ACE

a collaborative editor

Report Implementation Algorithm

Berne University of Applied Sciences
School of Engineering and Information Technology

Date:	21.06.2005
Version:	0.1
Projectteam:	Mark Bigler (biglm2@hta-bi.bfh.ch) Simon Räss (rasss@hta-bi.bfh.ch) Lukas Zbinden (zbinl@hta-bi.bfh.ch)
Receivers:	Jean-Paul Dubois (doj@hta-bi.bfh.ch) Claude Fuhrer (frc@hta-bi.bfh.ch)
Location:	Subversion Repository

Preface

This document describes the most central part of any collaborative application: the concurrency control algorithm.
This document is split into the following chapters:

Chapter 1: Decisions Describes why we chose the *Jupiter* algorithm as base for our implementation of a concurrency control algorithm. Further it reasons about the choice of transformation functions.

Chapter 2: Concepts Explains how the concurrency control algorithm works. It contains many examples that show possible issues that must be resolved by the algorithm.

Chapter 3: Implementation Shows how the concepts from chapter 2 were implemented.

Prerequisities

In order to understand this report, the reader should at least be familiar with the introduction of 'Report Evaluation Algorithm'. It is also recommended to read the appendix A: 'Undo in multiuser collaborative applications' in the same document.

Referenced Documents

- Report Evaluation Algorithms
- Report Implementation Testframework
- Report Evaluation Network

Contents

List of Tables

List of Figures

Chapter 1

Decisions

1.1 Selection of algorithm

Based on the pre-selection in 4.3.5 of document Report Evaluation Algorithm, only two algorithms passed the criterias, namely *Jupiter* and *adOPTed*. We finally agreed on the *Jupiter* algorithm. Following is a list of arguments that justifies our decision.

- In our discretion, *Jupiter* is the less complex algorithm than *adOPTed*. We considered *adOPTed* the algorithm which entails a higher risk in the occurence of unforeseen implementation problems. Therefore, we followed the well-known KISS programming paradigma.
- More technical issues remained unanswered on the *adOPTed* algorithm, e.g. concurrent joining/writing. Therefore, we would have taken a higher risk when choosing *adOPTed*.
- *Jupiter* entails a better scalability. Whereas the number of communication paths in *adOPTed* increases with $n(n-1)$, where n is the number of clients, it only rises linearly with the number of clients in *Jupiter*.
- In our humble opinion, the client/server model of *Jupiter* matches better the concept of collaborative editing, i.e. one client announces a document and becomes the server, whereas other clients connect with the server and join the collaborative editing session.

Recapitulating, with *Jupiter* we have chosen the algorithm with the higher feasibility, that is to say, the one algorithm which is saver for a succesful implementation.

1.2 Selection of transformation function

The *Jupiter* algorithm makes use of a set of transformation functions. Nevertheless, no such functions are proposed in the paper. Therefore, we had to choose a set of transformation functions which would meet the needs of *Jupiter*. The follwoing list identifies the three preconditions for the transformation functions:

- only IT required
- TP1 must be satisfied (cf. section 4 in Report Evaluation Algorithm)

- should be extensible to stringwise transformations

Following is a list of possible transformation functions that meet some or all of the preconditions.

- IT from *SOCT2*, satisfies TP1 but not TP2 [1], only characterwise
- IT from *SDT*, satisfies TP1 but not TP2 [2], only characterwise
- IT from *Imine et al.* [3], satisfies both TP1 and TP2, only characterwise
- IT from *GOTO*, satisfies TP1 but not TP2 [4], character- and stringwise

It is clear that all the three preconditions are only met by the transformation functions of the *GOTO* algorithm. They are proposed in "Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems", Sun et al.

[1] Prove in "Achieving convergence with operational transformation in distributed groupware systems", Imine et al.

[2] Prove in "Achieving convergence with operational transformation in distributed groupware systems", Imine et al.

[3] cf. "Achieving convergence with operational transformation in distributed groupware systems", Imine et al.

[4] Prove in "Proving correctness of transformation functions in real-time groupware", Imine et al.

Chapter 2

Concepts

2.1 Introduction

The *Jupiter* system is a collaboration system that supports shared documents, shared tools, and, optionally, live audio/video communication. It was developed by *Xerox*. It is conceptually a collaborative windowing toolkit. The low-level communication facilities are based on operational transformation.
The operational transformation algorithm employed in the *Jupiter* system is derived from *dOPT*. A centralized architecture and thus the reduction to point-to-point connections makes the algorithm significantly simpler than other operational transformation algorithms. The basic *Jupiter* algorithm is only suitable for two sites. However, it was shown in [2] and in greater detail in [3] how to use several point-to-point connections to build a tree-structured n-site algorithm (see figure 2.1).

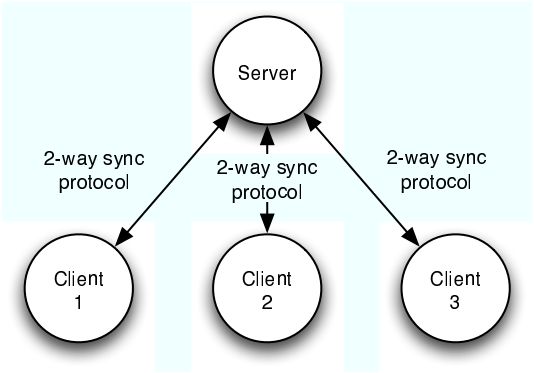

Figure 2.1: multiple 2-way sync to achieve n-way sync

2.2 Jupiter Algorithm

A collaborative editing application requires that local operations are immediately applied to the local document and then sent in a request (that includes a vector time) to other sites. This can result in two or more operations that are executed concurrently. The general tool for handling these conflicting (i.e. concurrent) requests is a transformation function, called

$xform$ in [2].

$$xform(c, s) = \{c', s'\}$$

This transformation function takes two operations, c from the client and s from the server, and returns two transformed operations c' and s'. The operations c' and s' have the property that if the client applies c followed by s', and the server applies s followed by c', both client and server wind up in the same final state (see figure 2.2). This property is also known as transformation property 1 (TP1):

$$cs' \equiv sc'$$

Conceptually, the $xform$ function is a combined inclusion transformation (IT) that returns $\{IT(c, s), IT(s, c)\}$. The parameters c and s must originate from the same document state. This is a necessary precondition of inclusion transformation. If this precondition is violated, the result of the transformation is not guaranteed to be correct.

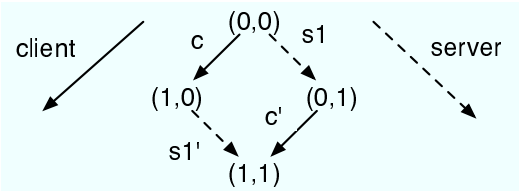

Figure 2.2: basic transformation situation in jupiter

2.2.1 State Space Graph

As we have seen, it is helpful to show the two dimensional state space that both client and server pass through as they process requests (see figure 2.3). Each state is labelled with the number of processed requests from both client and server to that point. For instance if the client is in the state $(2, 3)$, it has generated and processed two requests of its own, and has received and processed three from the server. The client and server requests are displayed on different axis in the state space graph.

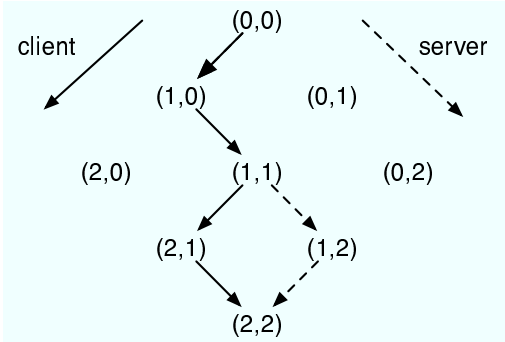

Figure 2.3: two dimensional state space example

If there is a conflict, the paths will diverge, as shown in figure 2.3. The client and server moved to the state $(1, 1)$ together by first processing a client request, and then a server request. At

that point, the client and server processed different requests (concurrently), moving to state $(2,1)$ and $(1,2)$ respectively. They each received and processed the other's message using the transformation function to move to state $(2,2)$.
The algorithm labels each request with the state the sender was in just before the message was generated (state vector). The recipient uses these labels to insert the request at the correct position into his state space. If the request does not start from the current state of the recipient, there is a conflict (concurrent request). Two concurrent requests have to be transformed, but they can only be transformed directly when they were generated from the same document state.
If client and server diverge by more than one step in the state space graph, the transformation function cannot be applied directly. Let us consider the state space in figure 2.4. The client has executed c and receives the conflicting request $s1$ from the server. It uses the transformation function to compute $s1'$ to get to the state $(1,1)$. The server then generates $s2$ from the state $(0,1)$, indicating that it still has not processed c. What should the client do now? It cannot use the transformation function directly because c and $s2$ were not generated from the same document state.

