Diss. ETH No. 16766

Maintaining Consistency in Collaboration over Hierarchical Documents

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences

presented by

Claudia-Lavinia Ignat

Diploma Engineer in Computer Science,
Technical University of Cluj-Napoca, Romania
born November 29, 1976
citizen of Romania

accepted on the recommendation of

Prof. Dr. M. C. Norrie, examiner
Prof. Dr. P. Molli, co-examiner

2006

To my parents and my brother

Abstract

Collaboration is a key requirement of teams of individuals working together towards some common goal. Computer-supported collaboration is an increasingly common occurrence, driven by the evolving global nature of business, science and engineering and enabled by improvements in computing and communication technologies. Collaborative editing systems have been developed to support a group of people editing a document collaboratively over a computer network. Since not all user groups have the same conventions and not all tasks have the same requirements, it is important to support collaboration for various types of documents. Moreover, since a task has a set of development stages that require various forms of activity, customisation of the collaborative environment should be offered to support different modes of working between different sub-communities of users at different points in time.

The goal of this thesis was to investigate different settings for collaboration over the most common types of documents, such as textual, graphical and XML, with the aim of building a general theoretical framework to support the development of a range of collaborative editors.

A key issue for a general framework for collaboration is a common model that abstracts a large class of documents. The hierarchical model encompasses a wide range of documents and offers support for semantically structured documents. XML documents conform to a hierarchical structure by definition. We modeled text documents using a tree, where the document consists of a sequence of paragraphs, each paragraph of a sequence of sentences, each sentence as a sequence of words and each word as a sequence of characters. Graphical documents are also modeled by a hierarchical structure: groups are represented as internal nodes, while simple objects are represented as leaves. In order to work in a uniform way with different semantic units of the document, we adopted a multi-level editing approach where we associated with an element of the document the editing operations targeting that element.

Operational transformation is a suitable mechanism for maintaining

consistency over copies of shared objects subject to collaboration. We extended the operational transformation approach to work for hierarchical models of documents that have the operations distributed throughout the tree. Our approach for maintaining consistency allows any existing operational transformation algorithm for linear structures to be applied recursively over the hierarchical structure of the document.

The document model enables flexible granularity for the propagation of changes over the network, for the detection and resolution of conflicts and for details about user activity on different parts of the document. For instance, for text documents, the granularity can be dynamically varied at the level of paragraphs, sentences, words or characters, and, for XML documents, at the level of elements, attributes, word nodes or characters.

We first show how the multi-level editing approach was applied to both XML and textual documents. We then show that different mechanisms for maintaining consistency are required for graphical documents.

A novel operation serialisation mechanism was used for consistency maintenance in the case of graphical editing. Conflict handling can be customised to suit the requirements of specific applications. We classified conflicts into real and resolvable, depending on whether an execution order between pairs of operations can be established or not. We offer users the possibility to define the types of conflicts between the operations and the policy for the resolution of conflicts. Our approach for graphical documents is the first one that deals with complex operations such as grouping and working with layers.

In addition to supporting different types of documents, the aim was to also support different modes of communication, such as real-time and asynchronous. Real-time collaboration implies that changes performed by users are seen immediately by other users and asynchronous collaboration over a repository implies that users work in isolation and synchronise their changes against the repository at a later time. We applied the same operational transformation mechanism for maintaining consistency over text and XML documents, for both real-time and asynchronous modes of collaboration. For maintaining consistency over graphical documents, we applied the same serialisation mechanism for both real-time and asynchronous modes of collaboration.

As a proof of concept of the theoretical framework described in the thesis, we built collaborative editors for text, XML and graphical documents, both for real-time and asynchronous collaboration relying on a shared repository.

Zusammenfassung

Kollaboration ist eine Hauptanforderung, um ein Team von Individuen, welche ein gemeinsames Ziel verfolgen, zu unterstützen. Angetrieben durch die zunehmende Globalisierung von Wirtschaft, Wissenschaft und des Ingenieurwesens, unterstützt durch Verbesserungen im EDV und Kommunikationsbereich, nimmt die Zahl der Applikationen für computerunterstütztes kooperatives Arbeiten stark zu. Kollaborative Editoren wurden entwickelt, um es einer Gruppe von Benutzern zu erlauben, Dokumente gemeinsam über ein Rechnernetz zu editieren. Dabei ist es wichtig, verschiedene Arten von Dokumenten zu unterstützen, da nicht alle Benutzergruppen nach den gleichen Richtlinien arbeiten und unterschiedliche Aufgaben verschiedene Anforderungen an ein kollaboratives System stellen. Weil eine Aufgabe in eine Menge von Unteraufgaben aufgeteilt werden kann, welche verschiedene Aktivitäten voraussetzen, ist es entscheidend, dass die kollaborative Umgebung an die unterschiedlichen Arbeitsweisen bestimmter Gruppen von Benutzern während bestimmter Phasen angepasst werden kann.

Das Ziel dieser Dissertation war es, verschiedene Szenarien für die Kollaboration mit verbreiteten Dokumenttypen wie Text-, Graphik- und XML-Dokumenten zu untersuchen und daraus einen allgemeinen theoretischen Rahmen abzuleiten, der die Entwicklung einer Reihe kollaborativer Editoren zu unterstützen vermag.

Eine Voraussetzung für ein allgemeines Kollaborationssystem ist ein allgemeingültiges Modell, welches eine möglichst grosse Klasse von Dokumenttypen abstrahiert. Ein hierarchisches Modell umfasst zahlreiche Dokumenttypen und unterstützt semantisch strukturierte Dokumente. XML Dokumente sind nur ein Beispiel von Dokumenten, welche per Definition hierarchisch strukturiert sind. Wir modellieren Textdokumente als Baumstruktur, wobei jedes Dokument aus einer Reihe von Paragraphen besteht. Ein Paragraph wiederum enthält eine Folge von Sätzen, welche ihrerseits aus einer Sequenz von Wörtern zusammengesetzt sind. Die einzelnen Wörter schliesslich bestehen aus einer Folge von Buchstaben. Graphi-

sche Dokumente werden ebenfalls mit Hilfe einer hierarchischen Baumstruktur modelliert, wobei Gruppen von Objekten als interne Knoten und einzelne Objekte als Blattknoten repräsentiert werden. Um die unterschiedlichen semantischen Einheiten eines Dokumentes einheitlich behandeln zu können, wenden wir einen mehrstufigen Editieransatz an, bei welchem ein Element des Dokumentes mit den dazugehörigen Editieroperationen verknüpft wird.

Das "Operational Transformation" Verfahren eignet sich, um die Konsistenz verschiedener Kopien eines verteilten Objektes zu gewährleisten. Wir haben dieses Verfahren dahingehend erweitert, dass es auch für hierarchische Dokumentenmodelle, bei denen die Operation über den ganzen Baum verteilt sind, eingesetzt werden kann. Unser Ansatz zur Konsistenzerhaltung wendet beliebige existierende Operational Transformation Algorithmen für lineare Strukturen rekursiv auf die hierarchische Struktur eines Dokumentes an.

Unser hierarchisches Dokumentenmodell ermöglicht eine flexible Granularität bei der Übertragung von Änderungen über ein Netzwerk. Die gleiche Flexibilität besteht bei der Erkennung und Auflösung von Konflikten, sowie beim Überwachen von Benutzeraktivitäten in unterschiedlichen Teilen eines Dokumentes. Für Textdokumente kann die Granularität auf dem Level von Paragraphen, Sätzen, Wörtern oder Buchstaben dynamisch variiert werden, während dies bei XML-Dokumenten auf dem Niveau von Elementen, Attributen, Wörtern oder Buchstaben möglich ist.

Wir zeigen zuerst, wie der mehrstufige Editieransatz für XML- und Textdokumente verwendet wurde. Anschliessend erörtern wir, wie sich die Mechanismen zur Konsistenzerhaltung in graphischen Dokumenten von denjenigen für Text- und XML-Dokumente unterscheiden.

Ein neuartiger Ansatz zur Serialisierung von Operationen wurde für die Konsistenzerhaltung im Falle eine graphischen Editors entwickelt. Dabei kann die Konfliktbehandlung entsprechend den Anforderungen einer spezifischen Applikation angepasst werden. Wir unterscheiden zwischen wirklichen und auflösbaren Konflikten, je nachdem ob eine Ausführungsreihenfolge für Paare von Operationen gefunden werden kann oder nicht. Benutzer haben die Möglichkeit, die Konflikttypen zwischen einzelnen Operationen und die entsprechenden Methoden zur Auflösung dieser Konflikte zu definieren. Unsere Lösung zum Editieren von graphischen Dokumenten ist die erste, die sich mit komplexen Operationen wie dem Gruppieren von Objekten und der Verwaltung von Layern beschäftigt.

Zusätzlich zur Unterstützung unterschiedlicher Dokumenttypen war es das Ziel dieser Dissertation verschiedene Kommunikationsarten, wie syn-

chrone (Echtzeit) und asynchrone Kommunikation zu unterstützen. Echtzeitkollaboration impliziert, dass die Änderungen eines jeden Benutzers sofort für andere Benutzer sichtbar sind, während die Benutzer bei asynchroner Kollaboration isoliert arbeiten und ihre Änderungen zu einem späteren Zeitpunkt mit den Daten eines Repositories abgleichen. Wir haben den gleichen Operational Transformation Mechanismus zur Konsistenzerhaltung in Text- und XML-Dokumenten sowohl für Echtzeit- als auch für asynchrone Kollaboration verwendet. Für die Konsistenzerhaltung bei graphischen Dokumenten kommt bei Echtzeit- und asynchrone Kollaboration ein und der selbe Serialisierungsmechanismus zur Anwendung.

Als Beweis für den theoretischen Rahmen dieser Dissertation haben wir kollaborative Editoren für Text, XML und graphische Dokumente implementiert, die sowohl Echtzeit- als auch asynchrone Kollaboration basierend auf einem gemeinsam benutzen Repository ermöglichen.

Acknowledgements

To begin with, I would like to express my gratitude to my supervisor, Prof. Moira C. Norrie, for the chance she gave me to perform my PhD in her research group and her support during my thesis. Her encouragements and trust in my work motivated me to do my best to obtain good results. She always gave me valuable advices at critical steps in my work. But, most importantly for my future career, she inspired me with the beauties of an academic career, including both teaching and research.

I would like to thank my co-supervisor, Prof. Pascal Molli, for his comments on my thesis and his confidence to offer me a postdoc position to continue my research on collaborative editing.

Many thanks to all members of the Globis group for their interest in my work and the stimulating working activity in the group, as well as for the nice moments we spent together during my PhD studies at ETH Zurich. Moreover, I would like to thank Ela Hunt for proofreading the manuscript and Beat Signer for his help for the german translation of the abstract as well as for the formatting of the thesis.

Special thanks to my colleague and friend Gérald Oster for his valuable help during my last year of PhD. I would like to thank him for reading so carefully my report and for his suggestions of improving various parts of it. I appreciated a lot the discussions we had on our research work and his patience of explaining me his opinion regarding some research approaches.

Last but not least I would like to thank to my parents and my brother who gave me all their support and love during my doctoral studies. Even far away, they were always near me to give me their advices and encourage me in the difficult moments of my doctoral studies. I specially dedicate this thesis to my father for his constant interest and concern on the evolution of my thesis.

Table of Contents

1 Introduction

Collaboration is a key requirement of teams of individuals working together towards some common goal. Computer-supported collaboration is an increasingly common occurrence, driven by the evolving global nature of business, science and engineering, and enabled by improvements in computing and communication technologies. A great part of everyday work is group work and therefore computers should provide support to not only help accomplish our personal tasks but also help us communicate and work with others. Central to collaboration is a shared information space which enables members of a community to develop together individual documents, collections of related documents or, more generally, any form of information materials relevant to their common goal. In spite of this need for collaboration, it is surprising to see how poorly computer systems support group activities. For instance, many documents are created by multiple authors, but there is no commercial tool yet to create such shared documents as easily as one can create a single-author document.

Computer Supported Cooperative Work (CSCW) is a rapidly growing multi-disciplinary field relying on the expertise and collaboration of many specialists of different disciplines, including computer scientists and social scientists, that looks at how people work together and seeks to create new tools to assist these groups.

The multiuser software supporting CSCW systems is known as groupware. Ellis and Gibbs [27] define groupware systems as being computer-based systems that support two or more users engaged in a common task

and that provide an interface to a shared environment.

The most important key disciplines that influence groupware are distributed systems, databases, communications, human-computer interaction, artificial intelligence and social theory. In what follows we will analyze the contributions each of these fields brings to groupware.

- **Distributed Systems Perspective**

  A distributed system is defined as a collection of independent computers that appears to its users as a single coherent system [111]. This definition emphasizes the fact that, even though users think they are dealing with a single system, the parts of the system are to some degree autonomous. Consequently, the distributed systems perspective explores and emphasizes the decentralisation of data and control. The distributed systems synchronisation approaches and the algorithms for consistency maintenance and replication have a lot of applications in groupware systems.

- **Databases Perspective**

  The groupware controlling the shared workspace is usually replicated at each participant's site where each site's software is kept synchronised with its counterparts by means of messages. Management of conflicts and concurrency control mechanisms used in distributed databases have inspired the development of consistency maintenance approaches in groupware.

- **Communication Perspective**

  This perspective emphasizes the exchange of information between remote agents. Primary concerns include increasing connectivity and bandwidth, and protocols for the exchange of many types of information such as text, graphics, voice and video.

- **Human-Computer Interaction Perspective**

  This perspective emphasizes the importance of the user interface in computer systems. Human-computer interaction is a multidisciplinary field, relying on diverse skills of graphics designers, computer graphics experts (who study display technologies, input devices and interaction techniques), and cognitive scientists (who study cognitive, perceptual and motor skills).

- **Artificial Intelligence Perspective**

This perspective seeks to develop techniques and technologies for developing machines with human-like attributes. The artificial intelligence approach is usually heuristic and augmentative, allowing information to be accumulated through user-machine interaction rather than being initially complete and structured. This approach suits groupware requirements. For example, groupware designed for use by different groups must be flexible and accommodate a variety of team behaviours and tasks: research suggests that different teams performing the same task use group technologies in different ways.

- **Social Theory Perspective**

  This perspective emphasizes social theory in the design of groupware systems. Awareness and coaching of users play an important part in the social theory applied to groupware applications.

The CSCW tools must be distributed and interactive. They also have to be responsive, i.e. the response time has to be as short as possible so as not to disturb the group activity. In addition they must be fault-tolerant and robust, i.e. the system should be able to recover from unusual circumstances such as component failures and unpredictable user actions. Moreover, the CSCW tools have to be independent from network protocols, operating systems and GUI platforms and provide a mechanism for authentication. In addition to these technical requirements, the groupware has to consider the human factor. Beside design and psychology methodologies, the usability issues involve social science approaches that analyse how people work together and how an organisation imposes and/or adapts to the work practices of its workers.

Groupware has been devised to support a face-to-face group or a group that is distributed over many locations. Moreover, groupware can support collaboration within both real-time interactions and asynchronous, non-real time interaction. The groupware time space matrix [27] is presented in Figure 1.1. The meeting room is an example of face-to-face interaction, i.e. interaction that takes place at the same place and at the same time, while a bulletin board is an example of asynchronous interaction that takes place at the same place but at different times. An example of groupware belonging to the synchronous distributed interaction is a group editor or a video conference system that allows real-time collaboration. An email system belongs to asynchronous distributed interaction.

Within the CSCW field, collaborative editing systems have been developed to support a group of people editing a document collaboratively over

	Synchronous/ Same Time	Asynchronous/ Different Times
Co-present/ Same Place	*Meeting Room*	*Bulletin Board*
Distributed/ Different Remote Places	*Group editor* *Video conference*	*Email*

Figure 1.1: Groupware Time Space Matrix

a computer network as shown in 1.2. The common edited documents can be of any type, such as textual, graphical or XML documents.

Figure 1.2: Collaborative Editing

These systems can be used in a wide range of advanced computing application areas, including collaborative writing, collaborative CAD (Computer Aided Design) and CASE (Computer Aided Software Engineering) and collaborative editing of music scores [9]. The major benefits of collaborative editing include reduced task completion time and distributed collaboration. On the other hand, the challenges that it raises are many, ranging from the technical challenges of maintaining consistency coupled with good performance to the social challenges of supporting group activities and conventions across many different communities.

Collaborative editing systems have been classified as being 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, working in parallel and afterwards synchronising their copies to reestablish a common view of the data.

In order to provide interactive response times for editing, real-time collaborative editors usually use a fully replicated architecture in which the document state is replicated at each site. Concurrency control techniques are required to ensure that a document's state in a replicated architecture remains consistent even when users attempt to modify the document simultaneously in a group editing environment. Moreover, concurrency control should ensure the consistency of the resulting state of the document with respect to the intentions of the users.

In the case of asynchronous systems, merging tools should also ensure consistency and preservation of user intentions, responding appropriately to conflicting changes.

1.1 Motivation

For a community of users, support for collaboration should be offered for various types of documents and for different modes of collaboration for the different stages of the development of a common task. For instance, in the case of collaborative architectural and product design, support for collaboration needs to be offered for text, graphical and XML documents. Graphical documents are needed for brainstorming sessions consisting of a graphical sketching of the issues to be discussed, the assignments of the tasks, as well as for carrying out the product and architectural design itself. Text documents are needed for collaborative writing of the documentation. Adding structure to documents is an activity that facilitates operations on documents. Structured documents such as XML are increasingly being used to store all kinds of information, including not only application data, but also all forms of metadata, specifications, configurations, templates, web documents and even code. Usually, in product design and architectural design, XML documents are used for publishing the design itself in order to make design and manufacturing information available to other applications such as those used by procurement, costing and production departments. The support for the two modes of collaboration, i.e. synchronous and asynchronous, and the possibility of switching from one mode of collaboration to the other, corresponding to different stages of a

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

In order to support concurrent work, users work on their copies of the document. One of the major problems is to maintain consistency of the copies of the shared document. Merging based on operations has been proven to be a suitable approach both for real-time [26, 88, 108, 101] and asynchronous communication [66, 94] and its advantages compared to state-based approaches are explained in what follows. State-based merging uses only the information about the states of the documents and no information about the evolution of one state into another is used. An operation-based merging approach keeps information about the evolution of one document state into another in a buffer containing a history of the operations performed between the two states of the document. Merging is done by executing the operations performed on a copy of the document onto the other copy of the document to be merged. In contrast to the state-based approach, the operation-based approach does not require documents to be transferred over the network between the local workspaces and the repository. Moreover, no complex differentiation algorithms for XML [119, 113, 18, 29] or diff [72] for text have to be applied in order to compute the delta between the documents. Therefore, the responsiveness of the system is better in the operation-based approach. Merging based on operations also offers better support for conflict resolution by offering the possibility of tracking user operations. In the case of operation-based merging, when a conflict occurs, the operation causing the conflict is presented in the context in which it was originally performed. In the state-based merging approach, the conflicts are presented in the order in which they occur within the final structure of the object. For instance, CVS [12] and Subversion [19] systems present the conflicts in the line order of the final document.

Optimistic approaches [89] for operation-based merging allow the modi-

fications to be executed as soon as they are generated and later they might be undone and redone in a serial order or in an order equivalent to the serial order on each copy of the document. The operation transformation approach has been identified as an appropriate optimistic approach to be used for a replicated architecture of a collaborative editing system for maintaining the consistency of the copies of the shared document. It allows a local operation to be executed immediately after its generation and a remote operation needs to be transformed against the executed operations. The specific transformation is dependent on the operation type and on the log of operations already performed. For instance, suppose that a remote operation O inserts a word in a sentence by specifying the position of insertion of the word in the sentence and the new word to be inserted. Suppose that the log contains some operations that insert other words at the beginning of the sentence. Operation O has to be transformed by shifting its position of insertion to the right, depending on the number of words inserted before the target position of O. The operation transformations are performed in such a manner that intentions of operations are preserved and, at the end, the copies of the documents converge.

Most of the existing collaborative editing approaches based on operation transformations [26, 88, 108, 101] adopt a linear document structure. For instance, text documents are seen as a sequence of characters. Operations target characters and, in the face of concurrent operations, syntactic consistency is achieved by ensuring the execution of all concurrent operations. Consider a shared document that contains the text:*"He like the book."* Assume that a user adds the letter *"s"* at the end of the word *"like"* in order to obtain *"He like**s** the book."* At the same time, another user, inserts the letter *"d"* at the end of the word *"like"* in order to obtain *"He like**d** the book."* The result obtained after the execution of the two concurrent operations is *"He like**sd** the book."* The definition and resolution of conflicts does not take into account the structure of the document, such as paragraphs, sentences or words. For instance, in the above example, a conflict could have been defined at the word level, such that two operations are conflicting when they refer to the same word. In this way, only one of the two operations in conflict would be executed, according to the conflict policy used. For example, the policy might be that the user with the highest priority can choose which of the two conflicting operations to execute.

Some of the operation transformation approaches for merging have been defined for hierarchical documents such as SGML [21], XML and CRC (Class, Responsibility, Collaboration) documents [69]. Even if the structure

of the documents is hierarchical, the operational transformation approach is similar to the approach for linear structures and it does not take advantage of the tree structure of the document. The existing operation-based approaches maintain a single history buffer where the executed operations are kept. Operations are not associated with the structure of the document and therefore it is difficult to select which operations refer to which node in the document. This fact has limitations for the definition and resolution of conflicts. The above mentioned approaches adopt only automatic resolution of conflicts where the effect of all operations is maintained and they do not allow for flexible definition and resolution of conflicts. For instance, they do not allow the possibility of defining that any operations that refer to the same node are conflicting and the user can later choose one of the versions of the node. To determine which operations from the history buffer refer to which node is very complex, since the structure of the document is dynamically changed with the execution of each operation.

Multi-level editing involves logging edit operations that refer to each node. In this way, conflicting operations that refer to the same subtree of the document are easily detected by the analysis of the histories associated with the nodes belonging to the subtree. Therefore, the resolution of conflicts is simplified in comparison to the approach using a single history buffer. Moreover, conflict levels can be dynamically varied and conflict units can be presented in the context in which they occurred or at a higher level. For instance, if conflict was defined at the level of an element, meaning that two operations changing that element are in conflict, the conflict can be presented at the level of the element or at the level of one of the ancestor elements.

By using multi-level editing, support for concurrency is increased. Two operations are considered in conflict only if they target a common node in the tree. In the approaches where operations are kept in a single buffer, when a new operation has to be integrated into the history buffer, the entire history has to be scanned and transformations need to be performed even though changes refer to completely different elements in the document and do not interfere with each other. In the multi-level editing approach, the number of transformations that have to be performed is significantly reduced as operations belonging to two nodes that are on different branches of the tree are commutative and they do not need transformations.

In the preceding we have demonstrated that there is a need to collaboratively edit various classes of documents such as text, XML and graphical documents, under various modes of collaboration, both real-time and asynchronous. As previously mentioned, current approaches do not offer

solutions for a flexible and efficient way of document merging.

1.2 Contribution of this thesis

In this thesis we review existing approaches for consistency maintenance in collaborative editing, both for the synchronous and asynchronous modes of collaboration and present an improved solution to consistency maintenance over hierarchical documents.

We first summarize the contributions of this thesis and afterwards present them in detail.

The main contributions of this thesis are as follows:

- We propose a consistency maintenance approach for hierarchical documents, based on an operational transformation approach recursively applied on a multi-level history buffer associated with the document. The approach allows us to define and resolve conflicts by using different semantic units corresponding to the document level and is more efficient than approaches that use a single history buffer. We analyse how our approach can be applied to various classes of documents – text, XML and graphical formats. We show that separate mechanisms are required to maintain consistency for text and XML documents, which use one mechanism, and for graphical documents, which require a new solution.

- Our approaches to consistency maintenance have been applied to both synchronous and asynchronous modes of collaboration. We analyse particular issues for maintaining consistency for both real-time communication and for asynchronous communication relying on a central repository, and we discuss collaborative editors supporting each of these communication modes.

- We discuss the issues arising in the implementation of editors that support the features found in single-user systems, such as grouping/ungrouping operations for graphical editing or auto-completion of elements for XML editing.

In what follows we present in detail the above contributions.

In this work, we propose a multi-level editing approach for maintaining consistency over documents with a complex structure. By using a structured model for the representation of documents, support for collaboration

is offered for a large class of documents. We have analysed collaboration for text, XML and graphical documents. We model the text document as being composed of a list of paragraphs, each paragraph containing a list of sentences, each sentence a list of words and each word a list of characters. XML documents conform to a hierarchical structure by definition. The composition of objects in an object-based graphical document was modelled by using a tree. Groups are represented as internal nodes, while simple objects are represented as leaves. A group can contain other groups or simple objects.

The hierarchical representation of documents allows the possibility of defining and resolving conflicts by using different semantic units corresponding to the document levels, such as paragraph, sentence, word or character, in the case of text documents, or elements, in the case of XML documents. Our approach achieves a higher efficiency than existing merging algorithms that maintain a single history buffer where the user operations are kept. In previous approaches, when a remote operation had to be integrated into the history buffer, the whole log of operations had to be scanned and transformations had to be performed. In our approach we keep the history distributed throughout the tree and when a remote operation has to be integrated, only those logs that are distributed along a certain path in the tree are scanned and transformations performed. Our approach applies an existing operational transformation algorithm for linear structures recursively over all the document levels.

We show how the multi-level editing approach is applied to both XML and text documents. Although the conceptual representation of textual and XML documents on the one hand, and graphical documents on the other, is the same, different mechanisms for maintaining consistency have been proposed for the various classes of documents. For consistency maintenance over text and XML documents, we have used the operational transformation approach whereas, for consistency maintenance over graphical documents, we have used a serialisation mechanism. In the case of text editing, each semantic unit (paragraph, sentence, word and character) can be uniquely identified by its position in the sequence of the child elements of its parent. Insertion and deletion operations on these elements may shift the positions of the sibling elements. Consistency maintenance in this model of representation requires an algorithm to adapt the positions of the elements in the face of concurrent operations. However, in the case of graphical documents, objects are not organised into sequences and identified by their position in the sequence. Rather, they are identified by unique identifiers and there is no need to adapt the identifiers due to concurrent

operations.

The novel serialisation mechanism that we used for consistency maintenance in graphical editing is based on the serialisation of operations relying on the reordering of nodes in a graph. The nodes of the graph represent user operations and the edges of the graph represent ordering constraints between these operations. We classified conflicts into real and resolvable, depending on whether an ordering of execution between pairs of operations can be established or not. We allow users to define the types of conflicts between the operations and the policy for the resolution of conflicts. In this way, conflict handling can be customised to suit the requirements of specific applications. Our approach is the first one that deals with complex operations, such as grouping and working with layers, in the collaborative environment.

Although the approach that we used for maintaining consistency for textual and XML documents is different from the approach adopted for graphical documents, the same mechanisms for consistency maintenance have been used for the synchronous and asynchronous communication over a certain class of documents [40, 41, 46].

We believe that it is important to consider all aspects of collaborative editing together, inclusive of theoretical foundations, technical aspects of implementation and issues of user interaction. Therefore, the theoretical ideas studied in this thesis have been integrated into collaborative editing applications. We built collaborative editors for text, XML and graphical documents, both for real-time and asynchronous collaboration, relying on a shared repository.

In the approaches that we propose we have taken into account some of the functionalities lacking in existing collaborative editing systems and tried to provide them in our systems. XML has recently become a popular format for marking up various kinds of data from web content to application data. Various tools for editing XML documents are available on the market, such as XMLSpy [6] or the XML editor from Stylus Studio [4]. Generally, XML editors provide the user with the possibility of editing XML documents from a graphical interface that visually presents the structure of the document or to textually edit the document. Textual editing is enhanced with auto-completion of elements being edited. We have encountered only a small number of collaborative XML editing tools and most of them have been developed only for research purposes to investigate specific problems. No real-time collaborative XML editor supports the editing of XML documents in the same manner as provided by the single user tools. The SAMS (Synchronous, Asynchronous and Multisynchronous System) [69] editor of-

fers the users a graphical interface to perform the operations of creation and deletion of elements and attributes and of attribute modification. By using the graphical interface, the user is not allowed to customise the element formats, such as the use of separators between the elements, as an implicit formatting of the nodes has to be used. Where a node has to be modified, the node has to be deleted first and a new node with the modified value has to be inserted. In our approach, we offer users the possibility of editing XML documents by using a text interface. We added some logic to the editor to ensure well-formed documents, such as the auto-completion of the elements or the consistency between the begin and close tags of an element, in the same way as support is offered to users in existing single-user XML editors.

Concerning the graphical editor application, our approach is the first collaborative application that deals with grouping operations applied to objects. We support the operations of grouping and ungrouping of objects, which is possible in the hierarchical structure of the graphical document that we adopted. Moreover, there is no collaborative system that allows asynchronous communication over graphical documents. Our graphical editor applications satisfy the requirements of working on collaborative architectural and product design. The requirements have been provided to us by reserachers from the Institute of Machine Tools and Manufacturing at ETH Zurich with whom we collaborate.

Our text and XML collaborative editors are the first editors that support multi-granularity in the definition and resolution of conflicts.

1.3 Structure of this thesis

This section presents the structure of this thesis by giving an overview of the content of each chapter.

Chapter 2 is a background chapter on collaborative editing with a focus on maintaining consistency over the copies of the documents subject to collaboration. We present an overview of existing approaches for consistency maintenance in both real-time and asynchronous editing systems, for text and XML documents, as well as for graphical documents. We classify the existing approaches into pessimistic and optimistic. From the family of pessimistic approaches we present the turn-taking protocols, non-optimistic locking and access control protocols. We classify optimistic approaches into social protocols, optimistic locking, validation techniques, approaches that require human intervention, serialisation, multi-versioning, reconciliation

mechanisms based on merging of states, reconciliation based on constraints, and operational transformation mechanisms. Due to the fact that our approach is based on operational transformation mechanism, we introduce the basic notions of optimistic replication based on the operational transformation mechanism and present in more detail the existing operational transformation approaches.

In Chapter 3 we present our multi-level editing approach for consistency maintenance over hierarchical structured documents. We present the model of document and of the operations exchanged during the collaboration and our general principles for consistency maintenance. We describe our treeOPT approach for maintaining consistency over documents conforming to a hierarchical structure, such as text and XML documents. We show how our multi-level editing mechanism offers support for a flexible definition and resolution of conflicts and a higher efficiency compared to other approaches. The treeOPT approach is based on the recursive application of an existing linear-based operational transformation algorithm over the different document levels. We also present the adaptation of the treeOPT approach for the asynchronous communication over a a shared repository. We conclude the chapter with a related work section where we compare our work with other approaches for consistency maintenance.

In Chapter 4 we present some implementation issues that we faced in the construction of collaborative editors relying on the treeOPT approach. In the first part of the chapter we describe our real-time collaborative text editor application relying on the treeOPT approach. In the second part of the chapter we present our asynchronous text editor application with a shared repository relying on the asyncTreeOPT algorithm.

Chapter 5 describes how the same treeOPT approach presented in Chapter 3 was applied for maintaining consistency in the case of the collaboration over XML documents. We present particular issues arising in the editing of XML documents, in comparison with simple text editing, such as ensuring well-formedness and the auto-completion of elements. We describe two approaches to the consistency over XML documents. In the first approach adopted in our asynchronous XML editor, we defined various types of nodes such as elements, attributes, words and separators and of operations that target these nodes in order to allow the specification of rules for the definition and resolution of conflicts. In our second approach, adopted for the real-time XML editor, we wanted to show the generality of the treeOPT algorithm, regarding its application for text and XML documents and build an editor that works both for text and XML documents and, therefore, we did not distinguish between different types of nodes. We

end the chapter by presenting a related work section where we compare our work with other existing systems for XML document merging.

Although the model for representation of textual and XML documents is the same as for graphical documents, we adopted different mechanisms for maintaining consistency for various classes of documents. In Chapter 6 we present the representation of the objects and of the operations in object-based graphical editing and show why the operational transformation approach could not be used to maintain consistency of graphical documents. We then describe our novel mechanism based on the serialisation of operations. We mapped the consistency maintenance problem in object-based graphical editing to the problem of node reordering in a graph, where the nodes of the graph represent operations and the edges represent ordering constraints between operations. We present the way conflicts are defined and resolved and the implementation of the real-time graphical editing application. We then describe how the serialisation approach was applied to the asynchronous communication based on a shared repository. We present an alternative approach to the merging of object-based graphical documents, based on the document states and compare the two approaches. We also compare the operational transformation approach with the serialisation approach. We end the chapter by comparing our serialisation mechanism with other approaches for maintaining consistency in object-based graphical editing.

Finally, in Chapter 7 we summarise the outcomes of this thesis and provide some future work directions.

2
Background

One of the important tasks of this thesis was to analyse existing collaborative editing tools and their approaches to consistency maintenance. Often researchers have looked at different approaches to concurrency control for both real-time and asynchronous communication and did not analyse these two aspects together. Thus, specific approaches have tended to address the challenges posed by only one form of document, for example, textual, graphical or XML and only one form of communication - synchronous or asynchronous.

We studied the forms of collaboration for each category of documents and the issues involved in both real-time and asynchronous communication, in order to find a document model that abstracts a large class of documents and approaches that efficiently maintain consistency. One of our main goals was to enable customisation. We want to flexibly define conflicts at different levels of granularity, and support different modes of collaboration, either synchronous or asynchronous. In the next chapters we are going to present the model that we adopted for the representation of documents and the issues of collaboration for each class of documents. For each category of documents - textual, XML and graphical documents - we are going to present the techniques for maintaining consistency and details regarding the collaborative tools that we built and studied for that specific class of documents. For each category of documents we applied the same techniques for consistency maintenance to both real-time and asynchronous collaboration.

Groupware allows a group of people to work together at the same time over a computer network. Groupware systems have a shared workspace where documents are stored. The groupware that controls the workspace is replicated at each participant site and the software at each site is kept synchronised with the other sites by means of control message exchange. Due to concurrent operations, inconsistencies can occur and, therefore, concurrency control is the key to the correct functioning of the groupware.

Due to the fact that concurrency control techniques are generally the same for real-time and asynchronous groupware, in this section we are going to give an overview of the existing approaches for consistency maintenance and specify for each one if it can be applied to the synchronous or/and asynchronous collaboration.

We classify the existing approaches into optimistic and pessimistic. Pessimistic approaches block access to a replica unless it is provably up to date. Optimistic approaches let data be read or written without synchronisation, based on the assumption that problems occur only rarely.

We start in section 2.1 by describing the main issues in collaborative editing and then giving an overview in sections 2.2 and 2.3 of known pessimistic and optimistic approaches, respectively, to consistency maintenance.

2.1 Main issues in collaborative editing

A collaborative editing system is constituted by a set of user sites with each user assigned to a site that communicate over a network via exchanged messages. The shared documents can be of any type, such as text, graphical or XML. In this section we are going to illustrate the challenges in maintaining consistency over the two main classes of documents, text and graphical. The messages exchanged between user sites represent operations that can be performed on the shared documents.

Most of existing collaborative editing approaches model text documents using a linear structure. These approaches usually consider that the operations that can be performed on the model of the document are the following:

- $insert(p, c)$ - inserts character c at position p
- $delete(p)$ - deletes the character at position p

Some of the existing approaches consider that the position of the first character in the document is 0, while other algorithms consider that the

position of the first character is 1. For uniformity reasons, throughout this thesis we are going to present all algorithms by considering that the position of the first character is 1.

Concerning consistency maintenance over object-based graphical documents, most of the approaches model the document as a set of objects that we are going to call scene of objects. The operations that can be performed on the objects are simple operations that modify the properties of individual objects, such as moving, resizing and changing the colour.

The notions of causal ordering relation and concurrent operations are necessary for understanding the different approaches to consistency maintenance, and therefore we are going to present them in what follows.

Definition 2.1.1 Causal ordering relation
Given two operations O_1 and O_2 generated at sites i and j, respectively, O_1 causally precedes O_2, $O_1 \rightarrow O_2$ iff: (1) $i = j$ and the generation of O_1 happened before the generation of O_2; or (2) $i \neq j$ and the execution of O_1 at site j happened before the generation of O_2; or (3) there exists an operation O_3 such that $O_1 \rightarrow O_3$ and $O_3 \rightarrow O_2$.

Definition 2.1.2 Concurrent operations
Two operations O_1 and O_2 are said to be concurrent, $O_1 \| O_2$ iff neither $O_1 \rightarrow O_2$, nor $O_2 \rightarrow O_1$.

In order to illustrate the challenging problems in collaborative editing, we are going to refer to some examples from the collaborative text editing and graphical editing domains.

Consider the text document illustrated in figure 2.1.

The paper discusses the concurency contrl issues. Our algorithm applie a linear merging algorithm

Figure 2.1: Initial text document for examples 2.1.1 and 2.1.2

Example 2.1.1 Initial text document
Consider that two users concurrently edit the document shown in Figure 2.1. Suppose that first user corrects the misspelling of word "concurency" *by adding letter* "r" *on position 31 in the document in order to obtain* "The paper discusses the concur**r**ency contrl issues.". *Further suppose that*

second user corrects the misspelling of word "contrl" *by adding letter* "o" *in order to obtain* "The paper discusses the concurency contr**o**l issues.". *The scenario of this example is illustrated in Figure 2.2.*

In Figure 2.2 each vertical line associated to a site represents the time axis, later times being represented higher than earlier ones.

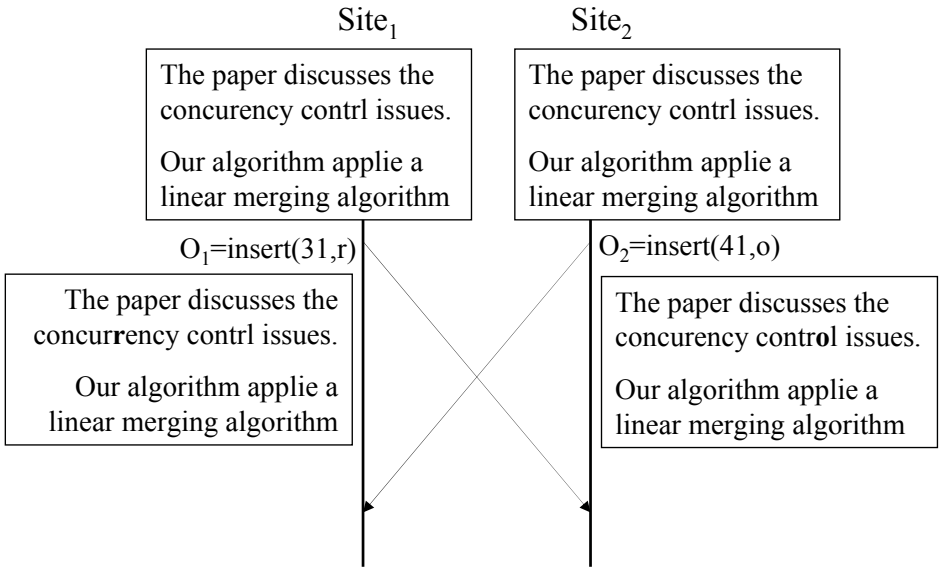


Figure 2.2: Text editing scenario of example 2.1.1

If there are no rules that restrict concurrent changes performed in the same sentence, the expected result after merging the changes of the two users is shown in Figure 2.3.

The paper discusses the concurrency control issues.

Our algorithm applie a linear merging algorithm

Figure 2.3: Final text document obtained in example 2.1.1

If rules are defined specifying that concurrent changes done on the same sentence are conflicting, then according to the resolution policy a decision is taken. For instance, a resolution policy would be to perform none of the concurrent modification. Another possible resolution policy would be to perform only one of the changes.

Example 2.1.2 Text Editing- Example 2
Consider that two users concurrently edit the document illustrated in Figure 2.1. Suppose that first user corrects the misspelling of word "applie"

by adding letter "s" *on position 72 in the document in order to transform the second sentence to* "Our algorithm applies a linear merging algorithm". *Afterwards, the user adds the new sentence* "The approach offers an increased efficiency." *at the end of the second sentence and deletes the word* "an" *from the new inserted sentence. Concurrently, second user corrects the misspelling of word* "applie" *by adding letter* "d" *at the end of the word, inserts the word* "recursively" *and adds the terminator "." in order to transform the second sentence to* "Our algorithm applied recursively a linear merging algorithm.". *The scenario of this example is illustrated in Figure 2.4.*

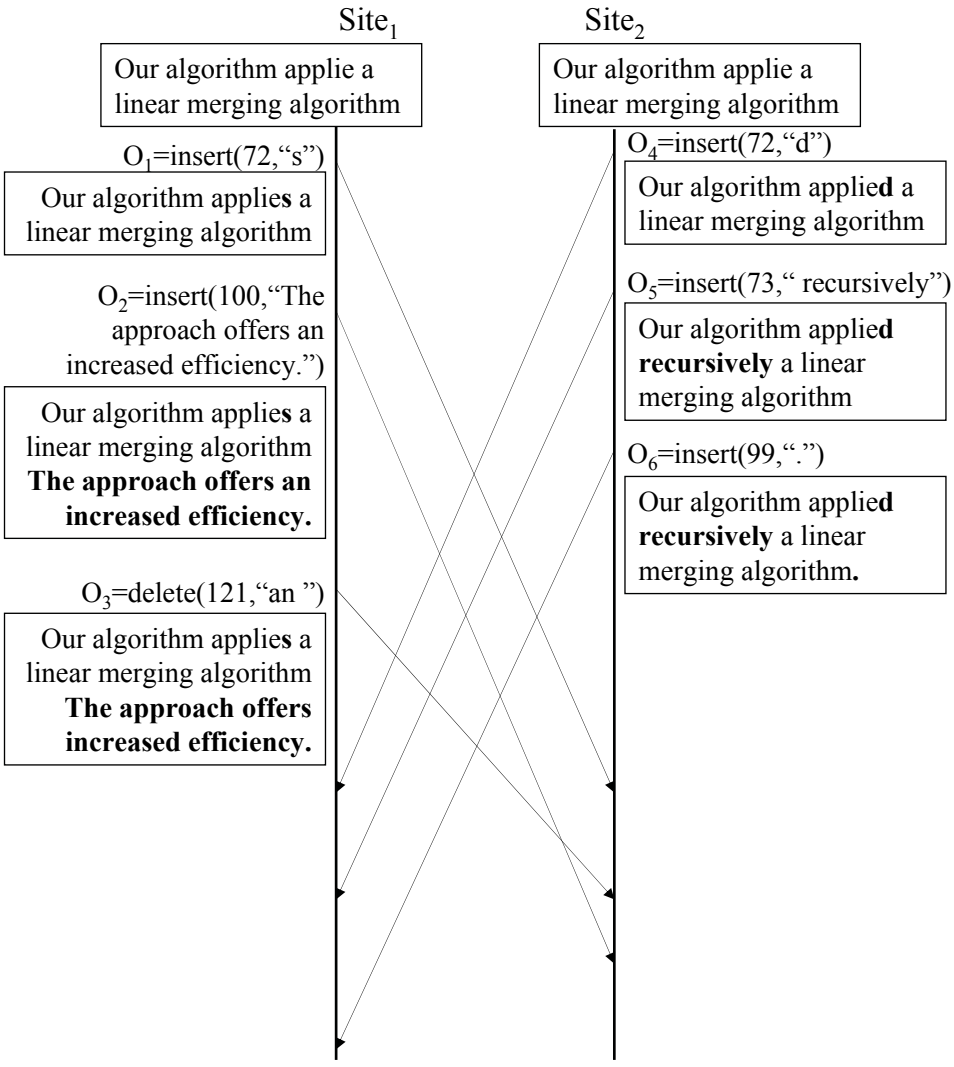


Figure 2.4: Text editing scenario of example 2.1.2

If no rules are defined to specify conflicts, all changes of the two users should be considered, the final result being shown in Figure 2.5.

The paper discusses the concurency contrl issues.

Our algorithm applie**sd recursively** a linear merging algorithm**. The approach offers increased efficiency.**

Figure 2.5: Final text document obtained in example 2.1.2 if no conflicts are defined

Rules could be defined to specify that concurrent changes are restricted to be performed on the same semantic unit such as paragraph, sentence or word. For instance, if no concurrent changes are allowed to be performed on the same paragraph, as all the modifications done by the two users refer to the same paragraph, a rule for the resolution of conflicts should specify that none of the changes should be considered or the changes done by only one user should be performed.

Another example could be the rule that restricts concurrent modifications done on the same sentence. In this case the changes performed by the first user on the initial sentence *"Our algorithm applie a linear merging algorithm"* in order to modify it to *"Our algorithm applie**s** a linear merging algorithm"* are in conflict with the changes done by the second user on the same sentence in order to modify it to *"Our algorithm applie**d recursively** a linear merging algorithm**.**"* According to the adopted policy, either none of the changes should be performed or the changes done by one user should be considered. For instance, if the changes of the second user are chosen, the final document is shown in Figure 2.6.

The paper discusses the concurency contrl issues.

Our algorithm applie**d recursively** a linear merging algorithm**. The approach offers increased efficiency.**

Figure 2.6: Final text document obtained in example 2.1.2 - conflicts are defined if changes are done on the same sentence

In the same way a rule could specify that concurrent changes performed on the same word are in conflict, and in this case the changes performed by the two users on the word *"applie"* would be in conflict. The other concurrent changes are not in conflict. If the resolution policy chooses that the changes performed by the first user are taken into consideration, the document obtained after merging is shown in Figure 2.7.

The paper discusses the concurency contrl issues.

Our algorithm applies **recursively** a linear merging algorithm**. The approach offers increased efficiency.**

Figure 2.7: Final document obtained in example 2.1.2 with conflicts defined if changes are done on the same word

We are going to provide next some examples regarding collaborative graphical editing in order to later show the challenging problems for maintaining consistency. The scene of objects referring to the graphical editing examples is shown in Figure 2.8.

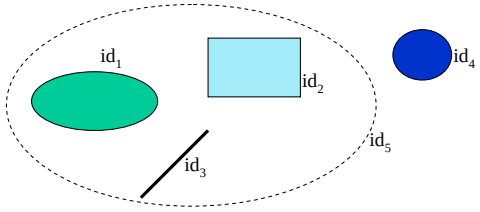


Figure 2.8: Graphical scene of objects for examples 2.1.3 and 2.1.4

Example 2.1.3 Graphical Editing - Example 1

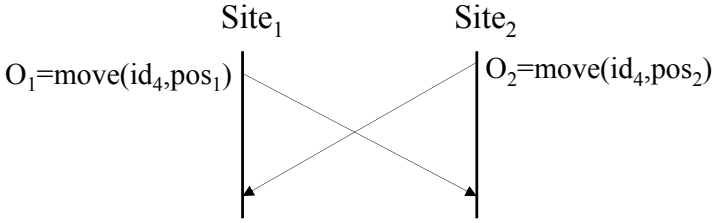


Figure 2.9: Graphical editing scenario of example 2.1.3

Suppose that the first user is moving the object identified by id_4 *to a certain position and, concurrently, the second user moves the same object* id_4 *to a different position.*

The concurrent changes done by the two users are in conflict and some rules should be defined how to merge the two changes. For instance, a conflict resolution rule could specify that only one of the two concurrent move operations can be performed and in this case the object id_4 would be moved to position pos_1 or pos_2 depending on the operation that is chosen to be performed.

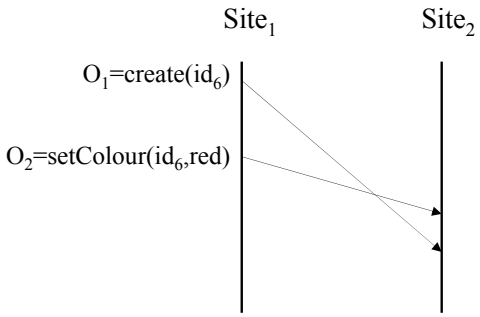


Figure 2.10: Graphical editing scenario of example 2.1.3

Example 2.1.4 Graphical Editing - Example 2
Suppose that the first user at Site_1 *creates a new object and changes its colour to red and that the operations arrive in reverse order at* Site_2.

In this example it is expected that object id_6 is created first and then coloured in red.

The above examples consider that there is no central server for the communication between the clients and that the operations issued by each client are directly transmitted to the others. But, the same scenarios occur if after their generation the messages are sent first to a central server and the server delivers afterwards the messages to the other clients. What is important is that at each site operations arrive in the order indicated in the examples.

The challenging problems in maintaining consistency in collaborative editing are divergence, causality violation and intention violation and we are going to illustrate each of them in what follows.

- **Divergence**

  In example 2.1.1 if we consider that operations are executed at each site in their original form, the first sentence of the example document in Figure 2.1 becomes *"The paper discusses the concur**r**ency cont**o**rl issues."* at $Site_1$ and *"The paper discusses the concur**r**ency contr**o**l issues."* at $Site_2$. The copies of the document at $Site_1$ and $Site_2$ are therefore divergent.

  Divergence occurs also in example 2.1.3 if operations are executed in their original form and in the order of their arrival. At $Site_1$, object id_4 is moved first to the position indicated by $User_1$ and afterwards to the position indicated by $User_2$. At $Site_2$, object id_4 is moved first to the position indicated by $User_2$ and afterwards to the position indicated by $User_1$.

- **Causality violation**

  Due to latencies in the network, operations might arrive at certain sites out of their causal-effect order. For instance, in the example 2.1.2, the first user inserts a sentence and then deletes a word in this sentence, by executing operations O_2 and O_3. If at $Site_2$ operations O_2 and O_3 arrive in the order O_3 followed by O_2, then O_3 refers to a word that does not exist.

  The same problem occurs in example 2.1.4 at $Site_2$, where operation O_2 refers to an object that does not exist.

  Two operations that are in a causal preceding order should be executed in this order at all sites.

- **Intention violation**

  When a user issues an operation, the operation is performed on the current state of the document. At the moment of execution of the operation the current state of the document might have changed and then the execution of operation in its initial form might not satisfy the initial intention of the operation. In section 3.2 we are going to present various ways for the definition of intention. For instance, in the example 2.1.1, operation O_2=*insert(41,"o")* intends to correct the misspelling of word *"contrl"* into *"control"* by inserting the character *"o"* at position 41 in the document. When operation O_2 arrives at $Site_1$, if character *"o"* is inserted at position 41, the intention of O_2 of correcting the misspelling of word *"contrl"* is not anymore satisfied as it is inserted between the characters *"t"* and *"r"*. The resulting sentence at $Site_1$ is therefore *"The paper discusses the concurrency contorl issues."*. This is due to the fact that the concurrent operation O_1 inserts a character to the left of the target insertion position of O_2 and therefore the position of O_2 should be adapted.

In what follows we describe the pessimistic and optimistic approaches for maintaining consistency and analyse how each of the approaches deals with the above mentioned issues: divergence, causality and operation intention.

2.2 Pessimistic approaches

In this section we present the existing pessimistic approaches for merging. Turn-taking and locking approaches are considered to be pessimistic and we

present both of these approaches. However, locking can be also optimistic and the optimistic locking makes the transition from pessimistic approaches to optimistic approaches. Access control is also an approach that restricts access before a user is allowed to edit a replica and therefore, we considered that access control belongs to the family of pessimistic approaches.

2.2.1 Turn-taking protocols

Some shared view systems use turn-taking protocols [32] to allow only one active participant at a time, the one who "has the floor", the other users being blocked from editing. Different types of turn-taking protocols exist:

- The *free floor* turn-taking protocol allows any participant to enter input at any time, with the floor control mediated usually through a voice channel (SHARE [31]; TIMBUKTU [28]). It is possible that multiple input streams are accidentally mixed.

- In the *pre-emptive* turn-taking protocol, a user can take the floor away at any time from the current floor holder (SHARE [31]; CAPTURE LAB [67]). Users may be interrupted by any other meeting participants in the same way that a speaker at a meeting can be interrupted. Unlike the spoken interruption, the interruption of taking the floor away cannot be ignored.

- The *explicit release* turn-taking protocol requires that the floor holder must explicitly release the floor before another participant may claim it (SHARE [31]; CANTATA [16]).

- In the *first in, first out queue with explicit release* turn-taking protocol, the participants line up to take turns, and the floor, once explicitly released by the floor holder, is given to the person at the front of the line (CANTATA [16], VCONF [59]).

- The *central moderator* turn-taking protocol uses a moderator to decide who should hold the floor (RTCAL [90], SHARE [31]).

- In the *pause detection* turn-taking protocol, the floor is released only after the system detects a suitable pause of activity by the floor holder (SHARE [31], EMCE [30])

No policy of turn-taking protocols suffices for all groups in all situations. A small group of users may prefer the free control choosing to mediate interaction by voice channel, while a larger cooperating group may employ

pre-emptive control to avoid accidental input merging. For distance education, a teacher may use a "central moderator" approach to give and take back control from the members of the audience posing questions. But, in a formal meeting context, a group decision support system may enforce a round-robin or queue policy. In some cases there is the need to switch between different policies during different stages of a project. For instance, in the case of users writing code together, the free floor turn-taking protocol is suitable for brainstorming, while the system controlled protocols giving access to one person at a time is needed to ensure that a particular piece of code is correctly coded.

The turn-taking protocols are however limited to situations where allowing a single active user is suitable for the model of collaborative working and are not suited to application environments where the nature of collaboration is characterized by concurrent streams of activities from multiple users.

Turn taking protocols do not allow that concurrent editing operations are performed by more users and therefore the issues of divergence, causality and intention violation do not occur. But, the payoff is that no concurrency is allowed. These protocols are usually used for real-time collaboration.

2.2.2 Non-optimistic locking

Another approach to concurrency control is locking [14, 114]. Locking guarantees that users access objects in the shared workspace one at a time and concurrent editing is allowed only if users lock and edit different objects.

The locking mechanisms can be non-optimistic or optimistic. *Non-optimistic locking* implies that an object can be manipulated only after the lock on the object has been acquired. *Optimistic locking* allows manipulation of objects before locks are granted.

In the case of non-optimistic locking, delays might create an unresponsive interface. The interface has to provide at least feedback showing that the object is waiting for a lock request to be served.

The granularity of locking can vary from system to system. A coarse granularity implies fewer lock requests, but it offers support for less concurrency. On the other hand, fine-grain locking allows better concurrency, but it implies greater locking overhead. In the case of collaborative editing of text documents, the locking mechanism can be applied at the level of sections, paragraphs, sentences, words or characters. For the editing of graphical objects, locking can also be applied at different levels, such as

groups of objects, individual objects or handles on objects, such as the endpoints of a line.

In MACE [75] a pair of locks specify the section of text to be exclusively edited by a user. Any number of users may edit the document within the regions defined by their own locks. A locked region may grow or shrink as its content is edited. The editor and the viewer have joint control over the degree of view sharing. A writer has the right to deny updates to a viewer and a viewer may choose to be updated at certain moments of time or on real-time with the changes performed by a remote user.

In SASSE [8] non-optimistic locking of regions of text has been adopted. A replicated architecture is used and a central communication server ensures that the messages are received in the same order at each collaborating site.

In the object-based drawing system GroupDraw [33], locks are requested on object handles. For instance, in the case of a line, two users can simultaneously grab the endpoints of the line and move them to different positions. However, the approach must ensure that object behaviour is managed consistently between users, so that, for instance, two people grabbing and dragging the same point on a line do not both succeed. The approach assigns an owner to each object and the owners have the task of maintaining consistency as they have the final authority on all operations affecting the object. Suppose that three users are working on a scene of objects. Suppose that a line from the scene of objects is owned by $User_1$ and $User_2$ and $User_3$ simultaneously select the same end point of the line object. Only one person is allowed to select the object and therefore this is a situation where object consistency could be compromised. When an object is manipulated, the process running the application sends a message to the owner of the object to require permission for the manipulation of the object. The processes running the applications of $User_2$ and $User_3$ send a message to the process of $User_1$ requesting control of the end point. The process of $User_1$ assigns permission to the first request. If the request of $User_2$ arrives first, a message granting the permission to grab the point is sent to $User_2$, while $User_3$ receives a message denying permission. When the user releases the point, the owner is notified and the point is available for selection. A distributed locking scheme has been implemented based on the object ownership relation and locking can be automatically performed by the system or it can be explicitly specified by the user. Object ownership is distributed among participants. If a participant leaves the editing session, their objects are transferred to other participants. When the last member leaves the session, ownership does not need to be retained, as it is relevant

only during a single session.

One of the models that has been adopted for maintaining the consistency of the database storing the information managed by the Colab tools [99] is the centralized locking model. The granularity size of locking is variable and can be one lock for the entire database or separate locks for different parts of it. This model yielded unacceptable delays for obtaining the locks, due to the fact that the processes are not prioritized and there is no way to guarantee limits on the delays in the system.

Let us analyse in what follows how locking deals with issues of divergence, causality preservation and intention preservation.

In the example 2.1.1 consider that locking is done by explicitly selecting the part of the document that is to be locked. Consider that at $Site_1$, $User_1$ locks the part of the document consisting of the word *"concurency"* before editing it. Further consider that at $Site_2$, $User_2$ locks the part of the document consisting of the word *"contrl"* before editing it. When operation O_2 arrives at $Site_1$ and is executed, the resulting first sentence of the document would be *"The paper discusses the concurrency control issues."*. At $Site_2$, after the arrival and execution of O_1, the resulting first sentence of the document would be *"The paper discusses the concurrency control issues."*. Therefore, locking on words did not prevent divergence in this example. Locking does not solve the divergence issue unless the granularity of locking is the whole document, which prohibits concurrency in the system.

Causality violation is only related to operation ordering and has nothing to do with whether operations refer to the same region. Therefore, locking used as a stand alone mechanism cannot resolve the causality violation problem.

Intention violation issue is not resolved by using only the locking mechanism, either. For instance, in the example 2.1.1, operation O_2=*insert(41, "o")* intends to insert the character *"o"* at position 41 in the document to correct the misspelling of word *"contrl"* in order to obtain the word *"control"*. Even if the part of the document composing the word *"contrl"* is locked, the intention of O_2 is not preserved when operation O_2 is executed at $Site_1$. If character *"o"* is inserted at position 41, the intention of O_2 is not maintained as *"o"* is inserted between *"t"* and *"r"*, the result sentence being *"The paper discusses the concurrency control issues."*.

As we have seen, locking mechanisms restrict concurrency over the set of shared objects, and a locked object can be accessed only by the person who owns the lock. Non-optimistic locking does not offer good responsiveness in real-time editing and optimistic locking causes confusion when the locks

are denied and the state of the document before the initiation of the locking mechanism has to be reestablished. Moreover, locking used as a separate mechanism does not ensure convergence and the preservation of intentions.

In [103] locking is offered as an optional mechanism that can be explicitly requested by users to restrict the free editing over some parts of the shared document. Users can freely perform modifications in the unlocked parts of the document.

Due to its simplicity, non-optimistic locking may be chosen when the response to a lock request is perceived as instantaneous. But in the case of long delays, non-optimistic locking translates into a poor user interface.

Non-optimistic locking was used not only for consistency maintenance in real-time groupware, but also for asynchronous communication. For instance, early version control systems, such as RCS (Revision Control System) [112], use locking as support for collaboration. When a document is checked out into the local workspace of a user from the repository where the versions of the document are kept, the document is locked until it is committed to the repository by the same user, preventing other users from concurrently changing the same document.

2.2.3 Access control

Access control determines which subjects – users or programmers – are authorised to do what type of operations on which objects. It is a mechanism that protects data objects from unauthorised access. As we have seen, locking prevents concurrent inconsistent changes to the editable data structure of a collaborative application. Access control is similar to locking in what concerns the control access to data structures. However, the difference is that access control prevents unauthorised changes while locking prevents inconsistent authorised changes.

The SUITE framework [96] associates fine-grained data displayed by a collaborative application with a set of collaboration rights that can be specified by the users. The collaboration rights include traditional read and write rights and several new rights such as viewing rights and coupling rights. A multi-dimensional, inheritance-based scheme is used to resolve conflicts.

Due to the fact that access control prevents access before the editing is allowed, the approach is pessimistic. But, taking into account that there is more than one user who can have access to a certain object, the approach can be considered as an optimistic approach, too.

Access control mechanisms have the same disadvantage as locking or

restricting concurrency over the set of shared objects. Access control could be used to replace the locking mechanism, for instance, for a text document, each paragraph can be assigned to be accessed by only one specified user. Similarly to locking approaches, access control could be seen as an additional way of maintaining consistency in an approach that allows freedom of editing for the users. Access control can be both used in both the synchronous and asynchronous modes of communication.

2.3 Optimistic approaches

In this section we present optimistic approaches. We start with the presentation of social protocols for mediation and then go on to describe optimistic locking, validation techniques, approaches that require human intervention, serialisation, multi-versioning, reconciliation mechanisms based on state merging, reconciliation based on constraints and operational transformation mechanisms.

2.3.1 Social protocols for mediation

In some systems concurrency control is not computer mediated. In some groupware systems inconsistencies might not matter or the systems rely on the fact that people can mediate their actions and repair potential conflicts. In the GroupSketch groupware application [34], there is no concurrency control and inconsistencies in the bit-mapped shared area are acceptable to the people involved in the collaboration. For example, consider the case where two users collaboratively work on a bitmap and each user can set or clear pixels belonging to the bitmap. Consider that a user draws a line by setting the set of pixels forming the line and the second user deletes an existing line that intersects the first line by clearing the pixels along that line. The final appearance of the bitmap depends on the order in which the drawing and deleting events arrive at each participant site, i.e. the pixel representing the intersection between the line that is drawn and the line that is deleted can be set or not. User studies performed with GroupSketch have shown that due to the rare occurrence of these situations and the minimal visual disruption to drawing, these inconsistencies are tolerable.

Some groupware systems rely on the fact that people naturally follow social protocols for mediating interactions, such as turn taking in conversations and the ways in which shared objects are used. In the Cognoter system [99], if two participants make simultaneous changes to the same

data, the result will depend on which change takes effect first, therefore the results can be different on different machines. The lack of concurrency control is acceptable as the participants are aware of the possible conflicts and use verbal cues ("voice locks") to coordinate their behaviour. Moreover, the system helps participants to avoid conflicts by the use of busy signals. By greying out screen items that are in use, the signal warns other collaborators not to change them. The busy signals do not make conflict impossible, but make it largely avoidable, by relying on the participants to notice that an item is being changed.

These systems do not address the problems of divergence, causality or intention violation. Social protocols for mediation are generally used for real-time collaboration.

Due to the fact that social protocols do not restrict the access of multiple users to data and do not prevent conflicts from happening, we classified this approach as optimistic. The difference from other optimistic approaches is that social protocols do not solve conflicts after detection.

2.3.2 Optimistic locking

Optimistic locking implies that after requesting a lock on an object, the requester acquires tentative approval and can start manipulating the object before knowing whether the lock has been approved. If the lock is acquired, the work continues as normal. But if the lock is denied, the object that has been manipulated must be returned to its original state. Optimistic locking can be considered a step from pessimistic approaches to optimistic ones.

Optimistic locking can be further classified into fully and semi-optimistic locking, depending on what the user is allowed to do if they have a tentative lock and have finished manipulating the object. In the case of *fully-optimistic locking*, the user can manipulate other objects that might require further lock requests. If the lock is denied, the state of the object for which the lock has been required should be restored. The difficulty comes in the case when other objects have been manipulated based on the tentative state of the object for which the lock has been requested. In this case, the state before the illegal manipulation should be restored. These problems are avoided in the semi-optimistic scheme, where the users are allowed to manipulate the objects with tentative locks, but they are not allowed to manipulate other objects until the lock is approved or denied.

Optimistic locking avoids delays, but it is not clear what to do when locks are denied. For instance, consider the case of text editing, when the

user starts typing a sentence after the lock has been requested. If the lock is denied, the text that has been written during the tentative approval period should disappear, an effect that is not very convenient for the user. In graphical editing, a line that was moved during the tentative approval period should be returned to its initial position.

In fully optimistic locking, the user can manipulate other objects during the tentative approval period. Suppose that a user manipulates a graphical scene of objects containing, among other objects, a rectangle and a line inside the rectangle. The user first moves the rectangle for which they acquire a tentative lock, but afterwards they also move the line. If the lock for manipulating the rectangle is denied, the rectangle should be moved back to its initial position. It is not clear whether the line should also be moved to its initial position. This would make sense only if these actions are dependent. If the line is moved back to its initial position, additional information should be provided to the user that the line was moved to the initial position due to a lock denial and not due to an action of another user. The same situation occurs in text editing, where a user may want to lock a paragraph and, after acquiring the tentative lock on the paragraph, starts editing different sentences belonging to that paragraph. If the lock is denied, the user should be provided with feedback that the multiple changes performed during the tentative approval period in the different parts of the paragraph have been cancelled. The problem is also how to highlight and present to the user the changes that have been cancelled.

In semi-optimistic locking, the user cannot proceed to manipulate other objects till the lock on the current object has been acquired. In text editing, where the locking granularity is at the character level, typing becomes annoying for the user since no operation can be performed before the lock for each character is acquired. In this case, continuity problems for semi-optimistic locking are the same as in the case of non-optimistic locking.

DistEdit [58] provides both automatic locking when the user does any editing and an explicit locking mechanism according to which users deliberately select and lock the region on which work is to be done. The toolkit also provides support for multi-operations, i.e. a set of operations that are treated as a single atomic action. DistEdit decides what locks are required for performing the action and then tries to acquire all the locks. If the lock request succeeds, the multi-operation succeeds and otherwise it fails.

Even though optimistic locking has its problems, it was used in many groupware applications. The reason is that people mediate their actions and conflicts are rare. The disadvantages of optimistic locking are high implementation costs and potential confusions due to denials of lock ac-

quisition. The way divergence, causality and intention issues are treated is the same as for non-optimistic locking.

2.3.3 Validation techniques

In [35] and [23] a centralised database approach to collaborative editing has been proposed. It is the only existing approach where a document is represented as a linked list of character objects that are stored in a database. Each character object has a unique identifier assigned by the database system. Editing commands for the operations of inserting and deleting characters are mapped to database transactions for inserting and deleting character objects and for relinking the character objects preceding directly or following inserted or deleted characters. By representing each character as an object in the database, additional information can be attached to each character. In this way, the layout and formatting of the text of the document, as well as security rights can be easily added. However, this approach assumes a database point of view and when a conflict generates inconsistency in the database, some transactions will be aborted. The concurrency control method used is validation. If the validation phase fails, the operations performed by some users are cancelled, and, therefore, the same disadvantages as in the case of the optimistic locking are present. When a list of characters is inserted or deleted by a transaction, the identifiers of the characters between which the insertion or deletion is performed are sent to the database together with the characters to be inserted, in the case of an insert. For instance, let us analyse the steps performed for the insertion of three characters at once. The identifiers of the previous $ID_{previous}$ and of the next ID_{next} characters between which the insertion has to be performed as well as the characters that have to be inserted are the parameters of the insert operation. Firstly, a new transaction for the character insertions is started. In the validation phase a check is performed whether the character before ID_{next} corresponds to $ID_{previous}$. Three sequential character identifiers are reserved corresponding to the characters that have to be inserted. The characters are inserted and the transaction is closed if no errors occurred. If the validation phase fails, the transaction is aborted.

In the case that two insertions are performed concurrently at the same position of the document, the operation that arrives first at the server where the database is kept will be executed and the other operation will be rejected. The reason for the rejection of the operation is that after the execution of the first operation, the two characters between which the

insertion is performed are no longer adjacent. Therefore, an inconsistency is generated. Inconsistency is generated also in the case that one user inserts a set of characters and a second user concurrently deletes a set of characters including at least one of the characters between which the insertion has been performed.

Let us analyse how this approach deals with the issues of divergence, causality and intention violation.

Convergence is achieved as the approach is centralised and each time a modification is sent by a client to the server and the database is modified, the server sends the operations to the other clients in order to update the local copies of the documents to reflect the central copy of the document.

The causality issue is not discussed in the Tendax [35] approach and the problem that could appear is that two causally ordered operations generated by a client arrive in reverse order at the server. For instance, consider that a client performs the operations illustrated in the example 2.1.2 at $Site_1$ and that the operations arrive in the same order at the server as they arrive at $Site_2$. If operations O_2 of insertion of a sentence and O_3 of deletion of a word from this sentence arrive in different order at the server, then O_3 would specify non existent characters for the delimitation of delete range. In this case, operation O_3 is rejected and operation O_2 is executed when it arrives at the server.

User intentions are not maintained when insertions are performed concurrently at the same position of the document or one user inserts a set of characters and a second user concurrently deletes a set of characters including at least one of the characters between which the insertion has been performed. For instance, consider the example 2.1.2 where one client executes the operations at $Site_1$ and the other client executes the operations at $Site_2$ and the operations are sent to the central server. Let us analyse first the effect of the execution of the pair of concurrent operations O_1 and O_4. The operation that arrives first at the server will be executed and the other one rejected as the characters between which insertion is performed - the last letter of word *"applie"*, i.e. *"e"*, and the space after the word *"applie"* are no longer neighboring characters. Similarly, only the first of the two concurrent operations O_2 and O_6 is executed as after the execution of one of these operations the last letter of word *"algorithm"*, i.e. *"m"* and the end character of the document are no longer neighboring characters.

The approach described in [35] and [23] is a database approach, different than the approach adopted in a collaborative environment. In a collaborative application partial intentions of as many users as possible should be preserved, without restricting users from performing some oper-

ations. Even if the result does not completely conform to the intentions of the users, the users can re-edit the text. Due to the fact that divergence is temporarily allowed, the approach is optimistic. In the systems we analysed the approach was used for real-time communication. Due to the fact that in the case of conflict some operations are aborted and the work of some users is lost, the approach is not very suitable for asynchronous collaboration. In the real-time communication users can quickly react to conflicts and losing of work of some users is acceptable. In the asynchronous communication conflicts may generate the cancellation of a large number of operations performed in parallel by users.

2.3.4 Human intervention

This subsection presents some approaches that require the intervention of users for resolving conflicts.

