

unDOit, an undo algorithm in a collaborative editor for a Native Text Database System

Diploma Thesis in Computer Science

Submitted by **David Weber**
Zurich, Switzerland

Register No. **97-923-296**

**Department of Informatics
University of Zurich
Prof. Klaus R. Dittrich**

Supervisor **Dr. Thomas B. Hodel-Widmer**

Date of submission **May 18th, 2005**

Abstract

The TeNDaX research project is dealing with a new approach to process text using database technology. When editing a document in the TeNDaX client program a lot of meta data is generated. Not only the typed text is stored in the database, but also who, when and where someone is doing something. Furthermore, it is kept track when someone is reading or from where someone is copying and pasting text into a document. These pieces of data are collected automatically and continually. The accumulation of these facts, which we call meta data, offers new possibilities for undo/redo.

The demand on an undo in a collaborative editor is significantly higher and more complex than the demand on an undo in a single-user editor. It has to be distinguished between the undoing of the last operation of any user and the undoing of the last operation of each individual user. Moreover, numerous conflicts can occur, which are a result of the dependencies between operations of different users.

In this thesis, within the scope of the TeNDaX project, the implemented undo/redo framework is extended in order to include several available meta data in the process. The objective is to enhance the existing concept in matters of stability, usability and performance and to ensure the availability of a working and in the TeNDaX research project embedded prototype.

Zusammenfassung

Das TeNDaX Forschungsprojekt beschäftigt sich mit einem neuen Ansatz, Datenbanktechnologie für die Textverarbeitung zu nutzen. Während der Bearbeitung eines Dokumentes im TeNDaX Client-Programm werden viele Meta-Daten generiert. Nicht nur der eingegebene Text, sondern auch wer, wann und wo jemand editiert, wird in der Datenbank gespeichert. Ebenso wird festgehalten, wann ein Dokument von wem gelesen wird und woher jemand Text in ein Dokument kopiert. Diese Daten werden ständig automatisch gesammelt. Die Ansammlung dieser Einzelheiten, die wir Meta-Daten nennen, eröffnet uns neue Möglichkeiten für das Zurücksetzen und Wiederherstellen von Operationen (Undo/Redo).

Die Anforderungen an ein Undo in einem kollaborativen Editor sind bedeutend höher und komplexer als die Anforderungen an ein Undo in einem Editor, der lediglich für einen Benutzer ausgelegt ist. Es muss zwischen dem Zurücksetzen der letzten Aktion eines beliebigen Benutzers und dem Zurücksetzen der letzten Aktion eines jeden einzelnen Benutzers unterschieden werden. Ausserdem können die Abhängigkeiten zwischen den Operationen unterschiedlicher Benutzer zu zahlreichen Konflikten führen.

In dieser Diplomarbeit soll die bestehende Undo/Redo Lösung im Rahmen des TeNDaX Projektes durch die Verwendung zusätzlicher Meta-Daten erweitert werden. Das Ziel dieser Arbeit ist, das bestehende Konzept in Hinsicht auf Zuverlässigkeit, Benutzerfreundlichkeit und Leistung zu verbessern und die Umsetzung eines funktionsfähigen Prototypen im TeNDaX Forschungsprojekt zu gewährleisten.

Computer Science is no more about computers
than astronomy is about telescopes.

- E. W. Dijkstra

Acknowledgment

I would like to gratefully thank the following people:

My parents, Hubert and Heide-Maria Weber for giving me the opportunity to attend the university and supporting me during my studies

Stefania Leone, Simone Mathieu and my father for proof-reading my thesis

Dr. Thomas B. Hodel-Widmer, my supervisor, for the support, which helped me to finish this thesis

Prof. Dr. Klaus R. Dittrich, the head of the Database Technology Group at the Department of Informatics

All of the people who were around me in the last six months, including my friends and the students working at the department of informatics

Contents

1	Introduction	1
1.1	Outline	1
1.2	Starting Point and Problem Formulation	2
1.2.1	Fundamental Problems	2
1.3	Objectives	4
1.4	Proceeding	4
1.5	Research	4
1.6	Content Overview	5
I	Preliminaries	6
2	Theory of Collaborative Undo / Redo	7
2.1	Notions	7
2.1.1	Undo	8
2.1.2	Redo	8
2.1.3	Groupware Systems	9
2.1.4	Concurrency Control	10
2.2	Formalized Examination of an Undo Effect	10
2.2.1	Formalized Basic Requirement of an Undo Effect	11
2.2.2	State of a Document	12
2.2.3	Undo Effect	13
2.2.4	Undo Property	14
2.3	Summary	15
3	Related Work	16
3.1	Former Undo/Redo Framework	16
3.2	Operational Transformation	17

3.2.1	Causality Preservation	18
3.2.2	User Intention Preservation	18
3.2.3	Convergence	19
3.3	Concepts of Collaborative Text Editors	19
3.3.1	Single-User Undo	20
3.3.2	Multi-User Undo	20
3.4	Summary	22
II	Concept	23
4	TeNDaX Approach	24
4.1	Text Editing as real-time Transaction	24
4.2	TeNDaX System	25
4.2.1	TeNDaX Architecture	25
4.2.1.1	Presentation Layer	26
4.2.1.2	Business Logic Layer	26
4.2.1.3	Data Layer	26
4.2.2	Storage of Text	26
4.2.3	Deletion of Text	28
4.2.4	Client-Document Model	28
4.2.5	Document State	28
4.2.6	Updating System	29
4.2.7	TeNDaX Schema	30
4.2.8	TeNDaX Database Classes	32
4.2.8.1	CFileNode	32
4.2.8.2	CFile	32
4.2.8.3	CSession	32
4.2.8.4	CDocumentSession	33
4.2.8.5	CChar	33
4.2.8.6	CUndoHistory	33
4.2.8.7	CRedoHistory	33
4.3	TeNDaX Algorithms	33
4.3.1	Insert Algorithm	34
4.3.2	Remove Algorithm	35
4.4	Summary	37

5	Concurrency Control	38
5.1	Collaborative Conflicts	38
5.1.1	Puzzles	38
5.1.1.1	Ambiguous Reference	39
5.1.1.2	Insert-insert-tie Dilemma	39
5.1.1.3	Do-undo-pair Trap	39
5.1.1.4	Overlapping Deletes Puzzle or Del-del-undo Puzzle	39
5.1.1.5	Insert-delete-undo Dilemma	39
5.1.1.6	Copy-paste-insert Dilemma	40
5.1.1.7	Reference Character Not Existent Anymore	40
5.2	Issues	40
5.2.1	Ambiguous Reference	41
5.2.2	Insert-insert-tie Dilemma	41
5.2.3	Do-undo-pair Trap	41
5.2.4	Overlapping Deletes Puzzle or Del-del-undo Puzzle	41
5.2.5	Insert-delete-undo Dilemma	41
5.2.6	Copy-paste-insert Dilemma	42
5.2.7	Reference Character Not Existent Anymore	42
5.3	Conflict Resolution	42
5.3.1	Reference Characters	43
5.3.1.1	First Order Reference Characters	43
5.3.1.2	Second Order Reference Characters	44
5.3.1.3	unDOit Reference Character Model	44
5.3.2	Overview	46
5.3.2.1	Configuration Strategies	48
5.3.3	Insert Conflict Resolution	49
5.3.3.1	Insert Categories	50
5.3.3.2	FORC-L=0 and FORC-R=0	52
5.3.3.3	FORC-L=1 and FORC-R=1	53
5.3.3.4	FORC-L=1 and FORC-R=0	59
5.3.3.5	FORC-L=0 and FORC-R=1	61
5.3.3.6	Client Update	63
5.3.4	Remove Conflict Resolution	63
5.3.4.1	Non Consecutive Removal Not Allowed (ANCR=0)	64
5.3.4.2	Non Consecutive Removal Allowed (ANCR=1)	64
5.3.4.3	Client Update	65

5.4	Test Cases	65
5.4.1	Notions	65
5.4.1.1	State of a Document	66
5.4.1.2	Keyboard Insertion	66
5.4.1.3	Copy-Paste Insertion	66
5.4.1.4	Removal	67
5.4.1.5	Local Undo	67
5.4.1.6	Global Undo	67
5.4.1.7	Local Redo	67
5.4.1.8	Global Redo	68
5.4.1.9	Example	68
5.4.2	Test Cases for Insert Conflict Resolution	70
5.4.2.1	Insert 0-0	70
5.4.2.2	Insert 1-1	72
5.4.2.3	Insert 1-0	78
5.4.2.4	Insert 0-1	79
5.4.3	Test Cases for Remove Conflict Resolution	81
5.4.3.1	Remove A	81
5.4.3.2	Remove B	81
5.4.3.3	Remove C	82
5.4.3.4	Remove D	82
5.4.3.5	Remove E	82
5.4.3.6	Remove F	83
5.4.4	Test Cases for Puzzles	83
5.4.4.1	Ambiguous Reference	83
5.4.4.2	Insert-insert-tie Dilemma	85
5.4.4.3	Do-undo-pair Trap	85
5.4.4.4	Overlapping Deletes Puzzle or Del-del-undo Puzzle	88
5.4.4.5	Insert-delete-undo Dilemma	88
5.4.4.6	Copy-paste-insert Dilemma	89
5.4.4.7	Reference Character Not Existent Anymore	92
5.4.5	Test Case for Redo	94
5.4.6	Issues	94
5.4.6.1	Insertion in one Part	94
5.5	Summary	96

6	Formal Depiction	97
6.1	Notions	97
6.2	Insert Algorithm	102
6.3	Remove Algorithm	107
6.4	Undo Algorithm	112
6.4.1	First Step - Global and Local Undo	114
6.4.2	Second Step - Only Global Undo	114
6.4.3	Second Step - Only Local Undo	115
6.4.4	Third Step - Global and Local Undo	118
6.5	Redo Algorithm	120
6.5.1	First Step - Global and Local Redo	122
6.5.2	Second Step - Only Global Redo	122
6.5.3	Second Step - Only Local Redo	123
6.5.4	Third Step - Global and Local Redo	125
6.6	Insert Algorithm invoked from Undo/Redo	128
6.6.1	First Step - Initiation	131
6.6.2	Second Step - Identifying a Conflict	136
6.6.2.1	Inactive - Inactive (0 - 0)	137
6.6.2.2	Active - Active (1 - 1)	137
6.6.2.3	Active - Inactive (1 - 0)	139
6.6.2.4	Inactive - Active (0 - 1)	140
6.6.3	Third Step - Conflict Resolution	140
6.6.4	Forth Step - Completion	143
6.7	Remove Algorithm invoked from Undo/Redo	145
6.7.1	First Step - Allowing and Not Allowing Removal of Consecutive Characters	147
6.7.2	Second Step - Not Allowing Removal of Consecutive Characters	148
6.7.3	Second Step - Allowing Removal of Consecutive Characters .	152
6.8	Summary	159
III	Documentation	160
7	Implementation	161
7.1	Technology	161
7.1.1	Caché Objectscript	162
7.2	Overview	162

7.3	Responsibility of the Participating Classes	164
7.3.1	Undo / Redo	165
7.3.2	Insert / Remove	166
7.4	Summary	168
IV	Evaluation	169
8	Conclusion	170
8.1	Retrospection	170
8.2	Stability	171
8.3	Usability	172
8.4	Performance	173
8.5	Summary	173
9	Summary	174
10	Future Investigations	175
10.1	Possible Extended Functionality	175
10.1.1	Selective Undo	176
10.1.2	Regional Undo	176
10.1.3	Intelligent Undo	176
10.1.4	Time Slot Based Undo	177
10.1.5	User-specific Undo	177
10.1.6	Combination of Different Undo Types	177
10.1.7	Undo of Layout	177
10.1.8	Undo of Access Rights	177
10.2	Possible Enhancements	177
10.2.1	Usability	177
10.2.2	Performance	178
10.3	Summary	179
A	Algorithm Flow Charts	180

List of Figures

4.1	Ring of double linked Fields	27
4.2	Interlinked Characters	28
4.3	TeNDaX System	29
4.4	TeNDaX InsertCChars Method	34
4.5	TeNDaX RemoveCChars Method	36
5.1	First Order and Second Order Reference Characters	45
5.2	unDOit Reference Character Model	45
5.3	Simplified Overview	47
5.4	Simplified Overview - Insert Conflict Resolution	51
5.5	Reference Character Model for FORC-L=0 AND FORC-R=0	52
5.6	Reference Character Model for FORC-L=1 AND FORC-R=1 simple	54
5.7	Reference Character Model for FORC-L=1 AND FORC-R=1 not possible B_Y_E	54
5.8	Reference Character Model for FORC-L=1 AND FORC-R=1 not possible B_X_E	54
5.9	Reference Character Model for FORC-L=1 AND FORC-R=1 complex	55
5.10	Reference Character Model for FORC-L=1 AND FORC-R=0	59
5.11	Reference Character Model for FORC-L=0 AND FORC-R=1	61
5.12	ANCR=1	64
7.1	ORM - Overview	163
7.2	Undo/Redo Buttons	164
A.1	Flowchart Undo	181
A.2	Flowchart Redo	182
A.3	Flowchart InsertCChars Not From Undo/Redo	183
A.4	Flowchart InsertCChars From Undo/Redo Overview	184
A.5	Flowchart Detail Insert Conflict Resolution From Undo/Redo	185

A.6	Flowchart Detail Insert From Undo/Redo Category Update	185
A.7	Flowchart RemoveCChars Overview	186
A.8	Flowchart Detail RemoveCChars Not From Undo/Redo or ANCR=0	187
A.9	Flowchart Detail RemoveCChars From Undo/Redo and ANCR=1 . .	188
A.10	Flowchart Detail Remove Special Update	189

List of Tables

3.1	Overview of the diverse Undo Techniques	22
4.1	Content of Table CChar	31
4.2	Log Entries in the History	32
5.1	Insert Conflict Resolution Case Overview	58
5.2	Insert Conflict Resolution Possible Cases	58
5.3	Example Database Content after Operation 1	69
5.4	Example Database Content after Operation 2	70
5.5	Ambiguous Reference Database Content after Operation 1	84
5.6	Ambiguous Reference Database Content after Operation 2	84
5.7	Ambiguous Reference Database Content after Operation 3	85
5.8	Ambiguous Reference Database Content after Operation 4	85
5.9	Do-undo-pair Trap Database Content after Operation 1	86
5.10	Do-undo-pair Trap Database Content after Operation 2	86
5.11	Do-undo-pair Trap Database Content after Operation 3	87
5.12	Do-undo-pair Trap Database Content after Operation 4	87
5.13	Do-undo-pair Trap Database Content after Operation 5	87
5.14	Insert-delete-undo Dilemma Database Content after Operation 1 . . .	88
5.15	Insert-delete-undo Dilemma Database Content after Operation 2 . . .	88
5.16	Insert-delete-undo Dilemma Database Content after Operation 3 . . .	89
5.17	Insert-delete-undo Dilemma Database Content after Operation 4 . . .	89
5.18	Copy-paste-insert Dilemma Database Content after Operation 1 . . .	90
5.19	Copy-paste-insert Dilemma Database Content after Operation 2 . . .	90
5.20	Copy-paste-insert Dilemma Database Content after Operation 3 . . .	91
5.21	Copy-paste-insert Dilemma Database Content after Operation 4 . . .	91
5.22	Reference Character Not Existent Anymore Database Content after Operation 1	92

5.23 Reference Character Not Existent Anymore Database Content after Operation 2	92
5.24 Reference Character Not Existent Anymore Database Content after Operation 3	93
5.25 Reference Character Not Existent Anymore Database Content after Operation 4	93
5.26 Reference Character Not Existent Anymore Database Content after Operation 5	93
7.1 Changed and Newly Created Classes	168
8.1 Puzzle Enhancements	172
10.1 Outlook Overview	179

Chapter 1

Introduction

The TeNDaX (**T**ext **N**ative **D**atabase **eX**tension) research project¹ is dealing with a new approach to process text using database technology with the aim to improve word processing and document management. Within this project, the TeNDaX system, a collaborative, real-time and multi-user environment is developed that offers collaboration support, which includes functions such as text editing, awareness, fine-grained security, sophisticated document management, versioning, business processes, text structure, data lineage, metadata mining, multi-channel publishing and as well a basic framework for undoing/redoin text editing operations [Hodel-Widmer and Dittrich, 2005].

In this diploma thesis, within the scope of this project, the implemented undo/redo is extended in order to include several available meta data in the process and to enhance the existing system in matters of stability, usability and performance.

1.1 Outline

Nowadays, most of the word processing programs are single-user applications, although the collaborative work already has a high significance and will be even more significant in the future since more and more people across the world have to work together.

Therefore, applications are needed that enable a group of users to edit documents simultaneously. Such applications are known as groupware that changes the emphasis from using the computer to solve problems, to using the computer to facilitate human interaction [Ellis and Gibbs, 1989]. Along with the demand on collaborative

¹<http://www.tendax.net>

editors comes the demand on a reliable undo, because undo (generally the taking back of a previous action) is not just an additional feature but an inherent part of any editor that gives the user a certain confidence for his work. [Yang, 1988] points out the importance of undo: 'Most sophisticated interface systems should be provided with an undo support facility.'

The collaborative word processing applications that exist, only offer restricted possibilities for undoing operations of a user. What should be provided by a sophisticated collaborative undo is the distinction between the undoing of the last operation of any user and the undoing of the last operation of each individual user. The former is called *global undo* whilst the latter is called *local undo*. Furthermore, the numerous conflicts that are a result of the dependencies between operations of different users, have to be solved in a flexible and user-friendly way while the stability and performance have to be guaranteed.

The task to resolve these conflicts while providing the common needs of a collaborative editor is one of the major differences between an undo in a single-user editor and an undo in a collaborative editor. Once this task is done, the system can be enhanced with additional functionality without worrying about low-level conflicts.

1.2 Starting Point and Problem Formulation

This diploma thesis is based on a framework that was developed within the scope of the TeNDaX research project [Jost, 2004]. The underlying algorithm is called GDU algorithm². One part of the subject of this thesis is concerned with the fundamental problems of that framework, which is implemented in the TeNDaX basis system.

The basis system consists of a database and a client. A more detailed description of the TeNDaX system is given in section 4.2 on pages 25 et seqq.

1.2.1 Fundamental Problems

Correctness Dependencies between operations of different users in a collaborative editor can cause numerous conflicts that have to be resolved by the system in order to satisfy the user's intention.

Consider a document that contains the word 'Tendax'. If a user (user 1) deletes the character 'n' at position three, the resulting string is 'Tedax'. Then, if another user (user 2) deletes the 'd', which is now located at position three

²GDU algorithm: German for Generic Database based Undo algorithm (**G**enerischer **D**atenbankbasierter **U**ndo **A**lgorithmus).

as well, the document content is 'Teax'. If user 1 afterwards wants to take back his removal of character 'n' and therefore triggers a local undo and then user 2 in the same way wants to take back his removal of character 'd', the resulting character order consequently would be 'Tendax'. But how does a system know where it has to insert the characters in such a case? It would also be possible that the system reinserts the character 'n' at position three of the string 'Teax', which leads to the string 'Tenax' and afterwards reinserts 'd' at position three what consequently results in the document content 'Tednax'. It is evident that in this case the resulting string 'Tednax' was certainly not the intention of the two users when they triggered a local undo. Such a case is called a *conflict* in a collaborative environment.

For a system, the user's intention and thus the correct insertion is not obvious in this case. Therefore, different approaches exist to store relevant data and control such occurring conflicts to ensure a conflict resolution as correct as possible.

Stability In collaborative systems, conflicting operations occur that can affect the stability. An undo respectively redo operation in a collaborative editor has to take back operations that possibly were executed at an utterly different document state. Thus, all special cases, which can occur have to be considered to ensure a stable system.

Performance A collaborative editor is a highly interactive system. Consequently, the performance of an undo/redox framework has to be high, since a user does not want to wait for his undo operation to be performed.

Limitation Desirable would be an undo algorithm that is adjustable to react as user-friendly as possible, but there exist limitations in matters of usability. Consider the example above. If it is not apparent for a system what the correct operation is, i.e. where to reinsert the character 'd' in the string 'Tenax', an undo algorithm has to decide based only on indications how this conflicting situation can be satisfyingly dissolved. Different possibilities are imaginable. E.g. it would be possible either to reinsert 'd' on the left or on the right side of character 'n'.

A limited undo algorithm would just reinsert the character according to its definite rules, but a user-friendly algorithm that goes beyond this limitation would be configurable how to react in such a situation or even would ask the user for a decision.

1.3 Objectives

The goal of this diploma thesis is to enhance the existing framework in matters of stability, usability and performance and to ensure the availability of a working prototype, which is embedded in the TeNDaX research project.

The following results should be achieved:

- Enhancement of the undo concept concerning conflict resolution, flexibility and usability
- Implementation of the concept within the scope of TeNDaX
- Ensure the stability of the implementation
- Ensure the deterministic behavior
- Enhancement of the performance
- Detailed depiction of the conflict resolution and the appropriate algorithms

1.4 Proceeding

In a first phase of this thesis, known techniques for undo/redo are studied. Then, new ideas based on the available meta data are collected and the feasibility of this ideas are critically evaluated in a technical way, as well as in the sight of usability. In the next phase the implemented undo/redo is analyzed in order to fulfill the requirements for stability, usability and performance.

Then, the conflict resolution is enhanced to ensure a more powerful algorithm that gives the possibility to configure the system.

1.5 Research

The purpose of the research was to become acquainted with the area of undo. Several frameworks and algorithms were studied in order to get a detailed knowledge of various undo techniques. Articles, papers and other work were primarily searched at the following sources:

- ACM (Association for Computing Machinery), <http://www.acm.org>
- CiteSeer, <http://citeseer.ist.psu.edu>

- IEEE (Institute of Electrical and Electronics Engineers), <http://www.ieee.org>
- Internet search engine Google, <http://www.google.com>

Besides a general research of work concerning the undo area, it was especially searched for information that describe occurring conflicts, concurrency control and test cases in collaborative systems for multi-user operation.

The search was concentrated on the terms 'collaborative', 'concurrency', 'conflicts', 'control', 'multi', 'puzzle', 'three', 'undo' and 'user'.

While there exist plenty of information about undo in collaborative systems, the number of sources that describe concrete multi-user conflicts and appropriate test cases is rather limited. No precise information could be found about a distinction between two-user, three-user and multi-user conflicts and a dependent increasing complexity. It can be concluded that either there is no difference between a two-user, a three-user and a multi-user conflict and that this fact is just too obvious, or that the research in this area is not that advanced or that the search terms were not correct.

1.6 Content Overview

In the first part of this thesis, the basic notions and principles in the area of undo/redo as well as the related work are presented. The second part covers the essential topics, which are concerned with the TeNDaX basis system, the concurrency control in the presented undo/redo framework and the formal depiction of all algorithms used to realize the concept. The chapter concurrency control includes a collection of test cases. In a third part, the documentation of the implemented framework is provided. Finally, the achievements, an overall summary and further ideas and suggestions are presented in the fourth part. In addition, the appendix contains flowcharts, which illustrate the algorithms used for the undo/redo framework.

Part I

Preliminaries

Chapter 2

Theory of Collaborative Undo / Redo

The tendency for almost any form of work goes from local to global. The need for distributed applications increases and more and more people have to work together, but at different locations. Thus, the request for applications that provide collaborative editing is obvious.

Undo is usually known as a command in word processors and text editors. It retracts changes in a document and reverts it back to an older state. [Abowd and Dix, 1992] regard undo not just as a system command to be implemented, but as a user intention to be supported by the system and according to [Berlage, 1994], an undo facility allows a user to work with an application without fear, knowing that any mistake can be undone. Therefore, the undo should be as complete as possible, i.e. the most actions should be undoable and it should be possible not only to undo the last action but wherever applicable any action. In addition, the undo algorithm should be predictable even by novice users. A simple but powerful undo is desirable, which is easy to use in stressful situations and of course is undoable by itself.

2.1 Notions

Prior to the formalized examination of the undo effect it is described briefly what is understood by undo, redo, groupware systems and concurrency control.

2.1.1 Undo

The notion *undo* generally means the taking back of a previous action, but in the context of collaborative editing undo has to be determined more precise.

It has to be distinguished between single-user undo and multi-user or group undo. The single-user undo is implemented in applications that allow only one user to edit a document at the same time, like OpenOffice.org¹. A multi-user undo is required if a document can be edited by two or more users simultaneously. In this case, the undo should be capable to manage the superior complexity. Conflicts between operations of different users can arise that are not easy to solve. It also has to be distinguished between a *local undo* and a *global undo*, which enables a user to undo his last action (local undo) instead of undoing the last action of any user (global undo). [Mancini et al., 1996] distinguishes two different definitions of the collaborative undo:

Definition 1 Undo is a system function that allows the user to reach **the** previous state.

Definition 2 Undo is a system function that allows the user to reach **any** previous state.

In the TeNDaX research project we follow the second definition, i.e. we aim to realize an undo algorithm that is capable of reaching any previous state even until the creation of a document. The database based approach of TeNDaX is a great advantage but a solid resolution of the occurring conflicts is still demanding. In this thesis an algorithm is presented that is capable of reaching stepwise any previous state in a document adjusted to the modified situation, which is enabled by a reliable conflict resolution.

2.1.2 Redo

The possibility to undo the effect of an operation in a document and to reach a former state is an important functionality. If an undo has lead to an undesirable document state, it should be possible to take back the undo itself. Therefore, the *redo* is needed. In a single-user undo the redo is just regarded as the inverse of the undo. In a collaborative system, the redo is more than that. There should also be a distinction between a global and a local redo, which offers the possibility to either take back the last global undo or the last local undo. Thus, there exists a

¹<http://www.openoffice.org>

causal dependence between undo and redo, which says that an undo must have been performed in order to perform a redo [Mancini et al., 1996].

Definition 3 Redo is a system function, which allows the user to recover **the** past state removed by the previous undo.

Definition 4 Redo is a system function that allows the user to recover **a** past state removed by any previous undo.

Comparable to the undo, the redo in the TeNDaX system satisfies definition four. The algorithm that is presented is capable of reaching stepwise any previous undone state in a document adjusted to the modified situation.

2.1.3 Groupware Systems

According to [Ellis and Gibbs, 1989] *groupware systems* are computer-based systems that give two or more users the possibility to solve a common task while accessing a shared environment. Furthermore, it is asserted that groupware systems require fine-granularity sharing of data and fast response times.

In [Ellis and Gibbs, 1989] it is not only distinguished between groupware systems and other multi-user systems that provide little notification, but it is also distinguished between non-real-time groupware systems and real-time groupware systems. Moreover, it is stated that real-time groupware systems are characterized by the following:

Highly interactive Response times must be short

Real-time Notification times must be comparable to response times

Distributed In general, one cannot assume that the participants are all connected to the same machine or even to the same local area network

Volatile Participants are free to come and go during a session

Ad hoc Generally, the participants are not following a preplanned script; it is not possible to tell a priori what information will be accessed

Focused During a session, there is high degree of access conflict as participants work on and modify the same data

External channel Often, participants are connected by an external (to the computer system) channel such as an audio or video link

2.1.4 Concurrency Control

When working with groupware systems, operations of the different users can cause several conflicts. The part of the system that resolves these occurring conflicts is called *concurrency control*². In this thesis, the entire chapter 5 on pages 38 et seqq. focuses on this central topic.

2.2 Formalized Examination of an Undo Effect

Before it is explained how an undo operation works, it has to be defined what an *undo effect* is. Consider an object, e.g. a character string, and an operation, e.g. an insertion of a character in a character string. If the operation is performed on that object, it has a certain effect, e.g. the insertion of a character between two other characters of the string. From an undo it is expected that it undoes a certain operation respectively that it undoes the effect of the operation that should be undone. Thus, the object state should be the same as it was before the execution of the operation.

If an undo is implemented in a single-user editor where the operations are performed sequentially, it is no problem to undo the last effect of an operation and to restore the former object state. But if an undo shall be implemented for a distributed editor, things are not that easy anymore because the state of an object might have changed in between, since operations can be undone in a nonlinear order. Thus, it is impossible to restore the former state of the object with just undoing the effect of one operation, but anyhow the effect of the appropriate operation has to be undone despite all other changes of the object state [Sun, 2002].

²*Conflict resolution* is used as a synonym for concurrency control.

2.2.1 Formalized Basic Requirement of an Undo Effect

The Partial Ordering of [Lamport, 1978] is a basic concept for the order of operations in distributed systems:

Definition 5: Definition of relation ' $\rightarrow$ '

Given a set of processes^a i and j and a set of operations^b O_a and O_b of a system, $O_a \rightarrow O_b$ is the smallest relation satisfying the following three conditions:

1. O_a and O_b are operations in the same process ($i = j$) and O_a comes before O_b , or
2. O_a and O_b are operations in different processes ($i \neq j$) and O_a is executed in process j before O_b , or
3. It exists an operation O_x in a way that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$

^aA process might be a client in a client-server system.

^bAn operation might be an insertion or a deletion of a character in a document.

A related concept is the one about dependencies of operations:

Definition 6: Dependent and concurrent operations

If two distinct operations O_a and O_b exist

- O_b is said to be dependent on O_a if it applies that $O_a \rightarrow O_b$
- they are said to be concurrent or not dependent on each other if $O_a \not\rightarrow O_b$ and $O_b \not\rightarrow O_a$ what is expressed with $O_a \parallel O_b$

Another related concept is about the intention of a user respectively of an operation [Sun, 2002]:

Definition 7: Intention of an operation

The intention of an operation O is the intended effect of an operation on a certain object state S .