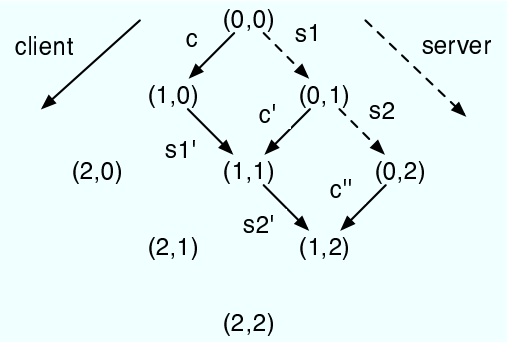

Figure 2.4: client and server diverging by more than one path step

The solution to this situation is as follows. When the client computes $s1'$ it must also remember c'. This represents a hypothetical request that the client could have generated to move from the state $(0,1)$ to $(1,1)$. When $s2$ arrives, the client can use c' to compute c''. It executes $s2'$ to get to the state $(1,2)$. If the server has processed the client's message, it will be in the state $(1,2)$ as well. If not, its next request will originate from $(0,3)$, so the client saves c'' just in case.
The algorithm guarantees that if the transformation function satisfies TP1, then no matter how far the client and server diverge in state space, when they do reach the same state (and they do, unless requests get lost), they will have equivalent states (so convergence is achieved).

2.2.2 Extending Jupiter to N-Way Communication

We have discussed how a single two-way connection works in *Jupiter*. We will now extend this systems to support n clients using multiple two-way connections. The figure 2.5 shows the basic setup for three clients. Each client talks only to the server over a standard previously discussed two-way connection. On the server, there is one so-called client proxy per client. This client proxy conceptually is just the server side algorithm.
When a client, let us say client 1, sends a request to the server, the corresponding client

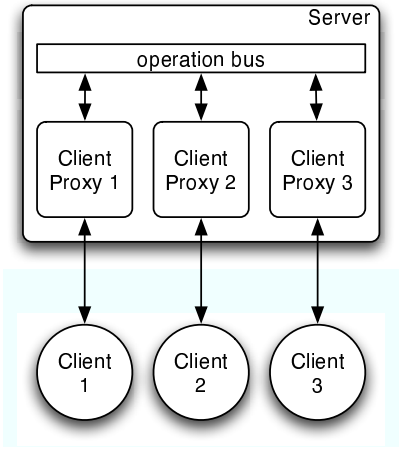

Figure 2.5: n-way synchronization using central server

proxy algorithm transforms the request against any concurrent requests the server already sent to this client but that were not acknowledged by it. Client proxy 1 then dispatches the transformed operation to all other client proxies. The other client proxies (client proxy 2 and 3) get this operation and insert it into their algorithm as local operation (resulting in a new request). The resulting request is then sent to the client 2 and 3 respectively.
Note: One thing must be taken care of in the implementation. Only one request may be processed by the server at any time. That is, the processing of requests from clients must be serialized. Otherwise, the algorithm will not work correctly.

n-way example: In order to better understand how the n-way synchronization work, we examine a simple example. In the following 4 figures, the bottom two state graphs represent two clients, the top two state graphs represent the server, or more exactly the client proxy algorithms on the server.
In the first step, both clients generate a request concurrently and send it to the server (see figure 2.6). Then the request from client 1 arrives at the server earlier than the request from client 2 and is therefore processed first (see figure 2.7). In the next step, the request from client 2 is processed by the server (see figure 2.8). In the last step both clients receive the transformed request from their corresponding client proxy on the server side (see figure 2.9). They insert the requests at the right positions in their state space, doing transformations as necessary (client 2 has to transform the incoming message, client 1 not).

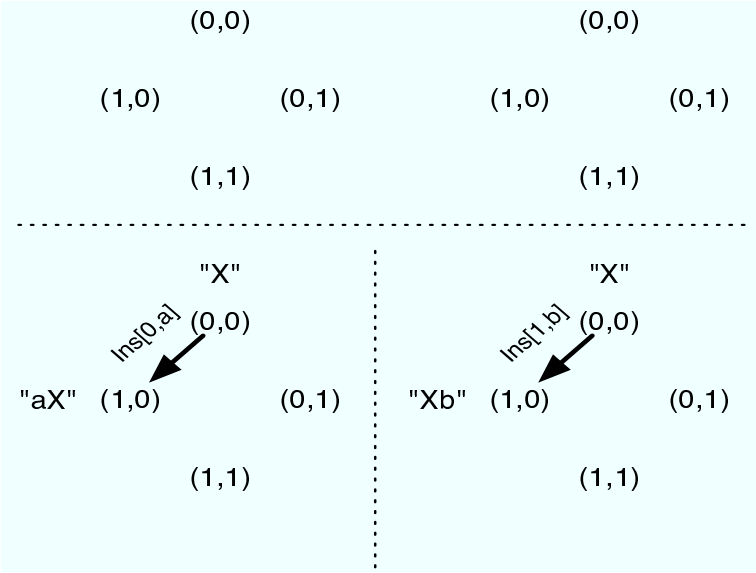

Figure 2.6: n-way example (step 1)

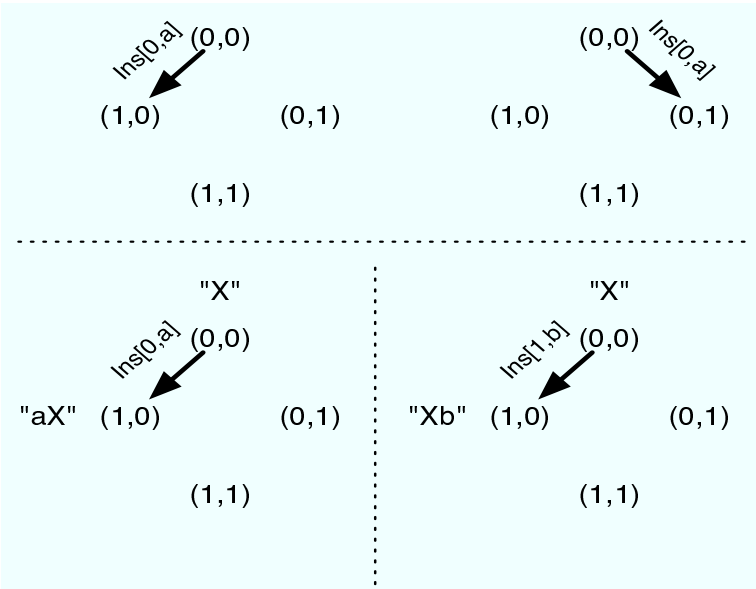

Figure 2.7: n-way example (step 2)

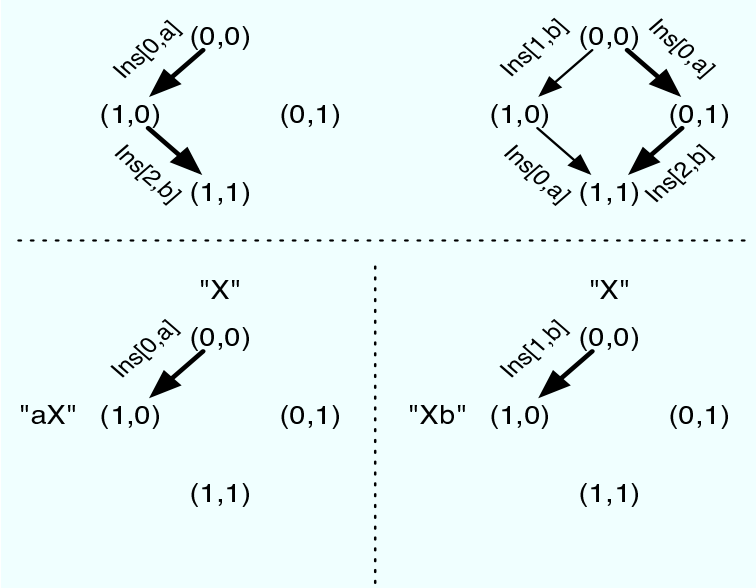

Figure 2.8: n-way example (step 3)