One of the models proposed for maintaining consistency in the Colab systems [99] is the dependency-detection model. A replicated architecture is used and data is annotated with stamps describing the author and time of change. When data are broadcast, the stamp associated with the new data as well as the stamp associated with the old data are transmitted. Upon receiving a message, a site checks whether the stamp of the previous data in the request is the same as the stamp of the local data. If concurrent changes have been performed on data, the two stamps are different and therefore a dependency conflict is signaled. Human intervention is required in order to solve the conflict. In the case that two participants change the data at the same time, at least one of the machines will detect the dependency conflict. However, if the messages about changes to data from different users arrive out of order, false detections of conflict will be signaled.

The CAMERA system [66] offers a merging approach based on operations for supporting the cooperative developments that are performed on an object-oriented database. The merging approach is flexible and offers the user the possibility of selecting one of the following approaches: to impose an ordering on the operations, to discard an operation involved in a conflict or to edit the result of merging. The merging algorithm takes two sequences of change operations and combines them into a single sequence, detecting both inconsistencies and conflicts. However, the algorithm is complex as a huge search space of potential merged operation sequences must be considered. This is the first proposed approach that uses operation-based merging. However no direct application of the merging algorithm proposed in [66] has been presented for a text-based or graphical collaborative editing

system.

In GINA [11] the merging is performed using command histories. Document versions are represented as branches of the command history. Each branch contains the editing commands that have been executed to modify the document. When a merge is performed, two branches are merged by applying one of the command object branches at the end of the other branch. The commands are automatically accepted when possible, but the user is given the choice of which change to keep when conflicts arise. The user may undo the conflicting operation from one branch and redo the operation from the second branch or simply not redo the operation of the second branch. GINA is shown as supporting generality with respect to the object merged because it is based on command histories. However, the merging is not semantics based and there are no merging policies defined by the user. Therefore, user intervention is required in the case of a conflict. Synchronous and decoupled modes of communication are supported, as well as transitions from one mode to the other.

One of the approaches that used the tree representation of documents as the basic unit for collaboration is the dARB approach [53]. The approach uses a distributed arbitration mechanism for dealing with the consistency of document replicas. Each site maintains a tree structure, each vertex having an associated unique identifier. In order to detect whether the operations overlap, a count is associated with each node showing how many operations have been applied to that node. The possible operations that can be applied are the insertion and deletion of a node and the modification of a node. In the case of the insertion and deletion operations applied locally on a certain node, the count associated with the parent node will be incremented. In the case of the modify operation, the count of the node itself is incremented. After an operation is locally generated, it is propagated to the remote sites. When a remote site receives an operation that overlaps with other executed operations, an arbitration phase is started in order to maintain consistency. The arbitration phase consists of sending special events on a totally ordered channel informing about the fact that some overlapping concurrent operations attempted to access the same vertex at a certain site. Each site that discovers concurrent conflicting operations sends such special events over the totally ordered channel. The site whose event arrives first at the server will be the one winning the arbitration. The site sends afterwards an update message to the other sites specifying the current state of the accessed vertex. The operations are assigned different priorities, for instance the insert/delete operations are assigned greater priorities than the modify operations. There are cases when the winning site

has to send to the other sites not only the state of the vertex, but also the state of the parent or grandparent. The arbitration scheme cannot resolve all concurrent accesses to documents automatically and, in some cases, must resort to asking the users to manually resolve inconsistencies. Rather than obtaining a partial combination of the intentions of the users that issued concurrent operations, the dARB approach preserves the intentions of only one user. A collaborative text editor and a 3D collaborative virtual environment was built relying on the functionality of the dARB algorithm. In the case of the text editor, the document is viewed as consisting of a set of paragraphs, each paragraph being formed of sentences, each sentence consisting of words and each word being composed of characters. However, the communication is done at the character level, i.e. an operation is generated each time a character is inserted. In order to show the generality of the dARB algorithm for maintaining consistency, a 3D collaborative application called cWorld was implemented. The participants can design a shared office space by creating and moving furniture until they reach an agreement. The scene of objects contains simple geometric figures such as boxes, cylinders, cones and spheres and other aggregate furniture objects such as desks, cabinets and bookcases. However, no information is provided on how concurrent operations on groups are performed.

In what follows we analyse how dARB as a representative approach of maintaining consistency based on the intervention of users deals with the issues of divergence, causality and intention violation.

Convergence is ensured by a relaxation of the total order relation between the operations. If operations are executed in the same order at all sites, called total order, convergence is ensured. But, total ordering has the disadvantage of poor responsiveness and the dARB approach is a relaxation of the total ordering approach. If concurrent operations do not overlap, such as two operations targeting different words, due to the fact that the operations are commuting, partial order between operations, i.e. the precedence order, is enough for maintaining consistency. But if concurrent operations overlap, but are not commuting, a totally ordered channel is used for the reordering of operations.

The causality precedence relation is maintained by the use of a reliable multicast protocol that ensures that messages are not lost and that they are delivered in the order of their generation.

However, a partial combined effect of the intentions of all users is not preserved, as illustrated in what follows by using the example 2.1.2. Operations O_1 and O_4 need an arbitration mechanism as they modify the same word. Due to the fact that the first sentence in the paragraph *"Our algo-*

rithm applie a linear merging algorithm" has no punctuation mark at the end of the sentence, the content inserted by operation O_2 is parsed and it is considered as a sequence of word insertions into the first sentence. Therefore, operations O_2, O_5 and O_3 need arbitration as they insert or delete words into the same sentence. As operation O_6 inserts a punctuation mark inside a sentence, a new sentence has to be inserted in the paragraph. An arbitration mechanism between O_6 and O_2 and O_3 is triggered and due to the fact that the operation of insertion of sentence has priority over the operations of modification of sentences, operation O_6 wins the arbitration. The whole paragraph containing the local modifications at $Site_2$, *"Our algorithm applied recursively a linear merging algorithm."* is sent to $Site_1$. Therefore, the intentions of $User_2$ are respected, and the modifications performed by $User_1$ are discarded. The preservation of the intentions of all users is thus not achieved.

Due to the fact that repairing actions are taken after detection of conflict and divergence is temporarily allowed, the approach is optimistic. User intervention is a mechanism used both in real-time and asynchronous communication. However, for the real-time communication the user might get annoyed if he/she is interrupted from working and consulted for every conflict that occurs.

2.3.5 Serialisation

Serialisation is a mechanism for maintaining consistency that executes user operations in a serial order. As with the locking mechanism, serialisation can be classified into optimistic and non-optimistic [14, 114].

Non-optimistic serialisation does not allow operations to be received out of order. An operation can be executed only after all operations preceding it have been executed.

Optimistic serialisation assumes the fact that operations are rarely received out of order. Therefore, operations are executed immediately and, only in the case that an out of order operation arrives at the site, will a certain repairing approach be adopted in order to guarantee the correct ordering. In what follows we discuss some of the existing systems based on the optimistic serialisation of operations.

A solution to repairing is the Time Warp mechanism [54] where the system returns to the state just before the out-of-order operation was received. Intermediate side effects have to be cancelled by sending anti-messages of the operations locally generated out-of-order and then the messages received are executed again, the late message being received in the right

order this time.

Another solution to repairing is the undoing of the operations that follow the remote operation in the serial order, followed by the execution of the remote operation and the re-execution of the undone operations.

The technique of repairing by rolling back the state of the system to just before the out-of-order operation and then redoing the operations in the correct order has been implemented in the GroupDesign [56] groupware that supports real-time collaborative editing of object-based graphical documents. The ORESTE (Optimal RESponse TimE) algorithm underlying the GroupDesign system uses a history buffer logging user operations and a queue with operations received that apply to nonexistent objects. Timestamps are used to define the total order among operations. When a remote operation is received, the operation is checked to see whether it applies to an object that was not yet created. If this is the case, the operation is queued into the list of operations applying to nonexistent objects. Otherwise, the local logical time is compared with the timestamp of the operation. If the time of the received operation is greater than the local time, then the operation is executed immediately. Otherwise the operations from the history buffer that are more recent than the one just received are undone, the new operation is executed and afterwards the undone operations are redone. In order to reduce the number of the undo-redo operations, total ordering of the operations is relaxed to a partial ordering. Operations are allowed to be executed out of order when their effects are the same as if they were executed in order, based on the definition of the relationships of commutativity and masking between the operations. However, the approach does not deal with complex operations such as grouping/ungrouping.

Serialisation mechanisms achieve convergence as operations are executed in total order at all sites, with the exception of the operations that commute or are masked. Concurrent operations are executed in the order of their timestamps and the intentions of users are not taken into account. For instance, in example 2.1.4 if operation O_1 has a lower timestamp than O_2, then O_1 is executed before O_2 and therefore object id_4 is moved to position pos_2. If operation O_1 has a larger timestamp than O_2, object id_4 is moved to position pos_1. Causality order between operations is considered or not by the serialisation approaches analysed in this section. For instance, GroupDesign does not consider causality between operations. Serialisation mechanism can be both used for the real-time and asynchronous modes of collaboration.

2.3.6 Multi-versioning

The multi-versioning approach tries to achieve all operation effects and preserves the intentions of all operations. For each concurrent operation targeting a common object, a new object version is created.

GRACE (GRAphics Collaborative Editing) [105] is an internet based prototype system that uses the multi-versioning approach to maintain the consistency of copies of object-based graphical documents subject to collaboration. It uses state vectors to define a total order among operations and unique identifiers assigned to objects. In the case that a remote operation is causally ready for execution, the collection of objects in the application scope of this operation is selected. Afterwards, the operation is checked against all concurrent operations referring to the objects in its application scope and, in the case of conflict, it creates new object versions.

TIVOLI [70] is a shared drawing system implementing the whiteboard metaphor and designed to support small group meetings, both in a single room and between remotely connected sites. Tivoli adopts an immutable object model, i.e. objects can only be deleted, but they cannot be changed. To change an object (e.g. to move it), it must be deleted and a new object that incorporates the change created. If two operations concurrently target the same object, the object will be deleted and two new objects will be created, respecting the intentions of the operations. Each change operation is represented in the history list as an old-new object pair allowing the undoing of the changes, but consuming a lot of space for saving the editing operations. In contrast to GRACE, where conflict occurs at the object attribute level, in Tivoli, conflict occurs whenever two concurrent operations target the same object.

POLO [123] is a distributed object-based real-time collaborative editor, where a document is modelled by a set of independent objects. Operations of deleting or creating an object, such as a rectangle, ellipse, line or textbox are provided. Each object has a set of attributes such as type, size, position and colour that can be updated, as in GRACE. POLO also uses a multi-versioning mechanism to preserve concurrent conflicting user intentions. The proposed multi-versioning approach is an extension of the approach proposed in GRACE, where the versions targeted by operations are defined by the contextual intention of users.

Multiversion approaches preserve the intentions of all users, as in case of conflict multiple versions of the common targeted object are created. These algorithms also maintain the causality order between operations. Moreover, multiversioning algorithms ensure that the same set of versions is created at

all sites. However, a disadvantage of the multi-versioning technique is that there is no correlation between the different versions of the same object and the base object. It is not clear what interface is suitable for such systems to allow the user to navigate between object versions. And at the end of the editing process, it is not clear how object versions will be merged and therefore how convergence is achieved. If after the version creation process ends, more than one user tries to resolve the conflicts, further conflicts can be generated. For instance, in example 2.1.4, two versions of object id_4 are created, a version situated at position pos_1 and the other at pos_2. Further conflicts are generated if the two users perform some concurrent operations on the two versions. The GRACE approach does not try to reconcile the conflicts by merging objects versions.

The POLO system is the only multi-versioning system that proposes a post-locking scheme [124] for conflict resolution. The post-locking scheme implies that a lock is automatically generated by the system after the generation of multiple versions. Some schemes are proposed that use specific rules for the assignment of the ownership of locks. For instance, one of the schemes involves that, after the generation of multiple versions due to conflicting operations and after each site has reached the stable set of versions, the system can pass the lock ownership to a representative of the involved users. The delegated representative may edit or delete some versions in order to satisfy the group intention and then unlock the intended versions. Another alternative to a delegated representative is that the system invokes a voting procedure and conflicts are resolved by an explicit group intention.

Multiversion techniques can be both used for the real-time and asynchronous modes of communication.

2.3.7 Reconciliation based on state merging

Most of the existing version control systems such as CVS [12], ClearCase [7] and Subversion [19] adopt state-based merging. State-based merging approaches are characterised by the fact that only the information about the states of the documents and no information about the evolution of one state into another is used. Each time a user wants to commit to the central repository the changes he/she performed locally, the updated document is transferred to the repository and a differentiation algorithm such as *diff* [72] is applied in order to generate deltas between the updated document and the original version. Each time an update is performed, the original version from the repository and the modified version of the document from the repository are transferred to the local workspace of the user

who requested the update. Merging algorithms such as *diff3* are then used to merge updates made at remote sites into the local copy. The core of diff algorithms seeks to compare two sequences of symbols and to discover how the first can be transformed into the second by a sequence of operations using the primitives *delete-subsequence* and *insert-subsequence*. The diff family of algorithms is used in text comparison, the basic unit of merging being the text line. Usually, a hashcode of each line is used to represent the line, and the sequences are sorted by hashcode in order to discover groups of identical lines. Once identical elements are found, they are gathered together into runs which were adjacent in the pre-sorted sequences.

Some diff algorithms specialised for hierarchical structures have been proposed for XML documents [119, 113, 18, 29].

One of the semantic state-based approaches for merging can be found in [36]. This paper proposes an algorithm to semantically merge three different versions of a program. The algorithm merges program dependence graphs for the base version and the modified versions into one dependence graph. Based on the merged dependence graph that is checked for interferences, the final merged program is obtained. The algorithm is based on the semantics of a particular programming language and therefore is restricted to operating on programs in that language.

Flexible diff [74] is another semantic state-based approach for merging that finds and reports differences between two versions of text. The PREP writing environment which relies on the diff approach is flexible, allowing the users to indicate the granularity of the changes they want to find, the choices being word, phrase, sentence or paragraph. Moreover, the users can control the granularity of how the changes are shown when they are reported, the choices being again word, phrase, sentence or paragraph. If the user chooses the granularity of a sentence, then any word differences are shown as an old sentence deleted and a new sentence inserted.

A flexible object framework that supports the definition of the merge policy based on the particular application and the context of the collaborative activity has been defined in [71]. Automatic, semi-automatic and interactive merges are possible. Collaboration objects are constructed from basic types such as integers, reals and strings or are aggregate types such as records and sequences. Objects have attributes which can be inherited via type or structure. According to object structure, semantic fine-grained policies for merging can be specified. The approach uses a merge matrix defining the merge functions for the possible sets of operations. The work proposes different policies for merging, but does not specify an ordering of concurrent operations, such as the ordering of execution of two insert

operations or an insert and delete. The authors do not describe how the differences between two document versions are generated. Computing the differences might be difficult, since the structure of the document is hierarchical. The approach could also be applied to operation-based merging and for real-time collaboration. The approach is general, but there is no description of how it could be applied to collaborative editing on text or XML documents.

In the approaches we have just discussed merging is based on states. However, document states do not reflect the evolution of a document by means of operations. Consequently, the issues of causality and intention, as presented at the beginning of this chapter, cannot be addressed. We cannot address causality as only final states of documents are preserved and not the operations representing the modifications performed. User intentions in terms of operations, cannot be deduced from the document states, i.e. the state of a document does not contain the changes performed on the document and, therefore, we cannot discuss the preservation of user intentions. Convergence is however maintained by using the merging algorithms based on states.

State-based merging is not suitable for real-time communication, as each time a user performs a change, the state of the document would have to be transmitted, so that when the operation arrives at a site, the merging of states can be performed. State-based merging has been adopted in asynchronous communication and it suits only this paradigm.

2.3.8 Constraint-based reconciliation

IceCube [86, 57] uses log-based reconciliation that is parameterised by application semantics. Application semantics are expressed by means of static and dynamic constraints imposed on pairs of actions. A static constraint relates two actions unconditionally, for instance it expresses a conflict between two appointment requests for a person at a certain time in two different places. On the other hand, a dynamic constraint consists of the success or failure of a single action depending on the current state, such as overdraft on an account. During the disconnected work phase, a site executes operations locally and it keeps them in a log. During the reconciliation phase the logs of two or more replicas are merged to bring the replicas to a consistent state. The first step in the reconciliation phase is the scheduling phase where the schedulers consider the combinations of the actions that satisfy the static constraints, such as a certain order of execution between pairs of actions. In the scheduling phase, dependence

relations between the operations are taken into account, as well as user defined parameters, to guide the scheduler into exploring those portions of the space of allowed sequences to produce good solutions. The next step in the reconciliation phase is the simulation stage where the schedules are played against replicas of the shared objects and schedules that do not satisfy dynamic constraints are aborted. The last stage is the selection stage that ranks and chooses among the outcomes from the simulation stage. The advantage of the IceCube approach is that it searches for the combinations of concurrent operations that minimize the reconciliation conflicts. But, it has the drawback that during the first stage of reconciliation a combinatorial explosion of the set of solutions to be analysed can occur. Moreover, it is not a purely automatic approach, but rather relies on user interaction.

As the approach is not incremental, i.e. operations are not integrated one by one into the log, causality preservation cannot be addressed. If intentions of users are expressed by means of constraints, constraints are satisfied during merging and, therefore, intentions are preserved. The merging mechanism ensures convergence. However, no application of IceCube has been used for the collaborative editing.

As the approach is not incremental, it is not suitable to be used in real-time communication, but in the asynchronous communication.

2.3.9 Operational transformation

The operational transformation approach has been identified as an appropriate approach for a replicated architecture of a collaborative editing system which maintains the consistency of the copies of the shared document. Its advantage is high responsiveness as it allows local operations to be executed immediately after their generation and remote operations are transformed against the executed operations without the need to undo operations. Transformations are performed in such a manner that user intentions are preserved and, at the end, the copies of the documents converge.

The operation-based approach to merging does not require that the documents are transferred over the network between the local workspaces and the repository. Moreover, no complex differentiation algorithms have to be applied in order to compute the delta between the documents. Therefore, the responsiveness of the system is better in the operation-based approach. Merging based on operations also offers better support for conflict resolution by giving the possibility of tracking user operations. In the case of operation-based merging, when a conflict occurs, the operation causing the

conflict is presented in the context in which it was originally performed. In the state-based merging approach, the conflicts are presented in the order in which they occur within the final structure of the object. For instance, CVS and Subversion present the conflicts in the line order of the final document, the state of a line possibly incorporating the effect of more than one conflicting operation.

The operational transformation approach has been mostly applied to the collaborative editing of text. But, the approach has been also applied to the collaborative editing of graphical documents and of spreadsheets. In what follows we are going to analyse some of the operational transformation approaches. The first operational transformation approach called dOPT [26] was implemented in the GROVE system. Several other works extended the initial proposed operational transformation approach, as described in this subsection.

We first present some of the operational transformation approaches appropriate for linear structures: dOPT [26], Jupiter [77], NetEdit [125], adOPTed [88], GOT/GOTO [108, 107], the SOCT family of algorithms [101, 117, 116], the operation effects relation approach [63], LICRA [55] and FORCE [94]. We then present some applications of the transformational mechanism, such as shared spreadsheets [82], CoWord [110, 122], extension of the operational transformation mechanism to hierarchical structures such as SGML-based documents [21] or XML documents and CRC cards [69], and the application for synchronising file systems [68].

Figure 2.11 illustrates a very simple example of the operation transformation mechanism. Suppose the shared document contains the following text *"concurency contrl"*. Two users, at $Site_1$ and $Site_2$, respectively, concurrently perform some operations on their local replicas of the document. $User_1$ performs operation O_1 of inserting the character *"r"* in the 7th position, in order to correct the misspelling of the word *"concurency"* to obtain *"concurrency contrl"*. Concurrently, $User_2$ performs operation O_2 of inserting the character *'o'* in the 17th position, in order to correct the misspelling of the word *"contrl"* to obtain *"concurency control"*. Let us analyse at each site the effects of executing the operations in their original form. At $Site_1$, after the receipt of operation O_2 and its execution in the original form, the state of the document becomes *"concurrency control"* which is not what the users intended. At $Site_2$, after the receipt of operation O_1 and its execution in the original form, the state of the document, fortunately, becomes a merge of the intentions of the two users, i.e. *"concurrency control"*. But, generally, executing the operations in their original form at remote sites, does not ensure that the copies of the documents at $Site_1$ and $Site_2$ con-

verge. So, we see the need to transform operations when they arrive at a remote site.

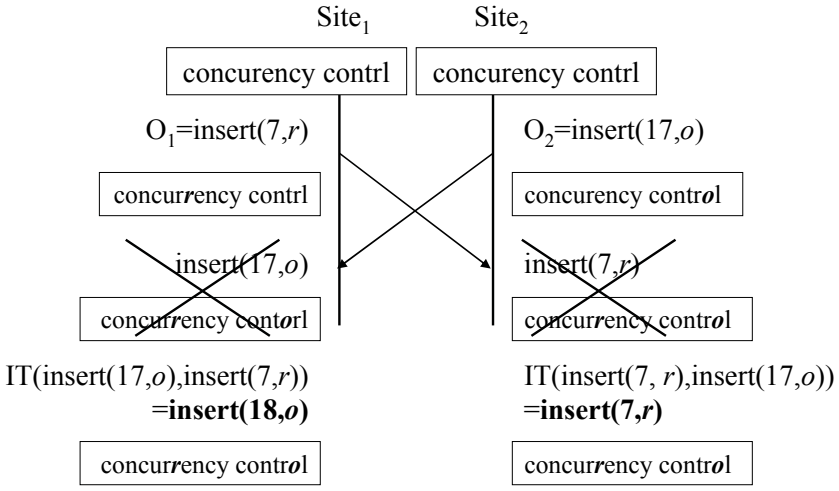


Figure 2.11: Scenario of a real-time cooperative editing session

At $Site_1$, when operation O_2 arrives, it needs to be transformed against operation O_1 to include the effect of this operation. As operation O_1 was generated at the same time as O_2 and it inserted a character before the character to be inserted by operation O_2, operation O_2 needs to adapt the position of insertion, i.e. increase its position by 1. In this way the transformed operation O_2 becomes an insert operation of the character *'o'* into position 18. The result becomes *"concurrency control"* and therefore the intentions of both users are satisfied. At $Site_2$, in the same way, operation O_1 needs to be transformed against O_2 in order to include the effect of O_2. The position of insertion of O_1 does not need to be modified in this case as operation O_2 inserted a character to the right of the insertion position of O_1. Therefore, the transformed operation O_1 has the same form as the original operation O_1. We see that the result obtained at $Site_2$ respects the intentions of the two users and, moreover, the replicas at the two sites converge.

To capture the causal relationship between the operations, a time-stamping scheme based on a data structure called *state vector* (SV) [26] can be used. If n is the number of sites involved in the collaboration, each site k maintains a state vector SV^k with n components. We present next the mechanism for updating the state vectors as described in [109]. Initially, $SV^k[i] = 0, \forall i \in \{0, \ldots, n-1\}$. After executing a local operation, the kth component of SV^k is updated as $SV^k[k] = SV^k[k] + 1$. After executing a local operation and updating SV^k according to the previous given rule, the local operation is timestamped by the current value

of SV^k and broadcast to all remote sites. After executing on site k a remote operation O with the timestamp SV_O, SV^k is updated as follows: $SV^k[i] = max(SV^k[i], SV_O[i]), \forall i \in \{0, \ldots, n-1\}$

For instance, in Figure 2.12, the state vectors associated with O_1, O_2 and O_3 are $SV_{O_1} = (1, 0, 0)$, $SV_{O_2} = (0, 1, 0)$ and $SV_{O_3} = (0, 1, 1)$, respectively.

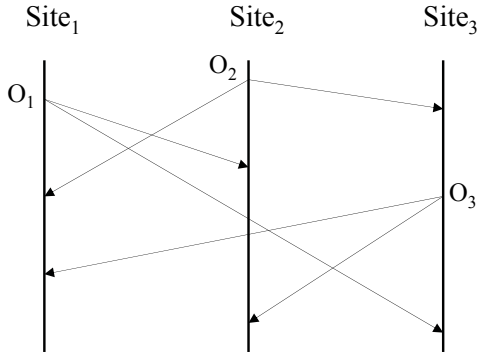


Figure 2.12: Scenario of a real-time cooperative editing session

An operation generated at $Site_i$ and timestamped by SV_i, is *causally ready* to be executed at $Site_j$ having the local state vector SV_j if the following conditions are satisfied:

- $SV_i[i] = SV_j[i] + 1$
- $SV_i[k] \leq SV_j[k], \forall k \in \{0, 1, \ldots, n-1\}$ and $k \neq i$.

The state vector is also used to determine the precedence relationship between two operations. Given two operations O_a and O_b, timestamped by SV_{O_a} and SV_{O_b} respectively, $O_a \rightarrow O_b (O_a \textit{ precedes } O_b)$ iff

1. for all $i \in \{0, \cdots, n-1\}, SV_{O_a}[i] \leq SV_{O_b}[i]$ and
2. there exists one j such that $SV_{O_a}[j] < SV_{O_b}[j]$

The *generation state or context* of an operation is the state of the document before the operation was generated and its *execution state or context* is the state of the document from the site where the operation is to be executed. The *state* of a document is associated with the list of operations that have to be executed to bring the document from its initial state to the current state.

Throughout this thesis we are going to use the following notations used in FORCE [94]. We use $O_a \sqcup O_b$ to denote that the two operations O_a

and O_b have the same generation context. $O_b \mapsto O_a$ denotes that two operations O_a and O_b have the property that the state resulting after the execution of O_b is the generation context of O_a.

dOPT

In dOPT [26] each site maintains a history buffer with all of the operations that have been executed at that site. When a remote operation arrives at a site and it is not causally ready, it is queued in the history buffer. In the case that the operation is causally ready for execution it is sequentially transformed against the concurrent operations from the history buffer. However, this algorithm does not work well when transformations have to be performed between two operations that do not have the same context.

For instance, in the scenario illustrated in Figure 2.12, when operation O_3 is executed at $Site_1$, it cannot simply be transformed against O_1 because O_1 and O_3 are associated with different contexts, i.e. O_1 does not include O_2 in its generation context, while O_3 includes O_2 in its generation context. The dOPT algorithm does not offer a solution for the previously described scenario.

Causality preservation is obtained by the use of state vectors. Convergence is not obtained for the cases previously described. As in most operational transformation approaches, preservation of intentions is not explicitly defined, but it is included in the definition of the transformation functions. Transformation functions are written to preserve the intentions of users.

SOCT2

Although SOCT2 [101, 102] is not chronologically the first algorithm that was developed after dOPT, it is an important representative of the class of algorithms that followed dOPT. Moreover, the implementation of our collaborative systems is based on the principles of SOCT2. Therefore, in what follows we are going to present the main ideas of the SOCT2 approach and then relate the other existing operational transformation algorithms to SOCT2.

Let s be the state of an object and $s \cdot O$ be the state obtained after the execution of the operation O on state s. The intention which is realised by operation O on the state s is denoted as $intention(O, s)$. The function that transforms an operation O_2 against another operation O_1 is defined

as being the operation denoted as ${O_2}^{O_1}$ defined on the state resulting from the execution of O_1 and realising the same intention as O_2:

$$\boxed{\begin{array}{c} transpose\_fd(O_1, O_2) = {O_2}^{O_1} \text{ with } \forall s \in S, \\ \text{where } S \text{ is the set of states where } O_1 \text{ is executed,} \\ intention({O_2}^{O_1}, s \cdot O_1) = intention(O_2, s) \end{array}}$$

In order to achieve convergence two conditions have been defined for the forward transposition function. Condition C_1 guarantees that the state resulting after the transformation of two concurrent operations does not depend on the order in which the operations are executed.

Definition 2.3.1 Condition C_1
Let O_1 and O_2 be two concurrent operations defined on the same state. The forward transposition verifies C_1 iff:

$$\boxed{O_1 \cdot {O_2}^{O_1} \equiv O_2 \cdot {O_1}^{O_2}}$$

where $\equiv$ denotes the equivalence of states obtained after applying both sequences starting from the same state.

Condition C_2 aims at making the transformation of an operation with a sequence of operations independent of the order of the operations in the sequence.

Definition 2.3.2 Condition C_2
Given O_1, O_2 and O_3, the forward transposition verifies C_2 iff:

$$\boxed{O_3^{O_1:O_2} = O_3^{O_2:O_1}}$$

where $O_i : O_j$ denotes $O_i \cdot O_j^{O_i}$.

The forward transposition function requires that both operations are defined on the same state. But, in the case that two concurrent operations do not have the same generation state, the forward transposition function cannot be directly applied. Consider the case illustrated in Figure 2.13. At $Site_1$, $O_1 \rightarrow O_2$ and, at $Site_2$, operation O_3 is generated concurrently with O_1 and O_2. Let us analyse what happens at $Site_2$. When operation O_1 arrives at the site, it is forward transposed against O_3, because O_1 and O_3 have the same generation state, the result being $O_1^{O_3}$. But, when operation O_2 arrives at $Site_2$, it cannot be forward transposed against O_3

as O_2 and O_3 do not have the same generation state. The generation state of operation O_2 contains the execution of operation O_1, while operation O_3 does not know about the execution of operation O_1. In order to deal with such cases another transposition function called *transpose_bk* has been defined.

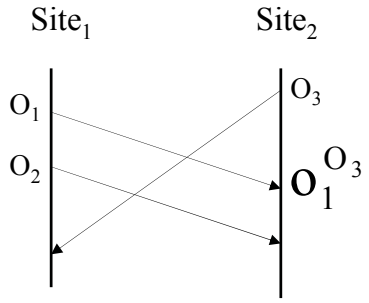


Figure 2.13: Concurrent operations with different generation contexts

Backward transposition is defined in order to change the execution order of a pair of operations while respecting user intention. Backward transposition of a pair of operations (O_1, O_2) executed in this order is the pair of operations (O_2', O_1') that corresponds to their execution in reverse order that leads to the same outcome.

$$\begin{array}{c} transpose\_bk(O_1, O_2) = (O_2', O_1') \\ \text{where } O_2 = transpose\_fd(O_1, O_2') \\ \text{and } O_1' = transpose\_fd(O_2', O_1) \end{array}$$

In the example shown in 2.13, when O_2 needs to be transformed against O_3, operations O_2 and O_3 must have the same generation state. By applying the backward transposition between O_3 and $O_1^{O_3}$, the transformed form of operation O_3 denoted O_3' will include in its context the effect of operation O_1. Therefore, O_3' and O_2 have the same context and the forward transposition between O_2 and O_3' can be safely applied.

In SOCT2 each site S involved in the collaboration maintains a local copy of the document and a history buffer $H_S(n) = O_1 \cdot O_2 \cdot \ldots \cdot O_i \cdot \ldots \cdot O_n$ consisting of a sequence of n operations executed on the local copy of the document. The operations in $H_S(n)$ are ordered according to their order of execution with the first operations being the older ones. When a local operation is generated, it is simply appended to the history buffer. When a remote operation arrives at a site, it is checked whether it is causally ready or not. If the operation is not causally ready, it is added to a queue and executed when it becomes causally ready, i.e. all operations that precede it have been executed.

In what follows we describe in detail the SOCT2 algorithm for integrating a causally ready operation into the history buffer.

The history buffer contains a mixed sequence of operations that are either concurrent or preceding the remote operation. The first step of the integration procedure consists of the reordering of the history buffer such that all operations that are causally preceding the remote operation come before the operations that are concurrent to the remote operation in the history buffer.

In order to achieve the reordering of operations, the history buffer is traversed from left to right. If an operation that precedes a remote operation is encountered, it is repeatedly backward transposed so that all operations concurrent with the remote operation are situated on its right side in the history buffer.

The second step in the integration mechanism consists of forward transforming the remote operation according to the sequence of concurrent operations.

The process of the integration of a remote operation into the history buffer is illustrated in Figure 2.14.

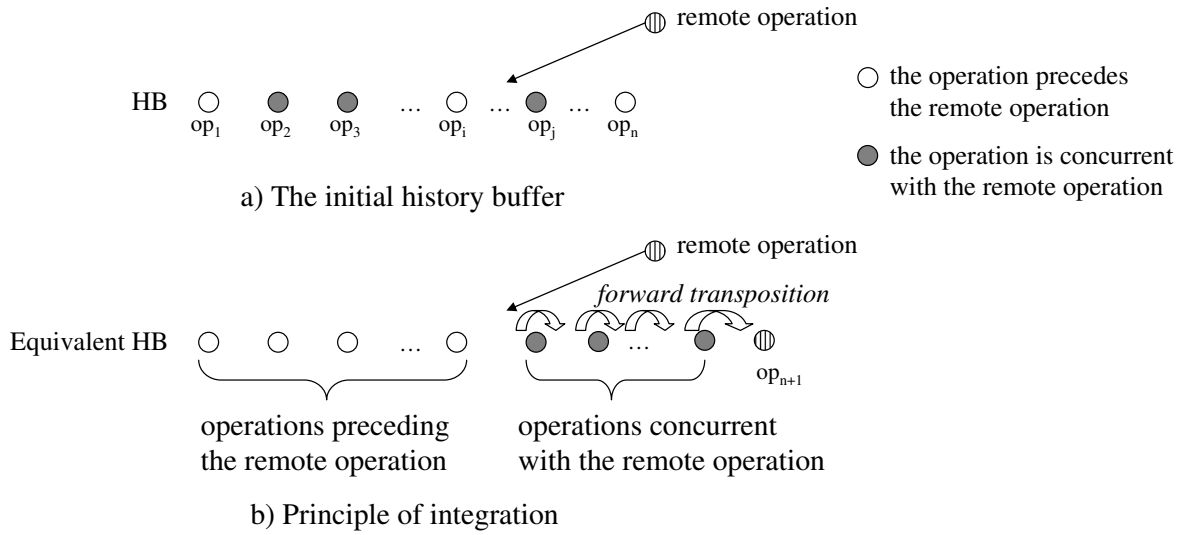


Figure 2.14: Integration of a remote operation in SOCT2

As we have already seen, the integration algorithm described above uses procedures to forward and backward transform two operations. In Appendix A we present the transformation functions as they have been used in the SOCT2 approach. We also present a scenario where inconsistency is obtained due to the transformation functions. SOCT2 is a generic algorithm, but the transformation functions for text documents rely on a linear representation of the document, where the operations that can be

performed are the insertion and deletion of single characters. There is no suggestion how the approach can be extended for a hierarchical structure of the document.

An application of the SOCT2 operational transformation approach to real-time collaborative graphical editing based on objects has been proposed in [100]. The objects subject to the collaboration are circles, rectangles and polygons. The operations that can be executed on the objects subject to the collaboration are dimension change, colour change, fill, move and rotate. One of the algorithms from the SOCT family is used to maintain consistency in the case of collaborative graphical editing. The forward and backward transformation functions are defined for each pair of operations. Temporal priorities are assigned to operations. In the case of a transformation between a pair of operations of the same type having different parameter values, the operation with the highest priority is executed, the other operation being cancelled. However, the proposed approach does not deal with groups of objects and operations of grouping and ungrouping.

In SOCT2 causality preservation is obtained by the use of state vectors. SOCT2 algorithm works on characters and therefore the operations in the example 2.1.2 are defined on characters and not on strings and the result obtained at the two sites would be *"Our algorithm applie**sd recursively** a linear merging algorithm .**The approach offers increased efficiency.**"* in the case where priority of $Site_1$ is higher than priority of $Site_2$. Convergence is obtained at the two sites for this example. However, as previously mentioned, due to the transformation functions, there are cases when convergence is not obtained as shown in Appendix A. Even if transformation functions implicitly include how the intention of an operation is kept when it is transformed against another operation, the operation intention is not formally defined and therefore we cannot judge if intention is preserved or not.

Jupiter

One of the operational transformation approaches following dOPT was the algorithm used in the Jupiter [77] collaboration system developed at Xerox PARC. Jupiter is a multiuser, multimedia virtual world intended to support long-term remote collaboration. Jupiter uses a central server which has the role of maintaining the consistency of shared objects, such as text objects. The consistency maintenance algorithm used by Jupiter is an adaptation of the dOPT algorithm to an environment with multiple replicated client sites as well as a central site.

Even though the Jupiter algorithm was published earlier than SOCT2, we are going to present the principles of the Jupiter algorithm by relating it to the SOCT2 algorithm previously described. In contrast to dOPT which implemented an n-way synchronisation, Jupiter uses a 2-way synchronisation approach that allows a client to synchronise with a server. Shared documents are replicated at all cooperating client sites, and also maintained at the central server. Consequently, only client-server communication is needed. A local operation is executed immediately and then propagated to the server. The server transforms the operation, if necessary, then executes the transformed operation on its copy of the document and broadcasts the transformed operation to all other client sites. When an operation sent by the central server is received at a client site, it has to be transformed before it is executed on the local copy of the document. The approach does not deal with the causality problem, since it uses an ordered, reliable communication channel. A 2-dimensional state space graph is used instead of a history buffer as in dOPT in order to keep track of the paths to follow when a new remote operation needs to be transformed against the previously performed operations.

The nodes of the state space graph represent application states, and are equivalent of state vectors in SOCT2. As synchronisation is performed between two sites only, the state vector contains two elements, the first element representing the number of operations locally generated and the second element representing the number of operations received from the other site. The edges of the graph represent either the original user requests or the result of transformations of operations. The transformation function involving two operations o_1 and o_2 returns as a result a pair of operations (o_1', o_2') where o_1' is obtained by forward transposing o_1 with o_2 and o_2' is obtained by forward transposing o_2 with o_1. The forward transpositions are the ones used in SOCT2. A transformation can be performed only when the two operations have been generated from the same state, a condition required by the forward transposition function defined in SOCT2. A transformation of an operation against another operation is obtained by translating the first operation along the vector representing the second operation. For instance, Figure 2.15 illustrates the fact that the transformation of vector s_1 against c produces s_1'. But, s_2 cannot be transformed against c as s_2 as c was not generated from the same state, c being generated in the state $(0, 0)$ and s_2 being generated in the state $(0, 1)$. By translating c from the state $(0, 0)$ to the state $(0, 1)$, i.e. transforming c against s_1, the resulting operation c' will have the same generation state as s_2. Therefore, s_2 can be transformed against c', the result being s_2'.

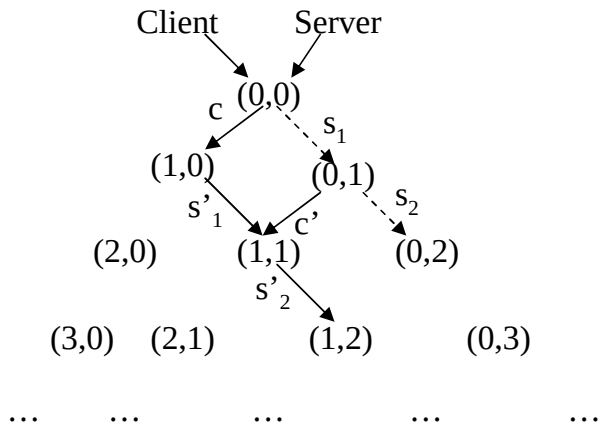


Figure 2.15: The state graph in the Jupiter approach

We now explain the main idea of the execution of a remote operation at a client site. Suppose that the last known state of the server at a client site was (x, y). Further, suppose that since that state the client sent k messages, the state at the client site being $(x + k, y)$, as illustrated in Figure 2.16.

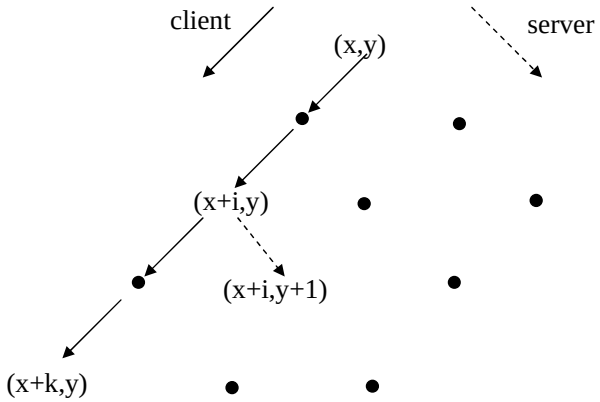


Figure 2.16: Scenario for the Jupiter algorithm

These k messages are kept in the state space graph as they are used in the process of transformation of the incoming server operations. The next incoming message from the server must originate from one of the states between (x, y) and $(x + k, y)$, i.e. the server must have processed some of the k messages generated by the client. Suppose that the server message originates in the state $(x + i, y)$ and therefore the current state of the server is $(x + i, y + 1)$. The saved operations locally generated between the states (x, y) and $(x + i, y)$ are discarded since they are no longer needed, the server having already processed them. Remote operations generated by the server have to be sequentially transformed against the client operations that are not included in their generation state. These operations are

generated between the states $(x + i, y)$ and $(x + k, y)$. As a result of this transformation, the edge of the graph originating at $(x + i, y)$ and ending at $(x + i, y + 1)$ is translated to the edge starting at $(x + k, y)$ and ending at $(x + k, y + 1)$. Meanwhile, the operations saved at the client have to be transformed in order to include the effect of the remote operation from the server, such that a new remote operation that arrives at the client site is correctly transformed against the saved operations. Therefore, the sequence of saved operations ranges between the server states $(x + i, y + 1)$ and $(x + k, y + 1)$.

Causality preservation is ensured by the use of a reliable communication channel. The approach ensures convergence. Intentions of operations are not explicitly defined, but transformation functions express the preservation of intentions between pairs of concurrent operations.

Jupiter uses a 2-way synchronisation approach where a client synchronises with a server. The 2-way synchronisation approach has been extended to a multi-way synchronisation in the NetEdit and adOPTed which are next presented.

NetEdit

NetEdit [125] extended the 2-way synchronisation approach used in Jupiter to a multi-way protocol for synchronisation. All clients maintain a local copy of the shared document and a state-space graph to keep track of the operations that have been executed locally. The server maintains a copy of the document and a state-space graph for each client. Each client-server pair synchronises their copies in the same way as was done by the 2-way communication in Jupiter. The generalisation of the 2-way synchronisation to the n-way synchronisation consists of the fact that when a message generated by a client is received by its corresponding server, it is transmitted to the other servers. In their turn, servers synchronise with the corresponding clients, as if that message had been generated locally. Forwarding a message from a server to other servers has to be atomically processed before another message is processed by any of the servers. In NetEdit concurrent messages are processed in the order in which they arrive at the server. A client processes immediately the locally generated operations and then the remote operations are processed in the order in which they have been processed at the server side. Therefore, the generality obtained by distributed collaborative systems where the messages arrive at the sites in an arbitrary order established by the latency in the network is not maintained. A disadvantage of the centralised approach,

compared to the distributed approach, is the latency in the transmission of the operations to the central server. Moreover, in the proposed approach, no information is provided about whether the causality ordering between the operations is maintained, i.e. whether two operations generated in a certain order by a client are processed in the same order by the other clients. Convergence is achieved and intention preservation is implicitly included in the transformation functions.

In summary, NetEdit decreases the responsiveness of a collaborative system as it relies on central server processing.

adOPTed

Another generalisation of the Jupiter approach is adOPTed [88]. It uses a multidimensional directed graph to model interactions between users during collaborative editing. The multi-dimensional graph supports a formal representation of necessary and sufficient conditions which ensure the correctness of the operational transformation algorithm. The multi-dimensional graph used in adOPTed is a generalisation of the two dimensional graph used in Jupiter that offers support for the selection of the correct path and the correct operations for transformation. As in the two dimensional graph, the vertices of the multidimensional graph represent application states, i.e. are equivalent to state vectors used in SOCT2 and the edges of the graph represent user requests that are either original or the result of the transformation of operations.

The number of dimensions of the multi-dimensional graph is equal to the number of the users involved in the collaboration. In Figure 2.17, operations r_1, r_2 and r_3 are generated concurrently by three users in state S_0.

The transformation function in adOPTed is called *L-Transformation*. Similarly to Jupiter, it returns as a result a pair of operations representing the forward transpositions used in SOCT2. L-transformation is performed in the multi-dimensional graph in the same way the transformation in Jupiter is performed in the two dimensional graph. In Figure 2.17 a transformation of r_3 against r_1 is represented by the edge denoted $tf_1(r_3, r_1)$.

A multi-dimensional graph has been used as a support for the correctness of the algorithm. Two transformation properties similar to the ones established in SOCT2 have been defined as constraints imposed on the transformation functions.

The first transformation property, similar to condition C_1 in SOCT2, ensures that the effect of executing request r_1 followed by request r_2 trans-

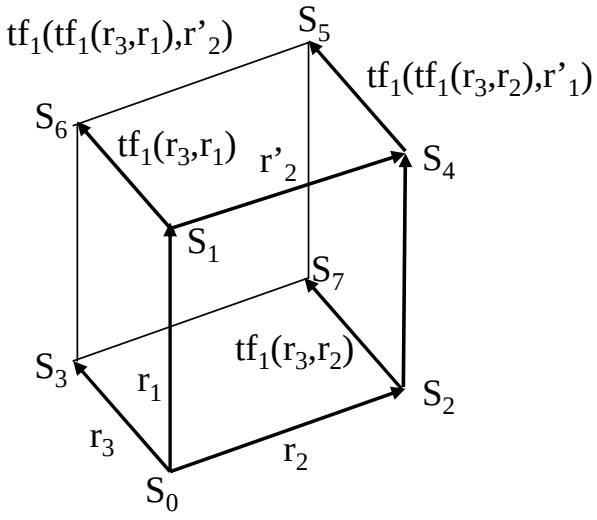


Figure 2.17: Multidimensional graph in the adOPTed approach

formed against r_1 is the same as executing request r_2 followed by request r_1 transformed against r_2. The two transformation paths should lead to the same state. For instance, in the example illustrated in Figure 2.17, the execution of r_1 followed by the transformation of operation r_2 against r_1 should lead to state S_4, as well as the execution of r_2 followed by the transformation of operation r_1 against operation r_2.

The second transformation property similar to condition C_2 in SOCT2 ensures that transforming a request against two other requests that have been generated on the same state does not depend on the order in which the transformations are performed. For instance, transforming r_3 against r_1 and then against r_2 should result in the same state as transforming r_3 against r_2 and then against r_1.

We saw that, in order to satisfy the condition of applying the forward transformation function that the operations must have the same generation context, another transformation called backward transposition was defined in SOCT2. In [88] no additional transformation function is defined, since the precondition of the forward transposition can be ensured by using the intermediate states of the interaction model. Intermediate states are obtained by computing the intermediate forward transformations, as explained in what follows. When a remote operation generated by $User_j$ arrives at a site, it is executed by translating that operation to the current state of the application. If operation r has to be translated to state v, as shown in Figure 2.18, an arbitrary intermediate predecessor v' of v is determined such that r may be recursively translated to v', the result being r'. If v' is a predecessor of v along the i-th coordinate axis, then the $v[i]$-th

operation generated by $User_i$ is recursively translated to v, resulting in r'_i. The L-Transformation between r' and r'_i is computed, the final result, i.e. the pair r" and r"$_i$, being stored in the interaction model.

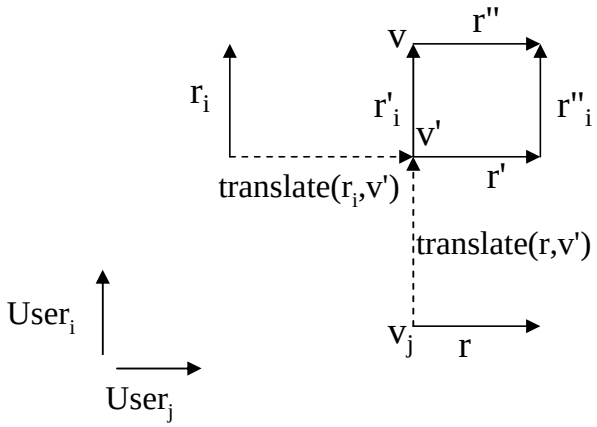


Figure 2.18: Recursive definition of a translate request (r,v)

The transformation functions for the text editing application of the adOPTed approach have been specified in [87]. These transformation functions are an adaptation of the transformation functions proposed in the dOPT approach and, as shown in [52], they contain some flaws.

We summarise next how adOPTed deals with the challenging problems in maintaining consistency - causality violation, intention violation and divergence. Causality relation between operations is maintained. Operation intention is not explicitly defined, but the transformation functions implicitly express what the intention of an operation is when it is transformed against another operation. The adOPTed approach ensures convergence, but due to the imperfect transformation functions there are cases when convergence is not achieved.

GOT

Another approach extending dOPT is the GOT [108] algorithm. GOT defines a total ordering between operations. The causal ordering determines an ordering only between causally preceding operations, the execution order of concurrent operations being arbitrary. A total ordering between operations is used in order to ensure convergence in the presence of concurrent operations. Total ordering between two operations is defined based on the sum of the elements of the state vectors. Operation O_1 precedes in total order operation O_2 if the sum of the elements of the state vector associated with O_1 is less than the sum of the elements of the state

vector associated with O_2 or the sums are equal, but the identifier of the site where O_1 was generated is less than the identifier of the site where O_2 was generated. Based on the total ordering between operations, an undo/-do/redo scheme has been defined. When a remote causally ready operation arrives at a site, all the operations in the history buffer that follow it have to be undone in order to restore the document to the state before their execution. Afterwards, the remote operation is transformed and executed and the operations that have been undone are in their turn transformed and re-executed.

In [106] two types of transformations have been defined: inclusion and exclusion. *Inclusion transformation* $IT(O_a, O_b)$ is defined similarly to the forward transposition used in SOCT2 as returning the transformed operation O_a that includes operation O_b in its context. The condition for applying the inclusion transformation is that O_a and O_b have the same generation context. *Exclusion transformation* $ET(O_a, O_b)$ returns the form of the operation O_a that excludes the preceding operation O_b from its context. The exclusion transformation corresponds to the first operation from the pair returned by the backward transposition function defined in SOCT2. The condition of applying the exclusion transformation is that the state obtained after the execution of O_b is the generation context of O_a. The inclusion and exclusion transformation functions have to be reversible, i.e. if O_a and O_b have the same definition context, then $O_a = ET(IT(O_a, O_b), O_b)$ and if the state resulted after the execution of O_b is the definition state of O_a, then $O_a = IT(ET(O_a, O_b), O_b)$.

In order to integrate a remote operation into the history buffer the following approach is applied. Consider a causally-ready remote operation O_{new} that has to be integrated into the history buffer $HB = [EO_1, \ldots, EO_m, \ldots, EO_n]$. Operations in the history buffer have to be ordered according to a global order. Therefore, the operations in *HB* are undone from right to left until an operation EO_m is found such that EO_m totally precedes O_{new}. Afterwards, O_{new} has to be transformed against the list of operations $HB' = [EO_1, EO_2, ..., EO_m]$ that precede in total order operation O_{new}. Some operations in the *HB'* are causally preceding O_{new} and some others are concurrent with O_{new}. O_{new} has to be transformed against the concurrent operations, but they have different generation contexts. By traversing *HB'* from left to right the first operation EO_k that is concurrent with O_{new} is determined. Let $EOL = [EO_{c1}, EO_{c2}, ..., EO_{cr}]$ be the list of operations that causally precede O_{new}. These operations have to be excluded from the context of O_{new}. In order to satisfy the precondition of the exclusion transformation, the operations from EOL have to exclude the effect of the

concurrent operations with O_{new} from their context. Let us denote the list of transformed operations with $EOL' = [EO'_{c1}, EO'_{c2}, ..., EO'_{cr}]$. Each element EO'_{ci} is obtained by first excluding from EO_{ci} the effect of the list of operations from *HB'* situated at its left side starting with the index k and then including the effect of any previously transformed operations $[EO'_{c1}, EO'_{c2}, ..., EO'_{ci-1}]$. After excluding the list EOL' from the context of O_{new}, O_{new} can safely include the effect of the operations from *HB* starting with the index k, the result of these transformations being O'_{new}. O'_{new} can then be executed.

Each operation that has been undone has to include the effect of the O'_{new} operation. But, in order to apply the inclusion transformation, the operations must have the same definition context. Let $UL = [EO_{m+1}, EO_{m+2}, \ldots, EO_n]$ be the list of operations that follow in total order the remote operation O_{new} and that have been undone. Further, let us denote by $UL' = [EO'_{m+1}, EO'_{m+2}, ..., EO'_n]$ the list of transformed forms of the undone operations. Each operation EO'_{m+i} starting from left to right is obtained by first excluding the list of operations $[EO_{m+1}, EO_{m+2}, ..., EO_{m+i-1}]$ and afterwards including the effect of O'_{new} and of the already transformed operations $[EO'_{m+1}, ..., EO'_{m+i-1}]$.

The inclusion and exclusion transformation functions have been defined for the domain of collaborative text editing for insert/delete operations working on strings. The transformation functions used in the GOT approach are given in Appendix B.

Causality between operations is preserved by means of state vectors. The approach proposed in [108] is the first approach that separates the issue of intention preservation from the issue of convergence. Convergence is achieved by means of the control algorithm and the transformation functions implicitly include the intentions of operations. However, operation intention was not formally defined.

GOTO

In [107] an optimised version of the GOT algorithm, called GOTO, is proposed. GOTO applies the same algorithm that has been used in SOCT2. When a remote operation has to be integrated into the history buffer, the history buffer is reordered such that the operations that causally precede the remote operation are situated in the history buffer before the operations that are concurrent with the remote operation. Afterwards, the remote operation has to be transformed against the operations that are concurrent with it. The same transformation functions that have been used in GOT

are also used in GOTO. In Appendix B we provide a scenario where the GOTO algorithm together with its transformation functions, produce the divergence of the document copies.

SOCT3

The verification of condition C_2 of the SOCT2 approach is not trivial. In [52] the authors propose a tool called SPIKE which automatically proves conditions C_1 and C_2 of the transformation functions. If correctness is violated, the tool provides counter-examples. However, correcting the transformation functions is hard even if counter examples are provided.

Condition C_2 can be replaced by ensuring that concurrent operations are ordered in the same way on all sites, as proposed in [117] by SOCT3. This can be achieved by a global serialisation order denoted $precede_S$ that respects the causal order $\rightarrow$. A sequencer for a distributed system can be used in order to deliver increasing timestamps. An operation generated on a site is executed without delay. Afterwards, a timestamp is assigned to the operation and the operation is transmitted to the other sites. The reception procedure ensures a sequential execution of operations according to the ascending order of their timestamps, by delaying the execution of an operation until the operations with lower timestamps have been executed. State vectors are used in this case only to determine concurrent operations.

The integration procedure of a remote operation orders the operations in the history buffer according to their timestamps. The history buffer has the following representation: $HB = op_1 \cdot op_2 \cdot \ldots \cdot op_i \cdot op_{L_1} \cdot op_{L_2} \cdot \ldots \cdot op_{L_m}$, where $op_1, op_2, \ldots, op_i$ have continuous timestamps, the local or remote operation op_j, $1 \leq j \leq i$ being sequentially delivered and $op_{L_1}, op_{L_2}, \ldots, op_{L_m}$ may have discontinuous ascending order timestamps, op_{L_k}, $1 \leq k \leq m$ being a local operation executed but not delivered.

When an operation op_{i+1} with timestamp $i+1$ is delivered by the reception procedure, it can be the case that op_{i+1} is a local operation op_{L_1} or it is a remote operation. If op_{i+1} is a local operation op_{L_1}, no additional computation is needed, because op_{L_1} has been already executed. If op_{i+1} is a remote operation, op_{i+1} needs to be integrated into the history buffer by determining the operation that realizes the same intention as op_{i+1} and reordering the history to conform to the ascending order of timestamps. In order to determine the operation that realizes the same intention as op_{i+1}, the same procedure as in SOCT2 is performed: the history buffer is divided into two sequences of operations, one that contains the operations that precede op_{i+1} and one that contains the operations that are concurrent

with op_{i+1}, and, afterwards, op_{i+1} is forward transposed to the sequence of concurrent operations. Additionally to SOCT2, after the correct execution form of the operation op_{i+1}, denoted op'_{i+1}, has been determined, the operation has to be placed at the correct position in the history buffer according to its timestamp. This is realised by backward transposing op'_{i+1} against all op_{L_k} in *HB*. The resulting history is identical to the one obtained if the operations were executed in the timestamp ascending order.

Although SOCT3 was meant to eliminate the C_2 condition for its transposition functions, in [116] it was shown that there are cases where the transposition functions have to ensure C_2. Therefore, SOCT3 approach has no advantage over the SOCT2 algorithm as they both require condition C_2. Only an algorithm that works without the backward transposition eliminates the need for C_2.

In SOCT3 causality relation between operations is maintained, as operations are executed in a serial order that respects the causal preceding order. Transformation functions are the same as in SOCT2 and they implicitly include the intentions of operations when they are transformed against other operations. However, as shown in [116], there are cases when convergence is not achieved, due to the transformation functions that do not satisfy condition C_2.

SOCT4

As in SOCT3, in SOCT4 [117], operations are globally ordered according to the timestamps generated by a sequencer. SOCT4 differs from SOCT3 regarding the broadcast of a locally generated operation. In SOCT3 an operation is broadcast as soon as a timestamp generated by the sequencer is assigned to the operation. In SOCT4 the broadcast of a local operation is deferred until all operations preceding it according to the timestamp order have been received and executed. Before the broadcast, the local operation is forward transposed with the concurrent operations, i.e. with remote operations received after its generation and preceding it in the global order.

In what follows we describe the integration procedure of a remote operation op_{i+1} with timestamp $i + 1$ at site S. Before being broadcast, the operation op_{i+1} has been transformed against the operations op_j, where $1 \leq j \leq i$. When the operation op_{i+1} is delivered on site S by the sequential reception procedure, all operations op_j, $1 \leq j \leq i$, have been received and executed on site S. If there are no local operations with a higher timestamp than $i + 1$, the operation op_{i+1} can be executed directly. Otherwise, considering that the history buffer has the

form $H_S = op_1 \cdot op_2 \cdot \ldots op_i \cdot op_{L_1} \cdot op_{L_2} \cdot \ldots \cdot op_{L_m}$ where op_{L_1}, op_{L_2}, $\ldots op_{L_m}$ are the local operations that are waiting to be broadcast, the following steps have to be performed:

- operation op_{i+1} is forward transposed according to the sequence of operations $op_{L_1} \cdot op_{L_2} \cdot \ldots \cdot op_{L_m}$ and the resulting operation is then executed on the local state.

- the history is reordered according to the timestamp order. The local operations $op_{L_1}, op_{L_2}, \ldots, op_{L_m}$ are forward transposed one after another to take into account the execution of the concurrent operation op_{i+1}, and the operation op_{i+1} is stored without any modification at position $i + 1$ in the history.

In contrast to SOCT3, in SOCT4 the state vectors as well as the backward transposition are not needed.

As we have already seen, the principle of SOCT4 relies on the deferred diffusion of operations. This means that an operation is sent to the other sites only after all the other operations with a lower timestamp have been executed at that site. Moreover, the operation is sent after it has been transposed with all concurrent operations preceding it in a total order. The advantage of SOCT4 is that it simplifies the integration procedure and the backward transposition is no longer necessary. The disadvantage of SOCT4 is that by using the deferred diffusion, the communications between the different sites are serialised. As we have seen, SOCT3 does not use a deferred diffusion of operations, therefore exploiting the parallelism of the communications among the sites.

Causality between operations is achieved due to the serialisation order according to the timestamping scheme. Intentions of operations are implicitly specified in the transformation functions and SOCT4 approach ensures convergence.

SOCT3 and SOCT4 rely on a form of centralisation and therefore they are not suited to peer to peer networks. In both algorithms, operations are globally ordered according to timestamps generated by a sequencer.

SOCT5

The goal of SOCT5 [116] is to combine the advantages of SOCT3 and SOCT4, i.e. to use an immediate diffusion of the operations, but not to use the backward transposition in the integration procedure.

The idea is to simulate on each site the behaviour of the whole collaborative system. This is done by replicating on each site the parts of the history used in SOCT4 for memorizing the operations attending diffusion. On each site S, the history of local operations attending the diffusion is becoming the history of operations attending the global serialisation, denoted H_{Sd_S} and the history of delivered operations is the history of serialised operations denoted H_{S_s}. In addition to these histories, each site keeps the histories of operations originating from different sites attending the serialisation, denoted H_{Sd_x}, where x is the name of the distant site.

Once an operation is generated on the site S, it is immediately executed and it is placed in the history H_{Sd_S}.

The procedure for processing remote operations is based on a causal ordering relation, i.e. when the site processes the distant operation op_{i+1} timestamped with $i+1$, all the operations op_j causally preceding it have already been received by the site. The timestamps of these preceding operations op_j satisfy $j \leq i$.

An operation op_{i+1} is ready to be serialised when all other operations with a lower timestamp have been serialised. The moment an operation op_{i+1} is serialised corresponds to the moment when the operation on the generation site is ready to be distributed in the SOCT4 algorithm. In SOCT4 when an operation is distributed it is already transposed against all concurrent operations that have been serialised before it. In the same way, in SOCT5, when an operation is ready to be executed on site S, it has to be forward transposed with respect to all concurrent operations serialised before it.

The serialisation of an operation op_{i+1} consists of the following steps:

- Determine the operation to be executed on the current state. If the operation op_{i+1} is a remote operation, it is forward transposed to take into account local concurrent operations $op_{L_1}, op_{L_2}, \ldots, op_{L_m}$ contained in H_{Sd_S}. Afterwards op_{i+1} is executed. If operation op_{i+1} is local, it has already been executed and no further processing is necessary.

- Determine the operations that have to be placed in the so called filters that help transposing the operations that are concurrent with the serialised operations, but have not been received yet. For every history $H_{Sd_{S'}}$ containing the operations from site S' that are waiting for serialisation, op_{i+1} is forward transposed with respect to the operations $op_{d_{S',1}}, op_{d_{S',2}}, \ldots, op_{d_{S',m'_S}}$ belonging to $H_{Sd_{S'}}$. The resulting

operation is placed at the end of the filter $H_{Sf_{S'}}$. This allows the forward transposition of the operations that arrive afterwards.

- Forward transpose the operations waiting for serialisation of each history H_{Sd_x} to take into account the execution of the concurrent operation op_{i+1}.

The steps above can be combined, in the same way as has been done in SOCT4. In what follows we describe the procedure of insertion of an operation into the filter and the procedure of integration of an operation that is ready to be processed.

- **Insertion of an operation into the filter**

  Suppose that operation op_{i+1} has to be serialised on the site S. op_{i+1} has to be forward transposed against the operations $op_{d_{S',1}}, op_{d_{S',2}}$, $\ldots, op_{d_{S',m'_S}}$ originating from site S'. Suppose that the filter at site S', $H_{Sf_{S'}}$, contains the operations $op_{c_{S',1}}, \ldots, op_{c_{S',k}}$. The operation $op_{c_{S',k+1}}$ resulting from the forward transposition of op_{i+1} against the operations in $H_{Sf_{S'}}$ is placed at the end of $H_{Sf_{S'}}$.

  Simultaneously to this filtering, the operations belonging to $H_{Sd'_S}$ are transformed with respect to op_{i+1}.

  If during the transformation process, a remote operation that causally follows op_{i+1} is encountered, the transformation process can be finished, as the operations are causally ordered and op_{i+1} causally precedes all the following operations. Moreover, the transformed operation op_{i+1} does not need to be kept as there is no possibility that other concurrent operations to op_{i+1} are received by site S.

- **Integration of an operation into the processing procedure**

  Suppose that the procedure processes operation op_k. In the case that the operation is local, no processing needs to be done as the operation is already present in H_{Sd_S}. Otherwise, suppose that the operation has been generated by site S'. op_k needs to be forward transposed with respect to the operations in the filter, to take into account the remote operations that have been previously serialised. Suppose that the operations belonging to filter $H_{Sf_{S'}}$ are $op_{c_{S',1}}, \ldots, op_{c_{S',k}}$. Further, suppose that the operations in the history $H_{Sd_{S'}}$ containing the operations waiting for serialisation are $op_{d_{S',1}}, op_{d_{S',2}}, \ldots, op_{d_{S',m'_S}}$. The operation obtained after the transformation of op_k against the

filter $H_{Sf_{S'}}$ is denoted by $op_{d_{S',m'_{S}+1}}$ and is placed at the end of $H_{Sd_{S'}}$, conforming to the causal order. Simultaneously, the operations from $H_{Sf_{S'}}$ are forward transposed with respect to op_k.

During this processing, the operations that causally precede the operation that is received are eliminated from the filter.

Both the causality relationship between operations and document convergence are achieved by SOCT5. Intention of operations is not formally defined and it is implicitly included in the transformation functions.

SOCT5 is clearly the most complicated approach from the family of the SOCT algorithms. However, it is the best solution because it circumvents the need to test condition C_2 and does not impose a serialisation of the communication between the sites.

The effects relation approach

As we have already seen, transformation functions have to satisfy conditions C_1 and C_2. Condition C_2 is very hard to check and, as shown in [52], most of the proposed transformation functions do not satisfy condition C_2. The basic problem of the transformation functions is the so called "false-tie" puzzle, a scenario where an insert operation has to be transformed against another insert operation, the two insert operations having the same position parameters. A tie-breaking rule based on the use of site identifiers where the operations have been generated or other total ordering schemes may generate incorrect results.

In [63] the authors propose an approach based on the effects relation. The effects relation between two operations relies on the fact that there exists a total order relation between the target characters of the operations. The effects relation between the operations is used as a criterion for breaking a tie, instead of the current positions of the operations. The last synchronisation point between the two operations is computed, i.e. the latest common state on which the operations are defined, and the relations between the operations according to the last synchronisation point are used as a criterion for breaking the tie.

The control algorithm used in the approach presented in [63] is the same as the control algorithm in the SOCT2 approach.

The effects relation approach is the first approach that formally defines the intention of operations. Preserving the intention of an operation is defined as being the preservation of the operation effects relation. The

causality relation between operations is preserved as in the previous approaches by the use of state vectors. The approach ensures convergence.

LICRA

The LICRA (Lock-free Interactive Concurrency Resolution Algorithm) [55] approach for real-time object-based graphical collaborative editing relies on direct dependency relations between generated operations as well as on the operation transformation mechanism. Direct dependency relations between operations are used instead of state vectors. When an operation is propagated to other sites, the message also contains the identifier of the last operation executed at the site. Relationships of commutativity, masking and conflict between operations are used in the operational transformation process, the limitations being the same as the ones mentioned in the GroupDesign approach presented in subsection 2.3.5. Beside a history buffer containing a list of operations, each site maintains a set of operation lists which hold the received operations that cannot be executed because some preceding operations have not been received yet, and also a list of awaited operations. An operation O is awaited by a site if the site already received an operation that directly depends on O before the reception of O itself. A disadvantage of LICRA is that the number of sites involved in the editing process needs to be constant, so a user cannot dynamically join or leave a group. In addition to operation semantics, operation transformation depends also on the priority of the site where the operation has been generated. A priority function is defined as a total order over the set of site identifiers. The approach proposed in LICRA does not deal with operations of grouping and ungrouping.

The causality precedence relation between operations is maintained and convergence is guaranteed. Intention of operations is not defined, however it is implicitly included in the definition of the relationships of commutativity, masking and conflict.

FORCE

In [94] an approach for merging text documents based on the operational transformation mechanism is proposed.

As stated in [22], syntactic conflicts occur at the system infrastructure level, while semantic conflicts are inconsistencies from the perspective of the application domain. Generally, operational transformation algorithms solve the syntactic inconsistency problems in collaborative text editing, but

they do not enforce semantic consistency. The merging approach proposed in FORCE is flexible, the semantic merging policies being separated from the syntactic merging. However, FORCE has been used only for asynchronous collaborative editing on text documents conforming to a linear representation.

Due to the fact that our approach for merging is based on FORCE, we present FORCE in detail in section 3.4.3.

In the next subsection we present some collaborative applications that use the operational transformation approach, such as shared spreadsheets, collaborative editing of SGML and XML documents and the synchronisation of file systems.

Applications of the operational transformation approach

- **Distributed Shared Spreadsheet**

  [82] presents the application of the operational transformation approach has been extended to collaborative editing of shared spreadsheets. A spreadsheet is a two-dimensional array of cells, a one-dimensional array of default formats and a set of name bindings. Each cell may contain a value, a formula and a format. The value of a cell is the result of evaluating the formula and the format controls the manner in which the value is displayed. The set of operations that can be performed on the spreadsheet are the following: setting a value for a cell, formatting a column using a given format, inserting/deleting rows or columns, copying a cell from a source position to a destination position, creating/removing a mapping from a name to a specific range of the spreadsheet. Transformation functions have been defined for each pair of operations.

- **CoWord**

  An operational transformation approach for collaborative word processing has been proposed in [110], as part of the CoWord project. Word processing requires that editing is not only performed on text, but also on graphics and images. The proposed approach assumes that the objects in the document are addressable from a linear address space [122]. In addition to the *insert* and *delete* operations required for simple text processing, an *update* operation has been introduced in order to update the attributes of the objects subject to concurrent editing, such as the font or size of text or the colour of an object. In addition to the transformation functions between the

insert/delete operations, transformation functions between the *update* operation and the *insert/delete* operations have been proposed. Transforming an *insert* or *delete* operation against the *update* operation does not need any changes to that operation. Transforming an *update* operation against an *insert* or *delete* operation requires that the position of the *update* operation is adapted with respect to the insertion and deletion positions. To resolve the conflict between two *update* operations a novel technique called Multi-Version Single Display (MVSD) has been proposed. In this technique, concurrent *update* operations generate different versions of the target object, but only one version of the object is displayed. Priorities are assigned to *update* operations and the operation having the highest priority is executed and the corresponding version displayed. If the update operation having the highest priority is undone, the effect of the update having the next highest priority will be displayed.