Another concept of [Sun, 2002] is about the consistency of a collaborative editing system:

Definition 8: Definition of the consistency of a collaborative editing system

A collaborative editing system is consistent if it meets all of the following three conditions:

1. Convergence:
If all sites/editors have executed the same set of operations, the object/document states of all replica are the same.
2. Causality-preservation:
For any pair of operations O_a and O_b , O_a has to be executed before O_b at any site if O_b is dependent on O_a ($O_a \rightarrow O_b$).
3. Intention-preservation:
For any operation O is ensured that the effect of the execution of this operation O is the same as the intention of O at any state of the object/document.

These consistency conditions should be maintained not only for all insert and delete operations but also for any undo operations in collaborative editing systems.

2.2.2 State of a Document

Given an initial document state S_0 and a list of operations

$$L = [O_1, \dots, O_{i-1}, O_i, O_{i+1}, \dots, O_n]$$

After the execution of the operations in L , the document state S_n can be written as

$$S_n = S_0 \circ [O_1, \dots, O_{i-1}, O_i, O_{i+1}, \dots, O_n]$$

In collaborative editors that are not based on a database, the list of operations is stored in the history buffer³ of the appropriate editors and determines the actual document state [Sun, 2000, Sun, 2002]. Two lists of operations L_1 and L_2 are called equivalent, written as $L_1 \equiv L_2$, if they are executed on the same state S and produce the same document state, e.g. $S \circ L_1 = S \circ L_2$. In collaborative editors that are based on a database like TeNDaX, the centrally stored history in the database determines the document state of any editor that has opened the appropriate document.

³In collaborative editors that are not based on a database, each editor manages its own history buffer that stores the executed operations.

2.2.3 Undo Effect

$Undo(O_i)$ is a command to undo the operation O_i at the document state S_n . If a user executes an $Undo(O_i)$, he expects the document to be restored to a state as if the operation O_i never had taken place but all other operations. Based on the state operations of a document the *undo effect* can be defined as follows [Sun, 2002]:

Definition 9: Definition of an undo effect

Given a document state:

$$S_n = S_0 \circ [O_1, \dots, O_{i-1}, O_i, O_{i+1}, \dots, O_n],$$

the effect of an $Undo(O_i)$ on S_n is the transformation of S_n in a new state S'_{n-1} , which can be expressed as follows:

$$S'_{n-1} = S_0 \circ [O_1, \dots, O_{i-1}, O'_{i+1}, \dots, O'_n],$$

whereas O'_x , $x \in \{i+1, i+2, \dots, n\}$ is the operation that the system would have executed instead of O_x if O_i would not have been executed before.

A concrete example shall illustrate what a correct undo should be:

Example 1: Undo effect

Starting from a text document with an initial state $S_0 = \text{'abc'}$, three operations $O_1 = \text{insert}(2, X)$, $O_2 = \text{insert}(4, Y)$ and $O_3 = \text{delete}(3, b)$ are sequentially executed on the document. O_1 inserts 'X' at position 2, O_2 inserts 'Y' at position 4 and O_3 deletes 'b' at position 3, what generates the following sequence of document states:

1. $S_0 = abc$
2. $S_1 = S_0 \circ [O_1] = aXbc$
3. $S_2 = S_1 \circ O_2 = S_0 \circ [O_1, O_2] = aXbYc$
4. $S_3 = S_2 \circ O_3 = S_0 \circ [O_1, O_2, O_3] = aXYc$

Now $Undo(O_2)$ is executed on S_3 . According to the undo effect in Definition 9 the following document state should be achieved:

$$S'_2 = S_3 \circ Undo(O_2) = S_0 \circ [O_1, O'_3] = aXc,$$

because,

1. the effect of O_2 should be undone, i.e. the character 'Y' that was inserted by the operation O_2 should be deleted from the document;

2. the effect of O_1 should be kept, i.e. the character 'X' that was inserted by the operation O_1 should remain at the origin position of the document; and
3. the effect of O_3 should be kept as well, i.e. the character 'b' that was deleted by the operation O_3 should remain deleted. This accords to the document state S'_2 , $O'_3 = Del[2, b]$ what would have been O_3 if O_2 would not have been executed before.

It should be pointed out that the definition of the undo effect just says how the new document state should be after the execution of an undo on a certain document state but not how the new document state is generated.

2.2.4 Undo Property

In Example 1 it is asserted that after the execution of $Undo(O_2)$ on S_2 the document state S'_2 is not equivalent to any previous state S_i whereas $i \in \{0, 1, 2, 3\}$. Thus, the returning to a previous state can not be a necessary condition for the correctness of undoing an operation. Nevertheless, there is a necessary condition for the correctness of undoing a group of operations when returning to a previous state: If a group of operations has lead from one document state to the actual one, an undo of the operations that is executed in any order should restore the original document state. This condition can be defined as *undo property* [Sun, 2002]:

Definition 10: Definition of an undo property

Given a document state:

$$S_n = S_0 \circ [O_1, \dots, O_{i-1}, O_i, O_{i+1}, \dots, O_n],$$

The undoing of all operations between O_i and O_n in any order has to restore the documents to the previous state:

$$S_{i-1} = S_0 \circ [O_1, \dots, O_{i-1}]$$

This property is essential for any undo solution because it ensures that a document can be restored to any previous state through an undo of all operations that have been executed after that certain state, regardless of the order of those operations. The undo effect in definition 9 and the undo property in definition 10 together with the consistency conditions of definition 8 form a set of necessary correctness criteria for group undo solutions in general.

2.3 Summary

In this chapter, the notions *undo*, *redo*, *groupware systems* and *concurrency control* have been introduced. Furthermore, it was presented what is understood by an *undo effect* and an *undo property*.

In conformity with other definitions of undo, in the TeNDaX research project we aim to realize an undo algorithm that is capable of reaching any previous state even until the creation of a document, while the occurring conflicts are treated as a naturally-arising side effect of the collaborative process. Therefore, the TeNDaX system contains a control structure, the concurrency control or conflict resolution that decides how to handle arising conflicts.

Redo is regarded as an action that takes back an undo operation. Consequently, it exists a causal dependence between undo and redo, which says that an undo must have been performed in order to perform a redo.

The undo effect describes, how the new document state should be after the execution of an undo on a certain document state but not how the new document state is generated. If a group of operations has lead from one document state to the actual one, an undo of the operations that is executed in any order should restore the original document state, which is defined as the undo property.

Chapter 3

Related Work

A lot research has been conducted in the area of collaborative editing. Consequently, different approaches exist to manage the appropriate tasks. Depending on the chosen approach, the preservation of consistency faces different problems. Many approaches are based on a technique known as operational transformation. The TeNDaX system is in contrast not based on operational transformation and has its own technique for consistency maintenance (cf. chapter 4 on pages 24 et seqq.). The following sections present a brief overview of the former framework [Jost, 2004], operational transformation and some of the research that has been conducted in the area of collaborative editing.

3.1 Former Undo/Redo Framework

The former solution is a general framework that offers the possibility to trigger a local or global undo/redo. It is based on a database approach and designed on the concept of 'soft delete' what means that a document could be undone until its beginnings.

The solution is based on a database history, i.e. all transactions are logged in a chronological sequence. To undo an operation, these operations have to be reversible. Thus, for each operation there has to be an inverse operation that undoes the effect of the original operation. The information to build that inverse operation is persistently stored in the database. That's why operations of previous sessions can also be undone.

The framework dissolves the conflicts that appear but it is only possible to undo/redo text editing operations, i.e. it is possible to undo/redo an insertion or a deletion of a sequence of characters.

The GDU algorithm that was developed, has the following properties:

- All operations are logged in a chronological sequence in the central history.
- Any effect of an operation can be undone directly on the database with an appropriate method.
- For each operation that is logged it exists exactly one referenced character in the table CChar.
- The algorithm is generic and provides a global and a local undo.
- The new document state is propagated to all clients, which have opened the same document.

The former solution tries to dissolve all the occurring conflicts without the interaction of the user. If a conflict occurs that cannot be solved, the user is informed with an error message. Similar to the undo history it exists a redo history that contains log entries if an undo operation has been executed. The difference is that the entry that is logged is the inverse of the undo operation and that all log entries of the appropriate document are deleted if a user inserts or deletes a character in the document. Thus, a redo is only possible immediately after an undo has been executed.

3.2 Operational Transformation

In the area of real-time group editors, research in the past came up with a technique for consistency maintenance, which is known as *operational transformation*. This technique was pioneered by the GROVE¹ system in 1989 [Ellis and Gibbs, 1989]. Since then, this technique has been used and enhanced by several research teams around the world.

In this section, some of the basic concepts of operational transformation are described. For additional information the reader is referred to [Ellis and Gibbs, 1989] and [Ferri et al., 2004].

A groupware system consists of several participant systems, which are called sites (usually users with workstations that are connected to each other through a communication network) and of a set of parameterized operators. Each site consists of a site process, a site object and a unique site identifier. The object shared by the

¹GROVE: **GR**oup **O**utline **V**iewing **E**ditor

users, e.g. a document or simply a character string, is replicated at each site and can be changed by definite operations.

Operations are generated locally by the site process after a user has specified an operation, e.g. the deletion of a certain character. The operation is immediately executed on the local site to guarantee a minimum response time and is then broadcasted to all other sites. Thus, each operation that is executed locally on a site is executed on all other copies as well. This maintains consistency between the copies while locks are not necessary. If a site L receives an operation from a remote site R, it is checked if the sending site R executed operations, which have not yet been executed at site L and if site L has executed operations, which have not yet been executed at the remote site R. Depending on this information, operations are queued, transformed and executed or just executed on the local copy of the object.

If consistency maintenance of object copies is needed, there exist three constraints:

- Causality preservation
- User intention preservation
- Convergence

In the following, it is described what these notions stand for.

3.2.1 Causality Preservation

If an operation O_1 causally precedes another operation O_2 , then O_2 depends on the effects of operation O_1 . Causality preservation is achieved by using state vectors and timestamps and it ensures that all related operations are executed in the same order on each copy.

3.2.2 User Intention Preservation

If there exist two operations O_1 and O_2 and neither O_1 causally precedes O_2 nor O_2 causally precedes O_1 the two operations O_1 and O_2 are not causally related, i.e. O_1 and O_2 are not dependent on each other. This relation is called concurrent.

Thus, they can be executed in any order but the intention of the user who generated O_2 has to be preserved. Therefore, an operation might have to be transformed before it is executed. For a concrete example consider the character string 'efect' as a site object. The two operations $\text{insert}(p, c)$ and $\text{delete}(p)$ are available. Where p marks the position in a certain string and c marks a certain character. Now, if user 1 wants to insert the character 'f' on position 2 and user 2 on a different site wants

to attach a 's' at the end, the two concurrent operation are $O_1 = \text{insert}(2, 'f')$ and $O_2 = \text{insert}(6, 's')$. If O_2 is just executed after O_1 the result is 'effecst' because the position has shifted. This result was not the intention of user 2, thus O_2 has to be transformed before it can be executed. The solution of this problem is based on the use of operational transformation. I.e., each remote operation has to be transformed considering the modifications made by all operations that are executed before and which are not causally related.

3.2.3 Convergence

It is not enough to take causality and user intention into account. Moreover, two conditions, called C1 and C2 are needed to control convergence of the copies. C1 checks if two concurrent operations that are executed on the same state of an object but in a different order, have the same effect. The second condition C2 is used if there exists a transposition of an operation with a sequence of at least two concurrent operations to ensure that the transposition does not depend on the order of the concurrent operations.

For undo purposes an inverse of the original operation that shall be undone is generated, executed and broadcasted. The inverse operation is treated as a regular operation but additional conditions are needed to ensure the correctness of the operations [Ferri et al., 2004].

With operational transformation it is possible that two users simultaneously edit the same position in a document, which is not allowed in the TeNDaX approach. Therefore, the concurrency control faces different problems (cf. chapter 5 on pages 38 et seqq.).

3.3 Concepts of Collaborative Text Editors

There exist various undo techniques that were developed by research teams all over the world. It can not just be distinguished between single-user and multi-user concepts but between many different approaches within those two main classes. In this section the most important concepts and projects that have influenced the area of collaborative undo are briefly described but it is not the aim of this section to give a detailed overview of the solutions that exist. For this thesis, most of these concepts were studied to gain more insights and ideas for the enhancement of the former

undo/redo framework of TeNDaX. For a detailed description it is referred to the respective references and a more detailed overview is given in [Jost, 2004].

3.3.1 Single-User Undo

The simplest undo solutions are found in single-user applications but they provide basic techniques that are used as well for undo in multi-user application. Therefore, the most important techniques are presented in this subsection.

Single-Step Undo The *single-step undo* just allows a user to take back the last operation. Usually a redo of the last undo is also provided, but it is not possible to restore any other state of a document than the last one. There exist many editors that are built upon this technique, e.g. the editor ETUDE [Hammer et al., 1981].

Linear History Model and US&R Model In the *linear history model* a history list is implemented that stores all executed commands. Only the last command in the list can be undone. If an undo is triggered, the last command of the history list is removed and inserted into a redo list. Thus, it is possible to repeat the undo until the history list is empty. If a redo is triggered, the command is deleted from the redo history and appended to the history list. A new command is inserted into the history list without effecting the redo list. The *US&R model* (Undo-Skip and Remove model) by [Vitter, 1984] additionally provides a skip operation to skip redoing a command. The command that was skipped is not deleted but marked as skipped. Furthermore, the US&R model uses a tree structure instead of a linear list.

History Undo The *history undo* is similar to the US&R model but it treats an undo operation as a normal operation and appends it at the end of the history list.

3.3.2 Multi-User Undo

Ellis and Gibbs One of the earlier group editors, called GROVE, was developed by [Ellis et al., 1988, Ellis and Gibbs, 1989]. It was especially designed to support group editing.

Choudhary/Dewan and Berlage/Genau The undo of [Choudhary and Dewan, 1995] and the one of [Berlage and Genau, 1993] are

based on a history list and suggest the undoing of an operation by just executing its inverse operation that has to be executable in the current state. The undo of Berlage and Genau is implemented in the GINA² [Spence and Beilken, 1991] application framework.

Prakash and Knister The *selective undo* approach of [Prakash and Knister, 1994] is based on history undo but allows operations to be undone selectively. It is implemented in the editor DistEdit³ and was the first multi-user undo that was based on operational transformation. In contrast to other projects like GROVE, DistEdit is not an editor but a toolkit that can be used to connect different single-user editors to the task of group editing.

Ressel and Gunzenhäuser The undo of [Ressel et al., 1996, Ressel and Gunzenhäuser, 1999] is also based on operational transformation and is implemented in the adOPTed⁴ system.

Sun The concept of operational transformation is used by [Sun, 2000, Sun, 2002] as well. His framework REDUCE⁵ is based on an algorithm called ANYUNDO.

²GINA: **G**eneric **I**nteractive **A**pplication

³DistEdit: A **D**istributed Toolkit for Supporting Multiple Group **E**ditors

⁴adOPTed: **a**dapted **d**OPT for **e**ditors

⁵REDUCE: **R**Eal-time **D**istributed **U**nconstrained **C**ooperative **E**ding

3.4 Summary

In this chapter, an insight to operational transformation, one of the most used techniques for collaborative undo, was given. Furthermore, several projects that influenced the area of undo were briefly presented (cf. table 3.1).

<i>Concept</i>	<i>UndoTechnique</i>
Single-User concepts	Single-Step Undo
	Linear History Model and US&R Model
	History Undo
Multi-User concepts	Undo of Ellis and Gibbs
	Undo of Choudhary/Dewan and Berlage/Genau
	Undo of Prakash and Knister
	Undo of Ressel and Gunzenhäuser
	ANYUNDO of Sun

Table 3.1: Overview of the diverse Undo Techniques (own contribution)

Part II

Concept

Chapter 4

TeNDaX Approach

TeNDaX is a research project at the Department of Informatics at the University of Zurich. The aim of the project is to improve word processing and document management by making use of a database system.

The TeNDaX system is a collaborative, real-time and multi-user environment and offers collaboration support, which includes functions such as text editing, awareness, fine-grained security, sophisticated document management, versioning, business processes, text structure, data lineage, metadata mining and multi-channel publishing [Hodel-Widmer and Dittrich, 2005].

TeNDaX creates an extension of DBMS¹ to manage text. This addition is carried out 'cleanly' and the responding data type represents a 'first-class citizen' of a DBMS (like integers, character strings, etc.) [Abiteboul et al., 2003].

4.1 Text Editing as real-time Transaction

According to [Hodel and Dittrich, 2004b], word-processing applications should store data in a 'native' form in a database, in order that editing text is finally represented as a real-time transaction. Text editing refers to the following well-known operations: writing and deleting text (characters), copying and pasting text, defining text layout and structure, inserting tables, pictures and so on, i.e. all the actions regularly carried out by word-processing users.

'Text editing as a real-time transaction' means that editing text (for example writing a character or word, setting the font for a paragraph, or pasting a section of text) invokes one or several database transactions so that anything that is typed, appears

¹DBMS = **D**atabase **M**anagement **S**ystem

within the editor as soon as these objects are stored persistently in the database. The database schema and the above-mentioned transactions are created in such a way that anything can be done within a multi-user environment, as supported by database technology. As a consequence, many of the actions that can be performed on a DBMS (such as data organization and querying, recovery, integrity and security enforcement, multi-user operation, distribution management, uniform tool access etc.) are now, by means of this approach, also available for word-processing [Hodel and Dittrich, 2004a].

4.2 TeNDaX System

In this section the TeNDaX system is briefly described especially in regards to what is relevant for the implementation of undo. For a closer look, the reader is referred to [Hodel-Widmer and Dittrich, 2005] and [Dubacher, 2003] where the TeNDaX basic system is well described and documented.

As mentioned before, TeNDaX is an approach to store document content and layout in a database. TeNDaX is not, in itself, a database; instead, it provides uniform coordinated access to a native text extension from current database management terms of transactions [Hodel and Dittrich, 2004b].

For the implementation of undo the following aspects of the TeNDaX system are of interest:

- Storage of text
- Deletion of text
- Updating system
- Client-Document Model

Before we explain these aspects in detail, an overview of the architecture of TeNDaX is given.

4.2.1 TeNDaX Architecture

The TeNDaX system is built on a three-tier architecture [Hodel-Widmer and Dittrich, 2005]. Each manipulation within a document, such as editing or defining business processes and structures, is ultimately a database transaction. These operations can be handled synchronously or asynchronously.

4.2.1.1 Presentation Layer

An editor in JavaTM implements the presentation layer. The aim of this layer is the integration in a well-known word-processing application such as OpenOffice.org.

4.2.1.2 Business Logic Layer

The business logic layer represents the interface between the database and the word-processing application. It consists of the following components:

- The *application server* enables the users to edit text within the database environment and takes care of awareness, security, document management etc., all within a collaborative, real-time and multi-user environment.
- The *real-time server component* is responsible for the propagation of information between all of the connected editors.
- The *web server* enables a web-based connection to the database for functions like login, browse, search and open document.

4.2.1.3 Data Layer

The storage engine primarily stores the content of documents within the database. All related data (metadata) is also stored within the database (information about users, dynamic folders, redo and undo information etc.).

The data layer and the business logic layer are realized with InterSystems'² post-relational database Caché. Caché provides both an object-oriented as well as a relational database interface. More detailed information on Caché will be provided in section 7.1 on page 161.

4.2.2 Storage of Text

In TeNDaX each character is stored in the database as an object with several properties. All characters in a document build a ring of double linked fields, i.e. each character holds a reference to the previous and to the next character in the document. This is a major difference to other text editor architectures where characters are primarily stored with their position in the document. Figure 4.1 shows in the central ring the individual characters, which are interlinked through two properties.

²<http://www.intersystems.com>

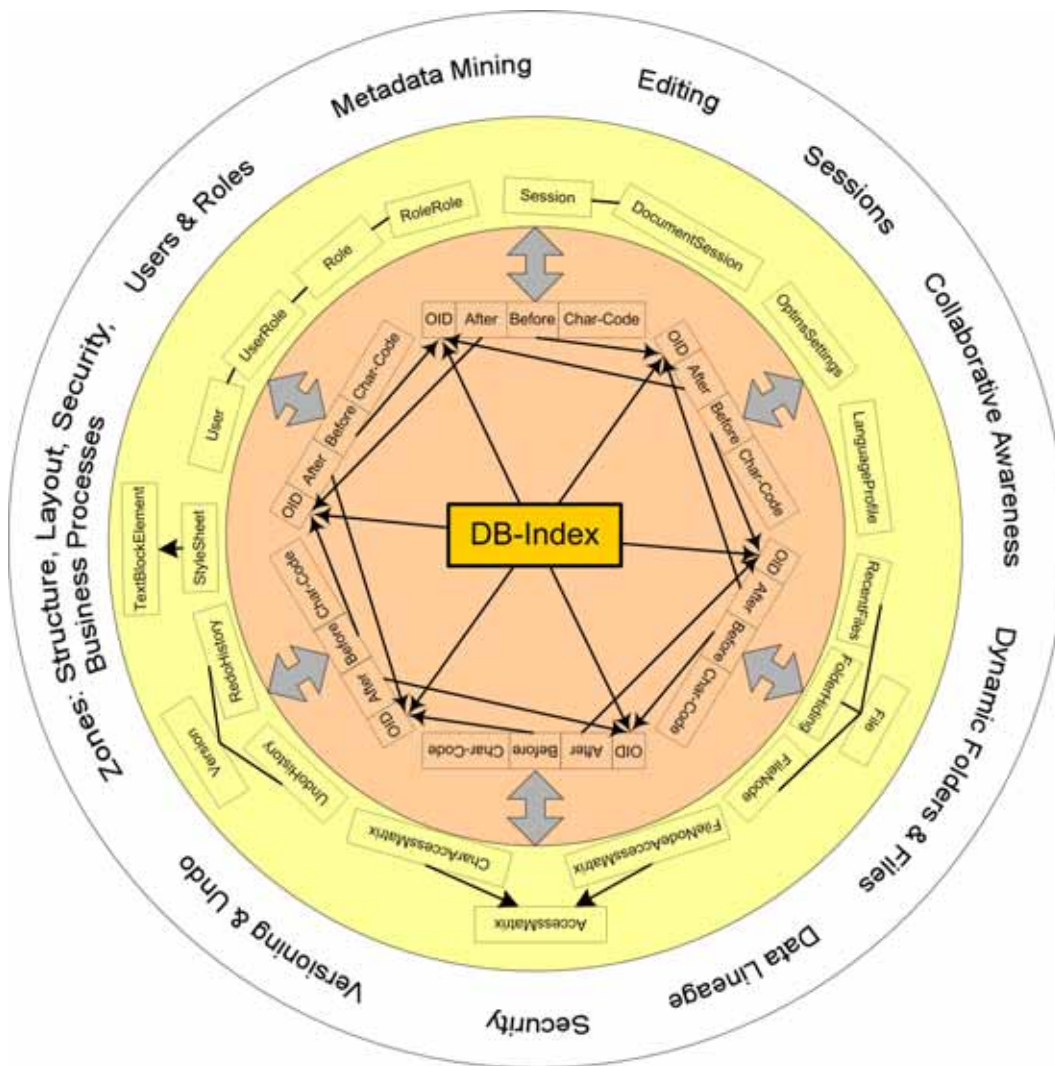


Figure 4.1: Ring of double linked Fields [Hodel-Widmer and Dittrich, 2005]

The property **After** holds the `OID`³ of the previous character object while the property **Before** holds the `OID` of the next one (cf. figure 4.2). The field properties of the previous respectively next character are indirectly accessible through the database index since one can access a database object if the database index is known [Dubacher, 2003]. Each character is stored as a tuple in the database. In TeNDaX the class respectively table that stores those tuples is called `CChar`, which will be introduced in section 4.2.8.5 on pages 33 et seqq.

³OID = **O**bject **I**dentifier

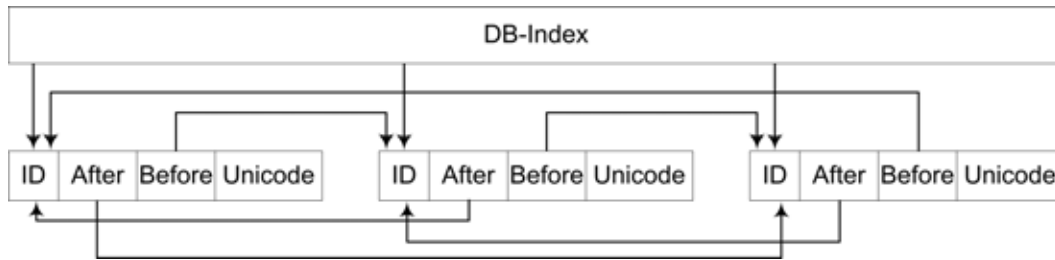


Figure 4.2: Interlinked Characters [Hodel-Widmer and Dittrich, 2005]

4.2.3 Deletion of Text

If a character is deleted, it just means that the character is set to an inactive state. The principle that lies behind this idea is called 'soft delete'. The reference characters before and after the appropriate character are kept while the character itself does not appear as active in the document anymore. This principle is very important not just for the undo concept but as well for the version control concept [Rapino, 2004] because no character is definitively deleted and therefore can be restored at anytime if needed. The characters that are situated before and after the deleted character are interlinked to each other to ensure that the ring of double linked characters is valid at any time. One disadvantage of this principle is that any character, which is inserted, remains in the document and with it also in the database. Thus, the database size grows faster than with a definitive deletion of each character.

4.2.4 Client-Document Model

The TeNDaX system consist of a central database system, a central message propagator and several distributed clients (cf. figure 4.3). The content of a document is stored in the database where each character is represented as a tuple. The client serves as the user interface and displays the document content and its structure. Each client holds a replica of the document content that is stored centrally in the database system.

4.2.5 Document State

For the TeNDaX system the definition of a document state has to be extended. Due to the ring of double linked fields a document always consists of at least two special characters:

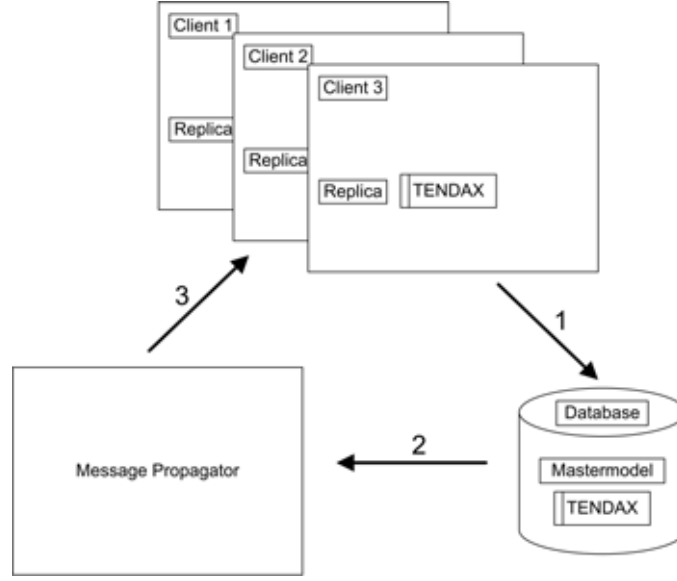


Figure 4.3: TeNDaX System (own contribution)

- The beginning of the document, which shall be abbreviated with either 'BOF' or ' β '
- The end of the document, which shall be abbreviated with either 'EOF' or ' ϵ '

In the client document model those two special characters are not visible for the user but they are essential since the undo algorithm operates almost exclusively in the database.

Therefore, the initial document state S_0 is never empty but always consists of those two special characters.

4.2.6 Updating System

Actually, the clients and the database system are separated processes but they have to communicate constantly because the changes in the distributed clients have to be propagated to the central database system and then again be propagated to all other clients, which have opened the same document. The task of communicating between the database system and a client is undertaken by the *message propagator*. The only operations that are possible are insert, remove and update operations. If one of these actions is triggered in a client, the appropriate method in the exchange interface is invoked. This interface connects the client to the database, thus the appropriate method on the database side is invoked. The database system completes the necessary operations and informs the message propagator about the changes.

The message propagator on its part informs the clients, which have opened the relevant document.

Both a synchronous and an asynchronous communication between clients and database exist. Whereas with the synchronous approach the client has to wait until the database system confirms the operation, the client can proceed with the asynchronous approach.

4.2.7 TeNDaX Schema

There exist several approaches to log events in order to implement an undo / redo solution like the ones on the basis of single or multiple timestamps that are compared in [Jost, 2004], but the implemented undo/redo in the TeNDaX system uses a history where all events are stored. On the basis of this history, effects of an action can be undone. The basic idea to maintain such a history is to log all transactions in a chronological sequence. The history then contains all operations that have changed the document state. The operations are stored in the same order as they appeared. At that time, the database based collaborative editor, as it is implemented in the TeNDaX system, advantages. In editors that are not based on a database, only the operations that have been executed since the document was opened can be logged. In most of the cases the number of entries in the history is even limited. The log entries in TeNDaX are persistently stored in the database what makes it possible to access the document history even if the document has not been edited for weeks. To take back operations they have to be reversible. I.e., for each operation there has to be an inverse operation that takes back the effect of the operation (cf. section 2.2.1 on page 11). In a text editor, the according operation of insert is delete. The inverse operation typically is dependent on the former document state as it is shown in section 2.2 on page 10. This leads to reflections about what information is needed to undo an operation. As shown in section 4.2 on page 25 a character in the client document model is defined through its value and its position relative to the linked characters in the document. In the database the ID⁴ of the inserted character suffices. The history based undo/redo solution that is implemented in TeNDaX operates in the first instance on the database, therefore a character in the history is referenced through its ID. As another necessary information the kind of operation (insert or delete) has to be logged in the history in order to generate the appropriate inverse operation. Furthermore, the references to the user and the document are

⁴ID = **I**dentification; ID is used as a synonym for the OID.

needed due to the collaborative environment. The following example illustrates this circumstance.