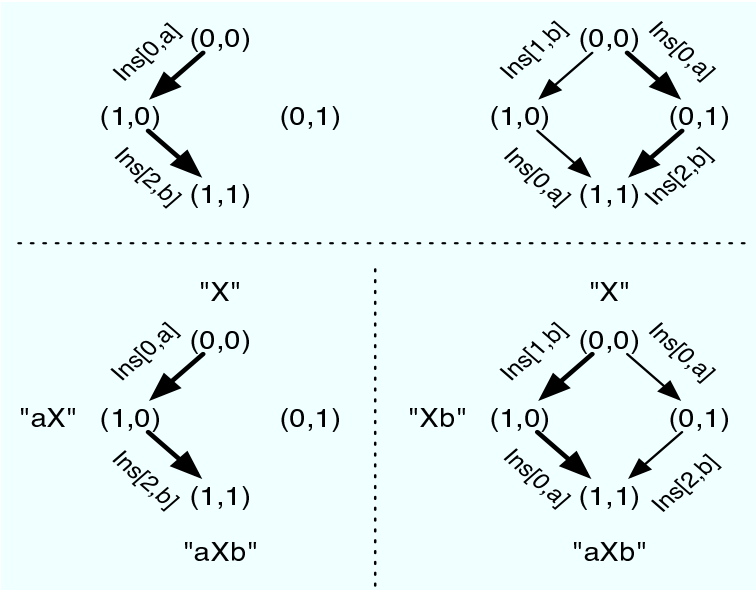

Figure 2.9: n-way example (step 4)

2.3 Undo/Redo

In the preceeding sections, we have presented the basic transformation algorithm. In this section we will show how undo/redo of local operations can be achieved in a real-time collaborative editor. This introduces a whole set of new problems that necessitate additions and modifications to the transformation algorithm. The concepts of this undo solution are partially taken from [1] (*adOPTed* algorithm) but are significantly extended and adapted to

the *Jupiter* algorithm.

2.3.1 Problems of Local Group Undo

Global group undo is not much different from common single-user undo schemes. It can be implemented by executing the inverse operation for each user operation to be undone. The operation to be undone is the globally last operation that was executed. Global undo always leads to a former application state. In contrast, local group undo (undoing the last operation from the local user) is more complex, because in most cases it does not lead to a former application state.

Example 1: Suppose the initial document state is 'm'. Now user X executes the operation `Ins[1,s]` to insert the character 's' at position 1 of the document. Then user Y executes `Ins[1,a]` to insert the character 'a' at position 1 resulting in a document state 'mas'. Now user X decides to undo his last insertion operation. What has to be done? The naive, but incorrect approach would just execute `Del[1,s]`, i.e. the inverse to `Ins[1,s]`. However, the current character at position 1 is not 's' but 'a'. The situation is depicted in figure 2.10.

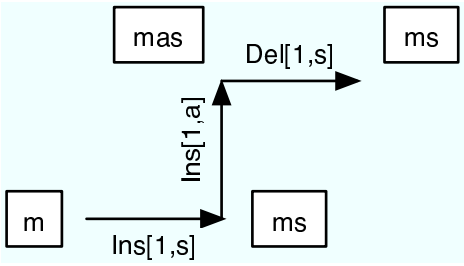

Figure 2.10: incorrect naive approach to local group undo

Obviously, just executing the inverse of the last local operation cannot achieve the correct result. In addition, there are more situations were local group undo methods can fail. One characteristic example is the following problem, which is called *order puzzle*.

Example 2: Suppose we start with a document containing 'ab'. Now user 'X' first deletes 'a' by executing `Del[0,a]`. Afterwards user Y deletes 'b' by issuing the operation `Del[0,b]`. This will leave the document empty. Now both users undo their operations. The expected application state should be 'ab' afterwards, because undoing all operations should not modify the original application state. Unfortunately, a naive undo solution could yield the wrong state 'ba'. The solution to the order puzzle will be presented in a later section (see section 2.3.4).

2.3.2 Transformation-Based Local Group Undo

Let us return to the example 1. Suppose we want to undo the last operation after an operation from another participant has already been executed. For correctness of local group undo, we require that the effect of the undo request does not depend on other users' requests. Therefore, we may safely assume, that we issued the undo operation immediately after the execution of

the operation to be undone. Then the appropriate transformations can be applied to gain the operation that has to be executed in the present state.
In example 1, we insert the inverse operation `Del[1,s]` immediately after the operation. The procedure of inserting the inverse operation is called *mirroring*. In the next step, the inverse operation `Del[1,s]` is transformed against the operation `Ins[1,a]` from user Y, which results in the correct operation `Del[2,s]`. See figure 2.11.

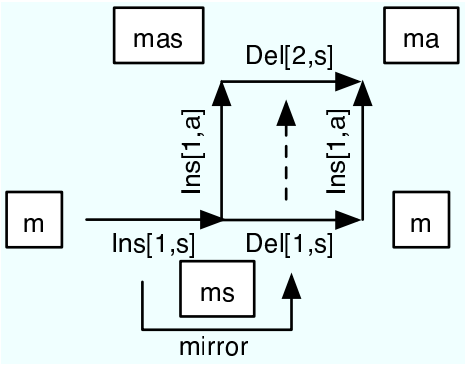

Figure 2.11: local undo by applying transformations

Unfortunately, the computation of requests in the interaction model by transformation and by mirroring is not sufficient to solve the *order puzzle* in example 2.

Example 3: Suppose both users X and Y simultaneously delete the single character 'b' (see figure 2.12). The corresponding transformation rule transforms both operations to `Noop` operations (no operation, operation without any effect on the document state), since deleting the same character twice is impossible. Now suppose, user Y wants to undo his operation `Del[0,b]`. We know this can be achieved by applying the mirror operator, producing the operation `Ins[0,b]`. Now, how can the missing operations be computed? By just applying the transformation rule, the operations would just be copied without modification, e.g., another `Noop` operation would lead from state 'b' to the end state 'b' (in the upper right corner). However, this violates the principle of correctness, namely that a set of do-undo pairs like `Del[0,b]` and `Ins[0,b]` should have no effect on other users' operations. If this were to be satisfied, the resulting state should be empty, like when executing user X's operation `Del[0,b]` only.

2.3.3 Fold Operator

To solve the puzzle in example 3, we observe that all parallel lines connected by do-undo pairs should actually contain congruent requests and states. Here in the example O_1'' should be `Del[0,b]`, and the resulting state should be empty, like the operation and the state at the bottom of the diagram (see figure 2.12).
We denote the copying of a request from the beginning of a do-undo pair - or more general a set of possibly nested do-undo pairs - to the end of the do-undo pair as *folding*. This illustrates the fact that we might actually fold the interaction model - here along the straight line defined by the horizontal `Noop` request in the center - so that both lines come to lie above each other.

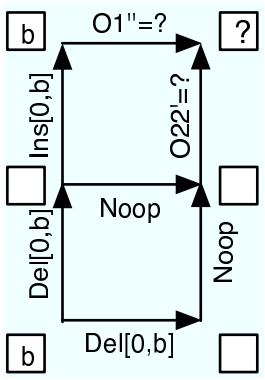

Figure 2.12: application of mirror and fold operators

To complete the interaction model we demand that O'_{22} will be computed by mirroring the operation `Noop` (in figure 2.12 at the bottom right). The resulting interaction model with all mirror and fold operators applied is displayed in figure 2.13.

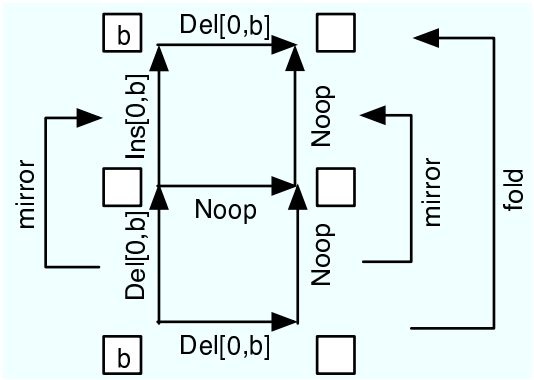

Figure 2.13: application of mirror and fold operators

2.3.4 Solution to the Order Puzzle

The solution to the order puzzle in example 2 can be computed with the same operators, mirror and fold. The desired result is shown in figure 2.14.

2.3.5 Additional Problems in n-Way Situation

Unfortunately, there exists another problem in the n-way situation that do not exist in the 2-way situation. To illustrate the problem and the solution we introduce another example.

Example 4: First, user X generates the request `Del[0,b]` from the initial document state 'b'. Then, he undoes that change immediately (generating the mirror request `Ins[0,b]`). User Y generates the request `Del[0,b]` concurrently to the two requests from user X (see figure 2.15). In figure 2.16 the first request from user X is processed by the server and then by the user Y. Nothing unusual happens in this situation. Then the request from user Y is processed by the server, resulting in a `Noop` operation (see figure 2.17).