- **Generalisation of OT for SGML**

  Operational transformation approach has been extended to documents conforming to a hierarchical structure. In [21] an approach for real time collaborative editing of documents written in dialects of SGML(Standard General Markup Language) including XML and HTML has been proposed. The architecture of the proposed system is replicated, each site maintaining a copy of the shared document. In the proposed hierarchical model a minimum number of nodes have an associated name, the other nodes being addressed by the path from their nearest named ancestor. The operations that can be performed are insertion of a subtree as a child of a specified node, deletion of a subtree and modification of the contents of a node. When a subtree is deleted, it is not destroyed since concurrent operations might refer to the deleted subtree. The deleted tree is excised as another subnode tree whose root is given a name. At a certain site, the deleted tree is garbage collected after all operations concurrent with the delete operation have been received at that site. Transformation functions have been defined for each pair of operations. These transformation functions are quite complex since operations can refer to any node in the document tree and different results have to be generated according to the relationships between paths of target nodes of operations. GOTO algorithm was used to maintain consistency.

  The approach presented in [21] uses a single history buffer and operations are not associated with the structure of the document. If

operations were associated with the structure of the document, flexibility of the definition and resolution of conflicts would be increased, as conflicting operations could be defined with reference to the nodes they target in the tree. Moreover, concurrency would be increased. In the approach proposed in [21], when an operation has to be transformed, the whole history buffer is scanned and transformations are performed. If the history of operations is distributed throughout the tree, when a remote operation has to be executed only the histories associated with a certain path in the tree are scanned and transformations performed.

In [21] the intention of users is not formally defined, the intention of operations being implicitly included in the transformation functions. However, the exclusion transformations between operations have not been defined and the set of all transformations is needed in order to determine the correctness of the transformation functions.

The approach proposed in [21] is theoretical and, to our knowledge, no system based on it has been implemented.

- **SAMS**

  Another operational transformation approach to real-time collaborative editing of hierarchical documents such as XML or CRC cards has been implemented in the SAMS (Synchronous, Asynchronous and Multi-Synchronous System) environment [69]. The environment allows the user to perform operations of creation and deletion of a new node, the creation and deletion of a certain attribute and the modification of an attribute through a graphical interface. By using a graphical interface for the insertion of elements an implicit formatting of XML nodes is performed. Therefore, it is not possible for the user to define their own formatting style by using separators between elements. If a node has to be modified, this operation has to be performed in two steps using the graphical interface: the node has to be deleted and a new node has to be inserted with the modified value. Moreover, for text nodes a lower granularity does not exist, such as words or characters. Therefore, each time a text node is modified it has to be deleted and the new value has to be inserted. The algorithm for maintaining consistency that was used is the SOCT4 algorithm. Transformation functions are defined between each type of operation. In the case of two concurrent *modify* operations applied to a certain attribute, the transformation function chooses the maximum value

that is set by the two operations.

Beside real-time communication, the SAMS environment offers support for asynchronous communication, by allowing the user to work in isolation and committing the changes at a later time.

Due to the fact that operations are not associated with the structure of the tree, support for definition and resolution of conflicts is limited and efficiency decreased as shown in section 3.3.

- **File synchronizer**

  In [68], the authors propose an operational transformation approach which defines a general algorithm for synchronizing file systems and file contents. File systems have a hierarchical structure, however, to merge text documents, the authors propose using a fixed working unit, i.e. a block unit consisting of several lines of text. The operations that work on the block units are: addblock, deleteblock and moveblock. Transformation functions are defined for each pair of operations.

2.4 Summary

In this chapter we analysed known approaches to consistency maintenance by classifying them into pessimistic and optimistic approaches. From the family of pessimistic approaches we presented the turn-taking protocols, non-optimistic locking and access control protocols. We classified optimistic approaches into social protocols, optimistic locking, validation techniques, approaches that require human intervention, serialisation, multiversioning, reconciliation mechanisms based on the merging of states, reconciliation based on constraints, and operational transformation mechanisms. We saw that operational transformation is a suitable approach for maintaining the consistency in collaborative systems, as it offers good responsiveness. However, operational transformation has been mostly applied to the consistency maintenance of linearly structured documents with the exception of the approaches proposed in [21] and [69] which work for documents conforming to an XML-like structure. However, the approaches that have been proposed for hierarchical documents use a single history buffer, as in the linear-based operational transformation approaches, and the nodes in the tree are not directly related to the operations that refer to them. This results in limited solutions for definition and resolution of conflicts.

We saw that there is a need to offer collaboration over documents with a complex structure and that existing solutions do not offer good support for flexible definition and resolution of conflicts. In this thesis, we propose a multi-level editing approach for maintaining consistency over hierarchical structured documents. In the tree-model of a document, a customisable approach to the definition and resolution of conflicts is offered, the users being allowed to work at different units of granularity corresponding to the different levels of the tree. Our multi-level editing approach increases support for concurrency as two operations are considered in conflict only if they target a common node in the tree. When a remote operation has to be executed, only the histories distributed along a certain path in the tree have to be scanned and transformations performed, as operations belonging to two nodes that are on different branches of the tree are commutative and they do not need transformations. The number of transformations that have to be executed is significantly decreased compared to approaches that use a linear history buffer where the entire history has to be scanned and transformations performed.

In this thesis we proposed mechanisms used in maintaining consistency over various classes of documents – text, XML and graphical documents and over two modes of collaboration – real-time and asynchronous communication over a shared repository. In the next chapter we present our treeOPT multi-level editing approach that can be directly applied to real-time collaboration over structured documents and its adaptation asyncTreeOPT for the asynchronous communication over a shared repository.

3
treeOPT Approach

In this chapter we present our multi-level editing approach to the maintenance of consistency in hierarchically structured documents. We first present a representation of the collaborative world, i.e. a model of a document and of the operations exchanged during collaboration. We then present our general principles for consistency maintenance. Afterwards we describe our approach, called treeOPT, which maintains consistency over documents conforming to a hierarchical structure, such as text and XML. We then present an adaptation of treeOPT to the asynchronous communication over a a shared repository. For the ease of presentation, throughout this chapter we illustrate an application of our approach to text documents. In chapter 5 we will present how our approach has been applied to XML documents. Finally, in this chapter, we relate our treeOPT approach to other existing approaches in the field.

3.1 Representation of collaborative world

The initial aim of the World Wide Web was to support the model of one author publishing to many readers [13]. Nowadays, the tendency exhibited by the Web is towards multiple interacting authors. Web Distributed Authoring and Versioning [25] tools and wikis [61] offer coarse-grained collaboration over the Web. Currently, these systems support concurrent editing of different documents, but do not support the editing of the same document.

Our goal was to investigate concurrent editing over the same document. The classes of documents that we investigated conform to a tree structure and they include XML (Extensible Markup Language) documents. We analysed consistency maintenance mechanisms both in synchronous and asynchronous modes of collaboration.

Apart from XML, a popular format for marking up various kinds of data from web content to data used by applications, the hierarchical model encompasses various other types of documents. Text documents can be seen as containing a list of paragraphs, each paragraph containing a list of sentences, each sentence being formed by a list of words and each word containing a list of characters. Therefore, text documents can be modelled by using a tree structure. Books, larger text documents, are formed by chapters, sections, paragraphs, sentences, words and characters. We also looked at the representation of graphical documents: a scene of objects can be modeled by a hierarchical structure: groups are represented as internal nodes, while simple objects represented as leaves. A group can contain other groups or simple objects. If a graphical document is expanded over more pages, the document contains an additional upper level, the pages, i.e. the document is formed of pages, groups, subgroups and objects.

Word processing documents containing not only simple text, but also graphical images such as Word documents, can be seen as conforming to a hierarchical structure, too. A document is seen as divided into pages, paragraphs, sentences, words, characters. Depending on its position in the document, a graphical image can belong to any of these granularity elements. For instance, if the image is drawn inside a paragraph, it is considered as belonging to the structure of that paragraph and if it is drawn between two paragraphs, then it is considered to be an element at paragraph level, as a direct child of a page. If the image belongs to a paragraph, the exact position of the image inside the paragraph is determined by recursively traversing the paragraph subtree. If the image is drawn inside a sentence, it is considered as belonging to that sentence and if it is drawn between two sentences, then it is considered to be an element at sentence level, as a direct child of the paragraph. The recursion is similarly applied at the word level. The scene itself can then be decomposed into groups, subgroups and objects in the same way as it has been done for a simple graphical document. Source code documents written in an object-oriented programming language also conform to a tree structure: a document contains a sequence of classes, each class contains a sequence of fields and methods and each method contains a sequence of lines of code.

In order to obtain good responsiveness, a replicated architecture was

used. Our aim was to find a general and efficient approach for maintaining the consistency of the copies of the documents. Since the hierarchical model is a suitable representation for a large class of documents, our focus was to propose a mechanism for maintaining the consistency of documents conforming to a tree structure. Operational transformation is used since it is a suitable approach for maintaining consistency in a replicated system. Local operations are executed on the local copy of a document immediately after their generation and remote operations need to be transformed against previously executed operations. We developed a novel consistency maintenance approach relying on the operational transformation mechanism applied to hierarchical documents.

We next present the representation of the document and of the operations carried out during collaboration.

3.1.1 Document model

Almost all operational transformation algorithms presented in section 2.3.9 keep a single history of operations that have been executed in order to compute the proper execution form of new operations. When a new remote operation is received, the whole history needs to be scanned and transformations need to be performed, even though different users might work on completely different sections of the document and do not interfere with each other. Keeping the history of all operations in a single buffer decreases efficiency. The existing algorithms for integrating a new causally ready operation into the history have a complexity of order n^2, where n is the size of the examined history buffer (for example GOT, GOTO and SOCT2), as it will be shown in section 3.3. Exceptionally, the dOPT algorithm has a complexity of order n, but convergence of copies is not always achieved. Consequently, a long history results in a higher complexity. This complexity negatively affects response time, which is a factor of critical importance in real-time editing systems.

Instead of keeping the history of operations in a single buffer, our idea was to distribute the history throughout the tree, and, when a new operation is transformed, only the history distributed on a single tree path is scanned.

We present in what follows our definition for a node of a hierarchical structure.

Definition 3.1.1 *A node N of a document is a structure of the form* $N =<$parent, children, length, history, content$>$, *where*

- parent *is the parent node for the current node. Except for the topmost node, parent is a valid reference to a node in the tree.*
- children *is an ordered list* $[child_1, ..., child_n]$ *of child nodes*
- length *is the length of the node in terms of the number of leaf nodes*

$$\text{length}= \begin{cases} 1, & \textit{if } N \textit{ is a leaf node} \\ \sum_{i=1}^{n} \text{length}(child_i), & \textit{otherwise} \end{cases}$$

- history *is an ordered list of operations executed on child nodes*
- content *is the content of the node, defined only for leaf nodes*

$$\text{content}= \begin{cases} aCharacter, & \textit{if } N \textit{ is a leaf node} \\ \sum_{i=1}^{n} \text{content}(child_i), & \textit{otherwise} \end{cases}$$

The level of a node is the height of the node, i.e. the length of the path from the root to the node.

In what follows we are going to show how definition 3.1.1 is applied to text and XML documents.

For a text document consisting of paragraphs, sentences, words and characters, the following levels are defined: document (level 0), paragraph (level 1), sentence (level 2), word (level 3) and character (level 4). In the case of text documents we refer to these levels as granularity levels, document being the highest granularity level and character being the lowest granularity level.

Consider for instance the following text document.

> We present a consistency maintenance algorithm relying on a tree representation of documents. The hierarchical representation of documents is a generalisation of the linear representation and, in this way, our algorithm can be seen as extending the existing OT algorithms.
> We present our approach that can be applied to the real-time and asynchronous modes of collaboration. The algorithm applies the same basic mechanism as existing OT algorithms for synchronous or asynchronous collaboration recursively over the different document levels. The algorithm is general. It can use any of the OT algorithms relying on a linear representation.

Figure 3.1 illustrates the following information about the structure of the above document. The document consists of two paragraphs and the second paragraph contains four sentences. The third sentence contains four words, the second word of the sentence being *"algorithm"*. Each node in the document except the leaf nodes has a history buffer containing the operations performed on the child nodes. For instance, the document history

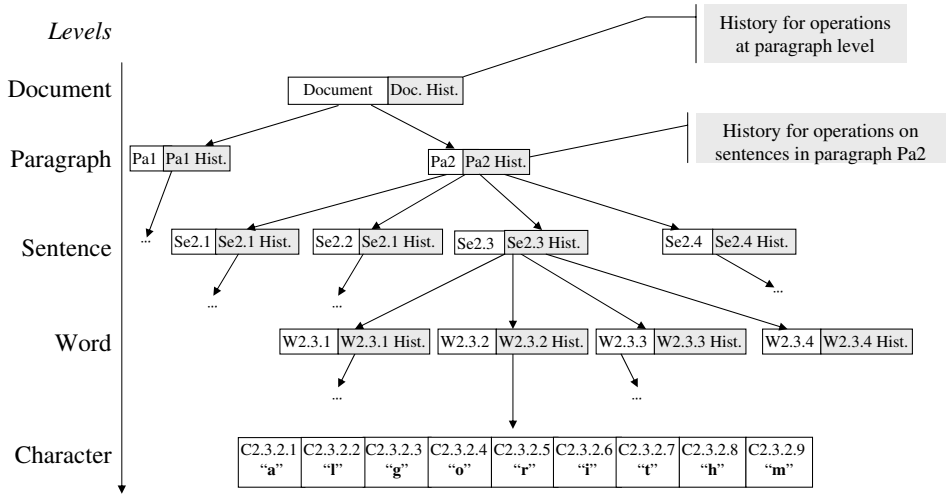


Figure 3.1: Structure of a text document

contains the operations that have been executed on whole paragraphs such as insertions and deletions of paragraphs and the history associated with paragraph $Pa2$ contains the operations that have been performed on whole sentences belonging to paragraph $Pa2$.

Next, let us consider the following example XML document.

```
<movieDB>
   <movie title="21 Grams" year="2003">
      <director>Alejandro González Iñárritu< /director>
      <actor>Sean Penn< /actor>
      <actor>Naomi Watts< /actor>
   < /movie>
   <movie title="Mar adentro" year="2004">
      <director>Alejandro Amenábar < /director>
      <actor>Javier Bardem< /actor>
      <actor>Belén Rueda< /actor>
      <actor>Lola Dueñas< /actor>
   < /movie>
< /movieDB>
```

A tree representation of this document is shown in figure 3.2. Note that the attributes of a node are considered children of that node, but are kept in a separate list than the list of child elements.

Due to the tree structure of the document, different semantic levels are associated with the levels of the document, lower granularity elements being defined in the context of higher granularity nodes. If a user deletes

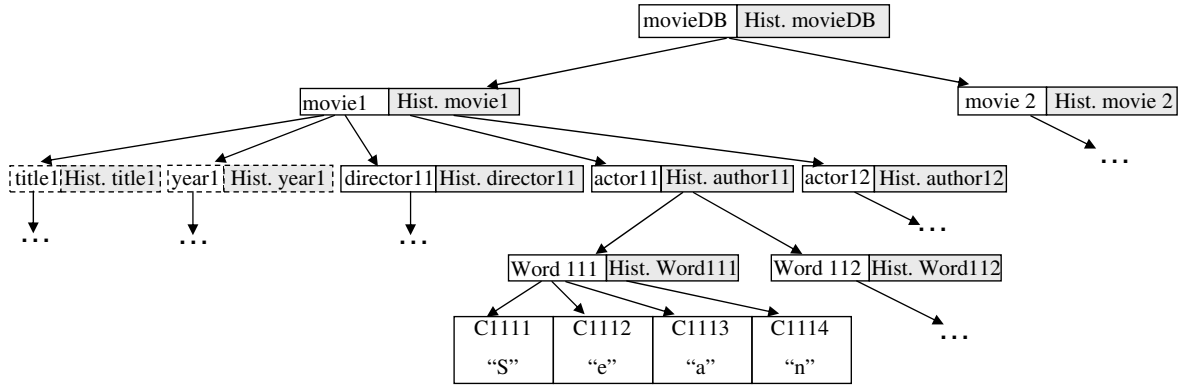


Figure 3.2: Structure of an XML document

an element of a higher granularity level, concurrent changes performed by other users targeting subnodes in the deleted tree will not be executed. This is the right decision to be taken from a semantics point of view, since the existence of a node of a lower granularity level does not make sense if one of its ancestor nodes is deleted. Consider, for instance, that a user is deleting a sentence, while concurrently another user is inserting a word in that sentence. From a syntactical point of view, respecting the intentions of both users results in the sentence deletion followed by the insertion of the word in the empty sentence. But, semantically this is not a correct result, since a word in an empty sentence loses its meaning. In the same way, an XML document might not conform to its DTD (Document Type Definition) or schema describing its structure if a subelement is kept while the parent element is deleted.

3.1.2 Operation representation

In this section we show how operations carried out during the collaboration are represented.

Operations exchanged between sites during collaboration refer to parts of the hierarchical structure of the document corresponding to different levels of the document. We call these operations composite operations, in order to distinguish them from regular operations defined for the linear structure of the documents.

Definition 3.1.2 *A composite operation is a structure of the form cOp=* <type, position, content, stateVector, initiator>*, where*

- type *is the type of the operation*

- position *is a vector of positions specifying the path starting from the root to the node where the operation has been applied*
- content *is a node representing the content of the operation*
- stateVector *is the state vector of the generating site*
- initiator *is the initiating site identifier*

The level of an operation is the level of the node in whose history the operation is kept. For instance, in a text editing application where the levels of the document are the document (level 0), the paragraph (level 1), the sentence (level 2), the word (level 3) and the character (level 4), an *insertParagraph* operation belongs to the document history and is of level 0, an *insertSentence* operation is of level 1, an *insertWord* operation is of level 2 and an *insertChar* operation is of level 3.

In a simple text editing application where a document is composed of paragraphs, sentences, words and characters, two types of operations have been used: insertion and deletion. A vector position associated with an operation specifies the positions for the levels corresponding to a coarser or equal granularity than the granularity of the operation. For example, if we have an insertion operation of word level (level 3), we have to specify the paragraph and sentence in which the word is located, as well as the position of the word within the sentence. For the sake of simplicity, in future examples concerning text and XML editing, we denote operations by specifying only their type, position and content of the node, ignoring other attributes. For example, consider the text document represented in Figure 3.1. Suppose we want to modify the third sentence in the second paragraph by adding the word *"treeOPT"* as the second word in the sentence, as shown in figure 3.3. In this case, the operation *insertWord(2,3,2,"treeOPT")* denotes an operation of type *insertion*, that is of word level and has to be applied to paragraph 2, sentence 3, at word position 2 inside the sentence, and has as content a node of type word represented by the string *"treeOPT"*. Further, suppose we want to modify the second paragraph by adding the new sentence *"The approach was applied to text and XML documents."* as the third sentence in the paragraph, as shown in figure 3.3. The representation of the insert operation would be *insertSentence(2,3,"The approach was applied for text and XML documents.")* denoting an operation of type *insertion*, that is of sentence level and has to be applied to paragraph 2, position 3 in the paragraph and has as content a node of type sentence

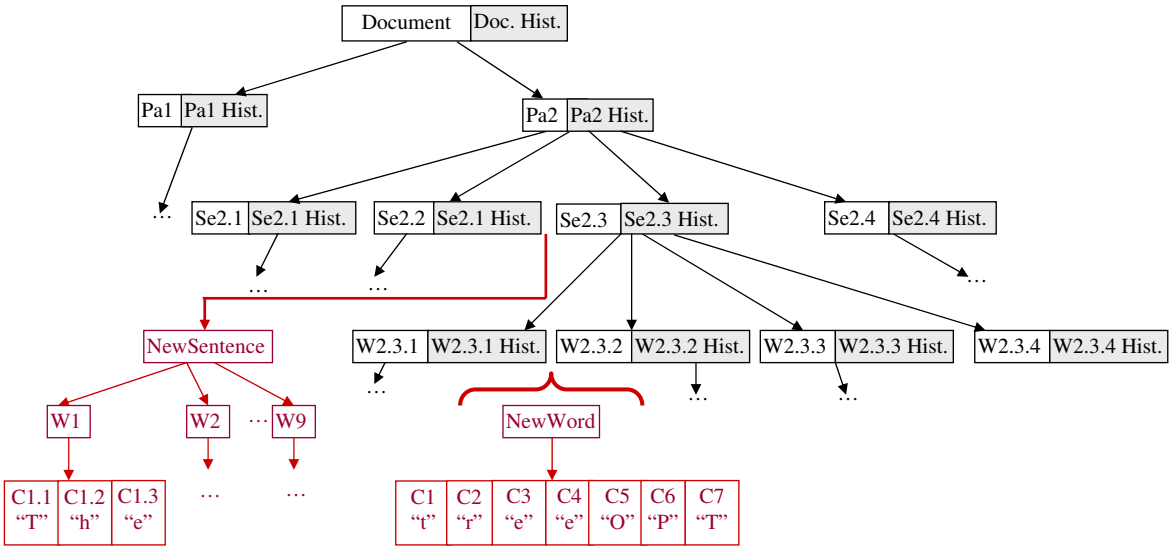


Figure 3.3: Examples of operation instantiation

represented by the string *"The approach was applied for text and XML documents."*. The nodes that are inserted have an empty history buffer.

In what follows we give an example of operation instantiation for the case of XML documents. Consider the document illustrated in figure 3.2. Suppose that we want to delete the second child element of the first *movie* element child of *movieDB*, i.e. element `<actor>Sean Penn</actor>`. The operation describing this action is *deleteElement(1,1,2)*.

For the editing of the XML documents, in addition to the operations of insertion and deletion of elements in the tree, a set of insert and delete operations have been defined for the processing nodes, attributes, words, separators and characters, as well as operations for the insertion and deletion of the closing tags, as described in chapter 5.

3.2 Principles of consistency maintenance

We saw in the previous chapter the main issues in collaborative editing, i.e. divergence, causality and intention violation.

Before presenting the principles of consistency maintenance, we describe in detail the notion of intention.

We distinguish three types of intentions: operation intention, user intention and group intention.

Suppose a document consists of the string *"sar"*. The execution of an operation transforms the string of characters into *"star"*. This change can be interpreted as an insertion of character *"t"* at position 2 or an insertion

of character *"t"* between *"s"* and *"a"* or an insertion of character *"t"* after *"s"* or an insertion of character *"t"* before *"a"*. Any of these interpretations can be considered as the *intention of the operation* of adding character *"t"* to the string.

A *user intention* is a different concept from the operation intention, as the intention of a user might be expressed by a set of operations. For instance, for the text editing application, an operation of replacing the word *"customizable"* with the word *"customisable"* is mapped to a pair of operations - delete operation of word *"customizable"* followed by the insertion of word *"customisable"*. For the graphical editing application, suppose that a user edits the entity relationship diagram in Figure 3.4.

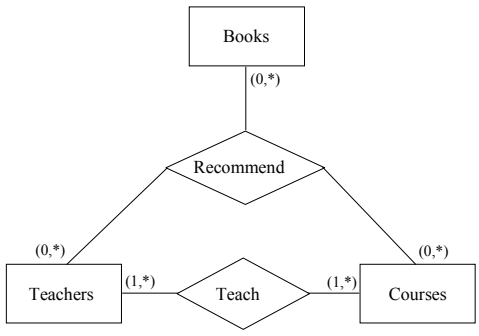


Figure 3.4: Entity relationship - ternary relation

Suppose that the user wants to transform the ternary relationship into a binary relationship, by moving the entities and the relations between them as shown in the figure 3.5.

Figure 3.5: Entity relationship - binary relation

The sequence of operations composing the intention of the user are the moving of the entities *Teachers*, *Courses* and *Books*, of the associations *Teach* and *Recommend* and of the lines connecting them.

As the intention of user is hard to be deduced from the content of a single operation, we do not consider the preservation of user intentions, but the preservation of operation intentions.

By group intention we understand the combined effect of the intentions of operations performed by a group of users. For instance, in the graphical editing application, if one user deletes a part of the text and another user inserts inside the deleted part of the text, the combined effect could be

the deletion of the text to be deleted and the insertion of the new text. For instance, suppose the shared text document consists of the sentence *"Version control systems are used to support a group of people working together on a set of documents over a network."* and one user deletes the part of the sentence *"are used to "* in order to obtain *"Version control systems support a group of people working together on a set of documents over a network."*, while the other user concurrently inserts the word *"widely "* in order to obtain *"Version control systems are **widely** used to support a group of people working together on a set of documents over a network."*. The combined effect of the two concurrent operations could be *"Version control systems **widely** support a group of people working together on a set of documents over a network."*.

Another combined effect could be the deletion of the text *"are used to "* without the insertion of the word *"widely "*, the result being *"Version control systems support a group of people working together on a set of documents over a network."*.

In graphical editing consider the scene of objects illustrated in the left part of figure 3.6. Suppose two users concurrently edit this scene.

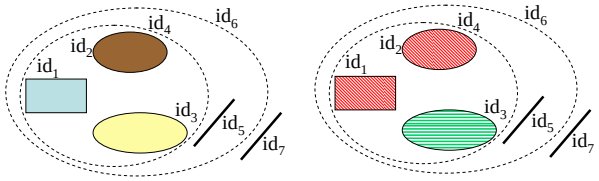


Figure 3.6: Graphical scene of objects; left - initial scene of objects; right - the combined effect of the concurrent operations $O_1 = chgColour(id_4, red)$ and $O_2 = chgColour(id_3, green)$

The first user wants to change the colour of the group of objects with the identifier id_4 to *red*, performing the operation O_1 =*chgColour(*id_4*,red)* while the second user concurrently wants to change the colour of the object having the identifier id_3 to green, performing the operation O_2 = *chgColour(*id_3*,green)*. The two operations conflict because they both target object id_3, one setting its colour to red and the other changing its colour to green.

One of the combined effects of the two concurrent operations would be to change the colour of the object id_3 to *green* and the colours of the objects id_1 and id_2 to *red*, as shown in the right part of figure 3.6. Another combined effect would be to execute just O_2 and to set the colour of object id_3 to *green*, or to execute just O_1 and to set the colour of group id_4 to red.

Preservation of intentions of operations at all sites does not mean that convergence is ensured. For instance, suppose that the shared document is the string *"sar"* and that two users concurrently modify the document as shown in Figure 3.7.

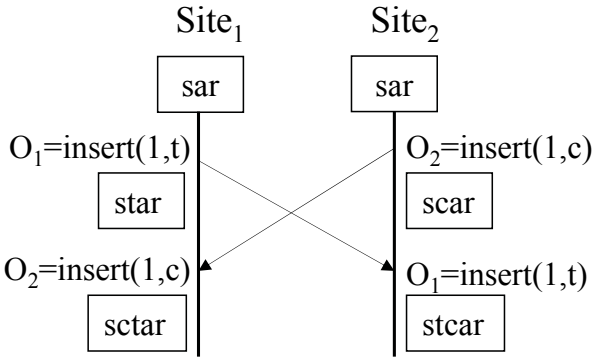


Figure 3.7: Preservation of intentions does not ensure convergence

Suppose that the first user at $Site_1$ performs the operation O_1=*insert(1, "t")* with the defined intention to insert *"t"* between *"s"* and *"a"*. Suppose that the second user at $Site_2$ performs the operation O_2=*insert(1,"c")* with the defined intention to insert *"c"* between *"s"* and *"a"*. When operation O_2 arrives at $Site_1$ it can be executed in its form as the intention of O_2 is that *"c"* is inserted between *"s"* and *"a"*. Therefore, the final state of the document at $Site_1$ is *"s**ct**ar"*. At $Site_2$, when operation O_1 arrives it can be executed in its form as the intention of O_1 is that *"t"* is inserted between *"s"* and *"a"*. The final state of the document at $Site_2$ is *"s**tc**ar"* which differs from the state of the document at $Site_1$. As we have seen, even if intentions of operations have been preserved, the copies of the documents might diverge.

Preserving operation intentions implies the definition of intentions for the individual operations, but also the definition of the group intention.

When a user invokes an operation, the operation is performed in a context seen by that user, i.e. the state of the document at the moment when the operation has been issued. But, when the operation is executed at another site, the generation context of the operation might be different than the execution context of the operation at that site. Transformation functions aim to express the preservation of the intention of an operation generated at one site when it is executed at other sites. Most of the existing operational transformation approaches specify a group intention by means of transformation functions. Transformation functions include the expression of the preservation of intention between pairs of operations. However, this is not enough for expressing a group intention. A group intention can-

not be defined by expressing the intention of each operation with respect to each concurrent operation, but it has to relate the intention of an operation according to the set of concurrent operations or to the structure of the document, i.e. it has to be related to a global structure of the collaborative environment.

Most approaches related the intention of an operation in turn with respect to other operations. An exception to this rule is the approach proposed in [63] where preserving the intention of an operation means preserving the operation effects relation, i.e. if an insert of character c is performed between two characters c_1 and c_2 in a document, preserving the intention of the operation means to preserve the order between the characters c, c_1 and c_2 (c_1 must precede c and c must precede c_2).

We consider that group intention cannot be preserved by only using operational transformation, but by offering the user a mechanism to define how intentions can be preserved. For instance, a locking mechanism allows the user to lock parts of the document such that no concurrent changes are allowed in the locked parts. Another mechanism for intention preservation is to let the user specify rules for defining and resolving conflicts. By using a structured document, rules can be defined referring to different units corresponding to the levels of the document. For instance, a rule could be easily defined to specify that in the case that two users concurrently change the same word the intentions are violated and the conflict could be resolved, for example, by executing none of the concurrent operations or executing only the operation performed by the user having the highest priority.

In what follows we present the principles of consistency maintenance adopted by different systems.

As we have seen, most of existing operation transformation approaches such as dOPT, Jupiter, NetEdit and adOPTed did not introduce operation intention to maintain consistency. These approaches considered only convergence and causality preservation as principles for maintaining consistency. Convergence requires that all copies of the same document are identical after the execution of the same set of operations. Causality preservation requires that, for any pair of operations O_a and O_b, if O_a precedes O_b ($O_a \rightarrow O_b$), then O_a is executed before O_b at all sites.

In [108] the CCI (Convergence, Causality and Intention) consistency model has been proposed consisting of the preservation of convergence, causality and intention. Intention preservation was for the first time defined in [108]. Intention preservation was defined as requiring that, 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.

To achieve causality preservation, most operational transformation approaches use a timestamping scheme based on state vectors [26].

To achieve convergence, GOT algorithm uses a total ordering relation between operations, as explained in subsection 2.3.9.

To achieve intention preservation, inclusion and exclusion transformation functions or forward and backward transformation functions, see section 2.3.9, have been defined.

The authors of GOTO and SOCT2 approaches claim that their algorithms will ensure convergence and preserve intention if the transformation functions satisfy C_1 and C_2. However, due to the fact that intention was not formally defined, it cannot be checked that transformation functions preserve intention.

Based on the operation effects relation, as presented in section 2.3.9, the CSM consistency model has been proposed in [63] consisting of the following properties:

- **Causality preservation** with the same meaning as in the CCI model [108]
- **Single-operation effects preservation** requiring that the effect of executing any operation in any execution state achieves the same effect as in its generation state
- **Multi-operation effects relation preservation** requiring that the effects relation of any two operations is maintained after they are both executed in any states

The preservation of operation effects achieves convergence and intention-preservation.

Most operational transformation algorithms deal only with syntactic consistency and they did not deal with semantic consistency. The approach proposed in [98] deals with semantic consistency by introducing integrity constraints in the transformational approach, such as checking that no two nodes of a CRC card description have the same name.

Let us analyse the principles of consistency maintenance that have been used in our approach.

Causality preservation is maintained by using state vectors.

Convergence and intention preservation are achieved in the case of text and XML documents by using the operational transformation mechanism

applied to hierarchical document structures. In our approach we recursively apply an operational transformation algorithm conforming to a linear structure over the document levels. Therefore, we adopt the definition of intention preservation specific to the chosen linear operational transformation approach. Due to the tree structure, as mentioned above, a special case arises in the preservation of user intentions: if a user deletes an element of a higher granularity level, concurrent changes performed by other users targeting subnodes in the deleted tree will not be executed. But this restriction could be regarded as a way of maintaining semantical consistency in that lower granularity elements are defined only in the context of higher granularity nodes. In terms of intention preservation we give the user an option to specify a set of user defined rules for the definition and resolution of conflicts.

In a graphical document, convergence and intention preservation are of a different nature and we are going to describe them in chapter 6.

3.3 The treeOPT algorithm

In this section we present our treeOPT [37, 38] operational transformation algorithm working on the hierarchical structures of documents. After theoretically presenting the treeOPT approach in subsection 3.3 we present in subsection 3.3.2 an adaptation of treeOPT using various linear operational transformation algorithms that were recursively applied to the tree structure of the document. We then describe in subsection 3.3.3 the problem of splitting and merging of elements and the solution that we adopted.

3.3.1 Description of the algorithm

In what follows we give an intuitive explanation of the algorithm, and afterwards describe it formally. As already mentioned, each site stores locally a copy of the hierarchical structure of the shared document. For a leaf node, the content of the node is explicitly specified in the content field. For nodes situated higher in the hierarchy, the content field will remain unspecified, but the actual content of each node will be the concatenation of the contents of its children. Each node (excluding leaf nodes) will keep a history of insertion and deletion operations associated with its child nodes.

The principles of the algorithm are described in what follows. Each site can generate composite operations, representing insertions or deletions of subtrees in the document tree. Note that each node of a subtree to

be inserted has an empty history buffer. The site generating a composite operation executes it immediately. The operation is also recorded in the history buffer associated with the parent node of the inserted or deleted subtree. Finally, the new operation is broadcast to all other sites, being timestamped using a state vector. Upon receiving a remote operation, the receiving site will test it for causal readiness. If the composite operation is not causally ready, it will be queued, otherwise it will be transformed and then executed.

We explain next in an intuitive way how the treeOPT approach is applied for a text document consisting of paragraphs, sentences, words and characters. We illustrate the way transformations are performed using an example.

Consider the initial state of the document illustrated in Figure 3.8.

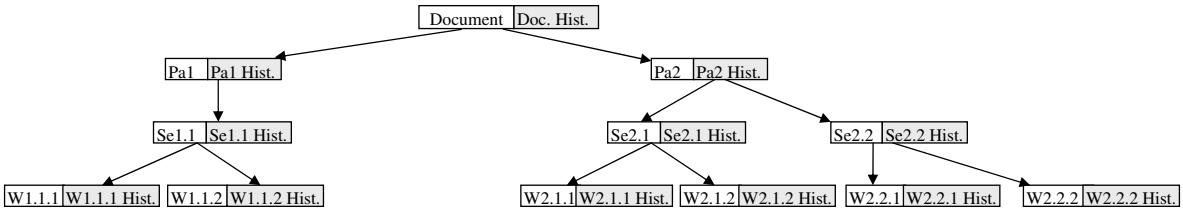


Figure 3.8: Initial State of the Document

The document consists of two paragraphs, the first paragraph containing a sentence with two words and the second paragraph containing two sentences, each sentence being composed of two words. Suppose two users concurrently edit this document. At $Site_1$, $User_1$ inserts the word *"algorithm"* in paragraph *2*, sentence *2*, as the *2*nd word, by issuing the operation *insertWord(2,2,2,"algorithm")*, as shown in Figure 3.9.

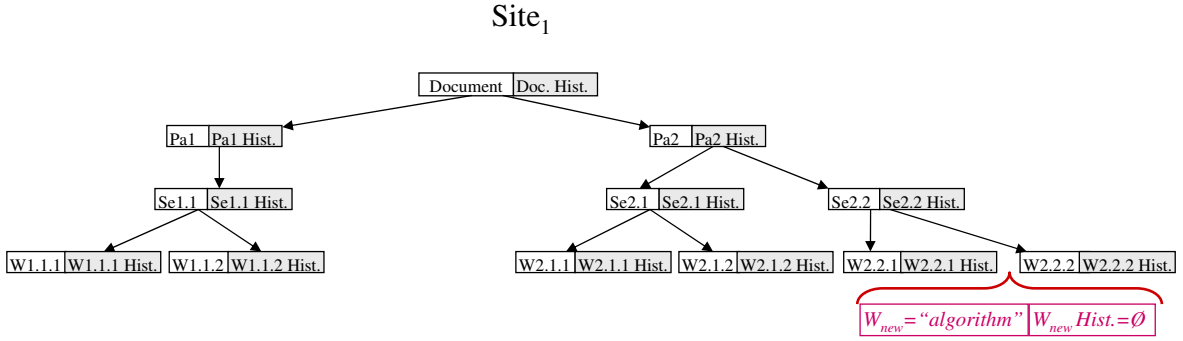


Figure 3.9: State of the Document at Site 1, before the insertion of word W_{new}

Suppose that at $Site_2$, $User_2$ inserts a new paragraph as the second

paragraph in the document and afterwards deletes the first sentence of the third paragraph as shown in figure 3.10.

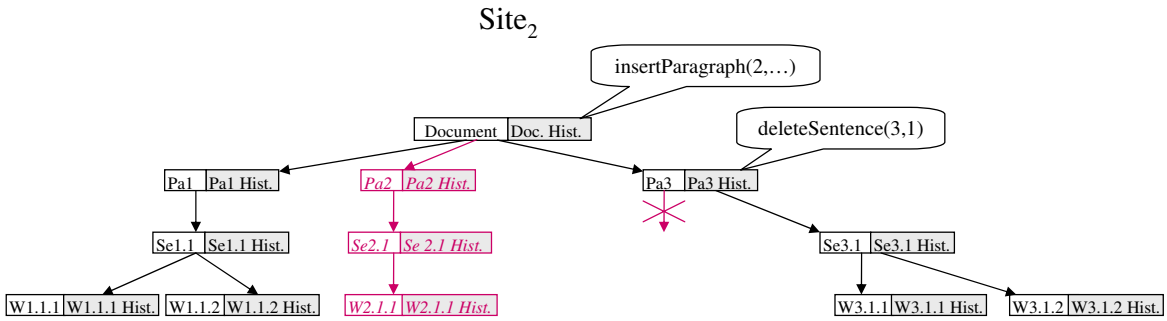


Figure 3.10: State of the Document at Site 2

Let us describe how the operation issued by the user at $Site_1$ is transformed when it arrives at $Site_2$ against the operations performed at $Site_2$ as illustrated in figure 3.11.

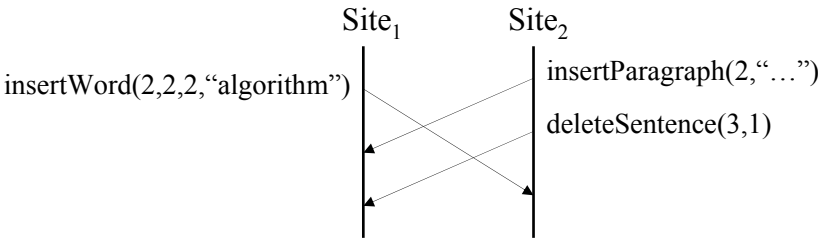


Figure 3.11: Sequence of operations

First of all, we consider the paragraph number specified by the remote composite operation issued by $User_1$, which in this case is equal to *2*. We do not know for sure that paragraph number *2* of the local copy of the document at $Site_1$ is the same paragraph as that referred to by the original operation. Due to the fact that a concurrent operation issued by $User_2$ inserts a whole new paragraph before paragraph *2*, the word *“algorithm”* should be inserted not in paragraph *2*, but in paragraph *3*. Therefore, the remote operation must be transformed against previous operations involving whole paragraphs, which are kept in the document history buffer. This can be done using any existing operational transformation algorithm working on linear structured documents, such as GOT(O) or SOCT2 algorithms. After performing these transformations, we obtain the position of the paragraph in which the operation has to be performed, paragraph number *3* in our example, as shown in the upper part of figure 3.12.

Consequently, the new composite operation will become *insertWord(3,2,2,“algorithm”)*. Here it is important to note that previous concurrent

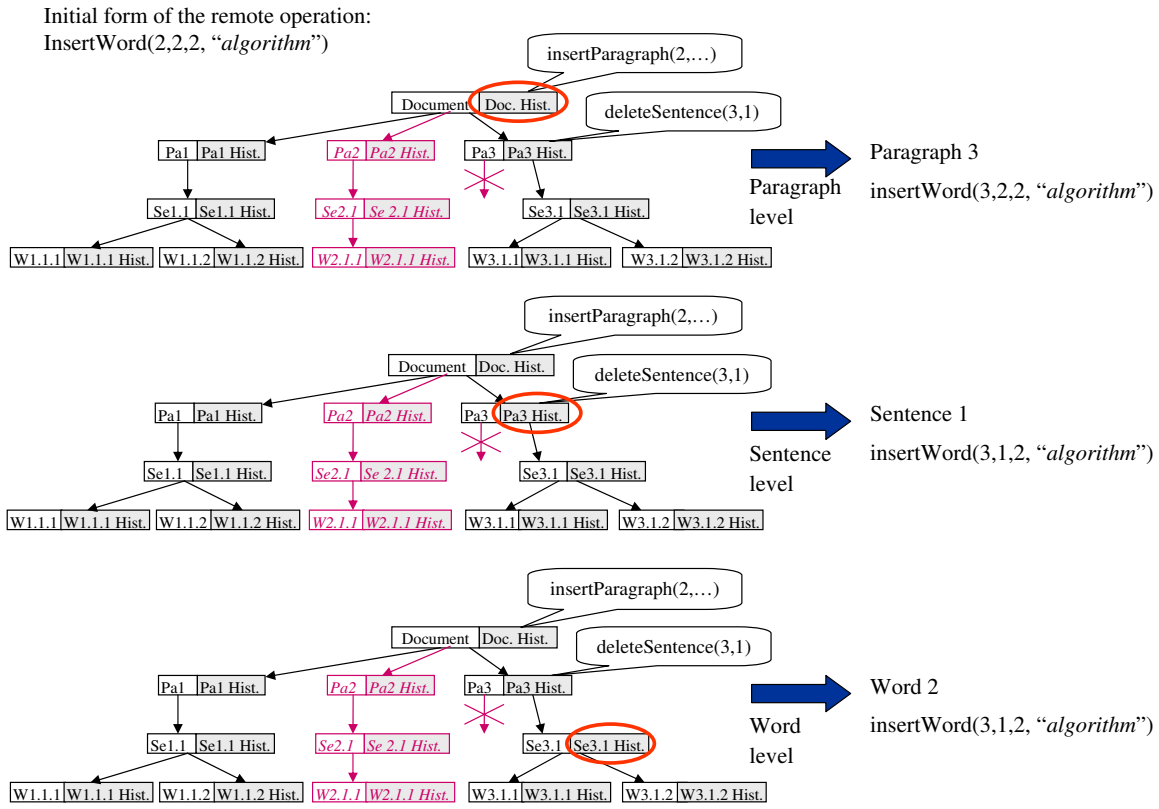


Figure 3.12: Steps for the transformation of the remote operation

operations of finer granularity are not taken into account by these transformations as the document history buffer contains only operations at paragraph level. Indeed, we are not interested in whether another user has just modified another paragraph, because this fact does not affect the number of the paragraph where the word *"algorithm"* has to be inserted. The next step obtains the correct number of the sentence where the word has to be inserted. Therefore, the new operation is transformed against the operations belonging to *Pa3 History*. *Pa3 History* only contains insertions and deletions of sentences that are children of paragraph *3*. We again apply an existing operational transformation algorithm, and obtain the correct sentence position. Because a concurrent delete operation of sentence 1 in paragraph 3 has been performed, the word *"algorithm"* should be inserted in sentence 1 and not sentence 2. The form of the operation becomes *insertWord(3,1,2,"algorithm")* as shown in Figure 3.12.

The algorithm continues by obtaining the correct word position in the same manner. Because there are no concurrent operations of insertion of words in sentence 1 of paragraph 3, the final form of the transformed operation is *insertWord(3,1,2,"algorithm")*, as shown in figure 3.12.

Finally, the operation can be executed and recorded in the history. As

it is an operation of word level, it must be recorded in the history associated with the parent sentence. As we can see, the algorithm achieves consistency by repeatedly applying an existing concurrency control algorithm on small portions of the entire history of operations, which, rather than being kept in a single linear structure, is distributed throughout the tree.

We now present the general form of the treeOPT algorithm.

Algorithm *treeOPT(O, RN, L):O* {
  CN:=RN;
  for *(l:=1;l≤L;l++)* {
    O_{new}:=*composite2Simple(O,l)*;
    EO_{new}:=*transform*(O_{new},*history(CN)*);
    position(O)[l]:=position(EO_{new});
    if *(level(O)=l)*
      return O;
    CN:=$child_i(CN)$, where *i=position*(EO_{new});
  }
}

Given a new causally ready composite operation, O, the root node of the hierarchical representation of the local copy of the document, RN, and the number of levels in the hierarchical structure of the document, L, the execution form of O is returned. In the case of the text editor, $L = 4$ and RN=*document*. As we saw in the previous examples, determining the execution form of a composite operation requires finding the elements of the position vector corresponding to a coarser or equal granularity level than that of the composite operation. For each level of granularity l, starting with paragraph level and ending with the level of the composite operation, an existing operational transformation algorithm is applied to find the execution form of the corresponding regular operation. Traditional algorithms do not perform transformations on composite operations, but rather on regular ones. Therefore, we had to define the function *composite2Simple*, that takes as arguments a composite operation, together with the granularity level at which we are currently transforming the operation, and returns the corresponding regular operation. The operational transformation algorithm is applied to the history of the current node CN whose granularity level is $l - 1$. Recall that, for example, to find the corresponding paragraph position, transformations need to be performed against the operations kept in the document history. The lth element in the position vector will be equal to the position of the execution form of the regular operation. If the current granularity level l is equal to the level of the composite operation, the algorithm returns the execution form of the composite

operation. Otherwise, the processing continues with the next finer granularity level, with CN being updated accordingly. By *transform(O,HB)* we denote any existing concurrency control algorithm, that, taking as parameters a causally-ready regular operation O and a history buffer HB, returns the execution form of O. The implementation of the *transform* method depends on the chosen consistency maintenance algorithm working on a linear structure of the document. We tested the operation of our algorithm combined with the SOCT2 and GOT algorithm, details about the implementation of these algorithms can be found in subsection 3.3.2.

treeOPT is a general algorithm in that it can be applied to any document having a hierarchical structure. A trivial application would be the case of a book modelled as being composed of chapters, with each chapter consisting of sections, each section of paragraphs, each paragraph of sentences and so on. The application of the treeOPT algorithm for XML-like documents can be found in chapter 5.

We have to mention that due to the tree structure of the document, if a user deletes an element of a higher granularity level, concurrent changes performed by the other users targeting subnodes in the deleted tree will not be executed. This is the right decision to be taken from a semantic point of view, since the existence of a node of a lower granularity level does not make sense if one of its ancestor nodes is deleted.

An important advantage of the algorithm is related to its improved efficiency compared to the operational transformation approaches conforming to a linear structure.

The complexity of the concurrency control algorithms for a linear structure is usually of $O(n^2)$, where n is the length of the spanned history. In what follows, we analyse the complexity of GOT [108] and SOCT2 [101] algorithms. In appendix B we present the LIT and LET transformation functions of including and respectively excluding a list of operations from another list of operations. The complexity of these transformation functions is $O(p \cdot q)$, where p and q are the sizes of the two lists of operations. In section 2.3.9 we described the undo/do/redo scheme for the GOT algorithm. Operations from the history buffer $HB = [EO_1, \ldots, EO_m, \ldots, EO_n]$ are undone from right to left till an operation totally preceding the remote operation O_{new} is encountered. This step is of complexity $n - m$. Afterwards the GOT algorithm has to be applied. We are going to analyse its complexity later. When operations are redone, they have to exclude the effect of the previous operations in the history buffer and then include the effect of the remote operation and of the previous operations that have been redone. This step involves $1 + \sum_{i=2}^{n-m}(i-1) + i$ transformations and has

a complexity of $O((n-m)^2)$. The GOT algorithm consists in transforming operation O_{new} against the list $HB' = [EO_1, EO_2, \ldots, EO_m]$ whose operations precede in total order O_{new}. The list *HB'* is scanned from left to right to find the first operation that is concurrent with O_{new} and afterwards to find the list of operations following the concurrent operation that are preceding the remote operation. This step requires m comparisons. Some operations in *HB'* causally precede O_{new} and some others are concurrent with O_{new}. Let $EOL = [EO_{c1}, EO_{c2}, \ldots, EO_{cr}]$ be the list of operations that causally precede O_{new}. In order to exclude the list of operations EOL from O_{new}, the operations from EOL have to exclude the effect of the concurrent operations with O_{new}. If we denote by $EOL' = [EO'_{c1}, EO'_{c2}, \ldots, EO'_{cr}]$ the list of transformed operations, then each element EO'_{ci} is obtained by first excluding from EO_{ci} the effect of the list of operations from *HB'* situated at its left side starting with index k and then including $[EO'_{c1}, EO'_{c2}, \ldots, EO'_{ci-1}]$. This step requires $(c_1 - k) + \sum_{i=2}^{r}(c_i - k)$ transformations which results in a complexity of $(r-k)^2$. Afterwards, the list EOL' has to be excluded from O_{new}, which requires r transformations and operation O_{new} has to include the operations from *HB* starting with index k, which requires $m - k$ transformations. Therefore, the complexity of the GOT algorithm is $O(m)$. And consequently the undo/do/redo procedure, i.e. the procedure of integration of a remote operation into the history buffer, has a complexity of $O(n^2)$, where n is the length of the history buffer.

In what follows we analyse the complexity of the SOCT2 algorithm presented in section 2.3.9. The SOCT2 algorithm for the integration of a remote history into the history buffer consists of two steps. In the first step of the algorithm, the operations in the history buffer are reordered such that the operations that precede the remote operation are situated before the operations concurrent with the remote operations. The reordering procedure considers each operation in the history buffer and if that operation precedes the remote operation, it is transposed towards the beginning of the history at the end of the sequence of operations that precede the remote operation. The second step of the algorithm consists in the transformation of the remote operation against the concurrent operations in the history buffer. The worst case scenario in the reordering process when the most number of transpositions have to be performed is the one when the operations that precede the remote operation are situated at the end of the history. Suppose there are m operations that precede the remote operation. Each of these operations has to be transposed against $n - m$ operations. In total there will be $m(n-m)$ transpositions to be performed.

Therefore, the process of reordering is of complexity $O(n^2)$, where n is the size of the history buffer and the maximum number of transformations is obtained if $m = n/2$. The second step of the SOCT2 algorithm of integrating the remote operation into the history buffer consisting of the forward transposition of the remote operation according to the sequence of concurrent operations is of order $O(n)$. Therefore, the complexity of the SOCT2 algorithm, i.e. of integration of a remote operation into the history buffer, is $O(n^2)$, where n is the length of the history buffer.

In our representation of the document, the history of operations is not kept in a single buffer, but rather distributed throughout the whole tree, and, when a new operation is transformed, only the history distributed on a single path of the tree will be spanned. This turns out to be a very important increase in speed. Moreover, when working on medium or large documents, operations will be localised in the areas currently modified by each individual user and these may often be non-overlapping. In these cases, almost no transformations are needed, and therefore the response time is very good. Recall the fact that in the case of algorithms working on linear structures, every operation interferes with any other, independently of the distance between the positions specified in the operations. Another important advantage is the possibility of performing, not only operations on characters, but also on other semantic units, such as words, sentences and paragraphs, in the case of the text documents, or element nodes, in the case of XML documents. The transformation functions used in the operational transformation mechanism are kept simple as in the case of character-wise transformations, not having the complexity of string-wise transformations as presented in [108]. An insertion or deletion of a node in the tree can be done in a single operation. Therefore, efficiency is further increased, because there are fewer operations to be transformed, and fewer to be transformed against. Moreover, the data is sent using larger chunks, thus the network communication is more efficient. Our approach also adds flexibility in using the editor, as the users are able to select the level of granularity they prefer to work on.

In what follows we are going to compare the complexity of the treeOPT algorithm with respect to the complexity of the operational transformation algorithms working on a linear structure, such as GOT and SOCT2 whose complexities we previously analysed.

The complexity of the treeOPT algorithm is the same as the complexity of the SOCT2 and GOT algorithms. The worst case occurs when the whole history is concentrated on a single element in the tree and the granularity of that element is the lowest existing granularity. The worst case corresponds

to the case when all users involved in the collaboration concurrently edit the same word in the document, by adding or removing characters belonging to that word. In this case the whole history of the document is the history attached to the node and it has to be traversed each time a transformation has to be performed. Therefore, the complexity is $O(n^2)$, where n is the number of operations performed, complexity that is the same as in the case of linear transformation approaches. But, the worst case rarely occurs in practice.

Therefore, in what follows we are going to analyse the complexity of the treeOPT algorithm in the general case where we consider that the tree document contains p operations of document level, i.e. insertions and deletions of paragraphs, s operations at paragraph level, i.e. insertions and deletions of sentences, w operations of sentence level, i.e. insertions and deletions of words and c operations of word level, i.e. insertions and deletions of characters. Further, suppose that we have the following average counters concerning the structure of the document. Suppose that sc is the average number of sentences in a paragraph, wc the average number of words in a sentence and cc the average number of characters in a word. If the remote operation is an operation referring to a paragraph, only the document history has to be traversed and transformations performed only at document level. A linear transformation algorithm is applied on the document history and therefore the complexity of the integration procedure is p^2. If the remote operation refers to a sentence, a linear operational transformation approach has to be applied for the document history and afterwards for the paragraph history corresponding to the paragraph where the sentence targeted by the remote operation belongs. Since there are s operations referring to sentences and sc is the average number of sentences in a paragraph, the number of operations associated with a paragraph is s/sc. Therefore, the complexity of the integration of a remote operation referring to a sentence is $p^2 + (s/sc)^2$. Similarly, the complexity of the integration of a remote operation referring to a word is $p^2 + (s/sc)^2 + (w/(sc * wc))^2$. And the complexity for the integration of a remote operation referring to a character is $p^2 + (s/sc)^2 + (w/(sc * wc))^2 + (c/(sc * wc * cc))^2$.

The complexity of an algorithm such as SOCT2 and GOT that could deal with operations of different levels of granularity would be $(p + s + w + c)^2$. But, SOCT2, for instance, has defined transformation functions only for operations targeting characters and therefore operations targeting words, sentences or paragraphs have to be split into operations targeting characters. For instance, an operation of insertion or deletion of a word would be divided into cc operations targeting the characters of

the word. Therefore, the complexity of the SOCT2 algorithm would be $(p * sc * wc * cc + s * wc * cc + w * cc + c)^2$. GOT algorithm has defined its transformation functions for strings and, in this way operations targeting paragraphs, sentences or words can be considered as targeting strings. In this case the complexity of the GOT algorithm remains $(p + s + w + c)^2$. Even for this case the treeOPT algorithm has a better efficiency due to the fact that the history is distributed throughout the tree.

We see that our algorithm has a much better complexity than the existing linear algorithms.

3.3.2 Combination of treeOPT with linear operational transformation algorithms

The treeOPT algorithm working for hierarchical structures of documents presented in section 3.3 recursively applies an existing linear operational transformation algorithm over the document levels.

In what follows we present the adaptation of the treeOPT algorithm to work together with the GOT and SOCT2 linear operational transformation algorithms.

Next we present the $treeOPT - GOT$ algorithm, i.e. the adaptation of the $treeOPT$ algorithm when combined with the undo/do/redo scheme and the GOT algorithm.

Algorithm *treeOPT-GOT(O,RN,L)*{
  $CN := RN$;
  for *(l:=1;l≤L;l++)*{
    O_{new}*:=composite2Simple(O,l)*;
    Undo operations in *history(CN)* from right to left until EO_m is found such that $EO_m \Rightarrow O_{new}$
    EO_{new}*:=GOT(*O_{new}*,history(CN)[1,m])*;
    position(O)[l]:=position(EO_{new}*)*;
    if *(level(O)=l)*{
      Do O;
      Update *length(CN)*;
      Store EO_{new} in *history(CN)* after EO_m;
      //Transform each $EO_{m+i} \in$ *history(CN)[m+1,n]* into EO'_{m+i}
      $EO'_{m+1} := IT(EO_{m+1}, EO_{new})$
      for *(i:=2;i≤n-m;i++)*{
        TO:=LET(EO_{m+i}*, (history(CN)[m+1, m+i-1])*$^{-1}$*)*;
        $EO'_{m+i} := LIT(TO, [EO_{new}, EO'_{m+1}, \cdots, EO'_{m+i-1}])$;
      }
      Redo $EO'_{m+1}, EO'_{m+2}, \cdots, EO'_n$ sequentially
    }

```
        else {
            if (length(EO_new)>0){
                CN := child_i(CN), where i=position(EO_new);
                Update length(CN);
            }
            Redo the undone operations
        }
    }
}
```

A short description of the GOT algorithm and its undo/do/redo scheme has already been presented in section 2.3.9, but we present them here again integrated in our treeOPT approach. The procedure *treeOPT-GOT* takes as arguments the causally ready composite operation, O, the root node of the hierarchical representation of the local copy of the document, RN, and the number of levels in the hierarchical structure of the document, L. Unlike in the *treeOPT* algorithm where the execution form of the composite operation is computed, but the operation is performed and saved after the call of the algorithm, the *treeOPT-GOT* algorithm deals with both the computation of the execution form of the operation, the operation execution and its integration in the history buffer.

For each level of the document, starting with the document level and finishing with the level of the operation that has to be integrated, the GOT algorithm is applied in order to compute the position vector of the operation to be executed. Due to the fact that existing algorithms do not perform transformations on composite operations but on regular ones, a function $Composite2Simple$ has been defined to transform a composite operation into a simple one by considering only the element of the position vector that is of interest at the moment.

```
Algorithm composite2Simple(CO,l):SO{
    create new simple operation SO
    if (level(CO)=l)
        type(SO):=type(CO);
    else
        type(SO):=delete;
    position(SO):=position(CO)[l];
    length(SO):=1;
    stateVector(SO):=stateVector(CO);
    initiator(SO):=initiator(CO);
    return SO;
}
```

In order to explain the generation of type of the simple operation we are going to use an example. Suppose that for the composite operation *insertWord(2,2,2,"algorithm")* we have to compute the position of the paragraph where insertion takes place. The intention is to find out the position of the paragraph where the word *"algorithm"* has to be inserted, not to insert or delete a paragraph. Existing algorithms work only with insert and delete primitives, modify primitives do not exist. The solution specific for the application of GOT algorithm is to simulate a delete, when in fact the paragraph is not to be deleted, but to be modified.

The reason is that, as previously mentioned, if an operation deletes a node in the tree and a concurrent operation targets a subnode in the node to be deleted, the latter operation has to be cancelled. We have to deal with the case that the operation O that has to be transformed might have to include the effect of a delete operation that deletes a parent node or the node targeted by O. In this case operation O has to be cancelled. In GOT transformation functions, the result of the transformation of a delete operation against another delete operation targeting the same unit is the cancellation of the operation. The transformation of an insert operation against a delete operation targeting the same position returns as result the original insert operation. Therefore, processing at a coarser granularity level than the level of the composite operation is simulated by a delete operation, and processing at the same granularity level as the level of the composite operation uses the actual operation type for the generation of the regular operation. The position associated with the simple operation is equal to the element of the state vector of the composite operation associated with the level of interest l. The simple operation has length 1, due to the fact that the algorithm operates only on one character, one word, one sentence or one paragraph at a time. The state vector and the initiator site of the simple operation equal the corresponding values of the composite operation.

After the composite operation is transformed into a simple operation O_{new}, the execution form of O_{new} operation is computed by transforming it against the operations in the history buffer associated with the current node. An undo/do/redo scheme has to be designed for ordering the history buffer according to a total global order [108]. The operations in history buffer are undone from right to left until an operation EO_m is found such that EO_m totally precedes O_{new}. The GOT algorithm is called to find the execution form EO_{new} of the O_{new} operation. Note that by list $[n \ldots m]$ we denoted the sublist of list *list* starting at index n and ending at index m. The position of the composite operation O that has to be transformed by

the *treeOPT-GOT* algorithm corresponding to the current level is updated according to the computed position of EO_{new}. If the current level equals to the level of the composite operation O, it means that the position vector has been computed and therefore operation O can be executed. Moreover, the operation can be stored in the history buffer associated with the current node. The history buffer stores simple operations and therefore EO_{new} rather than O is stored in the history buffer. The length of the current node is updated accordingly to include the insertion or deletion of a child node as expressed by the composite operation O.

The list of operations $[EO_{m+1}, EO_{m+2}, \ldots, EO_n]$ from the history buffer associated with the current node containing operations that follow in total order O_{new} and have been undone, has to be redone to include the execution form EO_{new} of O_{new}. The list $[EO'_{m+1}, EO'_{m+2}, \ldots, EO'_n]$ is the list of transformed forms of the undone operations. Each operation EO'_{m+i} starting from left to right is obtained by excluding the list of operations $[EO_{m+1}, EO_{m+2}, \ldots, EO_{m+i-1}]$ and afterwards including the effect of EO_{new} and of the already transformed operations $[EO'_{m+1}, EO'_{m+2}, \ldots, EO'_{m+i-1}]$.

The GOT algorithm presented in [108] is given below.

Algorithm $GOT(O, HB) : O'\{$
  Scan $HB = [EO_1, EO_2, \ldots, EO_m]$ from left to right to find first $EO_k \| O$
  if *(*no EO_k*)*
    $O' := O$
  else {
    Scan $HB[k+1, m]$ to find operations EO_p such that $EO_p \rightarrow O$
    if *(*no EO_p*)*
      $O' := LIT(O, HB[k, m])$;
    else {
      $EOL := [EO_{c_1}, \cdots, EO_{c_r}]$, where $EO_{c_i} \in HB[k+1, m]$ s.t. $EO_{c_i} \rightarrow O$
      Compute $EOL' := [EO'_{c_1}, \ldots, EO'_{c_r}]$, where
        $EO'_{c_1} := LET(EO_{c_1}, HB[k, c_1 - 1]^{-1})$
        for *(i:=2;i≤r;i++)* {
          $TO := LET(EO_{c_i}, HB[k, c_i - 1]^{-1})$;
          $EO'_{c_i} := LIT(TO, [EO'_{c_1}, \ldots, EO'_{c_{i-1}}])$
        }
      $O_{new} := LET(O, EOL'^{-1})$;
      $O' := LIT(O_{new}, HB[k, m])$;
    }
  return O'
}

The algorithm takes as input the remote operation O that has to be transformed and the list $HB = [EO_1, EO_2, \ldots, EO_m]$ containing the operations that precede in total order O. Some of the operations in this list

causally precede O and some others are concurrent with O. O has to be transformed against the concurrent operations, but the precondition of the transformations is that the operations have the same generation contexts. The history buffer is traversed from left to right to find the first operation EO_k such that $EO_k \| O$. If no such operation is found, O does not need to be transformed against any operation and therefore it keeps its original form. Otherwise, the list of operations $HB[k+1, m]$ is scanned to find those operations EO_p that precede O. If no operation that precedes O is found, it means that all operations in the list $HB[k, m]$ are concurrent with O, and therefore O has to include their effects. Otherwise, let $EOL := [EO_{c_1}, \ldots, EO_{c_r}]$ be the list of operations that causally precede O. In order to satisfy the precondition of the exclusion transformation, the operations from EOL have to exclude the effect of the concurrent operations with O from their context. The list of transformed operations is denoted by $EOL' := [EO'_{c_1}, \ldots, EO'_{c_r}]$. Each element EO'_{c_i} is obtained by excluding first from EO_{c_i} those operations belonging to the list $HB[k, m]$ and situated at its left hand side. Afterwards, the effect of the previously transformed operations $[EO'_{c_1}, EO'_{c_2}, \ldots, EO'_{c_{i-1}}]$, if any, is included into EO_{c_i}. After excluding the list EOL' from the context of O, O can include the effect of the operations in the list $HB[k, m]$. The inclusion and exclusion transformation functions for simple operations or for lists of operations are presented in appendix B. In our approach we do not need the transformation functions defined for operations working on strings, but only for operations defined on characters, the character unit being regarded as abstracting any element unit, such as paragraph, sentence, word or character.

We present next the implementation of the *treeOPT* algorithm using *SOCT2*.

```
Algorithm treeOPT-SOCT2(O, RN, L) {
   CN:=RN;
   for (l:=1;l≤L;l++) {
      O_new:=composite2Simple(O,l);
      EO_new:=SOCT2(O_new,history(CN));
      position(O)[l]:=position(EO_new);
      if (level(O)=l) {
         Do O;
         append(EO_new,history(CN));
         Update length(CN);
      }
      CN:=child_i(CN), where i=position(EO_new);
   }
}
```

Since SOCT2 algorithm does not require an undo/do/redo scheme, the *treeOPT-SOCT2* algorithm is simpler than the *treeOPT-GOT* algorithm. The function for transforming a composite operation into a regular operation is the same as in the case of *treeOPT-GOT*. Note that as opposed to the *treeOPT* algorithm where performing the remote operation and saving it in the history buffer has to be done after calling the algorithm, the *treeOPT-SOCT2* and *treeOPT-GOT* deal with the execution of the remote operation and its integration in the history buffer.

The principles of SOCT2 were shortly presented in section 2.3.9, but we describe the SOCT2 approach here again, integrated into our *treeOPT* algorithm.

Algorithm *SOCT2(O,HB):O′* {
  n_1*:=separate(O,HB)*;
  for *(i:=*n_1*+1;i≤size(HB);i++)*
    O′:=transpose_fd(HB[i],O)
  return *O′*;
}

The arguments of the *SOCT2* function are the remote operation O that has to be integrated, and the history buffer *HB* that contains the operations against which the remote operation has to be transformed. In the SOCT2 algorithm the operations in the history buffer are separated by means of the function *separate* into two parts. The first part contains the operations that precede O and the second part contains the operations that are concurrent with O. The function *separate* returns the length of the first part of the history buffer, i.e. the number of elements that precede O. The operation O is then forward transposed against the concurrent operations and the resulting execution form of the operation, O'_i, is returned by the function.

The function *separate* is presented below.

Algorithm *separate(O,HB):n* {
  n:=0;
  for *(i:=1;i≤size(HB);i++)* {
    O_i*:=HB[i]*;
    if *(*$O_i \rightarrow O$*)* {
      for *(j:=i;j≥n+2;j−−)* {
        O_j*:=HB[j]*;
        O_{j-1}*:=HB[j-1]*;
        (O_j*,*O_{j-1}*):=transpose_bk(*O_{j-1}*,*O_j*)*;
        HB[j]:=O_{j-1};
        HB[j-1]:=O_j;

```
            }
            n:=n+1;
        }
    }
    return n;
}
```

As already mentioned, the function *separate* reorders the history buffer HB such that the operations which precede the operation O that has to be integrated are regrouped before the operations that are concurrent to O. In order to perform reordering, the transpose backward transformation has to be applied to each couple (O_i, O_j) such that $O_j \rightarrow O$ and $O_i \| O$. The backward transposition is repeatedly applied until all the operations that precede O are located before the operations concurrent to O. Function *separate* returns the number of operations preceding O.

3.3.3 The split/join problem

In this subsection we want to report on some problems we encountered when adapting the treeOPT algorithm for hierarchical text documents and the solutions we have adopted to overcome these problems.

Even though the algorithm works very well with insert and delete primitives at different levels of the hierarchy, in practice these two primitives are not sufficient to perform all possible operations. Actually this happens due to the introduction of the different hierarchical levels. Let us consider the following example. Assume the second paragraph of a document consists of the following sentence: *"Merging is flexible and efficiency is obtained."* shown in the left part of Figure 3.13.

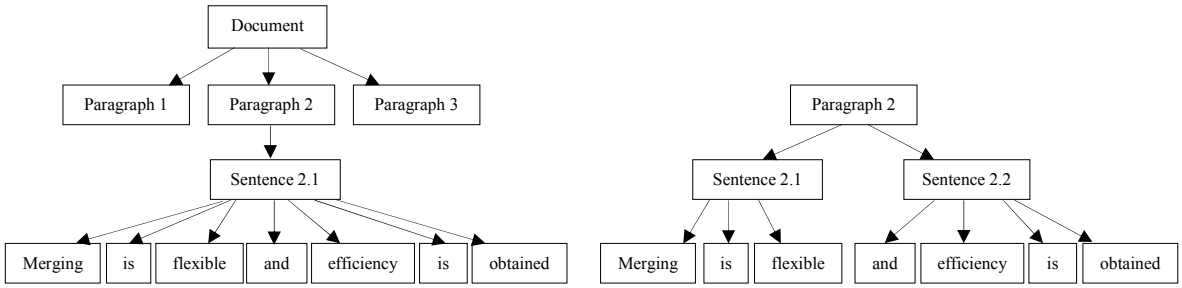


Figure 3.13: Example of split problem

Suppose a user splits the sentence *"Merging is flexible and efficiency is obtained."* into two sentences: *"Merging is flexible. and efficiency is obtained."* One alternative of simulating the split operation is to first delete

the words *"and"*, *"efficiency"*, *"is"* and *"obtained"*, from the first sentence, and then to insert the whole sentence: *"and efficiency is obtained."*. As a result of performing these operations, the new structure of Paragraph 2 is illustrated in the right part of Figure 3.13.

Unfortunately this approach does not work directly as desired. Suppose that concurrently with the split operation of the sentence, another user inserts the word *"better"* as the 5th word in the sentence in order to obtain *"Merging is flexible and better efficiency is obtained."* The operation sequence is illustrated in the left part in Figure 3.14.

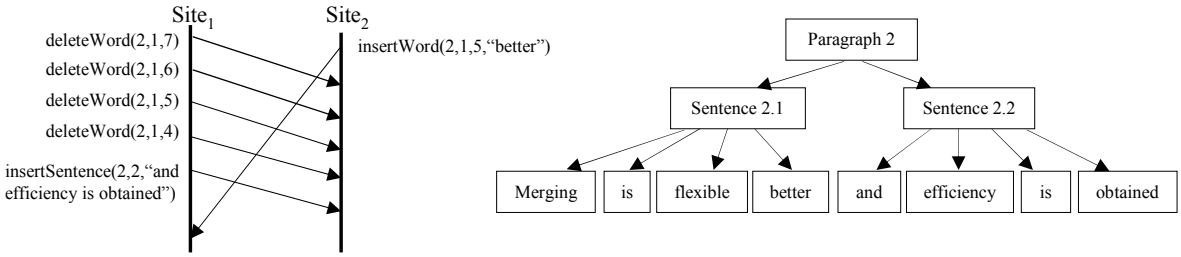


Figure 3.14: Erroneous result due to split operation

As we can see, by the time operation *insertWord(2,1,5,"better")* is received at $Site_1$, the words *"and"*, *"efficiency"*, *"is"* and *"obtained"* are already deleted from the paragraph 2, sentence 1, and these operations of word deletion are kept in the history of Sentence 2.1. By applying the algorithm in its original form, the operation *insertWord(2,1,5,"better")* is transformed into *insertWord(2,1,4,"better")*. The resulting structure of the paragraph, shown in the right part of Figure 3.14, is not what the user at $Site_2$ intended.