Consider the insertion of the character string 'Tendax' in an empty document. The single characters are stored in a ring of double linked fields through the interface between client and database in the database table **CChar** (cf. table 4.1). The order is defined through the ID and the fields **After** and **Before**. The other fields store the character value (**CharacterValue**⁵), the document to which the character belongs (**FileId**), the user who has inserted the character (**UserFk**) and the status⁶ that the character has in the document (**IsActive**; **IsActive=1** for active characters, **IsActive=0** for inactive characters). At the same time, the characters are also

CharacterValue	ID	After	Before	FileId	UserFk	IsActive
T	35	1	36	15	20	1
E	36	35	37	15	20	1
N	37	36	38	15	20	1
D	38	37	39	15	20	1
A	39	38	40	15	20	1
X	40	39	2	15	20	1
	1	2	35	15	0	0
	2	40	1	15	0	0

Table 4.1: Content of Table CChar (own contribution)

logged in the history, which is realized in the database table **CUndoHistory**. The following information is stored in a **CUndoHistory** tuple: The highest ID marks the last executed operation; the **ActionId** defines the kind of operation whereas a 1 stands for an insert operation and a 2 stands for a delete operation. In addition, the ID of a character (**CharId**), the document on which the operation has been executed (**DocumentId**) and the user that has triggered the operation (**UserId**) are logged (cf. table 4.2).

Based on this information it is now possible to take back any operation of a user that has been executed on any document.

Using the presented solution, any operation on a document can be chronological retraced. Furthermore, the implementation does not limit the number of possible

⁵In fact the Unicode of the appropriate character is stored in the database.

⁶The status variable shows if the character is active (1) in the document or if the character has been deleted (2). Cf. 'soft delete' in section 4.2.3 on page 28.

ID	ActionId	CharId	DocumentId	UserId
1	1	35	15	20
2	1	36	15	20
3	1	37	15	20
4	1	38	15	20
5	1	39	15	20
6	1	40	15	20

Table 4.2: Log Entries in the History (own contribution)

undo steps. Given that the information is stored persistently in the database, operations of former sessions⁷ can be undone. The concept is smoothly extensible due to the fact that a new kind of operation (other than insert or delete) can be defined, which makes it possible to log and undo any kind of character formatting operation. The TeNDaX version control [Rapino, 2004] as well uses this information in the history to reconstruct an old version of a document.

4.2.8 TeNDaX Database Classes

In this section, a brief overview of the most important classes of the TeNDaX database schema is given.

4.2.8.1 CFileNode

The class **CFileNode** holds the folder structure. Each instance of the class represents either a static or dynamic folder or a document [Hacmac, 2003, page 49].

4.2.8.2 CFile

The **CFile** class represents a text document and includes properties like author, file size, last modification etc. The properties **StartCCharId** and **EndCCharId** are references to the start and the end character of the document [Hacmac, 2003, page 66].

4.2.8.3 CSession

The **CSession** class handles the session management. As soon as a client connects to the server, an instance of **CSession** is created [Dubacher, 2003, page 66].

⁷Session means in this case the time span between opening and closing of a document.

4.2.8.4 CDocumentSession

The class `CDocumentSession` manages all open documents. When a document is opened, an instance of `CDocumentSession` is created that holds the information about the current session of the user and the opened document [Dubacher, 2003, page 67].

4.2.8.5 CChar

Instances of this class represent single characters of a text document [Dubacher, 2003, page 68]. With the properties `Before` and `After` the idea of a document as a 'ring of double linked fields' is realized, as each character object has a reference to the previous and the next character. As mentioned above, the character value is stored in Unicode as the property `CharacterValue`. The property `ElementBorder` indicates if the object is a start or end border of a zone.

The methods `InsertCChars` and `RemoveCChars` build the basic editing functionality and will be discussed in detail in section 4.3.1 on page 34 and in section 4.3.2 on page 35.

The methods `CUndo` and `CRedo` together with `InsertCChars` and `RemoveCChars` build the main part of the undo/redo framework.

4.2.8.6 CUndoHistory

The class `CUndoHistory` represents the history where all operations are logged for undo purposes.

4.2.8.7 CRedoHistory

The class `CRedoHistory` represents the history where all undo operations are logged for redo purposes.

4.3 TeNDaX Algorithms

In this section an overview of the two TeNDaX algorithms (insert and remove) that are most relevant for the undo/redo is given. The formal depiction follows in chapter 6. These algorithms provide the main functionalities for inserting and removing text. The discussion is mainly focused on the database side, i.e. not on the client side, as the undo/redo algorithms are exclusively realized on the database side and not in the TeNDaX multi-user editor.

4.3.1 Insert Algorithm

The TeNDaX insert algorithm inserts one or more characters into an existing document. A character is always inserted between two existing characters. When inserting one or more characters, the method `InsertCChars` of the class `CChar` is invoked on the database side. The method either returns the ID of the first inserted character if the insertion was successful or `NULL` if the insertion failed. The definition of the `InsertCChars` method is shown in figure 4.4.

```
InsertCChars = <characterId, nextCharacterId,
                previousCharacterId, where,
                characterValues, characterIds, attributes,
                document, session, userFk, isActive, flag,
                documentSession, undoCounter, lineageCharIds,
                copyPasteId, noUndo>
```

Figure 4.4: TeNDaX InsertCChars Method [Maendli, 2004]

The following list describes the set of parameters required to invoke the `InsertCChars` method.

- `characterId` holds the ID of the first character. If the `characterId` is `NULL`, the database will set the IDs (for synchronous database calls), otherwise the IDs of the characters are set by the client (for asynchronous database calls).
- `nextCharacterId` is the ID of the next character object
- `previousCharacterId` is the ID of the previous character object
- `where` indicates the approximate position of the inserted character(s). This is only used to increase the performance of the client updates.
- `characterValues` contains the characters to be inserted
- `characterIds` is a list that contains all IDs of the characters that have to be inserted
- `attributes` is used to pass metadata about the character

- `document` is an instance of `CFileNode`
- `session` is an instance of `CSession`
- `userFk` is the ID of the current user
- `isActive` is a status flag that indicates whether a character is active or not (used in relation to the remove algorithm discussed in section 4.3.2)
- `flag` indicates whether the insert action comes from an undo function or not
- `documentSession` is an instance of `CDocumentSession`
- `undoCounter` is used for undo purposes only, characters that are inserted by copy-paste get the same `undoCounter`
- `lineageCharIds` is a list of character IDs. It is used for the data lineage plug-in implemented in TeNDaX [Müller, 2004].
- `copyPasteId` indicates if the insertion comes from a copy-paste operation
- `noUndo` is used for undo operations to identify an import of a character

More about the insert algorithm and possible consistency conflicts due to the multi-user environment can be found in [Maendli, 2004].

4.3.2 Remove Algorithm

The TeNDaX remove algorithm removes one or more characters from a text document and returns a boolean value indicating whether the deletion has been successful or not.

To provide undo and redo functionality, a so-called *soft delete* is implemented (cf. section 4.2.3 on page 28). With the soft delete, the removed characters are not physically deleted from the database. Instead, a status flag is set to declare whether a character is active or inactive. This is necessary as the information about the character itself as well as meta information should still be available, i.e. stored in the database for possible undo, redo, or data lineage operations. When a character is deleted, the property `isActive` of the class `CChar` is set to '0'. The deleted characters are removed from the chain of double linked fields and the character previous to the deleted character(s) is concatenated with the next character not affected by the deletion.

The definition of the `RemoveCChars` method is shown in figure 4.5.

```
RemoveCChars= <characterIds, previousCharacterId, where,  
               userFk, document, session, flag,  
               documentSession, undoCounter>
```

Figure 4.5: TeNDaX RemoveCChars Method [Maendli, 2004]

The following list describes the set of parameters needed to invoke the **RemoveCChars** method.

- **characterIds** is a list that contains all IDs of the characters that have to be removed
- **previousCharacterId** is the character ID of the character previous to the character(s) that should be removed
- **where** indicates the approximate position of the removed character(s). This is only used to increase the performance of client updates.
- **userFk** is the ID of the current user
- **document** is an instance of **CFileNode**
- **session** is an instance of **CSession**
- **flag** indicates whether a remove operation is called from an undo or a client
- **documentSession** is an instance of **CDocumentSession**
- **undoCounter** is used for undo purposes only, characters that are inserted by copy-paste get the same **undoCounter**

More detailed information about the TeNDaX remove algorithm, and possible consistency conflicts due to the multi-user environment can be found in [Maendli, 2004].

4.4 Summary

The research project TeNDaX provides a totally new approach for storing documents within a database, which supports text editing on content and structure, all within a collaborative environment. This approach benefits from the security, concurrency control, integrity and consistency provided by DBMSs.

The TeNDaX system, a collaborative editor, offers the functionality of collaborative text editing within text documents stored in a special way in the database as described in [Hodel-Widmer and Dittrich, 2005].

Collaborative text editing includes operations for adding characters to a document and removing characters from a document and changing the document's structure by one or more users at the same time. In addition, the TeNDaX system supports sophisticated functionality in connection with collaborative text editing such as adding zones, e.g. layout, template, workflow, security or note zones, to a sequence of characters, full-text search, undo functionality, data lineage and visual text data mining.

In this chapter the basic concepts of the collaborative editor within the scope of the TeNDaX research project have been explained and the database based approach that was chosen has been described.

Chapter 5

Concurrency Control

The central problem in the area of collaborative editing is the consistency of data produced within a collaborative system. Whenever different people work simultaneously on the same document, conflicts can occur that affect the consistency. Therefore, it is essential to manage these conflicts in a way that the data is kept consistent. Different approaches exist that follow this intention.

It has to be distinguished if a system tries to avoid conflicts from the beginning through consistency control, resolution protocols and the like or if it just treats the conflicts as a naturally-arising side effect of the collaborative process. Then, a system has to contain a control structure that decides how to handle arising conflicts [Edwards, 1997].

In this chapter, it is presented, which conflicts can occur and how the conflict detection and resolution works in TeNDaX.

5.1 Collaborative Conflicts

If we talk about conflicts in a collaborative editor, we mean that an operation of a user tries to change the state of a document in a way that it either can not be applied to the document anymore or that it would leave a document state that is not desirable. That in turn means that the semantic of the document would be false afterwards.

5.1.1 Puzzles

Sequences of operations in a collaborative editor that lead to a conflict are denoted as puzzles. [Berlage, 1994], [Ressel and Gunzenhäuser, 1999], [Sun, 2000] and

[Jost, 2004] identified several undo puzzles that are described in the following subsections. For the given examples consider a document with the string 'Tendax' as its content.

5.1.1.1 Ambiguous Reference

If user 1 deletes 'd' and user 2 inserts a 'Y' between 'n' and 'a', the document state is 'TenYax'. Now suppose user 1 then wants to reinsert the deleted 'd' and therefore does a local undo it is not clear where to insert the 'd'. Two logical document states are possible: 'TendYax' or 'TenYdax'. Thus, this puzzle is called ambiguous reference because it is ambiguous where the 'd' should be reinserted (q.v. test case in section 5.4.4.1 on page 83).

5.1.1.2 Insert-insert-tie Dilemma

If user 1 deletes 'd' and user 2 wants to insert 'Y' before 'd' at the same time, the document state is 'TenYax'. If then user 1 wants to reinsert the 'd' and therefore does a local undo, then the document state should become 'TenYdax'. No other state would be correct but it might occur that the result of this puzzle is 'TendYax'.

5.1.1.3 Do-undo-pair Trap

If user 1 deletes 'n' and user 2 deletes 'd' the document state is 'Teax'. If then user 1 and user 2 want to reinsert the deleted characters and therefore do a local undo. The document state should become 'Tendax' again, what is the only correct state. Although, it could occur that the emerging state is 'Tednax' (q.v. test case in section 5.4.4.3 on page 85).

5.1.1.4 Overlapping Deletes Puzzle or Del-del-undo Puzzle

If user 1 and user 2 simultaneously delete 'd', the document state becomes 'Tenax'. If this is possible, there could occur problems, because to restore the original document state both users would have to do a local undo.

5.1.1.5 Insert-delete-undo Dilemma

This conflict occurs if a user does a local undo to undo his last insert operation but the character that should be deleted is already removed from the document by another user. The desired undo is not possible anymore and has to be skipped. Consider an empty document state. If user 1 inserts the string 'Tendax' and user

2 afterwards deletes the characters 'd' and 'a', the document state is 'Tenx'. Now user 1 does a local undo, which undoes the insert of character 'x'. A second undo of user 1 causes a conflict because it should undo the insertion of character 'a' but this character is already deleted by user 2 and has been removed from the document. The system should now detect that this operation is not possible anymore (q.v. test case in section 5.4.4.5 on page 88).

5.1.1.6 Copy-paste-insert Dilemma

If a block of connected characters is inserted into a document in one transaction by user 1 while user 2 works with this section, the copy-paste-insert dilemma can occur. If user 1 does a local undo, i.e. he deletes the previously inserted block of characters, only the changes of user 2 in this section should remain. This causes a non deterministic state, which in some circumstances has no semantic sense anymore. In such a case it may possibly make sense to stop the local undo and to call on the user to do the changes manually or to use the global undo (q.v. test case in section 5.4.4.6 on page 89).

5.1.1.7 Reference Character Not Existent Anymore

This is one of the most complex conflicts. It occurs if a user wants to undo his last delete operation of a character but the reference characters before and after it are already removed from the document. Consequently, it is not obvious where the character has to be inserted. Consider a document with the string 'Tendax'. If user 1 deletes the character 'd', the document state is 'Tenax'. If user 2 now deletes the characters 'na' and inserts in that position 'YYY', the document state is 'TeYYYx'. If user 1 then wants to undo his deletion of character 'd' by triggering a local undo, the reference characters 'na' are already deleted from the document and it is not obvious, where to reinsert 'd' (q.v. test case in section 5.4.4.7 on page 92).

5.2 Issues

This section identifies the problems of the former TeNDaX undo/redo solution, considering the presented puzzles, and briefly points out the improvements in the unDOit¹ algorithm.

¹The designation unDOit is chosen by the author.

5.2.1 Ambiguous Reference

The GDU algorithm is said to handle this conflict but tests have revealed that the behavior of the GDU algorithm in TeNDaX is in this case not at anytime deterministic. Sometimes the system is hanging and sometimes it skips the delete operation, undoes the next older operation if possible or informs the user of a not dissolvable conflict.

This puzzle is analyzed in detail and the solution is enhanced to an absolute deterministic behavior that covers all known special cases.

5.2.2 Insert-insert-tie Dilemma

In TeNDaX it is not possible to provoke this conflict because two operations can not be executed at the same time, therefore it is not possible for user 2 to insert 'Y' before 'd'. Thus, this dilemma does not exist in TeNDaX respectively causes the same conflict as the ambiguous reference.

5.2.3 Do-undo-pair Trap

The GDU algorithm uses only a very simple conflict resolution to resolve this puzzle. Moreover, there is a problem if at least one of the deleted characters is adjacent to the start or end character of a document. Then, the system hangs if a local undo is triggered.

This puzzle is analyzed in detail and the solution is enhanced to an absolute deterministic behavior that covers all known special cases.

5.2.4 Overlapping Deletes Puzzle or Del-del-undo Puzzle

In TeNDaX it is not possible to provoke this conflict because of the database based architecture.

5.2.5 Insert-delete-undo Dilemma

In such a case, the GDU algorithm selects the next older log entry of the user that triggers the local undo. In the example described in section 5.1.1.5 on page 39 this would be the insert operation of character 'd' of user 1, which also has been deleted by user 2. At the third attempt the GDU algorithm finds a log entry of user 1 that can be undone. The character 'n' is deleted and the new document state is 'Te'. In the UndoHistory the conflicting operations are treated as they would have never

occurred. The status variables are set to 0 (`IsActive = 0`).

This puzzle seems to be solved well but the conflict resolution that underlies, skips all operations, which try to manipulate characters that are edited by another user, what leads to numerous problems.

The solution is enhanced to a totally different conflict resolution that does not skip a character if it was just edited by another user. The new conflict resolution checks if a character is already deleted by another user. Therefore, undo operations that affect characters, which were edited by other users, are now generally possible.

5.2.6 Copy-paste-insert Dilemma

The GDU algorithm is capable to identify such a conflict and, if identified, undoes all transactions on the database side that were already executed. Furthermore, the user is informed of such a conflict. Thus, he can edit the appropriate section by himself. This puzzle is solved well.

Although, the solution is enhanced to a totally different conflict resolution that is configurable and allows the removal of non consecutive characters. Consequently, it is possible to undo an insertion of an area, in which characters have been inserted in the meantime.

5.2.7 Reference Character Not Existent Anymore

The GDU algorithm tries to solve this problem by using additional information of the history, in which for all operations the previous reference character and the subsequent reference character are stored. Usually the GDU algorithm inserts the character(s) at the leftmost possible position. Tests have admittedly revealed that the behavior of the GDU algorithm in TeNDaX is in this case not at anytime deterministic. Sometimes the desired result is achieved but sometimes errors are thrown or the application hangs.

The new framework is enhanced with a more powerful, stable and deterministic conflict resolution that is even configurable.

5.3 Conflict Resolution

It is not always obvious what the intention of a user is. Thus, the resolution of conflicts might not result in the state, the user desired. In general, the conflict resolution system should be capable to provide the user with the possibilities to decide what to do if a conflict occurred that possibly leads to an undesired document

state. The system should present a choice of operations that are possible to the given point of time including the possibility to just abort the operation that caused the undesired state.

The conflict resolution of the unDOit algorithm that is explained in this section in detail, detects occurring conflicts and categorizes the different circumstances. It then resolves the conflicts depending on configurable settings. Thus, the conflict resolution is flexible in reacting on the various conflicting situations although the user is not asked for a decision. Due to the fact that the unDOit algorithm categorizes the conflicts and provides the possibility to set different strategies to resolve conflicting operations, an extension that builds upon the existing solution could manage the user query while the existing algorithm handles the conflict resolution. Before the algorithm is presented in detail, we introduce the notion of first order and second order reference characters that is necessary to explain the insert conflict resolution.

5.3.1 Reference Characters

As seen in section 4.2.2 on page 26 in TeNDaX each character is stored in the database as an object with several properties. All characters in a document build a ring of double linked fields, i.e. each character holds a reference to the previous and to the next character in the document.

5.3.1.1 First Order Reference Characters

Consider a string of characters, e.g. 'Tendax' in a document. Due to the ring of double linked fields each character of this string holds a reference to the preceding and one to the succeeding character. Those references are stored in the properties **After** and **Before** of each character. **After** marks the reference to the preceding character since the character that holds the reference is situated after the reference character in the document. Vice versa, **Before** marks the reference to the succeeding character because the character that holds the reference is situated before the reference character in the document. From now on such reference characters of a character are called *First Order Reference Characters* and are abbreviated as *FORC*. Furthermore, we call the *First Order Reference Character* that is marked with the property **After**, the *First Order Reference Character on the left side* and abbreviate it with *FORC-L* while we call the one that is marked with the property **Before**, *First Order Reference Character on the right side* (*FORC-R*). In the string

'Tendax', the two FORCs of 'd' are: on the left side 'n' (**After**) and on the right side 'a' (**Before**). If a character is the first one in the document, its FORC-L is the beginning of the document (`CharacterValue = 99998`) and if it is the last character in the document, its FORC-R is the end of the document (`CharacterValue = 99999`). Likewise, the FORCs of a string shall be defined as follows: On the left side it is the FORC-L of the first character in the string while it is on the right side the FORC-R of the last character in the string.

5.3.1.2 Second Order Reference Characters

Consider the string of characters in a document as mentioned above. Each character in this string has two FORCs, which both have itself two FORCs. In a document with the active (`IsActive = 1`) characters 'Tendax', 'n' is the FORC-L of 'd' and 'n' itself has the two FORCs 'e' (FORC-L) and 'd' (FORC-R). That means 'd' has 'n' as FORC-L while 'n' has 'd' as FORC-R, what ensures that the list of double linked fields is valid in between those two characters.

As we will see in detail later on in the conflict resolution of undo/redo, it is possible that two characters do not have each other as FORCs. Thus, the notion *Second Order Reference Character* shall be introduced in this section. From now on we call the FORC of a FORC of a character that points in the opposite direction the *Second Order Reference Character* of a character and abbreviated it as *SORC*. Consider the example mentioned above. 'n' is the FORC-L of 'd'. The FORC-R of 'n' (opposite direction) is 'd' what makes 'd' to its 'SORC-L'.

5.3.1.3 unDOit Reference Character Model

The unDOit reference character model is based on the notions of *FORC* and *SORC* and is used in the conflict resolution of the unDOit algorithm.

In figure 5.1 the model is illustrated. At the top of the figure the characters i, n and s symbolize a character string 'ins' with its first character 'i' and its last character 's' that are represented in the implementation with the variables `Fid` (**F**irst **i**d) respectively `Lid` (**L**ast **i**d). On the left side of the figure the character 'B' stands for the FORC-L of the string 'ins' and is represented in the implementation with the variable `tmpAfter` while on the right side of the figure the character 'E' stands for the FORC-R of the string 'ins' that is represented with the variable `confBefore` in the implementation. Situated on the bottom of the figure are the two SORCs of the string 'ins', 'X' as SORC-L and 'Y' as SORC-R, represented in the implementation

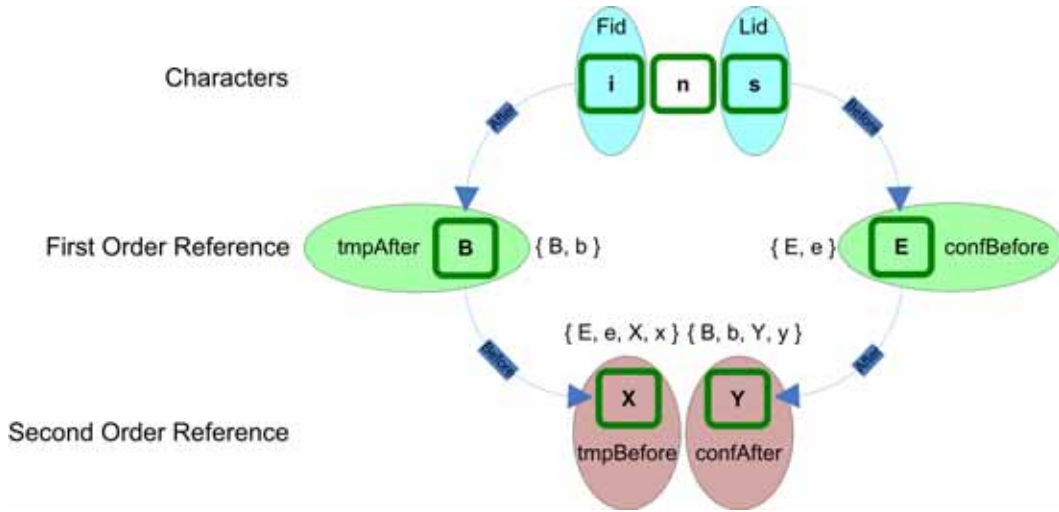


Figure 5.1: First Order and Second Order Reference Characters (own contribution)

as **tmpBefore** and **confAfter** respectively. The arrows that connect the components in the model symbolize the attributes **After/Before** of a character.

To include the status (**IsActive** = 1 or **IsActive** = 0) of a character in the model, the content of the curly brackets in figure 5.2 shall be explained.

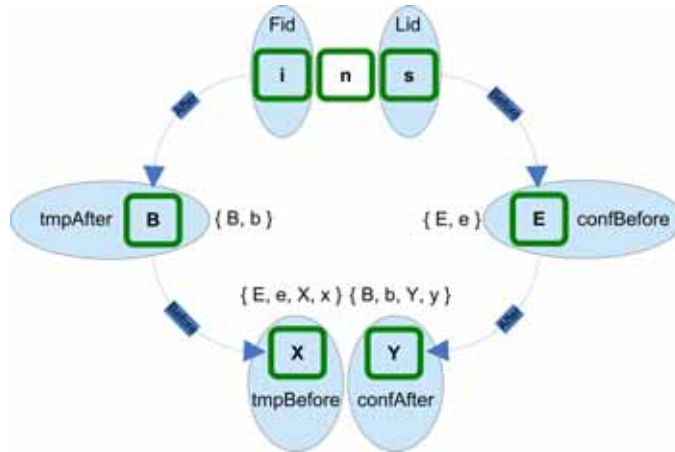


Figure 5.2: unDOit Reference Character Model (own contribution)

A capital letter marks an active character while a lower case letter marks an inactive character. The different letters (E/e, B/b, X/x, Y/y) characterize the type of a character reference. The FORC-L of a string/character is symbolized with a 'B' (**B**egin) for active respectively with a 'b' for not active. The FORC-R of a string/character is symbolized with an 'E' (**E**nd) for active respectively with an 'e' for not active.

The SORC-L of a string/character is symbolized with an 'E' (active) or an 'e' (not active) if it is the same character as the FORC-R and if it is another character it is symbolized with a 'X' (active) or a 'x' (not active). The SORC-R of a string/character is symbolized with a 'B' (active) or a 'b' (not active) if it is the same character as the FORC-L and if it is another character it is symbolized with a 'Y' (active) or a 'y' (not active). Generally, the elements in the curly brackets enumerate the possibilities for the reference characters.

5.3.2 Overview

In this subsection, a simplified overview of the complete undo/redo algorithm is given in order to emphasize how the conflict resolution is embedded in the TeNDaX system. The detailed discussion of all the possible cases follows in the particular subsections 5.3.3 (page 49 et seqq.) and 5.3.4 (page 63 et seqq.).

As it is depicted in figure 5.3, the beginning of an undo/redo action is located at the client, if a user pushes one of the four undo/redo buttons (global undo, local undo, local redo, global redo). In few words, the command invokes through the database interface the appropriate method in the database. Depending on the according button that was pressed, either the global or local undo algorithm in `CChar.CUndo` or the global or local redo algorithm in `CChar.CRedo` is triggered. The first and comparatively smaller part of the conflict resolution takes place in these class methods.

A local undo/redo searches in the undo/redo history for the last entry of the user that has triggered the undo/redo action but only the last entry that makes sense to undo/redo is chosen. In contrast to the local algorithm, the global undo/redo just searches in the undo/redo history for the last entry of any user and does not have to prove that it is reasonable² to undo/redo that operation.

From the methods `CChar.CUndo`/`CChar.CRedo` the appropriate method `CChar.InsertCChars` or `CChar.RemoveCChars` is invoked, depending on what kind of operation has to be undone/redone. In these class methods the second and more complex part of the conflict resolution is implemented.

If a removal has to be performed, the algorithm distinguishes if the removal of non consecutive characters is allowed and removes the appropriate characters if possible. The detailed discussion follows in subsection 5.3.4 on page 63 and for the formal

²A reasonable history entry is one that makes sense to undo/redo. E.g., it is not reasonable to delete a character that has already been removed.

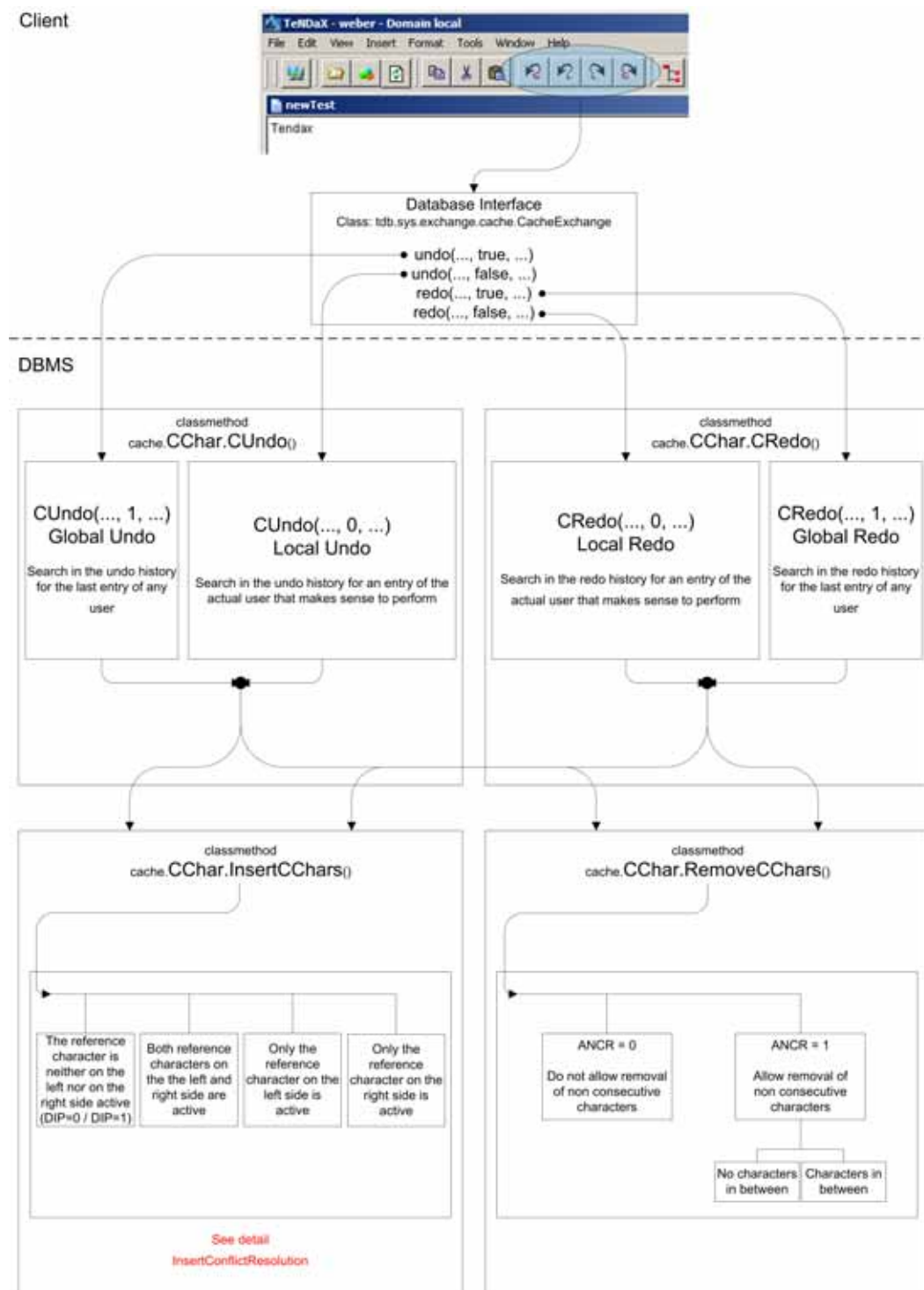


Figure 5.3: Simplified Overview (own contribution)

depiction of the remove algorithm invoked from an undo/redo it is referred to section 6.7 on page 145.