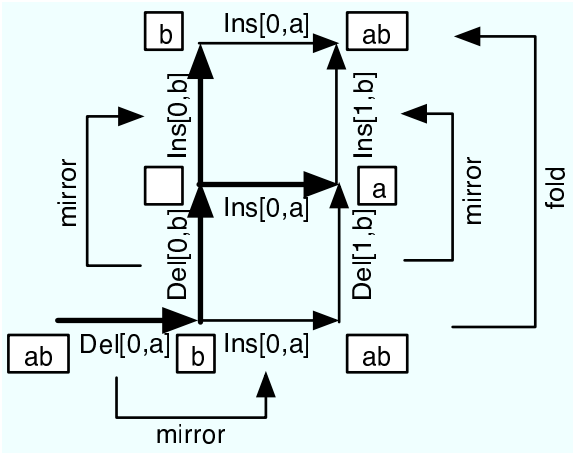

Figure 2.14: Solution to the Order Puzzle

The next step (see figure 2.18) reveals the problem. The `Noop` operation is inserted into the state space of user X. Now we need to calculate operation O_x. Because the `Noop` operation is between a do-undo pair (mirror), O_x must be calculated by folding the state space graph, that is by putting the dashed operation O_x from the bottom to the top line. The problem is, we do not know what this operation was, so we cannot apply the fold operator.
In figure 2.19 we see a similar problem on the server-side. Here, the undo request from user X has arrived at the server and is inserted into the state space graph at the correct position (from $(0,1)$ to $(0,2)$). Because we have a do-undo pair (mirror) on the bottom line, we must have a mirror operation on the second line too according to our transformation rules. In order to calculate O_y we need to know the operation marked with three question marks first. This operation can be calculated by transforming `Del[0,b]` against O_x. Once again, it is the operation O_x that is missing.
So how do we find out O_x? The answer lies in the fact that the `Noop` request resulted from a transformation, so O_x was known by the client proxy of user Y (`Del[0,b]`). If we save the transformation history of an operation, we can use these previous forms of an operation to successfully apply mirror and fold operators and thus solve the problem presented in this example.

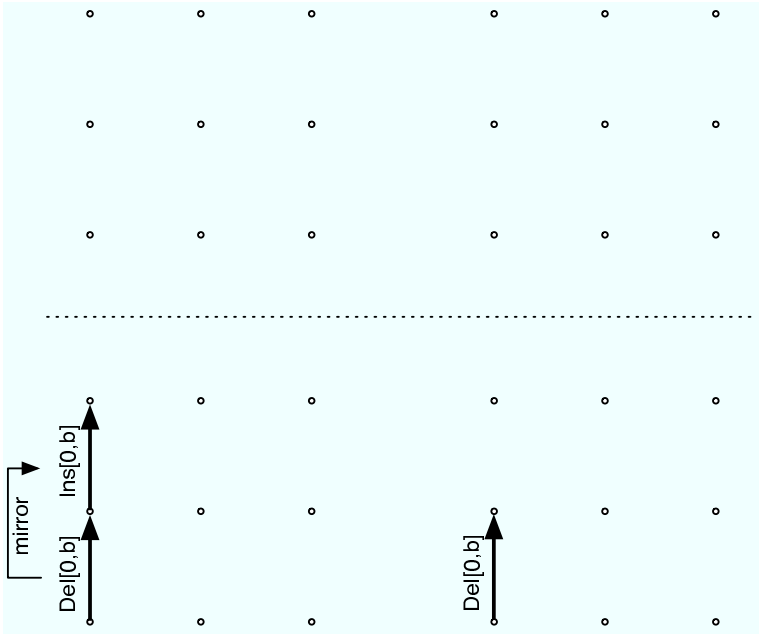

Figure 2.15: n-way undo problem (step 1)

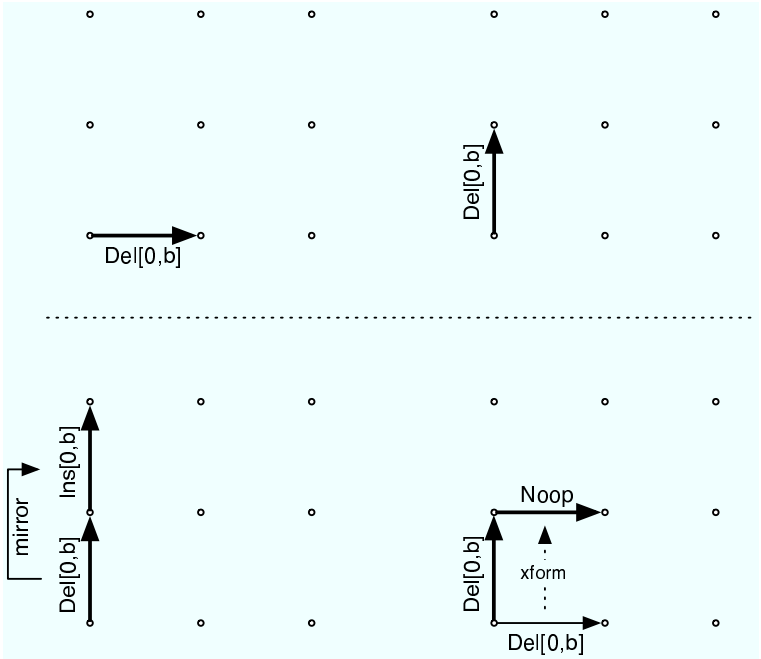

Figure 2.16: n-way undo problem (step 2)

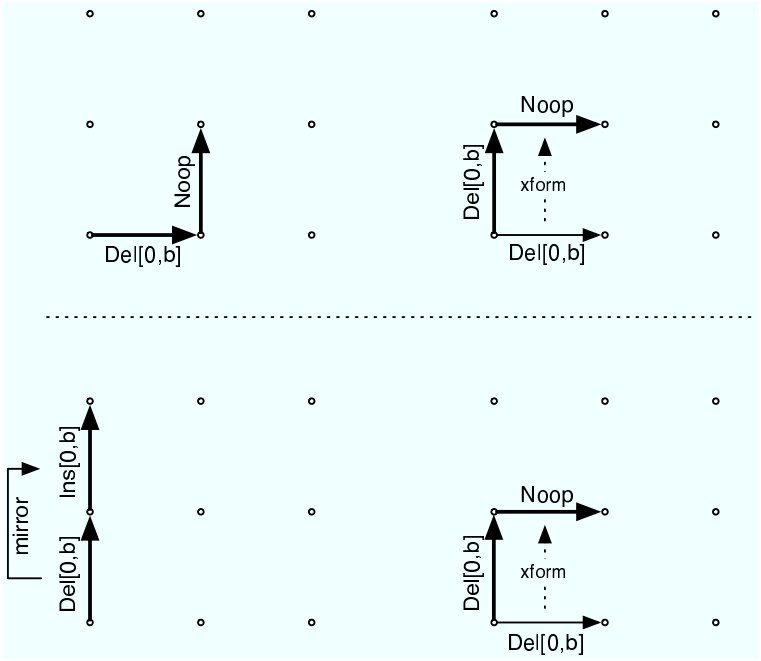

Figure 2.17: n-way undo problem (step 3)

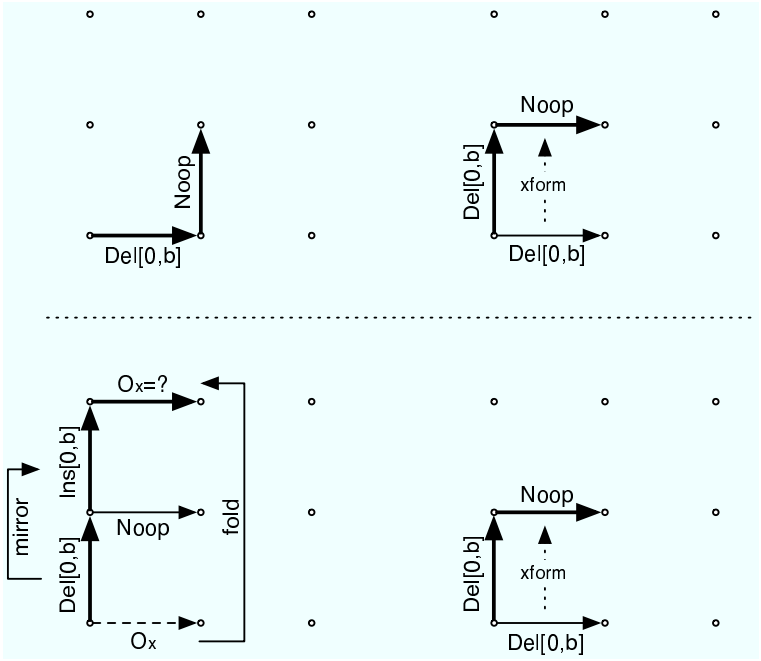

Figure 2.18: n-way undo problem (step 4)