A solution to the split problem would be to delete the unit that has to be split and insert two new units containing the split parts. For instance, in the previously described example of splitting the sentence *"Merging is flexible and efficiency is obtained."*, the solution would be to first delete the sentence by generating the operation O_1=*deleteSentence(2,1)* and then insert two new sentences *"Merging is flexible."* and *"and efficiency is obtained."*, by issuing the operations O_2 =*insertSentence(2,1,"Merging is flexible.")* and O_3 =*insertSentence(2,2,"and efficiency is obtained.")*. The problem is that executing these two operations has to be atomic at all sites, which is very hard to ensure in a real-time system, where the operations arrive with different delays. For instance, if a user at one site sees only the effect of O_1 and O_2, the effect of O_3 being delayed, the user could misinterpret the action of splitting the sentence. By deleting the original sentence, all concurrent operations referring to the original sentence are cancelled.

Moreover, transformation functions would have to be adapted for groups of operations. To our knowledge, the only paper that deals with the transformation of a group of operations against another group of operations for the real-time collaboration is the work described in [108]. However, their extended transformation functions working for groups of operations are bound to the transformation functions written for strings of characters and their approach cannot be used for working with groups of operations, as needed in our approach.

In what follows we are going to explain the approach used in [108] for the extension of transformation functions for groups of operations and show why this approach could not be applied in our case. The reason why inclusion and exclusion transformation functions had to be adapted for a sequence of operations is that inclusion and exclusion functions generate as result not a simple operation, but a composite operation consisting of two simple operations. Remember that the transformation functions in GOT approach were defined for operations working on strings of characters. Consider the case of two concurrent operations one deleting a string of characters and the other inserting at a position inside the string of characters deleted by the other concurrent operation. When the deletion operation has to be transformed against the insertion operation, the result is a composite operation composed of two delete operations targeting respectively the two ranges of characters to be deleted, the ranges being the result of the split by the insert operation. The two operations in the sequence are defined on the same context. However, when a composite operation has to be included against a list of operations, the operations composing the composite operation are contextually serialised and treated further as simple operations. It is not specified what the state vector associated with these operations is. However, this would not fulfil our requirements as we would like that the operations which are part of the composite operation are treated together when transformations are performed and the atomicity of the composite operation is kept. For instance, if operation O_1 is transformed against O_2 and the result is the composed operation $O_c = O_{r_1} \oplus O_{r_2}$, where O_{r_1} and O_{r_2} have the same context and O_c has to include the effect of another operation O, then the O_c transformed against O returns the list $[O'_{r_1}, O'_{r_2}]$, where O'_{r_1} and O'_{r_2} are contextually serialised. O'_{r_1} and O'_{r_2} are further considered as simple operations and they do not know that they originate from the composed operation O_c. Further, the reversibility property is not maintained. If O_c excludes the effect of O_2, the result should be operation O_1 and not a sequence of two operations as the transformation functions in the GOT approach return.

Some other possible ways of simulating the split operation using only insertions and deletions exist, but none of them are feasible. The reason is that a structural element might appear to be different on two hosts at the same time, and the two structures converge only because the history of operations on that element is kept at both sites. When an element is split into two parts, its history must be also split. By using only elementary insert/delete operations the cases when the history needs to be split or not cannot be detected. Operations of insert and delete do not determine the splitting of a history buffer. Only the introduction of a new split primitive would trigger the history buffer to be split. The same problem is encountered in the case of joining two elements. For example, if we delete a sentence separator, the two adjacent sentences will be joined into a single one implying the joining of the histories of the two sentences.

An alternative solution would be therefore to introduce two other primitives: split and join, and to modify the algorithm by implementing operational transformation functions for these primitives as well. By means of an example, we show that this solution is not feasible either. Suppose that the initial state of the document, as in the previous example, consists of the sentence S_1 = *"Merging is flexible and efficiency is obtained."* and that both local copies of document at $Site_1$ and $Site_2$ contain sentence S_1. A scenario illustrating the concurrent editing of the document is given in Figure 3.15.

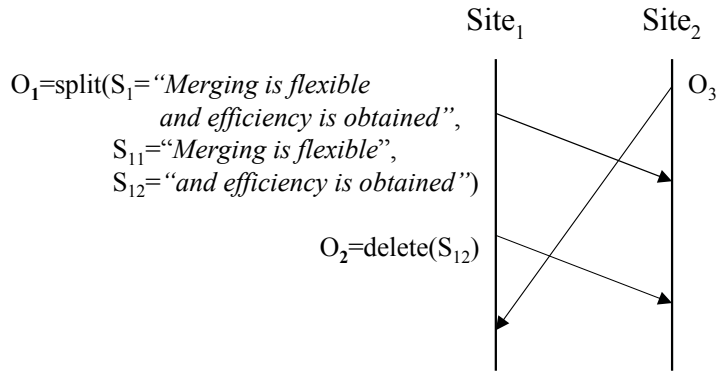


Figure 3.15: Counterexample for split primitive

Operation O_1 initiated by the user at $Site_1$ splits sentence S_1 into two sentences S_{11}=*"Merging is flexible"* and S_{12}=*" and efficiency is obtained."*. Operation O_2 deletes sentence S_{12}. When operation O_2 arrives at $Site_2$, it has to be transformed against O_3, whatever O_3 is. But the precondition of the transformation is that O_2 has the same initial context as O_3. In the case that GOT algorithm is applied, the intuitive explanation of the action that has to be performed is that O_1 has to be excluded from the

context of O_2. But, let us analyse in detail how GOT algorithm is applied at $Site_2$. When O_1 arrives at $Site_2$, it is transformed against O_3, the result of the transformation being O_1'. The history buffer is $HB = [O_3, O_1']$. When O_2 arrives at $Site_2$, both O_3 and O_1' are preceding in total order O_2, so no operations have to be undone. O_3 is concurrent with O_2 and O_1' is preceding O_2. According to GOT algorithm, O_1' is transformed to exclude operation O_3 from its context, resulting into operation O_1'' which should return the original operation O_1. Operation O_2 should then exclude the effect of O_1'' in order to include afterwards the effect of $[O_3, O_1']$. The problem that occurs is when O_2 excludes the effect of $O_1'' = O_1$.

If we exclude the effect of O_1 that splits S_1 into S_{11} and S_{12}, the deletion of the sentence *" and efficiency is obtained."* will have to be transformed into four different operations, which are not of the same granularity level: *deleteWord("and")*, *deleteWord("efficiency")*, *deleteWord("is")* and *deleteWord("obtained")*. This is not an acceptable solution as the idea of the treeOPT algorithm is to transform operations against other operations at the same level of granularity, the result being an operation at the same level of granularity as well.

However, the previously described scenario occurs only in the GOT algorithm. If O_1 is excluded from O_2, when O_1 and O_2 are generated at the same site and O_1 precedes O_2 does not appear in the SOCT2 family of algorithms, it occurs only in the GOT algorithm.

Anyway, in all existing operational transformation approaches the case that a delete operation is transformed against a concurrent split operation occurs, and may result in a composite operation consisting of two delete operations. Consider again the document consisting of the sentence S_1 = *"Merging is flexible and efficiency is obtained."*. Consider that $User_1$ performs the operation O_1 of splitting sentence S_1 into two sentences S_{11}=*"Merging is flexible"* and S_{12}=*" and efficiency is obtained."*. Consider that $User_2$ concurrently with the operation performed by $User_1$ issues operation O_2 to delete sentence S_1. When operation O_2 is transformed against operation O_1, the result is the composed operation containing the deletion of the sentence S_{11} and the deletion of the sentence S_{12}. Working with composite operations leads to the issue of defining the inclusion and exclusion transformation functions for groups of operations which generates a set of problems as previously mentioned. Therefore, introducing the additional operations split and join is not a suitable solution.

The most appropriate solution we found so far for real-time collaboration, although somehow disappointing, is not to split or join elements in the tree structure. For example, given the text *"multi version"* composed of

two words, deleting the space between the two words, still has as a result the two words, even though not separated by any separator. The same approach can be adopted in the case of splits. If we insert a sentence separator, for instance a dot, or even a paragraph separator, for instance new line, inside a sentence, the text is kept as a single sentence. Embracing this approach leads however to degenerated elements. For example, the text *"The approach offers flexibility. A better efficiency is obtained"* might be a single large degenerated word, and the text *"merge"* can be stored in three degenerated sentences: *"m"*, *"er"*, and *"ge"*. Obviously, the hierarchical structure resulting in the case of degenerated elements is different from the one obtained by parsing the text and by delimiting the elements using their natural separators.

Even with this drawback, the algorithm works well. When issuing an operation, the positions of the elements of different granularity levels (paragraph, sentence and word) are computed by taking into account the length of the previous elements. Consequently, the fact that the elements are degenerated does not matter. The efficiency of the algorithm remains unaffected by the degenerated elements, because the structure of the document remains hierarchical, and operations are transformed locally, spanning only a small part of the whole history of operations. Unfortunately, semantic consistency is more difficult to maintain. However, the problem is not as severe as it seems, as the reparsing of the whole document is performed every time a new user begins the editing of the same document, or when the document is reloaded. Reparsing restores the semantic consistency of elements, only non-degenerated elements being generated. Reparsing the document should be enforced as often as possible. But, parsing the document implies having the same copy of the document at all sites. This means that reparsing can be performed only in moments of quiescence. Either the system can detect the moments of quiescence and initiate reparsing on copies of the document at all sites, or quiescence could be enforced by the system from time to time.

To avoid the join/split problem, the interface could restrict the user from performing these operations. We found that this solution is too frustrating for the user and therefore we adopted the element degeneration as a solution.

3.4 The asyncTreeOPT algorithm

In this section we present how our treeOPT algorithm was applied to maintain the consistency over hierarchical documents in the asynchronous collaboration with a central repository.

We first present in subsection 3.4.1 the basic architecture of a version control system and the set of operations implemented by the version control systems. As we saw in section 3.3, an algorithm for merging working for linear structures has been recursively applied over the document levels. In subsection 3.4.2 we motivate why we chose the FORCE [94] algorithm for merging to be used in our approach. We then present in subsection 3.4.3 the FORCE approach that we apply recursively over the document structure. In subsection 3.4.4 we present our asyncTreeOPT approach for merging for text documents. In subsection 3.4.5 we present the transformation functions that we used in our approach and in subsection 3.4.6 we present an example illustrating the asynchronous communication. Subsection 3.4.7 presents the way conflicts can be dynamically defined and resolved at different levels of granularity corresponding to the document levels. Subsection 3.4.8 presents the solution that we adopted for splitting and joining elements in the asynchronous communication.

3.4.1 Basic operations of version control systems

The basic configuration of a version control system is illustrated in Figure 3.16. It consists of a repository and a set of clients that work in isolation in their local workspaces and synchronise at different moments of time their local copies of the documents with the public copy of the documents stored in the repository.

Most configuration management tools support the copy/modify/merge paradigm. It consists basically of three operations applied on a shared repository storing multiversioned objects: checkout, commit and update. A *checkout* operation creates a local working copy of an object from the repository. A *commit* operation creates a new version of the corresponding object in the repository by validating the modifications done on the local copy of the object. The condition of performing this operation is that the repository does not contain a more recent version of the object to be committed than the local copy of the object. An *update* operation performs the merging of the local copy of the object with the last version of that object stored in the repository.

In Figure 3.17 a scenario is illustrated in order to show the functional-

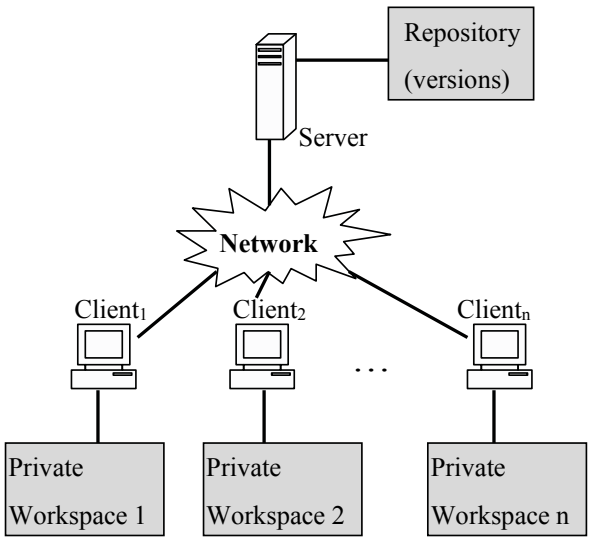


Figure 3.16: Configuration of a version control system

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

Early version control systems such as RCS [112] do not support merging. When a document is checked out from the repository by a user, it is locked until it is committed to the repository by the same user, preventing other users from concurrently performing changes to the same document.

However, current commercial version control systems such as CVS [12], ClearCase [7] and Subversion [19] all support merging.

The merging approaches can be classified as *state-based* or *operation-based*. The state-based approaches are characterised by the fact that only the information about the states of the documents and no information about the evolution of one state into another is used. On the other hand, operation-based merging approaches keep the information about the evolution of one state of the document into another in a buffer containing

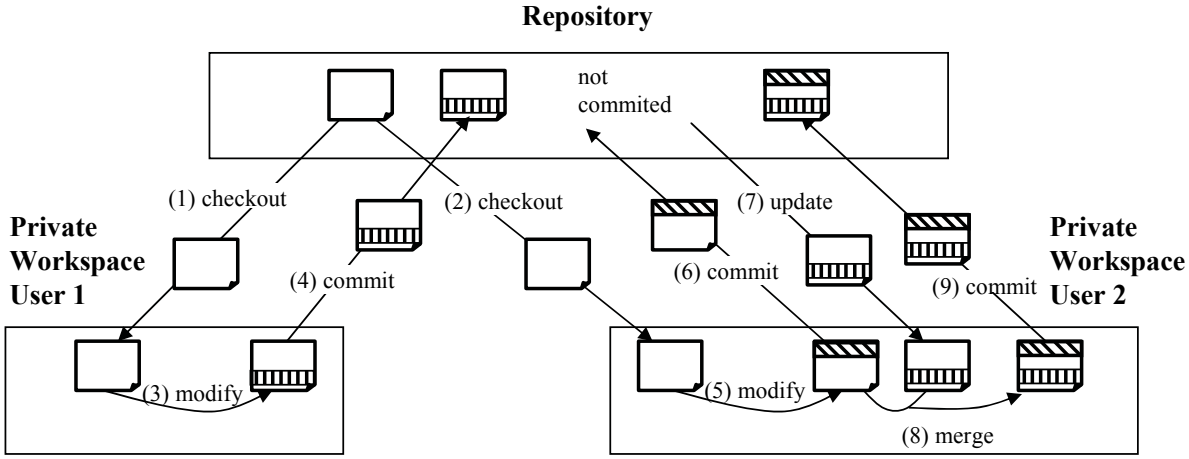


Figure 3.17: Copy/Modify/Merge paradigm

the operations performed between the two states of the document. The merging is done by executing the operations performed on a copy of the document onto the other copy of the document to be merged.

We adopted the operation-based approach. In what follows we motivate why we chose the FORCE operation-based approach for merging and we present how FORCE approach working for linear structures of the document was used in our algorithm to merge hierarchical structures.

3.4.2 Reusing an existing linear merging approach

In this section we motivate why we chose an existing linear merging approach and adapted it for the asynchronous communication with a central repository.

A question that arises is why we did not use the same algorithm that we implemented for the real-time communication. In what follows we present the motivations for our decision.

In what follows we present the requirements of a merging algorithm in terms of the implementation of the basic operations in an asynchronous communication with a shared repository, i.e. commit, checkout and update.

In the commit phase of merging a check is first performed as to whether the user can commit the changes to the repository. If the base version of the document is in the local workspace, i.e. the last version from the repository that the user started working on is equal to the last version in the repository, a commit can be performed. Otherwise, an update is necessary before committing the data. If a commit is allowed and is to be performed, the repository should simply execute the operations that were performed in the local workspace.

In the checkout phase, a request should be sent to the repository to specify the version of the document that is to be checked out. Using the set of operations stored in the repository as delta, the system should be able to provide to the local workspace either the state of the required version of the document or the set of operations that allows the computing of the required version.

In the updating phase, the repository should send to the local workspace a list of operations representing the delta between the latest version in the repository and the base version in the local workspace. Upon receiving the list of operations from the repository, the local workspace should perform a merging algorithm to update the local version of the document. The merging scenario is illustrated in Figure 3.18. The local user started working from version V_k on the repository but cannot commit the changes because meanwhile the version from the repository has been updated to version V_{k+n}. Let us denote by LL the list of operations executed by the user in their local workspace and by DL the list of operations representing the delta between versions V_{k+n} and V_k. Two basic steps have to be performed. The first step consists of applying the operations from DL on the user's local copy in order to update the local document to version V_{k+n}. The operations from the repository, however, cannot be executed in their original form as they have to be transformed in order to include the effect of all the local operations before they can be executed in the user workspace. The second step consists of transforming the operations in LL in order to include the effects of the operations in DL. The resulting list of transformed local operations represents the new delta in the repository.

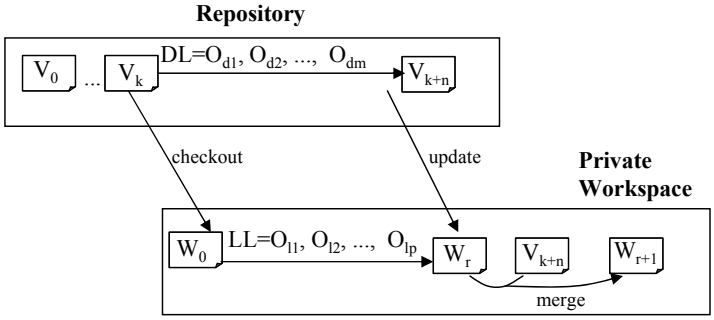


Figure 3.18: Updating Stage of the Merging

In the list of operations in DL not all of them can be executed in the local workspace as some of these operations may be in conflict with some of the operations from LL. Let us consider $DL = [O_{d1}, \ldots, O_{d(i-1)}, O_{di}, O_{d(i+1)}, \ldots, O_{dm}]$. In the case that O_{di} is in conflict with at least one operation from LL and O_{di} cannot be executed in the local workspace,

a mechanism for undoing O_{di} should be provided such that the effect of O_{di} is excluded from the operations that follow it in DL and the effect of undoing O_{di} is obtained by executing some new operations after the operations from DL were executed. The reason that O_{di} should be excluded from the operations O_{dj} that follow O_{di} in DL is that when O_{dj} has to be transformed against the list LL, the form of O_{dj} has to be adapted to illustrate the fact that O_{di} was cancelled. The fact that the undoing of O_{di} should be obtained by executing some new operations after the operations from DL were executed is required by the fact that the list DL is already stored in the repository and it cannot be modified, as it represents the delta between two existent versions in the repository. The effect of cancelling O_{di} in the repository due to a conflicting local operation can be made visible in the repository only after the local user commits their changes to the repository. When a commit is performed, the new delta should contain the operations whose effect cancels O_{di}.

If a conflict between O_{di} and an operation in the local log LL occurs, and the local operation has to be cancelled, the cancelled local operation should be excluded from the operations that follow it in LL such that when these operations are transformed against operations in DL they should reflect the fact that an operation preceding them was cancelled.

We have shown that a mechanism for performing the integration of an operation into the log and the cancellation of an operation from the log in the way described above has to be provided.

In [85, 104] undo mechanisms have been proposed, so one of these mechanisms could be used to cancel an operation. To integrate an operation into a log, one of the algorithms working for real-time communication such as SOCT2 [101] or GOTO [107] could be applied. However, specialised algorithms for asynchronous communication such as FORCE [94] are likely to perform better, as shown below.

Suppose that $DL = [O_{d1}, \ldots, O_{d(i-1)}, O_{di}, O_{d(i+1)}, \ldots, O_{dm}]$ and $LL = [O_{l1}, \ldots, O_{l(i-1)}, O_{li}, O_{l(i+1)}, \ldots, O_{ln}]$. The operations in DL and in LL are contextually preceding and O_{d1} and O_{l1} have the same initial context and all operations in DL are concurrent with the operations in LL. Let us analyse the number of transformations that have to be performed to integrate each operation belonging to DL into LL and each operation belonging to LL into DL using the SOCT2 [101] algorithm. Let us analyse first the integration of the operations belonging to DL into LL. When O_{d1} is transformed against all operations in LL, n inclusion transformations will be performed, the result being operation O'_{d1}. When O_{d2} has to be integrated into the transformed local log $LL' = [O_{l1}, \ldots, O_{ln}, O'_{d1}]$,

the operations in the log have to be reordered such that the first part of the log contains the operations that precede O_{d2} and the last part of the log contains the operations that are concurrent with O_{d2}. Therefore, O'_{d1} has to be transposed at the beginning of the history buffer. Each step of the transposition involves the computation of an inclusion and exclusion transformation and, therefore, the transposition process requires $2 * n$ transformations. Afterwards, O_{d2} has to be transformed against the concurrent operations and, in this case, n inclusion transformations will be performed. Therefore, the integration of O_{d2} requires $3 * n$ transformations. The integration of all operations in DL into LL requires therefore $n+3*n*(m-1) = 3*n*m-2*n$ transformations to be performed. Similarly, the integration of all operations belonging to LL into DL requires $3 * n * m - 2 * m$ operations to be performed. Therefore, the total number of transformations are $6 * n * m - 2 * m - 2 * n$.

The FORCE [94] approach transforms each operation O_{di} in DL in turn with respect to each operation O_{lj} in LL and, after such a transformation is performed, the symmetric transformation of O_{lj} with respect to O_{di} is also performed. The approach requires $2 * n * m$ transformations to be performed and the logs have to be traversed only once.

We therefore applied the FORCE algorithm in our merging approach recursively over the document levels.

3.4.3 FORCE linear approach for merging

In this subsection we present the FORCE merging algorithm that has been applied recursively over the document levels by our merging approach.

In the commit phase of merging, a check is first performed as to whether the user can commit the changes to the repository. If the base version of the document in the local workspace, i.e. the last version from the repository that the user started working on, is equal to the last version in the repository, a commit can be performed. Otherwise, an update is necessary before committing the data. In the case that a commit is allowed, the repository simply executes sequentially the operations that were performed in the local workspace in order to generate the full state of the latest version from the repository. The previous version from the repository is replaced with the operations received from the local workspace, representing the delta between the new version and the previous version. Additionally, the corresponding base version number from the local workspace as well as the latest version number from the repository are increased and the local log from the local workspace is emptied.

In the checkout phase a request is sent to the repository including the version number of the document that is intended to be checked out. In the case that the requested version number is larger than the latest version number in the repository, the repository sends a reject reply. In the case that the requested version number equals the latest version number in the repository, the repository sends the full state of the last version of the document to the local workspace. In the case that the requested version number is less than the latest version number from the repository, the repository generates the state of the requested version by executing the inverses of the operations representing the deltas between the latest version in the repository and the requested version. In the case of a positive reply from the repository, the local site makes the sent document the working copy and sets the base version number to be equal to the version number of the document that was sent.

In the updating phase, the site sends the number of the base version to the repository. The repository sends to the site a list of operations representing the delta between the latest version in the repository and the base version. Upon receiving the list of operations from the repository, the local workspace performs the merging algorithm and updates the base version number. The merging scenario is illustrated in Figure 3.18. Remember that LL is the list of operations executed by the user in their local workspace and DL is the list of operations representing the delta between versions V_{k+n} and V_k. As described in section 3.4.2, two basic steps have to be performed. The first step consists of applying the operations from DL on the local copy by transforming operations in DL against operations in LL. The second step consists of computing the new delta in the repository by transforming the operations in LL in order to include the effects of the operations in DL.

FORCE adopts an additional abstraction layer which allows a complete separation of the syntactic merging from the semantic merging by means of the semantic conflict function. A semantic merging policy is specified as a set of semantic merging rules and a function *semanticConflict* determines whether two concurrent operations are semantically conflicting. Let us consider $DL = [O_{d1}, ..., O_{d(i-1)}, O_{di}, O_{d(i+1)}, ..., O_{dm}]$, where $O_{d1} \mapsto ... \mapsto O_{dm}$. In the case that O_{di} is in conflict with at least one operation from LL, O_{di} cannot be executed in the local workspace. Moreover, all operations following it in the list DL need to exclude its effect from their context. But, as we have already seen, the condition to exclude an operation O_a from an operation O_b is that $O_a \mapsto O_b$. Therefore, in order to exclude the effect of operation O_{di} from the context of all the operations following it

in the list DL, we need to transpose operation O_{di} towards the end of the list DL. As a result of this transposition the following condition should be fulfilled: $O_{d1} \mapsto O_{d2} \mapsto ... \mapsto O_{d(i-1)} \mapsto O_{d(i+1)} \mapsto ... \mapsto O_{dm} \mapsto O_{di}$.

The *transpose* function that changes the execution order of the operations O_a and O_b and transforms them such that the same effect is obtained as if the operations were executed in their initial order and initial form is defined below.

Algorithm $transpose(O_a, O_b) : (O'_b, O'_a)$ {
  $O'_b := ET(O_b, O_a)$;
  $O'_a := IT(O_a, O'_a)$;
  return (O'_b, O'_a);
}

The condition of performing the *transpose* function is that $O_a \mapsto O_b$ and after the call of $transpose(O_a, O_b)$, $O'_b \mapsto O'_a$, where O'_b and O'_a are the transformed forms of O_b and O_a, respectively.

In order to combine the two steps of the merging, i.e. the computing of the transformations of the operations from the repository against the operations from the local log and the transformations of the operations from the local log against the repository, the symmetric inclusion operation has been defined:

Algorithm $symmetricInclusion(O_a, O_b) : (O'_b, O'_a)$ {
  $O'_a := IT(O_a, O_b)$;
  $O'_b := IT(O_b, O_a)$;
  return (O'_b, O'_a);
}

In what follows we present the merge procedure. It takes as input arguments two logs, the remote log RL containing the operations from the repository and the local log LL containing the local operations and the base version number $V.bv$ at the local site. The merge procedure generates as output two other logs, the new remote log NRL and the new local log NLL, each of which is modified to include the effects of the operations in the other log. The new remote log NRL will contain 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 the non conflicting operations from the original remote log, modified in order to include the effects of the operations in the local log. The new local log NLL will store the list of operations which represent the delta between the new version and the old version in the repository and will have to be sent to the repository.

It contains the operations in the local log transformed in order to include the effect of the operations in the remote log. Additionally, it might also include the inverse of the conflicting operations from the remote log. The implementation of the merge procedure is given below.

```
Algorithm merge(RL, LL, V.bv):(NRL,NLL) {
  RCT:=V.bv; //initial remote context
  for (i:=1;i≤|RL|;i++) {
    //make copies of the LL and RL[i] current states
    CLL:=makeCopy(LL);
    CRL_i:=makeCopy(RL[i]);
    LCT:=RCT;//initial local context
    for (j:=1;j≤ |LL|;j++) {
      if semanticConflict(SMR, RL[i], LL[j], LCT) {
        // Recover the states of LL and RL[i]
        LL:=CLL;
        RL[i]:=CRL_i;
        //remove RL[i] from RL
        O:=removeOperation(i, RL);
        i:=i-1;
        //append Ō to NLL
        append(makeInverse(O),NLL);
        //Exit the loop since a conflict occured
        break;
      } else {
        //update local context
        LCT:=execute LL[j] on LCT;
        //transform RL[i] and LL[j] against each other
        symmetricInclusion(RL[i],LL[j]);
      }
    }
    //If RL[i] is not conflicting with any operation in LL
    //append it to NRL
    if (j> |LL|) {
      append(RL[i],NRL);
      //update remote context
      RCT:=execute CRL_i on RCT;
    }
  }
  //All transformed operations in LL are appended to NLL
  append(LL,NLL);
  return (NRL,NLL)
}
```

In order to perform the correct transformations, the local and remote contexts need to be updated accordingly. Initially, the remote context equals the base version of the document in the repository. For each operation in the remote log, a sequence of steps is performed. At the beginning of

the iteration, a copy of the local log is saved in case the local log needs to be restored later. Also, a copy of the current operation from the remote log is saved for possible restoration later. All operations in the local log are iterated and a check for conflict between the local operation and the remote one is performed. The function *semanticConflict(SMR,RL[i],LL[j],CT)* determines whether the two concurrent operations $RL[i]$ and $LL[j]$ are conflicting according to the context CT on which they were executed, as specified by the set of semantic merging rules SMR. Two cases are distinguished depending on the existence of conflict.

If the two operations are not in conflict, the symmetric inclusion procedure will be called in order to transform the remote operation against the local operation and vice-versa. The local context is updated in order to include the last local operation. If the remote operation is not in conflict with any of the local operations, by the end of the iteration over the local log list, the remote operation will have orderly included the effect of each of the local operations and each of the local operations will have included its effect. Therefore, the remote operation is added at the end of the new remote log and the remote context is updated in order to include the initial form of the remote operation. By the end of the iteration over the remote log, each of the operations in the local log will have included the effect of each of the operations in the remote log. Therefore, the transformed operations from the local log can be added to the new local log, as their context includes all the operations from the repository.

If the remote and local operations are in conflict, according to the resolution conflict policy adopted, one of these two operations will be kept and the other one cancelled. In the merge procedure presented above, the local operation is chosen automatically as the winner of the conflict and the remote one is cancelled. In this case, the remote operation should be eliminated from the remote log. The local log has to be restored to its form before some of the local operations had the effect of the remote operation to be removed included in their definition. The remote operation needs to be reset to its original form before including the effect of the local operations up to the current conflicting local operation. Next, the *removeOperation* procedure has to be applied in order to successively transpose the remote operation to the end of the remote log.

The *removeOperation* function is presented in what follows.

Algorithm *removeOperation(k,L):*O_k {
  for *(i=k;i<|L|;i++)*
    transpose(L[i],L[i+1]);
  $O_k := L[i]$;
  remove(L[i]);
  return O_k;
}

The *removeOperation(k,L)* function removes the kth operation O_k from the list L and returns O_k on the current context. As result of the application of this function, the following condition holds: $O_1 \mapsto O_2 \mapsto ... \mapsto O_{(k-1)} \mapsto O_{k+1} \mapsto ... \mapsto O_{|L|} \mapsto O_k$.

By the application of the *removeOperation* function, the remote operation includes the effect of the operations that follow it in the remote log and its inverse can be safely added to the beginning of the new local log. The inverse operation simply cancels the effect of the original operation from the repository. Once the iterations are finished, the operations from the local log need to be added to the new local log.

3.4.4 Description of asyncTreeOPT

The asyncTreeOPT [45, 44, 49] approach can be seen as a generalisation of the FORCE merge algorithm, presented in the previous section, which works on a hierarchical document structure. Our merging algorithm recursively applies the FORCE linear approach for merging over document levels. The asyncTreeOPT merging algorithm can be applied to any documents conforming to a hierarchical structure, such as XML-like and text based documents. In this section we present the application of the asyncTreeOPT algorithm to text document merging. The application of the asyncTreeOPT approach to XML merging is presented in chapter 5.

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

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

implemented in the same way as described for the linear representation of documents. The main difference is that, in the FORCE approach, the latest version in the repository is the state of the document and the previous versions are represented by a list of operations constituting the delta between the versions. However, in our approach, all versions are represented by the delta list of operations and only the first version in the repository contains the state of the document.

The *update* procedure, presented in what follows, achieves the actual update of the local version of the hierarchical document with the changes that have been committed by other users to the repository and kept in the remote log. The remote log contains a linearisation of the logs that were initially part of the document tree structure. The goal of the update procedure is the same as of the *merge* procedure presented in section 3.4.3 generalised for the level of the entire document tree. It aims to replace the local log associated with each node with a new one which includes the effects of all non conflicting operations from the remote log and to 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 update procedure is presented in what follows.

```
    Algorithm update(CN, RL) {
1:      LLL:=log(CN);
        bInd:=|RL|;
        RLL:=[];
        for (i:=0;i<|RL|;i++){
            O:=RL[i];
            if (level(O)=level(CN))
                append(O,RLL);
            else {
                bInd:=i;
                break;
            }
        }
2:      updateOpInds(LLL,indices(CN));
3:      (NRL, NLL):=merge(RLL,LLL);
4:      for (i:=0;i<|NRL|;i++)
            applyOperation(NRL[i]));
        setLog(CN,NLL);
5:      ChildRL:=[];
        for (i:=0;i<noChildren(CN);i++)
            ChildRL[i]:=[];
        for (i:=bInd;i<|RL|;i++){
            O:=RL[i];
            for (j:=0;j<|NLL|;j++)
                include(O,NLL[j]);
```

```
        append(O,ChildRL[index(O,level(CN))]);
    }
    for (i:=0;i<noChildren(CN);i++)
        update(childAt(CN,i),ChildRL[i]);
}
```

The CN argument of the *update* procedure represents the current node in the tree traversal. In the initial call of the procedure the current node is equal to the root of the document tree. The parameter RL represents the remote log. Block 1 of the algorithm represents the initialisation phase. The local level log LLL and the remote level log RLL have the same purpose as in the basic merge algorithm, the only difference being that they contain only the part of the remote and local logs referring to the current node. The RLL is initialised with the remote operations pertaining to the current node, by iterating over the remote log and keeping those operations whose level is identical to the level of the current node. Recall that the level of an operation is equal to the level of the node in whose history the operation is kept. For text documents composed of paragraphs, sentences, words and characters, an *insertParagraph* operation belongs to document history and is of level 0, an *insertSentence* operation is of level 1, an *insertWord* operation is of level 2 and an *insertChar* operation is of level 3. The *bInd* variable stores the index of the first operation that refers to a lower level than the level of the current node.

Block 2 of the algorithm includes the update of the indices of all the operations in the LLL so that they correspond to the current position in the tree of the node to whose log they belong. 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. As the positions of the nodes change, it is clear that all operations belonging to the log of the nodes whose position have changed will no longer have valid indices. For example, if the local level log contains the operations *deleteChar("d",1,3,4,5)* and paragraph 1 has been shifted two positions to the right by the insertion of two new paragraphs before it, the operation has to be transformed to *deleteChar("d",3,3,4,5)*.

In block 3 of the algorithm the basic merge algorithm is called in order to merge the RLL and the LLL and generate two new logs, NRL and NLL. In the version of the update procedure presented in this paper, due to the merge procedure, the conflict resolution policy is that the local version of the operation is the one that is kept in the case of conflicts. In the current implementation of the asynchronous text editing system, other policies for merging have also been implemented, as described in one of the

next subsections.

Afterwards, in block 4 of the algorithm the operations from the NRL are applied to the local copy of the document in order to update it and the local log of the current node is then replaced with the NLL. We mention that for our merging algorithm we can use any existing linear approach for the merging of two lists of operations. However, in our current implementation, we have used the FORCE merging algorithm.

In block 5 of the *update* procedure the remaining part of the remote log, i.e. the part including the operations following *bInd*, needs to be divided among the children of the current node and the update method called recursively for each child. Each operation in the remote log starting from position *bInd* will be transformed in order to include the effects of all the operations in the NLL. 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. Afterwards, the transformed remote operations will be added into the corresponding *ChildRL* elements chosen by analysing the modified index corresponding to the level of the current node. *index(O,L)* returns the index of operation O corresponding to the level L. By the end of the iteration, all remote operations will have been transformed and placed in the correct list. Finally, the update method is recursively called with each of the previously created lists of operations as remote logs.

3.4.5 Transformation functions

In this subsection we present the include and exclude methods for the hierarchical representation of documents. The operations that are allowed are the insertion and deletion of elements. An operation that has no effect is referred to as a *NOP* operation.

When merging is recursively performed for one granularity level, the two logs of operations that have to be merged contain operations of different levels of granularity. If merging is performed at a certain granularity level, the local log contains operations of that granularity level and the remote log contains operations of that granularity level and other operations of lower granularity levels. Transformation functions have been defined for operations at different granularity levels.

We start by presenting the include function.

Algorithm *include*(O_a,O_b):O'_a {
1: O'_a:=O_a;
if (*type*(O_b)=*NOP* or *type*(O_a)=*NOP*) return O'_a;

```
        if (level(O_b)>level(O_a)) return O'_a;
        for (i:=0;i<level(O_b);i++)
            if (index(O_a,i)≠index(O_b,i)) return O'_a;
2:      if (type(O_b)=delete{
            if (index(O_a,level(O_b))=index(O_b,level(O_b)))
                if ((level(O_a)≠level(O_b)) or
                    (level(O_a)=level(O_b) and type(O_a)=delete))
                        type(O'_a):=NOP;
            if (index(O_a,level(O_b))>index(O_b,level(O_b)))
                index(O'_a,level(O_b))−−;
        }
3:      else if (type(O_b)=insert){
            if (index(O_a,level(O_b))=index(O_b,level(O_b)) and
                type(O_a)=insert and level(O_a)=level(O_b))
                if (content(O_a)>content(O_b))
                    index(O'_a,level(O_b))++;
            else if (index(O_a,level(O_b))≥index(O_b,level(O_b)))
                index(O'_a,level(O_b))++;
        }
        return O'_a;
    }
```

The first argument of the *include* function, O_a, denotes the operation that has to be transformed and the second argument, O_b, denotes the operation against which the transformation has to be performed. In block 1 of the *include* function some special cases are analysed when operation O_a does not need any transformation. An operation does not change by including a NOP operation and a *NOP* operation does not change by including any other operation. If O_a is an operation of a coarser granularity level than O_b, no changes have to be performed on the current operation. For instance, an *insertWord* operation cannot influence a *deleteSentence* operation in any way, even if both operations refer to the same sentence. Next, a test is performed to check if O_b refers to a node that is on the path from the root to the node referred to by O_a. If it is not the case, the operation does not need modifications. For instance, if O_b is an *insertSentence* and O_a is a *deleteWord* operation, the *insertSentence* operation could influence the sentence index of the *deleteWord* operation only if the word is deleted from a sentence that is in the same paragraph as the one to which the *insertSentence* operation is referring.

When processing reaches block 2 we know that O_b might actually have an effect on O_a. Block 2 analyses the cases when O_b is a delete operation. Two cases are considered. The first case is if the index of O_a corresponding to the level of O_b is equal to that of O_b. In this case, there are two subcases when O_a needs to be cancelled. The first one is when O_a is of a lower level

than O_b, meaning that O_b is actually deleting the tree containing the node to which O_a refers. The second subcase when O_a has to be cancelled is when the two operations refer to the same level and O_a is also a delete, meaning that both operations are deleting the same semantic unit.

The next case to be considered in block 2 when O_b is a delete, is whether the index of O_a corresponding to the level of O_b is greater than that of O_b. If it is, the index needs to be decreased by one since O_b deleted a node.

Step 3 in the *include* function analyses the cases when O_b is an insert operation. Two cases need to be considered. The first is when the lowest available index of O_b is equal to the corresponding index in O_a, both operations refer to the same level of the tree and O_a is also an insert, meaning that both operations insert a semantic unit at the same index relative to the parent node. It is irrelevant in what order the two semantic units are inserted, but it is essential that they are inserted in the same order both locally and on the repository. We have chosen the order of insertion depending on the contents of the two operations. The second case to be considered is the one for which the index of O_a corresponding to the level of O_b is greater than or equal to that of O_b, in which case the index of O_a has to be increased.

In what follows we describe the exclude function.

```
    Algorithm exclude(Oa,Ob):O'a {
1:    O'a:=Oa;
      if (type(Oa)=NOP or type(Ob)=NOP) return O'a;
      if (level(Ob)>level(Oa)) return O'a;
      for (i:=0;i<level(Ob);i++)
          if (index(Oa,i)≠index(Ob,i)) return O'a;
2:    if (type(Ob)=insert) {
          if (index(Oa,level(Ob))=index(Ob,level(Ob)))
              if ((level(Oa)≠level(Ob)) or
                  (level(Oa)=level(Ob) and type(Oa)=delete))
                  type(O'a):=NOP;
          if (index(Oa,level(Ob)>index(Ob,level(Ob)))
              index(Oa,level(Ob))−−;
      }
3:    else if (type(Ob)=delete)
          if (index(Oa,level(Ob))≥index(Ob,level(Ob)))
              index(Oa,level(Ob)++;
      return O'a;
    }
```

The first argument of the *exclude* function, O_a, represents the operation that has to be transformed and the second argument, O_b, represents

the operation against which the exclusion has to be performed, i.e. the operation whose effects have to be excluded from O_a.

Block 1 in the *exclude* function deals with the cases when operation O_a does not need any transformation. As in the case of the include method, excluding *NOP* from an operation or excluding an operation from *NOP* has no effect on the result. Another case when O_a keeps its original form as result of the exclusion transformation is when O_b is of a finer granularity level than O_a. Operation O_a is not modified also if O_b does not refer to a node that is on the path from the root to the node referred to by O_a.

The next cases to be analysed are the ones for which O_b might actually have an effect on O_a. These cases are treated in blocks 2 and 3 of the *exclude* function.

Block 2 analyses the cases when O_b is an insert operation. As with include transformation, there are two cases to be considered. The first case occurs when the indices of the two operations corresponding to the level of O_b coincide. If O_a deletes the same node inserted by O_b, by excluding the effect of O_b, O_a has to be cancelled. If O_a inserts or deletes a node in or respectively from the subtree that has been inserted by O_b, by excluding the effect of O_b, operation O_a has to be cancelled.

The second case appears when the index of O_a corresponding to the level of O_b is greater than that of O_b. In this case, by excluding the effect of the insert operation O_b, the index of O_a has to be decreased by one.

Block 3 of the exclude transformation deals with the case when O_b is a delete operation. If the index of O_a corresponding to the level of O_b is greater than that of O_b, by excluding O_b, the index of O_a has to be increased.

The modified version of O_a is returned at the end of the procedure.

In the field of collaborative editing, a lot of work was done regarding the inclusion and exclusion functions and proving their correctness [52, 63]. In our approach we implemented simple transformation functions and the special cases that might cause problems are treated outside of the transformation functions. We have to mention that there are far fewer special cases to be analysed in the case of asynchronous communication than in real-time communication.

The modified form of the *transpose* function is presented in what follows. The arguments of the *transpose* function represent the two operations O_a and O_b that have to be transposed and the result of the function is the pair (O'_b, O'_a) representing the transposed forms of the initial operations.

```
    Algorithm transpose(O_a,O_b):(O'_b,O'_a) {
1:    O:=exclude(O_a,O_b);
      O'_a:=O_a;
      O'_b:=O_b;
2:    if (index(O_a,level(O_a))=index(O_b,level(O_b)) and
         type(O_a)=insert and type(O_b)=delete {
         O'_a:=NOP;
         O'_b:=NOP;
         return (O'_b,O'_a) ;
      }
3:    if (index(O_a,level(O_a))=index(O_b,level(O_b)) and
         type(O_a)=insert and type(O_b)=insert
         index(O'_a,level(O'_a))++;
      else
4:       if (index(O_a,level(O_a))=index(O,level(O)) and
            type(O_a)=insert and type(O)=insert)
            O'_a:=O_a;
5:       else
            O'_a:=IT(O_a,O);
      O'_b:=O;
      return (O'_b,O'_a);
   }
```

Block 1 of function *transpose* computes the effect of the exclusion of O_a from O_b. The operations O'_a and O'_b that have to be returned by *transpose* are initialised with the initial values of these operations.

The *transpose* procedure deals with the following cases. The first case described in block 2 occurs when O_a and O_b respectively insert and delete an element from the same position. If the transpose procedure is executed in the form presented in section 3.4.3, we would first exclude the effect of O_a from O_b. This would yield a NOP operation since otherwise O_b would attempt to delete content that does not exist. The problem comes at the next step, when the effect of the NOP operation has to be included into O_a. Performing an inclusion transformation of O_a against a NOP yields an unmodified O_a as a result. This yields an incorrect result since the effect of applying the modified O_b followed by O_a as a result of applying the transpose procedure should be the same as applying the original O_a followed by O_b. This does not hold if O_b becomes NOP and O_a remains unchanged. The solution that we have adopted was to handle separately this case in the *transpose* procedure and transform both O_a and O_b to NOP.

The second case described in block 3 occurs when the two insert operations O_a and O_b have the same position for insertion. For instance, consider that the initial document state is the empty document and that two op-

erations O_a =*insertChar(3,1,2,0,"x")* and O_b =*insertChar(3,1,2,0,"y")* are executed in sequence. After the execution of the two operations, the state of the document becomes *"yx"*. In order to perform the transposition, operation O_b should exclude from its context operation O_a, the result being the initial form of operation O_b, i.e. O'_b =*insertChar(3,1,2,0,"y")*. According to the first version of *transpose* function presented in section 3.4.3, O_a would have to include O'_b and result in O'_a =*insertChar(3,1,2,0,"x")*, due to the fact that the content of *"x"* is less than the content of *"y"*. This yields as a result the state *"xy"*, different from the initial string. The solution that we adopted was to increment the position of insertion of the first operation when it is transposed.

The third special case described in block 4 appears when O_a and O, the version of O_b which no longer contains the effect of O_a, are both *insert* operations applied at the same position. For example, consider the case of an empty document and that O_a=*insertChar(3,1,2,0,"y")* and O_b =*insertChar(3,1,2,1,"x")* are executed on this document state. The state of the document produced by the execution of O_a followed by O_b is *"yx"*. According to the first version of *transpose*(O_a, O_b) function presented in section 3.4.3, after excluding the effect of O_a from O_b, the result would be O'_b =*insertChar(3,1,2,0,"x")*. If the inclusion of O_a would simply be performed against O'_b, then O'_a would become O'_a =*insertChar(3,1,2,1,"y")*, as the content of O_a, i.e. *"y"*, is alphabetically greater than the content of O, i.e. *"x"*. The final result would not be correct since the effect of applying the modified O_b followed by the modified O_a would yield the state *"xy"* instead of *"yx"* obtained after executing the initial form of O_a followed by O_b. The solution that we adopted was that the new form of the operation O_a, i.e. operation O'_a, does not need to include the effect of O'_b, but just keeps the initial form of O_a. In the case of the example discussed above we would obtain O'_b =*insertChar(3,1,2,0,"x")* and O'_a =*insertChar(3,1,2,1,"y")*, the result being *"yx"*.

If none of the described cases above occurs, the *transpose* function keeps its original form presented in section 3.4.3. This case is treated in block 5.

Note that the *transpose* function is called only for operations of the same level. Therefore, inside *transpose* no check was done whether operations have the same level.

Note that the relations *exclude(include*(O_1,O_2)*,*O_2*)*=O_1 and *include(exclude*(O_1,O_2)*,*O_2*)*=O_1 do not hold true if, as result of computing *include(*O_1,O_2*)* or *exclude*(O_1,O_2), O_1 is cancelled, becoming O_1=*NOP*. Once an operation is cancelled it cannot be reactivated by excluding or including

the effect of the operation that cancelled it. For instance, suppose that the repository contains operation O_1 and a user executes operation O_2 in their local workspace. Before committing to the repository, the user has to update the local version of the document. Suppose that the two operations are conflicting and the user is choosing to keep the remote operation, the local operation being therefore cancelled. Operation O_2 will be stored in the repository as O_2=NOP. Suppose that another user is performing an update and a commit and operation O_1 has to be cancelled. In this case operation O_1 has to be transposed to the end of the history buffer and the operations following it have to exclude the effect of O_1. If operation O_2=NOP excludes operation O_1 from its context, the result remains NOP and the original form of operation O_2 before the inclusion of O_1 is not obtained.

Transformation functions adapted for linear structures can be used in our approach as explained in what follows. As shown in the update procedure, when merging is performed at the level of the current node of granularity i, the local level log LLL contains operations of granularity i and the remote log contains operations of granularity i composing the remote level log RLL and other operations that refer to the current node but of a finer granularity than i.

Operations from LLL and RLL are of the same level of granularity i, so the transformations of the operations from LLL against the operations from RLL and conversely modify the ith index of the position vector of the operations.

Each composite operation from LLL and RLL can be transformed into a simple operation, the position parameter of a simple operation representing the ith index of the position vector of the corresponding composite operation. Inclusion and exclusion transformation functions for simple operations can then be applied to compute the transformed positions of these operations. The transformed position of the simple operations will represent the ith index of the position vector of the transformed form of the corresponding composite operation.

The operations in RL that follow the operations in RLL are of a finer granularity than i and they have to include the effect of the operations in the new local log NLL. The operations in NLL are of granularity i and they might affect only the position of the operations in RL corresponding to level i. The operations in RL can be transformed into simple operations corresponding to level i. Transformation functions working for linear structures can be then applied to find the transformed form of the simple operations. The simple operations can be transformed back to their

Customisable Operation-based Merging of Hierarchical Documents

Abstract
Version control systems are widely used to support a group of people working together on a set of documents over a network. [...]

Conclusions
Our algorithm applie a linear merging procedure.

Figure 3.19: Example document

corresponding complex operations by modifying the ith index in the position vector of the composite operation with the position parameter of the transformed form of the simple operation.

Therefore, the same transformation functions working for linear structures, such as the ones in the SOCT2 algorithm, can be recursively applied in our treeOPT and asyncTreeOPT approaches.

3.4.6 Example

In what follows we will illustrate the asynchronous communication over text documents by means of an example. Assume the repository contains as version V_0 the document shown in Figure 3.19. The document is divided into sections, each section consisting of a list of paragraphs, each paragraph being formed by a list of sentences, each sentence consisting of a list of words and each word consisting of a list of characters.

Suppose a conflict arises between two operations concurrently modifying the same word and the policy of merging is that, in the case of conflict, local modifications are kept automatically. Further, assume two users check out version V_0 from the repository into their private workspaces. In order to explain in detail the functioning of the merging algorithm, we assume that the users are concurrently editing the paragraph p belonging to section s of the initial document *"Our algorithm applie a linear merging procedure."* All the operations will refer to section s, paragraph p and the form of the operations leaves out the reference for the section and for the paragraph. The first user performs operations O_{11} and O_{12}, where O_{11}=*insertChar("d",1,3,7)* and O_{12}=*insertWord("recursively",1,4)*. Operation O_{11} inserts the character *"d"* in the first sentence, third word at position 7 and operation O_{12} inserts the word *"recursively"* into first

sentence, as the fourth word in order to produce

> ...
> Our algorithm applie*d recursively* a linear merging procedure.
> ...

where [...] denotes the other sections of the document that have not been modified. The second user performs the operations O_{21}=*insertSentence("The approach offers an increased efficiency.",2)* and O_{22}=*insertChar("s",1,3,7)* in order to obtain

> ...
> Our algorithm applie*s* a linear merging procedure. *The approach offers an increased efficiency.*
> ...

If both users try to commit, but $User_1$ gets access to the repository first, $User_2$'s request will be queued. After the commit operation of $User_1$, the last version in the repository will be V_1="[...]*Our algorithm applied recursively a linear merging procedure.*[...]". DL_{10} representing the difference between V_1 and V_0 in the repository is obtained as a result of the linearisation of the history buffer distributed throughout the tree, $DL_{10} = [O_{12}, O_{11}]$.

When $User_2$'s request is processed, the repository sends to $User_2$ a message to update the local copy. To update the local copy of the document, the update procedure is applied. The local tree generated at the site of $User_2$ is traversed in a top-down manner. First we analyse the document level history. In this example there are no remote operations of section level to be merged. The update is then applied at the section level. Since the operations in our example refer to section *s*, we analyse the log referring to section *s*. There are no remote operations of paragraph level to be merged. The update is then applied at the paragraph level. The log of paragraph p is analysed and again there are no remote operations at sentence level, so the processing is applied at the sentence level. The local document contains two sentences, but there are no remote operations referring to sentence *2*, so we will analyse the merging for sentence *1*. Operation O_{12} is of word level, and because there are no local operations of word level, O_{12} will keep its original form and will be executed locally. The update procedure will be recursively applied for each of the words belonging to sentence *1*. We will analyse only the update applied to the third word of sentence *1*, since the remote logs corresponding to the other words

in the sentence are empty. The merge procedure will be applied to the list of operations consisting of O_{11} and the list consisting of O_{22}. O_{11} and O_{22} are conflicting and, according to the assumed policy, the local operation will be kept. As a result of this merging, for the third word of sentence 1, the new local log NLL, i.e. the list of operations to be transmitted to the repository, will be $[inv(O_{11}), O_{22}]$ and the new remote log NRL, i.e. the list of operations to be applied to the local copy of the document will be empty. Therefore, the new local version of the document in the workspace of $User_2$ will be *"*[...]*Our algorithm applies recursively a linear merging procedure. The approach offers an increased efficiency.* [...]*"* This version of the document represents also the new version V_2 of the document in the repository after $User_2$ commits. D_{21} becomes $D_{21} = [O_{21}, inv(O_{11}), O_{22}]$.

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

This example illustrated the fact that only a small number of transformations have to be performed using a tree-model of the text document with the local log distributed throughout the tree. The operations of a specific granularity do not need to be transformed against the operations of lower level granularity. The performance gain obtained by using a tree representation compared to using the linear representation of the text documents increases with the number of operations to be merged. In this example we have also seen that it is easy to define generic conflict rules involving different semantic units, such as specifying that concurrent insertions in the same word are conflicting. We mention that in the case of versioning systems such as CVS and Subversion, when $User_2$ is updating the local copy a conflict between the line *"Our algorithm applied recursively a linear merging procedure."* from the repository and the line *"Our algorithm applies a linear merging procedure. The approach offers"* from the workspace will be detected, as well as the addition of the line *"an increased efficiency."* $User_2$ will have to manually choose between the two conflicting lines and to add the additional line. Most probably, $User_2$ will decide to keep his changes and choose the line he edited, as well as adding the additional line. In order to obtain a combined effect of the changes, $User_2$ has to manually

add the word *"recursively"* in the local version of the workspace.

3.4.7 Conflict definition and resolution

In the tree document model conflicts can be defined at different granularity levels: paragraph, sentence, word or characters. In our current implementation we have defined that two operations conflict if they modify the same semantic unit: paragraph, sentence, word or character. The semantic unit is indicated by the granularity level chosen by the user. If a semantic unit is deleted, conflicts between the deleted unit and concurrent changes performed on the deleted unit can be defined only at a higher level. For instance, if a conflict should be detected between sentence deletion and word insertion in that sentence, the granularity level should be set to paragraph. If granularity is set at the sentence level, the sentence is automatically deleted. Conflicts can be visualised at the chosen granularity levels or at a higher level of granularity. For instance, if the user chooses to work at the sentence level it means that any two concurrent operations modifying the same sentence are in conflict. Conflicts can be presented at the sentence level so that the user can choose between the two versions of a sentence. It may happen that in order to choose the right version, the user has to read the whole paragraph to which the sentence belongs, i.e. the user can choose to visualise the conflicts also in the context of the paragraph or at a higher level. However, other rules for defining the conflicts can be specified by the implementation of the *semanticConflict* function presented in section 3.4.3. For instance, the *semanticConflict* function could check if some grammar rules are satisfied, a test that can be easily implemented using the semantic units defined by the hierarchical model.

In what follows we are going to describe the conflict resolution policies implemented by our system.

The main distinction between conflict resolution policies is given by whether conflicts are resolved automatically or manually. Automatic resolution of conflicts means that the user will not be prompted for a decision regarding any kind of conflict and the result is obtained by automatically applying a merging policy. Manual resolution, on the other hand, means that, if conflicts among operations arise, the user will be asked to manually choose one version or the other.

If the user chooses automatic resolution, the default behavior is that local operations will be the ones to be kept in case of conflict. Another policy for automatic resolution is to keep the remote operations.

Automatic conflict resolution policies can also be used in direct user

synchronisation. The user is offered two choices: synchronise as master or synchronise as slave. The former case implies that the local operations are chosen as the winners of conflicts, while the latter implies that the operations of the other user are kept should conflicts appear.

Concerning manual resolution policies, the user can choose between operation comparison and conflict unit comparison policies. The operation comparison policy means that when two operations are in conflict, the user is presented with the effects of both operations and has to decide which of the effects to preserve. This is, however, not a user friendly approach for presenting conflicts. Consider the case of two users concurrently inserting three characters into the same word. This yields three *insertChar* operations on each site. Since the lowest conflict unit is the word, all operations from one site are in conflict with all operations on the other. The user is prompted to choose between all pairs of conflicting operations in order to decide on the final version of the word. This is not a quick method of solving conflicts. Moreover, it is also likely that if one user modified a word by adding a few characters to it and the other user modified it in a different way, the user performing the merging probably wants either one word or the other, not a combination of them. This means that the user would choose either all local operations modifying the initial word or all remote operations modifying the word and not a combination of the two sets of operations. Therefore, this technique is more a debugging conflict resolution policy than one for a real user.

In the conflict unit comparison policy the user has to choose between the set of all local operations and the set of all remote operations affecting the selected conflict unit (word, sentence or paragraph). The user is presented with the two different effects achieved by applying all the local operations and all the remote operations, respectively, pertaining to the conflict unit. The user can then choose between the two alternatives. By changing the conflict unit, the user can decide how many choices to make when performing an update: the higher up in hierarchy the conflict unit, the fewer the number of choices.

Both manual conflict resolution policies can be used when performing direct user synchronisations.

The rules for the definition of conflict and the policies for conflict resolution can be specified by each user before an update is performed and they do not have to be uniquely defined for all users. Moreover, for different update steps, users can specify different definition and resolution merge policies.

3.4.8 The split/merge problem

The split/merge also remains an issue in asynchronous collaborative editing. The solution that we adopted in this case was to treat the split as a deletion of the old semantic unit and the insertion of the two new semantic units and the merge as the deletion of the two old semantic units and the insertion of the new semantic unit. However, the issue is not as critical as in the case of real-time editing, since there is not the possibility that operations interfere due to delays in the network as merging involves the reciprocal transformation of two ordered lists of operations. Moreover, some particular cases of split and merge can be detected and treated in special ways, as shown in section 4.2.2.

3.5 Related work

Starting with the dOPT algorithm of Ellis and Gibbs (1989), various algorithms using operational transformation for maintaining consistency in collaborative systems have been proposed such as Jupiter [77], NetEdit [125], adOPTed [88], GOT [108], GOTO [107], SOCT algorithms [101, 117, 116] and the operation effects relation approach [63]. As mentioned previously, all of these algorithms are based on a linear representation of the document whereas our algorithm uses a tree representation and applies the same basic mechanisms as these algorithms, recursively over the document levels.

All of these operational transformation algorithms keep a single history of operations already executed in order to compute the proper execution form of new operations. When a new remote operation is received, the whole history needs to be scanned and transformations need to be performed, even though different users might work on completely different sections of the document and do not interfere with each other. Keeping the history of all operations in a single buffer decreases the efficiency. In our approach, the history of operations is not kept in a single buffer, but rather distributed throughout the whole tree, and, when a new operation is transformed, only the history distributed on a single path of the tree is scanned.

Another important advantage is the possibility of performing not only operations on characters, but on elements of the tree, such as words, sentences and paragraphs, in the case of text documents. Transformation functions used in the operational transformation mechanism are kept simple as in the case of character-wise transformations, not having the complexity of string-wise transformations presented in [108]. Moreover, an insertion or

deletion of a node in the tree can be done in a single operation. Therefore, efficiency is further increased, because there are fewer operations to be transformed, and fewer to be transformed against. Moreover, the data is sent using larger chunks, thus the network communication is more efficient.

dARB [53] also uses a tree model for document representation, however it is not able to automatically resolve all concurrent accesses to documents and, in some cases, must resort to asking the users to manually resolve inconsistencies. Their approach is similar to the dependency detection approach for concurrency control in multi-user systems where operation timestamps are used to detect conflicting operations and the conflict is then resolved through human intervention [99]. dARB may also use special arbitration procedures to automatically resolve inconsistencies. For example, it uses priorities to discard certain operations, thereby preserving the intentions of only one user. In our approach, we preserve the intentions of all users, even if, in some cases, the result is a strange combination of all intentions. However, the use of colour in the editor interface provides awareness of concurrent changes made by other users and the main point is that no changes are lost. Moreover, because operations of delete and insert are defined only at the character level in this algorithm, i.e. sending only one character at a time, the number of messages sent through the network increases greatly. Further, if one site wins the arbitration it needs to send the state of the tree node on which concurrent changes were performed. There are cases when the winning site has to send not only the state of the tree node itself, but maybe also the state of the parent or other ancestor of the tree node. Sending whole paragraphs or even the whole document in the case that a winning site has performed a split of a sentence or a paragraph, respectively, is not a desirable option. In our approach, we tried to reduce the number of messages and transformations as much as possible, thereby reducing the notification time, which is a very important factor in groupware. For this purpose, our algorithm is not a character-wise algorithm, but an element-wise one, i.e. it performs insertions/deletions at different granularity levels. Moreover, we do not need retransmissions of whole sentences, paragraphs or of the whole document in order to maintain the same tree structure of the document at all sites.

The main difference between our approach and the existing versioning systems such as CVS [12] and Subversion [19] is that we have adopted a flexible way of defining conflicts. As opposed to a fixed unit of conflict (a line) adopted by these versioning systems, we allow that conflicts are defined using semantic units such as paragraph, sentence, word or character. The above mentioned versioning systems are using a state-based approach, as

opposed to operation-based merging adopted in our approach, which offers a better solution for conflict resolution and also for user activity tracking. In our approach we offer not only manual resolution for conflicts, but also other automatic resolution policies.

In [94] an operation-based merging approach has been proposed that uses semantic rules for conflict resolution. However, this approach can be applied only to a linear document representation. Our approach uses a tree representation and recursively applies any of the existing linear merging algorithms over the tree structure. In section 3.4 we presented our approach based on FORCE [94]. By using the tree model we deal with different semantic units.

XML documents conform to a hierarchical structure by definition. XML document merging has also been discussed in [119, 113, 18, 29]. These approaches are state-based and a detailed comparison of our approach with these approaches is carried out in chapter 5. Here we just mention that our approach is operation-based, offering better support for conflict resolution and for user activity tracking.

In [68] the authors propose using the operational transformation approach to define a general algorithm for synchronising file systems and file contents. File systems have a hierarchical structure, however, for the merging of the text documents the authors proposed using a fixed working unit, i.e. a block unit consisting of several lines of text. Transformation functions are defined for each pair of operations: addblock, deleteblock and moveblock. In our approach we deal with semantic working units (paragraph, sentence, word, character) and we use one generic transformation function that is applied for all semantical units. In [69], the same authors proposed a linear transformational approach applied to the collaborative editing of XML and CRC (Class, Responsibility, Collaboration) documents. In our approach the log is distributed throughout the tree rather than being linear, resulting in improved performance and a better support for the dynamic definition and resolution of conflicts.

4 Collaborative Editors Relying on the treeOPT Approach

In this chapter we present some implementation issues that we faced in the construction of collaborative editors relying on the treeOPT approach. We built a real-time collaborative text editor relying on the treeOPT approach and an asynchronous text editor application with a shared repository relying on the asyncTreeOPT algorithm. Therefore, the chapter is structured into two sections - first section describing the real-time application and the second section describing the asynchronous application.

4.1 A real-time collaborative text editor

This section presents some implementation aspects of the real-time collaborative text editor [73, 60] that we built. In subsection 4.1.1 we present the network communication module on which our application is based. The issue of user joining and leaving a collaboration group is analysed in subsection 4.1.2. The management of the identifiers assigned to sites and users is described in subsection 4.1.3. The issues involved in the parsing of inserted and deleted text are presented in subsection 4.1.4. An optimised document representation is presented in subsection 4.1.5. Subsection 4.1.6

describes the functionality of the collaborative text editor application.

4.1.1 Network communication module

The collaborative applications that we developed have been all implemented in Java for portability reasons. The real-time collaborative editor applications that we built all rely on a distributed architecture where each user performs local changes immediately and sends them afterwards to the other users. When a remote operation is received at a site, it has to be transformed against the operations that have been executed at that site, and only afterwards can it be executed at that site.

The applications that we built rely on a multicast communication [111] or TCP (Transmission Control Protocol) [111] communication based on a client/server architecture .

The multicast communication implies that the sites involved in the collaboration subscribe to a group, such that when a message is sent to the group all the sites that have subscribed receive that message. Unfortunately, multicasting is connectionless and is based on the UDP transport protocol which is unreliable. Some implementations of reliable multicasting services exist, such as [64, 65]. However, we used the standard Java multicast sockets for implementing the multicast communication.

As multicasting is unreliable and is usually used in LAN (Local area Network), we also designed the communication by using the connection oriented transmission. The TCP protocol ensures a reliable communication, but a connection has to be established before a transmission and the delay is high due to the confirmation messages. We used the TCP protocol with a client/server architecture, where the clients are represented by users involved in a collaboration and the server is responsible for the storage of the documents and of client information, the forwarding of packets between the users, and the implementation of the protocols of user joining and leaving a session. When the server receives data from a user, it forwards the data to the other users involved in the collaboration.

The application needs to deal with several types of packets:

- *Data packets* that are used to transmit the content of operations
- *Control packets* that are used for group session joining and leaving protocols, which will be described in section 4.1.2.
- *User information packets* that contain information about users, such as name and email address. These packets are also used during group

session joining protocol.

The packet format is illustrated in figure 4.1 and described in the following.

- *File ID* is the file identifier to which the packet refers. In the case of a data packet it specifies the file on which the operation must be performed. In the case of a control or user packet, it specifies the session file involved in the group session joining protocol.
- *User Key* is a unique key assigned to a user and it is used during the group session joining protocol.
- *Type* identifies the packet type
- *Data* represents the information carried out in the packet. In the case of data packets, it represents the timestamped operation to be performed. In the case of control packets it contains the control primitive, i.e. the joining and acknowledgement messages, as described in section 4.1.2.

Figure 4.1: Packet Format

4.1.2 Joining/leaving a group session

Any number of users can join a collaborative editing session. The users can leave or join the session at any time. We make the assumption that the network is reliable, i.e. the packets are not lost over the network.

We built the collaborative text editing application by using multicasting and therefore we describe next the joining protocol that we designed.

When users begin to edit a shared document, they must first obtain the permission of the other users already editing the document. Awareness is offered to the users involved in the collaboration, therefore the members of the group are informed when a member joins or leaves a group. When a new user joins a session, he/she must be provided with the last version of the modified document in order to create a local copy. In order to provide the final version of the modified document to the new user, the system

enforces a quiescence state, i.e. it falls quiet for an extended period of time. Quiescence means that users are blocked from editing the document. It has to be enforced each time a user joins or leaves a session, due to the timestamping scheme using state vectors. When a user joins or leaves a session, the number of components of the state vector has to increase or decrease, respectively. Therefore the history of operations becomes invalid and the consistency maintenance algorithm needs to be restarted.

In order to deal with dynamic joining and leaving of users, we proposed a user-join protocol illustrated in Figure 4.2.

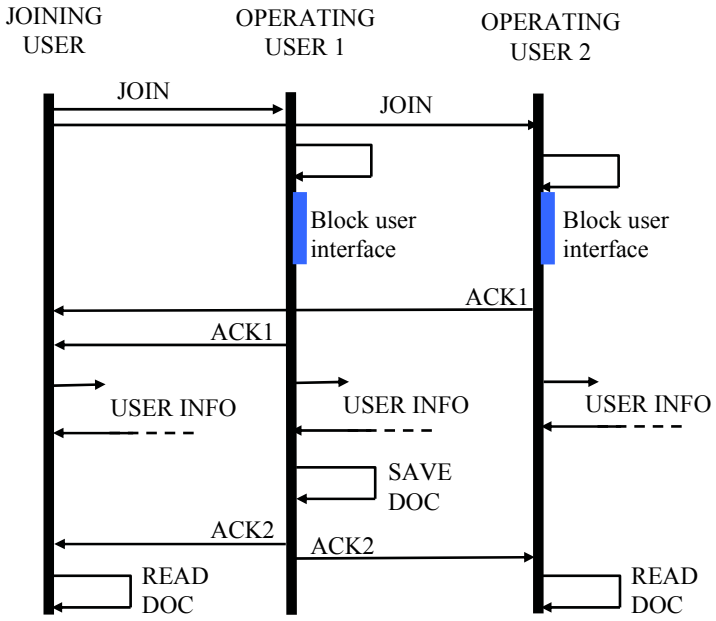


Figure 4.2: Protocol for joining

When a new user starts editing a document, a new join control packet is sent to the other users involved in the collaboration. After receiving a join packet, the users check whether they are currently editing the document or not. If yes, the users from those sites are blocked for a period of time from editing the document, quiescence is imposed, and all local copies of the document converge. After obtaining the state of quiescence, an acknowledgement ACK_1 is sent to the user that joins the session. After a timeout set for receiving the acknowledgements, the new user, as well as the other users involved in the collaboration multicast their user information. Each site can then update the local user tables containing information about the users involved in the collaboration. One of the users involved in the collaboration has to save the document so that the new user joining the session starts working on the modified version of the document. In order that only one user saves the document, we need to decide how to identify

the user who saves the document. The scheme that we propose relies on the fact that each site keeps the list of users ordered by their identifiers. The user with the smallest identifier among the "old" users is responsible for saving the local copy of the document and for notifying by ACK_2 message other users that the new copy of the document can be loaded.

4.1.3 Management of site and user identifiers

A mechanism to uniquely identify a site had to be provided. The generation of a unique identifier can be accomplished in one of the following ways:

1. Using a centralised authority which coordinates the process and issues unique identifiers to all sites requesting an identifier.

2. Establishing a protocol that can guarantee that the outcome is a unique identifier.

3. Generating an identifier based on some globally unique data available on all sites.

We first designed a protocol in which all sites, when joining the distributed environment, were given a unique identifier based on the order in which they were joining, as described in the previous section. While this provides a solution to the problem, the protocol involves exchanging a lot of messages and it might fail if the assumption of a reliable network does not hold, i.e. if packets are lost over the network. Therefore, generating a unique identifier without the need for such a protocol would be a better option. Subsequently, we decided to use an algorithm based on UUIDs (Universal Unique Identifier).