If an insertion has to be performed, the algorithm distinguishes much more cases as for a removal. The algorithm determines, based on the unDOit reference character model, where the characters shall be inserted and inserts the appropriate characters if possible. The algorithm distinguishes in a first step, which FORCs are active and can be used for the conflict resolution. The following listing shows the four possibilities.

- Neither FORC-L nor FORC-R is active
- Both FORC-L and FORC-R are active
- Only FORC-L is active
- Only FORC-R is active

For the formal depiction of the insert algorithm invoked from an undo/redo it is referred to section 6.6 on page 128.

As mentioned above, the conflict resolution for an insertion leads to a more complex case distinction. Thus, a more detailed overview is given in the subsection 5.3.3 (page 49 et seqq.).

5.3.2.1 Configuration Strategies

The unDOit algorithm does not only have one strategy to handle occurring conflicts. It provides a configurable conflict resolution that can be adjusted if needed. There are three strategies that influence the setting of the algorithm. Namely, the *Default Insert Policy* (DIP), the *Allow Non Consecutive Removal* (ANCR) and the *Empty RedoHistory After Edit* (ERAE) that are explained in the following paragraphs. Each strategy is binary and can be switched by setting it to either 0 or 1 in the implementation.

Default Insert Policy (DIP) The DIP is always applied if it is not obviously decidable if a character shall be inserted either on the left or on the right side. E.g., if a string should be inserted but no FORC of it is active, the algorithm has no indication where the next active character is located in the document and therefore has to search for it. In that case, the DIP is used to decide if the algorithm should search left of the FORC-L or if it should search right of the FORC-R until an active character is found. In short, the DIP defines if a character is inserted on the left

side or on the right side if a decision is needed (DIP=0: search left, DIP=1: search right).

Allow Non Consecutive Removal (ANCR) The ANCR controls if a string of characters can be removed even if its characters are not consecutive in the document. This situation can occur if one user inserts a string of characters, another user edits this string by inserting characters in between and the first user then triggers an undo, which tries to remove the inserted characters. The remaining characters of the second user might have no sense without the removed ones. Thus, the possibility exists to restrict the algorithm (ANCR=0: do not allow removal of non consecutive characters, ANCR=1: allow removal of non consecutive characters).

Empty RedoHistory After Edit (ERAE) After each insertion/removal, all entries in the RedoHistory of the appropriate document are deleted if ERAE=1. Thus, if so, a redo action is only possible if the former action was neither an insertion nor a removal not triggered by an undo/redo (ERAE=0: do not empty the RedoHistory after an insertion/removal, ERAE=1: empty the RedoHistory after an insertion/removal).

5.3.3 Insert Conflict Resolution

The conflict resolution for an insertion triggered by an undo/redo action depends on the availability of active reference characters that surround the insertion string. If an insertion has to be performed, the algorithm determines, based on the unDOit reference character model, where the characters shall be inserted and inserts the appropriate characters if possible. The algorithm distinguishes in a first step, which FORCs are active and can be used for the conflict resolution.

Four main categories can be distinguished:

- 0-0 The reference character is neither on the left nor on the right side active
(FORC-L=0 AND FORC-R=0)
- 1-1 Both reference characters on the left and on the right side are active
(FORC-L=1 AND FORC-R=1)
- 1-0 Only the reference character on the left side is active
(FORC-L=1 AND FORC-R=0)
- 0-1 Only the reference character on the right side is active
(FORC-L=0 AND FORC-R=1)

From this main distinction follow twenty seven special cases that are possible, based on the ring of double linked fields. Figure 5.4 provides an overview. In the following, it is described, which cases are theoretically possible, how these cases are characterized and how they are categorized. In section 5.4 on pages 65 et seqq., the according test cases are presented. All the cases can be categorized in nine distinct categories that represent the type of an insertion, i.e. where the algorithm determines the insertion of the character string. To explain the different cases, not only the notion of the unDOit reference character model of section 5.3.1.3 is used, but also the attribute **CharCounter** and the OID of a character. We use the abbreviations *CC* for the attribute **CharCounter** and *id* for the OID. For example, consider the active SORC-L that is not equal to the FORC-R and therefore is symbolized with an 'X'. The attribute **CharCounter** of X is indicated by X_{CC} while the OID is indicated by X_{id} . The similar applies for the first character of the insertion string, the **Fid** (i_{CC}/i_{id}) and the active SORC-R that is not equal to the FORC-L (Y_{CC}/Y_{id}). The notions β and ε for the beginning respectively the end of the document are already introduced in section 4.2.5 on page 28.

5.3.3.1 Insert Categories

As mentioned above, all the possible cases that can be distinguished end up in a decision where to insert the character string relative to the reference characters of the unDOit reference character model. Only nine categories are imaginable but three of them are theoretically not possible. The following listing shows all imaginable categories:

- Category one (CAT 1):
Insertion between B/b (**tmpAfter**) and E/e (**confBefore**)
- Category two (CAT 2):
Insertion between B (**tmpAfter**) and X (**tmpBefore**)
- Category three (CAT 3):
Insertion between Y (**confAfter**) and E (**confBefore**)
- Category four (CAT 4):
Search left for a reference character
- Category five (CAT 5):
Search right for a reference character

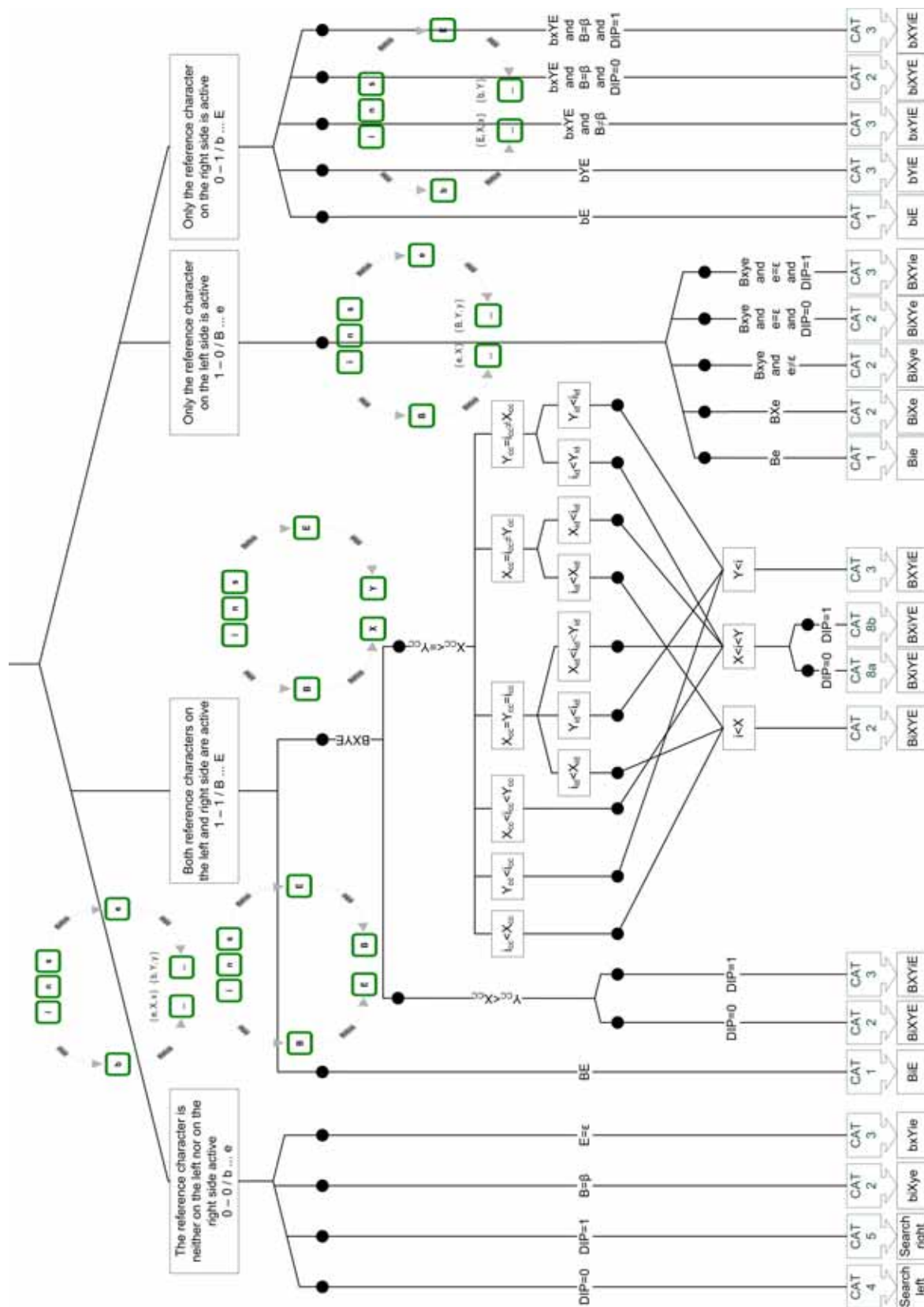


Figure 5.4: Simplified Overview - Insert Conflict Resolution (own contribution)

- Category six (CAT 6):
Insertion between B (`tmpAfter`) and Y (`confAfter`), not possible
- Category seven (CAT 7):
Insertion between X (`tmpBefore`) and E (`confBefore`), not possible
- Category eight (CAT 8):
Insertion between X (`tmpBefore`) and Y (`confAfter`)
- Category nine (CAT 9):
Insertion between Y (`confAfter`) and X (`tmpBefore`), not possible

5.3.3.2 FORC-L=0 and FORC-R=0

This is the only case where the algorithm has no indication where to insert the characters since all reference characters have been removed and neither the reference character on the left nor on the right side is active (cf. figure 5.5). In this case,

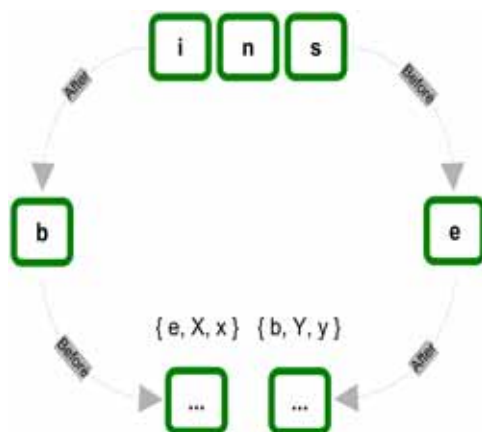


Figure 5.5: Reference Character Model for FORC-L=0 AND FORC-R=0 (own contribution)

there exist the two possibilities to search for an active character: on the left side of the FORC-L or on the right side of the FORC-R. The DIP controls, on which side the algorithm searches. If DIP=0 the algorithm runs into category four (CAT4) and begins the search with the character that is situated before the FORC-L in the document and continues until an active character is found or the beginning of the document is reached. Once the left reference character is determined, the algorithm gets the FORC-R of it and inserts the string between those two characters (cf. test case 0-0 A on page 70). If DIP=1 the algorithm runs into category five (CAT5) and

begins the search with the character that is situated after the FORC-R in the document and continues until an active character is found or the end of the document is reached. As soon as the right reference character is determined, the algorithm gets the FORC-L of it and inserts the string between those two characters (cf. test case 0-0 B on page 71).

It is possible that one or both of the FORCs are not just inactive characters that were inserted from a user, but special characters that mark the beginning or the end of the document. If FORC-L is the beginning and FORC-R the end of the document, the character string that shall be reinserted was removed as the only content in the document, thus it makes sense to insert the characters depending on the DIP as described above. If only one of the FORCs is such a special character it might be reasonable to insert the string right next to it. Therefore, such a case is treated as if one character would be active. If FORC-L is the beginning of the document ($b=\beta$) the characters are inserted between b and X (CAT2, cf. test case 0-0 C on page 71) and if FORC-R is the end of the document ($e=\epsilon$) the characters are inserted between Y and e (CAT3, cf. test case 0-0 D on page 71).

5.3.3.3 FORC-L=1 and FORC-R=1

The situation where both reference characters on the left and on the right side are active, is the most likely and most complex main category since there are many cases that can be distinguished.

The simplest case is the one where all reference characters correspond to each other (cf. figure 5.6), when the FORC-L and the FORC-R are adjacent, i.e. the characters are exactly reinserted where they were at the time of the removal (cf. test case 1-1 A on page 72).

$$\begin{aligned} FORC - L (\text{tmpAfter}) &= SORC - R (\text{confAfter}) \\ \text{AND} \\ FORC - R (\text{confBefore}) &= SORC - L (\text{tmpBefore}) \end{aligned}$$

It causes the algorithm to run into category 1 (CAT1):

$$\text{Insertion between } B (\text{tmpAfter}) \text{ and } E (\text{confBefore})$$

The model would allow another configuration where only one FORC and SORC are equal (cf. figures 5.7 and 5.8) but such a combination is contradictory to the ring of double linked fields, because only one FORC is linked to the other FORC what would result in a not valid ring of characters. This combination would cause an insertion between $B (\text{tmpAfter})$ and $Y (\text{confAfter})$ respectively an insertion

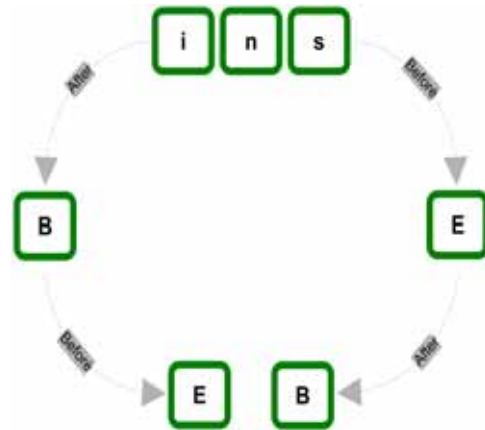


Figure 5.6: Reference Character Model for FORC-L=1 AND FORC-R=1 simple (own contribution)

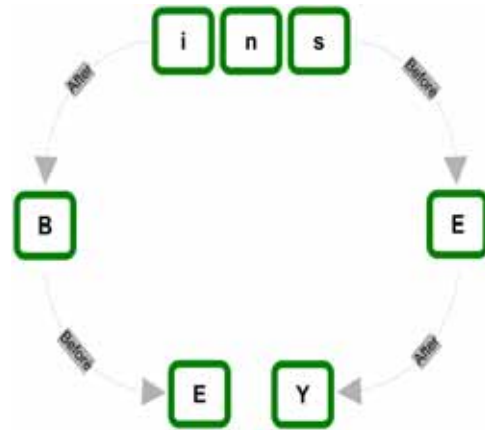


Figure 5.7: Reference Character Model for FORC-L=1 AND FORC-R=1 not possible B_Y_E (own contribution)

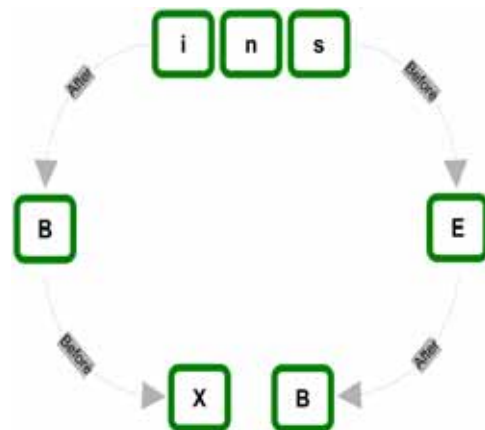


Figure 5.8: Reference Character Model for FORC-L=1 AND FORC-R=1 not possible B_X_E (own contribution)

between X (*tmpBefore*) and E (*confBefore*) what must not happen. Thus, those two categories six and seven (CAT6 and CAT7) are not possible in a valid ring of double linked fields and does not appear in the overview (cf. figure 5.4).

The remaining cases of this main category (FORC-L=1 AND FORC-R=1) emerge from the same combination in the reference character model where no reference character corresponds to another (cf. figure 5.9) when the FORC-L and the FORC-R are separated by at least one character:

$$\begin{aligned}
 &FORC - L \text{ (tmpAfter)} \neq SORC - R \text{ (confAfter)} \\
 &\text{AND} \\
 &FORC - R \text{ (confBefore)} \neq SORC - L \text{ (tmpBefore)}
 \end{aligned}$$

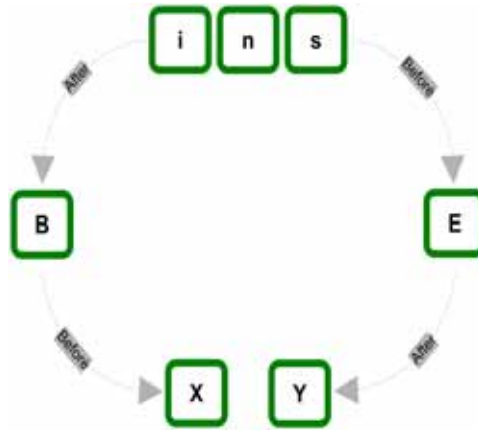


Figure 5.9: Reference Character Model for FORC-L=1 AND FORC-R=1 complex (own contribution)

If this combination occurs, what is quite probable if two or more users edit the same sentence or word, both reference characters FORC-L and FORC-R are available, i.e. the algorithm has the exact information where the characters were situated before but the problem is that there are characters in between (at least one) and the algorithm has to decide where the insertion string has to be reinserted. The sequence of reference characters in the document is B X Y E, where X and Y could be the same character. Thus, there are only three possibilities for an insertion:

- CAT2: Insertion between B (*tmpAfter*) and X (*tmpBefore*)
- CAT3: Insertion between Y (*confAfter*) and E (*confBefore*)
- CAT8: Insertion between X (*tmpBefore*) and Y (*confAfter*)

Category eight has to be divided into two different cases because the SORC-L (X) and SORC-R (Y) probably are not adjacent, assuming that they are not the same character. Thus, a string can not just be inserted exactly between X and Y in order to maintain a valid ring of double linked fields. It has to be decided if the string is inserted directly right of SORC-L (X) or directly left of SORC-R (Y). Then, the string is inserted between X and its FORC-R or between Y and its FORC-L.

- CAT8a: Insertion between X (`tmpBefore`) and FORC-R of X
- CAT8b: Insertion between FORC-L of Y and Y (`confAfter`)

The case where X and Y are adjacent results in the same sequence, regardless of which decision was made.

Generally, it is not at all explicit where the string has to be inserted because this is a question of semantic coherence in the document. The only meta data that can be used to get more information is the attribute `CharCounter` and the OID of each character. With this information it can be determined, which characters were inserted before the others but the chronological order does not guarantee a correct semantic: if many characters, words or sentences are belatedly inserted in a document, the chronological order of the insertions does not represent the sequence of characters in the document. Thus, it is difficult to estimate where the best insertion position would be.

The unDOit algorithm tries to resolve this combination by analyzing the `Fid` (i), the SORC-L (X) and the SORC-R (Y) to perform a reinsertion that maintains the chronological order, if possible. With this strategy, the characters are reinserted in the most logic way if they were inserted as common from left to right. If the SORC-L and SORC-R are adjacent the algorithm will determine the correct position. If there are characters in between the SORCs the algorithm might not be able to determine the correct position, because it only analyzes the FORCs and SORCs. A possible enhancement would be to check also the characters in between what would determine the correct position but would take a long time. All these circumstance make it extremely hard to estimate the best insert position, thus it is also possible to inactivate this special conflict resolution. Then, the algorithm inserts the character string according to the DIP on the left side, between B and X (DIP=0) or on the right side, between Y and E (DIP=1).

To determine the chronological order, the attribute `CharCounter` and in some cases even the OIDs of the characters are analyzed. First, the `CharCounter` is used to determine if a character was inserted before another, since this attribute definitely

represents the chronology of insertions unless the characters were inserted by a copy-paste operation. If so, the characters have the same **CharCounter** and the OID is taken to compare, which character was inserted first. The OID can only be taken for characters that have the same **CharCounter** because otherwise it is not guaranteed that the OID of an earlier inserted character is lower than the OID of a character that was inserted afterwards.

The reference characters in a document are located in the order 'BX ... YE'. It is not mandatory but possible that the SORC-L (X) and the SORC-R (Y) are adjacent or even the same character. All or some reference characters can have the same **CharCounter** (copy-paste) but all reference characters have a different OID, if they are not the same character. Although many cases can occur, there are just four resulting insertion categories, namely category two (CAT2), three (CAT3), eight a (CAT8a) and eight b (CAT8b), as mentioned above.

To decide where it makes sense to reinsert a character string, it is analyzed if the first character of the string (**Fid**) was inserted after the SORC-L ($i > X$), if the **Fid** was inserted after the SORC-R ($i > Y$) and which SORC was inserted first ($X < Y$ or $Y < X$). It also might be possible that the SORC-L is equal to the SORC-R ($X = Y$). Table 5.1 shows the possible situations, where 1 symbolizes true and 0 false.

Cases 4, 6, 8 and 9 are impossible because of the contradictory variable codomains. Case 2, 5 and 11 are questionable because the chronological order of X and Y is not equal to the sequence of X and Y in the document, thus it makes no sense to insert the string with a strategy that tries to keep the chronological order. Cases 1, 3, 7, 10 and 12 make sense to resolve with this strategy even if it can not be guaranteed that the semantic coherence in the document can be maintained.

Cases 1 and 3 result in the same insertion between B and X, therefore the distinction between ($X < Y$) and ($X = Y$) is in this case unnecessary, as well for the cases 10 and 12, which result in the same insertion between Y and E. Thus, there result these six cases, which are presented in table 5.2.

First, the **CharCounter**'s of SORC-L (X) and SORC-R (Y) are compared. If Y_{CC} is lower than X_{CC} the chronological order of the characters is not equal to the sequence in the document and the characters are inserted between B and X (cf. test case 1-1 B on page 72) or Y and E (cf. test case 1-1 C on page 72) depending on the DIP. If Y_{CC} is not lower than X_{CC} it is clear that X and Y are in a chronological order. If so, the **CharCounters** of **Fid** (i), SORC-L (X) and SORC-R (Y) are compared. If all **CharCounters** are different, the chronological order is obvious but

<i>Case</i>	$i > X$	$i > Y$	$X < Y$	$Y < X$	$X = Y$	<i>Situation</i>	<i>Insertion</i>
01	0	0	1	0	0	$i < X < Y$	BiX ... YE
02	0	0	0	1	0	$i < Y < X$	(BiX ... YE)
03	0	0	0	0	1	$i < X = Y$	BiXYE
04	0	1	1	0	0	<i>not possible</i>	-
05	0	1	0	1	0	$Y < i < X$	(BXi ... iYE)
06	0	1	0	0	1	<i>not possible</i>	-
07	1	0	1	0	0	$X < i < Y$	BXi ... iYE
08	1	0	0	1	0	<i>not possible</i>	-
09	1	0	0	0	1	<i>not possible</i>	-
10	1	1	1	0	0	$X < Y < i$	BX ... YiE
11	1	1	0	1	0	$Y < X < i$	(BX ... YiE)
12	1	1	0	0	1	$X = Y < i$	BXYiE

Table 5.1: Insert Conflict Resolution Case Overview (own contribution)

<i>Case</i>	<i>Situation</i>	<i>Insertion</i>	<i>Category</i>
01	$(X < Y) \ \&\& \ (i < X)$	BiX...YE	CAT2
02	$(X < Y) \ \&\& \ (X < i < Y) \ \&\& \ (DIP = 0)$	BXi...YE	CAT8a
03	$(X < Y) \ \&\& \ (X < i < Y) \ \&\& \ (DIP = 1)$	BX...iYE	CAT8b
04	$(X < Y) \ \&\& \ (Y < i)$	BX...YiE	CAT3
05	$(X > Y) \ \&\& \ (DIP = 0)$	BiX...YE	CAT2
06	$(X > Y) \ \&\& \ (DIP = 1)$	BX...YiE	CAT3

Table 5.2: Insert Conflict Resolution Possible Cases (own contribution)

otherwise also the OIDs of **Fid**, X and Y have to be compared as well to ensure the correct determination of the chronological order. For a detailed description of the statements that are checked, we refer to figure 5.4 on page 51, which shows the appropriate decision tree. Once the order is determined, the corresponding categories are set. If $i < X$ the insertion takes place between B and X (CAT2), if $X < i < Y$ the characters are inserted depending on the DIP between X and **FORC-R** of X (DIP=0, CAT8a) or between **FORC-L** of Y and Y (DIP=1, CAT8b) and if $Y < i$ the string is inserted between Y and E (CAT3). In the following listing a short overview is presented, which shows the different combinations that occur when the **CharCounter** of **SORC-R** is not lower than the one of **SORC-L** ($X_{CC} \leq$

Y_{CC}):

- $i_{CC} < X_{CC}$ (cf. test case 1-1 D on page 73)
- $Y_{CC} < i_{CC}$ (cf. test case 1-1 E on page 73)
- $X_{CC} < i_{CC} < Y_{CC}$ (cf. test case 1-1 F on page 73)
- $X_{CC} = Y_{CC} = i_{CC}$ AND $i_{id} < X_{id}$ (cf. test case 1-1 G on page 74)
- $X_{CC} = Y_{CC} = i_{CC}$ AND $Y_{id} < i_{id}$ (cf. test case 1-1 H on page 74)
- $X_{CC} = Y_{CC} = i_{CC}$ AND $X_{id} < i_{id} < Y_{id}$
(cf. test cases 1-1 I on page 75)
- $X_{CC} = i_{CC} \neq Y_{CC}$ AND $i_{id} < X_{id}$ (cf. test case 1-1 J on page 75)
- $X_{CC} = i_{CC} \neq Y_{CC}$ AND $X_{id} < i_{id}$ (cf. test cases 1-1 K on page 76)
- $Y_{CC} = i_{CC} \neq X_{CC}$ AND $i_{id} < Y_{id}$ (cf. test cases 1-1 L on page 76)
- $Y_{CC} = i_{CC} \neq X_{CC}$ AND $Y_{id} < i_{id}$ (cf. test case 1-1 M on page 77)

5.3.3.4 FORC-L=1 and FORC-R=0

In this situation, where only the reference character on the left side is active (cf. figure 5.10), it seems to be the most logic to reinsert the character string between the active FORC and the appropriate SORC but it might also be possible that the inactive reference character is the end of the document.

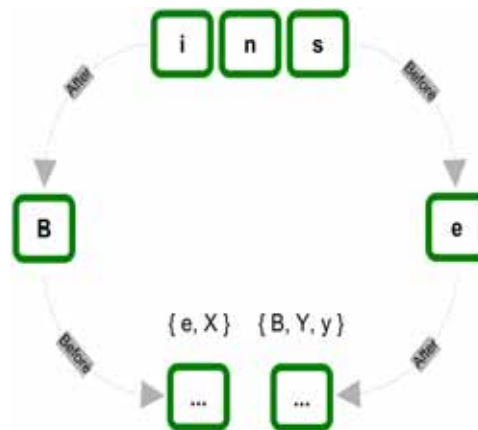


Figure 5.10: Reference Character Model for FORC-L=1 AND FORC-R=0 (own contribution)

Thus, three combinations in the unDOit reference character model are possible that accord to a valid ring of double linked fields.

Be The simplest case is the one where all reference characters correspond to each other what means in this special case that the characters were removed as trailing ones in a document and FORC-R is the end of the document (cf. test case 1-0 A on page 78).

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &= \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &= \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

It causes the algorithm to run into category 1 (CAT1):

$$\text{Insertion between B}(\text{tmpAfter}) \text{ and e}(\text{confBefore})$$

Bye If only FORC-R and SORC-L are equal, the ring of double linked fields would not be valid and therefore this situation is not possible. FORC-L (B) is active and has a not active character as its FORC-R (e) in the document. Thus, this character (e) must be the end of the document but has as its FORC-L (Y) another character as reference than B, what would leave a not valid ring of double linked fields.

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &\neq \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &= \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

BXe If only FORC-L and SORC-R are equal, a common situation is represented where FORC-R is an already deleted reference character (cf. test case 1-0 B on page 78).

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &= \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &\neq \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

It causes the algorithm to run into category 2 (CAT2):

$$\text{Insertion between B}(\text{tmpAfter}) \text{ and X}(\text{tmpBefore})$$

BXye If none of the reference characters are equal,

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &\neq \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &\neq \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

it can be distinguished between a common situation where FORC-R is an already deleted reference character and a situation where the FORC-R is the end of the document.

In the common situation only the FORC-L can be used, because FORC-R is an inactive character. Thus, the algorithm runs into category 2 (CAT2, cf. test case 1-0 C on page 78) and inserts the string between B (**tmpAfter**) and X (**tmpBefore**). If FORC-R is the end of the document ($e=\varepsilon$) the situation is similar to where both reference characters are active but separated by other characters. Thus, it depends on the DIP where the characters are reinserted. Category 2 (CAT2, cf. test case 1-0 D on page 79) is set, if $DIP=0$ (insert left) and the string is reinserted between B (**tmpAfter**) and X (**tmpBefore**). If $DIP=1$ (insert right), category 3 (CAT3, cf. test case 1-0 E on page 79) is set and the characters are reinserted between Y (**confAfter**) and E (**confBefore**).