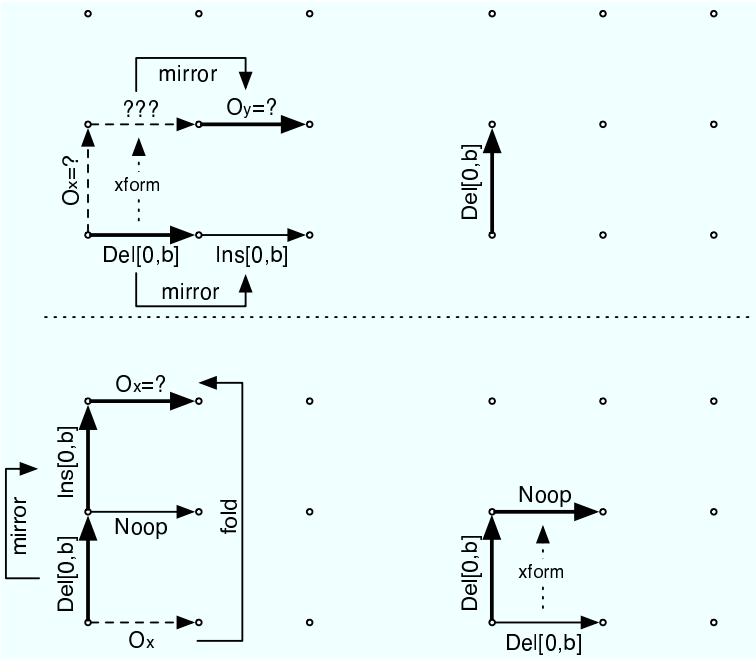

Figure 2.19: n-way undo problem (step 5)

2.3.6 Redo

The integration of a redo operation, as far as we know, has never been considered before in the area of operational transformation algorithms. However, it turned out to be rather straightforward to implement a redo feature. A redo can only be done, if the last operation from a user has been an undo or there are only undo-redo pairs between the current state and an undo operation. A do operation between the current state and an undo operation effectively disables the redo functionality (this is the normal behavior in any modern text processing application).

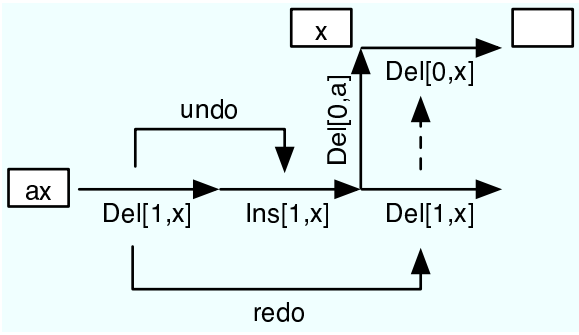

Figure 2.20: Redo Operation

In figure 2.20 we see how a redo is inserted into the state space. First, the last operation from this user must be a undo (or there are only undo-redo pairs between the current state and an undo operation). This is clearly the case in this example. We then copy the corresponding do operation to the current column in the state space on the same line as the original operation

(see arrow labeled redo). Then, we have to transform this copied operation to the current state, so that it can be applied.

2.4 GOTO Transformation Functions

2.4.1 Overview

The GOTO (generic operation transformation optimized) transformation functions are designed to work with strings as well as characters. The advantage is that less transformations have to be done when a string is inserted into a text. To understand the exigence of transformation functions see *Report Evaluation Algorithms*.
The inclusion transformation functions are used to include the effect of a given operation B into another operation A. The transformed operation A' is returned. To transform an operation means to adapt position and text of the operation. In the majority of cases this is a very simple process. But sometimes it is necessary to extract a text fragment or to split up an operation into two parts. For more details about splitting operations see 3.1.
All possible transformation *cases* with two insert/delete operations are depicted in figure 2.21 and explained in the following sections:

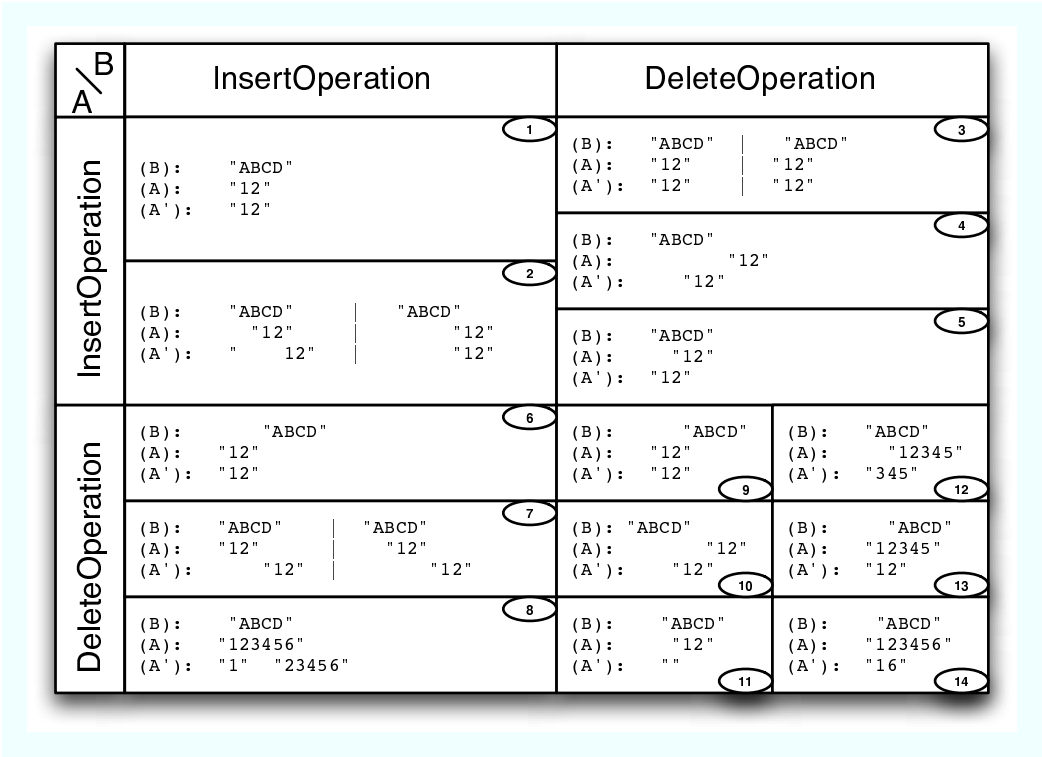

Figure 2.21: Transformation Overview

2.4.2 Insert/Insert

- **Case 1:** Operation A starts before or at the same position as operation B. Sometimes a special handling is necessary. See *Special Case* in section 2.4.2 for detailed information

about that.

- **Case 2:** Operation A starts inside or after operation B. Index of operation A' must be increased by the length of the text of operation B.
- **Special case:** It can happen that two insert operations (for example `Ins(0,'a')` and `Ins(0,'b')`) have the same position. How should the correct transformation look like (does the insertion of 'a' have to be before or after 'b')? This is an undecidable problem. Therefore some extra rules must be defined. The first approach to solve this problem is by preferring the operation with the lower ascii value. With this method the final text would be 'ab'. The following figure demonstrates that this solution can violate the user's intention:

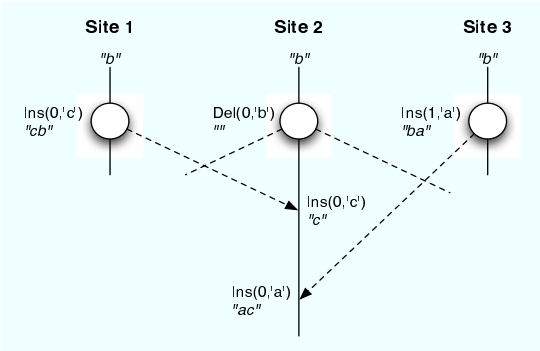

Figure 2.22: Transformation based on character code

The scenario depicted in figure 2.22 demonstrates an editing session with 3 users (sites). Each site generates a request concurrently. For simplicity only the process flow at site 2 is shown. The initial document state is 'b'. After applying the own operation (`Del(0,'b')`) the document becomes empty. Then the two other operations arrive. First the operation `Ins(0,'c')` from site 1 and then `Ins(1,'a')` from site 3, which will be transformed into `Ins(0,'a')`, because the delete operation has an effect on it. The two operations (`Ins(0,'a')` from site 3 and `Ins(0,'c')`) from site 1 must be transformed and the resulting document state becomes 'ac', which violates the user intentions.

The second approach works with some extra information. If an operation has been generated, the original position is saved too. Adapted to the scenario in figure 2.22 the insert operation of site 1 would look like `Ins(0,'a',0)`. Similar to the insert operation of site 1 the insert operation of site 3 looks like `Ins(1,'b',1)`. The added third parameter represents the original position. At the beginning the position and the original position have the same value. The difference of the two positions is shown after a transfomation. Unlike the position (first parameter) the original position will never be transformed/changed and remains the same for the livetime of the operation.