An UUID is a 128-bit number used to uniquely identify some object or entity on the Internet. Depending on the specific mechanisms used, a UUID is either guaranteed to be different or is, at least, extremely likely to be different from any other UUID that will be generated until 3400 A.D. The UUID relies upon a combination of components to ensure uniqueness. A guaranteed UUID contains a reference to the network address of the host that generated the UUID, such as MAC address, a timestamp and a randomly generated component. Because the network address identifies a unique computer, and the timestamp is unique for each UUID generated from a particular host, those two components should be sufficient to ensure uniqueness. However, the randomly generated element of the UUID is added as a protection against any unforeseeable problems.

Once a unique identifier has been assigned to a site, a message is sent over the network to inform the other sites involved in the collaboration that a new site has been created.

An issue that was raised from the use of UUIDs is concerning the state vectors in use for the operational transformation algorithms. In the join/leave protocol described in subsection 4.1.2, an integer between *0* and *N - 1*, where *N* represents the number of sites, was used in generating the site identity. Therefore, having a simple array or vector representing the state vector was sufficient. The new method of generating the site identity returns a 128-bit integer that is not suitable to be an index in any array or vector. However, it can be used as a key in a hashtable. By converting the state vector to a state hashtable, a fast lookup of site states can be ensured. While the method of constructing a state vector with a fixed number of elements required re-dimensioning of the array or vector, whenever a new site joins a session, the use of a hashtable automatically extends its capacity when new key/value pairs are added.

Documents are also identified by the use of UUIDs in order to ensure that a document is uniquely identified throughout the system.

4.1.4 Parsing

Two important operations in a collaborative text editor which uses a hierarchical document structure are the streaming that transforms the hierarchical structure into a linear one and the parsing that transforms the linear structure into a hierarchical one.

Streaming is reduced to the problem of flattening a tree into a list.

Parsing is performed when a document is opened for the first time, when it is reloaded during another user is log in, and when quiescence is detected or enforced and the re-parsing of the document is initiated.

In order to design a parser for the document, we first need to define a grammar describing the document structure. This is not straightforward, as a choice has to be made among several solutions, all of them featuring both advantages and disadvantages.

One issue is how to encode the element separators. The alternatives are to keep element separators as part of the elements themselves or to create elements containing only the separators. The choice of keeping separators as part of the elements they separate has the advantage that fewer elements are kept at each level of granularity. On the other hand, some other difficulties might arise. Suppose a sentence is the last one in a paragraph and it ends with a sentence separator such as *"?"* followed by a paragraph

separator such as *"\r\n"*. The question that arises is where this separator should be kept, whether it should be part of the last word or should it be a word part of the last sentence.

A possible solution encoding an element separator as part of the element itself is shown below.

WordSeparator = *WhiteSpace** ","? *WhiteSpace+* |
WhiteSpace+ ","? *WhiteSpace**

SentenceSeparator = *WordSeparator** [".", "!", "?", ":", ";"]

ParagraphSeparator = *SentenceSeparator** "\r\n"

NormalWord = *Character** *WordSeparator*

SentenceTerminationWord = *Character** *SentenceSeparator*

ParagraphTerminationWord = *Character** *ParagraphSeparator*

NormalSentence = *NormalWord** *SentenceTerminatingWord*

ParagraphTerminatingSentence = *NormalWord** *ParagraphTerminatingWord*

Paragraph = *NormalSentence** *ParagraphTerminatingSentence*

Document = *Paragraph**

The grammar rules are described using regular expressions [62]. In this notation, $*$ following an expression means the expression can appear 0 or more times. Similarly, + means 1 or more times, and ? means 0 or 1 times.

This solution would be suitable if the parsing algorithm could split words, sentences and paragraphs. Unfortunately, our algorithm does not allow element splitting and joining. Therefore, parsing the document in such a manner would give rise to an unreasonably large number of operations resulting in degenerated elements. For example, consider the case where a user modifies the sentence *"Our approach offers an increased efficiency !!"* by adding a few words at the end of the sentence and obtaining *"Our approach offers an increased efficiency compared to linear-based approaches !!"*. As the separators are kept as parts of words, *"efficiency !!"* is represented as a single word. Inserting the text *" compared to linear-based approaches"* would mean splitting the word *"efficiency !!"*. The only possibility is to insert *" compared to linear-based approaches"* as a substring of

a word, and therefore the degenerated word *"efficiency compared to linear-based approaches !!"* will result. The same problem arises each time an insert between an element and a separator (or inside a separator) is issued. Even worse, the same problem appears when we insert at the end of the document. As we see, there are a lot of cases where degenerated elements result. Even worse, thinking of the way people edit documents, we realised the fact that the cases mentioned above tend to happen extremely often as users usually do not insert text in the middle of a word, or even in the middle of a sentence, but rather they insert text at the end of elements, often between element and separator, or inside separators.

Due to the above described disadvantages, we propose a different solution, where each separator constitutes an element of its own. Therefore, each white space will be stored as a separate word, each dot as a separate sentence and each carriage return as a separate paragraph. The grammar that we adopted in our approach is presented below, where the regular expressions notation $[c]$ denotes a single character c and $[\hat{}c]$ represents a character that is anything else but c.

WordSeparator = *WhiteSpace* | “,”
SentenceSeparator = [“.”, “!”, “?”, “:”, “;”]
ParagraphSeparator = “\r\n”

NonSeparator = [ˆ{*WordSeparator*} {*SentenceSeparator*} {*ParagraphSeparator*}]

Word = *NonSeparator** | *WordSeparator*
WordS = *WordSeparator*
WordP = *SentenceSeparator*

Sentence = *Word** | *WordS*
SentenceP = *WordP*

Paragraph = *Sentence** | *SentenceP*

Document = *Paragraph**

Let us take an example in order to illustrate the structure of the document complying with the above grammar. Consider the document:

"Our approach offers an increased efficiency!!And a flexible merging support."

Figure 4.3 shows the structure of the document after parsing the text.

As we can see, even though in natural language the text contains only two sentences, in our grammar we obtain five different sentences. Increasing the number of elements is not particularly efficient. If someone inserts

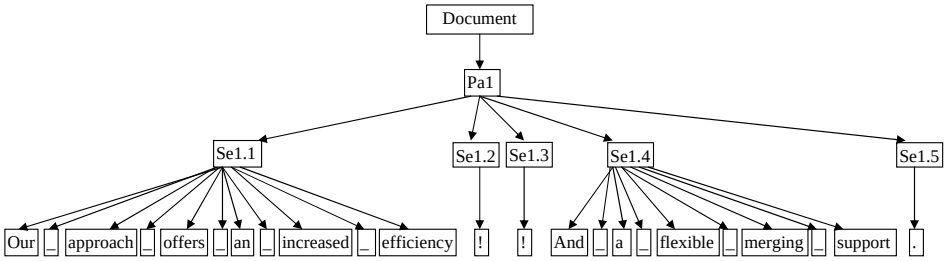


Figure 4.3: Structure of the document after parsing

this text into a paragraph, five *insertSentence* operations are generated and must be processed, which means that the consistency maintenance algorithm is applied a lot more often than strictly necessary. On the other hand, element degeneration is avoided in the majority of cases, as insertion between elements and their separators can be done in a natural way.

Taking into account the advantages and disadvantages of both grammars, the second one was finally chosen in order to avoid the large number of degenerated elements.

Parsing inserted and deleted text

A difficult problem involving parsing is the analysis of the inserted or deleted portions of text, in order to generate the correct insert and delete operations.

We illustrate the way parsing of deleted text is performed by means of an example. Consider that the initial text is *"Our approach offers an increased efficiency!!And a flexible merging support."* and that a user wants to delete the text *"n increased efficiency!!And a"*. The following operations have to be generated:

- *deleteChar(1,1,7,2)* –deletes character *"n"*, i.e. character 2 from paragraph 1, sentence 1, word 7
- *deleteWord(1,1,11)* – deletes word *"efficiency"*, i.e. word 11 from sentence 1, paragraph 1
- *deleteWord(1,1,10)* – deletes word "_ ", i.e. word 10 from sentence 1, paragraph 1
- *deleteWord(1,1,9)* – deletes word *"increased"*, i.e. word 9 from sentence 1, paragraph 1
- *deleteWord(1,4,3)* – deletes word *"a"*, i.e. word 3 from sentence 4, paragraph 1
- *deleteWord(1,4,2)* – deletes word "_ ", i.e. word 2 from sentence 4, paragraph 1

- *deleteWord(1,4,1)* – deletes word *"And"*, i.e. word 1 from sentence 4, paragraph 1
- *deleteSentence(1,3)* – deletes sentence *"!"*, i.e. sentence 3 from paragraph 1
- *deleteSentence(1,2)* – deletes sentence *"!"*, i.e. sentence 2 from paragraph 1

The procedure for parsing a deleted text is presented below.

Algorithm *determineDeleteOperations*(*startPoint*,*endPoint*){
  startPos:=position(startPoint);
  endPos:=position(endPoint);
  for *(l:=L;l≥1;l−−)*{
    *currentLevelStartNode:=*node corresponding to *startPos[0,...,l]*
    *currentLevelEndNode:=*node corresponding to *endPos[0,...,l]*
    upperLevelStartNode:= node corresponding to *startPos[0,...,l-1]*
    upperLevelEndNode:= node corresponding to *endPos[0,...,l-1]*
    if *(upperLevelStartNode=upperLevelEndNode)*
      Delete all child elements of *upperLevelStartNode* situated between *currentLevelStartNode* and *currentLevelEndNode*
    else{
      Delete all child elements of *upperLevelStartNode* situated after *currentLevelStartNode*
      Delete all child elements of *upperLevelEndNode* situated before *currentLevelEndNode*
    }
  }

The procedure *determineDeleteOperations* identifies the delete operations resulting from the deletion of the text starting at the character *startPoint* and ending at the character *endPoint*. The function *position* returns the position vector of the node given as argument, i.e. the path from the root to the node. For each level of granularity starting from the lowest granularity level and finishing with the highest granularity level, the elements of that granularity level that have to be deleted are identified. For instance, in the case of a text document with the levels of granularity document, paragraph, sentence, word and character, the processing starts at character level and finishes at paragraph level. At the character level, a check is done whether the characters corresponding to *startPosition* and to *endPosition* belong to the same word. The character corresponding to *startPosition* is identified by *startPosition[0,1,2,3,4]* denoting the path (*startPosition[0]*, *startPosition[1]*, *startPosition[2]*, *startPosition[3]*, *startPosition[4]*). The elements of the path specify the document, paragraph, sentence, word and character positions that identify the character. If the characters identified by *startPosition[0,1,2,3,4]* and by *endPosition[0,1,2,3,4]* belong to the

same word, the characters between them belonging to the word *startPosition[0,1,2,3]* are deleted and the processing finishes. Otherwise, the characters following the *startPosition[0,1,2,3,4]* character in the word *startPosition[0,1,2,3]* are deleted, as well as all characters preceding *endPosition[0,1,2,3,4]* character in the word *endPosition[0,1,2,3]*. The processing moves up one level in the tree in order to determine the words that have to be deleted between *startPosition[0,1,2,3]* and *endPosition[0,1,2,3]*. The processing is recursively repeated at higher levels till the paragraph level is reached when the deletion of paragraphs between *startPosition[0,1]* and *endPosition[0,1]* is determined.

In what follows we describe the way insertions are performed. Due to the fact that we wanted to provide the users with support for working at different levels of granularity, parts of text have to be parsed and transmitted over the network at different moments of time. For instance, if the level of granularity is sentence, inserted text is sent when a sentence separator is inserted. But there are cases when the inserted text has to be sent before insertion of a separator, such as the movement of the cursor to a new position in order to insert/delete text somewhere else or the elapsing of a period of time. Therefore, parts of text have to be parsed to determine a minimum number of insert operations. Our aim was to improve the efficiency and therefore, operations of insertion and deletion are not generated each time a new character is inserted and deleted, but rather the operations are defined on different granularity elements, such as characters, words, sentences and paragraphs. In this way, fewer operations are sent over the network. Due to the fact that our algorithm does not allow element splitting, insertions should be generated in a manner that complies with this constraint. The insertions we generate should not be of a coarser granularity than the syntactic elements neighboring the inserted text. For instance, if the insertion is performed inside a word, only character insertions inside the word are generated. If insertion is performed between two words, the only allowed operations are insertion of characters and of words. If insertion is performed between two sentences, the only allowed operations are insertion of characters, words and sentences. If insertion is performed between two paragraphs, insertions of characters, words, sentences and paragraphs are generated.

Let us consider the following example. The initial document is

Our approach offers an increased efficiency!!And a flexible merging support.

Further, a user first deletes the part of the text we show as crossed through

Our approach offers a~~n increased efficiency!!And a~~ flexible merging support.

in order to produce

Our approach offers a flexible merging support.

Note that, even though it seems the text resulting from the deletion consists of a single sentence, due to the way deletions are performed, the result consists of two sentences: *"Our approach offers a"* and *" flexible merging support."*. Now the user inserts the highlighted part of the text (written in italics) at the end of the first sentence

Our approach offers a*n operational transformation mechanism for consistency maintenance of hierarchical documents.*\r\n *For the asynchronous collaboration it offers a* flexible merging support.

The above illustrated insertion takes place between two different sentences. Therefore, as a result of parsing of the inserted text, paragraph insertions are not allowed to be generated, the maximum level for a generated insertion being the sentence.

In what follows the *maximum granularity level* will be the level corresponding to the coarsest allowed granularity for an insertion operation and the *minimum granularity level* will be the level corresponding to the finest allowed granularity for insertion.

When insertions are generated, the parser should take into account the maximum and minimum granularity levels and the types of elements neighboring the inserted text.

In our example, the insertion of text can be translated into the following operations:

- *insertChar(1,1,7,2,"n")*, i.e. insertion of character *"n"* in paragraph 1, sentence 1, word 7 and character 2;
- *insertWord(1,1,8,"_ ")*, i.e. insertion of word *"_ "* in paragraph 1, sentence 1, as word 8;
- *insertWord(1,2,1,"For")*;
- *insertWord(1,2,2,"_ ")*;
- *insertWord(1,2,3,"the")*;
- *insertWord(1,2,4,"_ ")*;
- *insertWord(1,2,5,"asynchronous")*;
- *insertWord(1,2,6,"_ ")*;

- *insertWord(1,2,7,"collaboration")*;
- *insertWord(1,2,8,"_")*;
- *insertWord(1,2,9,"it")*;
- *insertWord(1,2,10,"_")*;
- *insertWord(1,2,11,"offers")*;
- *insertWord(1,2,12,"_")*;
- *insertWord(1,2,13,"a")*;
- *insertSentence(1,2,"\r\n")*;
- *insertWord(1,1,9,"operational")*;
- *insertWord(1,1,10,"_")*;
- *insertWord(1,1,11,"transformation")*;
- *insertWord(1,1,12,"_")*;
- *insertWord(1,1,13,"mechanism")*;
- *insertWord(1,1,14,"_")*;
- *insertWord(1,1,15,"for")*;
- *insertWord(1,1,16,"_")*;
- *insertWord(1,1,17,"consistency")*;
- *insertWord(1,1,18,"_")*;
- *insertWord(1,1,19,"maintenance")*;
- *insertWord(1,1,20,"_")*;
- *insertWord(1,1,21,"of")*;
- *insertWord(1,1,22,"_")*;
- *insertWord(1,1,23,"hierarchical")*;
- *insertWord(1,1,24,"_")*;
- *insertWord(1,1,25"documents")*;
- *insertSentence(1,2,".")*;

Usually *"\r\n"* is parsed as a paragraph separator. However, paragraphs are not split and consequently *"\r\n"* is parsed as a degenerated sentence in the degenerated paragraph.

Inserted text is parsed differently, depending on the place of insertion, i.e. in a word, between words, between sentences, or between paragraphs,

and on the nature of neighboring elements, i.e. words, word separators, sentence separators or paragraph separators.

In the previous example, the left neighboring element was *"a"* , a word, and the right neighboring element was " " , a word separator. As a consequence, the leftmost part of the text was parsed as a sequence of characters belonging to the word *"a"*, the result word being *"an"*. The rightmost text was inserted as a word due to the fact that " " is a word separator and nothing can be "glued" to it.

We present below the procedure for the initialisation of parsing for insertion and then the insert generation algorithm.

```
Algorithm initialiseParseForInsertions {
  Tokenize text and obtain the token list;
  Determine the position of insertion in order to compute
            the maximum_granularity_level;
  Examine the left neighbour to determine the starting_level_of_granularity;
  Examine the right neighbour to determine the ending_level_of_granularity;
  parseForInsertion( token list, starting_level_of_granularity,
                     ending_level_of_granularity, maximum_granularity_level);
}

Algorithm parseForInsertion( token_list, starting_level_of_granularity,
                             ending_level_of_granularity,
                             maximum_granularity_level) {
1:  if (starting_level_of_granularity<ending_level_of_granularity){
      scanning order for list of tokens = left_to_right;
      current_granularity_level:=starting_level_of_granularity;
    }
    else{
      scanning order for list of tokens:=right_to_left;
      current_granularity_level:=ending_level_of_granularity;
    }

2:  scan token list until (
      (granularity_level(current token)≥current_granularity_level)
      or
      (token list finished));

    if (token list finished) {
      construct an element having current_granularity_level from the scanned list;
      perform insertion of the element;
      return;
    }

    construct element having current_granularity_level
            from scanned list, without current token;
```

```
        perform insertion of constructed element;

3:      current_granularity_level:=min( maximum_granularity_level,
                                        granularity_level(current_token));

        construct element having current_granularity_level from current token;
        perform insertion of constructed element;

4:      if (order for scanning was left_to_right)
           parseForInsertion( rest of token list, current_granularity_level,
                              ending_granularity_level, maximum_granularity_level);
        else
           parseForInsertion( rest of token list,starting_granularity_level,
                              current_granularity_level,maximum_granularity_level);
    }
```

The procedure *initializeParseForInsertions* tokenises the text, i.e. transforms the text into a list of tokens such as word, word separator, sentence separator and paragraph separator. Each of these tokens corresponds to a different level of granularity, i.e. character, word, sentence and paragraph, respectively. A string value is associated with each token.

The next step is to establish the maximum level of granularity the insertions might have, by examining where the insertion takes place. For example, if the insertion is inside a word, the maximum granularity level is character, if the insertion is between two sentences, the insertion can be of sentence level.

Next, the left neighbor is examined in order to determine the starting granularity level. The starting granularity level represents the minimum level of granularity the left-most insertion will have. Note that the relation $<$ is used in the sense of "finer than". The token type of the left neighbor has to be examined. In the previous example, the left neighbor for the insertion position was the word "a", which is of type normal word and, therefore, the starting level of granularity is character granularity. That is why the leftmost insertion *insertChar(1,1,7,2,"n")* is of character level. The right neighbor is the word " ", which is of type word separator, and consequently the rightmost insertion *insertWord(1,2,13,"a")* is of word level. Due to the fact that the right neighbor is a word separator, the word *"a"* could not be concatenated to it. However, the rightmost insertion could have been of type sentence, due to the fact that insertion occurs between two sentences. The purpose of the starting and ending levels of granularity is to establish lower bounds for the level of granularity of the rightmost and leftmost insertions respectively.

The *parseForInsertion* procedure establishes in block 1 the direction of token list scanning, from right to left or from left to right, according to the relationship between the starting and ending levels of granularity. If the starting level is lower, the list of tokens is scanned from left to right. The other case is symmetric.

In block 2 the list of tokens is scanned until a token having a granularity level greater or equal to the starting level is found. For example, if the starting level is sentence, scanning is performed until the first sentence or paragraph separator is found. Then all the words parsed until the separator are combined into a single sentence, and a sentence insert is issued.

In block 3 a new *current_granularity_level* is established. For example, if the separator that was found is of type paragraph separator, a paragraph containing only the separator has to be inserted. If the maximum granularity level allows it, a paragraph can be inserted. However, if the maximum granularity level is sentence, a paragraph insertion is not allowed, and consequently only an operation of sentence insertion can be issued. This is the reason why *current_granularity_level* is the minimum level between the *maximum_granularity_level*, and the level of the separator found.

After determining the *current_granularity_level* and inserting the separator itself, in block 4 the algorithm is applied recursively over the rest of the token list.

To explain why at the beginning of the procedure the token list has to be traversed from the side with the lower level of granularity, we provide the following example. Suppose that a paragraph separator was encountered and that the *maximum_granularity_level* is paragraph level and therefore a paragraph insertion is allowed. This means that the paragraph separator is inserted as a paragraph, and the procedure *parseForInsertion* is called recursively, with the *starting_granularity_level* being equal to the paragraph level. If the recursive call of the procedure *parseForInsertion* scans again the list of tokens from left to right, only whole paragraphs can be inserted. It might happen that not only entire paragraphs appear further in the list of tokens, but also some sentences that can be inserted into the right neighboring paragraph. Therefore, the scanning of the token list has to start with the side which has the lowest level of granularity.

4.1.5 An optimised text document representation

In the model that we proposed in chapter 3.1.1 each character is represented by a node. An optimised version of the model specific for text documents that we adopted in our implementation is to represent words as nodes that

contain strings and to have no nodes for the representation of characters. In this way memory can be saved by not storing each character as a separate object. Due to the fact that we do not work with fonts, sizes or other properties assigned to characters, a character node itself does not store any additional information other than the character itself. There is no log associated with a virtual character node because there are no operations which can be done at the character level in order to modify a character. The operations with the finest granularity in our model are the operations that modify a word and pertain to the word level, i.e. insertions and deletions of single characters. Therefore, by making a few adjustments we were able to reduce memory usage without any loss in generality. In the definition 3.1.1 of a node of the document, the fields whose interpretation we had to change in order to obtain the previously described changes, were *content* and *length*. Since character level nodes no longer exist, the *content* of word level 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 word level nodes is now the length of the content string.

4.1.6 Functionality of the text editor

The text editor allows several users to edit the same documents collaboratively. The graphical interface of the application is presented in figure 4.4. Awareness is provided to the users by the assignment of a different colour to each user. Due to the fact that text inserted by a user has the colour assigned to that user, users are aware of who is working on what part of the document. Users choose their assigned colour at the moment when they join the application. The mapping between users and their assigned colours can be checked at any time from the menu `Users` of the application. In figure 4.4 it is shown that two users modified the document `glossa.txt`, one user having assigned the blue colour and the other user having assigned the black colour.

Users can work on a set of documents and they can switch from one document to the other. The menu *Users* provides information about the users editing the current document, such as name, email address and a photo. The *Granularity level* menu allows users to switch working between different levels of granularity, as shown in figure 4.5. The levels of granularity are word, sentence and paragraph. Choosing a certain level of granularity means that modifications are sent to the other sites only when finishing editing the element of the chosen granularity level. For instance, if the granularity level is sentence and the user starts adding a new sentence and

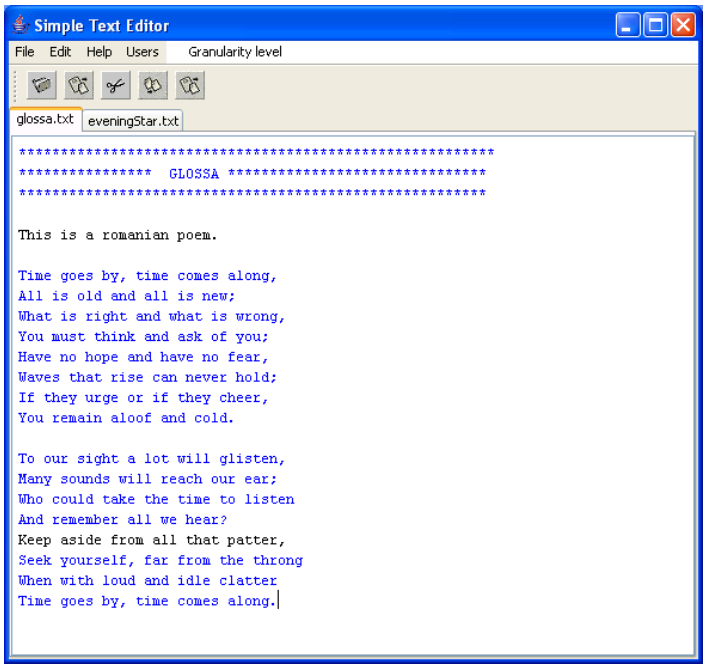


Figure 4.4: Graphical User Interface of the Text Editor

continuously inserts text, the text is sent when the user finished editing the sentence, i.e. when a sentence separator is inserted.

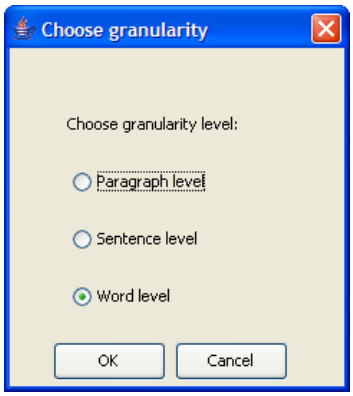


Figure 4.5: Granularity Chooser for the Text Editor

4.2 An asynchronous text editor

This section presents the asynchronous text editing application [24] based on the asyncTreeOPT algorithm.

4.2.1 Application of an operation to the tree structure

Although real-time text editing and asynchronous collaboration have a large number of similarities, different solutions have been adopted in some implementation aspects of the respective applications. The grammar describing a text document in the asynchronous collaboration is the same as the one we have used for real-time communication. However, the way operations are generated while editing is different. In real-time collaboration, due to the fact that users can work at different levels of granularity, the edited text is kept in a buffer and the operations are transmitted when a separator for that level of granularity has been reached. The text in the buffer is then parsed and corresponding operations are sent to the other sites. Operation generation adopts the solution for the split/merge problem presented in section 3.3.3 in which the number of degenerate elements is minimised. In asynchronous communication the complex parsing of inserted and deleted text can be avoided due to the fact that before merging a compression of the log of operations is performed. In this way each character insertion is kept in a log and in the compression phase the number of operations is reduced. The deletion operations are propagated up the tree when the last element of a unit is deleted. For instance, when the last character of a word is deleted, the deletion of the word is propagated up the tree. When a deletion of a unit is issued, the whole unit from the tree is deleted. Therefore, compression is not applied for delete operations.

In chapter 3.1.1 which presented the document model we mentioned that operation content is the tree node the operation refers to. However, operation content can also be the string representation of the node. None of those two approaches is clearly superior. If operation content is a tree structure, the tree structure has to be serialised to be sent over the network to the other sites. When content representation of the operation is done by means of a string, the sending over the network is simplified. On the other hand, the parsing of the string has to be performed at each receiving site.

We applied the same optimisation as in the case of real-time text editing by representing words as nodes that contain strings and not representing characters as separate nodes.

In what follows we present the way operations are applied to the document structure in order to update it. An operation has to be applied to the document structure when local operations are executed and when remote operations are integrated into the history buffer.

The *applyOperation* procedure that takes as argument the operation O that has to be applied and the current tree node CN on which the

operation has to be applied is presented below. When the procedure is initially called, the current tree node is the document root.

```
    Algorithm applyOperation(O,CN) {
1:    if (type(O)==NOP)
          return;

      ChildNo:=index(O,level(CN));

      if (level(O)!=level(CN))
          applyOperation(O,childAt(CN,ChildNo));
      else
2:        if (type(O)=insert)
              if (level(O)=wordLevel)
                  content(CN):=content(CN)[0,ChildNo-1] +
                                content(O)+
                                content(CN)[ChildNo,size(content(CN))];
              else
                  addChildAt(CN,ChildNo,parse(content(O),level(O)))
          else
3:            if (type(O)=delete)
                  if (level(O)=wordLevel)
                      content(CN)= content(CN)[0,ChildNo-1]+
                                   content(CN)[ChildNo+1,size(content(CN))];
                  else {
                      inverse(childAt(CN,ChildNo));
                      content(CN):=content(childAt(CN,ChildNo));
                      removeChild(CN,ChildNo);
                  }
    }
```

In block 1 of the procedure a check is performed whether the operation to be applied is a NOP operation. In this case the procedure returns as no changes have to be made to the document tree. Otherwise, *ChildNo* indicates the number of the child of the current node that the operation index points to. The traversal is done recursively down the tree until the node where the changes actually have to take place is reached. If the operation is an *insertWord*, for example, the changes have to take place when the sentence in which the word has to be inserted is reached.

When the node in the tree where the changes have to be performed is reached, a check is done for the type of the operation. In block 2 of the procedure a check is performed whether the operation to be applied is an insert. If the operation is an *insertChar*, i.e. the level of the operation is the word, the content of the word node has to be modified to contain the inserted character at the specified position inside the word. If the operation

is of a higher level, the parse method has to be called to parse the operation content. The content can be a paragraph, sentence or word, depending on the level of the operation. As result of parsing, the root of the subtree that correctly models the content of insert is returned. This root is added as a new child of the current node in the position indicated by *ChildNo*.

Block 3 of the procedure corresponds to the case of a delete operation. 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. A delete operation at a higher level in the tree removes the child indexed by *ChildNo* from the list of children of current node. Additionally, the content of the locally generated operation has to be correctly stored. The *inverse()* method executes the inverse of all operations from the log of the subtree received as parameter. As a result of an inverse operation, an insert is transformed into a delete operation and a delete operation is transformed into an insert. The operations are inverted in the reverse order than that in which they have been stored in the log. In this way the changes that might have been made locally to the subtree to be deleted are reversed. Therefore, the content stored in the delete operation reflects the content of the node excluding the changes executed on it. Since the operations executed on a node are lost when the node is deleted, this could lead to inconsistencies when the operation is sent to other sites if the content of delete operation is not updated as previously described. Once the *inverse()* method is called, the operation content can be safely stored.

Inversion is executed for remote operations as well. The procedure *applyOperation* is called as a result of an update process between a local site and the repository. The effect of inversion for remote operations is irrelevant because the remote operations, once executed locally, are discarded. Even if the content of the operation is changed, 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.

4.2.2 Split/join

Some cases of element splitting and joining can be eliminated when one of the parts of the element that is split or one of the elements to be merged was locally inserted by the same user.

For instance, suppose that the initial shared document contains the word *"algorithm"*. Suppose that the user starts editing the word by adding *"the"* at the beginning of the word, resulting in *"thealgorithm"*. The user

inserts then a space between *"the"* and *"algorithm"* and this is regarded as a split operation. The word *"thealgorithm"* is deleted and two new words *"the"* and *"algorithm"* are inserted. Concurrent changes performed on the word *"algorithm"* are lost due to split operation that is simulated by a delete of the initial word and the insertion of the two parts that were split. Such a case can be avoided by checking if all characters inserted before the space character that caused the split operation have been inserted by the local user. If this is the case, the characters starting at the first position in the word and ending at the space that generated the split constitute a word and this word is inserted without the need to simulate a split. In this case, the characters *"the"* from the word *"thealgorithm"* are deleted and the new words *"the"* and *" "* are inserted. In this way, concurrent modifications made by other users on the word *"algorithm"* are preserved.

The same verification can be carried out for the part following the space delimiter. For instance, consider the word *"algorithm"*. If *"that"* is added at the end of the word in order to obtain *"algorithmthat"* and afterwards a space is added between *"algorithm"* and *"that"*, a check can be made whether all characters starting from the position where space is inserted till the last position of the word were inserted by the local user. In this case, the sequence of characters *"that"* from the word *"algorithmthat"* are deleted and the new words *" "* and *"that"* are inserted after the initial word. Concurrent modifications made by other users on the word *"algorithm"* are preserved.

The case described above, word splitting, can be generalised to any unit in the document. Suppose we work on a unit of level i and a set of insertions and deletions of units at level $i + 1$ are performed on this unit. When a separator for the unit of granularity i is inserted, a check is done whether the child nodes of the unit of level i starting at the position with index 0 in the list of children and ending at the position identifying the delimiter were inserted by the local user. If this is the case, the child nodes are deleted from the content of the parent node and a new node containing these child nodes, as well as a new node containing the delimiter are inserted before the initial node. Another check is made to see whether the child nodes of the unit of level i starting at the position of the delimiter and ending at the last index were inserted by the local user. In this case, these child nodes are deleted from the content of the parent node and a new node containing the delimiter and a new node containing as children the deleted nodes are inserted after the initial node.

The join operation is symmetric to the split operation and we perform the same checking. When a join between two units of level i is performed,

a check is made whether one of these units was inserted locally and in this case the content of the unit that was locally inserted is added to the content of the other unit. For instance, consider that the initial version of the document contains the word *"Ana"* and in the local repository a user adds the word *"Maria"* in order to obtain *"Ana Maria"*. If the user joins the two words by deleting the space, the characters composing the word *"Maria"* are added at the end of the word *"Ana"*. In this way concurrent operations modifying the word *"Ana"* will be taken into account in the case of an update.

4.2.3 Log compression

Log compression is the mechanism of transformation of a log into an equivalent log with a reduced size. Two logs are equivalent if by sequentially executing the operations in the log on the same state, the same state is obtained. Log compression mechanisms have been proposed in [95, 15]. In our approach, the local log is compressed 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. For instance, several *insertChar* operations which insert characters in the same word, can be grouped into one single *insertWord* operation inserting the word formed by the target characters of the *insertChar* operations. In the same way we can combine several *insertWord* operations into a single *insertSentence* and several *insertSentence* operations into a single *insertParagraph*.

The log compression procedure is called before a user updates his local workspace with the changes in the repository, in order to reduce the size of the local log. It is also called before a user commits his changes to the repository, in order to send a reduced form of the local log to the repository for bandwidth minimisation and for storage space minimisation at the repository. Also, the procedure is called before a direct user synchronisation process, both on the side of the user who requests the synchronisation and on the side of the user who accepts it.

Algorithm *compressLog*(CN) {
  if *(level(CN)=Word)*
    return;

  for *(i:=0;i<noChildren(CN);i++)*
    compressLog(childAt(CN,i));

  Log:=log(CN);
  for *(i:= 0;i<size(Log);i++)* {

```
        O:=Log[i];

        if (type(O)=delete)
            continue;

        LInd:=index(O,level(CN));
        UnitDeleted:=false;
        for (j:=i+1;j<size(Log);j++) {
            Oj:=Log[j];
            LIndj:=index(Oj,level(CN));

            if (type(Oj)=insert)
                if (LIndj≤LInd)
                    LInd++;
            else
                if (type(Oj)=delete)
                    if (LIndj<LInd)
                        LInd−−;
                    else
                        if (LIndj=LInd) {
                            UnitDeleted:=true;
                            break;
                        }
        }

        if (UnitDeleted)
            continue;

        content(O):=toString(childAt(CN,LInd));

        emptyLog(childAt(CN,LInd));
    }
}
```

compressLog is a recursive procedure taking as parameter the current node CN for which the compression is performed. Initially, the procedure is called with the root of the document tree as the current node. The recursive call of the procedure ends when the leaves of the tree are reached. 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 the log of the current node is processed, all lower level logs have been compressed.

We are going to illustrate first the functioning of the *compressLog* procedure by means of an example. Consider the example of a user typing in the word *"hello"*. This generates an *insertWord(...,"h")* at the sentence level

and four *insertChar* operations at the word level, i.e. *insertChar(...,"e")*, *insertChar(...,"l")*, *insertChar(...,"l")*, *insertChar(..., "o")*. These operations have to be combined into a single operation *insertWord(...,"hello")*. We see that the insertion of a word is distributed at sentence and word level. What we need to do is to iterate over the log of operations associated with the current node and, if it is an insert operation, achieve the effect we described above.

Delete operations are skipped as delete operations are propagated up the tree when the last element of a unit of given granularity is deleted. For instance, the compression of several *deleteChar* operations into a *deleteWord* is achieved at the moment when the last character of the word is deleted, so this case does not need to be considered by the compress procedure.

The next analysis refers an insert operation O. The operation is in fact the insertion of a child of the current node. However, as it has been seen in the previous example, the content of the operation is not the current content of the child node. This is due to the fact that operations other than the insertion of the first unit element are currently stored one level lower. Therefore, the content of the operation has to be replaced with the current content of that child and the child log has to be emptied. Considering again the example with *insertWord(...,"hello")*, this translates to the fact that the content of the *insertWord(...,"h")* operation has to be changed from the current *"h"* to *"hello"*. Additionally, the four *insertChar* operations have to be deleted.

The position of the child whose content is needed is determined. Subsequent operations in the log might change the position of the child. Depending on the type of the operations, i.e. insert or delete, the position of the operation in the log is increased and decreased, respectively. Another case that needs to be handled is the one where one of the subsequent operations deletes the semantic unit which the current operation inserted. In this case the content of the insert operation no longer has to be modified. This is due to the way deletion operations are propagated up the tree that was shortly explained at the beginning of section 4.2.1 and in detail described in the presentation of the *applyOperation* procedure. In the example considered above, when *"hello"* is deleted, the content of the delete operation will be *deleteWord(...,"h")*. The pair insert/delete operation is consistent in this way.

The content of operation O is replaced with the content of the child whose position was found and the log of the child is emptied.

When the compression algorithm finishes its execution, all operations

which can be combined into higher level operations will have been combined and the distributed log is minimized with regard to the number of operations.

4.2.4 Description of the application

Using the asynchronous text editing application involves starting the server application where the repository resides and then starting the client applications that connect to the server. The current implementation allows that only one document together with its versions is stored in the repository, but it could be easily extended to allow the asynchronous editing of multiple documents. The server application listens for the requests of the client applications.

The client applications support the editing in isolation of the document in the repository and the synchronisation of the local version of the document with the version of the document on the repository. The interface of the client application is shown in figure 4.6.

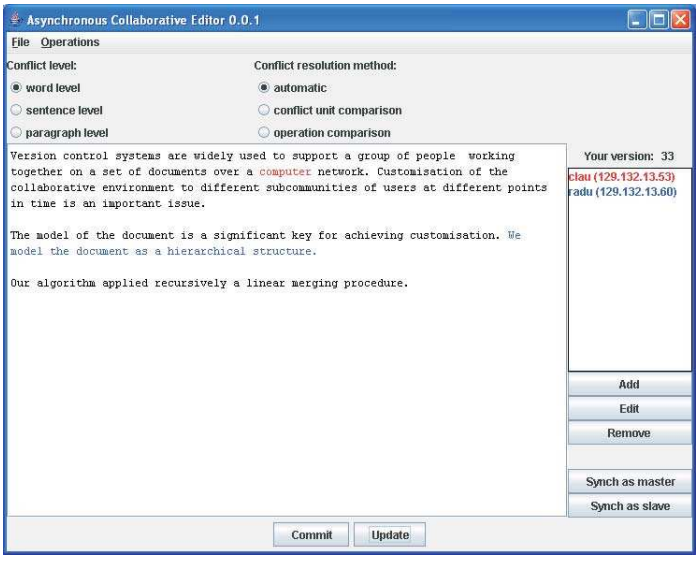


Figure 4.6: Interface of the client application

The user has the option to choose the semantic unit for the detection of conflicts, i.e. paragraph, sentence or word. If the conflict level is chosen to be the sentence, then two concurrent operations are conflicting if they modify the same sentence. The conflict resolution policies that the user can choose are automatic. Conflict unit comparison and operation comparison have been explained in section 3.4.7.

Version control systems are widely used to support a group of people working together on a set of documents over a network. Customisation of the collaborative environment to different subcommunities of users at different points in time is an important issue.

The model of the document is a significant key for achieving customisation.

Our algorithm applie a linear merging procedure.

Figure 4.7: Example document for the functionality of the application

On the right hand side of the GUI, the user can specify the addresses of the users whose changes the user wants to be aware of or against whose workspaces the user wants to synchronise the local copy of the document. Users can be added or removed from the list by using the `Add` or `Remove` buttons. Information about a user can be edited by using the button `Edit`. Colours can be assigned for each user in the list and the associated colours are used to distinguish the parts of the text modified by the users.

If the user wants to directly synchronise with one of the users in the list, the user is selected and one of the buttons `Sync as master` or `Sync as slave` is activated depending whether the synchronisation is performed as master or slave. The synchronisation as master means that in the case of conflict the local operation is kept. The synchronisation as slave means that in the case of conflict the operations performed by the remote user are kept. The synchronisation request is sent to the selected user and the remote user can accept or deny the request. If the request is accepted, the workspace of the remote user plays the role of a repository for the local user. The local version of the document will then include the remote changes.

In order to better understand the functionality of the application and the use of conflict definition and resolution, we are going to provide a scenario starting from a similar example to the one provided in section 3.4.6.

Suppose that the initial version of the document is the one illustrated in Figure 4.7.

Now, two users concurrently edit this version of the document by modifying the last sentence: *"Our algorithm applie a linear merging procedure"*. The first adds the character *"d"* at the end of the word *"applie"* and inserts the word *"recursively"*, as illustrated in figure 4.8. The second user

is adding *"s"* at the end of the word *"applie"* and a new sentence *"The approach offers an increased efficiency."*, as illustrated in figure 4.8.

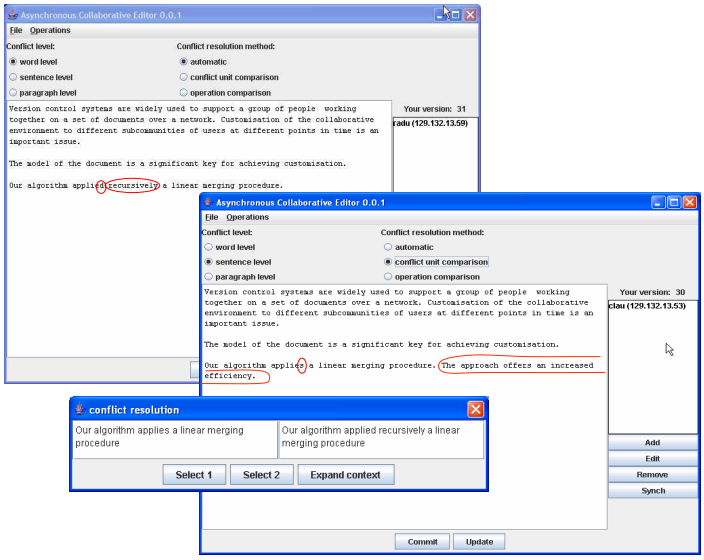


Figure 4.8: Scenario illustrating the functioning of the asynchronous collaborative text editing application

Suppose that after performing their modifications both users try to commit their changes to the repository, and the request of the first user is fulfilled. When the second user tries to commit, the local version of the document has to be updated. If the user has chosen the conflict level to be sentence and the policy for merging to be conflict unit comparison, the user is presented with the two sentences that are in conflict, as illustrated in figure 4.8. Suppose that the user chooses the variant corresponding to the local version. After the second user performs a commit, the new version of the document in the repository is illustrated in Figure 4.9:

If the second user had had chosen word level granularity, the conflict would have been detected for the word *"applie"*. The two words in conflict would have been *"applied"* and *"applies"*. Suppose that the variant corresponding to the local version is chosen. After performing a commit, the new version of the document in the repository is illustrated in Figure 4.10:

This chapter presented the collaborative text editing prototypes that we built based on the treeOPT approach. In the next chapter we describe how the treeOPT approach was used for the collaboration using XML documents.

Version control systems are widely used to support a group of people working together on a set of documents over a network. Customisation of the collaborative environment to different subcommunities of users at different points in time is an important issue.

The model of the document is a significant key for achieving customisation.

Our algorithm applie**d** a linear merging procedure. **The approach offers an increased efficiency.**

Figure 4.9: The new version of the document in the repository for the case of sentence level granularity

Version control systems are widely used to support a group of people working together on a set of documents over a network. Customisation of the collaborative environment to different subcommunities of users at different points in time is an important issue.

The model of the document is a significant key for achieving customisation.

Our algorithm applie**s recursively** a linear merging procedure. **The approach offers an increased efficiency.**

Figure 4.10: The new version of the document in the repository for the case of word level granularity

5 Consistency Maintenance for XML Documents

In this chapter we describe how the same treeOPT approach presented in chapter 3 was applied to maintain consistency in the case of the collaboration for XML documents.

5.1 Requirements

In this section we describe some features present in existing single-user XML editors and that should be offered also by collaborative XML editors. For example, single-user XML editors, such as XML Spy [6] or XML editor from Stylus Studio [4], offer features of auto-completion to speed up and make more convenient editing of well-formed XML documents. In collaborative editing a necessary condition for obtaining a well-formed reconciled document is that the two XML documents to be merged are well-formed. Therefore, our goal was to build a collaborative editor that uses auto-completion during editing in order to maintain well-formed documents.

Consider that a user edits an XML document, e.g. by adding the line '`<test>hello world</test>`' character by character. In this way, the XML document will not be well-formed until the closing tag is completed. Our editor provides support to insert complete elements, so that the operations can be tracked unambiguously at any time in the editing process. For

instance every time the user inserts a '<' character, the insertion of '<></>' is performed. Of course an empty tag, such as '<></>' is not a valid XML element, but at least it constitutes a good support for the creation of a new valid element.

Additional rules for the deletion of characters have to be provided. A user should be prevented from deleting parts of the element structure, such as the begin or end tag, unless the whole element is deleted. For instance, the user cannot delete '</test>' from an element '<test>hello world</test>'.

Another issue regarding editing of elements are the two different forms that an element can take: the form containing both the opening and closing tags such as '<test></test>', or the form of an empty element such as '<test/>' containing only the closing tag meaning that no further child elements are defined. The user is prevented from directly deleting the closing tag ('</test>'). Instead the user can insert a '/' character at the end of the starting tag ('<test>' $\Rightarrow$ '<test/>') in order to tell the system that the element should be transformed into an element containing only a closing tag. The operation is not performed if the element contains other *child nodes*. On the other hand, the deletion of the '/' character in an empty element leads to the creation of an element containing a begin and end tag.

The editor that we built supports users editing XML documents by automatically validating the syntax of the documents. The user can format the document, i.e. insert white spaces to make the content more readable, which is not possible using a graphical interface where the user has only a structured view of the content.

In what follows we present our model for XML documents and the set of operations used in the editing process of XML documents. In the case of text documents the operations that can be performed are the insertion and deletion of elements of different granularities, such as paragraphs, sentences, words and characters. Regarding the classification of node types in XML documents we have adopted two solutions.

In our first approach, we classified the nodes of XML documents into various types such as element, attribute, word and separator nodes, in order to allow the specification of rules for the definition and resolution of conflicts. The types of operations that can be applied are insert and delete targeting one of the types of nodes, such as *insertElement*, *deleteElement*, *insertAttribute*, *deleteAttribute*, *insertWord*, *deleteWord*, *insertChar* and *deleteChar*. The drawback of the definition of a large set of operations is that inclusion and exclusion transformations have to be specified for

the whole set of operations. But, as already mentioned, the large set of operations offers advanced possibilities to define conflict rules.

In our second solution we wanted to show the generality of the treeOPT algorithm regarding its application for both text and XML documents. To this end, we built a collaborative editor that works for both text and XML. XML documents have been structured in the same way as text. The structure of a text document is a tree whose nodes are of a single generic type, each node containing a set of child nodes and the content of the document being kept in the leaf nodes. In a similar way, a node of the XML document contains a list of children. Subsequently, an XML element contains a list of child element nodes and a list of child attribute nodes. However, the difference between the structure of XML and text documents is that internal nodes of the XML structure have content. For instance, for an XML element its name is kept as content of the node that represents the element. In addition to insert and delete operations performed on XML nodes, a modify operation has been defined in order to modify the content of an XML node. The modify operation could be simulated by a delete operation of the node to be modified and an insert operation of the node with the new content. However, in this way, operations concurrent with the modify operations targeting one of the child nodes are discarded, as operations targeting a subnode of a node to be deleted are not taken into consideration. Modify operations introduce a new possibility of defining and resolving conflicts. Two concurrent modify operations applied on the content of an element such as the name of an element or the name of an attribute could be considered conflicting and only one of the operations could be performed. The decision about which operation to execute is taken according to a priority-based policy, i.e. the operation performed by the user with the highest priority is performed. In this way a single user intention is maintained and not a combination of partial intentions of several users.

We have applied the first approach in an asynchronous XML editor and the second approach in a real-time XML collaborative editor. Therefore, in what follows we describe in turn the approaches that we used for consistency maintenance of XML documents in both real-time and asynchronous modes of communication.

5.2 Asynchronous collaboration for XML

In this section we present how the asyncTreeOPT approach was applied for maintaining the consistency over XML documents in the asynchronous communication using a shared repository [50, 48].

5.3 Node types

In the following we present all types of nodes that we used to structure XML documents. The *root node* is a special node representing the virtual root of the document. The user cannot perform operations on this node.

Processing nodes can be used to define processing instructions in the XML document such as '`<?xml version="1.0"?>`'. In order to keep the XML valid and to allow insertions of whole elements, the insertion of processing nodes is restricted to complete processing nodes, i.e. '`<??>`' and the deletion of elements referring to the structure of a processing node can be done only if the whole processing node is deleted.

Element nodes represent XML element structures and they consist of an *element name*, as well as some optional *attribute* and *child nodes*. For the following element node '`<test att="val">hello world</test>`', the string '`test`' is the *element name*, '`att="val"`' is an *attribute node* and '`hello world`' is composed of three *child nodes*, namely two *word nodes* and one *separator node*. Similar to processing nodes, in order to ensure well-formed XML documents, only complete element nodes having the form '`<></>`' are allowed, and the deletion of characters modifying the structure of the element node is restricted. Element nodes present a further issue, as the element name should not be different in the opening tag and the closing tag. As the update of the element name is an atomic operation, the editor alters the element names automatically whenever the user adds/removes some characters to/from the tag name.

Attribute nodes can be used either by the processing nodes or the element nodes and they consist of a single attribute string. To support the user, the editor will insert the '`=""`' characters automatically whenever the user adds a new attribute.

Separator nodes are used to preserve the formatting of the *XML* document and they represent *white spaces* and *quotation marks*. Separator nodes consist of a single character and the user can only insert and delete separator nodes, but not update the existing ones. Therefore, the system does not deal with conflicts concerning separator nodes, as it does for

example for *word nodes* or *attribute nodes*.

An element node is composed of *word* nodes. The element and attribute nodes of the XML structure that we adopted are similar to the element and attribute nodes in the DOM (Document Object Model) model [120]. However, for the text nodes in the DOM model that include white spaces and words, some further granularity has been added by the introduction of two node types, i.e. the word node and separator node.

5.4 Operations

In this section we present the operations used to describe the actions performed by users during XML editing. The operations have been chosen to be as minimal as possible, but to allow a flexible definition and resolution of conflicts. Even though the operations we defined for text and XML documents differ, the mechanism for consistency maintenance is the same.

For XML editing, the set of operations that have been defined is presented in what follows.

- *insertProcessing* inserts a new processing node.
- *insertElement* inserts a new element node that can either be a child of the *root node*, or a child of another *element node*.
- *insertAttribute* inserts a new attribute node that can either be added to a *processing node* or to an *element node*.
- *insertWord* inserts a new word node that can be added to any *element node*.
- *insertSeparator* inserts a new separator node. In order to maintain well-formed documents, the user is restricted to splitting only *processing targets*, *elements* or *attribute names* by means of separators.
- *insertChar* inserts a character that can be added to update *processing targets*, *element names*, *attributes* and *words*.
- *insertClosingTag* adds a closing tag.
- *deleteProcessing* deletes a processing node.
- *deleteElement* deletes an element node.
- *deleteAttribute* deletes an attribute node.

- *deleteWord* deletes a word node.
- *deleteSeparator* deletes a separator node.
- *deleteChar* deletes a character that can be removed from processing, element, attribute and word nodes.
- *deleteClosingTag* removes a closing tag. If an element consists of an opening and closing tag, but does not have any child nodes, this operation transforms the element into an empty element having only a closing tag.

The two operations *insertClosingTag* and *deleteClosingTag* were considered as a user may want to keep different forms for the representation of empty elements in a certain document and does not want to have an implicitly established form for the representation of empty elements. Our solution of considering both representation forms is more general than the solution of having a single form for the visualisation of empty elements.

Operations targeting child elements or attributes such as *insertElement*, *deleteElement*, *insertWord*, *deleteWord*, *insertAttribute* and *deleteAttribute* change the element structure, while operations targeting the tags of an element such as *insertClosingTag*, *deleteClosingTag*, *insertChar* and *deleteChar* change the content of the element. Operations that change the element structure have to be kept in the history associated with that element. In the same way, an operation *insertChar* or *deleteChar* targeting a character of a word are kept in the history buffer associated with that word.

The main decision that we faced was where to keep operations that change the name of an element. We decided to keep these types of operations in history associated with the node they refer, for the following reasons. Consider an empty element containing the begin and close tags. Further consider that a user is deleting the closing tag. Consider that a second user inserts a child element into the empty element. Operations of deletion of a closing tag and of insertion of elements as direct children of the element whose closing tag has to be deleted cannot be both applied. As seen in section 5.1, the execution of one of these operations will make impossible the execution of the other operation. We have chosen to cancel the *deleteClosingTag* operation and to keep the inserted elements, due to the fact that a *deleteClosingTag* operation means simply to rewrite the form of an empty element. Due to the fact that operations targeting closing tags have to be transformed against operations targeting child elements and vice-versa, we had to keep these operations in the same history buffer.

Moreover, the *deleteClosingTag* operation is issued by specifying a "/" at the end of the name of the empty element in the begin tag of the element. The *insertClosingTag* operation is issued by deleting the "/" at the end of the name of the empty element. Therefore, the *deleteClosingTag* and *insertClosingTag* are implemented as operations of character insertion in the name of the empty element. Therefore, operations targeting closing tags and characters in the element name are kept in the history buffer associated with that element.

5.5 Adapting asyncTreeOPT to XML

As we saw in section 3.4.3, if an operation O has to be cancelled and it has to be removed from the log, the operation has to be transposed to the end of the log. Transposition of two operations changes the execution order of the operations and transforms them such that the same effect is obtained as if the operations were executed in the initial order.

The FORCE approach considers that a transposition between two operations can always be performed. For operations applied on strings this fact holds true if the two operations do not have overlapping ranges. In FORCE operations in the log are transformed into non-overlapping operations by a compression procedure [95].

In our asyncTreeOPT approach applied to text documents described in section 3.4.4, we compressed the log, but just to transform operations of lower granularity into operations of higher granularity. For example, we compress a sequence of word insertions composing a sentence into an insertion of the sentence. We did not compress the insertion of a unit followed by the deletion of that unit, but we considered the case of an insertion of an element followed by the deletion of that element as a special case in the *transpose* function. Recall that we recursively applied the FORCE linear merging procedure at each document level. We therefore perform merging between operations of the same granularity level. Overlapping of the target range of the operations can happen only in the case of an element insert and of a delete targeting the same element. When a transpose between the two operations has to be performed, the result of the transformation is a pair of *NOP* operations, the result being equivalent to the compression of the two operations.

However, generally, ordering constraints between operations exist which do not allow operations to be executed in reverse order. This general case cannot be resolved by a compression procedure. In what follows we

present the cases that we encountered in the editing of XML documents that restrict the change of order between operations and the solutions that we adopted.

Relations of *dependency* or *before* constraints restrict the possibility of changing operation order. Consider an empty element of the form '`<elem/>`'. To insert a child element of this element, an operation *insertClosingTag* has to be issued. The operation *insertElement* of child element insertion, such as '`<subelem></subelem>`', in order to obtain '`<elem><subelem></subelem> </elem>`', is said to *depend* on operation *insertClosingTag*. This means that operation *insertElement* could not have been issued if *insertClosingTag* had not been executed.

Further, consider that the element '`<subelem></subelem>`' is deleted by issuing the operation *deleteElement*, and the operation *deleteClosingTag* is issued for the element '`<elem></elem>`' in order to obtain '`<elem/>`'. Between the operations *deleteElement* and *deleteClosingTag* there is a before constraint, meaning that the order between the operations should be maintained, i.e. the *deleteElement* operation and any other operation referring to a child element of a certain node should be executed before the *deleteClosingTag* operation applied on that node.

Therefore, the operations of *insertElement*, *deleteElement*, *insertWord*, *deleteWord*, *insertProcessing*, *deleteProcessing* included between *insertClosingTag* and *deleteClosingTag* operations and targeting child nodes of the element targeted by *insertClosingTag* and *deleteClosingTag* cannot be transposed to a position outside the range defined by *insertClosingTag* and *deleteClosingTag*.

The solution that we adopted was to detect the cases when the removal of an operation would result in making the transposition of that operation violate existing ordering constraints. The history associated with an element node contains the operations targeting child nodes, operations of *insertClosingTag* and *deleteClosingTag* and operations targeting characters that compose the name of the element node. For the histories associated with an element node, operations that target characters do not need to be transformed against operations targeting attributes and conversely. Moreover, operations that target attributes or characters do not need to be transformed against operations of *insertClosingTag* and *deleteClosingTag* and other operations targeting child nodes of the element node. The operations targeting characters have to be transformed against other operations targeting characters, and the operations targeting attributes have to be transformed against other operations targeting attributes. *insertClosingTag* and *deleteClosingTag* and other operations targeting child nodes of

the element node have to be transformed against other operations of these types.

When an update is performed, operation logs are compressed and the pairs of operations of deletion and insertion of the same child elements are eliminated, as well as pairs of *insertClosingTag* and *deleteClosingTag*. When *insertClosingTag* has to be removed and by the transpose mechanism it has to be transposed against a dependent operation, it is removed from the log together with all its dependents. When an operation of deletion of a child element has to be removed and it has to be transposed against a *deleteClosingTag* operation, the operation of deletion of the child element together with the *deleteClosingTag* operation is removed from the log. The removal of these operations is performed taking into account the fact that these operations do not have any impact on the operations targeting characters or attributes. Moreover, there are no other operations targeting child elements or other *insertClosingTag* and *deleteClosingTag* following the last removed operation in the log that can be affected by the removal of the above mentioned operations. When an *insertClosingTag* operation targeting a node N has to be transposed against a dependent operation, it is removed together with all dependent operations, i.e. all operations targeting the children of N. Therefore, no other operations targeting child nodes are left in the log. Moreover, no other *deleteClosingTag* operations are present in the log as a pair of *insertClosingTag* and *deleteClosingTag* would have been compressed. Also, no other *insertClosingTag* operation follows the *insertClosingTag* operation, as these two operations should be separated by a *deleteClosingTag* and in this case the pair *insertClosingTag* - *deleteClosingTag* should have been compressed. No other operations targeting child elements follow a *deleteClosingTag*, as one is not allowed to insert children after the ending tag is deleted, unless an *insertClosingTag* is issued. If an *insertClosingTag* had occurred after a *deleteClosingTag*, the pair of operations would have been compressed. Therefore, the removal actions that we considered are safe. The inverses of the operations that have to be removed are appended at the beginning of the new local log that in the case of a commit operation represents the delta between the new version of the document in the local workspace and the last committed version in the repository.

5.6 Transformation functions

The principles of the transformation functions for XML documents are similar to those for text documents described in section 3.4.5. In this section we present the transformation functions applied for operations that refer to the same level of granularity. Therefore, the transformation functions are not written for composite operations that are characterised by a vector of positions indicating the target path in the tree, but for simple operations that are characterised by a single position corresponding to a level of granularity.

The precondition of an inclusion transformation $include(O_a, O_b)$ is that O_a and O_b are defined on the same context. Due to this precondition and the fact that the condition of performing an *insertClosingTag* or a *deleteClosingTag* is that the target node has no child elements, the *include* function does not need to analyse the transformations between a *deleteClosingTag* and the operation of deletion of a child of the target node, such as *deleteElement*, *deleteProcessing* or *deleteWord*. It is not possible that *deleteClosingTag* and the deletion of a child of the target node have the same context, as *deleteClosingTag* requires that the target element has no children and the deletion of the child of the target node requires the existence of that child. In the same way, an *insertClosingTag* operation does not need to be transformed against an *insertElement*, *insertProcessing*, *insertWord*, *deleteElement*, *deleteProcessing* and *deleteWord*. *insertClosingTag* requires that the target element is empty, consisting only of an ending tag, while the operation inserting a child node requires that the form of the target element has a begin and end tag. An *insertClosingTag* operation does not need to be transformed against a *deleteClosingTag* operation referring to the same element as it is not possible that the two operations have the same context.

The inclusion transformation function for XML documents is presented below.

Algorithm *include*(O_a,O_b,*increment*):O'_a {
1: $O'_a := O_a$;
if (*type*(O_b)=*NOP* or *type*(O_a)=*NOP*) return O'_a;
// closing tag operations
2: if (*type*(O_b)=*deleteClosingTag*){
if (*type*(O_a)=*deleteClosingTag*)
O'_a:=*NOP*;
if (*type*(O_a)=*insertElement* or *type*(O_a)=*insertWord* or
type(O_a)=*insertProcessing* or
(*type*(O_a)=*insertSeparator* and *typeTarget*(O_a)=*ChildIndex*))

```
        return insertClosingTag ⊕O'a;
      return O'a;
    }
3:  if (type(Ob)=insertClosingTag){
      if (type(Oa)=insertClosingTag)
        O'a := NOP;
      return O'a;
    }
4:  if (type(Oa)=deleteClosingTag){
      if (type(Ob)=insertElement or type(Ob)=insertWord or
        type(Ob)=insertProcessing or
        (type(Ob)=insertSeparator and typeTarget(Ob)=ChildIndex)){
        O'a := NOP;
        return O'a;
      }
      if (type(Ob)=insertChar and index(Ob)≤index(Oa))
        index(Oa)++;
      if (type(Ob)=deleteChar and index(Ob)<index(Oa))
        index(Oa)−−;
      return O'a;
    }
5:  if (type(Oa)=insertClosingTag){
      if (type(Ob)=insertChar and index(Ob)≤index(Oa))
        index(Oa)++;
      if (type(Ob)=deleteChar and index(Ob)<index(Oa))
        index(Oa)−−;
      return O'a;
    }
6:  // other operations
    if (typeTarget(Oa)!=typeTarget(Ob) return O'a;
    if(type(Oa)=type(Ob) and (type(Oa)=deleteElement or
      type(Oa)=deleteProcessing or type(Oa)=deleteWord or
      type(Oa)=deleteSeparator or type(Oa)=deleteAttribute or
      type(Oa)=deleteChar) and index(Oa)=index(Ob)) {
      O'a:=NOP;
      return O'a;
    }
    if (index(Ob)≤index(Oa))
      if (type(Ob)=deleteElement or type(Ob)=deleteProcessing or
        type(Ob)=deleteWord or type(Ob)=deleteSeparator or
        type(Ob)=deleteChar or type(Ob)=deleteAttribute)
        if (index(Ob)<index(Oa))
          index(O'a)−−;
      else
        if (index(Ob)=index(Oa)){
          if ((type(Oa)=insertElement or type(Oa)=insertProcessing or
            type(Oa)=insertWord or type(Oa)=insertSeparator or
            type(Oa)=insertChar or type(Oa)=insertAttribute)
```

```
        and increment=true)
        index(O'a)++;
      if ((type(Oa)=deleteElement or type(Oa)=deleteProcessing or
        type(Oa)=deleteWord or type(Oa)=deleteSeparator or
        type(Oa)=deleteChar or type(Oa)=deleteAttribute)
        index(O'a)++;
    }else index(O'a)++;
  return O'a;
}
```

The first argument of the *include* function, O_a, represents the operation that has to be transformed and the second argument of the function, O_b, represents the operation against which O_a has to be transformed. The third argument *increment* is a boolean specifying if in the case of a tie the position of the transformed operation should be increased or not. The *include* function returns operation O'_a representing the transformed form of O_a. Note that O'_a can be a composition of two operations. Also note that O_a and O_b are operations of the same level of granularity that target the same node.

Block 1 of the function initialises the result O'_a of the function with the initial operation O_a. Block 1 analyses also the following case. An operation does not change by including a *NOP* and a *NOP* does not change by including any other operation.

The algorithm contains two main parts. The first part of the algorithm analysed in blocks 2, 3, 4 and 5 refers to operations that target closing tags, i.e. *insertClosingTag* and *deleteClosingTag*. The second part of the algorithm analysed in block 6 refers to the other types of operations excluding operations on closing tags.

Blocks 2-5 analyse the case where one of the operations O_a or O_b refers to a closing tag. Usually, for a set of operations, transformation functions have to be written for each pair of operations. In what follows we explain that due to the types of elements targeted by these operations or due to the compression procedure applied before operation transformations, some pairs of operations do not need to be transformed.

As previously mentioned, a *deleteClosingTag* operation can be performed only if the element whose closing tag has to be deleted has no children. Similarly, when an *insertClosingTag* is issued, the element whose closing tag is inserted has to be empty. The effects of deletion of a closing tag and of insertion of elements as direct children of the element whose closing tag has to be deleted cannot be both satisfied. We have chosen to cancel the *deleteClosingTag* operation and to keep the inserted elements, due to the fact that a *deleteClosingTag* operation means to simply rewrite the form

of an empty element. An operation targeting an attribute of the element or the name of the element whose closing tag is to be deleted or inserted is not influenced and does not influence the *deleteClosingTag* or *insertClosingTag*. An *insertClosingTag* does not need to be transformed against a *deleteClosingTag*. If the two operations refer to different elements, they do not need to be transformed as they do not affect each other. The case that the two operations *insertClosingTag* and *deleteClosingTag* refer to the same element cannot occur, as the two sequences of operations that have to be merged have the same initial state and before a merge is performed the sequences of operations are compressed. By applying the compression procedure, an *insertClosingTag* is cancelled by a *deleteClosingTag* and vice versa. If in one sequence of operations, after the compression is performed, the operation *insertClosingTag* occurs, it is not possible that in the other compressed sequence of operations, the operation *deleteClosingTag* referring to the same element occurs. If it occured, due to the fact that the initial context of the two sequences of operations is the same, an operation *insertClosingTag* should have preceded *deleteClosingTag*, in which case the pair of operations *insertClosingTag* and *deleteClosingTag* would have been compressed resulting in the null operation.

Block 2 deals with the case that O_b is a *deleteClosingTag* operation. If O_a is also a *deleteClosingTag* operation targeting the same node, then O_a is cancelled.

As previously discussed, if O_b is the *deleteClosingTag* operation and the concurrent operation O_a inserts a child element of the target node of O_b, the combination of both operations cannot be executed. We took the decision that the insertion of children takes priority and therefore the transformed form of O_a should first cancel the effect of *deleteClosingTag* and then perform the insertion of the children. This is due to the fact that the insertion of child nodes requires that the target element contains both a begin and an end tag.

In the other cases in which operation O_b deletes a closing tag, but operation O_a does not satisfy one of the above conditions, operation O_a preserves its original form.

If none of the cases analysed in block 1 and 2 occurred, the algorithm tests in block 3 the case that O_b is an *insertClosingTag* operation. If O_a inserts the same closing tag as the local operation O_b, O_a is cancelled. Other operations transformed against an operation of insertion of a closing tag keep their initial form.

If none of the conditions of blocks 1, 2 and 3 were fulfilled, processing continues with the test of block 4 if O_a is a *deleteClosingTag* operation. If

O_a deletes a closing tag and O_b inserts an element, processing node, word or a separator between child elements as a direct child of the element whose closing tag is deleted, O_a is cancelled. The function $typeTarget(O)$ returns the type of the position index of operation O. The index indicates a child, attribute or character position. If O_b inserts a character to a position less or equal to the position of the target element of O_a, then O_a has to increment its position. Recall that *deleteClosingTag* operation is simulated by the insertion of the character "/" at the end of the element's starting tag. If operation O_b deletes a character to the left of the position of the target element of O_a, then operation O_a has to decrement its position. If none of the above cases occurs, operation O_a keeps its original form.

Processing of block 5 is reached only if none of the tests performed in blocks 1-4 succeeded. Block 5 tests the case that O_a is an *insertClosingTag* operation. If operation O_a inserts a closing tag and operation O_b inserts a character situated at a position less or equal to the position of O_a, then operation O_a has to increment its position. Recall that *insertClosingTag* operation is simulated by the deletion of the character "/" at the end of the element's name and it is tracked as having a $Normal\_Index$ type. If a remote operation O_b deletes a character situated to the left of the position of the target element of O_a, then operation O_a has to decrement its position. If none of the above cases occurs, O_a keeps its original form.

If the two argument operations O_a and O_b did not satisfy any of the cases analysed in blocks 1-5, processing reaches block 6. Block 6 performs the following tests. If O_a and O_b target different types of units, such as attributes, elements or characters, the operations do not affect each other and therefore the original form of operation O_a is returned. In the case that both operations O_a and O_b delete the same element, processing node, word, separator, attribute or character, operation O_a is cancelled.

If operation O_a targets an element at a position greater or equal to the position of O_b, depending on the type of operation O_b two further subcases can be distinguished. The first subcase corresponds to the delete type of operation O_b: if the position of O_a is greater than that of O_b, the position of the transformed operation has to be decreased and if the positions of O_a and O_b are equal, the transformed operation keeps its original form. The second subcase corresponds to O_b being an insert. If the position of insertion of O_b is situated to the left of O_a, the position of the transformed operation has to be increased. But, if the position of insertion of O_b is equal to the target position of O_a, special cases can be distinguished depending on whether O_a is an insert or delete.