5.3.3.5 FORC-L=0 and FORC-R=1

The situation where only FORC-R is active is comparable with the situation where only the FORC-L is active (cf. figure 5.11). It seems to be the most logic to reinsert the character string between the active FORC and the appropriate SORC but it might also be possible that the inactive reference character is the beginning of the document.

Thus, three combinations in the unDOit reference character model are possible that accord to a valid ring of double linked fields.

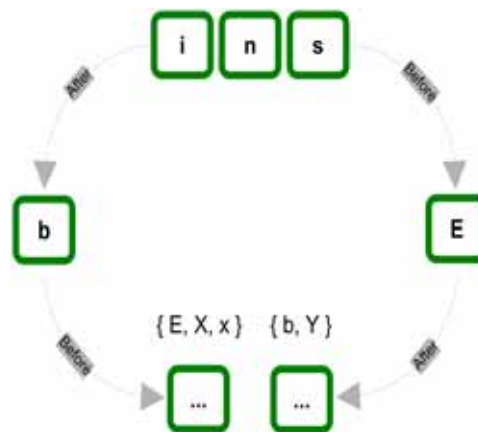


Figure 5.11: Reference Character Model for FORC-L=0 AND FORC-R=1 (own contribution)

bE The simplest case is the one where all reference characters correspond to each other, i.e. the characters were removed as leading ones in a document and FORC-L is the beginning of the document (cf. test case 0-1 A on page 79).

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &= \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &= \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

It causes the algorithm to run into category 1 (CAT1):

Insertion between b (tmpAfter) and E (confBefore)

bYE If only FORC-R and SORC-L are equal, a common situation is represented where FORC-L is an already deleted reference character (cf. test case 0-1 B on page 80).

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &\neq \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &= \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

It causes the algorithm to run into category 3 (CAT3):

Insertion between Y (confAfter) and E (confBefore)

bxE If only FORC-L and SORC-R are equal, the ring of double linked fields would not be valid and therefore this situation is not possible. FORC-R (E) is active and has a not active character as its FORC-L (b). Therefore, this character (b) must be the beginning of the document but has as its FORC-R (x) another character as reference than E, what would leave a not valid ring of double linked fields.

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &= \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &\neq \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

bxYE If none of the reference characters correspond, it can be distinguished between a common situation where FORC-L is an already deleted reference character and a situation where the FORC-L is the beginning of the document.

$$\begin{aligned} \text{FORC} - L(\text{tmpAfter}) &\neq \text{SORC} - R(\text{confAfter}) \\ \text{AND} \\ \text{FORC} - R(\text{confBefore}) &\neq \text{SORC} - L(\text{tmpBefore}) \end{aligned}$$

In the common situation only the FORC-R can be used, because FORC-L is an inactive character. Thus, the algorithm runs into category 3 (CAT3, cf. test case 0-1 C on page 80) and inserts the string between Y (`confAfter`) and E (`confBefore`). If FORC-L is the beginning of the document ($b=\beta$) the situation is similar to where both reference characters are active but separated by other characters. Consequently, it depends on the DIP where the characters are reinserted. Category 2 (CAT2, cf. test case 0-1 D on page 80) is set, if DIP=0 (insert left) and the string is reinserted between B (`tmpAfter`) and X (`tmpBefore`). If DIP=1 (insert right), category 3 (CAT3, cf. test case 0-1 E on page 80) is set and the characters are reinserted between Y (`confAfter`) and E (`confBefore`).

5.3.3.6 Client Update

Finally, for any insertion from an undo/redo that contains not consecutive characters, the clients are informed of an insertion not by sending the complete string but by sending each single character.

5.3.4 Remove Conflict Resolution

The conflict resolution for a removal triggered by an undo/redo action is completely different to the one of an insertion. It does not depend on the availability of active reference characters and therefore does not cause so many special cases. Unlike to the insertion conflict resolution, the remove conflict resolution begins already when the operation that has to be undone/redone is chosen. For a local undo/redo only a reasonable operation is chosen to undo/redo, i.e. an operation that would try to remove a character, which has already been removed, will be skipped.

The redoing of a removal and the undoing of an insertion provoke such a verification of a reasonable operation. The character string that shall be removed is checked if it contains characters that are already deleted in the document (`IsActive=0`). If the algorithm finds at least one such character, it skips this operation and searches in the history for another operation. This is repeated until an operation is found that removes only active characters or that is an insert operation. Once the reasonable operation is found, the removal of the character string can be performed.

For the removal of characters, the unDOit algorithm provides two different strategies (cf. ANCR in subsection 5.3.2.1 on page 48). In addition to a restricted algorithm (ANCR=0) it exists an enhanced algorithm that allows the removal of characters, which are not consecutive in the document. Therefore, we divide this subsection

into two parts that describe the restricted algorithm (subsection 5.3.4.1), which does not allow the removal of non consecutive characters and the enhanced algorithm (subsection 5.3.4.2), which allows the removal of non consecutive characters.

5.3.4.1 Non Consecutive Removal Not Allowed (ANCR=0)

If the removal of non consecutive characters is not allowed, the algorithm tries to remove the characters one by one, setting their property `IsActive` to 0 ('Soft Delete') while checking if each character is situated in the document after the previously deleted character. If the algorithm reaches a character that is not consecutive with the previously deleted one it aborts the undo/redo operation.

5.3.4.2 Non Consecutive Removal Allowed (ANCR=1)

If the removal of non consecutive characters is allowed, the conflict resolution is significantly more complex than with ANCR=0. At first, the first character that shall be removed has to be consecutive with the preceding one.

To explain how the algorithm works, consider a document where a user has entered the text 'Tendax', then he has additionally inserted the letters 'ABCD' with a copy-paste operation and another user has belatedly inserted '12' in a way that a document state 'TenAB12CDdax' results. Now, if the first user triggers a local undo the insertion of the string 'ABCD' shall be undone but the characters of this string are not anymore consecutive in the document since they are separated by '12'.

Figure 5.12 illustrates the following procedure: The algorithm stores the characters that are situated before (`areaAfter`) and after (`areaBefore`) the string and then traverses it character by character, checking if the actual character is consecutive with the next one to be removed. Each character that is consecutive with the next

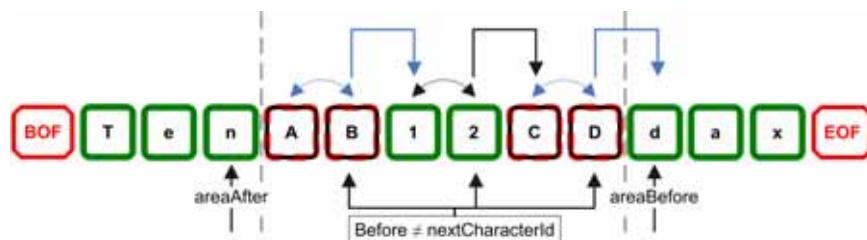


Figure 5.12: ANCR=1 (own contribution)

one is removed by setting its property `IsActive` to 0 ('Soft Delete'). If a character is reached that is not consecutive with the next one (`before≠nextCharacterId`) the

character that is not consecutive and all of its succeeding characters until the next character of the removal string are stored in a list that contains all of the characters, which are situated in between the string. After this procedure, all characters of the string are removed from the document (`IsActive=0`) and all characters that were situated in between are stored in a special list. If there were characters in between, each character of the list is linked in a way that the ring of double linked fields is valid again and the removed characters are linked to each other as well.

5.3.4.3 Client Update

If characters were in between the removal string, the clients are informed of a combined operation. First, the removal of all characters (removal string and characters in between) and subsequently the insertion of the characters in between.

5.4 Test Cases

In this section several test cases are presented that demonstrate the functionality of the implemented `unDOit` algorithm in `TeNDaX`. First, a special notion is introduced to simplify the description of the test cases. Then, the test cases that correspond to the section 5.3 for the insert and remove conflict resolution are presented, followed by the test cases for the puzzles of section 5.1.1.

5.4.1 Notions

To describe the test cases we introduce a few simplified notions for the operations we use. The operation we need are:

- Insertion of a single character or a character string by typing
- Insertion of a character string by copy-paste from another source
- Removal of a single character or a character string
- Local undo
- Global undo
- Local redo
- Global redo

Furthermore, we have to describe the state of a document that means a representation of its content.

5.4.1.1 State of a Document

The content of a document will simply be presented as characters in quotation marks. For example:

$$S_1 = 'Tendax'$$

represents the state of a document that consist of the two special characters that mark the beginning and the end of a document (β , ε) and the characters T,e,n,d,a and x.

A document state can be given or is a result of an operation

$$S_0 = ''$$

$$O_1 \rightarrow S_1 = 'Tendax'$$

5.4.1.2 Keyboard Insertion

An insertion typed with the keyboard is depicted as

$$O_1 = Ins_{user}[p, K(string), n]$$

- 'Ins' = keyword for insertion
- 'user' = representation of the user that has triggered the operation
- 'K' = keyword for **K**eyboard insertion
- 'string' = the character string to be inserted
- 'p' = previous character to the character string
- 'n' = next character to the character string

This means that the string is inserted with single insertion operations for each character. Therefore, an undo operation can not just undo the whole operation O_1 but each single insertion of the characters. For example ($Undo(O_1, Ins_{user}[p, K(c), n])$) would undo the insertion of character 'c' that was inserted in operation O_1 .

5.4.1.3 Copy-Paste Insertion

An insertion that originates from a copy-paste action is depicted as a keyboard insertion but with the keyword 'P' for copy and **p**aste

$$O_1 = Ins_{user}[p, P(string), n]$$

- 'P' = keyword for copy-paste insertion

5.4.1.4 Removal

A removal is depicted as

$$O_1 = Del_{user}[p, string]$$

- 'Del' = keyword for removal (**delete**)
- 'user' = representation of the user that has triggered the operation
- 'string' = the character string that shall be removed
- 'p' = previous character to the character string

5.4.1.5 Local Undo

A local undo is depicted as

$$O_1 = LocUndo_{user}$$

- 'LocUndo' = keyword for **Local Undo**
- 'user' = representation of the user that has triggered the operation

5.4.1.6 Global Undo

A global undo is depicted as

$$O_1 = GlobUndo_{user}$$

- 'GlobUndo' = keyword for **Global Undo**
- 'user' = representation of the user that has triggered the operation

5.4.1.7 Local Redo

A local redo is depicted as

$$O_1 = LocRedo_{user}$$

- 'LocRedo' = keyword for **Local Redo**
- 'user' = representation of the user that has triggered the operation

5.4.1.8 Global Redo

A global redo is depicted as

$$O_1 = GlobRedo_{user}$$

- 'GlobRedo' = keyword for **Global Redo**
- 'user' = representation of the user that has triggered the operation

5.4.1.9 Example

An Example of a sequence of operation would be:

$$\begin{aligned} S_0 &= "" \\ O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\ O_2 &= Ins_1[n, P(PQR), d] \rightarrow S_2 = 'TenPQRdax' \\ O_3 &= Del_2[e, nPQRd] \rightarrow S_3 = 'Teax' \\ O_4 &= LocUndo_2 \rightarrow Undo(O_3) \\ &\rightarrow Undo(Del_2[e, nPQRd]) \rightarrow S_4 = S_2 = 'TenPQRdax' \end{aligned}$$

Where the last operation also could be abbreviated with:

$$O_4 = LocUndo_2 \rightarrow S_4 = 'TenPQRdax'$$

The example above represents the following situation:

User 1 inserts the characters T,e,n,d,a and x in an empty document (O_1). He then inserts the character string 'PQR' between 'n' and 'd' by copying and pasting it (O_2). Afterwards, user 2 removes the character string 'nPQRd' in one operation (O_3) and then triggers a local undo (O_4).

The test cases are sometimes completed with an extraction of the document and history content as it is stored in the database.

For the document content the following values are extracted (cf. left table in 5.3):

- After = the preceding character
- Char = the character
- Before = the succeeding character
- IsActive = the status in the document
- User = the User that has performed the operation

For the undo history content the following values are extracted (cf. right table in 5.3):

- After = the preceding character
- Char = the character
- Before = the succeeding character
- Action = the type of action according to the attribute **ActionId** (**ActionId**=1: insert action, **ActionId**=2: remove action)
- Relevant = the status in the document according to the attribute **IsActive** (**IsActive**=1: relevant for undo, **IsActive**=0: not relevant for undo)
- User = the User that has performed the operation
- CC = the character counter according to the attribute **CharCounter** that is used to determine characters that are part of the same operation

For the redo history content the same values are extracted as for the undo history but without the attribute **Action**.

The tables are always arranged in the same way. If there are three tables presented, the upper left is the document content, the upper right is the undo history content and the lower right is the redo history content. The redo history content is omitted if it is empty.

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.3: Example Database Content after Operation 1 (own contribution)

If multiple tables are presented that show the sequence of operations and their effect in the database, the fields that have changed are marked with a darker background color (cf. tables 5.4).

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	P	1	1
R	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
n	P	Q	1	1
P	Q	R	1	1
Q	R	d	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	P	Q	1	1	1	7
P	Q	R	1	1	1	7
Q	R	d	1	1	1	7

Table 5.4: Example Database Content after Operation 2 (own contribution)

5.4.2 Test Cases for Insert Conflict Resolution

In this subsection the test cases for each situation of the insert conflict resolution (cf. subsection 5.3.3) that can occur are presented. An overview of all situations is given in figure 5.4 on page 51. If a special strategy is used for a test case, it is written just above the test sequence. If nothing is annotated, the test case runs with DIP=0 and DIP=1. All test cases run with ERAE=1.

5.4.2.1 Insert 0-0

Insert 0-0 A

This test case is run with DIP=0 (search left).

$$S_0 = ''$$

$$O_1 = Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax'$$

$$O_2 = Del_1[n, d] \rightarrow S_2 = 'Tenax'$$

$$O_3 = Del_2[e, na] \rightarrow S_3 = 'Tex'$$

$$O_4 = Ins_2[e, K(PQR), x] \rightarrow S_4 = 'TePQRx'$$

$$O_5 = LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TedPQRx' (*)$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; n, a, n, a
 $\Rightarrow$ InsertCategory 4 - search left)

Insert 0-0 B

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[n, d] \rightarrow S_2 = 'Tenax' \\
O_3 &= Del_2[e, na] \rightarrow S_3 = 'Tex' \\
O_4 &= Ins_2[e, K(PQR), x] \rightarrow S_4 = 'TePQRx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TePQRdx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; n, a, n, a
 $\Rightarrow$ InsertCategory 5 - search right)

Insert 0-0 C

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(a), \varepsilon] \rightarrow S_1 = 'a' \\
O_2 &= Del_2[\beta, a] \rightarrow S_2 = "" \\
O_3 &= Ins_1[\beta, K(b), \varepsilon] \rightarrow S_3 = 'b' \\
O_4 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_4 = 'ab' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = a; β , b, b, ε
 $\Rightarrow$ InsertCategory 2 - biX)

Insert 0-0 D

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[a, x] \rightarrow S_2 = 'Tenda' \\
O_3 &= Del_2[d, a] \rightarrow S_3 = 'Tend' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tendx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = x; a, ε , d, ε
 $\Rightarrow$ InsertCategory 3 - Yie)

5.4.2.2 Insert 1-1**Insert 1-1 A**

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, n] \rightarrow S_2 = 'Tedx' \\
O_3 &= Del_2[e, d] \rightarrow S_3 = 'Teax' \\
O_4 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_4 = 'Tedx' (*) \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'Tendax' (**)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, a, e, a

$\Rightarrow$ InsertCategory 1 - BiE)

(**: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, d, e, d

$\Rightarrow$ InsertCategory 1 - BiE)

Insert 1-1 B

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, nd] \rightarrow S_2 = 'Teax' \\
O_3 &= Ins_2[e, K(D), a] \rightarrow S_3 = 'TeDax' \\
O_4 &= Ins_2[e, K(N), D] \rightarrow S_4 = 'TeNDax' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TendNDax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, N, D, a

$\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-1 C

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, nd] \rightarrow S_2 = 'Teax' \\
O_3 &= Ins_2[e, K(D), a] \rightarrow S_3 = 'TeDax' \\
O_4 &= Ins_2[e, K(N), D] \rightarrow S_4 = 'TeNDax' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TeNDndax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, N, D, a

$\Rightarrow$ InsertCategory 3 - YiE)

Insert 1-1 D

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, nd] \rightarrow S_2 = 'Teax' \\
O_3 &= Ins_2[e, K(ND), a] \rightarrow S_3 = 'TeNDax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'TendNDax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, N, D, a
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-1 E

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[e, nd] \rightarrow S_2 = 'Teax' \\
O_3 &= Ins_1[e, K(ND), a] \rightarrow S_3 = 'TeNDax' \\
O_4 &= Del_1[e, ND] \rightarrow S_4 = 'Teax' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_5 = 'Tendax' \\
O_6 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_6 = 'TendNDax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = N; e, n, d, a
 $\Rightarrow$ InsertCategory 3 - YiE)

Insert 1-1 F

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[e, n] \rightarrow S_2 = 'Tedx' \\
O_3 &= Del_2[d, a] \rightarrow S_3 = 'Tedx' \\
O_4 &= Del_1[e, d] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Teax' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_6 = 'Tenax' \\
O_7 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_7 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8a - Xi)

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[e, n] \rightarrow S_2 = 'Tedx' \\
O_3 &= Del_2[d, a] \rightarrow S_3 = 'Tedx' \\
O_4 &= Del_1[e, d] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Teax' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_6 = 'Tenax' \\
O_7 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_7 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8b - iY)

Insert 1-1 G

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[n, da] \rightarrow S_2 = 'Tenx' \\
O_3 &= Del_1[e, n] \rightarrow S_3 = 'Tex' \\
O_4 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tedx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, d, a, x
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-1 H

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[e, nd] \rightarrow S_2 = 'Teax' \\
O_3 &= Del_1[e, a] \rightarrow S_3 = 'Tex' \\
O_4 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tendx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = a; e, n, d, x
 $\Rightarrow$ InsertCategory 3 - YiE)

Insert 1-1 I

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[d, a] \rightarrow S_2 = 'Tendx' \\
O_3 &= Del_2[e, n] \rightarrow S_3 = 'Tedx' \\
O_4 &= Del_1[e, d] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tenx' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_6 = 'Tenax' \\
O_7 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_7 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8a - Xi)

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[d, a] \rightarrow S_2 = 'Tendx' \\
O_3 &= Del_2[e, n] \rightarrow S_3 = 'Tedx' \\
O_4 &= Del_1[e, d] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tenx' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_6 = 'Tenax' \\
O_7 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_7 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8b - iY)

Insert 1-1 J

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tend), \varepsilon] \rightarrow S_1 = 'Tend' \\
O_2 &= Ins_1[d, P(ax), \varepsilon] \rightarrow S_2 = 'Tendax' \\
O_3 &= Del_2[n, da] \rightarrow S_3 = 'Tenx' \\
O_4 &= Del_1[e, n] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tedx' \\
O_6 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_6 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, d, a, x
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-1 K

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tend), \varepsilon] & \rightarrow S_1 &= 'Tend' \\
O_2 &= Ins_1[d, P(ax), \varepsilon] & \rightarrow S_2 &= 'Tendax' \\
O_3 &= Del_2[d, a] & \rightarrow S_3 &= 'Tendx' \\
O_4 &= Del_2[e, n] & \rightarrow S_4 &= 'Tedx' \\
O_5 &= Del_1[e, d] & \rightarrow S_5 &= 'Tex' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_4) & \rightarrow S_6 &= 'Tenx' \\
O_7 &= LocUndo_2 \rightarrow Undo(O_3) & \rightarrow S_7 &= 'Tenax' \\
O_8 &= LocUndo_1 \rightarrow Undo(O_5) & \rightarrow S_8 &= 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8a - Xi)

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Tend), \varepsilon] & \rightarrow S_1 &= 'Tend' \\
O_2 &= Ins_1[d, P(ax), \varepsilon] & \rightarrow S_2 &= 'Tendax' \\
O_3 &= Del_2[d, a] & \rightarrow S_3 &= 'Tendx' \\
O_4 &= Del_2[e, n] & \rightarrow S_4 &= 'Tedx' \\
O_5 &= Del_1[e, d] & \rightarrow S_5 &= 'Tex' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_4) & \rightarrow S_6 &= 'Tenx' \\
O_7 &= LocUndo_2 \rightarrow Undo(O_3) & \rightarrow S_7 &= 'Tenax' \\
O_8 &= LocUndo_1 \rightarrow Undo(O_5) & \rightarrow S_8 &= 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8b - iY)

Insert 1-1 L

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, P(Ten), \varepsilon] & \rightarrow S_1 &= 'Ten' \\
O_2 &= Ins_1[n, P(dax), \varepsilon] & \rightarrow S_2 &= 'Tendax' \\
O_3 &= Del_2[d, a] & \rightarrow S_3 &= 'Tendx' \\
O_4 &= Del_2[e, n] & \rightarrow S_4 &= 'Tedx' \\
O_5 &= Del_1[e, d] & \rightarrow S_5 &= 'Tex'
\end{aligned}$$

$$\begin{aligned}
O_6 &= LocUndo_2 \rightarrow Undo(O_4) \rightarrow S_6 = 'Tenx' \\
O_7 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_7 = 'Tenax' \\
O_8 &= LocUndo_1 \rightarrow Undo(O_5) \rightarrow S_8 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8a - Xi)

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, P(Ten), \varepsilon] \rightarrow S_1 = 'Ten' \\
O_2 &= Ins_1[n, P(dax), \varepsilon] \rightarrow S_2 = 'Tendax' \\
O_3 &= Del_2[d, a] \rightarrow S_3 = 'Tendx' \\
O_4 &= Del_2[e, n] \rightarrow S_4 = 'Tedx' \\
O_5 &= Del_1[e, d] \rightarrow S_5 = 'Tex' \\
O_6 &= LocUndo_2 \rightarrow Undo(O_4) \rightarrow S_6 = 'Tenx' \\
O_7 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_7 = 'Tenax' \\
O_8 &= LocUndo_1 \rightarrow Undo(O_5) \rightarrow S_8 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; e, n, a, x
 $\Rightarrow$ InsertCategory 8b - iY)

Insert 1-1 M

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, P(Ten), \varepsilon] \rightarrow S_1 = 'Ten' \\
O_2 &= Ins_1[n, P(dax), \varepsilon] \rightarrow S_2 = 'Tendax' \\
O_3 &= Del_2[e, nd] \rightarrow S_3 = 'Teax' \\
O_4 &= Del_1[e, a] \rightarrow S_4 = 'Tex' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tendx' \\
O_6 &= LocUndo_1 \rightarrow Undo(O_4) \rightarrow S_6 = 'Tendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = a; e, n, d, x
 $\Rightarrow$ InsertCategory 3 - YiE)

5.4.2.3 Insert 1-0**Insert 1-0 A**

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(ab), \varepsilon] \quad \rightarrow S_1 = 'ab' \\
O_2 &= Del_2[a, b] \quad \rightarrow S_2 = 'a' \\
O_3 &= Del_1[\beta, a] \quad \rightarrow S_3 = '' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_3) \rightarrow S_4 = 'a' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_5 = 'ab' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = b; a, ε , a, ε
 $\Rightarrow$ InsertCategory 1 - Bie)

Insert 1-0 B

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \quad \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, n] \quad \rightarrow S_2 = 'Tedx' \\
O_3 &= Del_2[e, d] \quad \rightarrow S_3 = 'Teax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tenax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = n; e, a, e, d
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-0 C

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \quad \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[n, d] \quad \rightarrow S_2 = 'Tenax' \\
O_3 &= Ins_2[n, K(PQR), a] \quad \rightarrow S_3 = 'TenPQRax' \\
O_4 &= Del_2[Q, Ra] \quad \rightarrow S_4 = 'TenPQx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TendPQx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; n, P, R, a
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-0 D

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[a, x] \rightarrow S_2 = 'Tenda' \\
O_3 &= Ins_2[a, K(PQ), \varepsilon] \rightarrow S_3 = 'TendaPQ' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'TendaxPQ' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = x; a, P, Q, ε
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 1-0 E

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[a, x] \rightarrow S_2 = 'Tenda' \\
O_3 &= Ins_2[a, K(PQ), \varepsilon] \rightarrow S_3 = 'TendaPQ' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'TendaPQx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = x; a, P, Q, ε
 $\Rightarrow$ InsertCategory 3 - YiE)

5.4.2.4 Insert 0-1**Insert 0-1 A**

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(ab), \varepsilon] \rightarrow S_1 = 'ab' \\
O_2 &= Del_2[\beta, a] \rightarrow S_2 = 'b' \\
O_3 &= Del_1[\beta, b] \rightarrow S_3 = "" \\
O_4 &= LocUndo_1 \rightarrow Undo(O_3) \rightarrow S_4 = 'b' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_5 = 'ab' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = a; β , b, β , b
 $\Rightarrow$ InsertCategory 1 - biE)

Insert 0-1 B

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[n, d] \rightarrow S_2 = 'Tenax' \\
O_3 &= Del_2[e, n] \rightarrow S_3 = 'Teax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tedx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; n, a, e, a
 $\Rightarrow$ InsertCategory 3 - YiE)

Insert 0-1 C

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[n, d] \rightarrow S_2 = 'Tenax' \\
O_3 &= Ins_2[n, K(PQR), a] \rightarrow S_3 = 'TenPQRax' \\
O_4 &= Del_2[e, nP] \rightarrow S_4 = 'TeQRax' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TeQRdax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = d; n, P, R, a
 $\Rightarrow$ InsertCategory 3 - YiE)

Insert 0-1 D

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[\beta, T] \rightarrow S_2 = 'endax' \\
O_3 &= Ins_2[\beta, K(PQ), e] \rightarrow S_3 = 'PQendax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'TPQendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = T; β , P, Q, e
 $\Rightarrow$ InsertCategory 2 - BiX)

Insert 0-1 E

This test case is run with DIP=1 (search right).

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[\beta, T] \rightarrow S_2 = 'endax'
\end{aligned}$$

$$\begin{aligned}
O_3 &= Ins_2[\beta, K(PQ), e] & \rightarrow S_3 &= 'PQendax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 &= 'PQTendax' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = T; β , P, Q, e
 $\Rightarrow$ InsertCategory 3 - YiE)

5.4.3 Test Cases for Remove Conflict Resolution

In this subsection the test cases for the remove conflict resolution (cf. subsection 5.3.4 on pages 63 et seqq.) are presented. If the strategy ANCR=1 is used for a test case, it is written just above the test sequence. If nothing is annotated, the test case runs with ANCR=0 and ERAE=1.

5.4.3.1 Remove A

This is the same test case as it is presented for the Insert-delete-undo dilemma on page 88.

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] & \rightarrow S_1 &= 'Tendax' \\
O_2 &= Del_2[n, da] & \rightarrow S_2 &= 'Tenx' \\
O_3 &= LocUndo_1 \rightarrow Undo(O_1, Ins_1[a, K(x), \varepsilon]) & \rightarrow S_3 &= 'Ten' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_1, Ins_1[e, K(n), \varepsilon]) & \rightarrow S_4 &= 'Te' (*)
\end{aligned}$$

(*: Two characters 'da', which are already deleted by another user are skipped ($Undo(O_1, Ins_1[d, K(a), \varepsilon])$, $Undo(O_1, Ins_1[n, K(d), \varepsilon])$) and the next insertion is undone, i.e. the one of character 'n' ($Undo(O_1, Ins_1[e, K(n), \varepsilon])$))

5.4.3.2 Remove B

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Te), \varepsilon] & \rightarrow S_1 &= 'Te' \\
O_2 &= Ins_1[\beta, P(nda), \varepsilon] & \rightarrow S_2 &= 'Tenda' \\
O_3 &= Ins_1[\beta, K(x), \varepsilon] & \rightarrow S_3 &= 'Tendax' \\
O_4 &= Del_2[n, d] & \rightarrow S_4 &= 'Tenax' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_3) & \rightarrow S_5 &= 'Tena' \\
O_6 &= LocUndo_1 \rightarrow Undo(O_1, Ins_1[T, K(e), \varepsilon]) \rightarrow S_6 &= 'Tna' (*) \\
O_7 &= LocUndo_1 \rightarrow Undo(O_1, Ins_1[\beta, K(T), \varepsilon]) \rightarrow S_7 &= 'na'
\end{aligned}$$

(*: $Undo(O_2)$ is skipped because the character 'd' of the string 'nda' is already deleted and the next insertion is undone, i.e. the one of character 'e'
 $(Undo(O_1, Ins_1[T, K(e), \varepsilon]))$)

5.4.3.3 Remove C

This test case is run with ANCR=0.

$$\begin{aligned}
 S_0 &= "" \\
 O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
 O_2 &= Ins_1[n, P(PQRS), d] \rightarrow S_2 = 'TenPQRSdax' \\
 O_3 &= Ins_2[R, K(12), Q] \rightarrow S_3 = 'TenPQ12RSdax' \\
 O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'TenPQ12RSdax' (*)
 \end{aligned}$$

(*: The undo ($Undo(Ins_1[n, P(PQRS), d])$) is aborted because the string '12' is in between the character string ('PQ12RS') and the removal of non consecutive characters is not allowed. The system presents the warning: 'Undo action not possible due to a conflict')

5.4.3.4 Remove D

$$\begin{aligned}
 S_0 &= "" \\
 O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
 O_2 &= Ins_1[n, P(PQRS), d] \rightarrow S_2 = 'TenPQRSdax' \\
 O_3 &= Ins_2[R, K(12), Q] \rightarrow S_3 = 'TenPQ12RSdax' \\
 O_4 &= LocUndo_2 \rightarrow Undo(O_3) \rightarrow S_4 = 'TenPQ1RSdax' (*)
 \end{aligned}$$

(*: The undo ($Undo(O_3, Ins_2[1, K(2), Q])$) is performed (no special case))

5.4.3.5 Remove E

This is the same test case as it is presented for the copy-paste-insert dilemma on page 89.