Considering the second approach, the site 1 sends the request `Ins(0,'c',0)` to site 2. After site 2 received the request, site 3 sends the request `Ins(1,'a',1)`. On site 2 this request will first be transformed against `Del(0,'b')` into `Ins(0,'a',1)`. This is necessary because while site 3 was generating the request, site 2 deleted the character `b`.

After this transformation the new request has to be transformed with the request from site 1. This transformation would look like `transform( Ins(0,'a',1), Ins(0,'c',0) )` which means that the effect from the operation `Ins(0,'c',0)`) will be included into operation `Ins(0,'a',1)`. Now the problem is the same as before, but with the extra information it can be solved. After detecting that the two operations have the same position the transformation function compares the original positions. With this solution the 'c' will be inserted before the 'a' and the user intention is preserved.

Unfortunately figure 2.22 shows that even with using original positions in the transformation function, problems can occur:

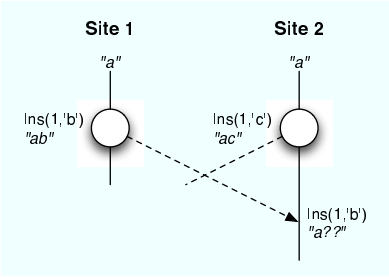

Figure 2.23: Transformation based on equal original positions

At the same time each site generates an insert operation with the same position. The insert operation from site 1 looks like `Ins(1,'b',1)` and the operation from site 2 is `Ins(1,'c',1)`. When site 2 receives the request from site 1 the operations have to be tranformed (`transform( Ins(1,'b',1) , Ins(1,'c',1) )`). The transformation function will detect that the two operations have the same position and will check the original positions. Unfortunately the original positions have the same values. Therefore the approach to solve this problem by using original positions will fail too.

This problem can only be eliminated by privileging one operation. But which operation should be preferred? And how to guarantee that on all sites this will be the same operation? One possible solution would be to use the site id's (e.g. site 1 is always privileged against site 2 and site 3, site 2 is only privileged against site 3). When using this transformation function with the *Jupiter* algorithm, the operation which has been issued by the server side is always privileged (client/server flag).

- **The concept of last synchronization point (LSP)** Again we consider the situation where to insert operation are to be transformed and both have the same positions. The comparison of the original positions works quite well as shown above. But it has shortcomings. What if the two operations do not relate on the same context and document state, respectively? Though the comparison of the original positions can still take place, it is a weak comparison because the user's intention may get lost as shown

in the example depicted in the following figure:

Only site 4 is considered. When all operations have arrived, the last transformation

to be done includes the two insert operations. Both are equivalent in that they have the same positions as well as the same original positions. The remarkable thing is that though the two original positions are equal, they do not relate on the same document state. Operation $Ins(2,'c',2)$ from site 3 was generated after the operation from site 4 had been received. It is clear that the user's intentions of site 3 and 1 are violated in that user 1 inserted 'b' after '1' and user 3 inserted 'c' before '1', but in the final document state the 'b' is placed before 'c'. Here the comparison of the original positions fails. We need a more advanced concept.

Therefore we introduce the concept of "last synchronization point" (LSP), analogously adapted to our implementation of *Jupiter*. It is taken from paper "Preserving Operation Effects Relation in Group Editors", Du Li and Rui Li. The idea comes right into play at the situation described above. When the two original positions need to be compared, it should first be ensured that both operations and original positions, respectively, rely on the same document state. If this happens, they can savely be compared and in case of equality the client/server flag included. If they do not relate on the same document state, we need to find a last synchronization point where the two operations can correctly be compared. In the example above, this would be achieved as follows. When the two insert operations are compared, it is detected that they cannot be compared as is. Therefore, the first operation from site 4 would be excluded from operation $Ins(2,'c',2)$, which in fact is the first one at site 3. With this exclusion, we arrive at the same document state for both operations and thus we can compare them correctly. Operation $Ins(2,'b',2)$ would be transformed so that 'b' gets inserted after 'c', which was the user's implicit but actual intention.

Recapitulating, we would like to implement this concept to improve the system's ability to preserve the user's intention also in the most complex situations. Yet we do not know if the LSP concept can be implemented at all in our system. This feature is left open as future work.

2.4.3 Insert/Delete

- **Case 3:** Operation A starts before or at the same position as operation B. Nothing has to be transformed.
- **Case 4:** Operation A starts after operation B. Index of operation A' must be reduced by the length of the text of operation B.
- **Case 5:** Operation A starts inside operation B. Index of operation A' must be the index of operation B.

2.4.4 Delete/Insert

- **Case 6:** Operation A is completly before operation B. Nothing has to be transformed.
- **Case 7:** Operation A starts before or at the same position as operation B. Index of operation A' must be increased by the length of the text of operation B.
- **Case 8:** Operation B is in the range of operation A. Operation A' must be splitted into two delete operations. For more details about the split operation see 3.1.

2.4.5 Delete/Delete

- **Case 9:** Operation A is completely before operation B. Nothing has to be transformed.
- **Case 10:** Operation A starts at the end or after operation B. Index of operation A' must be reduced by the length of the text of operation B.
- **Case 11:** Operation A and operation B are overlapping. Operation B starts before or at the same position as operation A and ends after or at the same position as operation A. Content of operation A has already been deleted by operation B. Therefore, nothing has to be deleted anymore by operation A. A' is called a noop (no-operation).
- **Case 12:** Operation A and operation B are overlapping. Operation B starts before or at the same position as operation A and ends before operation A. The overlapping part of the two operations has been deleted by operation B. Operation A' has to delete only the remaining text (text after the overlapping text of the two operations).
- **Case 13:** Operation A and operation B are overlapping. Operation B starts after operation A and ends after or at the same position as operation A. The overlapping part of the two operations has been deleted by operation B. Operation A' has to delete the remaining text (text before the overlapping text of the two operations).
- **Case 14:** Operation A and operation B are overlapping. Operation B is fully inside operation A. The overlapping part of the two operations has been deleted by operation B. Operation A' has to delete the remaining text (text before and after the overlapping text of the two operations).

Chapter 3

Implementation

3.1 Operation

Operations are used to decribe changements made in a document. For this purpose they must contain the information: *what* has been changed and *where* it has been changed.
In case of simple text editing only two operations are required. An insert operation (e.g. `Ins(1,'hello')`) which will insert the string 'hello' at position 1 and an delete operation (e.g. `Del(10, 'world')`) which will delete the string 'world' at position 10.

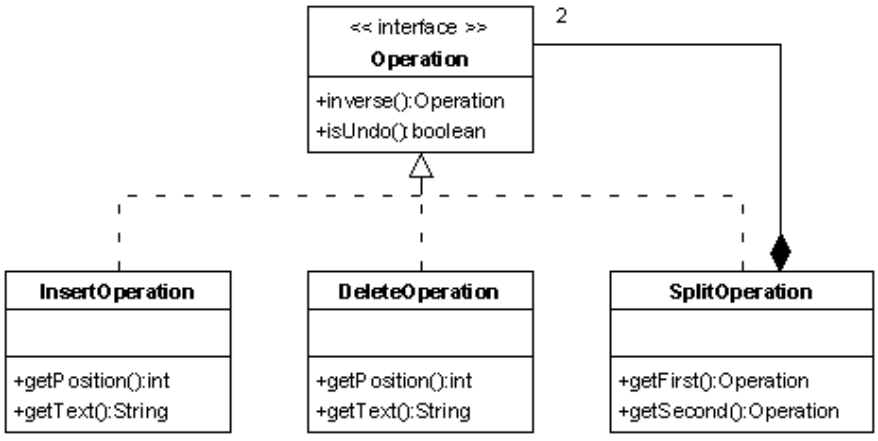

Figure 3.1: Operation Hierarchy

As depicted in figure 3.1 there exists a third operation besides the `InsertOperation` and `DeleteOperation`. The so-called `SplitOperation` is a helper object used to encapsulate two operations (usually two delete operations). This special operation is required when transforming an insert operation which occurs in the range of a delete operation. For more details about transformation functions see 2.4, especially the spitting scenario described in *Case 8* in 2.4.4.