If O_a deletes an element situated at the insertion position of O_b, the

position of the transformed O_a should be increased by 1. If both operations O_a and O_b insert an element at the same position, we formulate a rule governing the order of execution of the two operations, i.e. the order of insertion of the elements targeted by the two operations. Any order of insertion is as good as the other, but the same order should be applied at all sites. If the argument *increment* of the function *include* is true, then the position of the transformed operation has to be incremented and if it is false the position does not need to be incremented. As for the case of the asynchronous communication over text documents (see section 3.4.5), the inclusion transformation function is performed in three cases during the merge process. The first case when inclusion transformation is performed is when pairs of non conflicting operations from the repository and from the local workspace are symmetrically included one against the other in order to compute the new list of operations that have to be executed in the local workspace and the new difference in the repository. In the case of symmetrical inclusion between two inserts targeting the same position, one of the inclusion transformations will be called with the argument *increment=true* and the other one with the argument *increment=false*. In this way we ensure that one of the insert operations increases its original position when it is transformed and the other one does not. Another solution for the arbitration mechanism of increasing the position of one of the concurrent insert operation would have been to compare the content of the two operations, as it has been done for text documents. The second case when inclusion transformation is performed during merging is when a conflicting operation has to be removed from the log in the repository and it has to be transposed to the end of the log. However, the special case of two concurrent operations inserting at the same position in the document is treated inside the transpose function, before the inclusion transformation is performed. The third case when inclusion transformation is performed occurs during the iteration process over the document levels after the merging of the list of operations from the local log and from the remote log, corresponding to the current level of granularity has been performed. It is the step when the operations from the remote log have to include in their context the list of operations representing the new delta in the repository. However, the case that an insert operation has to be transformed against another insert operation of the same granularity that inserts an element at the same position does not occur. This is due to the fact that after the merging corresponding to a node in the document, the operations in the remote log associated with that node that require transformation are of a finer granularity than the local transformed operations and therefore, the

value of the third argument of the function *increment* does not matter.

The exclusion transformation function is presented in what follows.

```
Algorithm exclude(Oa,Ob):O'a {
  O'a := Oa;
  if (type(Ob)=NOP or type(Oa)=NOP) return O'a;
  if (typeTarget(Oa)!=typeTarget(Ob)) return O'a;
  if (index(Ob)≤index(Oa))
    if (type(Ob)=insertProcessing or type(Ob)=insertWord or
        type(Ob)=insertSeparator or type(Ob)=insertChar or
        type(Ob)=insertAttribute or type(Ob)=insertClosingTag)
          if (index(Ob)<index(Oa))
            index(O'a)−−;
    else
        if (type(Ob)!=deleteClosingTag)
          index(O'a)++;
  return O'a;
}
```

The argument O_a represents the operation that has to be transformed and O_b represents the operation whose effects have to be excluded from O_a. The *exclude* function returns the transformed form of O_a.

Excluding *NOP* from an operation or excluding an operation from *NOP* has no effect on the result. The discussion follows the same pattern as for the include transformation function. For any kind of remote operation that can have an impact on the local one, i.e. they have the same operation position index types, the local index has to be incremented or decremented.

In the previous subsection 5.5 we stated that there are some combinations of operation types which we did not consider when writing the inclusion transformation. This is also the case with the exclusion transformation. We do not modify the local operation if the remote operation has a different index type. Also, due to the compression procedure, there are a few combinations which may never appear. The number of discussed cases is reduced even more in the exclusion function because some situations are solved within the transposition function presented in what follows.

```
      Algorithm transpose(Oa,Ob):N {
1:      if (type(Oa)=insertClosingTag and
          (type(Ob)=insertProcessing or
          type(Ob)=insertElement or type(Ob)=insertWord
          (type(Ob)=insertSeparator and typeTarget(Ob)=ChildIndex)))
          return 1;
2:      if ((type(Oa)=deleteProcessing or
          type(Oa)=deleteElement or type(Oa)=deleteWord
```

```
        (type(O_a)=deleteSeparator and typeTarget(O_a)=ChildIndex))
        and type(O_b)=deleteClosingTag
        return -1;
3:  O := exclude(O_a, O_b);
4:  if (index(O_a)=index(O_b) and
        ((type(O_a)=insertElement and type(O_b)=deleteElement) or
        (type(O_a)=insertProcessing and type(O_a)=deleteProcessing) or
        (type(O_a)=insertWord and type(O_b)=deleteWord) or
        (type(O_a)=insertSeparator and type(O_b)=deleteSeparator and
        typeTarget(O_a)=ChildIndex and typeTarget(O_b)=ChildIndex) or
        (type(O_a)=insertSeparator and type(O_b)=deleteSeparator
        and typeTarget(O_a)=AttributeIndex and typeTarget(O_b)=AttributeIndex) or
        (type(O_a)=insertAttribute and type(O_b)=deleteAttribute) or
        (type(O_a)=insertChar and type(O_b)=deleteChar))){
        O_a:=NOP;
        O_b:=NOP;
        return 0;
    }
5:  if (index(O_a)=index(O_b) and
        (type(O_a)=insertElement or type(O_a)=insertProcessing or
        type(O_a)=insertWord or type(O_a)=insertSeparator or
        type(O_a)=insertChar or type(O_a)=insertSeparator) and
        (type(O_b)=insertElement or type(O_b)=insertProcessing or
        type(O_b)=insertWord or type(O_b)=insertSeparator or
        type(O_b)=insertChar or type(O_b)=insertAttribute) and
        (typeTarget(O_a)=typeTarget(O_b))){
        index(O_a)++;
        O_b:=O_a;
    }
    else
6:      if (index(O_a)=index(O) and
            (type(O_a)=insertElement or type(O_a)=insertProcessing or
            type(O_a)=insertWord or type(O_a)=insertSeparator or
            type(O_a)=insertChar or type(O_a)=insertAttribute) and
            (type(O)= insertElement or type(O)=insertProcessing or
            type(O)=insertWord or type(O)=insertSeparator or
            type(O)=insertChar or type(O)=insertAttribute) and
            (typeTarget(O_a)=typeTarget(O))) {
            O_b:=O_a;
            O_a:=O;
        }
7:      else {
            O_b:=include(O_a,O,false);
            O_a:=O;
        }
    return 0;
}
```

The function *transpose* changes the execution order of the operations O_a and O_b and transforms them such that the same effect is obtained as if the operations were executed in their initial order and initial form. The *transpose* function returns an integer depending on the action that has to be taken outside of the *transpose* function. We are going to explain the action that has to be taken outside the transpose function at the moment when the special value is returned.

The first case that has to be analysed is the one shown in block 1 where operation O_a is an *insertClosingTag* and operation O_b is an operation that inserts a child element. The condition of insertion of a child element is that the parent element has the form containing an initial and an end tag. The context of generation of O_a is that the target element is an empty element containing only a begin tag. Therefore, the two operations cannot be transposed. The solution that we take is to do not change the operation order between O_a and O_b. But when a conflict occurs and *insertClosingTag* operation has to be removed, due to the result returned by the transpose function, all the operations that depend on its existence are removed. The case that O_a is an *insertClosingTag* and O_b is a deletion operation of a child element cannot occur as an operation of insertion of that child element has to precede the deletion of that element and follow *insertClosingTag*. However, due to the compress procedure, the pair insert-delete element would have been eliminated.

Symmetrically, as shown in block 2, a *deleteClosingTag* can be executed only after deleting all the content of the element node, it cannot be executed if the element contains any child nodes. The solution that we take is to do not transpose this operations. When an operation that deletes some of the element node's content needs to be transposed against a *deleteClosingTag* operation, these two operations are simply removed from the log. In the log associated with the target node any operation of insertion of a child node preceding *deleteClosingTag* is cancelled by a delete operation of that child node. This is due to the fact that the condition of execution of *deleteClosingTag* is that the target element has no child nodes. The pairs insert-delete child element are cancelled from the log by the compression procedure. Moreover, there are no operations of insertion and deletion of child elements or *insertClosingTag* following *deleteClosingTag* operation in the log. The operation of insertion and deletion of child elements following *deleteClosingTag* should be preceded by *insertClosingTag*, as it is not allowed to insert child nodes to an empty element containing only a begin tag. If this would be the case, the pairs of *deleteClosingTag* and *insertClosingTag* would have been eliminated by the compression procedure.

As operations targeting characters are situated at the beginning of the log by the compression procedure, the removal of operation *deleteClosingTag* cannot influence the operations targeting characters.

The other special cases analysed in the *transpose* function have been treated also in the transpose function for text documents. These cases are analysed in blocks 4, 5 and 6. As we have seen in section 3.4.5, for all these special cases, the pair of operations result of the transposition can be computed. Therefore, for all these special cases as well as for the usual case of transposition, no additional action outside the *transpose* function has to be taken. Block 3 computes the exclusion of O_a from O_b. The first special case to be analysed is the one shown in block 4 when O_a is an insert on a specific position and O_b is a delete from the same position. If we would execute the usual transpose function, we would first exclude the effect of the insertion of O_a from the delete operation O_b. This would yield a *NOP* since otherwise O_b would delete an element that does exist. The problem arises at the step when the effect of the *NOP* operation is included into O_a. Including O_a against a *NOP* yields an unmodified O_a as a result. This is obviously not correct since the effect of applying the modified O_b and O_a in this order after the execution of transpose should be the same as applying the original O_a and O_b. The solution that we adopted for this particular case is that both O_a and O_b are transformed to *NOP*.

The second case described in block 5 occurs when the two insert operations O_a and O_b have the same position for insertion. For instance, consider the situation when O_a=*insertChar(@c3.c1.c2.0,"x")* and O_b=*insertChar(@c3.c1.c2.0,"y")* are executed in sequence. After the execution of the two operations, the string generated is *"yx"*. In order to perform the transposition, operation O_b should exclude from its context operation O_a, the result being the initial form of operation O_b, i.e. O'_b=*insertChar(@c3.c1.c2.0,"y")*. According to the first version of *transpose* function presented in section 3.4.3, O_a would have to include O'_b. The two operations O_a and O'_b insert a unit on the same position and this tie case is broken by the boolean parameter *increment*, that decides if the position of insertion should be incremented or not. If the position of insertion would not be incremented, the result of the transformation would be O'_a =*insertChar(@c3.c1.c2.0,"x")*. This yields as a result the state *"xy"*, different from the initial string *"yx"*. The solution that we adopted was to increment the position of insertion of the first operation when it is transposed.

The third special case analysed in block 6 occurs when O_a and O, the transformed form of O_b which no longer contains the effect of O_a, are both *insert* operations at the same position. For example, consider the situation

when O_a=*insertChar(@c3.c1.c2.0,"a")* and O_b=*insertChar(@c3.c1.c2.1, "b")*. The string generated is *"ab"*. After excluding the effect of O_a from O_b, the transformed form of O_b would be O=*insertChar(@c3.c1.c2.**0**,"b")*. If we would simply include O_a against O, the two operations insert a unit on the same position and this tie case is broken by the boolean parameter *increment*, that decides if the position of insertion should be incremented or not. If the position of insertion would be incremented, then the new form of O_b would be O_b=*insertChar(c3.c1.c2.**1**,"a")*. In this case the final result would not be correct as the effect of applying the modified O_a and then the modified O_b would be the string *"ba"* instead of the initial string, *"ab"*. The solution that we adopted was to simply assign to the new form of O_b the old form of O_a, without including the operation O. In this case the result would be O_b=*insertChar(@c3.c1.c2.**0**,"a")*, which means that, by executing the modified form of O_b and then the one of O_a, we obtain the correct string, *"ab"*.

The case analysed in block 7 is the usual case of transposition where the new form of O_b is obtained by performing an inclusion of O_a against O and the new form of O_a is O.

The *update* function for XML documents is similar to the *update* function for text documents described in section 3.4.5. We are not going to present the *update* function as it does not introduce new concepts or ideas. The main purpose of this chapter was to show that the same principles that have been used for the communication over text documents have been used also for the communication over XML documents.

5.7 An asynchronous XML editor

As in the asynchronous text editor, the XML editor involves starting the repository and then starting the client. However, for XML, the repository allows for storing multiple documents together with their versions.

The client application supports the editing of documents from the repository and the synchronisation of the local version of the documents with the corresponding versions of the document from the repository.

Once the user has selected a document, the editor loads it and builds up an internal document model. In the following we present the different parts of the client interface, shown in figure 5.1.

1. The buttons toolbar (1) allows the user to *open a new document*, *commit the changes* or *update the document*, as well as perform actions of

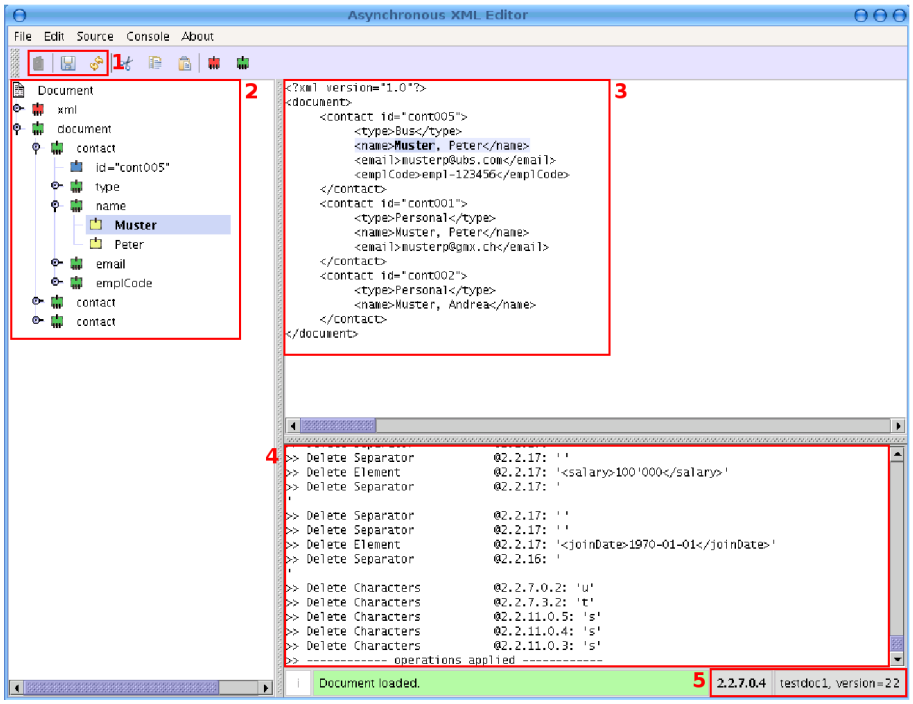


Figure 5.1: XML Editor, Document View. 1 is the buttons toolbar, 2 a tree view, 3 a text pane, 4 the console and 5 current position

cut, *copy* and *paste* parts of the document. The same functionalities are also offered by the application menu.

2. In the left part, the editor presents a *tree view* (2) of the document. The user can click on a tree node in order to highlight the corresponding text fragment in the text pane.

3. The *text pane* (3) contains the text representation of the XML document. If the user moves the caret, the corresponding tree node is automatically selected in the tree view. Views 2 and 3 are linked.

4. Below the text pane, the user is presented with a *log console* (4), where the edit operations are displayed.

5. At the bottom of the editor, the user can see the current position of the caret in the document structure, as well as the name and version of the open document (5).

Conflict resolution

When an update is performed, conflicts between the local copy of the document and the version of the document in the repository might appear. In order to visualise conflicts and to offer the user the option to choose which version to keep, the editor offers a conflict resolution dialog. In order to illustrate the conflict resolution approach let us consider the following example.

Suppose that two users start working on the same version of the document, illustrated below.

```
<?xml version="1.0"? >
<document>
   <movieDB>
      <movie title="21 Grams" year="2003">
         <director>Alejandro Inarritu< /director>
         <actor>Sean Penn< /actor>
         <actor>Naomi Watts< /actor>
      < /movie>
      <movie title="Mar adentro" year="2004">
         <director>Alejandro Amenabar < /director>
         <actor>Javier Bardem< /actor>
         <actor>Belen Rueda< /actor>
      < /movie>
   < /movieDB>
< /document>
```

Suppose that the two users concurrently add an actor to the second *movie* element, as illustrated in Figure 5.2.

Suppose that the first user commits their modifications to the repository and the second user performs an update before committing to the repository. The updated version of the document is illustrated in Figure 5.3. As we can see, the two *actor* elements added by the two users are present in the merged version of the document.

There are cases when a user does not want to perform an automatic merging of the changes, but prefers to set the detection of conflict for different nodes and manually choose between the conflicting versions of the nodes. For instance, suppose that the two users continue their work and the first user adds an *actor* element to the first *movie* element and the second user adds another *actor* to the same *movie* element, as shown in Figure 5.4.

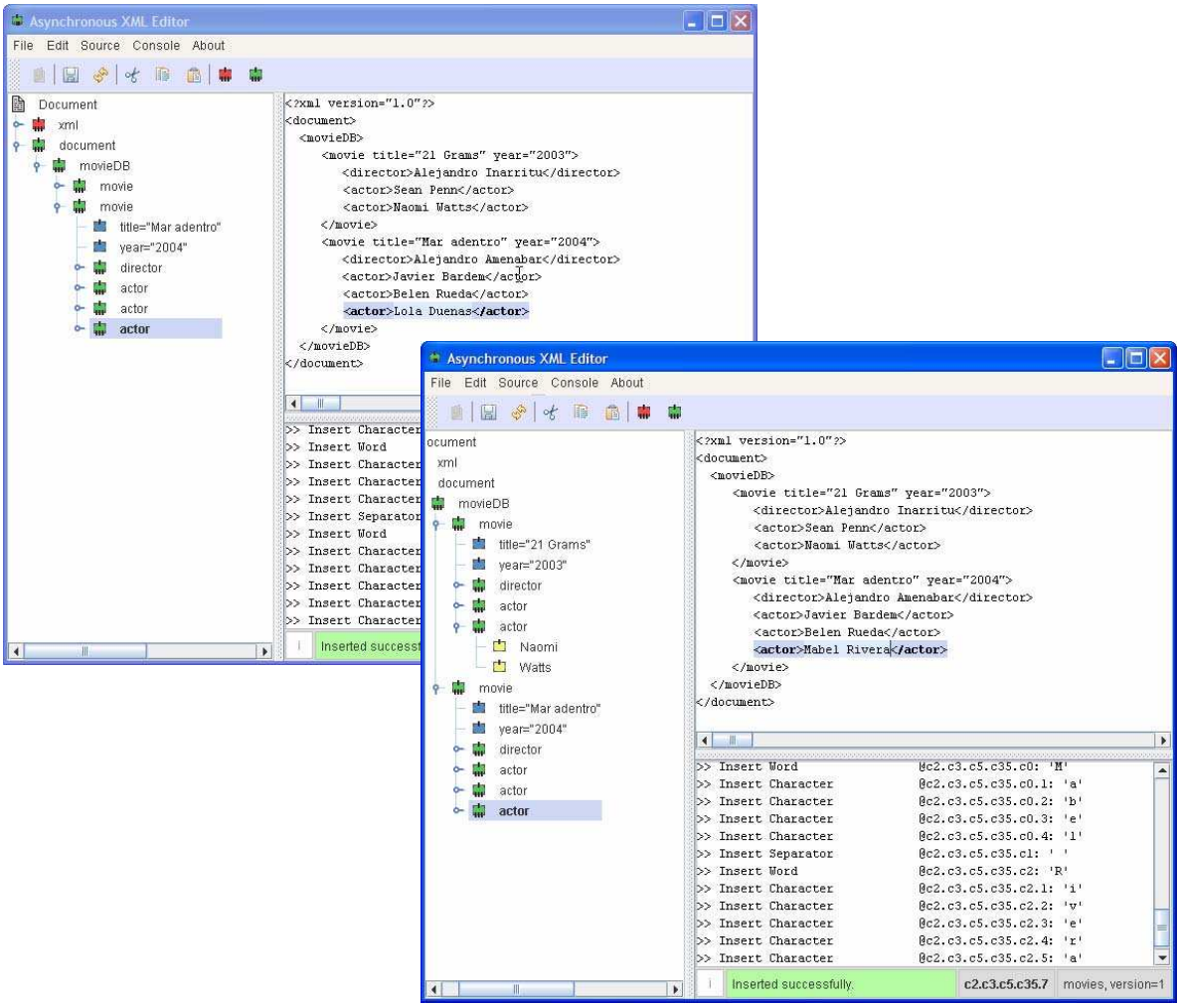


Figure 5.2: XML Editor - First example of concurrent operations performed by two users

Suppose the first user commits their changes to the repository and the second user before updating the changes from the repository sets a semantic conflict for the first *movie* element. The setting of a semantic conflict for a node can be done from the tree view of the interface by selecting the node and setting conflict detection for it. As two concurrent modifications were performed on the first *movie* element, the second user is presented with the two versions of the document as shown in figure 5.5.

The user can then choose to keep the local version of the document or the remote version.

There are cases when the user does not want to be presented with a conflict dialog, but instead keep local changes for some parts of the document. The user is offered the possibility to lock some nodes before performing an update. If a node is locked and conflict occurs on that node, the user is

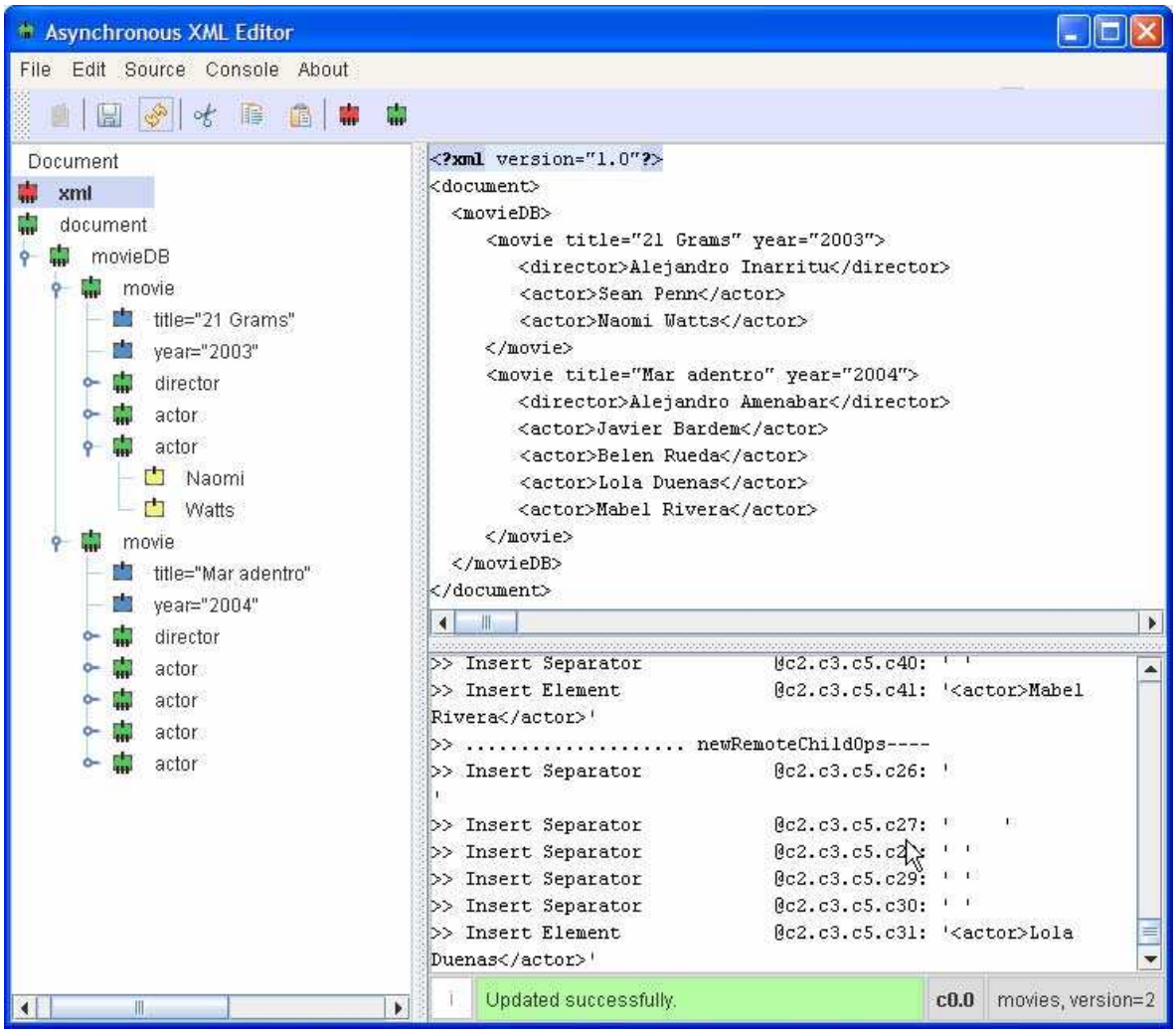


Figure 5.3: XML Editor - Merging result

not asked to manually solve the conflict, but keeps the local version.

5.8 Real-time collaboration for XML

In this section we present some issues related to the adaptation of treeOPT to a real-time collaborative editor for XML documents [10].

5.8.1 Document model

In our second solution to XML editing, we extended the structure of a node as defined in the definition 3.1.1, by allowing internal nodes to have data content. The new definition of the structure of a node is given below:

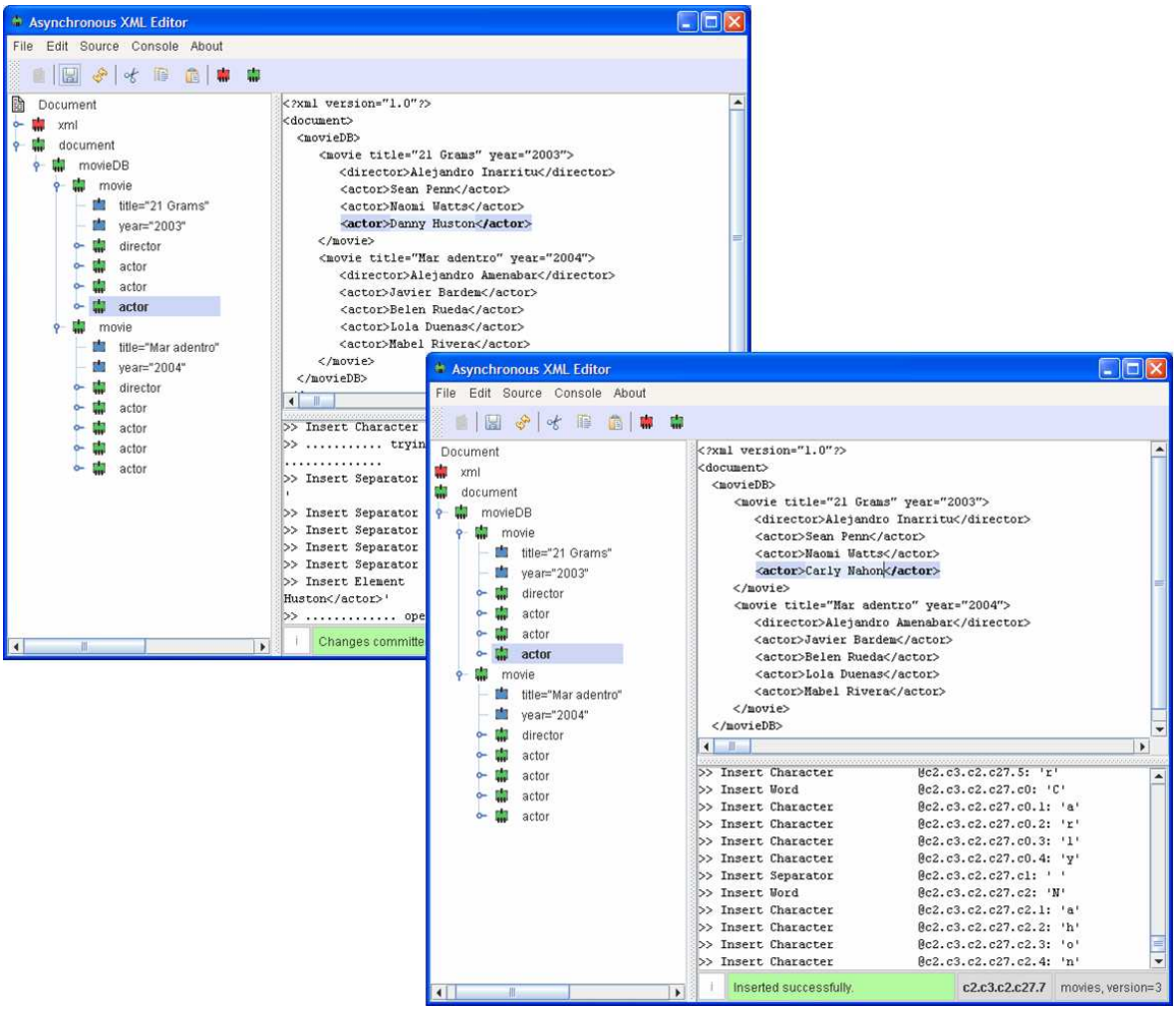


Figure 5.4: XML Editor - Second example of concurrent operations performed by two users

Definition 5.8.1 *A node N of a document is a structure of the form* $N =< \text{parent}, \text{children}, \text{length}, \text{history}, \text{content} >$, *where*

- parent *is the parent node for the current node. Except for the topmost node, parent is a valid reference to a node in the tree.*
- children *is an ordered list* $[child_1, ..., child_n]$ *of child nodes*
- length *is the length of the node, i.e. the sum of the lengths of its children and of the length of its content.*
- history *is an ordered list of operations executed on child nodes*
- content *is the content of the node*

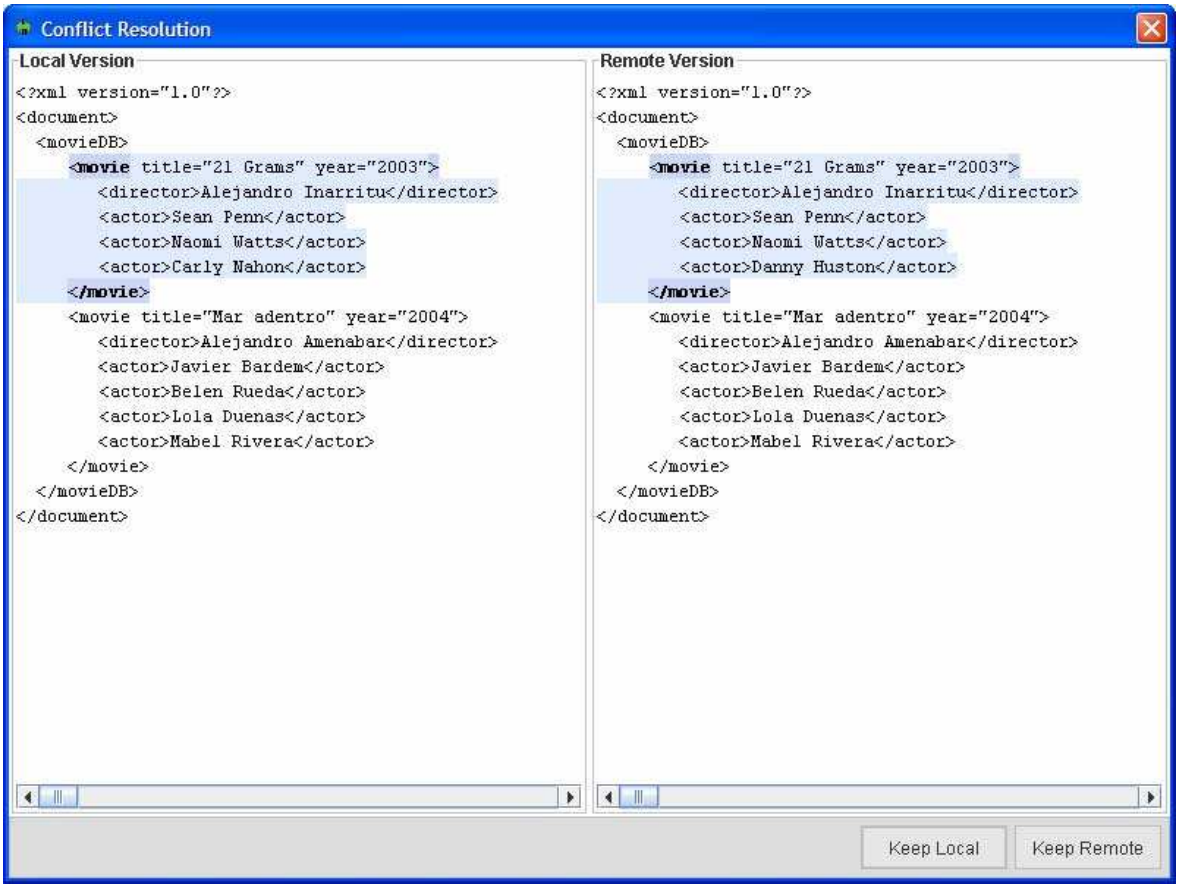


Figure 5.5: XML Editor - Conflict resolution

The definition given above differs from definition 6.1.1 in that the content of a node is defined for every internal node and not only for leaf nodes. Therefore, the length of a node is computed as a sum of not only the lengths of its children, but also of its content.

An issue resulting from previous work on the maintenance of consistency in text documents was where the separators should be encoded. In our previous solutions, separators were seen as children of the node where they appear. For instance, for text documents, the space separators between the words are seen as children of the sentence to which they belong. We adopted a similar solution to the representation of XML documents in asynchronous collaboration, where the separators between the element nodes are represented as children of the parent node. While in text documents the number of spaces separating words is roughly equal to the number of words, the overhead of separator nodes raises dramatically in XML documents.

One of the alternative solutions to the representation of separator nodes as elements was to store the separators as data within the parent nodes of the nodes they are separating. We have adopted this solution in the implementation of a real-time XML editor. However, this solution raises the following issue. In order to generate operations, one must map the linear position where the user is editing to a tree path and therefore, the computation of the start and end position of any node in the tree is required. In the solution where the separators are seen as separate nodes, the computation of the start of any node in the tree requires the addition of the lengths of all preceding siblings and the computation of the end position requires the addition of the lengths of all preceding siblings and the node length.

If separators are kept in the content of the parent node, this way of computing the start and end position of a node is not possible. In order to facilitate this computation, the starting position of each node is stored in the document model. By using this information, a mapping from the linear representation of the document to the tree representation is straightforward. The starting position of a node is already stored in the node and the end position is computed by adding to the starting position the node length. Mapping linear positions to tree paths becomes fast, but updating is expensive. Insertions and deletions of nodes require an update of the starting positions of the nodes. By storing the absolute position of the node within the linear structure, the insertion at the beginning of the document requires the updating of most of the nodes in the tree. We used therefore starting positions relative to the parent node. The trade-off is that search is slower as the starting positions are recursively computed by adding the start positions of the ancestor nodes.

In the XML document given below, the relative positions of the nodes are illustrated in figure 5.6.

<**items**> $\boxed{\backslash n}$
␣ ␣ <**item**> $\boxed{\backslash n}$
␣ ␣ ␣ ␣ <**order**> 1 < /**order**> $\boxed{\backslash n}$
␣ ␣ ␣ ␣ <**name**> alpha < /**name**> $\boxed{\backslash n}$
␣ ␣ < /**item**> $\boxed{\backslash n}$
␣ ␣ <**item**> $\boxed{\backslash n}$
␣ ␣ ␣ ␣ <**order**> 2 < /**order**> $\boxed{\backslash n}$
␣ ␣ ␣ ␣ <**name**> beta < /**name**> $\boxed{\backslash n}$
␣ ␣ < /**item**> $\boxed{\backslash n}$
<**items**>

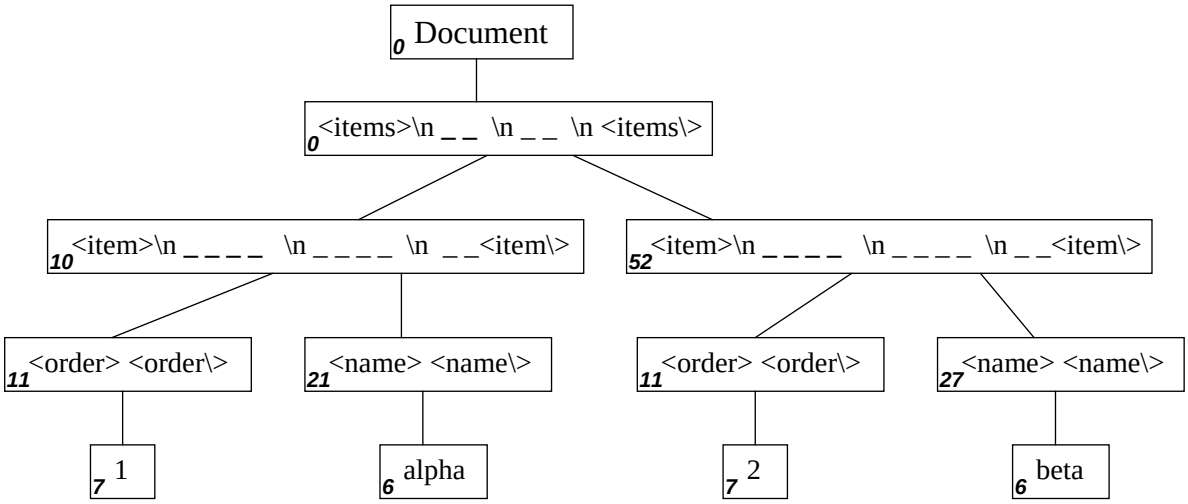


Figure 5.6: XML document - relative positions of the nodes

The types of operations that were defined on the XML structures are *insertion* and *deletion* of nodes as in the case of text documents, and *modify* operations in order to perform atomic modifications of node data.

The transformation functions used in the SOCT2 algorithm have been modified to include modify operations.

5.8.2 Testing

The XML editors that we built rely on a multicast communication module or on a TCP/IP communication protocol. The user can choose one of the two modes of communication.

Testing software to verify its correctness is a very important task. For the real-time editors there is a need to test the validity of the algorithms maintaining consistency in certain specific scenarios. The difficulty lies in simulating the delays of operations such that certain special scenarios are obtained. In other words, there was a need to generate a specific sequence of operations, edit it, if necessary, and rerun the operations at a specific site.

The TCP/IP communication core relies on a connection oriented strategy to broadcast data and it needs a server to connect to. Hermes stands for Hermes Message Server and is the relay server that the TCP/IP core requires in order to broadcast its messages. Once the server is on-line, a couple of clients applications can be started. As the clients connect, the server will generate a column for each site in its main message view. Once the clients are connected, the server automatically records all message from

all the clients as they arrive, as shown in figure 5.7.

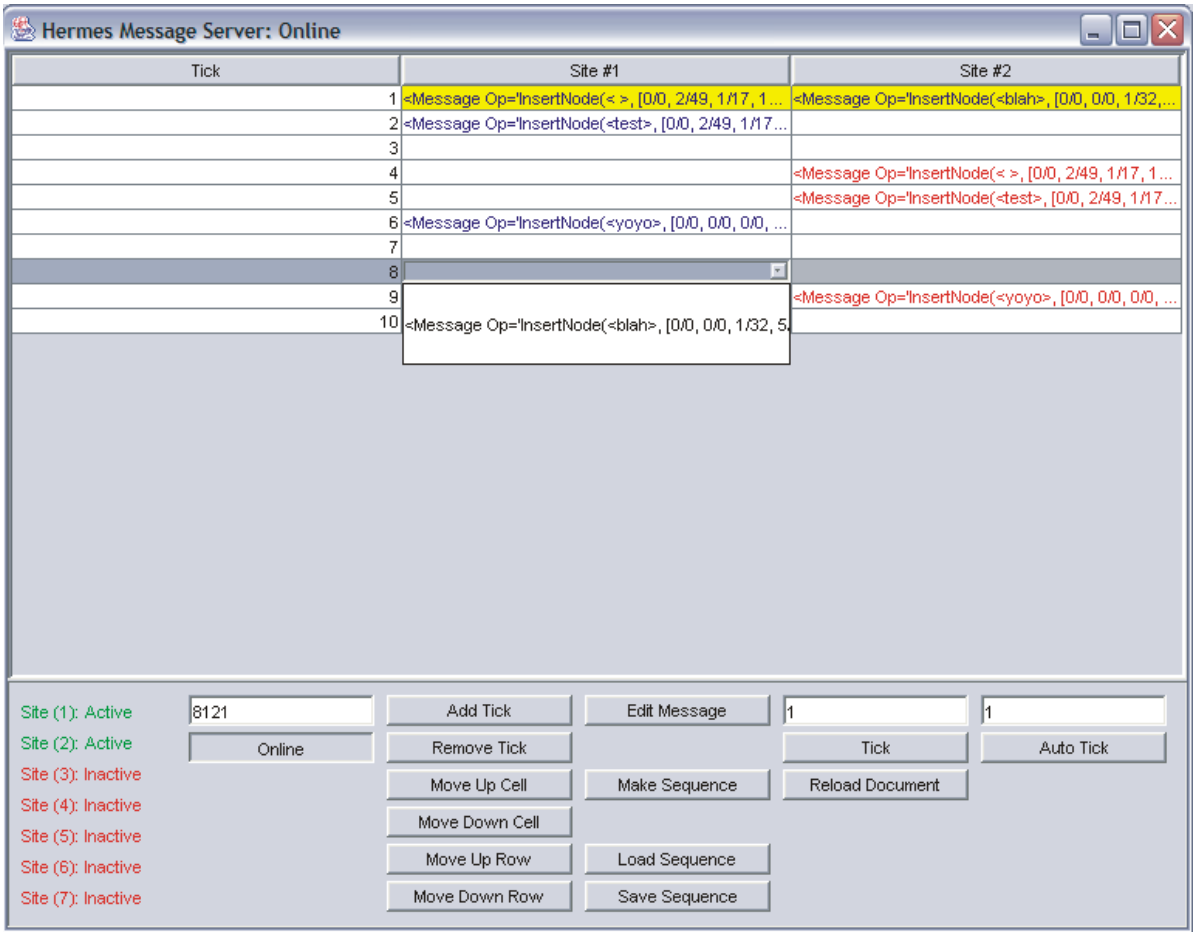


Figure 5.7: Hermes Message Server. Site #1 and Site #2 are shown with user operations

Hermes is also capable of saving the current sequence of operations to a file for later use as well as loading a previously used sequence from file.

Once operations are recorded by the server, they can be reordered to make up a new test case. The server provides means of changing the order of operations, deleting operations, as well as editing operations so that the tester can create any sequence of operations required for their test cases.

To simulate the current sequence, one just needs to connect clients to the server and either execute the operation sequence step by step or all at once. Simulating the sequence step by step will allow the tester to observe the inner workings of the clients by looking at the logs while applying all operations at once will only show the end result.

5.9 Related work

As already presented in chapter 1, the approaches for merging can be classified into state-based and operation-based. Some state-based approaches were described in section 6.7.2. As we have seen, besides the complexity for computing the difference between two versions, state-based approaches have the disadvantage that they do not offer support for tracking user activities. These approaches take into account just the final and initial states of the document and lose information about the process of transformation from one state to the other, such as the order of execution of the operations. There is usually more than one function that can be used to transform an initial state of document into a final one and this function does not reflect user edits. As already pointed out, state-based approaches are not suitable to be used in the real-time communication as the difference between versions has to be computed each time an operation is performed.

However, state-based merging approaches can be used in the asynchronous communication and, therefore, in what follows, we present some algorithms that compute the difference between two XML documents.

The algorithms proposed in [119, 113, 18, 29] compute and represent the difference between two XML documents as a sequence of edit operations. Usually, the algorithms for computing the difference can be classified into one of the following categories [17]:

- Algorithms that compute the minimum edit sequence between two versions of an XML document. The minimum edit sequence iscomputed among all possible edit scripts, including also move operations. The problem is NP-hard and these algorithms would have an exponential cost. There is no tool that uses such an algorithm.

- Algorithms that consider *insert*, *delete* and *update* operations and compute the minimum edit sequence among the scripts that contain these types of operations. The cost of such an algorithm is quadratic or polynomial in the number of nodes of the XML document. The XDiff [119] algorithm belongs to this category. It is designed to handle unordered trees and it has a polynomial complexity.

- Algorithms that do not always find the minimum edit script, but are based on quadratic computations of the minimum edit sequence and run faster than the algorithms that compute the "optimal" difference between XML documents. DeltaXML [29] is an algorithm that belongs to this category.

- Algorithms that match nodes in both documents and afterwards generate the edit script that describes the corresponding changes. These algorithms have a linear complexity. One of the algorithms belonging to this category is XyDiff [18]. It considers beside insert, delete and update operations also move operations and has the complexity of $O(n \cdot log(n))$. However, the delta obtained is not minimal.

The above mentioned state-based algorithms for computing the difference between two versions of XML documents [119, 18, 29] focus on how to minimise the difference between two XML documents. But, as mentioned in [113], these algorithms are not designed for consistency control systems. A consistency control system based on one of these algorithms would generate inappropriately merged XML documents which include unexpected child and sibling relationships that do not appear in any of the two versions to be merged as shown in [113]. The algorithm described in [113] analyses parent-child and sibling relationships of XML elements more precisely than other existing algorithms and generates topology-sensitive differences suitable for the consistency control of XML. This algorithm deals also with *move* operations detecting the movement of one node together with its subtree from one part of the document to another. However, the algorithm proposed in [113] makes the assumption that no element has a text sibling, which significantly restricts the classes of XML documents to be merged. Due to the consideration of move operations, the algorithm proposed in [113] has the complexity of $O(N^2)$, where N is the number of nodes. Moreover, the algorithm is a theoretical approach and a version control system based on this algorithm was not implemented.

The main purpose of the previously described algorithms is to detect changes between two XML documents. Our algorithm is operation-based and the delta between two versions of the document is stored as a list of operations representing the edits performed by the user. Therefore, no complex algorithm is used for computing the difference between two versions of the document. However, the previously described algorithms could be used in our approach for detecting the changes between two document versions. Our merging approach can be equally used for merging stored operations as in operation-based approaches as well as operations resulting from delta computation.

In our approach move operations are implemented as a sequence of delete and insert operations. If a user moves a part of the document while another user concurrently modifies the part of the document that was being moved, due to the representation of move operations and the deletion of

the part of the document that was moved, the modifications performed on the part of the document that was being moved are discarded. But, the move operation could be detected in the editing process and considered as a separate operation in the merging process. The inclusion and exclusion transformation functions would need to be extended to deal with the *move* operation. We are planning to implement in the future the *move* operation.

As previously mentioned, state-based merging approaches are not adequate for synchronous collaboration. Concerning real-time communication, we can compare our work with two other operation-based approaches described in [21] and [69]. In [21] an approach for real-time collaboration over SGML documents has been proposed. The operations that can be performed on the tree are the insertion of a subtree as a child of a specified node, the deletion of a subtree and the modifications of the content of a node. The main difference between our approach and the one proposed in [21] is that in our work the history buffer containing the operations is distributed throughout the tree rather than being linear as in [21] and therefore only a part of the history has to be scanned and a smaller number of transformations have to be performed. Moreover, in our approach an existing linear algorithm can be recursively applied. Therefore, the transformation functions are kept simple, as in the case of linear structured data, and they do not have to be adapted to the tree structure as in [21]. Due to the fact that in [21] nodes in the tree are not deleted, but rather excised from the tree, concurrent modifications of the part that was deleted are executed on the excised part of the tree. In our approach when a node is deleted, concurrent modifications performed on the deleted part of the tree are discarded. However, in [21] it is not explained if and how the excised parts of the tree are reintegrated into the initial tree. Specific issues arising in the editing of XML documents such as auto-completion of elements have not been discussed and no editor based on the ideas proposed in [21] has been reported on.

In [69] another operational transformation approach has been proposed for the real-time and asynchronous editing over hierarchical documents such as XML. The environment that was built based on the approach proposed in [69] is called SAMS (Synchronous, Asynchronous and Multisynchronous System) and it allows the user to perform operations of creation and deletion of elements and attributes and of modification of attributes by means of the graphical interface. As opposed to our approach that provides a text interface for the editing of XML documents, the SAMS environment offers a graphical interface for the editing of the documents. By using the graphical interface, the user is not allowed to customise the formatting for the

elements, such as the use of separators between the elements, as an implicit formatting of the nodes has to be used. If a node has to be modified, the node has to be deleted and a new node with the modified value has to be inserted. Text nodes do not offer a lower granularity such as words or characters, and each time a text node is modified, it has to be deleted and a new text node with the modified value of its content has to be inserted. In our approach we provided a text interface for the editing of XML documents and we added some logic to the editor to ensure well-formed documents, such as the auto-completion of the elements or the consistency between the begin and close tags of an element.

Operation-based approaches can be used for both real-time and asynchronous communication. As we wanted to develop an approach that can be used for any of the two communication modes, our approaches are based on operations. Another reason for choosing an operation-based approach for merging is the fact that we wanted to offer support for conflict management. As stated in [66], merging based on operations also offers better support for conflict resolution by having the possibility of tracking user operations.

6

Consistency Maintenance for Graphical Documents

In this chapter we present our approach [42, 39, 47] for maintaining the consistency in the collaboration using object-graphical documents. We present first in section 6.1 the document model used in the representation of a graphical scene of objects and then in section 6.2 the operations that support the editing of a scene of objects. We then explain in section 6.3 why operational transformation mechanism could not be applied for the graphical editing. In section 6.4 we present the relations that we defined between operations. We then present in section 6.5 our operation serialisation mechanism. Our real-time graphical editor application based on the operation serialisation mechanism is presented in section 6.6. In section 6.7 we present two approaches for merging XML documents in the asynchronous communication based on a shared repository: an operation-based approach relying on the operation serialisation and a state-based merging approach. We also discuss the advantages and disadvantages of using the two approaches. In section 6.8 we give a general comparison between the operational transformation mechanism and the operational serialisation approach. We end the chapter by section 6.9 where we compare our operation serialisation mechanism with other approaches for maintaining consistency over graphical documents.

6.1 Document model

A scene of objects can be modeled by a hierarchical structure: groups are represented as internal nodes, while simple objects are represented as leaves. A group can contain other groups or simple objects.

Definition 6.1.1 *A node N of a document is a structure of the form* $N =< \text{parent}, \text{children} >$, *where*

- parent *is the parent node for the current node. Except for the topmost node, parent is a valid reference to a node in the tree.*
- children *is an unordered list* $[child_1, ..., child_n]$ *of child nodes*

The *children* of an internal node, i.e. a group of objects, consist of the objects contained in the group. As opposed to text and XML documents where a node is identified by its position in the parent node, the order between the children in a group of graphical objects does not matter. A node is identified by its identifier and not by its position in the parent structure. A leaf node does not have any children, and it can be any simple object.

The simple objects allowed by our system are:

- rectangles
- circles
- ellipses
- lines
- text boxes
- polylines (open/closed)
- freehand polylines
- bitmaps

Freehand polylines are defined by a set of points connected by lines. The large number of points composing the polyline gives the impression of a freehand shape.

Consider the scene of objects illustrated in Figure 6.1. Each object or group of objects is assigned an identifier. The scene consists of an object

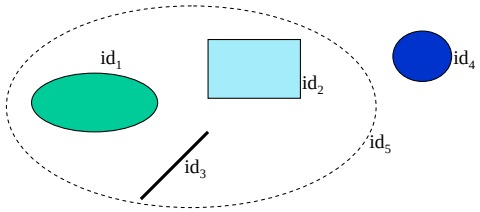


Figure 6.1: A graphical scene of objects

identified by id_4 and a group of objects identified by id_5 containing id_1, id_2 and id_3.

A hierarchical representation of the scene of objects illustrated in Figure 6.1 is shown in Figure 6.2.

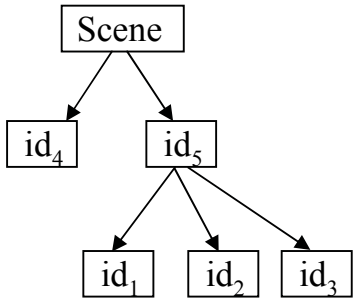


Figure 6.2: A tree representation of a graphical document

We extended the document model to contain a set of layers. Each layer has a root object that references the scene of objects containing groups and simple objects. The layers may be set to be visible or not, which determines whether or not the objects that belong to them appear in the displayed scene of objects. The structure of the document is illustrated in Figure 6.3.

6.2 Operation representation

The following set of operations have been defined for the scene of objects:

- *create(Obj)* to create an object *Obj*.
- *delete(Obj)* to delete an object or group of objects *Obj*
- *group([*$Obj_1, Obj_2, \ldots, Obj_n$*],G)* to group the objects $Obj_1, Obj_2, \ldots Obj_n$ into the group G. $Obj_1, Obj_2, \ldots Obj_n$ can be objects or groups of objects.

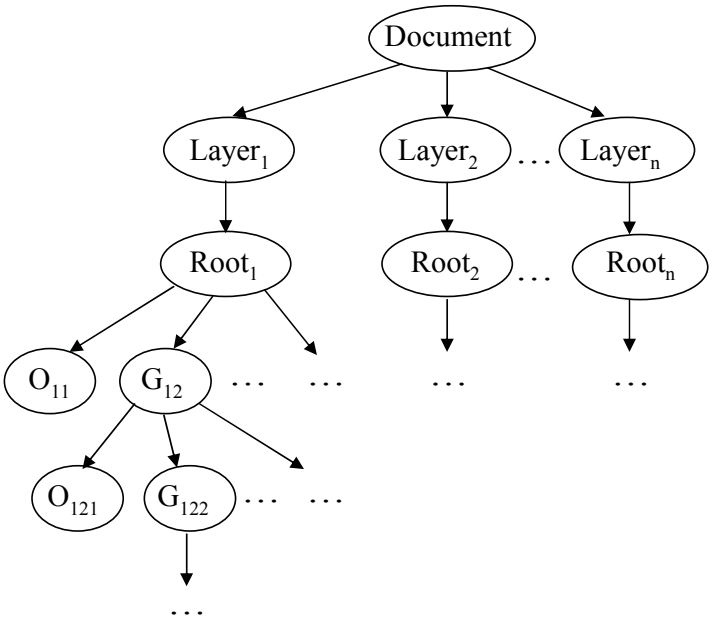


Figure 6.3: The structure of a Graphical Document

- *ungroup(G)* to ungroup the group G
- *move(Obj,d_x,d_y)* to move object *Obj* to a relative position given by the distances d_x and d_y on the x and y axes, respectively
- *scale(Obj,r_x,r_y)* to scale the object or group *Obj* with the ratio r_x and r_y on the x and y axes, respectively, around the object centre
- *rotate(Obj,angle)* to rotate the object or group *Obj* in counterclockwise direction by angle *angle* around the object centre
- *chgColour(Obj,colour)* to change the fill colour of the object or group *Obj* to colour *colour*
- *chgLineColour(Obj,colour)* to change the line colour of the object or group *Obj* to colour *colour*
- *chgStroke(Obj,width)* to change the width of the line of the object or group *Obj* to the value *width*
- *chgText(Obj,text)* to change the text contained in the textbox *Obj* to *text*
- *chgTextSize(Obj,size)* to change the font size of the text contained in the textbox to *size*

- *sendToBack(Obj)* and *sendToFront(Obj)* operations that move the object or group *Obj* to the back or to the front of the scene of objects. These operations use the auxiliary function *setZPosition(Obj,zPos)* to set the depth of the object or group *Obj* to the value *zPos*.

- *movePoint(Obj,index,d_x,d_y)* to move the point given by the index *index* belonging to the polyline *Obj* to a relative position given by the distances d_x and d_y on the x and y axes respectively.

- *createLayer(L,name)* to create a layer identified by L with the name *name*.

- *removeLayer(L)* to remove the layer identified by L

- *moveToLayer(Obj,L)* to move the object or group identified by *Obj* to the layer L.

- *createAnnotation(A,Obj)* creates an annotation A that is attached to the object *Obj*

Each operation has an associated state vector and an identifier of the site which generated the operation.

6.3 Unsuitability of OT for graphical editing

Our first attempt to maintain the consistency of the copies of the shared graphical document was to use the operational transformation mechanism.

Text and XML documents conform to an ordered structure of elements. For instance, a text document is composed of an ordered sequence of paragraphs, each paragraph being composed of an ordered sequence of sentences, each sentence of an ordered sequence of words and each word of an ordered sequence of characters. XML elements conform also to a linear sequence in the list of child nodes of the parent element. Both in text and XML documents each element unit can be uniquely identified by its position in the sequence of child elements of its parent. The operations that were considered were insertions and deletions of unit elements. These operations may shift the positions of an element in the sequence of child elements of its parent element and the main issue was to maintain the order between elements in the face of concurrent operations. Graphical documents have a larger set of objects and operations than text and XML documents and there is no order between the objects. Objects are not

identified by their position, but by a unique identifier and there is no need to adapt the identifiers due to concurrent operations.

We defined the group intention of concurrent operations for pairs of concurrent operations, by means of constraints. Preserving the group intention that we defined was not possible by using operational transformation, as we show in what follows.

We offer the user the possibility to define a set of conflicts and to resolve the conflicts according to different policies. We want to allow the definition of resolution policies that maintain as many user intentions as possible in the face of concurrent operations.

Consider the scene of objects illustrated on the left hand side of figure 6.4. Suppose two users concurrently edit this scene.

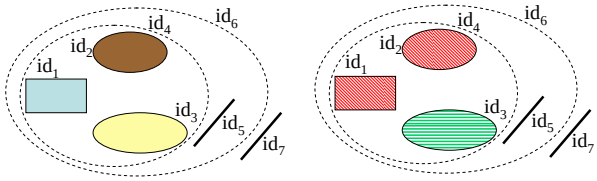


Figure 6.4: Graphical scene of objects; left - the original scene of objects; right - the combined effect of the concurrent operations O_1=*chgColour(*id_4*,red)* and O_2=*chgColour(*id_3*,green)*

The first user changes the colour of the group of objects with the identifier id_4 to *red*, performing the operation O_1=*chgColour(*id_4*,red)* while the second user concurrently changes the colour of the object having the identifier id_3 to *green*, performing the operation O_2=*chgColour(*id_3*,green)*. The two operations conflict because they both target id_3, one setting its colour to *red* and the other changing it to *green*.

One alternative to deal with such situation is to consider that only one of the two operations can be performed. Another alternative would be to consider a partial combined effect of the two intentions, i.e. to change the colour of the object id_3 to *green* and the colours of the objects id_1 and id_2 to *red*, as shown in the right part of figure 6.4. In this case we consider that the effect of an operation performed on an object overwrites the effect of an operation performed on the group to which the object belongs. The partial combined effect is obtained by the serialisation of the two operations: first the operation referring to the group followed by the operation performed on the individual object. The last alternative allows the users to act freely and to combine the effect of the conflicting operations rather than maintaining consistency by forbidding some user actions. By capturing the group intention as described above, some individual intentions

might not be achieved. However, we know that the scene of objects before the generation of conflicting operations can always be reestablished by the user by means of undo. We want to offer the user the option to choose between the two alternatives and therefore offer support for both.

Another example illustrating group intention is given in what follows. Consider the scene of objects illustrated on the left hand side of figure 6.5. Consider that one user wants to ungroup the group with identifier id_5 by issuing the operation O_1=*ungroup(*id_5*)*, while the second user concurrently groups the group id_5 with the object id_4 by issuing the operation O_2=*group(*id_4*,*id_5*)*. The two operations could be considered conflicting and only one of them executed, or a combined effect of the two operations can be achieved by grouping the individual objects of id_5, i.e. id_1, id_2 and id_3, with the object id_4. The combined effect can be achieved by executing first the group operation O_2 followed by the ungroup operation O_1. Again, we want to offer users the possibility of customising the policies for conflict resolution and to support both solutions.

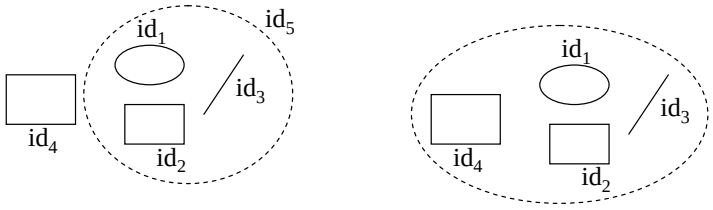


Figure 6.5: The combined effect of two concurrent operations O_1=*group(*[id_4,id_5]*,*id_6*)* and O_2=*ungroup(*id_5*)*; left hand side - before editing; right hand side - after editing

Due to the grouping/ungrouping operations and our aim of maintaining a partial combined effect of conflicting operations, the operational transformation approach is not suitable. The main reason is that transforming an operation targeting a group against a conflicting operation targeting an object belonging to that group would result in a set of operations targeting the individual objects from that group. This would lead to a sort of problems that we are going to highlight in what follows.

Consider the scene of objects illustrated in the left hand side of figure 6.4. Suppose that three users concurrently perform the following operations. At $Site_1$, the first user performs O_1=*chgColour(*id_4*,red)*, at $Site_2$, the second user issues O_2=*chgColour(*id_1*,green)* and at $Site_3$ the third user issues O_3=*chgColour(*id_6*,blue)*, as shown in figure 6.6.

Let us analyse what happens at $Site_2$ if after the execution of O_2 the operations O_1 and O_3 arrive and the operational transformation mech-

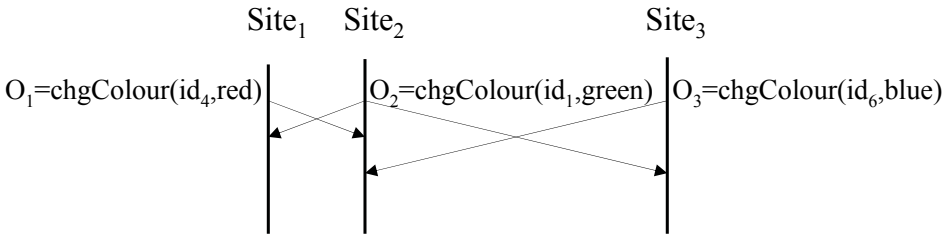


Figure 6.6: A sequence of operations applied to the scene of objects in Figure 6.4

anism is applied. When O_1 arrives at $Site_2$, the transformation of O_1 against O_2 would result in the list of operations [O'_{11}=*chgColour(id*$_2$*,red),* O'_{12}=*chgColour(id*$_3$*,red)*] in order to obtain the combined effect of O_1 and O_2. When O_3 arrives at $Site_2$, O_3 is first transformed against O_2 resulting in the list of operations [O'_{31}=*chgColour(id*$_2$*, blue),* O'_{32}=*chgColour(id*$_3$*,blue),* O'_{33}=*chgColour(id*$_5$*,blue)*]. Further, the list of transformed operations [O'_{31}, O'_{32}, O'_{33}] needs to be transformed against the list of operations [O'_{11}, O'_{12}]. The pairs of operations O'_{31} and O'_{11} as well as O'_{32} and O'_{12} conflict and only the effect of one of them can be satisfied. Using only the information provided by the transformed operations without taking into account the target of the original operation they were generated from, it is not possible to specify rules for obtaining a combined effect of the conflicting operations. In this example, it is not possible to specify the fact that a node in the tree should have the colour specified by the conflicting operation targeting its closest parent node (including the node itself). In order to specify the rules for obtaining the combined effect in the transformation functions, the transformed operation should include additional information. Each operation needs to specify for its target object/group the parent group (including the node itself) from which the target object/group has inherited the property as a result of the transformation. For instance, O'_{31} should have the form *chgColour(id*$_2$*,blue,id*$_6$*)* meaning that as a result of the transformation of O_3 against O_2, object id_2 should have the colour *blue*, inherited from the colour of the group id_6. Therefore, the result of the transformation of $[O'_{31}, O'_{32}, O'_{33}]$ against $[O'_{11}, O'_{12}]$ would be $[O''_{31}, O''_{32}, O''_{33}]$, where O''_{31}=*chgColour(id*$_2$*,red,id*$_4$*)*, O''_{32}=*chgColour(id*$_3$*,red,id*$_4$*)*, O''_{33}=*chgColour(id*$_5$*,blue,id*$_6$*)*.

There are cases when the sequence of operations that are the result of transformation should be considered as a group of operations when further transformations have to be performed. In the operational transformation approach, not only the inclusion transformation needs to be performed, but also the exclusion transformation. Inclusion transformation includes

the effect of an operation into the context of another operation, while exclusion transformation achieves the inverse operation, i.e. excluding an operation from the context of the other one. Consider the set of operations O_1, O_2, O_3 and O_4 defined for the scene of objects in the left hand side of figure 6.4, where O_1=*group([id$_6$,id$_7$],id$_8$)*, O_2=*ungroup(id$_6$)* and O_3 =*chgColour(id$_6$,red)* and O_4 is an arbitrary operation. The scenario is illustrated in Figure 6.4.

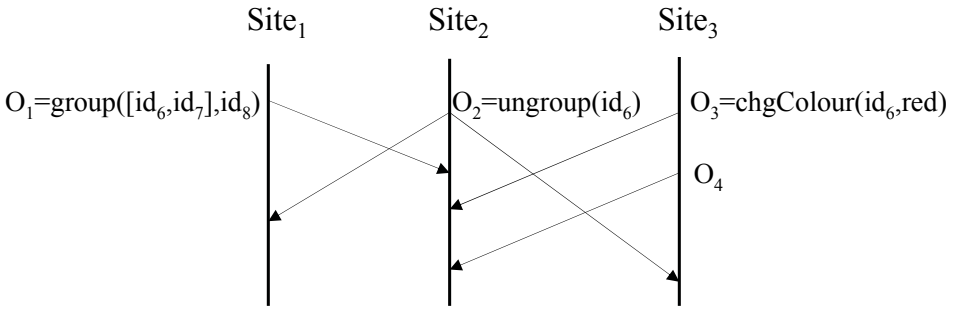


Figure 6.7: Another sequence of operations for the scene of objects in Figure 6.4

Let us analyse what happens at *Site*$_2$. When O_1 arrives at the site it needs to be transformed against O_2. The transformed operation would be O'_1=*group([id$_4$,id$_5$,id$_7$],id$_8$)*. O_3 transformed against $[O_2, O_1]$ produces $[O'_{31}, O'_{32}]$, where O'_{31}=*chgColour(id$_4$,red,id$_6$)* and O'_{32}=*chgColour(id$_5$,red, id$_6$)*. According to any operational transformation algorithm, such as SOCT2, the history buffer $[O_2, O'_1, O'_{31}, O'_{32}]$ needs to be reordered so that the operations preceding O_4, i.e. O'_{31} and O'_{32}, precede the operations concurrent with O_4, i.e. O_2 and O'_1. In order to reorder the operations, O'_{31} and O'_{32} need to exclude the effect of O_2 and O'_1. If the effect of $[O_2, O'_1]$ is excluded in turn from O'_{31} and O'_{32}, it is very difficult to infer that the results of the exclusion for operations O'_{31} and O'_{32} represent the original operation O_3. Moreover, an issue that needs to be discussed refers to the state vector associated with the operations O'_{31} and O'_{32}. If both O'_{31} and O'_{32} had been associated with the state vector of O_3, problems can arise when applying the operational transformation algorithms. As we have seen, considering separately each operation from the sequence of operations that are the result of the transformation would not be a good solution. If the sequence of operations that are the result of transformation is treated as a group, we need to define the operational transformation functions on lists of operations and not on simple operations, which will greatly increase the complexity of the application of the transformation approach.

Due to many issues of applying the operational transformation mecha-

nism to our work, we instead adopted a serialisation mechanism which we are going to describe in section 6.5. In what follows we present the types of conflicts that we identified in our approach.

6.4 Relations between operations

In this section we present the relations between operations, the types of conflicts between operations and the types of relations between operations that impose ordering constraints.

We classified the conflicts between the operations into real conflicts and resolvable conflicts.

We consider that two concurrent operations are conflicting if they modify the same property of a common target object to different values, or one operation targets an object that is destroyed by deletion or ungrouping by the other operation.

Real conflicting operations are those conflicting operations for which a combined effect of their intentions is not desired or cannot be established. The class of real conflicting operations includes those operations for which no serialisation order can preserve the intentions of the operations: executing one operation will make the execution of the other operation impossible or will completely mask the execution of the other one. An example of this kind of real conflicting operations is when two concurrent operations *chgColour(*id_1*,red)* and *chgColour(*id_1*,blue)* both target the same object and change the colour of that object to different values. The real conflicting operation that wins the conflict is chosen according to a priority scheme, in our case according to priorities assigned to users, with the operation generated by the user with the highest priority being the one that wins.

Resolvable conflicting operations are those conflicting operations for which a partial combined effect of their intentions can be obtained by serialisation. Consequently, ordering relations can be defined between the two concurrent operations. Any two resolvable conflicting operations can be defined as being in the right order, or in the reverse order. For instance, for the pair of concurrent operations - a grouping operation of a group G with another object Obj and an ungrouping operation of group G, a combined effect can be obtained by executing the group operation followed by the ungroup operation. If the ungroup operation is executed first, the group operation would target the group G that does not belong to the structure of the document. Note that conflicting operations that can be classified as

resolvable conflicting may be defined as being real conflicting operations by certain applications.

The operation serialisation mechanism involves the reordering of the operations from the history buffer. Between the operations from the history buffer there exist *precede* and *depends on* relationships that do not allow two operations to be executed in reverse order during the serialisation process.

The *precedes* relation between two operations is the causal precedence relation described in section 2.3.9.

An operation O_2 from the history buffer is said to *depend on* O_1 (O_2 *depends on* O_1) if O_1 creates an object or group that belongs to the target list of O_2. An operation cannot be executed before the operation on which it depends. Moreover, an operation has to be cancelled if the operation it depends on is cancelled. If two operations are in a *depends on* relation, they are also in a *precedes* relation.

6.5 Operation serialisation

In this section we present the operation serialisation mechanism that we adopted for maintaining consistency. We first give an intuitive explanation of the issues that occur in the reordering of operations in the presence of real and resolvable conflicts. We then present the algorithms for the integration of an operation into the history buffer containing the previously executed operations at that site. We describe how conflicts are defined in our system, and provide some information about our application.

6.5.1 Intuitive explanation of the approach

Operation serialisation is the mechanism by which operations in the history buffer HB are re-executed in an order such that the partial combined effect of the user intentions is achieved. The serialisation order takes into account ordering constraints that hold between operations. The conflicts as well as the *precedes* relations have to be considered. In the case of two real conflicting operations, depending on the policy of conflict resolution, at most one of them can be executed. In the case of resolvable conflicting operations, operations are executed in the order defined by the ordering relation. The serialisation order has to conform to the *precedes* relation that holds between the operations.