This test case is run with ANCR=1.

$$\begin{aligned}
 S_0 &= "" \\
 O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
 O_2 &= Ins_1[n, P(PQRS), d] \rightarrow S_2 = 'TenPQRSdax' \\
 O_3 &= Ins_2[R, K(12), Q] \rightarrow S_3 = 'TenPQ12RSdax' \\
 O_4 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'Ten12dax' (*)
 \end{aligned}$$

(*: The undo ($Undo(Ins_1[n, P(PQRS), d])$) is performed although the string '12' is in between the character string ('PQ12RS') since the removal of non consecutive characters is allowed)

5.4.3.6 Remove F

This test case is run with ANCR=1.

$$\begin{aligned}
 S_0 &= "" \\
 O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] & \rightarrow S_1 &= 'Tendax' \\
 O_2 &= Ins_1[n, P(ABCDEFGF), d] & \rightarrow S_2 &= 'TenABCDEFGFgdx' \\
 O_3 &= Ins_2[A, K(1), B] & \rightarrow S_3 &= 'TenA1BCDEFgdx' \\
 O_4 &= Ins_2[C, K(23), D] & \rightarrow S_4 &= 'TenA1BC23DEFgdx' \\
 O_5 &= Ins_2[E, K(4), F] & \rightarrow S_5 &= 'TenA1BC23DE4FGdx' \\
 O_6 &= LocUndo_1 \rightarrow Undo(O_2) & \rightarrow S_6 &= 'Ten1234dx' (*)
 \end{aligned}$$

(*: The undo ($Ins_1[n, P(ABCDEFGF), d]$) is performed although there are characters in between the string given that the removal of non consecutive characters is allowed)

Operation 4 could also be a copy-paste operation ($O_4 = Ins_2[C, P(23), D]$), the result would be the same.

5.4.4 Test Cases for Puzzles

In this subsection we present, for each puzzle that is described in section 5.1.1 on page 38, one test case and show an extraction of the database. If the strategy DIP=1 or ANCR=1 is used for a test case, it is written just above the test sequence. If nothing is annotated, the test case runs with DIP=0, ANCR=0 and ERAE=1.

5.4.4.1 Ambiguous Reference

$$\begin{aligned}
 S_0 &= "" \\
 O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] & \rightarrow S_1 &= 'Tendax' \\
 O_2 &= Del_2[n, d] & \rightarrow S_2 &= 'Tenax' \\
 O_3 &= Ins_1[n, K(D), a] & \rightarrow S_3 &= 'TenDax' \\
 O_4 &= LocUndo_2 \rightarrow Undo(O_2) & \rightarrow S_4 &= 'TendDax' (*)
 \end{aligned}$$

(*: Character 'd' could also be inserted between 'D' and 'a' what would result in $S_4 = 'TenDdax'$)

The enhanced insert conflict resolution of the unDOit algorithm inserts 'd' between

'n' and 'd' because 'd' was inserted after 'n' but before 'D'. The simple insert conflict resolution of the unDOit algorithm inserts 'd' according to the DIP. If DIP=0 'd' is inserted left of 'D' and if DIP=1 'd' is inserted right of 'D'.

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.5: Ambiguous Reference Database Content after Operation 1 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	a	1	1
n	d	a	0	1
n	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	1	7

Table 5.6: Ambiguous Reference Database Content after Operation 2 (own contribution)

After	Char	Before	IsActive	User
n	D	a	1	2
BOF	T	e	1	1
T	e	n	1	1
e	n	D	1	1
n	d	a	0	1
D	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	1	7
n	D	a	1	1	2	8

Table 5.7: Ambiguous Reference Database Content after Operation 3 (own contribution)

After	Char	Before	IsActive	User
d	D	a	1	2
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	D	1	1
D	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	0	1	7
n	D	a	1	1	2	8

After	Char	Before	Action	User	CC
n	d	a	2	1	1

Table 5.8: Ambiguous Reference Database Content after Operation 4 (own contribution)

5.4.4.2 Insert-insert-tie Dilemma

The Insert-insert-tie dilemma can not be provoked in TeNDaX.

5.4.4.3 Do-undo-pair Trap

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] & \rightarrow S_1 &= 'Tendax' \\
O_2 &= Del_1[e, n] & \rightarrow S_2 &= 'Tedx' \\
O_3 &= Del_2[e, d] & \rightarrow S_3 &= 'Teax' \\
O_4 &= LocUndo_1 \rightarrow Undo(O_2) & \rightarrow S_4 &= 'Tenax' \\
O_5 &= LocUndo_2 \rightarrow Undo(O_3) & \rightarrow S_5 &= 'Tendax' (*)
\end{aligned}$$

(*: Character 'd' could also be inserted between 'e' and 'n' what would result in $S_5 = 'Tednax'$)

The enhanced insert conflict resolution of the unDOit algorithm inserts 'd' between 'n' and 'a' because 'n' is in between the two active reference characters 'e' and 'a' but 'd' was inserted after 'n'. The simple insert conflict resolution of the unDOit algorithm inserts 'd' according to the DIP. If DIP=0 'd' is inserted left of 'n' and if DIP=1 'd' is inserted right of 'n'.

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.9: Do-undo-pair trap Database Content after Operation 1 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	d	1	1
e	n	d	0	1
e	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
e	n	d	2	1	1	7

Table 5.10: Do-undo-pair trap Database Content after Operation 2 (own contribution)

After	Char	Before	IsActive	User	After	Char	Before	Action	IsActive	User	CC
BOF	T	e	1	1	BOF	T	EOF	1	1	1	1
T	e	a	1	1	T	e	EOF	1	1	1	2
e	n	d	0	1	e	n	EOF	1	1	1	3
e	d	a	0	1	n	d	EOF	1	1	1	4
e	a	x	1	1	d	a	EOF	1	1	1	5
a	x	EOF	1	1	a	x	EOF	1	1	1	6
EOF	BOF	T	0	0	e	n	d	2	1	1	7
x	EOF	BOF	0	0	e	d	a	2	1	2	8

Table 5.11: Do-undo-pair trap Database Content after Operation 3 (own contribution)

After	Char	Before	IsActive	User	After	Char	Before	Action	IsActive	User	CC
BOF	T	e	1	1	BOF	T	EOF	1	1	1	1
T	e	n	1	1	T	e	EOF	1	1	1	2
e	n	a	1	1	e	n	EOF	1	1	1	3
e	d	a	0	1	n	d	EOF	1	1	1	4
n	a	x	1	1	d	a	EOF	1	1	1	5
a	x	EOF	1	1	a	x	EOF	1	1	1	6
EOF	BOF	T	0	0	e	n	d	2	0	1	7
x	EOF	BOF	0	0	e	d	a	2	1	2	8

After	Char	Before	Action	User	CC
e	n	d	2	1	1

Table 5.12: Do-undo-pair trap Database Content after Operation 4 (own contribution)

After	Char	Before	IsActive	User	After	Char	Before	Action	IsActive	User	CC
BOF	T	e	1	1	BOF	T	EOF	1	1	1	1
T	e	n	1	1	T	e	EOF	1	1	1	2
e	n	d	1	1	e	n	EOF	1	1	1	3
n	d	a	1	1	n	d	EOF	1	1	1	4
d	a	x	1	1	d	a	EOF	1	1	1	5
a	x	EOF	1	1	a	x	EOF	1	1	1	6
EOF	BOF	T	0	0	e	n	d	2	0	1	7
x	EOF	BOF	0	0	e	d	a	2	0	2	8

After	Char	Before	Action	User	CC
e	n	d	2	1	1
e	d	a	2	2	2

Table 5.13: Do-undo-pair trap Database Content after Operation 5 (own contribution)

5.4.4.4 Overlapping Deletes Puzzle or Del-del-undo Puzzle

The Overlapping deletes puzzle can not be provoked in TeNDaX.

5.4.4.5 Insert-delete-undo Dilemma

$$S_0 = ''$$

$$O_1 = Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax'$$

$$O_2 = Del_2[n, da] \rightarrow S_2 = 'Tenx'$$

$$O_3 = LocUndo_1 \rightarrow Undo(O_1, Ins_1[a, K(x), \varepsilon]) \rightarrow S_3 = 'Ten'$$

$$O_4 = LocUndo_1 \rightarrow Undo(O_1, Ins_1[e, K(n), \varepsilon]) \rightarrow S_4 = 'Te' (*)$$

(*: Two characters 'da', which are already deleted by another user are skipped ($Undo(O_1, Ins_1[d, K(a), \varepsilon])$, $Undo(O_1, Ins_1[n, K(d), \varepsilon])$) and the next insertion is undone, the one of character 'n' ($Undo(O_1, Ins_1[e, K(n), \varepsilon])$))

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.14: Insert-delete-undo dilemma Database Content after Operation 1 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	x	1	1
n	d	a	0	1
d	a	x	0	1
n	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	2	7
d	a	x	2	1	2	7

Table 5.15: Insert-delete-undo dilemma Database Content after Operation 2 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	EOF	1	1
n	d	a	0	1
d	a	x	0	1
n	x	EOF	0	1
EOF	BOF	T	0	0
n	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	0	1	6
n	d	a	2	1	2	7
d	a	x	2	1	2	7

After	Char	Before	Action	User	CC
n	x	EOF	1	1	1

Table 5.16: Insert-delete-undo dilemma Database Content after Operation 3 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	EOF	1	1
e	n	EOF	0	1
n	d	a	0	1
d	a	x	0	1
n	x	EOF	0	1
EOF	BOF	T	0	0
e	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	0	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	0	1	6
n	d	a	2	1	2	7
d	a	x	2	1	2	7

After	Char	Before	Action	User	CC
n	x	EOF	1	1	1
e	n	EOF	1	1	2

Table 5.17: Insert-delete-undo dilemma Database Content after Operation 4 (own contribution)

5.4.4.6 Copy-paste-insert Dilemma

This test case is run with ANCR=1.

$$S_0 = ''$$

$$O_1 = Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax'$$

$$O_2 = Ins_1[n, P(PQRS), d] \rightarrow S_2 = 'TenPQRSdax'$$

$$O_3 = Ins_2[R, K(12), Q] \rightarrow S_3 = 'TenPQ12RSdax'$$

$$O_4 = LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_4 = 'Ten12dax' (*)$$

(*: The enhanced remove conflict resolution of the unDOit algorithm (ANCR=1) allows the removal of non consecutive characters. Thus, the undo ($Undo(Ins_1[n, P(PQRS), d])$) is performed although the string '12' is in between the character string ('PQ12RS'). If the removal of non consecutive characters would not be allowed (ANCR=0), the system would not perform the local undo, but present a warning to user 1 saying that the undo can not be performed because of a conflict.

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.18: Copy-paste-insert dilemma Database Content after Operation 1 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	P	1	1
S	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
n	P	Q	1	1
P	Q	R	1	1
Q	R	S	1	1
R	S	d	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	P	Q	1	1	1	7
P	Q	R	1	1	1	7
Q	R	S	1	1	1	7
R	S	d	1	1	1	7

Table 5.19: Copy-paste-insert dilemma Database Content after Operation 2 (own contribution)

After	Char	Before	IsActive	User
Q	1	2	1	2
1	2	R	1	2
BOF	T	e	1	1
T	e	n	1	1
e	n	P	1	1
S	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
n	P	Q	1	1
P	Q	1	1	1
2	R	S	1	1
R	S	d	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	P	Q	1	1	1	7
P	Q	R	1	1	1	7
Q	R	S	1	1	1	7
R	S	d	1	1	1	7
Q	1	R	1	1	2	8
1	2	R	1	1	2	9

Table 5.20: Copy-paste-insert dilemma Database Content after Operation 3 (own contribution)

After	Char	Before	IsActive	User
n	1	2	1	2
1	2	d	1	2
BOF	T	e	1	1
T	e	n	1	1
e	n	1	1	1
2	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
n	P	Q	0	1
P	Q	R	0	1
Q	R	S	0	1
R	S	d	0	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	P	Q	1	0	1	7
P	Q	R	1	0	1	7
Q	R	S	1	0	1	7
R	S	d	1	0	1	7
Q	1	R	1	1	2	8
1	2	R	1	1	2	9

After	Char	Before	Action	User	CC
n	P	Q	1	1	1
P	Q	1	1	1	1
2	R	S	1	1	1
R	S	d	1	1	1

Table 5.21: Copy-paste-insert dilemma Database Content after Operation 4 (own contribution)

5.4.4.7 Reference Character Not Existent Anymore

$$S_0 = "$$

$$O_1 = Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax'$$

$$O_2 = Del_1[n, d] \rightarrow S_2 = 'Tenax'$$

$$O_3 = Del_2[e, na] \rightarrow S_3 = 'Tex'$$

$$O_4 = Ins_2[e, K(PQR), x] \rightarrow S_4 = 'TePQRx'$$

$$O_5 = LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'TedPQRx' (*)$$

(*: Character 'd' has 'n' and 'a' as its two reference characters but they are both deleted. Therefore, 'd' could also be inserted between 'R' and 'x' what would result in $S_5 = 'TePQRdx'$)

The enhanced and the simple insert conflict resolution of the unDOit algorithm insert 'd' depending on the DIP. If DIP=0 'd' is inserted left of 'PQR' and if DIP=1 'd' is inserted right of 'PQR'.

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	d	1	1
n	d	a	1	1
d	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6

Table 5.22: Reference Character Not Existent Anymore Database Content after Operation 1 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	n	1	1
e	n	a	1	1
n	d	a	0	1
n	a	x	1	1
a	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	1	7

Table 5.23: Reference Character Not Existent Anymore Database Content after Operation 2 (own contribution)

After	Char	Before	IsActive	User
BOF	T	e	1	1
T	e	x	1	1
e	n	a	0	1
n	d	a	0	1
n	a	x	0	1
e	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	1	7
e	n	a	2	1	2	8
n	a	x	2	1	2	8

Table 5.24: Reference Character Not Existent Anymore Database Content after Operation 3 (own contribution)

After	Char	Before	IsActive	User
e	P	Q	1	2
P	Q	R	1	2
Q	R	x	1	2
BOF	T	e	1	1
T	e	P	1	1
e	n	a	0	1
n	d	a	0	1
n	a	x	0	1
R	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	1	1	7
e	n	a	2	1	2	8
n	a	x	2	1	2	8
e	P	x	1	1	2	9
P	Q	x	1	1	2	10
Q	R	x	1	1	2	11

Table 5.25: Reference Character Not Existent Anymore Database Content after Operation 4 (own contribution)

After	Char	Before	IsActive	User
d	P	Q	1	2
P	Q	R	1	2
Q	R	x	1	2
BOF	T	e	1	1
T	e	d	1	1
e	n	a	0	1
e	d	P	1	1
n	a	x	0	1
R	x	EOF	1	1
EOF	BOF	T	0	0
x	EOF	BOF	0	0

After	Char	Before	Action	IsActive	User	CC
BOF	T	EOF	1	1	1	1
T	e	EOF	1	1	1	2
e	n	EOF	1	1	1	3
n	d	EOF	1	1	1	4
d	a	EOF	1	1	1	5
a	x	EOF	1	1	1	6
n	d	a	2	0	1	7
e	n	a	2	1	2	8
n	a	x	2	1	2	8
e	P	x	1	1	2	9
P	Q	x	1	1	2	10
Q	R	x	1	1	2	11

After	Char	Before	Action	User	CC
n	d	a	2	1	1

Table 5.26: Reference Character Not Existent Anymore Database Content after Operation 5 (own contribution)

5.4.5 Test Case for Redo

If a redo is triggered, the algorithms that are invoked are almost the same as the ones invoked by an undo. Therefore, we present in this subsection just one example of a redo test case.

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_1[e, n] \rightarrow S_2 = 'Tedx' \\
O_3 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_3 = 'Tendax' \\
O_4 &= LocRedo_1 \rightarrow Redo(O_1) \rightarrow S_4 = 'Tedx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_5 = 'Tendax' \\
O_6 &= LocRedo_1 \rightarrow Redo(O_1) \rightarrow S_6 = 'Tedx'
\end{aligned}$$

5.4.6 Issues

Situations can occur, in which the algorithm can not guarantee that the user's intention of an undo operation is achieved, since the information is not sufficient that can be gathered from the meta data or the resolution of that kind of conflict is not implemented yet. The test cases in this subsection present such situations.

5.4.6.1 Insertion in one Part

The characters of a string are always inserted in one part. Cases exist, where it would be reasonable to split the string into single characters and check were to insert them but this resolution is not implemented yet. Thus, the following problems occur:

- when reinserting an area with non consecutive characters by triggering an undo of the insertion and then redoing the insertion
- when reference characters of deleted characters are removed in one action and afterwards all character removals are undone

This test case is run with ANCR=1.

$$\begin{aligned}
S_0 &= "" \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Ins_1[n, P(ABCDEFGFG), d] \rightarrow S_2 = 'TenABCDEFGFGdax' \\
O_3 &= Ins_2[A, K(1), B] \rightarrow S_3 = 'TenA1BCDEFFGdax' \\
O_4 &= Ins_2[C, K(23), D] \rightarrow S_4 = 'TenA1BC23DEFFGdax' \\
O_5 &= Ins_2[E, K(4), F] \rightarrow S_5 = 'TenA1BC23DE4FGdax'
\end{aligned}$$

$$\begin{aligned}
O_6 &= LocUndo_1 \rightarrow Undo(O_2) \rightarrow S_6 = 'Ten1234dax' (*) \\
O_7 &= LocRedo_1 \rightarrow Redo(O_2) \rightarrow S_7 = 'TenABCDEFGH1234dax' (**)
\end{aligned}$$

(*: The undo ($Ins_1[n, P(ABCDEFGH), d]$) is performed although there are characters in between the string because the removal of non consecutive characters is allowed)

(**: The redo of the insertion ($Ins_1[n, P(ABCDEFGH), d]$) cannot be split up in single characters and is therefore inserted as one string ($S_7 \neq S_5$))

This test case is run with DIP=0 (search left).

$$\begin{aligned}
S_0 &= '' \\
O_1 &= Ins_1[\beta, K(Tendax), \varepsilon] \rightarrow S_1 = 'Tendax' \\
O_2 &= Del_2[n, d] \rightarrow S_2 = 'Tendax' \\
O_3 &= Del_1[e, na] \rightarrow S_3 = 'Tex' \\
O_4 &= LocUndo_2 \rightarrow Undo(O_2) \rightarrow S_4 = 'Tedx' \\
O_5 &= LocUndo_1 \rightarrow Undo(O_3) \rightarrow S_5 = 'Tenadx' (*)
\end{aligned}$$

(*: Fid; FORC-L, SORC-L, SORC-R, FORC-R = T; β , P, Q, e
 $\Rightarrow$ InsertCategory 3 - YiE)

5.5 Summary

One of the main topics of this thesis is the conflict resolution in the collaborative undo/redo of the TeNDaX system, which was presented in this chapter. The issues in the former framework have been described, considering the conflicting situations (known as puzzles) ambiguous reference, insert-insert-tie dilemma, do-undo-pair trap, overlapping deletes puzzle, insert-delete-undo dilemma, copy-paste-insert dilemma and reference character not existent anymore.

The conflict resolution in the TeNDaX system is embedded in the appropriate class methods that provide the operations, which shall be undone/redone, namely the insertion and removal of characters.

The concurrency control for an insertion is quite different and more complex than for a removal, not just due to the fact that the TeNDaX system stores characters in a ring of double linked fields. Thus, a character reference model was introduced to describe the different situations if characters shall be reinserted.

A configurable conflict resolution was presented that provide the possibility to react different to occurring conflicts depending on the three preferences default insert policy, allow non consecutive removal and empty redohistory after edit.

The last section represents a collection of test cases that cover the conflicts, which the concurrency control encounters, if an undo of an insertion or removal is triggered.

Chapter 6

Formal Depiction

In this section various formal definitions are introduced that build upon the definitions of section 2.2 on pages 10 et seqq. In a first part the insert and remove algorithms of the TeNDaX system are described with the help of those definitions. Then, the TeNDaX unDOit algorithm is formally defined in separated sections for undo, redo and the invocations of insert and remove from an undo/redo. In several cases, the formal depiction is completed with the appropriate SQL¹ statements that are used to query the database.

6.1 Notions

The algorithms described later on rely on the database based approach of the TeNDaX system. Thus, the relational algebra² is used for the definitions. In the following description, the objects and functions used for insertion and removal are portrayed. When it comes to the depiction of undo and redo, the objects and functions are extended in the appropriate subsections. All the objects are also used to represent their OID.

The symbol u stands for the object 'user' that executes a certain action.

$$u = user$$

The symbol d stands for the object 'document', on which a certain action is executed.

$$d = document$$

¹SQL: **S**tructured **Q**uery **L**anguage is both an ANSI and an ISO standard. It is a programming language for getting information from and updating a database.

²For an introduction to the relational algebra it is referred to [Dittrich, 2004, pages 32-39].

The symbol s stands for the object 'document session'.

$$s = \textit{document session}$$

The symbol c stands for the object 'character'.

$$c = \textit{character}$$

The symbol p stands for the previous character of a character object.

$$p = \textit{previous character}$$

The symbol n stands for the next character of a character object.

$$n = \textit{next character}$$

The symbol l stands for a list of character objects.

$$l = \textit{list of characters}$$

The symbol c_1 stands for the first character in the list l .

$$c_1 = \textit{first character in list } l$$

The symbol c_i stands for a character in the list l at the position i , whereas i is a value between 1 and the length of the list l .

$$c_i = \textit{character at position } i \textit{ in list } l$$

The symbol c_n stands for the last character in the list l .

$$c_n = \textit{last character in list } l$$

The symbol β stands for the special character that marks the beginning of a document (CharacterValue = 99998).

$$\beta = \textit{beginning of document}$$

The symbol ε stands for the special character that marks the end of a document (CharacterValue = 99999).

$$\varepsilon = \textit{end of document}$$

The symbol e stands for the object 'exception'.

$$e = \textit{exception}$$

The symbol I stands for an integer.

$$I = \text{integer}$$

The expression $l.c_i$ represents the character c_i of the character list l .

$$l.c_i = \text{character } c_i \text{ of list } l$$

The function *startTA* starts a transaction.

$$\text{startTA} = \text{start transaction}$$

The function *commitTA* commits a transaction that was started.

$$\text{commitTA} = \text{commit transaction}$$

The function *rollbackTA* performs a rollback of a transaction that was started.

$$\text{rollbackTA} = \text{perform rollback}$$

The function *checkWriteAccess* checks if the write access for a document session s is granted.

$$\text{checkWriteAccess}(s) = \text{check if write access for document session } s \text{ is granted}$$

The function *lock* acquires an exclusive lock³ for a character c and returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{lock}(c) &= \text{acquire the lock for character } c \\ \text{success} : \text{return } 1, \text{ no success} : \text{return } 0 \end{aligned}$$

The function *releaseLocks* releases all locks that a transaction has acquired so far.

$$\text{releaseLocks} = \text{release all locks}$$

The function *increment* increases the value of an integer I by one.

$$\text{increment}(I) = \text{increase the value of } I \text{ by one}$$

The function *getPrevious* returns the previous character of a character c .

$$\text{getPrevious}(c) = \text{return previous character of character } c$$

The function *getNext* returns the next character of a character c .

$$\text{getNext}(c) = \text{return next character of character } c$$

³In contrast to binary and shared locks.

The function *linkBefore* links a preceding character p with a succeeding character x . The field **Before** in table **CChar** of character p is set to character x . It returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{linkBefore}(p, x) &= \text{link character } p \text{ to character } x \\ \text{success} : \text{return } 1, \text{ no success} : \text{return } 0 \end{aligned}$$

The function *linkAfter* links a succeeding character n with a preceding character y . The field **After** in table **CChar** of character n is set to character y . It returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{linkAfter}(n, y) &= \text{link character } n \text{ to character } y \\ \text{success} : \text{return } 1, \text{ no success} : \text{return } 0 \end{aligned}$$

The function *updateString* links a character p with the first character c_1 of a character list l and a character n with the last character c_n of a character list l . The field **Before** in table **CChar** of character p is set to character c_1 and the field **After** in table **CChar** of character n is set to character c_n . It returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{updateString}(l, p, n) &= \\ &\text{linkBefore}(p, c_1) \wedge \text{linkAfter}(n, c_n) \\ \text{success} : \text{return } 1, \text{ no success} : \text{return } 0 \end{aligned}$$

The function *updateCharacter* links a character c with the characters p and n . The field **After** in table **CChar** of character c is set to character p and the field **Before** in table **CChar** of character c is set to character n . It returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{updateCharacter}(c, p, n) &= \\ &\text{linkAfter}(c, p) \wedge \text{linkBefore}(c, n), \text{ success} : \text{return } 1, \text{ no success} : \text{return } 0 \end{aligned}$$

The function *insertChar* inserts a character c in the table **CChar** with the fields **After** set to a character p and **Before** set to a character n . Furthermore, it links the character p and the character n with the character c in such a way that the field **Before** in table **CChar** of character p and the field **After** in table **CChar** of character n are set to character c .

$$\begin{aligned} \text{insertChar}(c, p, n) &= \\ &\text{linkAfter}(c, p) \wedge \text{linkBefore}(c, n) \wedge \text{linkBefore}(p, c) \wedge \text{linkAfter}(n, c) \end{aligned}$$

The function *removeChar* marks a character c as inactive by setting the field **IsActive** in the table **CChar** to 0.

$$\text{removeChar}(c) = \text{mark character } c \text{ as inactive}$$

The function *insertInUndoHistory* inserts a character c into the table **CUndoHistory** with the fields **After** set to a character p , **Before** set to a character n , **UserId** set to the current user u , **ActionId** set to an integer $I_{actionId}$ (1: representing an insertion, 2: representing a removal) and **CharCounter** set to the current value of CharCounter $I_{charCounter}$. It returns 1 for a success and 0 for no success.

$$\begin{aligned} \text{insertInUndoHistory}(c, p, n, u, I_{actionId}, I_{charCounter}) = \\ \text{insert character } c \text{ into UndoHistory} \\ \text{success : return 1, no success : return 0} \end{aligned}$$

The function *emptyRedoHistory* deletes the RedoHistory for document d . All entries in table **CRedoHistory** that suffice the condition **DocumentId**= d are deleted.

$$\text{emptyRedoHistory}(d) = \text{empty RedoHistory for document } d$$

The function *checkPreceding* determines the CharacterValue of the previous character of a character c and if the previous character's status is active.

$$\begin{aligned} \text{checkPreceding}(c) = \\ \text{return status and CharacterValue of the previous character} \end{aligned}$$

The function *checkSucceeding* determines the CharacterValue of the next character of a character c and if the next character's status is active.

$$\text{checkSucceeding}(c) = \text{return status and CharacterValue of the next character}$$

The function *checkCharValue* determines the CharacterValue of a character c .

$$\text{checkCharValue}(c) = \text{return CharacterValue of character } c$$

The function *informInsertOthers* informs all clients except for the one that has triggered the operation about the insertion of a character list l in the session s . This is done by calling the appropriate service routine with the appropriate parameters.

$$\text{informInsertOthers}(l, s) = \text{inform other clients about insertion of } l \text{ in } s$$

The function *informRemoveOthers* informs all clients except for the one that has triggered the operation about the removal of a character list l in the session s . This is done by calling the appropriate service routine with the appropriate parameters.

$$\textit{informRemoveOthers}(l) = \textit{inform other clients about removal of } l \textit{ in } s$$

The function *throwException* throws an exception e .

$$\textit{throwException}(e) = \textit{throw exception } e$$

The function *quit* quits an algorithm and returns an integer or the OID of a character as an integer I to where it was invoked from.

$$\textit{quit}(I) = \textit{quit algorithm returning } I$$

6.2 Insert Algorithm

The insert algorithm is invoked either from a 'normal' insert action of a client, what means a user has inserted one or more characters, or it is invoked from an undo or redo algorithm if a user has triggered the undo or redo function in a client.

First, it is explained how the insert algorithm works if it is invoked from a normal insert action. Later on, when the undo and redo algorithms are described, the invocation from the undo/redo algorithm will be discussed in detail.

In TeNDaX, the insert algorithm is implemented in the class method **InsertCChars** of the class **CChar** that is invoked with several parameters. Therefore, we assume that at the beginning of the algorithm specific information is available from those parameters.

The parameters that are relevant are introduced in the following listing and will be used for the definitions beneath:

- **nextCharacterId** (OID): reference character situated next to the string to be inserted
- **previousCharacterId** (OID): reference character situated previously to the string to be inserted
- **characterIds** (List): List of characters, which have to be inserted
- **flag** (Integer): indicates whether the insert action comes from an undo function: 1, a redo function: 2, or from a client insertion: 0

- noUndo (Integer): 0 normal case (fill in undo history), 1 for import (do not fill undo history)

Furthermore, there is the *Global*⁴ UndoCounter that shall be accessible by the variable charCounter and a flag emptyRedoHistoryAfterEdit that is either set to 1 if the RedoHistory shall be deleted or to 0 if the RedoHistory shall not be deleted after an insertion.

The appropriate flowchart for this algorithm is given in appendix A.3 on page 183.

Thus, the insertion of characters can be defined stepwise as follows:

Start a transaction.

startTA

Check the write access on the document session *s* that has the document session OID *s*.

checkWriteAccess(s) =
WriteAccess(documentSessionWrite)
 $\leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession))$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

If write access is granted, continue, else throw a SecurityException *e*.