3.2 Request

A request is used to distribute changes of a document over the network. It contains all important information such as an operation (what has changed and at which position), a timestamp (document state on which the operation is based) and the site id of the generating site. (see figure 3.2)

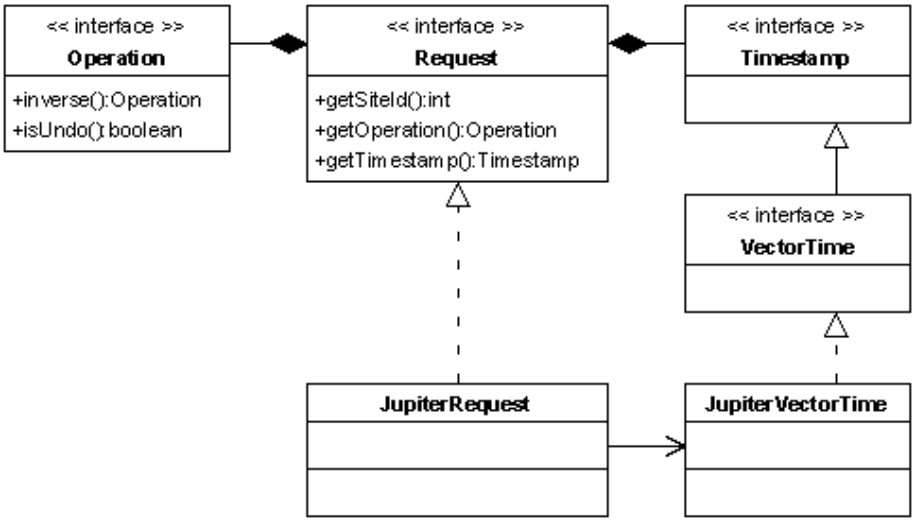

Figure 3.2: Request Class Diagram

How a request will be sent to a server or another client is not part of this chapter and depends on the network technology (see *Report Evaluation Network*) used.

3.3 Algorithm

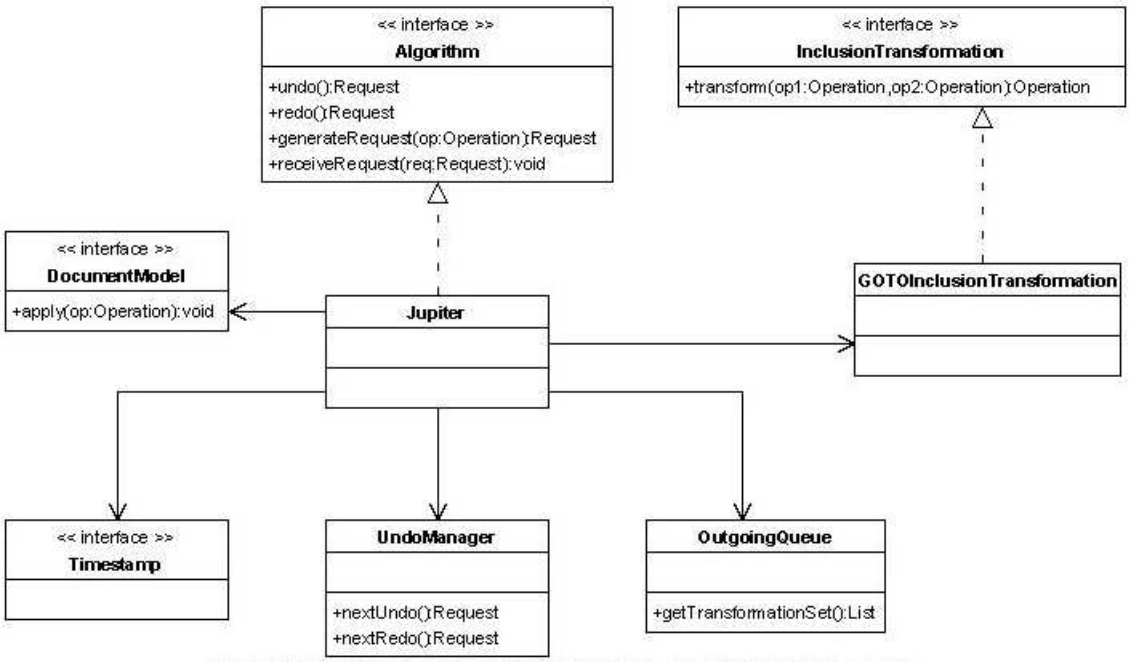

Figure 3.3: Algorithm architecture

The algorithm component centers around the Jupiter class, which is an implementation of the Algorithm interface. Jupiter processes all requests with the collaboration of the GOTOInclusionTransformation. All transformed operations are applied to the DocumentModel instance. The Timestamp class represents the current state vector, i.e. the location in the 2-dimensional state space of the *Jupiter* algorithm. The UndoManager class is apparently used for undo/redo functionality and is further described in 3.6.

3.4 Client

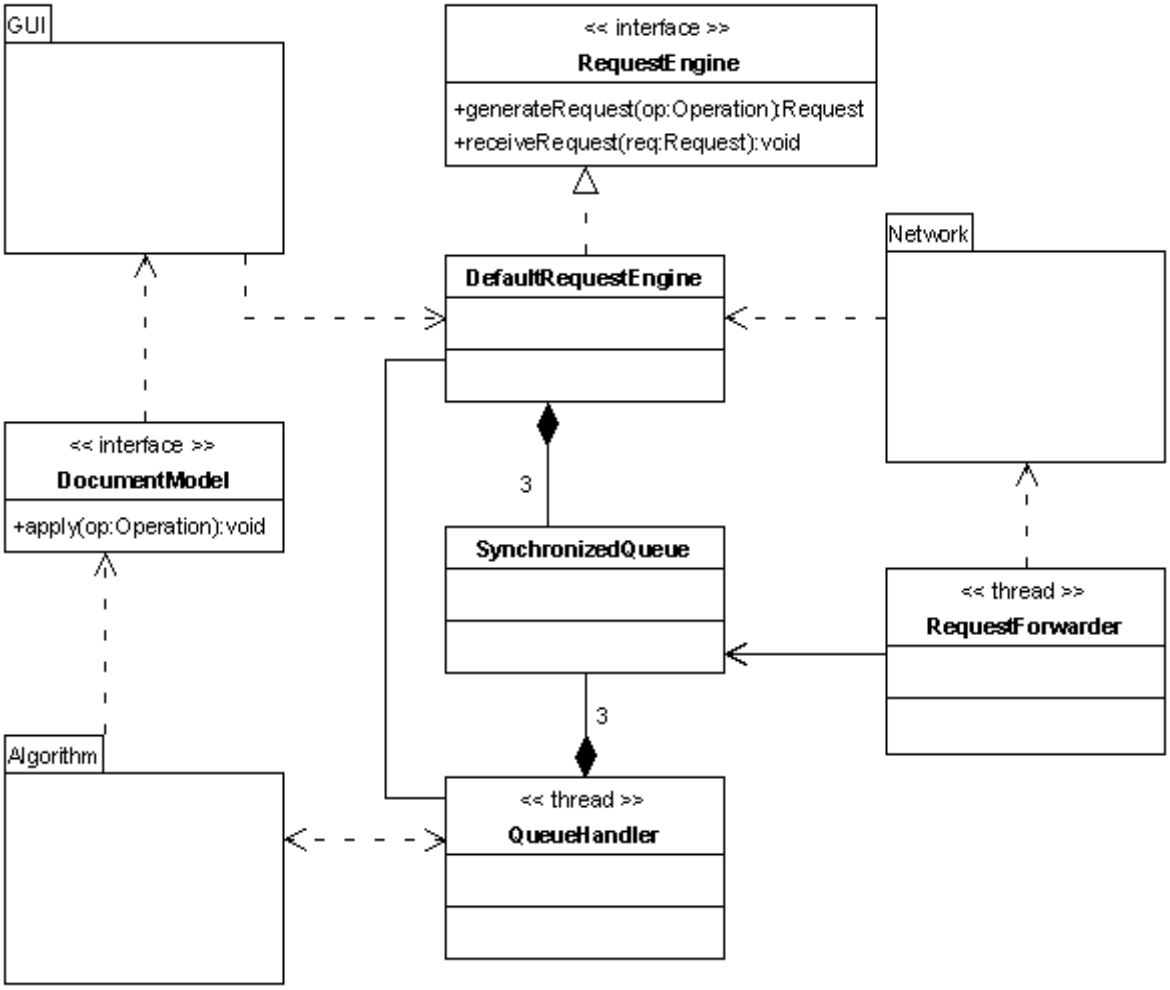

Figure 3.4: Client architecture

The client component centers around the RequestEngine class. The RequestEngine is a mediator between the GUI, network and algorithm components. It receives requests from the GUI and the network, respectively. All requests are passed to synchronized queues, one for local requests and one for remote requests. The requests are read out from the queues by the QueueHandler class. This thread waits on the queues until a request has been received. Each request is then handed to the algorithm package for processing. Processing involves transforming the operation and applying it to the document model.
Locally generated requests are returned from the algorithm package to the QueueHandler which in turn passes it to another SynchronizedQueue, namely the outgoing queue. From there, all outgoing requests are forwarded to the network package by the RequestForwarder class.

