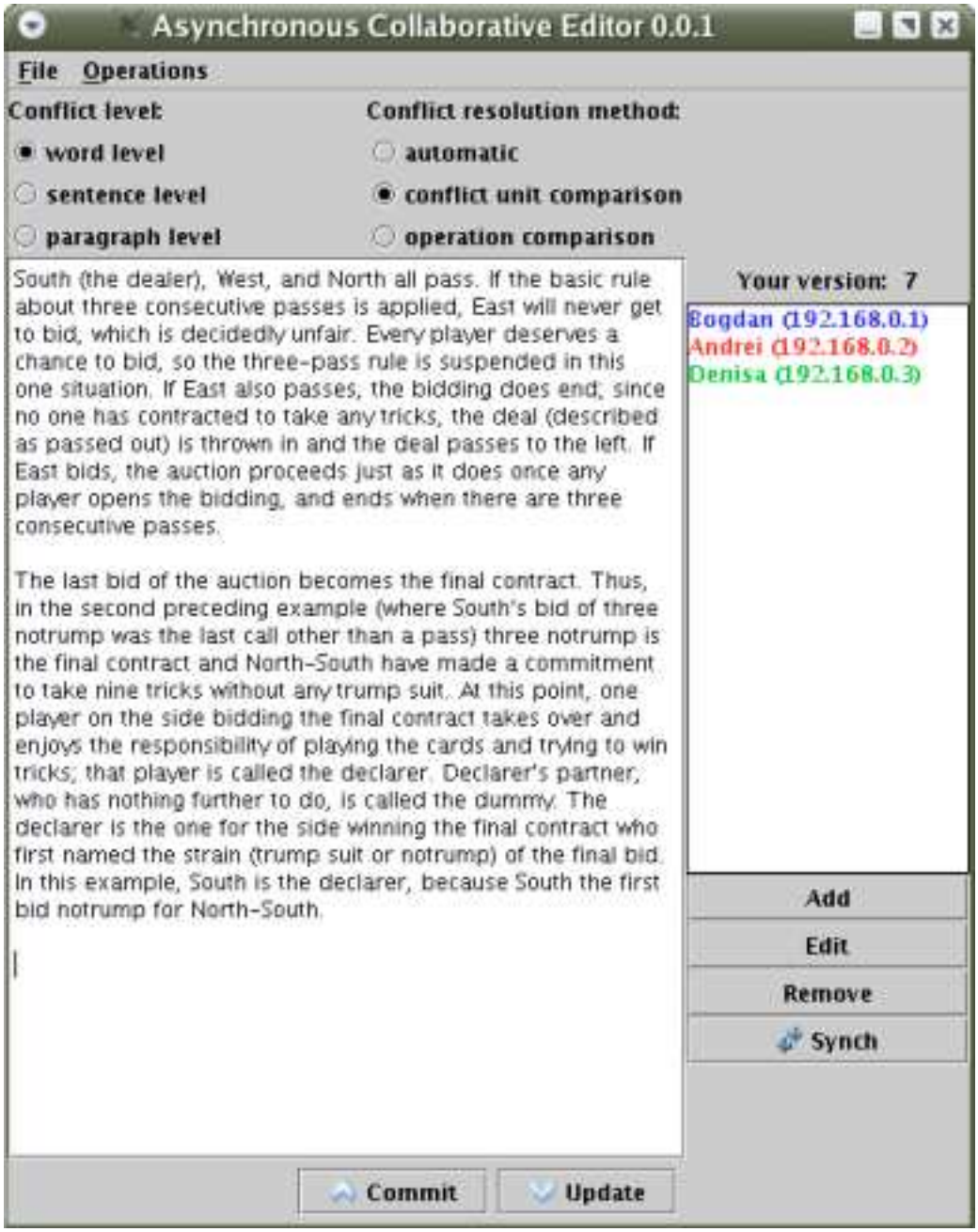


Figure 6.5: A screen shot of the client GUI

## 6.5 Putting it all together

As all the subsystems of our application have been briefly introduced, let us now illustrate how they all work together in order to provide a fully-functional and integrated system. We do this by means of the following figure:

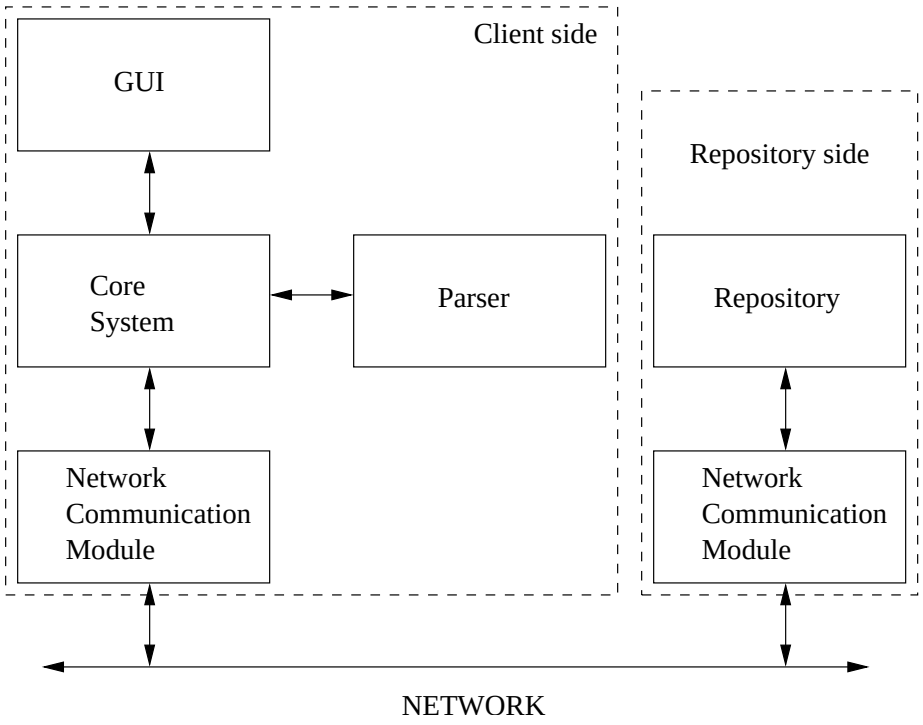


Figure 6.6: The entire system

The figure illustrates that the client side is governed by the core system which is the central part of the system. The GUI is a "wrapper" over the core system which allows the editing of the text document and ensures a simplified user control over the core system. The parser module is used by the core system in order to transform linear representations of text into equivalent tree representations. The communication (over any network) between the client and the repository is achieved by the network communication module.

The functioning of all of the subsystems that appear in the figure has been described in the previous sections of this chapter. The overall architecture is highly modular and, consequently, easy to extend. This means that the tree structure can be changed, the operation set can be enlarged with new operations and the GUI (for various reasons) could be redesigned without any influence on the other modules.

All in all, we have adopted a modern, modular, architecture in the application built around our core system, the final purpose of which was to allow users to benefit from the performance improvements we provide. The architecture, however, came second in our efforts as the largest amount of time was allocated for the development of the algorithms described in the previous chapter.

# Chapter 7

# Future Development and Conclusions

This chapter presents some ideas which we think could be taken into account in order to direct future efforts in the group. We end the chapter with some brief conclusions which try to sum up our achievements.

## 7.1 Future development

There are several things which we thought about during our work worth mentioning here, hoping that they might be useful to current or future members of the group. These are all just ideas, not actual solutions to clear problems and should be regarded as such. Some of them might turn out to be feasible, while others might turn out to be rather difficult to achieve.

#### Mixed text/graphical editor

As currently we have two distinct asynchronous editors available, a text editor and a graphical editor, it seems like a good idea to work towards an integration of the two, such that we can create a system which would allow the user to insert graphics within the text (s)he is editing. The way we see this, the system should treat the graphic unit as a special character (which could seen as be a separate semantic unit, either a word or a sentence or a paragraph). In this way, the text algorithm can be applied without any modifications (with the exception of the introduction of the special character represented by the graphic) in order to edit the document. None of the operations, however, would have to be modified because if the graphic is a special character then we can simply use the same insertion/deletion operations as the ones for normal characters. The benefit of this would be that the merging algorithm would work exactly in the same way and no new inclusion/exclusion procedures would have to be created.