If documentSessionWrite $\neq 1$ throwException(e)

Lock the character that follows the string, which shall be inserted and continue if successful, else throw a LockException *e*.

If lock(nextCharacterId) $\neq 1$ throwException(e)

Select the character that is situated before the character that follows the string, which shall be inserted.

getPrevious(nextCharacterId) =
 $PrevChar(prevCharId) \leftarrow \pi_{After}(\sigma_{ID=nextCharacterId}(CChar))$

⁴A *Global* is a named multi-dimensional array that is stored within a physical Caché database [InterSystems, 2004].

SQL-Statement:

```

select After into :prevCharId
from cache.CChar
where ID = :nextCharacterId

```

Continue, if the selection was successful, else throw a `GeneralException` e .

If $prevCharId = \emptyset$ *throwException*(e)

Acquire the lock for the character that is situated before the character that follows the string, which shall be inserted and continue if successful, else throw a `LockException` e .

If $lock(prevCharId) \neq 1$ *throwException*(e)

Check if the characters that are located before and after the string, which has to be inserted are consecutive and continue if successful, else throw a `GeneralException` e .

If $prevCharId \neq previousCharacterId$ *throwException*(e)

Increment the `CharCounter`.

increment($charCounter$)

At that time, the list *characterIds* contains the characters c_1 to c_n that shall be inserted.

$characterIds = \{c_1, \dots, c_n\}$

The characters that are located directly before and after the position where the string shall be inserted are already locked whereupon the insertion of each character can take place.

Each character of the string is inserted at the appropriate position by linking the preceding and the succeeding character to it. Moreover, each character is filled into the `UndoHistory` as an insert action if the insertion does not come from an import.

For each character c_i *of* *characterIds* :

insertChar(c_i, p, n)

If $noUndo \neq 1$:

$status = insertInUndoHistory(c_i, p, n, u, 1, charCounter)$

If $status \neq 1$ *throwException*(e)

Whereas $c_i \in \{c_1, \dots, c_n\}$

SQL-Statement:

```
insert into cache.CChar
  (CListId,CharacterValue,After,Before
   UserId,isActive,FileId)
values (:newCharacterId,:newCharacterValue,:newPrevCharacterId,
       :newNextCharacterId,:userId,1,:fileId)
```

SQL-Statement:

```
insert into cache.CUndoHistory
  (CharId,After,Before,DocumentId,UserId,ActionId,
   TimeStamps,IsActive,CharCounter,HorologStamp,Offset)
values (:newCharacterId,:newPrevCharacterId,
       :newNextCharacterId,:fileId,:userId,1,
       :stamp, 1,:undoCounter,:horologStamp,:where)
```

Empty the RedoHistory for document d if it shall be emptied after an insertion.

If emptyRedoHistoryAfterEdit = 1 emptyRedoHistory(d)

SQL-Statement:

```
delete from cache.CRedoHistory
where DocumentId = :documentId
```

Check if the preceding character is active or if it is the beginning of the document and if so continue, else throw a GeneralException e .

checkPreceding(prevCharId) =
IsOK(IsActive,CharacterValue)
 $\leftarrow \pi_{IsActive,CharacterValue}(\sigma_{ID=prevCharId}(CChar))$
If IsActive $\neq 1 \wedge CharacterValue \neq \beta$ throwException(e)

Check if the succeeding character is active or if it is the end of the document and if so continue, else throw a GeneralException e .

checkSucceeding(nextCharacterId) =
IsOK(IsActive,CharacterValue)
 $\leftarrow \pi_{IsActive,CharacterValue}(\sigma_{ID=nextCharacterId}(CChar))$
If IsActive $\neq 1 \wedge CharacterValue \neq \varepsilon$ throwException(e)

Update characters before and after the string that shall be inserted.

updateString(characterIds, prevCharId, nextCharacterId)

SQL-Statement:

```
update cache.CChar set Before = :startId
where ID = :prevCharId
and (IsActive=1 or CharacterValue=99998)
```

SQL-Statement:

```
update cache.CChar set After = :endId
where ID = :nextCharacterId
and (IsActive=1 or CharacterValue=99999)
```

If there exists an exception, release all locks, rollback the transaction and quit with return value 0, else continue.

If $\exists e$:

releaseLocks

rollbackTA

quit(0)

Release all locks.

releaseLocks

Commit transaction.

commitTA

Inform other clients about the insertion of the entire string.

informInsertOthers(characterIds, s)

Quit algorithm and return the OID of the first character that was inserted.

quit(c₁)

6.3 Remove Algorithm

Similar to the insert algorithm, the remove algorithm is invoked either from a 'normal' remove action of a client, what means a user has removed one or more characters, or it is invoked from an undo or redo algorithm if a user has triggered the undo or redo function in a client.

First, it is explained how the remove algorithm works if it is invoked from a normal remove action. Later on, when the undo and redo algorithms are described, the invocation from the undo/redo algorithm will be discussed in detail.

In TeNDaX, the remove algorithm is implemented in the class method `RemoveCChars` of the class `CChar` that is invoked with several parameters. Therefore, we assume that at the beginning of the algorithm specific information is available from those parameters.

The parameters that are relevant are introduced in the following listing and will be used for the definitions beneath:

- `characterIds` (List): list of characters, which have to be removed
- `previousCharacterId` (OID): reference character situated previously to the string to be removed
- `flag` (Integer): indicates whether the remove operation is called from an undo function: 1, from a redo function: 2, or from a client removal: 0

Furthermore, there is the *Global* `UndoCounter` that shall be accessible by the variable `charCounter` and a flag `emptyRedoHistoryAfterEdit` that is either set to 1 if the `RedoHistory` shall be deleted or to 0 if the `RedoHistory` shall not be deleted after a removal.

The appropriate flowchart for this algorithm is given in appendix A.8 on page 187.

Thus, the removal of characters can be defined stepwise as follows:

Start a transaction.

startTA

Check the write access on the document session *s* that has the document session OID *s*.

$$\begin{aligned} checkWriteAccess(s) = \\ & WriteAccess(documentSessionWrite) \\ & \leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession)) \end{aligned}$$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

Lock the first character c_1 of the list *characterIds* and continue if successful, else throw a *LockException* e .

If $lock(c_1) \neq 1$ *throwException*(e)

Select the character that is situated before the first character c_1 .

getPrevious(c_1) =

$PrevChar(prevCharId) \leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar))$

SQL-Statement:

```
select After into :prevCharId
from cache.CChar
where ID = :startCharId
```

Continue, if the selection was successful, else throw a *GeneralException* e .

If $prevCharId = \emptyset$ *throwException*(e)

Acquire the lock for the character that is situated before the first character that shall be removed and continue if successful, else throw a *LockException* e .

If $lock(prevCharId) \neq 1$ *throwException*(e)

Select the character that is situated after the last character c_n that shall be removed.

getNext(c_n) =

$NextChar(nextCharId) \leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

SQL-Statement:

```
select Before into :nextCharId
from cache.CChar
where ID = :endCharId
```

Continue, if the selection was successful, else throw a *GeneralException* e .

If $nextCharId = \emptyset$ *throwException*(e)

Acquire the lock for the character that is situated after the last character that shall be removed and continue if successful, else throw a `LockException` e .

If $lock(nextCharId) \neq 1$ *throwException*(e)

Increment the `CharCounter`.

increment(*charCounter*)

At that time, the list *characterIds* contains the characters c_1 to c_n that shall be removed.

$characterIds = \{c_1, \dots, c_n\}$

The characters that are located directly before the first character, the first character itself and the one located directly after the last character of the character list are already locked whereupon the removal of each character can take place.

Each character of the string is removed by setting its status to inactive (`IsActive=0`) and each character is filled into the `UndoHistory`.

For each character c_i *of* *characterIds* :

Select the character before the actual character that shall be removed

getPrevious(c_i) =

$PrevChar(after) \leftarrow \pi_{After}(\sigma_{ID=c_i}(CChar))$

Select the character after the actual character that shall be removed

getNext(c_i) =

$NextChar(before) \leftarrow \pi_{Before}(\sigma_{ID=c_i}(CChar))$

SQL-Statement:

```
select After, Before, ElementBorder
into :after, :before, :elementBorder
from cache.CChar
where ID = :id
```

If the selection was not successful throw a `GeneralException` e

If $after = \emptyset \vee before = \emptyset$ *throwException*(e)

Continue, if the actual character is consecutive to the preceding one, else throw a `GeneralException` e

If $after \neq previousCharacterId$ *throwException*(e)

If write access is granted, continue, else throw a `GeneralException` e

If documentSessionWrite $\neq 1$ throwException(e)

Remove the actual character c_i if it is neither the beginning nor the end of the document, else throw a GeneralException e

checkCharValue(c_i) =

GetValue(charValue) $\leftarrow \pi_{CharacterValue}(\sigma_{ID=c_i}(CChar))$

If charValue = $\beta \vee$ charValue = ε throwException(e)

Else removeChar(c_i)

SQL-Statement:

```
update cache.CChar set IsActive = 0
where ID = :id
and CharacterValue <> 99998 and CharacterValue <> 99999
```

Insert character into the UndoHistory as a remove action and continue if successful, else throw a GeneralException e

status = insertInUndoHistory(c_i , after, before, u, 2, charCounter)

If status $\neq 1$ throwException(e)

SQL-Statement:

```
insert into cache.CUndoHistory
(CharId, After, Before, DocumentId, UserId, ActionId, TimeStamps,
IsActive, CharCounter, HorologStamp, Offset)
values (:id, :after, :before, :fileId, :userId, 2, :stamp, 1,
:undoCounter, :horologStamp, :where)
```

Set previousCharacterId to the actual character c_i

previousCharacterId = c_i

Whereas $c_i \in \{c_1, \dots, c_n\}$

Afterwards, it applies:

$\forall c_i \in characterIds \Rightarrow IsActive = 0$

Empty the RedoHistory for document d if it shall be emptied after a removal.

If emptyRedoHistoryAfterEdit = 1 emptyRedoHistory(d)

SQL-Statement:

```
delete from cache.CRedoHistory
where DocumentId = :documentId
```

Check if the preceding character is active or if it is the beginning of the document and if so continue, else throw a `GeneralException` e .

```

checkPreceding(prevCharId) =
    IsOK(IsActive, CharacterValue)
     $\leftarrow \pi_{IsActive, CharacterValue}(\sigma_{ID=prevCharId}(CChar))$ 
    If IsActive  $\neq 1 \wedge$  CharacterValue  $\neq \beta$  throwException( $e$ )

```

Check if the succeeding character is active or if it is the end of the document and if so continue, else throw a `GeneralException` e .

```

checkSucceeding(nextCharId) =
    IsOK(IsActive, CharacterValue)
     $\leftarrow \pi_{IsActive, CharacterValue}(\sigma_{ID=nextCharId}(CChar))$ 
    If IsActive  $\neq 1 \wedge$  CharacterValue  $\neq \varepsilon$  throwException( $e$ )

```

Link the preceding and the succeeding characters to each other and continue if successful.

```

status1 = linkBefore(prevCharId, nextCharId)
status2 = linkAfter(nextCharId, prevCharId)
If status1  $\neq 1 \vee$  status2  $\neq 1$  throwException( $e$ )

```

SQL-Statement:

```

update cache.CChar set Before = :nextCharId
where ID = :prevCharId
and (IsActive=1 or CharacterValue=999998)

```

SQL-Statement:

```

update cache.CChar set After = :prevCharId
where ID = :nextCharId
and (IsActive=1 or CharacterValue=999999)

```

If there exists an exception, release all locks, rollback the transaction and quit with return value 2 (no write access) or 0 (any other exception), else continue.

```

If  $\exists e$  :
    releaseLocks
    rollbackTA
    If documentSessionWrite  $\neq 1$  quit(2)
    Else quit(0)

```

Release all locks.

releaseLocks

Commit transaction.

commitTA

Inform other clients about the removal of the entire string.

informRemoveOthers(characterIds, s)

Quit algorithm and return 1.

quit(1)

6.4 Undo Algorithm

The undo algorithm is invoked if a user has triggered the undo function in a client. There is either the possibility to trigger a local or a global undo. This distinction is passed from the client to the database class method **CUndo** of the class **CChar**, which is invoked with several parameters.

The parameter that is relevant is 'isGlobal' (Boolean), which is a flag that indicates if the operation is a global or a local undo (isGlobal=1: global, isGlobal = 0: local). With this parameter, the undo algorithm can distinguish if a local or global undo shall be performed. The control flow of the undo algorithm does not differ that much, regardless of which undo is performed, but the consequences for the conflict resolution are enormous. The only difference between the global and local undo is, the determination of the operation to be undone. Therefore, we divide the formal depiction in three steps. The first step (section 6.4.1) and the third step (section 6.4.4) apply to both the local and global undo. In the second step, which describes how the operation that shall be undone is determined, it is distinguished between the global (section 6.4.2) and the local (section 6.4.3) undo.

From the class method **CUndo** either the class method **InsertCChars** or **RemoveCChars** of the class **CChar** is invoked, depending on the kind of operation that has to be undone.

Thus, we will first explain the control sequence in the class method **CUndo** by abstracting the nested invocation of the class methods **InsertCChars/RemoveCChars** to discuss the control sequence in those methods in detail afterwards.

The appropriate flowchart for this algorithm is given in appendix A.1 on page 181.

To describe the undo algorithm some definitions have to be added:

The function *getGlobalUndoHistoryEntry* returns the last operation of any user that is logged in the UndoHistory for a document *d* and that is relevant for undo.

getGlobalUndoHistoryEntry(d) =
get the last operation of any user that is logged in CUndoHistory for document d
and that is relevant for undo

The function *getLocalUndoHistoryEntry* returns the last operation of the actual user *u* that is logged in the UndoHistory for a document *d* and that is relevant for undo.

getLocalUndoHistoryEntry(d) =
get the last operation of user u that is logged in CUndoHistory for document d
and that is relevant for undo

The function *buildList* creates a list *l* of characters, which have a parameter *z* in common.

buildList(z) = create a list of characters depending on parameter z

The function *active* returns *true* if a character *c* is active and *false* if not.

active(c) = true if character c is active, else false

The function *callMethodInsert* invokes the class method *InsertCChars* with the appropriate parameters and returns an integer or the OID of a character as an integer *I*.

callMethodInsert = invoke InsertCChars and return I

The function *callMethodRemove* invokes the class method *RemoveCChars* with the appropriate parameters and returns an integer *I*.

callMethodRemove = invoke RemoveCChars and return I

The function *setRedoHistoryNotEmpty* sets the RedoHistory status for a document *d* to 'notEmpty'. A *Global* that indicates the status of the RedoHistory is set to 1.

setRedoHistoryNotEmpty(d) = set RedoHistory to notEmpty for d

The function *markNotRelevantForUndo* marks a character c as not relevant for the undo. The field *IsActive* in the table *CUndoHistory* is set to 0. If the update was successful, it returns 1, else 0.

markNotRelevantForUndo(c) = mark c as not relevant for undo
success : return 1, *no success* : return 0

6.4.1 First Step - Global and Local Undo

Thus, the undo of an operation can be defined stepwise as follows: Start a transaction.

startTA

Check the write access on the document session s that has the document session OID s .

checkWriteAccess(s) =
WriteAccess(*documentSessionWrite*)
 $\leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession))$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

If write access is granted, continue, else throw a *NoWriteAccessException* e .

If documentSessionWrite $\neq 1$ *throwException*(e)

6.4.2 Second Step - Only Global Undo

It applies: *isGlobal* = 1

Select the last edited character that is logged for the actual document d and relevant for undo (field *IsActive* in table *CUndoHistory* has value 1).

$A \leftarrow \sigma_{DocumentId=d \text{ and } IsActive=1}(CUndoHistory)$
 $B \leftarrow ActionId, CharId, CharCounter F_{max ID}(A)$
getGlobalUndoHistoryEntry(d) =
 $GHE(action, char, charCounter) \leftarrow \pi_{ActionId, CharId, CharCounter}(B)$

SQL-Statement:

```

select ID, ActionId, CharId, CharCounter, Offset
      into :Id, :actionId, :charId, :charCounter, :offset
from cache.CUndoHistory
where ID =
      (select MAX(ID)
       from cache.CUndoHistory
       where DocumentId = :docId and IsActive = 1)

```

Continue, if the selection was successful, else throw a NoPossibleUndoInHistory exception e .

If char = $\emptyset$ throwException(e)

6.4.3 Second Step - Only Local Undo

It applies: $isGlobal = 0$

Select the last edited character that is logged for the actual document d and relevant for undo (field `IsActive` in table `CUndoHistory` has value 1). In addition to the global undo only the entries of the actual user u are queried.

$$\begin{aligned}
 A &\leftarrow \sigma_{DocumentId=d \text{ and } IsActive=1 \text{ and } UserId=u}(CUndoHistory) \\
 B &\leftarrow ActionId, CharId, CharCounter F_{max ID}(A) \\
 getLocalUndoHistoryEntry(d) &= \\
 LHE(action, char, charCounter) &\leftarrow \pi_{ActionId, CharId, CharCounter}(B)
 \end{aligned}$$

SQL-Statement:

```

select ID, ActionId, CharId, CharCounter, Offset
      into :Id, :actionId, :charId, :charCounter, :offset
from cache.CUndoHistory
where ID =
      (select MAX(ID)
       from cache.CUndoHistory
       where DocumentId = :docId
          and UserId = :userId and IsActive = 1)

```

Continue, if the selection was successful, else throw a NoPossibleUndoInHistory exception e .

If char = $\emptyset$ throwException(e)

At this point a minor part of the conflict resolution is realized because if it is an insert action that has to be undone, it could be that some characters are already deleted and the removal would make no sense. Thus, in case of an insert action, it has to be checked if the characters are not already deleted. If so, the previous operation in the UndoHistory is selected and checked in turn. This is repeated until an operation is found that either is a remove operation or an insert operation without characters that are already deleted.

If it is a remove action that has to be undone, continue with the third step, else check if the characters are already deleted and if needed search for another operation.

repeat

Check if there exist multiple simultaneous executed operations (copy-paste).
Simultaneously edited characters have the same CharCounter.

buildList(charCounter) =

List(char) $\leftarrow \pi_{CharId}(\sigma_{CharCounter=charCounter}(LHE))$

SQL-Statement:

```

DECLARE RemoveCursor CURSOR FOR
    select CharId
        into :charIds
    from cache.CUndoHistory
    where CharCounter = :charCounter
OPEN RemoveCursor
    For {
        FETCH RemoveCursor
        Quit:SQLCODE
        Set list = list_$ListBuild(charIds)
    }
CLOSE RemoveCursor

```

Thus, the following list is generated that contains all simultaneously edited characters:

$$l = \{char_1, \dots, char_n\}$$

and it applies:

$$NumOfChar \leftarrow F_{count\ char}(List)$$

if NumOfChar = 1 then n = 1 else n = NumOfChar

Check each character of the list l if it is active.

For each character c_i in l :

$active(c_i) = CheckActive(active) \leftarrow \pi_{IsActive}(\sigma_{ID=c_i}(CChar))$

If active $\neq 0$ break

SQL-Statement:

```
select isActive into :isActive
from cache.CChar
where ID = :UId
```

If a character is found that is already deleted, the previous operation is selected.

If active = 0

$A \leftarrow \sigma_{DocumentId=d \text{ and } IsActive=1 \text{ and } UserId=u}$
 $\text{and } ID < c_i \text{ and } CharCounter \neq u(CUndoHistory)$

$B \leftarrow ActionId, CharId, CharCounter F_{max ID}(A)$

$getLocalUndoHistoryEntry(d) =$

$LHE(action, char, charCounter)$

$\leftarrow \pi_{ActionId, CharId, CharCounter}(B)$

SQL-Statement:

```
select ID, ActionId, CharId, CharCounter, Offset
into :Id, :actionId, :charId, :charCounter, :offset
from cache.CUndoHistory
where ID =
(select MAX(ID)
from cache.CUndoHistory
where DocumentId = :docId and UserId = :userId
and IsActive = 1 and ID < :Id
and CharCounter <> :charCounter)
```

Continue, if the selection was successful, else throw a NoPossibleUndoInHistory exception e .

If char = $\emptyset$ throwException(e)

until active = 1 $\vee$ action = 2

6.4.4 Third Step - Global and Local Undo

At this point, the last entry that is relevant for the undo and that has not already been deleted by another user is read in.

Check if there exist multiple simultaneous executed operations (copy-paste).

$$\begin{aligned} buildList(charCounter) = \\ List(char) \leftarrow \pi_{CharId}(\sigma_{CharCounter=charCounter}(LHE)) \end{aligned}$$

SQL-Statement:

```

DECLARE UndoCur CURSOR FOR
    select CharId
        into :charIds
    from cache.CUndoHistory
    where CharCounter = :charCounter
OPEN UndoCur
For {
    FETCH UndoCur
    Quit:SQLCODE
    Set list = list_$ListBuild(charIds)
}
CLOSE UndoCur

```

Thus, the following list is generated that contains all simultaneously edited characters:

$$l = \{char_1, \dots, char_n\}$$

and it applies:

$$\begin{aligned} NumOfChar &\leftarrow F_{count\ char}(List) \\ if\ NumOfChar = 1\ then\ n &= 1\ else\ n = NumOfChar \end{aligned}$$

Select the reference character that is situated before the first character c_1 of the list l .

$$\begin{aligned} getPrevious(c_1) = \\ PrevChar(after) &\leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar)) \end{aligned}$$

SQL-Statement:

```

select After into :after
from cache.CChar
where ID = :startCharId

```

Select the reference character that is situated after the last character c_n of the list l .

$getNext(c_n) =$
 $NextChar(before) \leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

SQL-Statement:

```

select Before into :before
from cache.CChar
where ID = :Sid

```

At this point the list l contains all characters that have to be changed, while the reference characters *after* and *before* hold the information of the position. It is now distinguished, which other class method has to be invoked depending on the kind of operation that shall be undone. If the *ActionId* of the selected operation is 1, it was an insertion and therefore now a removal has to be executed (trigger *RemoveCChars*). If otherwise the *ActionId* of the selected operation is 2, it was a removal and therefore now an insertion has to be executed (trigger *InsertCChars*).

Check what kind of operation has to be undone and invoke the appropriate method.

If it is an insert action that has to be undone, invoke *RemoveCChars* and set the *RedoHistory* to not empty.

Only continue, if the removal was successful, else throw a *GeneralException* e .

If action = 1

status = *callMethodRemove*

If status = 0 *throwException*(e)

setRedoHistoryNotEmpty(d)

If it is a remove action that has to be undone, invoke *InsertCChars*, set the *RedoHistory* to not empty and mark each character of list l as not relevant for undo anymore.

Only continue, if the insertion was successful and if all characters are marked correctly, else throw a *GeneralException* e .

If action = 2

status = *callMethodInsert*

If $status = 0$ *throwException*(e)
setRedoHistoryNotEmpty(d)
For each character c_i *in* l :
 $status = markNotRelevantForUndo(c_i)$
If $status = 0$ *throwException*(e)

SQL-Statement:

```

update cache.CUndoHistory
set IsActive = 0
where CharId = :Pid and ActionId = 2
  
```

If there exists an exception, rollback the transaction and quit with return value -1 (no write access) or 0 (any other exception), else continue.

If $\exists e$:
 $rollbackTA$
 If $e = NoWriteAccessException$ $quit(-1)$
 Else $quit(0)$

Commit transaction.

$commitTA$

Quit algorithm and return 1.

$quit(1)$

6.5 Redo Algorithm

The redo algorithm is invoked if a user has triggered the redo function in a client. Similar to the undo algorithm there is either the possibility to trigger a local or a global redo. This distinction is passed from the client to the database class method **CRedo** of the class **CChar** that is invoked with several parameters.

The parameter that is relevant is 'isGlobal' (Boolean), which is a flag that indicates if the operation is a global or a local redo (isGlobal=1: global, isGlobal = 0: local). With this parameter, the redo algorithm can distinguish if a local or global redo shall be performed. The control flow of the redo algorithm does not differ that much, regardless of which redo is performed, but the consequences for the conflict resolution are enormous. The only difference between the global and local redo is,

the determination of the operation to be redone. Therefore, we divide the formal depiction in three steps. The first step (section 6.5.1) and the third step (section 6.5.4) apply to both the local and global redo. In the second step, which describes how the operation that shall be redone is determined, it is distinguished between the global (section 6.5.2) and the local (section 6.5.3) redo.

From the class method **CRedo** either the class method **InsertCChars** or **RemoveCChars** of the class **CChar** is invoked, depending on the kind of operation has to be redone. Note that for the redo algorithm these invocations are contrarily to the ones for the undo algorithm.

We will first explain the control sequence in the class method **CRedo** by abstracting the nested invocation of the class methods **InsertCChars/RemoveCChars** to discuss the control sequence in those methods in detail afterwards.

The appropriate flowchart for this algorithm is given in appendix A.2 on page 182.

To describe the redo algorithm some definitions have to be added:

The function *getGlobalRedoHistoryEntry* returns the last operation of any user that is logged in the RedoHistory for a document *d*.

getGlobalRedoHistoryEntry(d) =
get the last operation of any user that is logged in CRedoHistory for document d

The function *getLocalRedoHistoryEntry* returns the last operation of the actual user *u* that is logged in the RedoHistory for a document *d*.

getLocalRedoHistoryEntry(d) =
get the last operation of user u that is logged in CRedoHistory for document d

The function *markRelevantForUndo* marks a character *c* as relevant for the undo. The field *IsActive* in the table *CUndoHistory* is set to 1. If the update was successful, it returns 1, else 0.

markRelevantForUndo(c) =
mark c as relevant for undo and return 1 for success or 0 for no success
success : return 1, no success : return 0

6.5.1 First Step - Global and Local Redo

Thus, the redo of an operation can be defined stepwise as follows: Start a transaction.

startTA

Check the write access on the document session *s* that has the document session OID *s*.

checkWriteAccess(s) =
 WriteAccess(documentSessionWrite)
 $\leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession))$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

If write access is granted, continue, else throw a *NoWriteAccessException* *e*.

If documentSessionWrite $\neq 1$ *throwException(e)*

6.5.2 Second Step - Only Global Redo

It applies: *isGlobal* = 1

Different to undo algorithm:

Select the last entry in table *CRedoHistory* that is logged for the actual document *d*.

$A \leftarrow \sigma_{DocumentId=d}(CRedoHistory)$
 $B \leftarrow \pi_{ActionId, CharId, CharCounter} F_{max ID}(A)$
getGlobalRedoHistoryEntry(d) =
 $GHE(action, char, charCounter) \leftarrow \pi_{ActionId, CharId, CharCounter}(B)$

SQL-Statement:

```
select ID, ActionId, CharId, CharCounter, Offset
      into :Id, :actionId, :charId, :charCounter, :offset
from cache.CRedoHistory
where ID =
      (select MAX(ID)
       from cache.CRedoHistory
       where DocumentId = :docId)
```

Different to undo algorithm:

Continue, if the selection was successful, else throw a `NoPossibleRedoInHistory` exception e .

If $char = \emptyset$ throwException(e)

6.5.3 Second Step - Only Local Redo

It applies: $isGlobal = 0$

Different to undo algorithm:

Select the last entry in table `CRedoHistory` that is logged for the actual document d . In addition to the global redo only the entries of the actual user u are queried.

$$\begin{aligned}
 A &\leftarrow \sigma_{DocumentId=d \text{ and } UserId=u}(CRedoHistory) \\
 B &\leftarrow \pi_{ActionId, CharId, CharCounter F_{max} ID}(A) \\
 getLocalRedoHistoryEntry(d) &= \\
 LHE(action, char, charCounter) &\leftarrow \pi_{ActionId, CharId, CharCounter}(B)
 \end{aligned}$$

SQL-Statement:

```

select ID, ActionId, CharId, CharCounter, Offset
      into :Id, :actionId, :charId, :charCounter, :offset
from cache.CRedoHistory
where ID =
      (select MAX(ID)
       from cache.CRedoHistory
       where DocumentId = :docId and UserId = :userId)

```

Different to undo algorithm:

Continue, if the selection was successful, else throw a `NoPossibleRedoInHistory` exception e .

If $char = \emptyset$ throwException(e)

At this point a minor part of the conflict resolution is realized because if it is a remove action that has to be redone, it could be that some characters are already deleted and the removal would make no sense. Thus, in case of a remove action, it has to be checked if the characters are not already deleted. If they are, the previous operation in the RedoHistory is selected and checked in turn. This is repeated until an operation is found that either is a remove operation or an insert operation

without characters that are already deleted.

If it is an insert action that has to be redone, continue with the third step, else check if the characters are already deleted and if needed search for another operation.

repeat

Check if there exist multiple simultaneous executed operations (copy-paste).

Simultaneously edited characters have the same CharCounter.

$buildList(charCounter) =$

$List(char) \leftarrow \pi_{CharId}(\sigma_{CharCounter=charCounter}(LHE))$

SQL-Statement:

```

DECLARE RemoveCursor CURSOR FOR
    select CharId
        into :charIds
    from cache.CRedoHistory
    where CharCounter = :charCounter
OPEN RemoveCursor
    For {
        FETCH RemoveCursor
        Quit:SQLCODE
        Set list = list_$(ListBuild(charIds)
    }
CLOSE RemoveCursor

```

Thus, the following list is generated that contains all simultaneously edited characters:

$l = \{char_1, \dots, char_n\}$

and it applies:

$NumOfChar \leftarrow F_{count\ char}(List)$

if $NumOfChar = 1$ *then* $n = 1$ *else* $n = NumOfChar$

Check each character of the list l if it is active.