The main idea of the serialisation mechanism that we used for maintaining the consistency is described in what follows. Given the current history buffer $HB = [O_1, O_2, \ldots, O_n]$, the remote operation O_{new} has to be integrated into HB such that, by re-executing the operations in the HB in a certain order, the partial combined intention of the users is preserved. We reduced the task of finding a serialisation order between $O_1, O_2, \ldots, O_n$ and O_{new} to a graph problem. The operations are represented as nodes of a directed graph. Between two operations O_x and O_y there is a directed arc from O_x to O_y if O_x has to be executed before O_y. The resolvable conflicting operations are then represented as arcs. The real conflicting operations are cancelled according to the resolution policy. If an operation is cancelled, its dependent operations also have to be cancelled. In their turn, the dependent cancelled operations cancel their dependent operations. Due to the fact that relations of real conflict between operations do not impose any ordering between the operations, as opposed to the precedes and resolvable conflict relations, two directed graphs are constructed: the real-conflict and serialisation graphs. The real-conflict graph determines which operations have to be cancelled due to real conflicts between operations. If there is a real conflict between operations O_1 and O_2 and operation O_2 has a lower priority than O_1, then the real conflict graph contains an edge directed from O_1 to O_2. The serialisation graph determines the order of operation execution.

Additional care has to be taken concerning conflicting operations in order to maintain consistency. Let us consider the scene of objects illustrated in Figure 6.8.

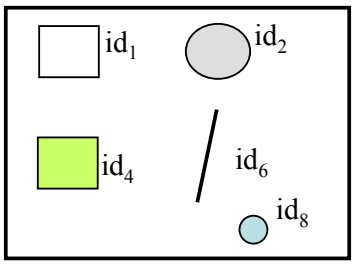


Figure 6.8: Scene of objects for the scenario with real conflicting operations

Suppose that three users concurrently edit the scene of objects illustrated in Figure 6.8. Suppose that the first user groups the objects identified by id_4 and id_6 into a group identified by id_7. Further, the user groups the newly created group id_7 with object id_8, the result being group id_9. Concurrently, the second user groups the objects identified by id_1 and id_4 into group id_5 and changes the colour of this group to *red*. Concurrently

with the operations executed by the first two users, a third user groups the objects identified by id_1 and id_2 into a group identified by id_3 and changes the colour of the group id_3 to *blue*. Suppose that the priorities of the sites $Site_1$, $Site_2$ and $Site_3$ are 1, 2 and 3 respectively and, in the case of conflict, the concurrent operation generated from the site with the highest priority wins the conflict. The highest priority corresponds to the lowest integer value assigned. For instance, priority 1 is higher than priority 2. Operations O_1 and O_3 as well as O_3 and O_5 are real conflicting operations as they target common objects. The editing scenario is illustrated in Figure 6.9.

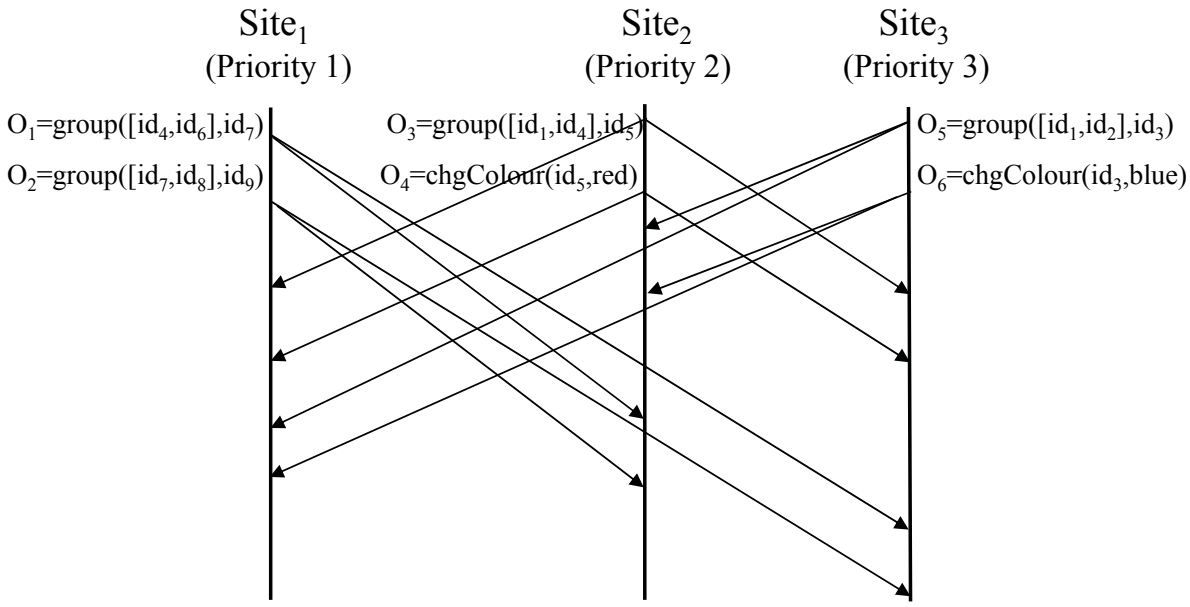


Figure 6.9: Scenario illustrating inconsistencies obtained due to the cancelling order of real conflicting operations

Let us analyse the steps for the construction of the conflict and serialisation graphs at $Site_1$. The edges of both the real conflict and serialisation graphs for $Site_1$ and $Site_3$, respectively, are drawn as single graphs in Figures 6.10 and 6.12. The edges of the serialisation graph are drawn using continuous lines, while the edges of the real conflict graph are drawn using dashed lines. After O_1 and O_2 are generated, the corresponding graph shown in Figure 6.10a contains an edge from O_1 to O_2 as O_2 depends on O_1. When operation O_3 arrives at the site, a real conflict between O_3 and O_1 is detected. As O_1 has a higher priority than O_3, O_3 is cancelled, shown in Figure 6.10b. When operation O_4 arrives at the site, it is cancelled, as O_3, the operation it depends on, was cancelled. The resulting graph is shown in Figure 6.10c. When O_5 arrives at the site no conflict is detected.

When O_6 arrives at the site, as shown in Figure 6.10d, an edge from O_5 to O_6 is added to the graph due to the fact that O_6 depends on O_5.

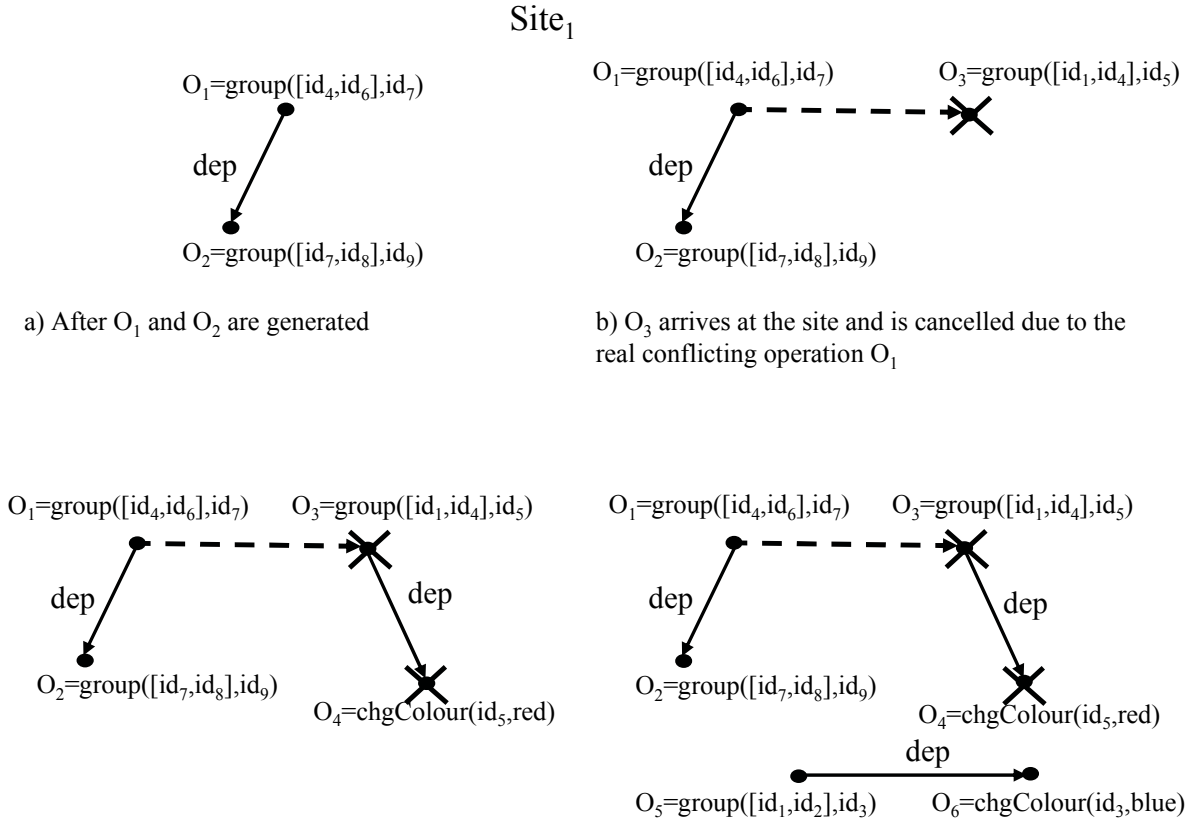


Figure 6.10: Steps in the construction of the graph at $Site_1$. The continuous lines represent edges of the serialisation graph, while the dashed lines represent edges of the real conflict graph.

The final scene of objects at $Site_1$ is shown in figure 6.11.

At $Site_3$, after O_5 and O_6 are generated, an edge between O_5 and O_6 is added to the graph, as shown in Figure 6.12a. When O_3 arrives at the site, the conflicting operation O_5 is cancelled as O_3 has a higher priority than O_5. As O_5 is cancelled, its dependent operation O_6 is also cancelled. After O_4 arrives at the site, the dependent edge between O_3 and O_4 is added to the graph, as shown in Figure 6.12b. If cancelled operations are not reconsidered, when operation O_1 arrives at the site, the real conflict between O_1 and O_3 is detected and operation O_3 is cancelled as O_1 has a higher priority than O_3. Due to the fact that O_3 is cancelled, O_4 is cancelled too, as O_4 depends on O_3. The graph obtained at this step is shown in Figure 6.12c. When O_2 arrives at the site, the dependent edge between O_1 and O_2 is added to the graph, as illustrated in Figure 6.12d.

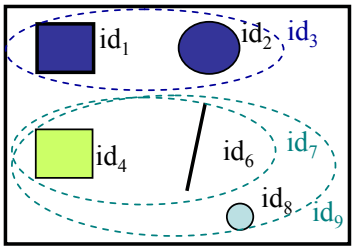
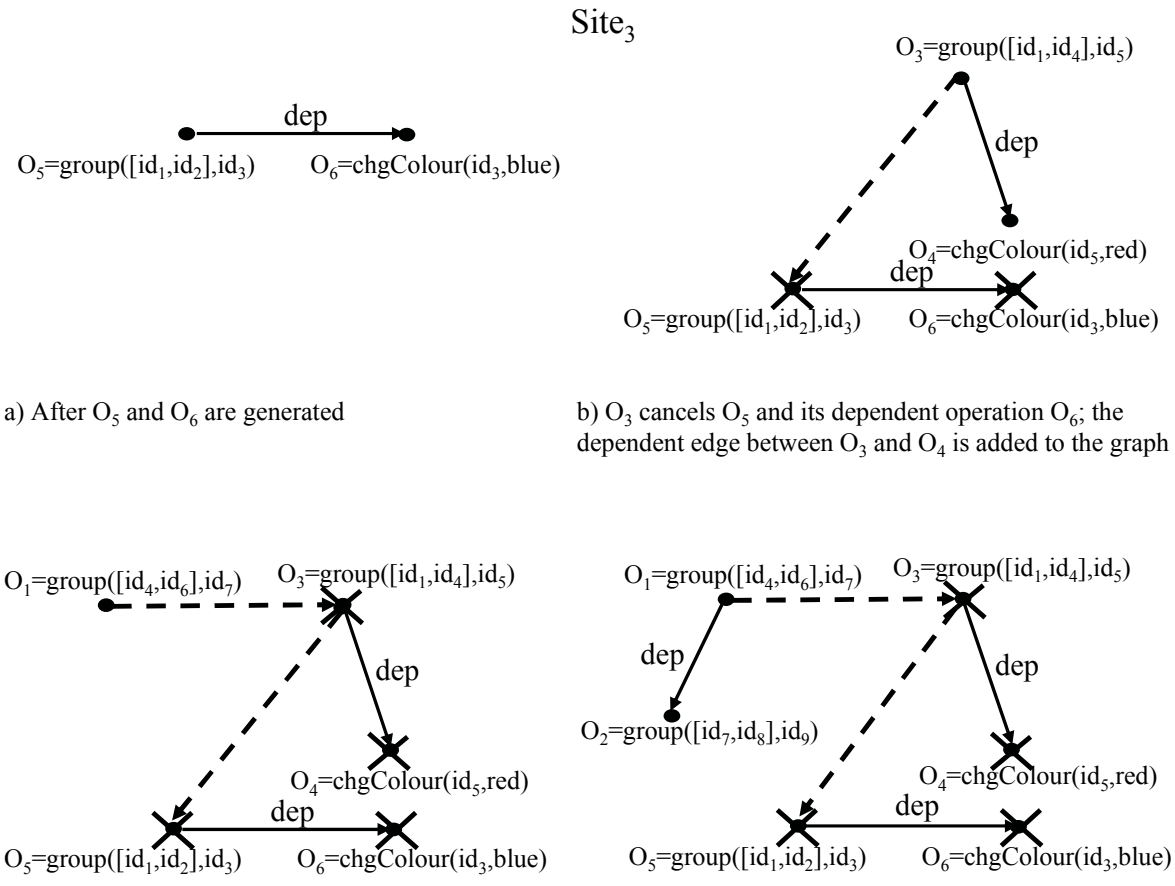


Figure 6.11: The final scene of objects at $Site_1$



a) After O_5 and O_6 are generated

b) O_3 cancels O_5 and its dependent operation O_6; the dependent edge between O_3 and O_4 is added to the graph

c) O_1 cancels O_3 and O_4, the dependent operation of O_3, is cancelled, too

d) The dependent edge between O_1 and O_2 is added to the graph

Figure 6.12: The steps for the construction of graph at $Site_3$. The continuous lines represent edges of the serialisation graph, while the dashed lines represent edges of the real conflict graph.

The final scene of objects at $Site_3$ is illustrated in the figure 6.13.

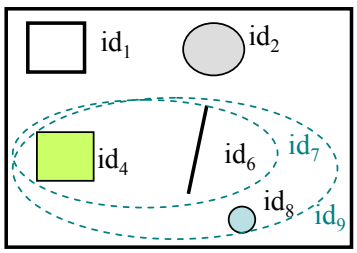


Figure 6.13: The final scene of objects at $Site_3$

As we can see, at $Site_1$ and $Site_3$ the final scenes of objects are not consistent. The reason is that, due to the order of execution of the real conflicting operations, all of the real conflicting operations that could have been executed, according to the priorities of the operations have not been considered. The solution to this issue is that the operations that are real conflicting with an operation have to be considered even if they have been cancelled. If an operation is in real conflict with other operations and it has the highest priority, the real conflicting operations having lower priorities have to be cancelled. By cancelling an operation, their dependent operations have to be cancelled recursively. If an operation is cancelled, the operations in real conflict with it have to be reconsidered as some of the cancelled operations in real conflict might be reactivated. By uncancelling an operation, its dependent operations might be reaccepted if they are not cancelled by other operations. The process for the cancellation and reactivation of an operation is repeatedly applied throughout the graph.

In our example, when operation O_1 arrives at $Site_3$, it cancels operation O_3 and its dependent operation O_4, as shown in Figure 5c. By the cancellation of O_3, the real conflicting operation O_5 previously cancelled due to O_3 should be reactivated. Due to the reactivation of O_5, O_6 should be reactivated too. In this way, the set of operations executed at $Site_1$ and $Site_3$ are the same and therefore consistency is preserved.

6.5.2 Integration of an operation

We next present a procedure for the integration of an operation into the history buffer.

Algorithm *integrate(O,SGraph,RCGraph,History)* {
  addNode(O,SGraph);
  addNode(O,RCGraph);
  //add the edges corresponding to the relations

```
    //between O and the other operations in History
    for (i:=0;i<size(History);i++)
        if (concurrent(O,History[i]))
            if (realConflict(O,History[i]))
                if (priority(O)>priority(History[i]))
                    addEdge(O,History[i],RCGraph)
                else
                    addEdge(History[i],O,RCGraph)
            else
                if (rightOrderConflict(O,History[i]))
                    addEdge(O,History[i],SGraph);
                else
                    if (reverseOrderConflict(O,History[i]))
                        addEdge(History[i],O,SGraph);
        else
            if (dependsOn(O,History[i])){
                addEdge(History[i],O,SGraph);
                markDependentEdge(History[i],O,SGraph);
            }
            else
                if (precedes(History[i],O))
                    addEdge(History[i],O,SGraph);
    //recursively cancel and reactivate nodes in the graph
    propagateCancelOps(O,SGraph,RCGraph);
    //compute the topological sort
    TopSort:=topologicalSort(SGraph,History);
    //find the maximum set of operations from History that
    //conform to the computed topological order
    i:=0;
    while (i<size(History) and History[i]=TopSort[i] and
           not(changedStatus(History[i])))
        i:=i+1;
    //undo the last operations in History that need to be reordered
    for (j:=size(History)-1;j≥i;j−−) undo(History[j]);
    //redo the undone operations in the order specified by
    //the computed topological sort
    for (j:=i;j<size(TopSort);j++) redo(TopSort[j]);
    setHistory(TopSort);
}
```

The procedure *integrate* integrates operation O into the history buffer *History*, by taking into account the real conflict graph *RCGraph* and the serialisation graph *SGraph* constructed from the nodes in *History*.

The new node O is added to *RCGraph* and *SGraph*. For each concurrent operation in *History*, *History[i]*, an edge between O and *History[i]* is added in *RCGraph* or *SGraph*, depending on the type of conflict between them. If the two operations O and *History[i]* are not concurrent, a check is made

to see whether O depends on *History[i]* or whether *History[i]* precedes O and the corresponding edges are added to *SGraph*. Due to the insertion of node O and the insertion of various types of edges between O and other nodes in the graph, some nodes might change their status, from accepted to cancelled or vice versa. A recursive propagation of the cancellation and reactivation of nodes is performed in the procedure *propagateCancelOps* presented later on in this subsection.

The topological sort of the nodes in *SGraph* that maintains the maximum set of ordered operations in *History* starting with the first operation is saved into the list *TopSort*. Therefore, the heads of the lists *History* and *TopSort* are common and the remaining operations in *History* have to be undone and re-executed in the order specified by *TopSort*. The starting index for the process of undoing the operations is determined by traversing the list *History* from left to right and finding the first operation that does not conform to the ordering in *TopSort* or has changed its status from accept to reject or the other way around.

We next present procedure *propagateCancelOps* that recursively propagates the cancellation and reactivation of nodes.

```
Algorithm propagateCancelOps(O_new,SGraph,RCGraph) {
  Nodes:=[];
  addFirst(O_new, Nodes);
  setStatus(O_new, cancel);
  //process each node from the list Nodes and check if
  //its status has to be changed
  while (!isEmpty(Nodes)) {
    changed:=false;
    O:=removeFirst(Nodes);
    //if O has status accept
    if (getStatus(O)=accept){
      //suppose the final status of O remains accept
      isFinalState:=true;
      //if other operations cancel O, the status of O becomes cancel
      DepOps:=getNeighbors(filterDependent(getInboundEdges(O,SGraph)));
      for (i:=0;i<size(DepOps);i++)
        if (getStatus(DepOps[i])=cancel) {
          setStatus(O,cancel);
          isFinalState:=false;
          break;
        }
      if (isFinalState) {
        RCOps:=getNeighbors(getInboundEdges(O,RCGraph)));
        for (i:=0;i<size(RCOps);i++)
          if (getStatus(RCOps[i])=accept); {
            setStatus(O,cancel);
```

```
                isFinalState:=false;
                break;
            }
        }
        //if an operation cancelled O, O changed its status
        if (!isFinalState)
            changed:=true;
    }
    else {
        //if O has status reject, assume O changes its status
        isFinalState:=false;
        //if other operations cancel O, O keeps its status
        DepOps:=getNeighbours(filterDependent(getInboundEdges(O,SGraph)));
        for (i:=0;i<size(DepOps);i++)
            if (getStatus(DepOps[i])=cancel) {
                isFinalState:=true;
                break;
            }
        if (!isFinalState) {
            RCOps:=getNeighbours(getInboundEdges(O,RCGraph));
            for (i:=0;i<size(RCOps);i++)
                if (getStatus(RCOps[i])=accept) {
                    isFinalState:=true;
                    break;
                }
        }
        //if no operation canceled O, O has to change its status to accept
        if (!isFinalState){
            changed:=true;
            setStatus(O,accept);
        }
    }
    //if status of O changed, add to Nodes the operations
    //that might change their status due to O
    if (changed) {
        setChangedStatus(O);
        for (O in getNeighbors(filterDependent(getOutboundEdges(O,SGraph))))
            addLast(O,Nodes);
        for (O in getNeighbors(filterRealConflict(getOutboundEdges(O,RCGraph)))
            addLast(O,Nodes);
    }
  }
}
```

The first argument of procedure *propagateCancelOps* is operation O_{new} that was added to the *SGraph* and *RCGraph*. It might be cancelled or generate the cancelling of other operations. The second and third argu-

ments of the procedure are the graphs *SGraph* containing the ordering relations between operations and *RCGraph* containing the real conflicting operations.

The idea of the algorithm is to add the nodes that might change their status to a list and check, for each node in the list, whether it should keep its current status and then add to the list the nodes that it might in turn cause to change status.

The list *Nodes* contains the nodes whose status has to be checked. At the beginning, operation O_{new} is added to the list with the status of a cancelled operation.

A set of iterations is performed over the list *Nodes* and, at each step, the first element in the list is checked to determine if it should keep its status. If the operation has to change its status from accepted to cancelled or the other way around, its dependent nodes and its real conflicting nodes that have a lower priority are added to the list *Nodes*. No more iterations have to be performed when the list *Nodes* is empty. For the first operation O in the list *Nodes*, two cases are distinguished depending on whether the status of O is accept or cancel.

The flag *isFinalState* indicates whether operation O can keep its status. If the status of O is accept, we make the assumption that the final state of O is accept and therefore set the flag *isFinalState* to *true*. A check has to be made to see whether *SGraph* contains a cancelled operation on which O depends that cancels O or *RCGraph* contains an active conflicting operation that cancels O. If this is the case, O has to be cancelled and therefore *isFinalState* has to be set to *false*. This means that O has changed its status and therefore the flag *changed* is set to *true*.

If the status of O is cancel, we make an assumption that this is not the final state of O and therefore set the flag *isFinalState* to *false*. We check whether *SGraph* contains a cancelled operation on which O depends that cancels O or *RCGraph* contains an active conflicting operation that cancels O. If this is the case, O has to be cancelled. Therefore, O keeps its original state and the flag $isFinalState$ has to be set to *true*. If there is no operation that cancels O, i.e. *isFinalState* remains set to *false*, it means that our assumption that the final state of O is accept holds. Therefore, O changed its status and flag *changed* has to be set to *true*.

If, as result of the verifications, operation O changed its status, all operations that might change their status due to the changing of the status of O are the dependent operations on O and the real conflicting operations of a lower priority than O.

The serialisation order of the operations is the topological sort of the

serialisation graph. If the graph has a cycle there is no solution for the serialisation order. As a graph has a set of corresponding topological sort orders, there are different ways of reordering the operations. To find an order between $O_1, O_2, \ldots, O_n, O_{new}$, we chose the topological sort that maintains the maximum set of ordered operations in the $HB = [O_1, O_2, \ldots, O_n]$ starting with the first operation. Therefore, a minimum number of operations from *HB* have to be undone in order to perform the reordering.

The *topologicalSort* function together with the auxiliary procedure *addToResultAndUpdate* that it calls is presented in what follows.

```
Algorithm topologicalSort(Graph,History):Result {
    Result:=[];
    Nodes:=getNodes(Graph);
    //NoEdges is a hashmap containing pairs between nodes
    //and the in-degrees of those nodes
    for (i:=0;i<size(Nodes);i++)
        put(NoEdges, (Nodes[i],getInDegree(Nodes[i],Graph)));
    k:=0;
    //add those nodes belonging to History to Result
    //if their in-degrees=0 and their status did not change
    while (k<size(History) and get(NoEdges,History[k])=0 and
            not(changedStatus(History[k]))){
            addToResultAndUpdate(Result,History[k],Graph,NoEdges);
            k++;
    }
    //sort the remaining nodes in their topological order
    //iterate size(Nodes)-k times over Nodes and choose
    //each time a node of in-degree 0
    for (i:=0;i<size(Nodes)-k;i++) {
        ind:=-1;
        for (j:=0;j<size(Nodes);j++)
            if (get(NoEdges,Nodes[j])=0)
                if (ind=-1)
                    ind:=j;
            else
                if (getId(Nodes[j])<getId(Nodes[ind]))
                    ind:=j;
        //if there is no node having the in-degree=0,
        //no topological order exists
        if (ind=-1){
            Result:=null;
            return Result;
        }
        addToResultAndUpdate(Result,History[ind],Graph,NoEdges);
    }
    return Result;
}
```

```
Algorithm addToResultAndUpdate(Result,Node,Graph,NoEdges) {
    append(Result,Node);
    //mark Node to not be considered in the next step of the topological order
    put(NoEdges,(Node,-1));
    //decrease the in-degree of the neighboring nodes
    //corresponding to the outgoing edges of Node
    for (NeighbourNode in getOutNeighbours(Graph,Node)) {
        NeighbourInDegree:=get(NoEdges,NeighbourNode);
        put(NoEdges,(NeighbourNode,NeighbourInDegree-1));
    }
}
```

The *topologicalSort* function takes as arguments the serialisation graph *Graph* that is used for the reordering of operations and the old history buffer *History* before the integration of the new operation. The function reorders the nodes in the graph according to the ordering relations between the operations represented by the edges in the graph. The process of reordering tries to maintain the maximum set of ordered operations in *History*, starting with the first operation in the list. Note that *Graph* contains the new remote operation that has to be integrated in the history buffer, while *History* does not contain it. The function returns the reordered list of operations in the serialisation graph. Therefore, *Result* will represent the new history buffer.

The process of building a topological sort of a graph implies considering, in turn, the nodes that have the in-degree 0. After a node that has the in-degree 0 is considered, the edges from that node to the neighbouring nodes are removed.

Nodes is initialised with the list of nodes in the graph and *NoEdges* is a hash map containing, for each node in the graph, the in-degree of the node, i.e. the number of edges having that node as target. The history is traversed from left to right and, as long as an operation *History[k]* has the in-degree 0 and has not changed its status in the recursive process of cancellation of operations, it is considered in the topological sort. During the process of cancellation, some operations might have changed their status, from being cancelled to being accepted or the other way around and, therefore, the ordering between these operations and the other operations might have changed and it no longer conforms to the ordering in *History*. The procedure *addToResultAndUpdate* appends the operation *History[k]* to *Result* and, in order that *History[k]* is no longer considered in the ordering process, the operation is marked to have the in-degree -1. The neighbouring operations of *History[k]* in the graph corresponding to the outgoing edges have to have their in-degree decreased by 1, as operation *History[k]*

has been already considered in the topological sort and its neighbouring edges have to be eliminated.

After the operations in *History* have all been considered or an operation that either has in-degree different than 0 or has its status changed, the other operations in the graph that were not considered have to be added to the topological sort. The number of nodes that still have to be added to the topological sort is equal to the number of total nodes minus the number of operations in *History* that have been included in the sort.

A node can be added to the topological sort order when it has the in-degree 0. However, there are usually several nodes that have in-degree 0. Therefore, history buffers obtained by different topological sorts are equivalent. Obtaining the same history that contains the operations in a global order is useful in the undo process. In order to obtain the same history at all sites, we use the criterion stating that when two nodes have their in-degree 0, we choose to execute first the operation that was generated from the site with the lowest identifier. Note that it is not possible for two operations generated at the same site to have an in-degree 0, as, between the two operations, a precedes relation exists and thus one of the two operations has an in-degree greater or equal to 1.

A number of iterations equal to the difference between the total number of nodes in the graph and the number of operations in *History* that have been included in the topological sort have to be performed. In each iteration, a node that has in-degree 0 has to be chosen and it has to be the operation generated from the site with the lowest identifier from the set of operations that have not been considered and have an in-degree 0. If in one of these iterations, no operation with in-degree 0 is found, the returned result is null, meaning that there exists no topological sort. This occurs if the set of conflicts between the operations was not correctly defined. If an operation satisfying the above-mentioned conditions is found during the iteration, the procedure *addResultAndUpdate* is called. By calling the procedure, the operation is appended to *Result*, its corresponding entry in the hashtable *NoEdges* is updated with value -1 and the in-degree of its neighbours in the graph is updated as a result of the deletion of the outgoing edges of the operation.

6.5.3 Definition of conflicts

The list of types of conflicts is specified by users in a separate document and can be modified according to various applications. A conflict is given by the specification of the type of the two operations in conflict, the condition

for conflict and the type of conflict. We illustrate this by means of some examples showing how conflicts are defined.

- A delete operation $o1$ is in a resolvable conflict with a group operation $o2$ if the target object of $o1$ belongs to the target list of $o2$. The conflict is resolved by executing first $o2$ operation followed by $o1$.

```
conflict {
    operation o1:delete o2:group
    condition o1.id in o2.inlist
    resolution reverse
}
```

- Two concurrent operations moving the same point of a polyline are in real conflict. The resolution policy is specified to be none, meaning that the conflict is a real conflict.

```
conflict {
    operation o1:pointmove o2:pointmove
    condition o1.id=o2.id and o1.pointid = o2.pointid
    resolution none
}
```

- Two change colour operations $o1$ and $o2$ are in conflict if the target of operation $o1$ belongs to the group targeted by operation $o2$. The resolution policy is to execute $o2$ first followed by $o1$. An additional condition is that the target objects of the two operations are different. If the two change colour operations target the same object, this is a real conflict defined in another conflict rule.

```
conflict {
    operation o1:changecolor o2:changecolor
    condition o1.id != o2.id and o1.id childof o2.id
    resolution reverse
}
```

- Two grouping operations targeting some common objects are in real conflict.

```
conflict{
    operation o1:group o2:group
    condition o1.childrenlist inter o2.childrenlist != emptyset
    resolution none
}
```

6.6 Draw-Together: a collaborative real-time graphical editor

An editor based on the algorithms described in this chapter has been implemented [20]. Running Draw-Together requires a connection to a server where the common documents are stored. The connection to the server requires a login and a password. The server provides to the clients the IP addresses of the other clients that work on the same document such that the client software can send then directly to the other clients a log of operations performed on the common document.

A screenshot of our Draw-Together application interface is given in Figure 6.14. The interface illustrates collaborative work of three users performing a brainstorming session about the organisation of the Collaborative Editing Workshop. One can notice the set of primitives that can be used for drawing - rectangles, lines, ellipses, polylines, text boxes, bitmaps and annotations and the set of operations that can be performed by means of the various buttons included in the toolbars. Note also the use of layers, such as the *Workshops* and *Organisers* layers that can be set visible or invisible, depending on whether the details about the related workshops and organisers of the workshop want to be shown or hidden.

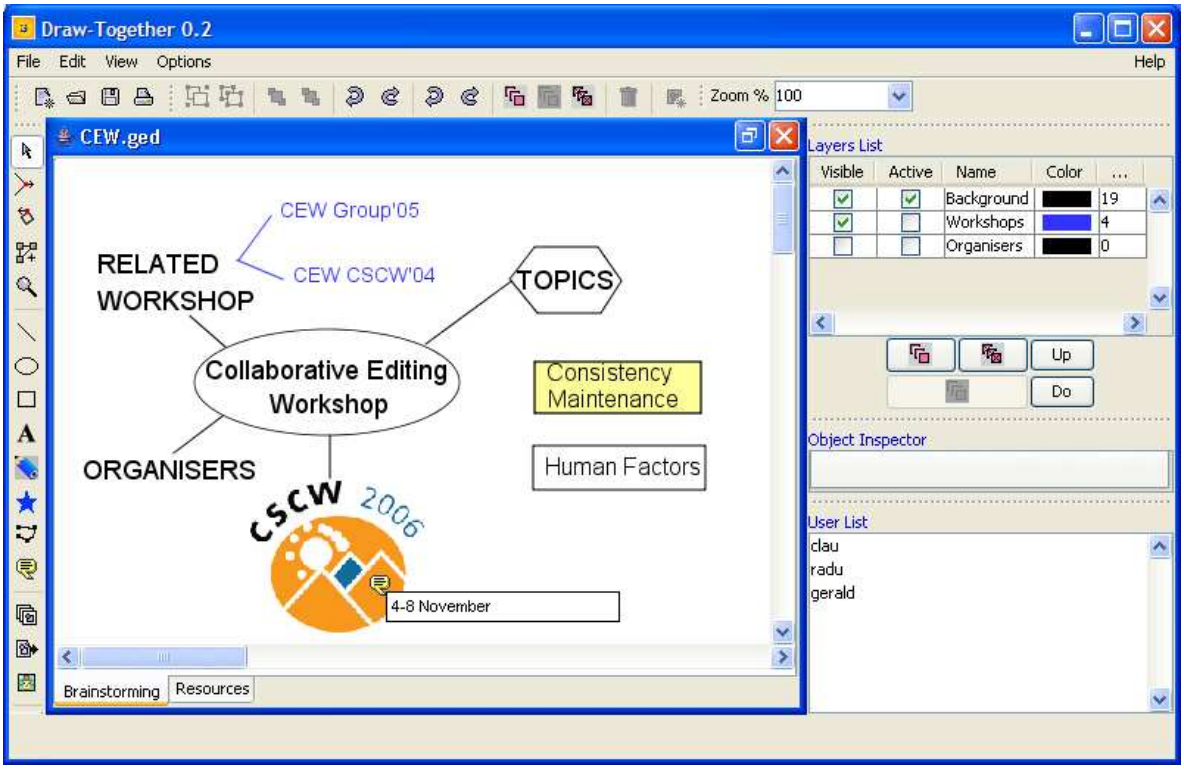


Figure 6.14: Draw-Together application

Another operation offered by Draw-Together that was not specified in the set of operations that can be collaboratively performed on the scene of objects is zooming. Zooming is not performed in a collaborative way as it refers to the view of the document at one site and does not influence the work of other users.

Another facility offered by Draw-Together is document export as SVG (Scalar Vector Graphics).

Draw-Together is based on requirements for collaborative drawing as specified by the Inovation Center Virtual Reality (Institut für Werkzeugmaschinen und Fertigung) Institute at ETH Zurich who are especially interested in supporting early stages of product development involving product sketching and brainstorming sessions.

6.7 An asynchronous graphical editor

In this section we present two approaches to merging for graphical documents [43]: an operation-based approach relying on the serialisation of operations and a state-based merging approach. We show that the same serialisation approach that was used for maintaining consistency in real-time communication could be applied to asynchronous collaboration with a central repository. We also discuss the advantages and disadvantages of using an operation-based or a state-based approach to merging and present our state-based approach to merging used in object-based graphical editing.

6.7.1 Merging based on serialisation

As in the case of text editing, in the operation-based approach to the merging of graphical documents, all the operations performed by the user are kept in a log at the client side. The repository keeps for each document version a list of operations that transform the previous version into the current version.

The repository stores the following information:

- *version*, representing the version of the document stored in the repository
- *delta[1..version]*, where *delta[i]* is a list of operations representing the difference between version *i+1* and version *i*.

On the client side, the following data are stored:

- *version*, representing the version of the local document
- *LD*, representing the local document
- *LSD*, representing the last synchronisation document, i.e. last version from the repository that was integrated into *LD*
- *Log*, consisting of the list of operations executed locally and not yet committed, representing the difference between *LD* and *LSD*.

In what follows we describe the basic operations of checkout, commit and update used in asynchronous communication.

The checkout operation creates a local working copy of the requested version of the document from the repository. The repository sends all the operations needed to transform an empty document into the document associated with the requested version. The commit operation creates a new version of the corresponding document in the repository by sending the client log to the repository. This operation is performed only if the local document is up-to-date, i.e. the repository does not contain a new version committed by other users.

The update operation performs a merge between the local copy of the document and the last version of the document from the repository. The list of operations from the repository representing the difference between the last version in the repository and the version in the local workspace is sent to the local workspace. Each operation is then integrated into the local log, by applying the serialisation mechanism that has been used for real-time communication. The delta to be stored in the repository is obtained by inverting the remote log list and adding the operations from the new local log. The inverse of the remote log has to be performed in order to cancel the effect of the operations from the repository such that the new local log already containing the effect of the remote operations can be considered as delta in the repository. The inverse function applied on a sequence of operations returns the list of operations that need to be executed right after the execution of the given sequence of operations in order to cancel all their effects. Therefore, $inverse([O_1,\ldots,O_N])=[inverse(O_N),\ldots,inverse(O_1)]$.

The *update* procedure is given below. The argument *LL* represents the local log and the argument *RL* represents the remote log. The procedure returns the new local log *NewLL* representing the local log transformed by the application of the operations from the repository and the *NewRL* representing the new delta in the repository.

```
Algorithm update(LL, RL):(NewLL,NewRL) {
    NewLL:=LL;
    for (i:= 0;i</RL/;i++)
        NewLL:=integrate(RL[i],SGraph,RCGraph,NewLL);
    NewRL:=inverse(RL)+NewLL;
    return (NewLL,NewRL)
}
```

The local document LD is then updated with the version of the document obtained by applying the operations from $NewLL$ on the LSD version of the document. The LSD is updated with the last version of the document from the repository, obtained by applying the list of operations LL on the old value of the LSD.

6.7.2 State-based merging

In the state-based approach the *checkout* routine consists of the repository sending the document corresponding to the requested version. In the *commit* routine, the client sends a local document to the repository. In the *update* routine, the last version of the document from the repository is merged with the local document. As checkout and commit routines involve just the sending of a document version, in what follows we are going to describe just merging.

Merging has, as arguments, two graphical documents - a copy of the local document LD and the remote document from the repository RD, and returns a new document resulting from combining the two documents. Merging is carried out relatively to the last synchronisation document LSD, i.e. the last version from the repository that was integrated into the local document. Two different merging algorithms are used: one for object attributes and one for tree structures. In the following both algorithms are presented.

Merging of object attributes is carried out only if leaf nodes are present in all three documents LSD, LD and RD. Otherwise, a change in the tree structure is detected and the case is handled by the algorithm for tree structure merging. If tree structures are the same and if a certain attribute from a leaf S was modified only in one document LD or RD, the modification is simply kept in the resulting document. A change concerning the same attribute in both documents is considered a conflict. In this case none or only one modification is kept depending on the chosen policy. The function *attributesMerge* applying to object Obj is presented below.

```
Algorithm attributesMerge(Obj,LSD,LD,RD):NewObj {
  for (A in attributes(Obj)) {
      attribute(NewObj,A):=attribute(Object(LSD,Obj),A);
      if ( attribute(Object(LSD,Obj),A)≠attribute(Object(LD,Obj),A) or
           attribute(object(LSD,Obj),A)≠attribute(object(RD,Obj),A))
          if (attribute(object(LSD,Obj),A)=attribute(object(LD,Obj),A))
              attribute(NewObj,A):=attribute(object(RD,Obj),A)
          else if (attribute(object(LSD,Obj),A)=attribute(object(RD,Obj),A))
                  attribute(NewObj,A):=attribute(object(LD,Obj),A)
          else if (no conflicting operation policy)
                  attribute(NewObj,A):=attribute(object(LSD,Obj),A)
          else if (local winner policy)
                  attribute(NewObj,A):=attribute(object(LD,Obj),A)
          else attribute(NewObj,A):=attribute(object(LD,Obj),A)
  }
  return Obj;
}
```

The precondition for applying the function is that the object belongs to each of the three documents LSD, LD and RD. The function *attributesMerge* is applied to each leaf belonging to all three documents LSD, LD and RD. The arguments passed to the *attributesMerge* function are Obj representing the leaf or the object for which the merging of attributes is applied, the local document LD, the remote document RD and the last version from the repository that was integrated into the local document LSD. An iteration over each attribute of the object Obj is performed and if a certain attribute was modified only in one document LD or RD, the modification is kept in the resulting object. The changing of an attribute in both documents is considered a conflict and depending on the policy adopted, none of the changes, the remote or the local change is kept. The function *object(D,Obj)* returns the object from document D corresponding to the reference Obj. The function *attribute(Obj,A)* returns the attribute of object Obj corresponding to the reference A.

The *tree merging algorithm* handles the changes related to the tree structures. The graphical documents LD, RD and LSD are transformed into directed acyclic graphs (DAG), the graphical objects becoming vertices, the edges being oriented from parents to children.

Three graphs, $G_L = (V_L, E_L)$, $G_R = (V_R, E_R)$ and $G_{LS} = (V_{LS}, E_{LS})$ are obtained as result of the transformation of LD, RD and LSD documents into DAGs.

To handle the conflicts that occur during the merging process, a priority function is defined:

$$priority(D_1, D_2) = \begin{cases} 0, & \text{if the changes in } D_2 \text{ have priority} \\ 1, & \text{if the changes in } D_1 \text{ have priority} \end{cases}$$

A cost is associated with each edge from G_L and G_R:

$$cost(u) = \begin{cases} 0, & \text{if } u \in E_{LS} \\ 1 + priority(LD, RD), & \text{if } u \notin E_{LS} \end{cases} \text{ where } u \in E_L$$

$$cost(u) = \begin{cases} 0, & \text{if } u \in E_{LS} \\ 1 + priority(RD, LD), & \text{if } u \notin E_{LS} \end{cases} \text{ where } u \in E_R$$

In what follows we describe the merging algorithm. Firstly, a new graph $G = (V_L \cup V_R, E_L \cup E_R)$ is constructed containing all the vertices and edges from both G_L and G_R. In order to convert G to a graphical document, some edges must be removed. A DAG is a tree if the in-degree of all vertices is 1, i.e. every vertex has only one parent. The vertices of the graph are connected either by one edge or by two edges to their parent. The vertices connected by one edge to their parent indicate the removal or the addition of a graphical object from/to a document. If the cost of the edge that connects the vertex with its parent is 0, the edge is removed from the graph. Otherwise, no modification is made. In vertices connected by two edges to their parent, the edge with lower cost that connects the given vertex with its parent is removed from the graph. In the end, all the vertices that are not in the same connected component as the root are removed from the graph.

Consider the scene of objects shown in the left hand side of figure 6.15 consisting of two groups identified by id_3 and id_8. Group id_3 contains two objects, id_1 and id_2 and group id_8 is formed by group id_6 and object id_7. Group id_6 is composed of objects id_4 and id_5. Two users, $User_1$ and $User_2$, checkout the version of the document representing this scene of objects and concurrently modify it. To resolve conflicts we consider that local changes have priority over remote changes. $User_1$ adds object id_9 to the scene, as shown in the middle part of figure 6.15 and commits the new version into the repository. $User_2$ groups the groups id_3 and id_8 into a new group id_{10}, adds object id_{11} to the scene and deletes id_2, as shown on the right hand side of figure 6.15. $User_2$ tries to commit the changes into the repository, but needs to first update the local workspace with the changes from the repository.

The merging routine consists of merging the graphs G_R and G_L. The intermediate graph resulting from the union of the vertices and the edges of the graphs G_R and G_L is shown in figure 6.16 (left). The intermediate graph is then transformed into the final graph shown in figure 6.16 (right).

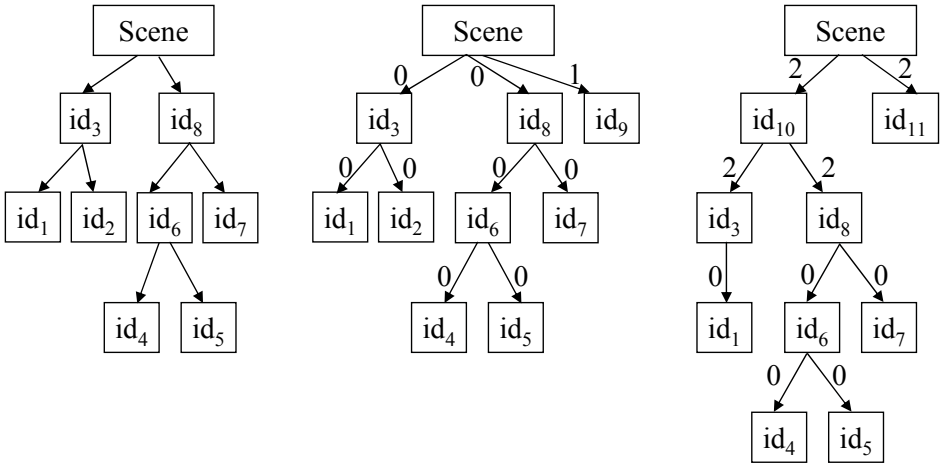


Figure 6.15: Graphs corresponding to the initial scene of objects and to the changes performed by two users; left - initial scene of objects (graph G_{LS}); middle - changes performed by $User_1$ (graph G_R); right - changes performed by $User_2$ (graph G_L)

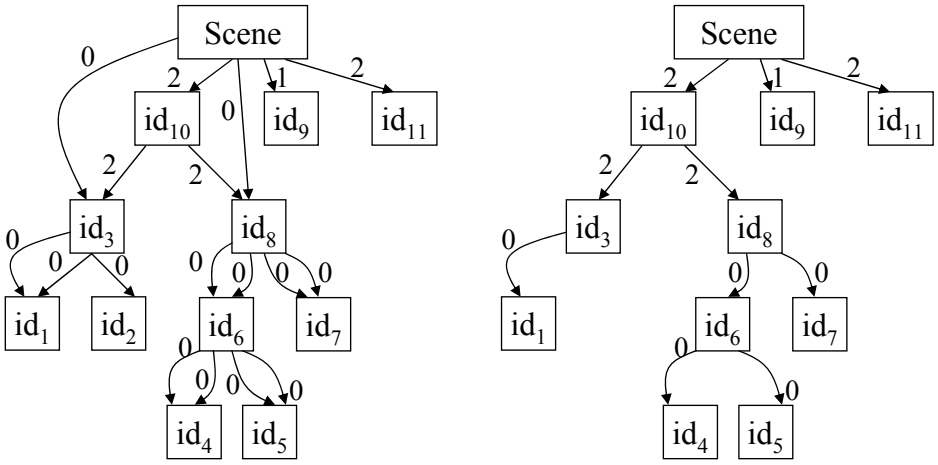


Figure 6.16: Merging Graphs; left - intermediate graph; right - final graph

6.7.3 Operation-based versus state-based merging

We presented two approaches for maintaining consistency for the object-based graphical documents in section 6.7.1 and section 6.7.2, respectively. In this section we are going to compare the two approaches. We implemented the two approaches described in the previous two sections [115].

The state-based merging algorithm compares the states of the documents to be merged in order to generate the merged result, while the operation-based merging algorithm uses the differences between the two documents and a reference document.

The operation-based approach is more efficient than the state-based approach in the case of large documents, because the number of operations that transform version i into version $i+1$ would be statistically smaller than the number of graphical objects. The state-based approach is more efficient than the operation-based approach to merging in the case of small documents where the number of operations is comparable with the number of objects from the graphical document. The operation-based approach has some advantages over the state-based approach for merging such as a better resolution of conflicts which preserves a combined intention of concurrent operations.

Consider a scene of objects consisting of a group identified by id_3 that contains a rectangle identified by id_1 and an ellipse with identifier id_2, all coloured white. Two users concurrently modify this scene. Suppose $User_1$ changes the colour of group id_3 to blue by issuing the operation O_1=*chgColour(*id_3*,blue)* and commits. Concurrently, $User_2$ changes the colour of the rectangle to green by issuing the operation O_2=*chgColour(*id_1*, green)* and tries to commit. $User_2$ needs to first update the version of his document. In state-based merging, a conflict is detected for the rectangle id_1 since its colour is modified both locally and remotely. In operation-based merging, the two operations are detected as conflicting, the type of conflict being specified by the application and it can be a real conflict or a resolvable conflict. If the conflict is real, the result is the same as in state-based merging, as none of the operations or only the local or the remote operation are performed. But, if the conflict is resolvable, and the order of execution of the operations is a colour change of the group followed by a colour change of the rectangle, a partially combined effect of the intentions of $User_1$ and $User_2$ is obtained, i.e. the rectangle will be coloured green and the ellipse blue.

In the operation-based approach, conflicts are detected only between real conflicting operations and a partial preservation of the intentions of

the users is enabled by a serialisation of the resolvable conflicting operations. On the other hand, in the state-based approach, conflicts are generated whenever a property of an object has different values in the two documents to be merged. Moreover, in operation-based merging, rules for the definition and resolution of conflicts can be defined. The real and resolvable conflicts can be defined between pairs of operations, depending on the application. In state-based merging, there is only one way to define and resolve conflicts. The only situation of conflict occurs when a property of an object has different values in the two documents to be merged and the policies for dealing with conflict are either to keep the local or remote changes or to let the user choose manually the modification to be kept.

6.8 OT versus constraint-based serialisation

The operational transformation approach transforms an operation against a set of previously executed operations without the need to modify any of the previous operations. In text editing objects are identified by their positions in the document. For instance, if the document is a sequence of characters, each character is identified by its position in the document. In our work we looked at text as a hierarchical structure and each element was identified by a position relative to its parent. For instance, a paragraph is identified by its position in the document, a sentence by its position in the paragraph it belongs to, and so on. In the same way, in XML documents an element is identified by its position in the list of children of its parent. Operations that are performed are *insert* and *delete* of elements, and, usually, operational transformation adapts the positions of these operations in the face of other concurrent operations. In most of the cases, the transformation of an operation does not need to cancel a previous operation. However, there exist cases when this cancelling is required such as in [68]. The cancellation is achieved by the use of composite operations as result of the transformations. In this case, the composite operation is composed of the operation that cancels the effect of one of the previously executed operation, followed by a transformed form of the operation. Since in this case the SOCT4 approach based on a centralised time-stamping scheme is used for maintaining consistency, no backward transposition is required. Therefore, the fact that the transformation of an operation results in a sequence of operations does not affect the correctness of the approach. However, no other decentralised approaches deal correctly with these composite operations.

The serialisation approach, on the other hand, undoes a sequence of operations and re-executes them with the possibility of cancelling some of the previous operations and reordering other operations. In this way, ordering constraints between operations can be defined and the serialisation approach reorders the operations in order to satisfy these constraints.

The operational transformation approach is a more efficient approach than serialisation, as it does not require that operations are undone. But due to the fact that there are cases when previous operations have to be cancelled or the transformation of an operation yields a sequence of operations, and therefore the approach does not satisfy the correctness conditions, serialisation is an alternative approach where ordering constraints can be defined between operations.

6.9 Related work

In this section we relate our work to some of the existing approaches for maintaining consistency in object-based graphical documents.

Some of the approaches for consistency maintenance are based on locking. Locking guarantees that users access objects in the shared workspace one at a time and concurrent editing is allowed only if users are locking and editing different objects. Non-optimistic locking introduces delays in acquiring a lock. Optimistic locking avoids the delays, but it is not clear what to do when locks are denied and the object optimistically manipulated by the user must be restored to its original state. Another problem associated with locking is deciding on locking granularity, the choice being a balance between locking overhead and the amount of concurrency desired. Systems such as Aspects [118], Ensemble [76] and GroupDraw [33] rely on locking.

Some of the existing approaches for maintaining consistency in graphical editing are based on a serialisation mechanism that ensures that the effect of executing a group of concurrent operations is the same as if the operations were executed in the same total order at all sites. If there is any conflict among concurrent operations, only the effect of the last operation in the total ordering is maintained. LICRA [55] and GroupDesign [56] are examples of prototypes implementing this technique. LICRA relies on dependency relations between the operations and on the operational transformation mechanism, while GroupDesign uses an undo/redo mechanism. The difference from our approach is that LICRA and GroupDesign do not consider operations of grouping and ungrouping, while our approach considers them. Moreover, LICRA and GroupDesign use a serialisation mech-

anism based on the total order between operations, the total order being defined according to the timestamps of the operations. Our approach is based on the serialisation of the operations in order to obtain a combined effect of the intentions of the concurrent operations.

Another solution for consistency adopted in the collaborative editing is the multi-versioning approach. The multi-versioning approach tries to achieve all operation effects, preserving the intentions of all operations. For each concurrent operation targeting a common object, a new version of the object is created. As already mentioned in chapter 2, the multi-versioning approach raises some issues related to the graphical user interface, such as the way the versions of an object are related to the base object or the way the navigation through the versions of an object is realised. GRACE [105] and TIVOLI [70] are two prototypes that rely on multi-versioning. GRACE and TIVOLI do not deal with operations of grouping and ungrouping.

In what follows we compare our approach with IceCube [86, 57]. IceCube proposes a consistency maintenance algorithm based on constraints between actions and the action-constraint framework as a formalism for maintaining consistency in replicated systems where constraints are defined. We compare our serialisation approach with IceCube and action-constraint framework approaches, as they are based on constraints which are similar to the ordering constraints in our approach. We afterwards compare our approach with CoGroup [121] as it deals with grouping and ungrouping operations as in our approach.

IceCube [86, 57] is a generic reconciliation system. Reconciliation is viewed as an optimisation problem of scheduling a maximum number of concurrent actions, being given a set of constraints. Constraints can be static or dynamic. Static constraints are evaluated independently of the object state. Dynamic constraints cannot be detected statically, i.e. without attempting to execute the actions. IceCube uses static constraints. A scheduling stage produces schedules that satisfy the static constraints. The schedules obtained in this phase are verified in a simulation stage, where actions are executed against a copy of the state to check dynamic conflicts. At the end a selection stage chooses the schedules that satisfy the dynamic constraints. The disadvantage of this approach is that during the reconciliation it generates a combinatorial explosion of solutions to be analysed. In order to achieve convergence of the n sites involved in the collaboration, in the IceCube a common site is responsible for achieving the reconciliation. All the other sites have to send to this site all operations executed since last synchronisation. The reconciliation mechanism is applied and the combined log is sent to all sites. In our approach the constraints are

defined by the relations between the operations: preceding, right order, reverse order, dependence and real conflict. We do not distinguish between static and dynamic constraints. As opposed to the centralised approach of IceCube, our approach is distributed. Moreover, the algorithm that we designed is incremental and it integrates a new operation into an already computed schedule by reusing a maximal set of orderings that remain valid after the integration of the remote operation. The new scheduler satisfies the total set of constraints between the remote operation and the previously integrated operations.

In [92, 91] a formalism modelling replication in a distributed system where users concurrently work on shared data has been proposed. The approach is based on actions representing operations executed by users and a set of constraints representing scheduling relations between actions. The approach unifies data semantics such as commutativity and conflicts with user intents such as causal dependence or atomicity. The approach offers support for reasoning about consistency properties of replication protocols. The Action-Constraint Framework [93] is a continuation of work described in [92, 91] and it defines three dimensions for consistency by the mapping to three problems on subgraphs of a multilog: conflict breaking, agreement and serialisation. Each site has a local view of known actions and constraints called multilog. The constraints defined for the multilog are the following:

- $\alpha \rightarrow \beta$ or "α Before β" indicating that α should be executed before β. A schedule that executes neither α nor β, or only α or only β or both α and β in this order is correct with respect to this constraint.

- $\alpha \triangleright \beta$ or "β MustHave α" indicating that if α is executed then β must also be executed, although not necessarily in this order. A schedule that executes only β, or that executes neither α nor β is correct with respect to this constraint.

- $\alpha \not\parallel \beta$ or "α non-commuting β" indicating that α does not commute with β. Two actions commute if by executing them in either order an equivalent state is obtained.

Some special sets of actions have been defined:

- *Guar(M)* is the set of guaranteed actions. It is the smallest set that satisfies the property that the initial action belongs to the set ($INIT \in Guar(M)$) and if an action is guaranteed, all the actions

it "must have" are guaranteed, too ($\forall \beta \in A :$ if $\alpha \in$ *Guar(M)* and $\beta \triangleright \alpha$ then $\beta \in$ *Guar(M)*).

- *Dead(M)* is the set of dead actions that are not executed in any schedule.

- *Serialised(M)* is the set of serialised actions. An action is serialised if it is ordered with respect to all non-commuting actions that are executed.

- *Decided(M)* is the set of decided actions, i.e. actions that are either dead or are both guaranteed and serialised.

An initial graph is constructed, where the nodes of the graph represent actions and the edges represent the *before*, *mustHave* and *non-commuting* relations between actions. As previously mentioned the consistency problem is divided into three subproblems by the division of the initial graph into three other graphs: before, mustHave and serialisation graphs. The before graph contains the before edges from the initial graph, the mustHave graph contains the mustHave edges from the initial graph and the serialisation graph contains the before and non-commuting edges from the initial graph. Processing of these graphs offers support for reasoning about the correctness of different consistency protocols.

We can map our problem for maintaining the consistency to the ACF framework. Our *precedes* and *resolvable conflicting* relations between the operations can be mapped to the *before* relations in the ACF framework. Our *depends* relations between operations are both *before* and *mustHave* relations. The set of non-commuting operations in the ACF framework includes the set of operations that are in the *before* relationship. The user specifies in a document the types of conflicts between the operations, i.e. the operations that are in a real conflict or resolvable conflict relation. The *dependent* relations between the operations are deduced according to the definition of dependent operations, i.e. O_2 depends on O_1 if O_1 creates an object or group that belongs to the target list of O_2. All other operations excluding the ones that are in a real conflict, resolvable conflict or dependency relation are considered as commuting. The *cancelled* operations correspond to the set of *dead* operations. In our approach we adopted an incremental way of integrating remote operations into the schedule that was built from the set of previous operations. The approach of ACF framework rather considers the whole set of actions that have to be taken into account for building the schedule. By considering the whole set of actions,

the dead, guaranteed, serialised and decided sets can be established. In our approach of incremental integration, we cannot be sure that an action that belongs to one of these sets at a certain step in the integration process belongs to the same set of actions at the next step of integration. For instance, an operation that was cancelled can be uncancelled in the next step of integration. In the same way, an operation that was serialised in one step of the integration can be cancelled in the next step. However, at any point in the incremental integration, the sets of dead, guaranteed, serialised and decided actions can be determined. Our approach deals also with a graph that contains the operations as nodes and the relations between nodes as edges, similar to the serialisation graph. Therefore, the consistency of our approach could be checked by using the ACF framework.

CoGroup [121] proposes an operational transformation solution to collaborative grouping. Their work was very recently published, after we proposed our solution based on the serialisation of operations. The grouping mechanism is proposed in the context of Transparent Adaptation (TA) approach [122] which can convert existing single-user editing applications into real-time collaborative versions, without changing their source code. In TA the shared single-user application is replicated at all sites and the API of the single user application can intercept user operations. The intercepted operations are then processed by an OT framework that achieves a combined effect of multi-user interactions. The transformed operations are then sent back to the single-user applications that generates the corresponding operations for the single-user applications. The main idea of the TA approach is to adapt the data model and the operations of the API of the shared application to the OT mechanism.

The data address model adopted by CoGroup is a tree of multiple linear address domains. An object in this data address model is identified by a position vector $vp = [p_0, p_1, \ldots, p_i, \ldots, p_k]$, where $vp[i] = p_i$, $0 \leq i \leq k$ represents one addressing point at level i. The scene of objects is then mapped to this data address model. The primitive operations supported by the OT model are:

- *insert[pos,obj]* of inserting an object *obj* at position *pos*
- *delete[pos,obj]* of removing the object *obj* at position *pos*
- *update[pos,key,old_value,new_value]* of changing the attribute *key* from *old_value* to *new_value*, of an object at position *pos*

The operations executed by users are complex operations. *createObj*

creates an object. *deleteObj* deletes an object. *changeAtt* changes an attribute (e.g. size, colour or position) of an object. *Group* groups a set of objects or groups and *ungroup* of ungroups a group of objects. The combined effects of these operations have been defined. *deleteObj* is compatible with the effect of any other concurrent operation targeting the same object. For instance, the combined effect of *changeAtt* and *deleteObj* is the change of the attribute and the deletion of the target object and the combined effect of *Group* and *deleteObj* is the creation of a group-object containing all member objects targeted by *group* except the member object targeted by *deleteObj*. *changeAtt* is compatible with any other operation except a *changeAtt* that targets the same element. For instance, the combined effect of a *group* operation with the *changeAtt* operation is the creation of a group-object containing all target member objects and the change of the attribute of one member object targeted by *changeAtt*. In the case of two conflicting *changeAtt* operations, the Multi-Version Single-Display (MVSD) technique has been applied [110]. In MVSD multiple versions of the common target objects are created to accommodate the effects of all conflicting operations, but only one version is displayed. Users are allowed to choose to display any version at a time by using the system undo facility. For instance, in the case of two conflicting *changeAtt* operations, one changing the colour of a group and the other changing the colour of an object from the group, two versions of the common object will be created, corresponding to the two conflicting operations. However, only the version corresponding to the operation with the highest priority is displayed, the other version being displayed in the case of performing an undo. A *group* operation may conflict with another *group* operation if they target common objects, but it is compatible with other operations. Two conflicting *group* operations succeed in creating the target group-objects. Both group objects contain their non-common target objects, but only one of the groups has the common target objects displayed. An *ungroup* operation is always compatible with other operations.

Due to the fact that the OT mechanism in CoGroup is defined just for primitive operations, complex operations have to be translated into primitive operations. The *createObj*, *deleteObj* and *changeAtt* operations are already primitive operations and therefore they do not need translation.

A *group* operation is translated into the following operations: the *insertion* of a group-object in the addressing domain before the first target object and the moving of all target objects into a lower level addressing domain, linked to the group-object. An *ungroup* operation is translated into the following operations: moving the member objects to the position

of the target group-object in the higher level addressing and the deletion of the target group-object A *move* operation is further translated into a sequence of *delete* and *insert* primitive operations.

The operational transformation mechanism is applied only to the primitive operations. Functions for the detection and resolution of conflicts between the complex operations involving Group operations are provided at the API level responsible for the translation of complex operations to the primitive ones. OT functions have been extended such that when a primitive operation PO_1 is transformed against a concurrent operation PO_2 and they have overlapping target objects, the reference to the associated complex operation of PO_2 is recorded in the transformed PO_1.

However, CoGroup [121] does not propose transformation functions for the inclusion and exclusion of operations have not been provided. Therefore, we cannot judge the correctness of the approach, but we can state where problems could appear. Our concern is related to the way exclusion transformations are performed between operations involving groups of objects. We take as example the transformations between two *group* operations. In order to explain this issue, we are going to present first how inclusion transformations are performed between the *group* operations in the CoGroup approach. The algorithm for the adaptation of a complex operation transforms the complex operation into a sequence of primitive operations and then each primitive operation is transformed against the log of primitive operations executed at that site. After each step transforming a primitive operation, the function *GAOConflictDetection(TPO)* is called to detect whether TPO is a primitive component of a group operation that is in conflict with another primitive operation originating from another group operation. If this is the case, a procedure *GAOConflictResolution(TPO)* is called to resolve the conflict between the two *group* operations. According to the priorities of the two *group* operations, the *move* operations associated with the primitive operations are changed accordingly. If the remote operation TPO originating from the group operation GO_1 has a higher priority than the local operation originating from the conflicting group operation GO_2, then the *move* operation associated with GO_1 has to specify that the common target object is moved from the group targeted by GO_2 to the group targeted by GO_1. Otherwise, if the remote operation TPO has a lower priority than the local operation, the *move* operation targeting the common object is cancelled. Consider a scene of objects consisting of the objects having the identifiers id_1, id_2, id_3 and id_4. Two users at $Site_1$ and $Site_2$ concurrently perform the operations of grouping some objects from the scene. The first user groups the objects identified by id_1 and id_2

into group g_1, by performing the operation O_1=*group(*[id_1,id_2]*,*g_1*)*. The second user groups the objects identified by id_2, id_3 and id_4 into group g_2 by performing the operation O_2=*group(*[id_2,id_3,id_4]*,*g_2*)*. Assume that operation O_2 has a lower priority than O_1. When O_2 arrives at $Site_1$, group operations are translated into primitive operations and transformations are performed between the primitive operations. The sequence of resulting operations is equivalent to an operation of grouping id_3 and id_4 into g_2. In this context the following question arise. What is the result of excluding the effect of O_1 from O_2? Is the result equivalent to the original form of O_2? How do the primitive operations achieve this result? The scenario presented in figure 6.7 remains an issue and we did not understand how CoGroup deals with such cases.

The approach is hard to follow and to check for correctness, as different techniques have been combined - the Transparent Adaptation (TA) approach, the Multi-Version Single-Display (MSVD) and the approach for maintaining the consistency over documents having a tree structure presented in [21]. Algorithms and details of the correlations between these approaches are not provided and therefore it is hard to check the correctness of the work.

Our approach offers a flexible way of defining conflicts according to the application needs. Conflicts can be classified into real or resolvable. For instance, if an application wants to define that a *group* operation conflicts with another operation that has a common target, it can do so by defining the two types of operations as real conflicting. A combined effect of these operations can be obtained by defining them as being in a resolvable conflict, i.e. specifying an order of execution, such as *group* operation followed by the other operation.

While it would be very hard to check the correctness of the CoGroup approach, the correctness of our approach could be checked by using the Action-Constraint Framework [93].

Concerning the implementation of the serialisation mechanism for the asynchronous communication over graphical documents, to our knowledge, there are no other systems that offer support for the collaboration on object-based graphical documents over a shared repository.

7 Conclusions

In this chapter we give a summary of the outcomes of this thesis and provide some future work directions in the area of collaborative editing.

7.1 Summary of outcomes

In this thesis we proposed a multi-level editing approach to consistency maintenance for documents with a hierarchical structure. We applied this approach to text and XML documents. Text documents are structured as an ordered sequence of paragraphs, each paragraph containing as an ordered list of sentences, each sentence as an ordered list of words, and each word an ordered list of characters. XML documents are also composed of an ordered list of nested elements.

Our multi-level editing approach implies that each element in the tree structure is associated with a list of operations that have been performed on that node. In this way, resolution of conflicts is simplified, as compared to the approaches that use a single history buffer, as conflicting operations that refer to the same subtree of the document are easily detected by the analysis of the histories associated with the nodes belonging to the subtree. Conflicts can be dynamically defined at different levels of granularity, corresponding to the different levels of the document. Moreover, logging edit operations that refer to each node allows user activity referring to different parts of the document to be easily tracked. As shown in [84] we used this

facility for maintaining group awareness in collaborative editing.

The multi-level editing approach is theoretically more efficient than existing merging algorithms that maintain a single history buffer where edit operations are logged. Rather than scanning the whole log of operations in order to perform transformations, as is the case in a single history buffer, by using the history distributed throughout the tree, when a remote operation is integrated, only the logs that are distributed along a certain path in the tree are scanned and transformations applied.

Our multi-level transformational approach recursively applies a linear operational transformation approach to the document tree. Although the document model is hierarchical, transformation functions do not need to be adapted to the tree model. Rather than writing new transformation functions for hierarchical documents, we reuse existing linear transformation functions for operations on characters and apply them at different levels of granularity.

We applied our multi-level editing approach to both XML and text documents. Text and XML documents conform to an ordered structure of elements and each element unit can be uniquely identified by its position in the sequence of children at any level. The transformational approach applied on multiple levels tries to maintain an order between elements in the face of concurrent operations of insertion and deletion so that the intentions of operations are satisfied. Graphical documents have a richer set of objects and operations, as compared to text and XML documents, and there is no order between the objects. Objects are not identified by their position, but by a unique identifier, and there is no need to adapt the identifiers due to concurrent operations. The group intention for pairs of concurrent operations has been differently defined than in the case of text and XML documents. For text documents the combined intention of concurrent operations was defined to adapt object positions, while for graphical documents it was defined by means of ordering constraints between operations. Preserving the group intention that we defined was not possible by using operational transformation.

To maintain consistency in object-based graphical documents, we proposed a novel operation serialisation algorithm based on the reordering of nodes in a graph. The nodes represent operations and the edges represent ordering constraints between operations. We have classified conflicts into real and resolvable, depending on whether an ordering of execution between pairs of operations can be established or not. The set of operations that we have implemented satisfies the requirements of architectural and product design as specified by members of Inovation Center Virtual

Reality (Institut für Werkzeugmaschinen und Fertigung) Institute at ETH Zurich. Our algorithm is the first one that deals with operations of grouping and ungrouping in the collaborative environment. In addition to complex operations of group/ungroup, we support concurrent operations targeting different pages and layers of objects. Moreover, we allow users to define the types of conflicts between the operations and the policy for the resolution of conflicts. In this way, conflict handling can be customised to suit the requirements of specific applications.

The same mechanisms for consistency maintenance have been used with both synchronous and asynchronous communication, i.e. operational transformation applied recursively to the history buffer distributed throughout the tree for text and XML documents, and the serialisation of operations based on constraints in the case of graphical documents.

We built collaborative systems relying on the algorithms proposed in this thesis, for text, XML and graphical documents, both for the real-time communication and the asynchronous collaboration relying on a shared repository.

7.2 Vision

My vision is the design of a *multi-synchronous editor* over hierarchical documents that supports the *real-time* and *asynchronous mode of communication*, but also a *combination of these two modes*. For instance, a part of users edit on real-time the shared documents, while another part of users work in isolation in asynchronous mode of collaboration. The multi-synchronous editor should allow each user to transparently switch from one mode of communication to another. Additionally, it could offer *other modes of collaboration*. For instance, a mode where a user works in isolation receiving in real-time the changes produced by other users which permits him/her to stay informed about the evolution of the document. Users can choose to integrate concurrent changes at a later time. Another mode of collaboration could allow users to choose which are the part of the changes they want to integrate and which are the changes they want to transmit to other users. This kind of system offers users the possibility to synchronise their changes without the constraint for the synchronisation against a central repository. The multi-synchronous editor should support collaboration over various types of documents such as *text*, *XML* and *graphical*. Users should be able to define rules for conflict based on the structure of the document and the system should detect conflicts, notify users about

conflicts and assist users to resolve the conflicts.