As far as the graphic itself is concerned, the graphical editing subsystem could be used there to facilitate its collaborative editing. In this way, the system would automatically switch between text and graphical style of asynchronous collaborative editing and store separate logs for the text editor as well as for each separate graphic that appears in the document.

This would be a feature with a lot of potential for most users, since there are many cases in which text documents need to include figures representing or describing the concepts from the text. The idea seems to have a rather straightforward implementation (given that the two subsystems already exist and seem to be working well), but there might be unexpected issues which might arise during the integration.

Aside from allowing the simple graphics which the existing graphical editor currently supports, the system could also permit the integration of various image formats into the editor. Of course, there would be no possibility to edit such images (especially not in a collaborative manner), but still they could be inserted and deleted. They too would have to be regarded as a special character from the point of view of the text editor, but they would have to be differentiated from the normal graphics since there would be no possibility of editing them.

**Extension of the editor features**

A more difficult extension of our text editor would consist in adding attributes (such as font, size, color, bold, italic and underlined text) to the characters. We say this would be significantly more difficult than the previous idea because both the data structure representing the document and the set of operations would have to be changed. The data structure would have to provide the possibility of knowing exactly what attributes each semantic unit has (and overwriting of the attributes by child nodes). The set of operations, on the other hand, would have to be extended in order to offer the possibility of changing each attribute separately.

These changes would have further implication in most of the algorithms used throughout the system. The merging algorithm would have to be transformed in order to cope with new types of conflicts and with new types of operations. The compression algorithm would have to be extended in order to be able to compress attribute changes both by means of combining several attribute changing operations at a lower level into a single attribute changing operation at a higher level and by means of canceling operations which have been overwritten by newer ones (this is case with all attribute changing operation which change the same attribute of the same semantic unit).

The most difficult issue, however, would be to change the inclusion/exclusion functions in order to deal with the fact that some of the new operations might have an effect on more than just one semantic unit (for example, the user could change the font of two and a half paragraphs). This would probably imply changing the whole concept of inclusion and exclusion, because it might be the case that the effect of including an operation might change the current operation to two different new operations (think of two users, one of which changes the color of an entire paragraph, while the other changes the color of a sentence - neither the first, nor the last - in that paragraph to a different color than the first user; when the change paragraph color would include the change sentence color it would have to be transformed in at least two new operations - one that changes the color of all the sentences before the one changed by the second user and one that changes the color of all the sentences after that one).

By combining this development of the text editor and the one mentioned previously we would obtain a collaborative editor suited for high-level text editing

with support for fonts, color, text size and graphics. This is, of course, still far from what current (non-collaborative) text editors offer (support for creating tables, text alignment, etc.) but it would a significant step forward. The final goal would obviously be the creation of an collaborative editor which would offer all the features of current non-collaborative editors, but there is still a lot of work which has to be done before reaching that stage.

**Extending the current data structure**

As we have mentioned throughout this paper, the current merging algorithm is scalable to working on trees with more than just the four current levels (paragraph, sentence, word and character). We therefore propose as a project topic the extension of the current editor with what we call editor modes. These could be selected by the user based on the type of document (s)he is working on. There could be several predefined modes (such as book, report, article, etc.) which would imply the use of various tree structures and sets of operations by the editor.

In a book mode, for example, certain semantic units should be added to the model (such as chapters, sections and subsections). Operations should also support the insertion/deletion of such units. The issue that would still have to be dealt with is that of finding the right separators for such units. Currently, it is pretty obvious what the paragraph, sentence and word separators are. In the case of units like chapters, however, the situation is bit more complicated. Possible solutions would be to introduce some special markers as chapter separators or to decide that a certain number of \n's separate a chapter from the other.