3.5 Server

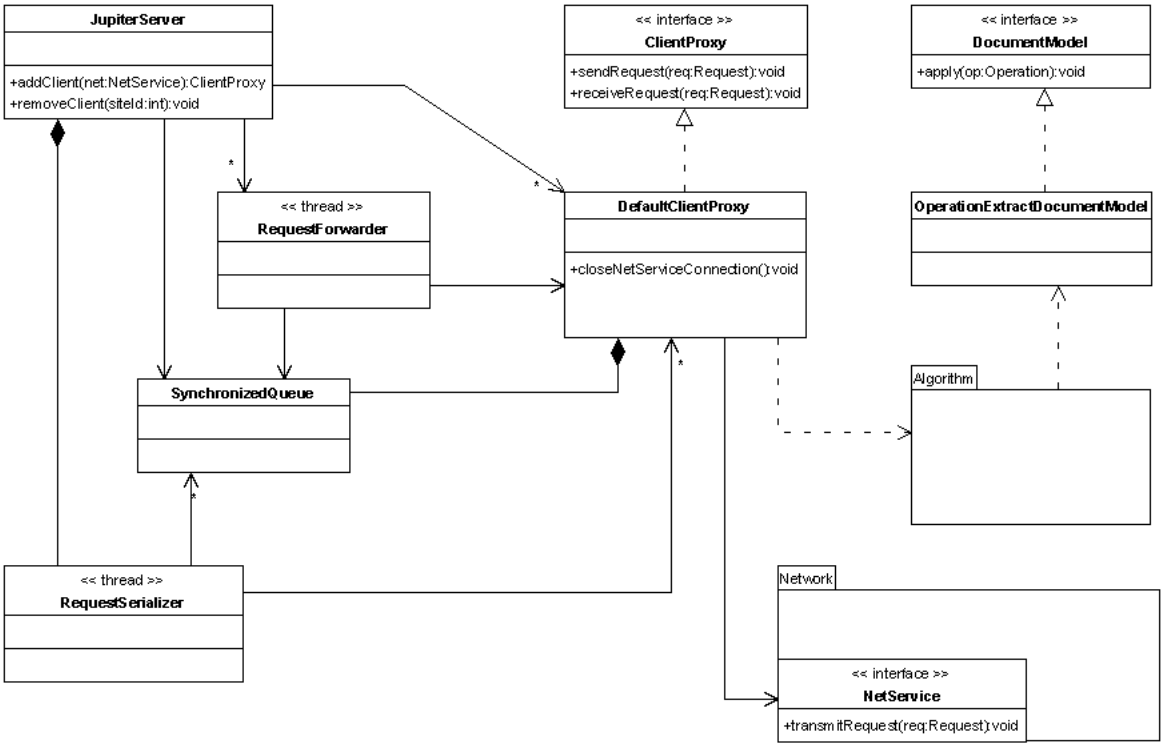

Figure 3.5: Server architecture

The *Jupiter* server component is the turntable in a multi-user collaborative editing session. It achieves n-way communication by use of the 2-way synchronization protocol of *Jupiter* between each client and its corresponding ClientProxy on the server side (instances of class ClientProxy). The *Jupiter* server component can be started and organized by creating a new instance of class JupiterServer. It allows to manage client proxies, that is to add and remove them.

3.5.1 Adding a client

When a new client is added to the server, a new ClientProxy is created. It is passed references to instances of the following types: NetService, Algorithm (Jupiter), SynchronizedQueue. The NetService class is used to forward outgoing requests to the network layer. The algorithm is responsible for the operational transformation. Since no real document model is used at the server side, a special implementation of the DocumentModel interface, OperationExtractDocumentModel, lets the algorithm extract the latest applied operation.
Afterwards, the ClientProxy instance is added to the RequestSerializer. Finally, a RequestForwarder for this ClientProxy is created. It is responsible for forwarding all requests that were issued by other clients and are to be sent to the opposite of the ClientProxy, namely the client itself. The RequestForwarder reads requests from the SynchronizedQueue and passes them to the ClientProxy by invoking $sendRequest(request)$.

3.5.2 Removing a client

When a client proxy is to be removed, firstly its connection to the net service is closed. Then the RequestForwarder, belonging to the client proxy, is shut down. Finally, the client proxy is removed from the RequestSerializer. The removal from the RequestSerializer is more complicated. At the time of removal, there may still be a couple of requests which have been issued by the client to be removed. Therefore, the client proxy may not be removed until all these requests have been processed, i.e. transformed and sent to all other clients. This is done by remembering the number of requests in the request queue, say r. After r requests have been processed from that moment on, the client proxy can savely be released. This mechanism is achieved by a helper class named Counter.

3.5.3 Receiving and sending a request

The received request is passed from the network layer to the corresponding ClientProxy instance. The ClientProxy itself only forwards the request to the SynchronizedQueue, namely request queue. Actually all ClientProxy's forward their requests to the same synchronized queue. By that, a global serialization is achieved. From the request queue, the request is read out by the RequestSerializer thread. The request is then passed to its corresponding client proxy for transformation. Afterwards, the transformed request is distributed to all other clients. This is done by first calling $generateRequest(operation)$ on each ClientProxy's algorithm and then adding the returned request to the outgoing queue of the ClientProxy. From there, the RequestForwarder will read out the request and pass it to the ClientProxy which in turn forwards it to the net service. Finally, the net service transmits the request to the client.
The SynchronizedQueue's enable us to have a flexible interaction of the different components such as ClientProxy, RequestSerializer, RequestForwarder. No component blocks except the ones which read from the queue, e.g. RequestSerializer, RequestForwarder.

3.6 Undo/Redo

For the undo implementation, we worked closely with the paper "Reducing the Problems of Group Undo" from Ressel et al., adapting the *adOPTed* undo for the *Jupiter* algorithm. The implementation of undo functionality has started but has not finished by the time. The undo works in simple cases both for characterwise and stringwise operations. But it fails in more complex scenarios (cf. classes $JupiterCharUndoTest.java$ and $JupiterStringUndoTest.java$, respectively. At first, we assumed that we could implement the undo with some auxiliary data structures and would not have to build the whole state space. In fact, it worked for example 3 as well as for example 4 (known as the "del-del-undo-puzzle") taken from the above paper, but it finally failed at the "order puzzle" in figure 6 of the same paper. The "order puzzle" showed us that the auxiliary data structures would not be sufficient for more complex undo scenarios and that we would need to implement the whole state space as is the case in the *adOPTed* algorithm and undo, respectively. Therefore we will not further describe the undo implementation we have done so far as much of it will become unusable as soon as we implement the state space. Building the state space means keeping track of every operation and calculating all possible transformation paths.

Nevertheless, two issues will remain. Firstly, each operation has a method $isUndo()$ to detect undo operations and to treat them separately. Secondly, the UndoManager class is used to manage undo and redo requests.

3.7 Tests

Testing the algorithm implementation was considered a very high priority. Without a correctly working algorithm, it will be difficult or even impossible to create a collaborative text-editor. Because of the significance of this algorithm we invested a lot of time to test it.

3.7.1 Testframework

In the literature about operational transformation, so called puzzles are omnipresent. These puzzles are exact specification of the order of events, like the generation of a request or the reception of a request. We were soon convinced that in order to test our algorithm it would be possible to have a testframework that accepts such a puzzle (called scenario) as input and replays the sequence of events on a set of algorithms. To find out more about this testframework, see the document emphReport Implementation Testframework.

3.7.2 Test Cases

We gathered many test cases form publications about operational transformation algorithms. The puzzles presented in these papers often represent particularly tricky issues that other algorithms failed to resolve. Here is a list of test cases that were taken from papers:

- Figures 3, 4 and 5 in "Proving Correctness of Transformation Functions in Real-Time Groupware", Imine et al.
- "C2 puzzle P1" and "C2 puzzle P2" (figures 3 and 5, respectively) in "Achieving Convergence with Operational Transformation in Distributed Groupware Systems", Imine et al.
- "A scenario of divergence and ERV" and "A divergence and ERV scenario of IMOR" (figures 1 and 12, respectively) in "Preserving Operation Effects Relation in Group Editors", Du Li and Rui Li
- figure 5 in "Concurrent Operations in a Distributed and Mobile Collaborative Environment", Suleiman et al.
- the "dOPT puzzle" (figure 2) in "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, Achievements", Sun et al.

Our algorithm is capable of solving these complex puzzles. This is a step in the right direction, but that alone is not enough. So we devised additional test cases that cover trivial as well as complex scenarios.

3.7.3 JUnit Tests

It is one of our goals to have a good test coverage with *JUnit* tests.

Bibliography

[1] Rul Gunzenhäuser Matthias Ressel. Reducing the problems of group undo. *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, 1999.

[2] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. *roceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, 1995.

[3] Ali A. Zafer, Clifford A. Shaffer, Roger W. Ehrich, and Manuel Perez. Netedit: A collaborative editor (master thesis). 2001.