The most general architecture offering support for this way of collaboration is the peer-to-peer (P2P) network where users work on their replicas. As users can perform concurrent changes on the shared data, a very important issue on which I want to focus in my research work is how to maintain consistent replicas. Another important issue is how to maintain users aware about the activity of the other users in the group, especially regarding conflicting changes. In my future research work I also want to focus on awareness issues specific for the multi-synchronous environments.

Maintaining Consistency of Shared Data

As shown in this thesis, operational transformation was used both for the real-time and asynchronous communication. Therefore it is a good candidate for the multi-synchronous communication where users can interact in various ways and can switch from one mode of collaboration to the other.

We have seen that treeOPT is a general mechanism for maintaining consistency over text and XML documents and it recursively applies a linear operational transformation algorithm over the hierarchical structure. The tombstone transformational approach [80] is the only existing correct operational transformation algorithm working in peer-to-peer networks. The tombstone transformational approach was developed only for linear structures of the document and it could be extended for hierarchical structured documents by a combination with my approach. In this way collaboration could be offered for a large class of documents, as the tree model encompasses a wide range of documents and offers support for semantically structured documents. The combination between treeOPT and tombstone transformational approach would be the first algorithm for maintaining consistency over hierarchical documents in peer-to-peer networks. Moreover, as nodes are never deleted in the tree, concurrent conflicting changes performed on deleted parts of the document are not lost. This is a key advantage for managing conflicts over hierarchical documents.

Operational transformation mechanism ensures convergence by merging all concurrent changes. It assumes that it is always possible to merge operations while preserving their intentions. But, some applications have to deal with antagonist operations and in this case operational transformation mechanism is not sufficient. For instance, in the collaborative graphical editor application presented in this thesis, constraints between operations exist. Users can define conflicts between operations and specify cancellation or ordering constraints between operations in order to resolve conflicts.

For dealing with ordering constraints between operations, we proposed a distributed operational serialisation mechanism. As both operational transformation and serialisation approaches have advantages and disadvantages, another future work direction is the combination of the two approaches.

One of the future work directions would be to explore the possibility of performing undo operations in the multi-level editing approach. As a relation is maintained between a semantic unit and its history, it is easy to develop an undo mechanism targeting a particular semantic unit. For instance, a previous state of a semantic unit can be restored by undoing operations belonging to the histories of that node and its child nodes.

Multi-level granularity locking following some of the ideas presented in [83] can be implemented as an option to be used on top of the free editing mechanism offered by our text and XML real-time editors. This could allow users to lock document units to ensure exclusive access. For instance, in text editing, users could lock paragraphs or sentences, and during editing of XML documents users could lock any element. Graphical editing could be also extended with locking facilities to allow users to lock specific objects or select regions of the shared document to be locked.

Another future work direction is the formalisation of consistency maintenance approaches for collaborative editing and the proof of correctness of the approaches for maintaining consistency, for both text and graphical editing.

In this thesis we investigated consistency maintenance approaches for systems where all data was replicated. But, in large systems managing large amounts of data it is suitable to partially replicate data. For instance, in Wikipedia [5] data could be clustered according to the language and replicated on different nodes. Therefore, a future direction is the investigation of issues of consistency maintenance in environments with partial replication.

Providing Group Awareness

Maintaining consistency in peer-to-peer networks is similar to the issue of maintaining consistency in real-time editing with the exception of conflict handling [51]. In real-time editing, handling conflicts is not necessary as even if users perform concurrent conflicting changes, they can quickly react to resolve conflicts, for instance by performing an undo of the conflicting changes. But, when users are working in isolation data might diverge a lot and the recovery from an arbitrary resolution of conflicting changes is very difficult. In this thesis we proposed a mechanism for conflict handling

in the asynchronous communication with a central repository. Handling conflicts in this case is simplified. Conflicts are detected when a user performs an update of their local changes with the changes in the repository. Conflicts are resolved by a single user at a time. Only that user has the right to publish their changes to the repository. The other users should take into account the resolution of conflicts by updating their copies before having a right to publish their own changes. On the other side, in peer-to-peer networks, a conflict will be notified to several users and they might concurrently resolve it in different ways or they might postpone the resolution of that conflict.

Specifying when two concurrent changes are conflicting is a complex task. Even if a merging tool can merge all concurrent changes, it is difficult to say if the resulting document is sound from the user point of view. Two approaches for dealing with conflicting changes depending if the conflicts are represented or not can be investigated.

In the first approach I want to investigate if it would be enough to merge all concurrent changes and mark concurrent updated operations in the result document. After the merging process, users have to review the document and mark the status of the document as being sound. As an awareness mechanism, the other users are notified about the review status of the document.

This approach might not be satisfactory, and therefore it would be interesting to investigate a second approach where conflicts are explicitly represented. Consequently, a mechanism for maintaining consistency in the presence of concurrent conflicting operations needs to be devised. After conflicts are presented to users, the system has to offer the possibility to recover from conflicting situations and return to previous states of the document by means of an undo mechanism.

Applications of the multi-synchronous editor

A multi-synchronous editor would be useful for a community of users that want to collaborate over a set of documents where subcommunities have different needs or preferences at different points in time in terms of the mode of collaboration and have to switch working from one mode of collaboration to the other. One of the examples of its applications is in the case of the massive collaborative editing where a large number of users can edit a set of documents. An example of massive collaborative editing is Wikipedia, the most popular wiki system, a free online encyclopedia written collaboratively by volunteers. However, wikis are centralised systems

and the system does not scale well and in the case of many users editing the same wiki page parallel modifications done on the same page are not automatically merged. On the contrary, the multi-synchronous environment would allow that Wikipedia is deployed on a peer-to-peer network where scalability is ensured and data is replicated. In this case, Wikipedia will not be managed by a single organisation, but by the whole community, and therefore the power to censor and the cost of deploying are shared. Moreover, Wikipedia will benefit from the multi-synchronous way of working. For instance, a group of users would work more efficiently for editing some wiki pages if changes are seen in real-time and appropriate awareness information is provided to help users to prevent conflicts. Moreover, users would be able to synchronise changes among themselves before publishing the modifications to the whole community. This mode of direct synchronisation between users is not possible to be realised with the current wiki systems where users have to publish their changes to the shared repository before synchronising changes with other users.

Another application that can be extended on top of the multi-synchronous editor is the collaboration over both papers and digital documents. While certain activities, such as the authoring, distribution and archiving of documents tend to be performed electronically, paper continues to be a preferred medium for many reading and writing activities. Moreover, it is a very suitable medium for mobile environments, such as trains and planes or other out of the office places. Our group is involved in a number of projects investigating how new technologies can turn paper into an interactive medium. Handwriting and document mark-up can be digitally captured and links on paper can be selected to invoke digital services such as playing a video clip or retrieving an annotation [78]. We want to enhance the interactive paper with facilities for collaboration and develop an environment that supports collaboration over both papers and digital documents. Our group developed a general framework for interactive paper called iPaper [97] that offers support for links from paper documents to digital information and vice-versa. In addition to supporting real-time collaborative editing of digital documents, it will be possible for authors to work in isolation on either a paper or digital version and later merge their work with the work of others. Research questions to be addressed include the design and capture of the different forms of markup and gestures used on paper and transformation of these markups into editing operations on the digital document representation. Another question is how to synchronise document versions in a way that preserves consistency and user intentions, and how to deal with annotations, conflicts and editing replays

at the user interface level.

Other applications of the multi-user editor could be found in various other domains such as software engineering and e-learning. Existing distributed version control systems such as Darcs [2], GNU Arch [3] and Codeville [1] fail to obtain convergence in some special cases. Therefore, the approaches for maintaining consistency in the multi-synchronous environment could be applied for building a distributed version system that works correctly. Another application of the multi-synchronous editor is the support for distributed pair programming. The proposed multi-synchronous editor could be specialised for an object-oriented programming environment. The description of a project could be viewed as a hierarchical structure: a document contains a set of classes, each class contains a set of fields and methods and each method contains a set of lines of code. Constraints can be also defined for the object-oriented programming language. Moreover, the multi-synchronous editor could be used in e-learning to support collaborative writing or group project activities.

User studies

User studies can be performed not only to evaluate our systems, but also to analyse how users work in a collaborative environment, how they divide their work, how often they interfere with the parts edited by other users or how they solve conflicts. It especially is interesting to analyse how architects and product designers use a graphical editor in their work and if real-time collaborative editing eases substantially their current design practice.

A

Transformation Functions in SOCT2

In this section we present transformation functions used in collaborative text editing in SOCT2 approach [101, 102] and provide a counterexample showing that these functions can lead to the divergence of the copies of the shared document.

We present both *transpose_fd* and *transpose_ bk* transformations. These functions have to be defined for each pair of operations that can be performed, in our case, insert and delete operations.

Determining correct transformation functions is very difficult. Therefore, in this section we present the process of refinement used in finding correct transformation functions in SOCT2. The initial form of the forward transformation function *transpose_fd(*O_1*,*O_2*)* of transforming operation O_2 against O_1 such that O_2 includes the effect of O_1 is presented in what follows for all combinations of insert and delete operations. The first argument of operation *insert(p,c)* represents the position of character insertion and the second argument represents the character to be inserted. The argument of operation *delete(p)* represents the position of the character to be deleted.

```
Algorithm transpose_fd(insert(p1,c1),insert(p2,c2))
{
  if (p1 < p2) then
      return insert(p2+1,c2);
  else if (p1 > p2) then
           return insert(p2,c2);
       else if (p1 = p2)then
                if (c1 = c2) then
                    return id
                else
                    return insert(p2,c2);
}
Algorithm transpose_fd(delete(p1),insert(p2,c2))
{
  if (p1 < p2) then
      return insert(p2-1,c2);
  else
      return insert(p2,c2);
}
Algorithm transpose_fd(insert(p1,c1),delete(p2))
{
  if (p1 ≤ p2) then
      return delete(p2+1);
  else
      return delete(p2);
}
Algorithm transpose_fd(delete(p1),delete(p2))
{
  if (p1 < p2) then
      return delete(p2-1);
  else if (p1 > p2) then
           return delete(p2);
       else
           return id;
}
```

We present only the forward transformation function for transforming an *insert* operation against another *insert* operation, the other functions being similar. If the position of insertion of the second operation, i.e. the operation that has to be transformed, is to the right of the position of insertion of the first operation, i.e. the operation against which the transformation has to be performed, the position of insertion has to be incremented. Otherwise, the second operation remains unchanged. If the positions of insertion of the two operations are the same, the result of the forward transposed operation is a null operation denoted *id*. This corresponds to the case that the first operation has been executed and, therefore, the second operation, identical with the first one, must be ignored. If the

insertion positions of the two operations are the same, but the characters to be inserted are different, the result of the transformation is the original form of the operation.

The main problem with the above specification of forward transformation for the pair *insert/insert* occurs when the two operations insert different characters at the same position. Consider the scenario illustrated in Figure A.1, where operations O_1 and O_2 insert concurrently characters c_1 and c_2 at position p. When the above transformation functions are applied, the transformation of O_2 against O_1 is the initial form of O_2 and the transformation of O_1 against O_2 is the initial form of O_1. In this case the sites are inconsistent: at $Site_1$ character c_2 is inserted before character c_1, while at $Site_2$ character c_1 is inserted before character c_2. Therefore, the forward transformation function for the *insert/insert* pair of operations has to be redefined for two insert operations which insert different characters at the same position in the document.

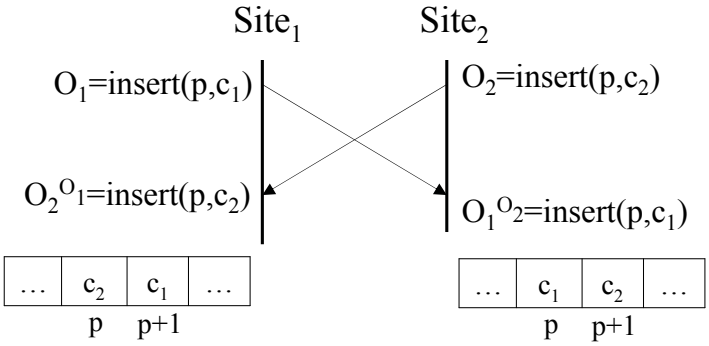


Figure A.1: Divergence due to two concurrent *insert* operations

A possible solution that seems to correct this problem is to associate a priority to each operation, according to the code of the character it inserts. In the case of a conflict, the character with the higher priority will be inserted before the one with the lower priority. The modified form of the forward transformation function is given below.

```
Algorithm transpose_fd(insert(p1,c1),insert(p2,c2))
{
  if (p1 < p2) then
      return insert(p2+1,c2);
  else if (p1 > p2) then
        return insert(p2,c2);
       else if (p1 = p2) then
              if (c1 = c2) then
                 return id
              else if (pr(c2)>pr(c1)) then
                      return insert(p2,c2)
                   else return insert(p2+1,c2);
}
```

The transformation functions defined previously satisfy condition $C1$, but they do not satisfy $C2$. Condition $C2$ has to be checked for every triplet of operations constructed from the set $\{insert, delete\}$. The functions presented above do not satisfy the $C2$ condition and it appears that the problem originates in the conflict between two insertion operations that insert characters at the same position. Consider the example illustrated in Figure A.2.

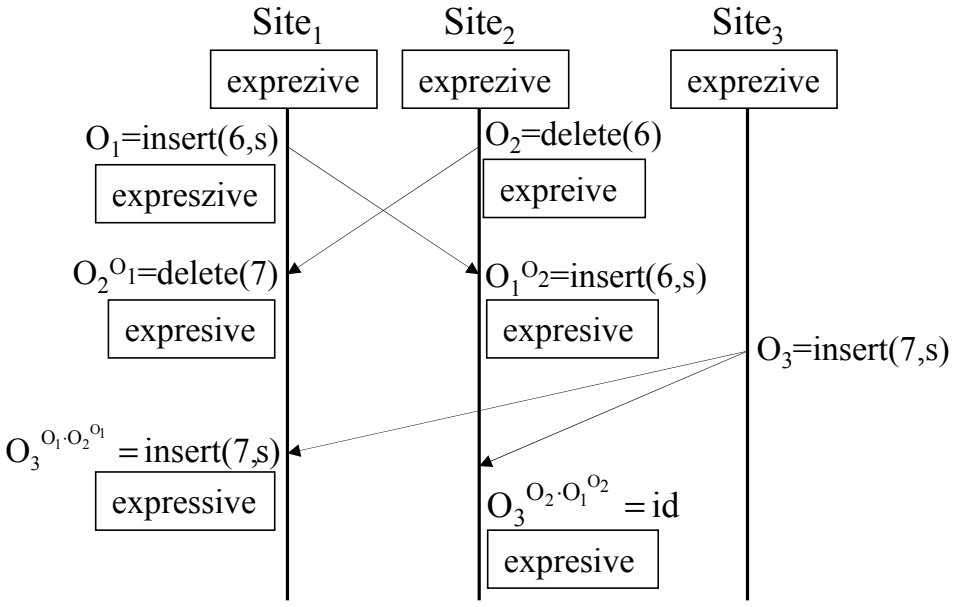


Figure A.2: Divergence due to incorrect forward transformation functions

Consider the initial state of the document is *"exprezive"* and three users are concurrently issuing some operations. The first user adds *"s"* before *"z"* by issuing operation O_1=*insert(6,"s")*. The second user deletes *"z"* by issuing operation O_2=*delete(6)*. Concurrently with the operations issued by $User_1$ and $User_2$, $User_3$ wants to add *"s"* before *"i"* by issuing operation O_3=*insert(7,"s")*. Further, at $Site_1$, after O_1 has been generated, the

remote operations are received in the order O_2 followed by O_3. At $Site_2$, after the generation of O_2, the remote operations are received in the order O_1 and O_3. Let us analyse in what follows the way operations are transformed when they arrive at $Site_1$ and $Site_2$. At $Site_1$, when O_2 arrives, it needs to be transformed against O_1, the result being $O_2^{O_1}$=*delete(7)*. When O_3 arrives at $Site_1$, it needs to be transformed against operations O_1 and O_2. Both O_1 and O_2 are concurrent with O_3. When O_3 is transformed against O_1, its execution form becomes $O_3^{O_1}$=*insert(8,"s")*. When the operation is further transformed against $O_2^{O_1}$, its execution form becomes $O_3^{O_1 \cdot O_2^{O_1}}$=*insert(7,"s")*. Therefore, the final state of the document at $Site_1$ is *"expressive"*. At $Site_2$, after the execution of O_2, when operation O_1 arrives at the site, it needs to be transformed against O_2, the result being $O_1^{O_2}$=*insert(6,"s")*. When operation O_3 arrives at $Site_2$, it is first transformed against O_2, the transformed form being $O_3^{O_2}$=*insert(6,"s")*. When the transformed form *insert(6,"s")* is further transformed against $O_1^{O_2}$=*insert(6,"s")*, it is cancelled as the two insert operations insert the same character at the same position. The final state of the document at $Site_2$ is therefore *"expresive"*. The document states at the $Site_1$ and $Site_2$ do not converge. The reason is that condition $C2$ has not been satisfied, i.e. $O_3^{O_1:O_2} \neq O_3^{O_2:O_1}$.

As previously mentioned, the case that needs special handling in the definition of the forward transformation functions is the case of two concurrent insertion operations inserting the same character at the same position in the document. The authors of SOCT2 algorithm detected that two cases can generate this problem: either two users have inserted two characters at the same position, or two users have inserted the two characters at two different positions and a third user has concurrently deleted characters between these positions. Therefore, in order to respect conditions C_1 and C_2, the authors modified the transformation functions by adding two additional parameters b and a to the insertion operation *insert(p,c,b,a)*. b and a denote operations that have concurrently deleted a character before and, respectively, after the inserted character c. In the case of two concurrent operations *insert(p,*c_1*,*a_1*,*b_1*)* and *insert(p,*c_2*,*a_2*,*b_2*)*, the following cases can be distinguished:

- if $b_1 \cap a_2 \neq \emptyset$, there exists at least one operation that deleted a character situated after the character c_2 and before character c_1. This means that at the point these two operations have been generated, the two positions were different with $p_2 < p_1$, where p_1 and p_2 are the initial positions of the two insert operations.

- if $a_1 \cap b_2 \neq \emptyset$, there exists at least one operation that deleted a character situated after character c_1 and before character c_2. This means that when the two operations were generated, the relation between their positions was $p_1 < p_2$.

- if $b_1 \cap a_2 = a_1 \cap b_2 = \emptyset$, c_1 and c_2 have been generated at the same position.

The modified forward transformation functions are given below:

Algorithm *transpose_fd(insert(p_1,c_1,a_1,b_1), insert(p_2,c_2,a_2,b_2))*
{
  if *($p_1 < p_2$)* then
    return *insert(p_2+1,c_2,a_2,b_2)*;
  else if *($p_1 > p_2$)* then
    return *insert(p_2,c_2,a_2,b_2)*;
  else if *($p_1 = p_2$)* then
    if *($b_1 \cap a_2 \neq \emptyset$)* then
      return *insert(p_2+1,c_2,a_2,b_2)*;
    else if *($a_1 \cap b_2 \neq \emptyset$)* then
      return *insert(p_2,c_2,a_2,b_2)*;
    else if *(code(c_2)>code(c_1))* then
      return *insert(p_2,c_2,a_2,b_2)*;
    else if *(code(c_2)<code(c_1))* then
      return *insert(p_2+1,c_2,a_2,b_2)*;
    else
      return *id(insert(p_2,c_2,a_2,b_2))*;
}

Algorithm *transpose_fd(delete(p_1),insert(p_2,c_2,a,b))*
{
  if ($p_1 < p_2$) then
    return *insert(p_2-1,c_2,a+delete(p_1),b)*;
  else
    return *insert(p_2,c_2,a,b+delete(p_1))*;
}

Algorithm *transpose_fd(insert(p_1,c_1,a,b),delete(p_2))*
{
  if *($p_1 \leq p_2$)*then
    return *delete(p_2+1)*;
  else
    return *delete(p_2)*;
}

Algorithm *transpose_fd(delete(p_1),delete(p_2))*
{
  if *($p_1 \leq p_2$)* then
    return *delete(p_2-1)*;
  else if *($p_1 \geq p_2$)* then
      return *delete(p_2)*;
    else
      return *id(delete(p_2))*;
}

The corresponding backward transposition functions can be deduced according to the definition of the backward transposition function given in section 2.3.9. The backward transposition functions are presented in what follows:

Algorithm *transpose_bk(insert(p_1,c_1,a_1,b_1),insert(p_2,c_2,a_2,b_2))*
{
  if *($p_1 < p_2$)* then
    return *(insert(p_2-1,c_2,a_2,b_2),insert(p_1,c_1,a_1,b_1))*;
  else
    return *(insert(p_2,c_2,a_2,b_2),insert(p_1+1,c_1,a_1,b_1))*;
}

Algorithm *transpose_bk(insert(p_1,c,a,b),delete(p_2))*
{
  if $p_1 < p_2$ then
    return *(delete(p_2-1),insert(p_1,c,a,b+delete(p_2-1)))*;
  else if *($p_1 > p_2$)*
      return *(delete(p_2),insert(p_2-1,c,a+delete(p_2),b))*;
    else
      return *('Error')*;
}

Algorithm *transpose_bk(delete(p_1),insert(p_2,c,a,b))*
{
  if *($p_1 < p_2$)* then
    return *(insert(p_2+1,c,a-delete(p_1),b),delete(p_1))*;
  else if *($p_1 > p_2$)*
      return *(insert(p_2,c,a,b-delete(p_1)),delete(p_1+1))*;
    else
      if *(delete(p_1) $\in$ a)* then
        return *(insert(p_2+1,c,a-delete(p_1),b),delete(p_1))*
      else
        return *(insert(p_2,c,a,b-delete(p_1)),delete(p_1+1))*
}

Algorithm *transpose_bk(delete(p_1),delete(p_2))*
{
  if *($p_1 \leq p_2$)* then
    return *(delete(p_2+1),delete(p_1))*;
  else
    return *(delete(p_2),delete(p_1-1))*;
}

The notion of identity operation *id(O)* is introduced denoting an operation that is ignored, i.e. an operation whose effect is null. The identity operation could have been obtained for instance by forward transposing a delete operation against an identical delete operation. The forward and backward transposition functions that we found in [100] and [116] seem either incorrect or incomplete. We have therefore added our own transformation functions regarding the identity operations:

$$
\begin{aligned}
&\textit{transpose\_fd(id}(O_1)\textit{,id}(O_2)\textit{)} = \textit{id(transpose\_fd}(O_1,O_2)\textit{)}, \forall O_1, O_2\\
&\textit{transpose\_fd(id}(O_1)\textit{,}O_2\textit{)} = O_2, \forall O, O_1, O_2 \text{ with } O_2 \neq \textit{id}(O)\\
&\textit{transpose\_fd}(O_1\textit{,id}(O_2)\textit{)} = \textit{id(transpose\_fd}(O_1,O_2)\textit{)},\\
&\qquad \forall O, O_1, O_2 \text{ with } O_1 \neq \textit{id}(O)
\end{aligned}
$$

$$
\begin{aligned}
&\textit{transpose\_bk(id}(O_1)\textit{,id}(O_2)\textit{)} = \textit{(id}(O_2')\textit{,id}(O_1')\textit{)}\\
&\textit{transpose\_bk(id}(O_1)\textit{,}O_2\textit{)} = \textit{(}O_2\textit{,id}(O_1')\textit{)}, \forall O, O_1, O_2 \text{ with } O_2 \neq \textit{id}(O)\\
&\textit{transpose\_bk}(O_1\textit{,id}(O_2)\textit{)} = \textit{(id}(O_2')\textit{,}O_1\textit{)}, \forall O, O_1, O_2 \text{ with } O_1 \neq \textit{id}(O) \text{ and } O_1 \neq O_2\\
&\textit{transpose\_bk}(O_1\textit{,id}(O_1)\textit{)} = \textit{(}O_1\textit{,id}(O_1)\textit{)}, \forall O, O_1 \text{ with } O_1 \neq \textit{id}(O)\\
&\text{where } \textit{transpose\_bk}(O_1,O_2) = (O_2',O_1')
\end{aligned}
$$

Our algorithms working for hierarchical structures of documents rely on an algorithm working for a linear structure of the document that is recursively applied in the tree. Our current implementations of real-time collaborative editing systems for text and XML documents use SOCT2 with the transformation functions extended as previously described.

However, even with these modifications SOCT2 transformation functions are not correct and therefore, in what follows we present a counterexample. This counterexample is a refinement of the counterexample presented in [79]. Consider the scenario in Figure A.3, with the initial state of the document *"123"* where three users execute concurrent operations as described in what follows.

At $Site_1$ a user deletes the third character *"3"* by issuing operation O_1=*delete(3)* and then inserts at the third position in the document the character *"x"* by issuing operation O_2=*insert(3,"x")*. Concurrently, at $Site_2$ a user inserts the character *"x"* at the third position in the document by issuing operation O_3=*insert(3,"x")* and at $Site_3$ a user inserts the character *"y"* as the fourth character in the document by issuing operation O_4=*insert(4,"y")*. We analyse how operations are executed at $Site_2$ and $Site_3$. Suppose that at $Site_2$, after O_3 has been generated remote operations are executed in the order O_4, O_1 and O_2. At $Site_3$, after O_4 has been generated, remote operations are executed in the order O_3, O_1 and O_2. Insertion operations include additional parameters b and a which initially are empty sets, i.e. O_2=*insert(3,"x",{},{})*, O_3=*insert(3,"x",{},{})* and O_4=*insert(4,"y",{},{})*.

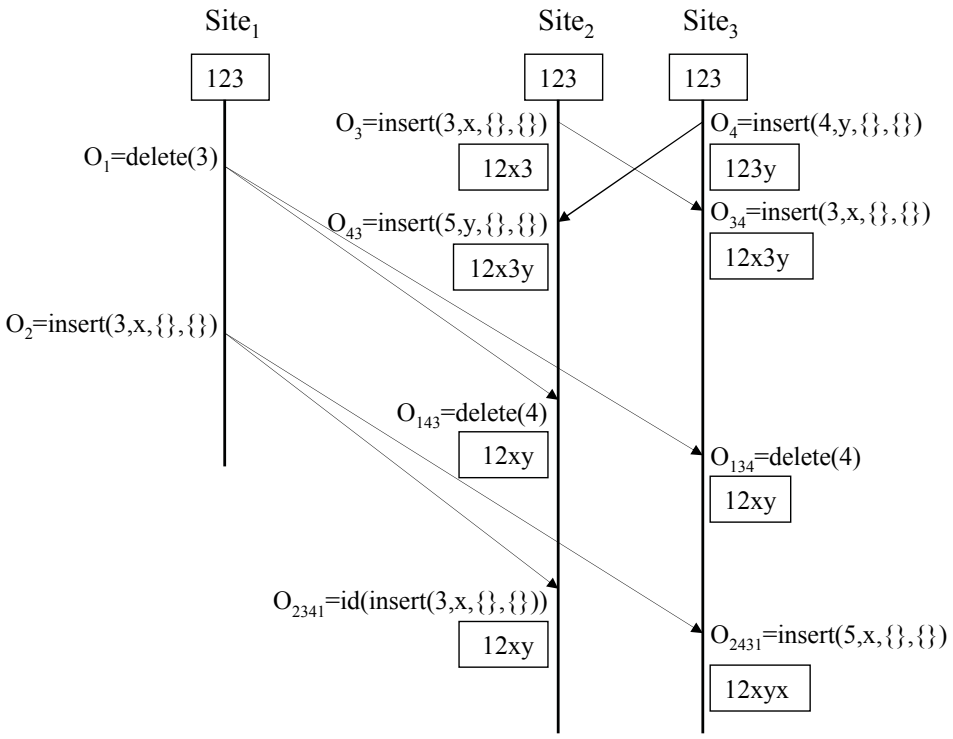


Figure A.3: Counterexample SOCT2

At $Site_2$, when O_4 arrives, it is transformed against the concurrent operation O_3, the transformed form being O_{43}=*insert(5,"y",{},{})*. When O_1 arrives, it needs to be transformed against the concurrent operations O_3 and O_{43}. The transformation of O_1 against O_3 is O_{13}=*delete(4)*. O_{13} transformed against O_{43} keeps the initial form of O_{13}, i.e. O_{143}=*delete(4)*. When O_2 arrives at $Site_2$ the history buffer contains the following operations HB=[O_3,O_{43},O_{143}]. The relations between the remote operation and the operations in the history buffer are the following: $O_2 \| O_3$,$O_2 \| O_{43}$ and $O_{143} \rightarrow O_2$. According to SOCT2, the history buffer has to be reordered so that the operations that precede O_2 are situated in the history buffer before the operations that are concurrent with O_2. Operation O_{143} has to be backward transposed towards the beginning of the history buffer. The first operation that is performed is *transpose_bk*(O_{43},O_{143})=*transpose_bk(insert(5, "y",{},{}),delete(4)) =(delete(4),insert(4,"y",{delete(4)},{}))* =[O'_{143},O'_{43}]. The next performed transformation is *transpose_bk*(O_3,O'_{143})=*transpose_bk(insert(3,"x",{},{}),delete(4))=(delete(3),insert(3,"x",{},{delete(3)}))* = [O''_{143}, O'_3]. As a result of the backward transposition, the history buffer becomes: [O''_{143}, O'_3, O'_{43}]. O_2 has to be transformed in turn with O'_3 and O'_{43}. O_2 transformed against O'_3 gives as a result the operation *id(insert(3, "x",{},{}))*. Further transforming this operation against O'_{43} returns the same form of the operation O_{2341}=*id(insert(3,"x",{},{}))*.

At $Site_3$, when O_3 arrives, it is transformed against the concurrent operation O_4, the transformed form of O_3 being O_{34}=*insert(3,"x",{},{})*. When O_1 arrives at the site, it needs to be transformed against the concurrent operations O_4 and O_{34}. The transformation of O_1 against O_4 is O_{14}=*delete(3)*. O_{14} transformed against O_{34} is O_{134}=*delete(4)*. When O_2 arrives at $Site_3$ the history buffer contains the following operations HB=[O_4,O_{34},O_{134}]. The relations between the remote operation O_2 and the operations belonging to HB are $O_2 \| O_4$,$O_2 \| O_{34}$ and $O_{134} \rightarrow O_2$. In SOCT2, the history buffer has to be reordered so that the operations that precede O_2 are situated in the buffer before the operations that are concurrent with O_2. Operation O_{134} has to be backward transposed towards the beginning of the history buffer. The first operation that is performed is *transpose_ bk*(O_{34},O_{134})=*transpose_ bk(insert(3,"x",{},{}),delete(4))=(delete(3),insert(3,"x",{},{delete(4)}))*=[O'_{134}, O'_{34}]. The next performed transformation is *transpose_ bk*(O_4,O'_{134})=*transpose_ bk(insert(4,"y",{},{}),delete(3))=(delete(3),insert(3,"y",{delete(3)},{}))*=[O''_{134},O'_4]. Therefore, the history buffer becomes [O''_{134},O'_4,O'_{34}]. O_2 has to be transformed in turn with O'_4 and O'_{34}. O_2 transformed against O'_4 gives as result the operation *insert(4,"x",{},{})*, since the code of *"x"* is smaller than the code of *"y"*. Further transforming this operation against O'_{34} has as result the operation O_{2431}=*insert(5,"x",{},{})*.

Therefore, inconsistency is obtained at the sites $Site_1$ and $Site_2$, as the outcome at $Site_2$ is *"12xy"* and the outcome at $Site_3$ is *"12xyx"*. A similar counterexample to the one presented above has been presented in [81].

Even though special cases that generate inconsistencies are present in SOCT2 transformation functions, we used SOCT2 algorithm as basis in the implementation of our approach working for the hierarchical structures. SOCT2 is a simple and easily understandable approach, its control algorithm being used by almost all linear operational transformation approaches, as detailed in chapter 2. However, the implementation of our systems is modular and the basic linear OT algorithm on which our approach is based can be easily replaced with a different algorithm in the future.

B
Transformation Functions in GOT/GOTO

In this section we present the transformation functions used by the GOT and GOTO algorithms.

In the GOT [108] and GOTO [108], transformation functions are defined for strings, with the primitive operations being *insert(S,P)*, denoting that string S has to be inserted at position P, and *delete(N,P)*, denoting that N characters have to be deleted starting from position P. Transformation functions used in GOT algorithm are presented in what follows. In order to facilitate the description of the transformation functions, the following notations have been introduced: for an insert O, $S(O)$ denotes the string to be inserted by an insert operation O, $P(O)$ denotes the position parameter of operation O and $L(O)$ denotes the length of the string to be inserted or deleted by O.

The convention adopted in the transformation functions in GOT and GOTO is that if multiple concurrent delete operations have overlapping delete ranges, the combined effect for the deletion is the union of the individual delete ranges. If multiple concurrent insert operations insert strings at the same position, all strings appear in the document as if they were inserted in some total order.

In what follows we present the inclusion transformation functions. The first argument of the transformation functions represents the operation that has to be transformed, while the second argument represents the operation

against which the transformation is performed.

Algorithm $IT\_II(O_a, O_b)$
{
if $(P(O_a) < P(O_b))$ $O'_a := O_a$;
else $O'_a := insert[S(O_a), P(O_a) + L(O_b)]$;
return O'_a;
}

Algorithm $IT\_ID(O_a, O_b)$
{
if $(P(O_a) \leq P(O_b))$ $O'_a := O_a$;
else if $(P(O_a) > P(O_b) + L(O_b))$ $O'_a := insert[S(O_a), P(O_a) - L(O_b)]$;
else $O'_a := insert[S(O_a), P(O_b)]$; $Save\_LI(O'_a, O_a, O_b)$;
return O'_a;
}

Algorithm $IT\_DI(O_a, O_b)$
{
if $(P(O_b) \geq P(O_a) + L(O_a))$ $O'_a := O_a$;
else if $(P(O_a) \geq P(O_b) + L(O_b))$ $O'_a := delete[L(O_a), P(O_a) + L(O_b)]$;
else $O'_a := delete[P(O_b) - P(O_a), P(O_a)] \oplus$
$delete[L(O_a) - (P(O_b) - P(O_a)), P(O_b) + L(O_b)]$;
return O'_a;
}

Algorithm $IT\_DD(O_a, O_b)$
{
if $(P(O_b) \geq P(O_a) + L(O_a))$ $O'_a := O_a$;
else if $(P(O_a) \geq P(O_b) + L(O_b))$ $O'_a := delete[L(O_a), P(O_a) - L(O_b)]$;
else
if $((P(O_b) \geq P(O_a))$ and $(P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b)))$
$O'_a := delete[0, P(O_a)]$;
else if $((P(O_b) \leq P(O_a))$ and $(P(O_a) + L(O_a)) > (P(O_b) + L(O_b)))$
$O'_a := delete[P(O_a) + L(O_a) - (P(O_b) + L(O_b)), P(O_b)]$;
else if $((P(O_b) > P(O_a))$ and $((P(O_b) + L(O_b)) > (P(O_a) + L(O_a))))$
$O'_a := delete[P(O_b) - P(O_a), P(O_a)]$;
else $delete[L(O_a) - L(O_b), P(O_a)]$;
$Save\_LI(O'_a, O_a, O_b)$;
return O'_a;
}

In order to explain the $Save\_LI$ procedure we first analyse the inclusion transformation function of an *insert* operation against a *delete* operation, $IT\_ID(O_a, O_b)$. If $P(O_a) \leq P(O_b)$, then O_a refers to a position situated to the left or at the position referred to by O_b, so the execution of O_b should not have any impact on the position of insertion of O_a. Therefore,

the transformed form of O_a is the same as the initial form of O_a. In the case of $P(O_a) > P(O_b) + L(O_b)$, i.e. the position of O_a is situated to the right of the right-most position in the range for deletion of O_b, the position of O_a has to be shifted by $L(O_b)$ characters to the left. Otherwise, it is the case that the position of O_a is in the range for deletion of O_b. The new insertion position of O_a is then $P(O_b)$. In this case, the information about the offset from $P(O_b)$ to $P(O_a)$ is lost. According to the reversibility requirement, the inclusion and exclusion functions should be reversible, i.e. if $IT(O_a, O_b)$ produces O'_a, then $ET(O'_a, O_b)$ should return as result O_a. In order to perform the exclusion transformation of the result obtained against the delete operation O_b, the information about the offset from $P(O_b)$ to $P(O_a)$ is lost and cannot be recovered by using only $P(O_b)$ in the later exclusion transformation. Therefore, the utility routine *Save_ LI(O'_a,O_a,O_b)* is used to save the lost information, i.e. the parameters of O_a before the translation and the reference to O_b, into an internal data structure associated with O'_a, which will be used by the exclusion transformation function to recover O_a.

When a delete operation O_a is transformed against an insert operation O_b and the position for insertion of O_b falls in the range of O_a, O_a should not delete any characters inserted by O_b. The range for deletion is then split into two segments as illustrated in function $IT\_DI(O_a, O_b)$.

The exclusion transformation functions are presented in what follows.

Algorithm $ET\_II(O_a, O_b)$

{
  if $(P(O_a) \leq P(O_b))$ $O'_a := O_a$;
  else if $(P(O_a) \geq (P(O_b) + L(O_b)))$ $O'_a := insert[S(O_a), P(O_a) - L(O_b)]$;
  else $O'_a := insert[S(O_a), P(O_a) - P(O_b)]$; $Save\_RA(O'_a, O_a, O_b)$;
  return O'_a;
}

Algorithm $ET\_ID(O_a, O_b)$

{
  if $(Check\_LI(O_a, O_b))$ $O'_a := Recover\_LI(O_a)$;
  else if $(P(O_a) \leq P(O_b))$ $O'_a := O_a$;
  else $O'_a := insert[S(O_a), P(O_a) + L(O_b)]$;
  return O'_a;
}

Algorithm $ET\_DI(O_a, O_b)$

{
if $((P(O_a) + L(O_a)) \leq P(O_b))$ $O'_a := O_a$;
else if $(P(O_a) \geq (P(O_b) + L(O_b)))$ $O'_a := delete[L(O_a), P(O_a) - L(O_b)]$;
else
if $((P(O_b) \geq P(O_a))$ and $((P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b))))$
$O'_a := delete[L(O_a), P(O_a) - P(O_b)]$;
else if $(((P(O_b) \leq P(O_a))$ and $((P(O_a) + L(O_a)) > (P(O_b) + L(O_b))))$
$O'_a := delete[P(O_b) + L(O_b) - P(O_a), (P(O_a) - P(O_b))] \oplus$
$delete[(P(O_a) + L(O_a)) - (P(O_b) + L(O_b)), P(O_b)]$;
else if $((P(O_a) > P(O_b))$ and $((P(O_b) + L(O_b)) \leq (P(O_a) + L(O_a))))$
$O'_a := delete[L(O_b), 0] \oplus delete[L(O_a) - L(O_b), P(O_a)]$;
else $delete[P(O_a) + L(O_a) - P(O_b), 0] \oplus delete[P(O_b) - P(O_a), P(O_a)]$;
$Save\_RA(O'_a, O_b)$;
return O'_a;
}

Algorithm $ET\_DD(O_a, O_b)$
{
if $(Check\_LI(O_a, O_b))$ $O'_a := Recover\_LI(O_a)$;
else if $(P(O_b) \geq (P(O_a) + L(O_a)))O'_a := O_a$;
else if $P(O_a) \geq (P(O_b)$ $O'_a := delete[L(O_a), P(O_a) + L(O_b)]$;
else $O'_a := delete[P(O_b) - P(O_a), P(O_a)] \oplus$
$delete[L(O_a) - (P(O_b) - P(O_a)), P(O_b) + L(O_b)]$;
return O'_a;
}

In the exclusion transformation of an *insert* O_a against another *insert* O_b, in $ET\_II(O_a, O_b)$, if $P(O_a) \leq P(O_b)$, O_a refers to a position to the left of the position referred to by O_b. In this case the undoing of the effect of O_b does not have an impact on the intended position of O_a, and O_a remains unmodified. If $P(O_a) \geq P(O_b) + L(O_b)$, meaning that the position of O_a is to the right of the rightmost position in the inserting range of O_b, then the position of O_a has to be shifted by $L(O_b)$ characters to the left. Therefore, the position parameter of O_a is decremented by $L(O_b)$. Otherwise, it is the case that the intended position of insertion of O_a falls in the middle of the string inserted by O_b. The exclusion transformation cannot be performed in this case since undoing O_b results in an undefined range for O_a. To deal with such cases, a relative addressing scheme has been proposed to save the position parameter of O'_a addressed with respect to the base operation O_b rather than to the document. To do that, the procedure *Save_RA(*O'_a*,*O_b*)* is called. Relatively addressed operations are converted into absolutely addressed operations by the adapted transformation functions working on lists.

In the above presented exclusion transformation functions the rou-

tine *Check_ LI(O_a,O_b)* was used to check whether O_b was involved in an information-losing inclusion transformation which resulted in O_a. If it is the case, *Recover_ LI(O_a)* recovers O'_a from the information saved in O_a. As in the case of the inclusion transformation functions, operation splitting can occur also for the exclusion transformation functions.

In order to cope with the splitting of operations, the inclusion and exclusion transformation functions have to be defined for the list of operations. The preconditions of applying the transformation functions between the list of operations are an extension of the precondition of the transformation functions applied for single operations [108]. The revised form of the inclusion transformation function proposed in [108] is presented in what follows.

Algorithm $LIT(OL_1, OL_2)$
{
  if ($OL_1 = []$) $RL := []$;
  else
    $TL_1 := LIT_1(OL_1[1], OL_2)$;
    $TL_2 := LIT(Tail(OL_1), OL_2 + TL_1)$;
    $RL := TL_1 + TL_2$;
  return RL;
}

Algorithm $LIT_1(O, OL)$
{
  if ($OL = []$) $RL := [O]$;
  else if ($Check\_RA(O)$ and not($Check\_BO(O, OL[1])$))
    $RL := LIT_1(O, Tail(OL))$
  else if ($Check\_RA(O)$ and ($Check\_BO(O, OL[1])$))
    $O' := Convert\_AA(O, OL[1])$;
    $RL := LIT_1(O', Tail(OL))$;
  else
    $RL := LIT(IT(O, OL[1]), Tail(OL))$;
  return RL;
}

The precondition of performing $O'_a = IT(O_a, O_b)$ is that O_a and O_b have the same generation context, i.e. $O_a \sqcup O_b$ and the postcondition is that the state resulted after the execution of O'_a is the generation context of O_b, denoted as $O_b \mapsto O'_a$. The preconditions for the input parameters of $LIT(OL_1, OL_2)$ are the following:

- for any operation O in OL_1, it must be that either $O \sqcup OL_2[1]$ or O is relatively addressed and O's base operation is in OL_2

- for any two consecutive operations $OL_2[i]$ and $OL_2[i+1]$, $OL_2[i] \mapsto OL_2[i+1]$.

The postcondition for the list OL'_1 representing the output of the LIT function is that $OL_2[l_2] \mapsto OL'_1[1] \mapsto OL'_1[2] \mapsto \cdots \mapsto OL'_1[l_1]$, where $l_2 = |OL_2|$ and $l_1 = |OL'_1|$.

To apply the inclusion transformation to the list of operations in OL_1 against the list of operations in OL_2, $OL_1[1]$ is first applied with the inclusion transformation against all operations in OL_2 to produce TL_1. Then, a recursive call of the $LIT()$ function is applied to the rest of the list OL_1 and $OL_2 + TL_1$ to produce TL_2. TL_1 must be appended to OL_2 to satisfy the postcondition of $LIT()$.

In LIT_1, the basic strategy is that if O is a relatively addressed operation, it is converted into an absolutely addressed operation by taking into account the information available in the base operation. The absolutely addressed operation is then transformed against those operations that are situated after the base operation in OL. The routine *Check_RA(O)* checks whether the operation O is relatively addressed, the routine *Check_BO(O_1,O_2)* checks whether O_1 has as base operation the operation O_2 and taking into account that O_2 is the base operation of O_1, the routine *Convert_AA(O_1,O_2)* converts O_1 into an absolutely addressed operation.

In order to cope with the split operation, the exclusion transformation has to be adapted and defined for two lists of operations. The revised form of the exclusion transformation function proposed in [108] is presented in what follows.

Algorithm $LET(OL_1, OL_2)$
{
  if $(OL_1 = [])$ $RL := []$;
  else
    $TL_1 := LET_1(OL_1[1], OL_2)$;
    $TL_2 := LET(Tail(OL_1), OL_2)$;
    $RL := TL_1 + TL_2$;
  return RL;
}

Algorithm $LET_1(O, OL)$
{
  if ($Check\_RA(O)$ or $OL = []$) $RL := [O]$;
  else $RL := LET(ET(O, OL[1]), Tail(OL))$;
  return RL;
}

The precondition of performing $O'_a = ET(O_a, O_b)$ is that the state resulting from the execution of O_b is the generation context of O_a, denoted as $O_b \mapsto O_a$. The postcondition of performing the exclusion transformation is that the context of O'_a and O_b are the same, i.e. $O_b \sqcup O'_a$. The preconditions for the input parameters of $LET(OL_1, OL_2)$ are the following:

- for any operation O in OL_1, either $OL_2[1] \rightarrow O$ or O is relatively addressed.
- for any two operations $OL_2[i]$ and $OL_2[i+1]$, $OL_2[i+1] \mapsto OL_2[i]$

The postcondition for the list OL'_1, the result of the exclusion transformation $LET(OL_1, OL_2)$ is that for any operation O in OL'_1, either $O \sqcup OL_2[len]$, where $len = |OL_2|$ or O is relatively addressed. The implementation of $LET()$ is very similar to the $LIT()$ function and therefore we do not provide further explanations. What we mention here is that, in contrast to $LIT()$, the outcome of $TL_1 = LET_1(OL_1[1], OL_2)$ is not appended to OL_2 in the recursive call of $LET()$ in order to satisfy the postcondition of LET.

In GOTO the transformation functions have to satisfy the C1 and C2 conditions presented in section 2.3.9. However, as shown in [52], the transformation functions do not satisfy condition C2. In what follows we present the counterexample for the GOTO transformation functions presented in [52].

Consider the scenario in Figure B.1. Three users concurrently edit the shared document consisting of the string *"abc"*. The first user inserts character *"x"* on position 2 of the string in order to obtain *"axbc"*. The second user deletes the second character in order to obtain *"ac"*. The third user inserts character *"y"* on position 3 of the string in order to obtain *"abyc"*. The operations arrive at the three sites in the order indicated in the scenario. Let us analyse the states obtained at $Site_2$ and $Site_3$. At $Site_2$, when O_3 arrives, it is transformed against operation O_2, the result being O'_3=*insert(2,"y")*. Additionally, the procedure *Save_LI(*O'_3*,*O_3*,*O_2*)* is called in order to recover from the situation when O'_3 has to exclude O_2 from its context. The state obtained after the execution of O_3 is *"ayc"*. When O_1 arrives at $Site_2$, it has to be transformed against O_2 and O'_3. The result of the transformation is $O'_1 = IT(O_1, [O_2, O'_3]) = IT(IT(O_1, O_2), O'_3) = IT($*insert(2,"x")*$,$ *insert(2,"y")*$) =$ *insert(3,"x")*. The final state obtained at $Site_2$ is *"ayxc"*. At $Site_3$, when O_2 arrives, it has to be transformed against O_3, the result being operation O'_2=*delete(2)*. The state obtained after the execution of O_2 is *"ayc"*. When operation O_1 arrives the site, it has to be

transformed against O_3 and O'_2. The result of the transformation is $O''_1 = IT(O_1, [O_3, O'_2]) = IT(IT(O_1, O_3), O'_2) = IT(insert(2,\text{“}x\text{”}), delete(2)) = insert(2,\text{“}x\text{”})$. The final state obtained at $Site_3$ is *“axyc”*. The states obtained at the two sites $Site_2$ and $Site_3$ are not identical and therefore inconsistency is obtained.

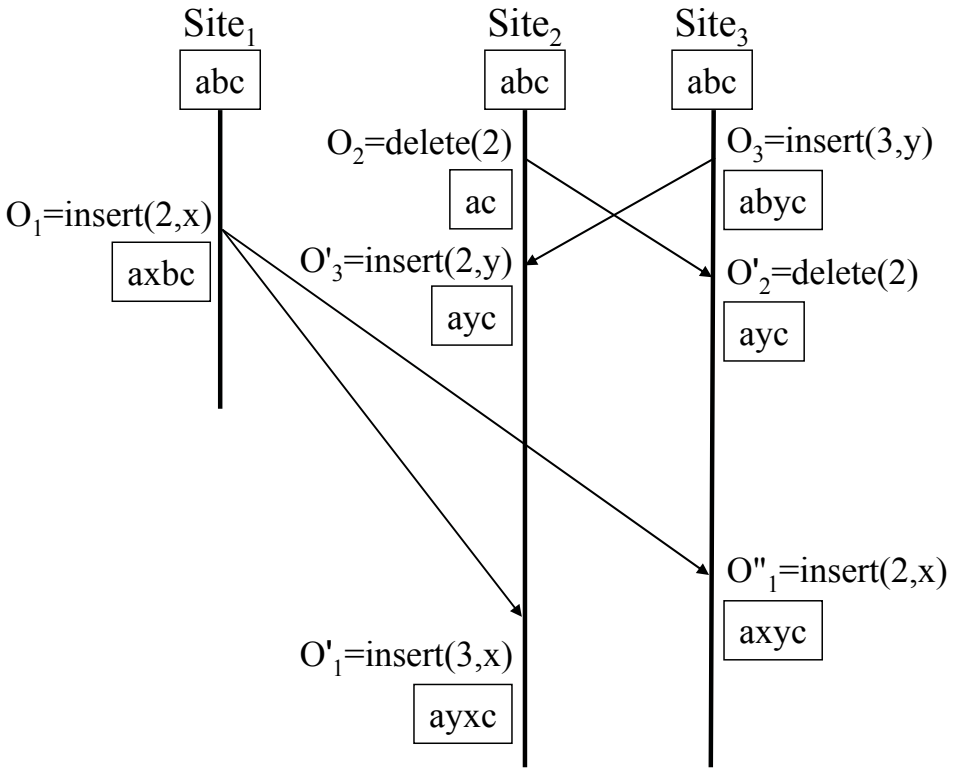


Figure B.1: Counterexample GOTO

Bibliography

[1] Codeville. http://codeville.org/.

[2] Darcs. *Distributed. Interactive. Smart.* http://darcs.net/.

[3] GNU Arch. http://www.gnu.org/software/gnu-arch/.

[4] Stylus studio. http://www.stylusstudio.com/.

[5] Wikipedia.http://www.wikipedia.org/.

[6] XMLSpy. http://www.altova.com/products_ide.html.

[7] Larry Allen, Gary Fernandez, Kenneth Kane, David B. Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. *Lecture Notes in Computer Science*, 1005:194—214, 1995.

[8] Ronald M. Baecker, Dimitrios Nastos, Ilona R. Posner, and Kelly L. Mawby. *The user-centred iterative design of collaborative writing software*, page 775—782. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1995.

[9] Pierfrancesco Bellini, Paolo Nesi, and Marius B. Spinu. Cooperative visual manipulation of music notation. *ACM Transactions on Computer-Human Interaction*, 9(3):194—237, 2002.

[10] Bogdan Berce. Generic collaborative editing. Diploma project, Institute for Information Systems, ETH Zurich, 2005.

[11] Thomas Berlage and Andreas Genau. A framework for shared applications with a replicated architecture. In *Proceedings of the 6th annual ACM symposium on User interface software and technology (UIST'93)*, page 249—257, Atlanta, Georgia, USA, 1993. ACM Press.

[12] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference*, page 341–352, Washington, District of Columbia, USA, January 1990.

[13] Tim Berners-Lee. *Weaving the Web.* Texere Publishing Ltd., November 1999.

[14] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[15] Christophe Bobineau. *Gestion de transactions en environnement mobile.* PhD thesis, Université de Versailles - Saint Quentin en Yvelines, 2002.

[16] Ernest Chang, Richard Kasperski, and Tony Copping. *Group coordination in participant systems.* Department of Advanced Computing and Engineering, Alberta Research Council, Calgary, Alberta, Canada, 1987. Unpublished.

[17] Gregory Cobena, Talel Abdessalem, and Yassine Hinnach. A comparative study of XML diff tools. Technical report, 2002.

[18] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 41–52, San Jose, California, USA, 2002.

[19] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version control with Subversion.* O'Reilly & Associates, Inc., 2004.

[20] Lorant Zeno Csaszar. Real-time collaborative graphical editor. Diploma project, Institute for Information Systems, ETH Zurich, 2003.

[21] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work (CSCW'02)*, page 58–67, New Orleans, Louisiana, USA, 2002. ACM Press.

[22] Paul Dourish. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. In

Proceedings of the 1996 ACM conference on Computer supported cooperative work (CSCW'96), page 268–277, Boston, Massachusetts, USA, 1996. ACM Press.

[23] Marco Dubacher. Gleichzeitiges, datenbankbasiertes bearbeiten von textdokumenten. Master's thesis, University of Zurich, Switzerland, 2003.

[24] Bogdan Dumitriu. Asynchronous collaborative text editing. Diploma project, Institute for Information Systems, ETH Zurich, 2004.

[25] Lisa Dusseault. *WebDAV: Next-Generation Collaborative Web Authoring.* Prentice Hall PTR, 2003.

[26] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):399–407, June 1989.

[27] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, 1991.

[28] Farallon. *Timbuktu user's guide.* Farallon Computing Inc., Berkely, California, 1988. User's manual.

[29] Robin La Fontaine. A delta format for XML: Identifying changes in XML files and representing the changes in XML. In *Proceedings of XML Europe*, Berlin, Germany, 2001.

[30] Jose Joaquin Garcia-Luna-Aceves, Earl J. Craighill, and Ruth Lang. An open-systems model for computer-supported collaboration. In *Proceedings of the 2nd IEEE Conference of Computer Workstations*, page 40–51, Santa Clara, 1988.

[31] Saul Greenberg. Sharing views and interactions with single-user applications. In *Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, page 227–237, Cambridge, Massachusetts, USA, 1990. ACM Press.

[32] Saul Greenberg. Personalizable groupware: Accomodating individual roles and group differences. In *Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW'91)*, page 17–32, Amsterdam, Netherlands, September 1991.

[33] Saul Greenberg, Mark Roseman, Dave Webster, and Ralph Bohnet. Human and technical factors of distributed group drawing tools. *Interacting with Computers*, 4(3):364—392, 1992.

[34] Saul Greenberg, Mark Roseman, David Webster, and Ralph Bohnet. Issues and experiences designing and implementing two group drawing tools. In *Proceedings of Hawaii International Conference on System Sciences (HICSS'92)*, volume 4, page 138—150. IEEE Press, January 1992.

[35] Thomas B. Hodel, Marco Dubacher, and Klaus R. Dittrich. Using database management systems for collaborative text editing. In *Proceedings of the Fifth International Workshop on Collaborative Editing, ECSCW'03*, Helsinki, Finland, 2003.

[36] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, page 133—145, San Diego, California, USA, January 1988. ACM Press.

[37] Claudia-Lavinia Ignat and Moira C. Norrie. Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems. *Fourth International Workshop on Collaborative Editing, CSCW 2002, IEEE Distributed Systems online*, 2002.

[38] Claudia-Lavinia Ignat and Moira C. Norrie. Customizable collaborative editor relying on treeopt algorithm. In *Proceedings of the 8th European Conference on Computer-supported Cooperative Work (ECSCW'03)*, page 315—334. Kluwer Academic Publishers, 2003.

[39] Claudia-Lavinia Ignat and Moira C. Norrie. Grouping/ungrouping in graphical collaborative editing systems. *Fifth International Workshop on Collaborative Editing, ECSCW'03, IEEE Distributed Systems online*, 2003.

[40] Claudia-Lavinia Ignat and Moira C. Norrie. Codoc: Multi-mode collaboration over documents. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, page 580—594, Riga, Latvia, 2004. Springer.

[41] Claudia-Lavinia Ignat and Moira C. Norrie. Extending real-time editing systems with asynchronous communication. In *Proceedings of*

the 8th International Conference on CSCW in Design (CSCWD'04), page 528—533, Xiamen, P.R.China, 2004. IEEE Press.

[42] Claudia-Lavinia Ignat and Moira C. Norrie. Grouping in collaborative graphical editors. In *Proceedings of the International Conference on Computer Supported Cooperative Work (CSCW'04)*, page 447—456, Chicago, Illinois, USA, 2004. ACM Press.

[43] Claudia-Lavinia Ignat and Moira C. Norrie. Operation-based versus state-based merging in asynchronous graphical collaborative editing. *Sixth International Workshop on Collaborative Editing, CSCW'04, IEEE Distributed Systems online*, 2004.

[44] Claudia-Lavinia Ignat and Moira C. Norrie. Flexible merging of hierarchical documents. *Seventh International Workshop on Collaborative Editing, GROUP'05, IEEE Distributed Systems online*, 2005.

[45] Claudia-Lavinia Ignat and Moira C. Norrie. Operation-based merging of hierarchical documents. In *Proceedings of the CAiSE'05 Forum, 17th International Conference on Advanced Information Systems Engineering*, page 101—106, Porto, Portugal, 2005.

[46] Claudia-Lavinia Ignat and Moira C. Norrie. Customisable collaborative editing supporting the work processes of organisations. *Computers in Industry*, 57(8-9):758—767, December 2006.

[47] Claudia-Lavinia Ignat and Moira C. Norrie. Draw-together: Graphical editor for collaborative drawing. In *Proceedings of the International Conference on Computer Supported Cooperative Work (CSCW'06)*, pages 269–278, Banff, Alberta, Canada, November 2006.

[48] Claudia-Lavinia Ignat and Moira C. Norrie. Flexible collaboration over xml documents. In *Proceedings of the International Conference on Cooperative Design, Visualization and Engineering (CDVE'06)*, pages 267–274, Mallorca, Spain, September 2006.

[49] Claudia-Lavinia Ignat and Moira C. Norrie. Flexible definition and resolution of conflicts through multi-level editing. In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'06)*, Georgia, Atlanta, USA, November 2006.

[50] Claudia-Lavinia Ignat and Moira C. Norrie. Supporting customised collaboration over shared document repositories. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, Luxembourg, Grand-Duchy of Louxembourg, June 2006.

[51] Claudia-Lavinia Ignat, Moira C. Norrie, and Gérald Oster. Handling conflicts through multi-level editing in peer-to-peer environments. *Eighth International Workshop on Collaborative Editing, CSCW'06, IEEE Distributed Systems online*, November 2006.

[52] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW'03)*, page 277—293, Helsinki, Finland, September 2003. Kluwer Academic Publishers.

[53] Mihail Ionescu and Ivan Marsic. Tree-based concurrency control in distributed groupware. *Computer Supported Cooperative Work*, 12(3):329—350, 2003.

[54] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404—425, July 1985.

[55] Rushed Kanawati. LICRA: A replicated-data management algorithm for distributed synchronous groupware applications. *Parallel Computing*, 22(13):1733—1746, 1996.

[56] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, page 195—202, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[57] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC'01)*, page 210—218, Newport, Rhode Island, USA, 2001. ACM Press.

[58] Michael Knister and Atul Prakash. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems*, 6(2):135—166, 1993.

[59] Jozef Paul Chris Lauwers. *Collaboration transparency in desktop teleconferencing environments*. PhD thesis, 1990.

[60] Christoph Lenggenhager. Analysis of SOCT algorithms as a basis for the treeOPT algorithm. Semester project, Institute for Information Systems, ETH Zurich, 2005.

[61] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, 2001.

[62] John R. Levine, Tony Mason, and Doug Brown. `lex` & `yacc`. O'Reilly & Associates, Inc., 1992.

[63] Du Li and Rui Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work (CSCW '04)*, page 457–466, Chicago, Illinois, USA, 2004. ACM Press.

[64] Tie Liao. Light-weight reliable multicast protocol. Technical report, INRIA, France, 1998.

[65] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, California, USA, March 1996.

[66] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, page 78–87, Tyson's Corner, Virginia, USA, 1992. ACM Press.

[67] Marilyn Mantei. Capturing the capture concepts: a case study in the design of computer-supported meeting environments. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work (CSCW'88)*, page 257–270, Portland, Oregon, USA, 1988. ACM Press.

[68] Pascal Molli, Gérald Oster, Hala Skaf-Molli, and Abdessamad Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work (GROUP'03)*, page 212–220, Sanibel Island, Florida, USA, November 2003. ACM Press.

[69] Pascal Molli, Hala Skaf-Molli, Gérald Oster, and Sébastien Jourdain. SAMS: Synchronous, asynchronous, multi-synchronous environments. In *Proceedings of the Conference on Computer-Supported Cooperative Work in Design (CSCWD'02)*, page 80—85, Rio de Janeiro, Brazil, September 2002.

[70] Thomas P. Moran, Kim McCall, Bill van Melle, Elin Pedersen, and Frank Halasz. Some design principles for sharing in tivoli, a whiteboard meeting-support tool. *Groupware for Real-Time Drawings: A designer's Guide*, page 24—36, 1995.

[71] Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work (CSCW '94)*, page 231—242, Chapel Hill, North Carolina, USA, 1994. ACM Press.

[72] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251—266, 1986.

[73] Sergiu Nedevschi. Concurrency control in real-time collaborative editing systems. Diploma project, Institute for Information Systems, ETH Zurich, 2002.

[74] Christine M. Neuwirth, Ravinder Chandhok, David S. Kaufer, Paul Erion, James Morris, and Dale Miller. Flexible diff-ing in a collaborative writing system. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Collaborative writing, page 147—154, Toronto, Canada, 1992. ACM Press.

[75] Richard E. Newman-Wolfe and Harsha K. Pelimuhandiram. MACE: a fine grained concurrent editor. In *Conference proceedings on Organizational computing systems (COCS'91)*, page 240—254, Atlanta, Georgia, USA, 1991. ACM Press.

[76] Richard E. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit locking in the ensemble concurrent object-oriented graphics editor. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work (CSCW '92)*, page 265—272, New York, NY, USA, 1992. ACM Press.

[77] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration

system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology (UIST '95)*, page 111—120, Pittsburgh, Pennsylvania, USA, 1995. ACM Press.

[78] Moira C. Norrie, Alexios Palinginis, and Beat Signer. Content publishing framework for interactive paper documents. In *Proceedings of DocEng 2005, ACM Symposium on Document Engineering*, Bristol, United Kingdom, 2005.

[79] Gérald Oster. *Réplication Optimiste et Cohérence des Données dans les Environnements Collaboratifs Répartis*. Thèse de doctorat, Université Henri Poincaré - Nancy I, November 2005.

[80] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Proceedings of the IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'06)*, page 1—10, Atlanta, Georgia, USA, November 2006. IEEE Computer Society.

[81] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Real-time group editors without operational transformation. Research Report RR-5580, LORIA —INRIA Lorraine, May 2005.

[82] Christopher R. Palmer and Gordon V. Cormack. Operation transforms for a distributed shared spreadsheet. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work (CSCW '98)*, page 69—78, Seattle, Washington, USA, 1998. ACM Press.

[83] Constantinos Papadopoulos. A multiple granularity locking protocol for cscw. *International Journal of Cooperative Information Systems*, 11(1-2):21–50, 2002.

[84] Stavroula Papadopoulou, Claudia-Lavinia Ignat, Gérald Oster, and Moira C. Norrie. Increasing awareness in collaborative authoring through edit profiling. In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'06)*, Georgia, Atlanta, USA, November 2006.

[85] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295—330, 1994.

[86] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proceedings of the Eleventh International Conference on Cooperative Information Systems (CoopIS'03)*, volume 2888 of *Lecture Notes in Computer Science*, page 38—55, Catania, Sicily, Italy, November 2003. Springer.

[87] Matthias Ressel. Kooperative interaktionsunterstützung in groupware. In *Software-Ergonomie '95, Mensch - Computer - Interaktion, Anwendungsbereiche lernen voneinander: Gemeinsame Fachtagung des German Chapter of the ACM, der Gesellschaft für Informatik (GI), der TH Darmstadt und GMD/IPSI*, page 311—329. Teubner, 1995.

[88] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'96)*, pages 288–297, Boston, Massachusetts, USA, November 1996. ACM Press.

[89] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42—81, March 2005.

[90] Sunil Sarin and Irene Greif. *Computer-based real-time conferencing systems (Reprint)*, page 397—422. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1988.

[91] Marc Shapiro and Karthik Bhargavan. The actions-constraints approach to replication: Definitions and proofs. Research Report MSR-TR-2004-14, Microsoft Research, March 2004.

[92] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *International Conference On Principles Of Distributed Systems (OPODIS'04)*, page 331—345, 2004.

[93] Marc Shapiro and Nishith Krishna. The three dimensions of data consistency. In *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR'05)*, November 2005.

[94] Haifeng Shen and Chengzheng Sun. Flexible merging for asynchronous collaborative systems. In *Proceeding of the Conference on*

Cooperative Information Systems (CoopIS'02), volume 2519 of *Lecture Notes in Computer Science*, page 304—321, Irvine, California, USA, November 2002. Springer-Verlag.

[95] Haifeng Shen and Chengzheng Sun. A log compression algorithm for operation-based version control systems. In *26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, page 867—873, Oxford, England, August 2002.

[96] HongHai Shen and Prasun Dewan. Access control for collaborative environments. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, page 51—58, Toronto, Ontario, Canada, 1992. ACM Press.

[97] Beat Signer. *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces*. PhD thesis, ETH Zurich, Diss. ETH No. 16218, 2005.

[98] Hala Skaf-Molli, Pascal Molli, and Gérald Oster. Semantic consistency for collaborative systems. In *Proceedings of the International Workshop on Collaborative Editing Systems, ECSCW'03*, Helsinki, Finland, 2003.

[99] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. *Beyond the chalkboard: computer support for collaboration and problem solving inmeetings (Reprint)*, page 335—366. Morgan Kaufmann Publishers Inc., 1988.

[100] Maher Suleiman. *Serialisation des operations concurrentes dans les systemes collaboratifs repartis*. PhD thesis, Université Montpellier II, July 1998.

[101] Maher Suleiman, Michéle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work (GROUP '97)*, page 435—445, Phoenix, Arizona, USA, 1997. ACM Press.

[102] Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the International Conference on Data Engineering - ICDE'98*, pages 36–45, Orlando, Florida, USA, February 1998. IEEE Computer Society.

[103] Chengzheng Sun. Optional and responsive fine-grain locking in internet-based collaborative systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):994–1008, 2002.

[104] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309—361, December 2002.

[105] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction*, 9(1):1—41, 2002.

[106] Chengzheng Sun, David Chen, and Xiaohua Jia. Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems. In *Proceedings of the 21st Australasian Computer Science Conference*, Perth, Australia, February 1998.

[107] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms and achievements. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'98)*, page 59—68, Seattle, Washington, USA, November 1998. ACM Press.

[108] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63—108, March 1998.

[109] Chengzheng Sun, Yun Yang, Yanchun Zhang, and David Chen. Distributed concurrency control in real-time cooperative editing systems. In *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security (ACSC'96)*, volume 1179 of *Lecture Notes in Computer Science*, page 84—95, Singapore, December 1996. Springer Verlag.

[110] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work (CSCW'04)*, page 437—446, Chicago, Illinois, USA, 2004. ACM Press.

[111] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, 2002.

[112] Walter F. Tichy. RCS - A system for version control, November 22 1991.

[113] Osamu TORII, Tetsuro KIMURA, and Junichi SEGAWA. The consistency control syste of xml documents. In *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)*, page 102, Washington, District of Columbia, USA, 2003. IEEE Computer Society.

[114] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[115] Andrei A. Vancea. Asynchronous collaborative graphical editors. Diploma project, Institute for Information Systems, ETH Zurich, 2004.

[116] Nicolas Vidot. *Convergence des Copies dans les Environnements Collaboratifs Répartis*. PhD thesis, Université Montpellier II, September 2002.

[117] Nicolas Vidot, Michèle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'00)*, page 171—180, Philadelphia, Pennsylvania, USA, December 2000. ACM Press.

[118] V. von Biel. Groupware grows up. *MacUser*, June:207—211, 1991.

[119] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE'03)*, page 519—530, Bangalore, India, 2003.

[120] Kevin Williams, Michael Brundage, and et al. *Professional XML Databases*. Wrox Press Ltd., 2000.

[121] Steven Xia, David Sun, Chengzheng Sun, and David Chen. Collaborative object grouping in graphics editing systems. In *Proceedings of IEEE 2005 International Conference in Collaborative Computing (CollaborateCom'05)*, San Jose, California, USA, December 2005.

[122] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work (CSCW'04)*, page 162—171, Chicago, Illinois, USA, 2004. ACM Press.

[123] Liyin Xue, Mehmet Orgun, and Kang Zhang. A multi-versioning algorithm for intention preservation in distributed real-time group editors. In *Proceedings of the 26th Australasian computer science conference (ACSC'03)*, page 19—28, Adelaide, Australia, 2003. Australian Computer Society, Inc.

[124] Liyin Xue, Kang Zhang, and Chengzheng Sun. Conflict control locking in distributed cooperative graphics editors. In *Proceedings of the First International Conference on Web Information Systems Engineering (WISE'00)*, pages 401–408, Washington, District of Columbia, USA, 2000. IEEE Computer Society.

[125] Ali Asghar Zafer. Netedit: A collaborative editor. Master's thesis, Virginia Polytechnic Institute and State University, 2001.

Curriculum Vitae

Name:	Claudia-Lavinia Ignat
Date of Birth:	November 29, 1976
Birthplace:	Cluj-Napoca, Cluj, Romania
Citizenship:	Cluj-Napoca, Cluj, Romania

1983-1991	Primary and Secondary School in Cluj-Napoca, Romania
1991-1995	Computer Science Highschool, Cluj-Napoca, Romania
1995	Diploma de Bacalaureat (Matura), Mathematics and Computer Science profile Computer Programmer Diploma
1995-2000	Study of Computer Science at Technical University of Cluj-Napoca, Romania
2000	Diploma Engineer in Computer Science, Technical University of Cluj-Napoca, Romania
2000-2006	Research and teaching assistant supervised by Prof. Moira C. Norrie in the Global Information Systems Research Groups, ETH Zurich