Either way, such editing modes would contribute to an even more increased efficiency of the entire system, especially if the chosen editing mode corresponds to the actual type of document that is being edited. To further support the idea, we believe that such an addition would be rather easily implementable.

**The split/merge issue**

Future work in this area could also focus on finding a better solution to the split/merge problem than the one we used. In a nutshell, the split/merge problem is this: when the user adds a character that is a word/sentence/paragraph separator in the middle of a word/sentence/paragraph (s)he is effectively *splitting* an existing semantic unit into two new semantic unit; when the users deletes such a separator (s)he is *merging* two existing semantic units into one. The actual problem arises when another user makes some modifications in the original semantic unit (in the case of a split) or in one of the two original semantic units (in the case of a merge). An operation encoding such a modification is very difficult to transform in order to be executed on the site of the user which performed the split or the merge. More precisely, it is very difficult to preserve user intention when doing such an operation. For example, in the case of a split, when one user inserts a word in the same place where a sentence separator (i.e. a '.') was inserted by the other user it is difficult to tell which of the two new sentences the inserted word should belong to.

Currently, our solution to the problem is to treat a split as a deletion of the old semantic unit and the insertion of the two new semantic units and a

merge as the deletion of the two old semantic units and the insertion of the new semantic unit. This ensures that the consistency of the local working copies is kept, but strange effects may appear in certain cases. In order to solve this, it is clear that two new special operations, split and merge, need to be introduced and the inclusion/exclusion procedures need to be modified in order to take these operations into account. However, this is not as simple as it sounds, as many research projects have dealt with the issue and a fully working solution has not yet been found. The split/merge problem is, consequently, still an open issue in the field of collaborative editing, and further efforts can and should be directed towards solving it.

#### XML editor

Another idea that reaches a bit further than a mere extension of the current work would be the creation of a collaborative XML editor. This is actually something we have attempted up to certain extent. What we had in mind was the creation of a DTD-driven collaborative editor which would take a DTD as input and allow several users to work together in order to graphically edit an XML document which would have to conform to the specified DTD. The tricky part here would be the ensuring of the conformity of the final document to the DTD (while allowing intermediate, local, version to be incorrect). This is something we have have given some thought to and appears to be no child's play.

The main difficulty arises from the fact that we wanted the entire editing to be done by means of a graphical interface. The problem we got stuck on was how to create such an interface which, on one hand, should not allow the user to stray away from the grammar but also be aware that there might be cases when the grammar should be relaxed (since there might be other users filling out certain parts of the document which, when merged with the work of the current user, would yield a final correct document). Had we had more time to think about this, some solutions to this problem might have appeared. Nevertheless, we think it is a good project topic for future consideration.

## 7.2 Conclusions

This project had the purpose of studying and improving existing techniques in the field of asynchronous collaborative editing. The result of this work was the design and implementation of an entire asynchronous editing system which allows a group of users to work together in order to create and modify a text document. The special achievements of this project were the creation of a system based on the hierarchical representation of the text documents and, more importantly, a system whose algorithms work with hierarchical logs of operations (as opposed to linear logs used in the other existing systems). We have successfully designed some new algorithms especially suited for this purpose and have informally proven that the performance improvements over their linear counterparts are significant.

We also think that another improvement worth mentioning is the possibility of allowing the user to choose the granularity level (s)he wants to be working at. This, to the best of our knowledge, is something no other system is currently

offering. In addition to that, another unique feature our system provides is the choice of the conflict resolution policy to be employed, allowing for automatic as well as manual resolution with several choices in both cases. Last, but not least, we are aware of no other system which offers the possibility of direct user synchronization as ours does.

There are still some unsolved issues which need to be addressed in the future (some of which we have mentioned in the previous section of this chapter), but the system as it is now is a fully functional one which has proven to work correctly and also to be very helpful with the editing of some chapters of this paper itself. We believe our work is valuable, holds a lot of potential and is a prototype whose commercial implementation (with a few additions, perhaps) would be most welcome by real users.

Our project fits into a greater project developed in the Global Information Systems group at Eidgenössische Technische Hochschule Zürich and current intentions suggest that it will be part of a future integrated system which would allow various types of collaborative editing, with the final aim of providing groups of people working together with a complete tool for editing as many types of documents as possible.

# Bibliography

[Collins 2004] Collins-Sussman, B.: Version Control with Subversion. *Online http://svnbook.red-bean.com/svnbook/book.html*, May 2004.

[Dourish 1996] Dourish, P.: Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. *Proceedings of the 1996 ACM Conference on Computer-Supported Cooperative Work*, Boston, USA, 1996, pp. 268-277.

[Ignat 2002] Ignat, C., Norrie, M.: Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems. *The 4th International Workshop on Collaborative Editing, Computer Supported Collaborative Work*, New Orleans, USA, Nov 2002.

[Ignat 2003] Ignat, C., Norrie, M.: Customizable collaborative editor relying on treeOPT algorithm. *Proceedings of the 8th European Conference on Computer Supported Collaborative Work*, Helsinki, Finland, Sep 2003, pp. 315-334.

[Ignat 2004a] Ignat, C., Norrie, M.: Extending real-time editing systems with asynchronous communication. *Proceedings of the 8th International Conference on Computer Supported Collaborative Work in Design*, Xiamen, P. R. China, May 2004.

[Ignat 2004b] Ignat, C., Norrie, M.: CoDoc: Multi-mode collaboration over documents. *Proceedings of the 16th International Conference on Advanced Information Systems Engineering*, Riga, Latvia, Jun 2004.

[Imine et al. 2003] Imine, A., Molli, P., Oster, G., Rusinowitch, M.: Proving correctness of transformation functions in real-time groupware. *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work*, Helsinki, Finland, Sep 2003.

[Ionescu and Marsic 2000] Ionescu, M., Marsic, I.: An arbitration scheme for concurrency control in distributed groupware. *Proceedings of the 2nd International Workshop on Collaborative Editing Systems, An ACM CSCW'2000 Workshop*, Philadelphia, USA, Dec 2000.

[Molli et al. 2002] Molli, P., Skaf-Molli, H., Oster, G., Jourdain, S.: Sams: Synchronous, asynchronous, multi-synchronous environments. *The 7th International Conference on Computer Supported Collaborative Work in Design*, Rio de Janeiro, Brazil, Sep 2002.

[Molli et al. 2003] Molli, P., Oster, G., Skaf-Molli, H., Imine, A.: Using the transformational approach to build a safe and generic data synchronizer. *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*, Sanibel Island, Florida, USA, pp. 212-220.

[Munson 1994] Munson, J. P., Dewan, P.: A flexible object merging framework. *Proceedings of the 1994 ACM Conference on Computer-Supported Cooperative Work*, Chapel Hill, USA, 1994, pp. 231-242.

[Nedevschi 02] Nedevschi, S.: Concurrency control in real-time collaborative editing systems *Diploma thesis*, Zürich, Switzerland, June 2002.

[Shen and Sun 2002] Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. *Proceedings of the 10th International Conference on Cooperative Information Systems*, Springer Verlag, USA, Nov 2002, pp. 304-321.

[Sun et al. 1996] Sun, C., Yang, Y., Zhang, Y., Chen, D.: A consistency model and supporting schemes for real-time cooperative editing systems. *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Jan 1996, pp. 582-591.

[Sun et al. 1998] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, vol. 5, no. 1, Mar 1998, pp. 63-108.

[Sun and Ellis 1998] Sun C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proceedings of ACM Conference on Computer Supported Cooperative Work*, Seattle, USA, Nov 1998, pp. 59-68.

[Bitkeeper Home Page] Bitkeeper home page. *Online http://www.bitkeeper.com/*, May 2004.

[VSS Home Page] Microsoft Visual SourceSafe home page. *Online http://msdn.microsoft.com/vstudio/previous/ssafe/*, May 2004.

[CVS] Concurrent Versions System. *Online http://en.wikipedia.org/wiki/Concurrent_versions_system*, May 2004.

[Subversion Home Page] Subversion home page. *Online http://subversion.tigris.org/*, May 2004.