For each character c_i *in* l :

$active(c_i) = CheckActive(active) \leftarrow \pi_{IsActive}(\sigma_{ID=c_i}(CChar))$

If $active \neq 0$ *break*

SQL-Statement:

```
select isActive into :isActive
from cache.CChar
where ID = :UId
```

Different to undo algorithm:

If a character is found that is already deleted, the previous operation is selected.

If active = 0

$A \leftarrow \sigma_{DocumentId=d \text{ and } UserId=u}$
 $\text{and } ID < c_i \text{ and } CharCounter \neq u(CRedoHistory)$

$B \leftarrow ActionId, CharId, CharCounter F_{max ID}(A)$

$getLocalRedoHistoryEntry(d) =$

$LHE(action, char, charCounter)$

$\leftarrow \pi_{ActionId, CharId, CharCounter}(B)$

SQL-Statement:

```
select ID, ActionId, CharId, CharCounter, Offset
into :Id, :actionId, :charId, :charCounter, :offset
from cache.CRedoHistory
where ID =
(select MAX(ID)
from cache.CRedoHistory
where DocumentId = :docId and UserId = :userId
and ID < :Id and CharCounter <> :charCounter)
```

Different to undo algorithm:

Continue, if the selection was successful, else throw a NoPossibleRedoInHistory exception e .

If char = $\emptyset$ throwException(e)

until active = 1 $\vee$ action = 2

6.5.4 Third Step - Global and Local Redo

At this point, the last entry that has not already been deleted by another user is read in.

Check if there exist multiple simultaneous executed operations (copy-paste).

$$\begin{aligned} buildList(charCounter) = \\ List(char) \leftarrow \pi_{CharId}(\sigma_{CharCounter=charCounter}(LHE)) \end{aligned}$$

SQL-Statement:

```

DECLARE RedoCur CURSOR FOR
    select CharId
        into :charIds
    from cache.CRedoHistory
    where CharCounter = :charCounter
OPEN RedoCur
For {
    FETCH RedoCur
    Quit:SQLCODE
    Set list = list_$ListBuild(charIds)
}
CLOSE RedoCur

```

Thus, the following list is generated that contains all simultaneously edited characters:

$$l = \{char_1, \dots, char_n\}$$

and it applies:

$$\begin{aligned} NumOfChar &\leftarrow F_{count\ char}(List) \\ if\ NumOfChar &= 1\ then\ n = 1\ else\ n = NumOfChar \end{aligned}$$

Select the reference character that is situated before the first character c_1 of the list l .

$$\begin{aligned} getPrevious(c_1) = \\ PrevChar(after) \leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar)) \end{aligned}$$

SQL-Statement:

```

select After into :after
from cache.CChar
where ID = :startCharId

```

Select the reference character that is situated after the last character c_n of the list l .

$getNext(c_n) =$
 $NextChar(before) \leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

SQL-Statement:

```
select Before into :before
from cache.CChar
where ID = :Sid
```

At this point the list l contains all characters that have to be changed, while the reference characters *after* and *before* hold the information of the position. It is now distinguished, which other class method has to be invoked depending on the kind of operation that shall be redone. If the *ActionId* of the selected operation is 1, it was an insertion and therefore now an insertion has to be executed (trigger *InsertCChars*). If otherwise the *ActionId* of the selected operation is 2, it was a removal and therefore now a removal has to be executed (trigger *RemoveCChars*).

Different to undo algorithm:

Check what kind of operation has to be redone and invoke the appropriate method.

If it is a remove action that has to be redone, invoke *RemoveCChars*.

Only continue, if the removal was successful, else throw a *GeneralException* e .

If action = 2

$status = callMethodRemove$

If status = 0 throwException(e)

If it is an insert action that has to be redone, invoke *InsertCChars* and mark each character of list l as not relevant again for undo.

Only continue, if the insertion was successful and if all characters are marked correctly, else throw a *GeneralException* e .

If action = 1

$status = callMethodInsert$

If status = 0 throwException(e)

For each character c_i in l :

$status = markRelevantForUndo(c_i)$

If status = 0 throwException(e)

SQL-Statement:

```
update cache.CUndoHistory
set IsActive = 1
where CharId = :Pid and ActionId = 1
```

If there exists an exception, rollback the transaction and quit with return value -1 (no write access) or 0 (any other exception), else continue.

```

If  $\exists e$  :
    rollbackTA
    If  $e = NoWriteAccessException$  quit(-1)
    Else quit(0)

```

Commit transaction.

```

commitTA

```

Quit algorithm and return 1.

```

quit(1)

```

6.6 Insert Algorithm invoked from Undo/Redo

Now, since it is explained how the undo and redo invokes an insertion or a removal, we come to the detailed depiction of the insertion and removal of characters invoked from the undo or redo algorithm, which actually builds the heart of the conflict resolution. In this subsection we present the formal depiction of an insertion from an undo or redo.

As it is described in the subsections 6.4 and 6.5 the insert algorithm, which is implemented in the class method **InsertCChars** of the class **CChar** is invoked from the class methods **CUndo** or **CRedo** of the class **CChar** if an undo or a redo respectively was triggered by a user. Those class methods invoke the insert algorithm with several parameters. Therefore, we assume that at the beginning of the algorithm specific information is available from those parameters. The important parameter that is used to determine that the invocation comes from an undo or redo is **flag**. This parameter is set to 1 if the invocation comes from an undo and it is set to 2 if it comes from a redo.

The parameters that are relevant are introduced in the following listing and will be used for the definitions beneath:

- **nextCharacterId (OID)**: reference character situated next to the string to be inserted
- **previousCharacterId (OID)**: reference character situated previously to the string to be inserted

- *characterIds* (List): List of characters, which have to be inserted
- *flag* (Integer): indicates whether the insert action comes from an undo function: 1, a redo function: 2, or from a client insertion: 0
- *noUndo* (Integer): 0 normal case (fill in undo history), 1 for import (do not fill undo history)

Furthermore, there are the Globals *UndoCounter* and *RedoCounter* that shall be accessible by the variable *undoCounter* and *redoCounter*. There are also a flag *emptyRedoHistoryAfterEdit* that is either set to 1 if the *RedoHistory* shall be deleted or to 0 if the *RedoHistory* shall not be deleted after an insertion, a flag *defaultInsertPolicy* that is either set to 0 if the algorithm shall search on the left side or to 1 if the algorithm shall search on the right side (cf. DIP in chapter 5, section 5.3.2.1) and a flag *simpleInsertConflictResolution* that determines the mode for the conflict resolution (0: enhanced, 1: simple conflict resolution). The notions of FORC and SORC are used for the depiction of the insert conflict resolution (cf. sections 5.3.1.1 and 5.3.1.2).

We divide the formal depiction in four steps. In the first step (section 6.6.1) it is described how the information is gathered that is used to resolve a conflict. In the second step (section 6.6.2) it is particularly described how a conflict is identified while in the third step (section 6.6.3) it is explained how a certain conflict is treated. In the forth step (section 6.6.4) the insert algorithm is completed.

The appropriate flowchart for this algorithm is given in appendix A.4 on page 184.

To describe the insert algorithm invoked from undo or redo, some definitions have to be added:

The function *reinsertChar* marks a character *c* as active. The field *IsActive* in the table *CChar* is set to 1. If the update was successful, it returns 1, else 0.

$$\begin{aligned} & \textit{reinsertChar}(c) = \textit{mark } c \textit{ as active} \\ & \textit{success} : \textit{return } 1, \textit{ no success} : \textit{return } 0 \end{aligned}$$

The function *checkConsecutive* compares the OIDs of two characters *c1* and *c2*. If the OID of *c1* and *c2* are consecutive, it returns 1, else 0.

$$\begin{aligned} & \textit{checkConsecutive}(c1, c2) = \\ & \textit{return } 1 \textit{ if the OID of } c1 \textit{ is consecutive with the OID of } c2 \textit{ or } 0 \textit{ if not} \end{aligned}$$

The function *insertInRedoHistory* inserts a character c into the table **CRedoHistory** with the fields **After** set to a character p , **Before** set to a character n , **UserId** set to the current user u , **ActionId** set to an integer $I_{actionId}$ (1: representing an insertion, 2: representing a removal) and **CharCounter** set to the current value of CharCounter $I_{charCounter}$. It returns 1 for a success and 0 for no success.

$$\begin{aligned} &insertInRedoHistory(c, p, n, u, I_{actionId}, I_{charCounter}) = \\ &\quad insert\ character\ c\ into\ RedoHistory \\ &\quad success : return\ 1, no\ success : return\ 0 \end{aligned}$$

The function *deleteFromRedoHistory* deletes a character c from the table **CRedoHistory**. It returns 1 for a success and 0 for no success.

$$\begin{aligned} &deleteFromRedoHistory(c) = delete\ character\ c\ from\ RedoHistory \\ &\quad success : return\ 1, no\ success : return\ 0 \end{aligned}$$

The function *getLeftRefCharAndStatus* determines for a character c the status and the succeeding character of the preceding character.

$$\begin{aligned} &getLeftRefCharAndStatus(c) = \\ &\quad return\ the\ status\ and\ succeeding\ character\ of\ the\ preceding\ character\ of\ c \end{aligned}$$

The function *getRightRefCharAndStatus* determines for a character c the status and the preceding character of the succeeding character.

$$\begin{aligned} &getRightRefCharAndStatus(c) = \\ &\quad return\ the\ status\ and\ preceding\ character\ of\ the\ succeeding\ character\ of\ c \end{aligned}$$

The function *getCharCounter* determines for a character c the attribute **CharCounter** in the table **CChar**.

$$getCharCounter(c) = return\ the\ CharCounter\ of\ c$$

The function *searchLeft* determines for a character c its OID, its status and its preceding character.

$$searchLeft(c) = return\ the\ OID, the\ preceding\ character\ and\ the\ status\ of\ c$$

The function *searchRight* determines for a character c its OID, its status and its succeeding character.

$$searchRight(c) = return\ the\ OID, the\ succeeding\ character\ and\ the\ status\ of\ c$$

The function *informInsertOwnAndOthers* informs all clients about the insertion of a character list l in session s . This is done by calling the appropriate service routine with the appropriate parameters.

informInsertOwnAndOthers(l, s) = *inform all clients about insertion of l in s*

The function *informInsertOwnAndOthersSingleChar* informs all clients about the insertion of a character list l in session s . Each character is submitted individually unlike to submitting the entire list. This is done by calling the appropriate service routine with the appropriate parameters.

informInsertOwnAndOthers(l, s) =
inform all clients about insertion of l in s by submitting the single characters

6.6.1 First Step - Initiation

Thus, the insertion of characters triggered by an undo or redo can be defined stepwise as follows:

Start a transaction.

startTA

Check the write access on the document session s that has the document session OID s .

checkWriteAccess(s) =
WriteAccess(*documentSessionWrite*)
 $\leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession))$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

If write access is granted, continue, else throw a *SecurityException* e .

If documentSessionWrite $\neq 1$ *throwException*(e)

Different to 'normal' invocation:

If the invocation comes from an undo, increment the redoCounter and if it comes from a redo, increment the undoCounter.

If flag = 1 increment(redoCounter)
If flag = 2 increment(undoCounter)

For each character of the insertion string:

If the character is not the first one, check if the character is consecutive with the preceding one and if not, annotate it in a flag *notConsecutiveInsertion*. Reinsert the character and if the invocation comes from an undo insert it into the RedoHistory else delete it from the RedoHistory and continue if successful, otherwise throw a GeneralException *e*.

For each character c_i of characterIds :

If $c_i \neq c_1$
 status = checkConsecutive(c_{i-1}, c_i)
 If status = 0
 notConsecutiveInsertion = i
 status = reinsertChar(c_i)
 If status $\neq 1$ throwException(e)
 If flag = 1
 Select the preceding character of the actual character that shall
 be inserted
 getPrevious(c_i) =
 PrevChar(after) $\leftarrow \pi_{After}(\sigma_{ID=c_i}(CChar))$
 Select the succeeding character of the actual character that shall
 be inserted
 getNext(c_i) =
 NextChar(before) $\leftarrow \pi_{Before}(\sigma_{ID=c_i}(CChar))$
 insertInRedoHistory(c_i)
 If flag = 2
 status = deleteFromRedoHistory(c_i)
 If status = 0 throwException(e)

Whereas $c_i \in \{c_1, \dots, c_n\}$

SQL-Statement:

```
update cache.CChar set IsActive = 1
where ID = :charId
```

From undo, SQL-Statement:

```
select After, Before into :after, :before
from cache.CChar
where ID = :charId
```

From undo, SQL-Statement:

```
insert into cache.CRedoHistory
(CharId, After, Before, DocumentId, UserId, ActionId,
 TimeStamps, CharCounter, Offset)
values (:charId, :after, :before, :fileId, :userFk, 1,
 :stamp, :redoCounter, :where)
```

From redo, SQL-Statement:

```
delete from cache.CRedoHistory
where CharId = :charId and ActionId = 1
```

Select the preceding character of the first character c_1 of the character list *characterIds* that shall be inserted.

$getPrevious(c_1) =$

$PrevChar(tmpAfter) \leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar))$

Select the succeeding character of the last character c_n of the character list *characterIds* that shall be inserted.

$getNext(c_n) =$

$NextChar(confBefore) \leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

At that time, *tmpAfter* represents the FORC-L and *confBefore* represents the FORC-R.

SQL-Statement:

```
select After into :tmpAfter
from cache.CChar
where ID = :Fid
```

SQL-Statement:

```
select Before into :confBefore
from cache.CChar
where ID = :Lid
```

Select the status of the preceding character and its succeeding character.

$$\begin{aligned} \text{getLeftRefCharAndStatus}(\text{tmpAfter}) = \\ \text{LeftRef}(\text{tmpBefore}, \text{isLeftActive}) \\ \leftarrow \pi_{\text{Before}, \text{IsActive}}(\sigma_{\text{ID}=\text{tmpAfter}}(\text{CChar})) \end{aligned}$$

SQL-Statement:

```
select Before, IsActive into :tmpBefore, :isLeftActive
from cache.CChar
where ID = :tmpAfter
```

Select the status of the succeeding character and its preceding character.

$$\begin{aligned} \text{getRightRefCharAndStatus}(\text{confBefore}) = \\ \text{RightRef}(\text{confAfter}, \text{isRightActive}) \\ \leftarrow \pi_{\text{Before}, \text{IsActive}}(\sigma_{\text{ID}=\text{confBefore}}(\text{CChar})) \end{aligned}$$

At that time, *tmpBefore* represents the SORC-L and *confAfter* represents the FORC-R.

SQL-Statement:

```
select After, IsActive into :confAfter, :isRightActive
from cache.CChar
where ID = :confBefore
```

Select the CharCounter of the FORC-L.

$$\begin{aligned} \text{getCharCounter}(\text{tmpAfter}) = \\ \text{CC}(\text{tmpAfterCharCounter}) \\ \leftarrow \pi_{\text{CharCounter}}(\text{CharCounter} F_{\min \text{ID}}(\sigma_{\text{CharId}=\text{tmpAfter}}(\text{CUndoHistory}))) \end{aligned}$$

SQL-Statement:

```
select CharCounter into :tmpAfterCharCounter
from cache.CUndoHistory
where ID = (select Min(ID)
            from cache.CUndoHistory
            where CharId =tmpAfter)
```

Select the CharCounter of the FORC-R.

$getCharCounter(confBefore) =$
 $CC(confBeforeCharCounter)$
 $\leftarrow \pi_{CharCounter}(CharCounter F_{min} ID(\sigma_{CharId=confBefore}(CUndoHistory)))$

SQL-Statement:

```

select CharCounter into :confBeforeCharCounter
from cache.CUndoHistory
where ID = (select Min(ID)
            from cache.CUndoHistory
            where CharId =confBefore)

```

Select the CharCounter of the FORC-L.

$getCharCounter(tmpBefore) =$
 $CC(tmpBeforeCharCounter)$
 $\leftarrow \pi_{CharCounter}(CharCounter F_{min} ID(\sigma_{CharId=tmpBefore}(CUndoHistory)))$

SQL-Statement:

```

select CharCounter into :tmpBeforeCharCounter
from cache.CUndoHistory
where ID = (select Min(ID)
            from cache.CUndoHistory
            where CharId =tmpBefore)

```

Select the CharCounter of the FORC-R.

$getCharCounter(confAfter) =$
 $CC(confAfterCharCounter)$
 $\leftarrow \pi_{CharCounter}(CharCounter F_{min} ID(\sigma_{CharId=confAfter}(CUndoHistory)))$

SQL-Statement:

```

select CharCounter into :confAfterCharCounter
from cache.CUndoHistory
where ID = (select Min(ID)
            from cache.CUndoHistory
            where CharId =confAfter)

```

Select the CharCounter of the first character that shall be inserted.

$getCharCounter(c_1) =$
 $CC(fidCharCounter)$
 $\leftarrow \pi_{CharCounter}(CharCounter F_{min ID}(\sigma_{CharId=c_1}(CUndoHistory)))$

SQL-Statement:

```

select CharCounter into :fidCharCounter
from cache.CUndoHistory
where ID = (select Min(ID)
            from cache.CUndoHistory
            where CharId =Fid)

```

Select the CharacterValue of the FORC-L.

$checkCharValue(tmpAfter) =$
 $GetValue(tmpAfterCharValue)$
 $\leftarrow \pi_{CharacterValue}(\sigma_{ID=tmpAfter}(CChar))$

SQL-Statement:

```

select CharacterValue into :tmpAfterCharValue
from cache.CChar
where ID = :tmpAfter

```

Select the CharacterValue of the FORC-R.

$checkCharValue(confBefore) =$
 $GetValue(confBeforeCharValue)$
 $\leftarrow \pi_{CharacterValue}(\sigma_{ID=confBefore}(CChar))$

SQL-Statement:

```

select CharacterValue into :confBeforeCharValue
from cache.CChar
where ID = :confBefore

```

6.6.2 Second Step - Identifying a Conflict

After each 'If - Else If - ...' structure there is an 'Else' statement that catches exceptions of cases that should not occur, what is omitted here for reasons of simplicity.

6.6.2.1 Inactive - Inactive (0 - 0)

None of the two FORCs is active:

Check if none of the two FORCs is active.

If isLeftActive = 0 && isRightActive = 0

Check if FORC-L is the beginning of the document
and if so set the appropriate insert category.

If tmpAfter = β

insertCategory = 2

Otherwise check if FORC-R is the end of the document
and if so set the appropriate insert category.

Else If confBefore = ε

insertCategory = 3

Otherwise set the appropriate insert category
depending on the DIP.

Else If defaultInsertPolicy = 0

insertCategory = 4

Else If defaultInsertPolicy = 1

insertCategory = 5

6.6.2.2 Active - Active (1 - 1)

Both FORCs are active:

Check if both FORCs are active.

If isLeftActive = 1 && isRightActive = 1

Check if all reference characters correspond
and if so set the appropriate insert category.

If confBefore = tmpBefore && tmpAfter = confAfter

insertCategory = 1

Otherwise check if no reference character corresponds.

Else If confBefore $\neq$ tmpBefore && tmpAfter $\neq$ confAfter

Check if the simple or enhanced conflict resolution shall be used.

If simpleInsertConflictResolution = 1

Set the appropriate insert category

depending on the DIP.

If defaultInsertPolicy = 0

insertCategory = 2

Else If defaultInsertPolicy = 1

insertCategory = 3

Else If simpleInsertConflictResolution = 0

Check if SORC-R was inserted before SORC-L and if so set the appropriate insert category depending on the DIP.

If tmpBeforeCharCounter > confAfterCharCounter

If defaultInsertPolicy = 0

insertCategory = 2

Else If defaultInsertPolicy = 1

insertCategory = 3

Otherwise SORC-R was inserted after SORC-L or by copy-paste.

Check the order of CharCounters and if necessary the order of OIDs as well and set the appropriate insert category.

If fidCharCounter < tmpBeforeCharCounter

insertCategory = 2

Else If fidCharCounter > confAfterCharCounter

insertCategory = 3

Else If tmpBeforeCharCounter < fidCharCounter && fidCharCounter < confAfterCharCounter

insertCategory = 8

Check if all the characters have been inserted by the same copy-paste operation.

Else If fidCharCounter = tmpBeforeCharCounter && fidCharCounter = confAfterCharCounter

Compare the OIDs of the characters.

If $c_1 < tmpBefore$

insertCategory = 2

Else If $confAfter < c_1$

insertCategory = 3

Else If $tmpBefore < c_1$ && $c_1 < confAfter$

insertCategory = 8

Check if just the first character and the SORC-L have been inserted by the same copy-paste operation.

Else If fidCharCounter = tmpBeforeCharCounter

Compare the OIDs of the characters.

If $c_1 < tmpBefore$

insertCategory = 2
Else If *tmpBefore* < *c*₁
 insertCategory = 8
 Check if just the first character and the SORC-R have been
 inserted by the same copy-paste operation.
Else If *fidCharCounter* = *confAfterCharCounter*
 Compare the OIDs of the characters.
 If *c*₁ < *confAfter*
 insertCategory = 8
 Else If *confAfter* < *c*₁
 insertCategory = 3

6.6.2.3 Active - Inactive (1 - 0)

Only the FORC-L is active:

Check if only the FORC-L is active.

If *isLeftActive* = 1 && *isRightActive* = 0
 Check if all reference characters correspond
 and if so set the appropriate insert category.
 If *confBefore* = *tmpBefore* && *tmpAfter* = *confAfter*
 insertCategory = 1
 Otherwise check if only FORC-L and SORC-R are equal
 and if so set the appropriate insert category.
 Else If *confBefore* ≠ *tmpBefore* && *tmpAfter* = *confAfter*
 insertCategory = 2
 Otherwise check if no reference character corresponds.
 Else If *confBefore* ≠ *tmpBefore* && *tmpAfter* ≠ *confAfter*
 Check if FORC-R is the end of the document and if so set the
 appropriate insert category depending on the DIP.
 If *confBefore* = ε
 If *defaultInsertPolicy* = 0
 insertCategory = 2
 Else If *defaultInsertPolicy* = 1
 insertCategory = 3
 Else

$insertCategory = 2$

6.6.2.4 Inactive - Active (0 - 1)

Only the FORC-R is active:

Check if only the FORC-R is active.

If $isLeftActive = 0 \ \&\& \ isRightActive = 1$

Check if all reference characters correspond

and if so set the appropriate insert category.

If $confBefore = tmpBefore \ \&\& \ tmpAfter = confAfter$

$insertCategory = 1$

Otherwise check if only FORC-R and SORC-L are equal

and if so set the appropriate insert category.

Else If $confBefore = tmpBefore \ \&\& \ tmpAfter \neq confAfter$

$insertCategory = 3$

Otherwise check if no reference character corresponds.

Else If $confBefore \neq tmpBefore \ \&\& \ tmpAfter \neq confAfter$

Check if FORC-L is the beginning of the document and if so set the appropriate insert category depending on the DIP.

If $tmpAfter = \beta$

If $defaultInsertPolicy = 0$

$insertCategory = 2$

Else If $defaultInsertPolicy = 1$

$insertCategory = 3$

Else

$insertCategory = 3$

6.6.3 Third Step - Conflict Resolution

Set the variables *characterBeforeFid* and *characterAfterLid* to the appropriate characters where the insertion shall take place depending on the determined insert categories.

If $insertCategory = 1$

$characterBeforeFid = tmpAfter$

$characterAfterLid = confBefore$

If $insertCategory = 2$

$characterBeforeFid = tmpAfter$

characterAfterLid = *tmpBefore*

If insertCategory = 3

characterBeforeFid = *confAfter*

characterAfterLid = *confBefore*

If insertCategory = 4

Search for the nearest character on the left side of FORC-L that is active or the beginning of the document.

repeat

provAfter = *tmpAfter*

searchLeft(tmpAfter) =

Left(tmpId, tmpAfter, IsActive)

$\leftarrow \pi_{ID, After, IsActive}(\sigma_{ID=tmpAfter}(CChar))$

until IsActive $\neq 0 \vee provAfter = \beta$

Check if the character that was found on the left side is the beginning of the document

If IsActive $\neq 1$

tmpId = β

At that time, *tmpId* represents the character that was found on the left side.

characterBeforeFid = *tmpId*

Select the succeeding character of the character that was found on the left side.

getNext(tmpId) =

NextChar(tmpBefore) $\leftarrow \pi_{Before}(\sigma_{ID=tmpId}(CChar))$

characterAfterLid = *tmpBefore*

SQL-Statement:

```
select ID, After, IsActive into :tmpId, :tmpAfter, :IsActive
from cache.CChar
where ID = :tmpAfter
```

SQL-Statement:

```
select Before into :tmpBefore
from cache.CChar
where ID = :tmpId
```

If insertCategory = 5

Search for the nearest character on the right side of FORC-R that is active or the end of the document.

repeat

provBefore = *confBefore*

searchRight(confBefore) =

Left(tmpId, confBefore, IsActive)

$\leftarrow \pi_{ID, Before, IsActive}(\sigma_{ID=confBefore}(CChar))$

until IsActive $\neq 0 \vee$ *provBefore* = ε

Check if the character that was found on the right side is the end of the document

If IsActive $\neq 1$

tmpId = ε

At that time, *tmpId* represents the character that was found on the right side.

Select the preceding character of the character that was found on the right side.

getPrevious(tmpId) =

PrevChar(confAfter) $\leftarrow \pi_{After}(\sigma_{ID=tmpId}(CChar))$

characterBeforeFid = *confAfter*

characterAfterLid = *tmpId*

SQL-Statement:

```
select ID, Before, IsActive into :tmpId, :confBefore, :IsActive
from cache.CChar
where ID = :confBefore
```

SQL-Statement:

```
select After into :confAfter
from cache.CChar
where ID = :tmpId
```

If insertCategory = 8

Set the variables *characterBeforeFid* and *characterAfterLid* depending on the DIP

If defaultInsertPolicy = 0
 Select the succeeding character of SORC-L
 $getNext(tmpBefore) =$
 $NextChar(afterSORCL) \leftarrow \pi_{Before}(\sigma_{ID=tmpBefore}(CChar))$
 $characterBeforeFid = tmpBefore$
 $characterAfterLid = afterSORCL$
Else If defaultInsertPolicy = 1
 Select the preceding character of SORC-R
 $getPrevious(confAfter) =$
 $PrevChar(beforeSORCR) \leftarrow \pi_{After}(\sigma_{ID=confAfter}(CChar))$
 $characterBeforeFid = beforeSORCR$
 $characterAfterLid = confAfter$

SQL-Statement:

```

select Before into :afterSORCL
from cache.CChar
where ID = :tmpBefore

```

SQL-Statement:

```

select After into :beforeSORCR
from cache.CChar
where ID = :confAfter

```

6.6.4 Forth Step - Completion

Lock the first and last character of the string that shall be inserted and the characters that are situated directly before and after the character string. Continue if successful, else throw a LockException e .

If lock(characterBeforeFid) $\neq 1$ throwException(e)
If lock(characterAfterLid) $\neq 1$ throwException(e)
If lock(c_1) $\neq 1$ throwException(e)
If lock(c_n) $\neq 1$ throwException(e)

Update characters before and after the string that shall be inserted and continue if successful, else throw a GeneralException e .

status = *updateString(characterIds, characterBeforeFid, characterAfterLid)*
If status = 0 *throwException(e)*

SQL-Statement:
 update cache.CChar
 set Before = :Fid
 where ID = :characterBeforeFid

SQL-Statement:
 update cache.CChar
 set After = :Lid
 where ID = :characterAfterLid

Update the first and last characters of the string that shall be inserted and continue if successful, else throw a *GeneralException e*.

status = *linkAfter(c₁, characterBeforeFid)*
If status = 0 *throwException(e)*
status = *linkBefore(c_n, characterAfterLid)*
If status = 0 *throwException(e)*

SQL-Statement:
 update cache.CChar
 set After = :characterBeforeFid
 where ID = :Fid

SQL-Statement:
 update cache.CChar
 set Before = :characterAfterLid
 where ID = :Lid

Adjust the variable *nextCharacterId* to the correct character.

nextCharacterId = *characterAfterLid*

If there exists an exception, release all locks, rollback the transaction and quit with return value 0, else continue.

If $\exists e$:

releaseLocks

rollbackTA

quit(0)

Release all locks.

releaseLocks

Commit transaction.

commitTA

If at least two characters of the insertion string do not have consecutive OIDs (checked at the beginning) inform other clients and own client about the insertion of each single character that is inserted.

Otherwise all characters of the insertion string have consecutive OIDs. Thus, inform other clients and own client about the insertion of the entire string.

If notConsecutiveInsertion $\neq 0$

informInsertOwnAndOthersSingleChar(characterIds, s)

Else

informInsertOwnAndOthers(characterIds, s)

Quit algorithm and return the OID of the first character that was inserted.

quit(c₁)

6.7 Remove Algorithm invoked from Undo/Redo

Finally, the formal depiction of a removal from an undo or redo is presented.

As it is described in the subsections 6.4 and 6.5 the remove algorithm, which is implemented in the class method **RemoveCChars** of the class **CChar** is invoked from the class methods **CUndo** or **CRedo** of the class **CChar** if an undo respectively a redo was triggered by a user. Those class methods invoke the remove algorithm with several parameters. Therefore, we assume that at the beginning of the algorithm specific information is available from those parameters. The important parameter that is used to determine that the invocation comes from an undo or redo is **flag**. This parameter is set to 1 if the invocation comes from an undo and it is set to 2 if it comes from a redo.

The parameters that are relevant are introduced in the following listing and will be used for the definitions beneath:

- `characterIds` (List): list of characters, which have to be removed
- `previousCharacterId` (OID): reference character situated previously to the string to be removed
- `flag` (Integer): indicates whether the remove operation is called from an undo function: 1, from a redo function: 2, or from a client removal: 0

Furthermore, there are the Globals `UndoCounter` and `RedoCounter` that shall be accessible by the variable `undoCounter` and `redoCounter`. There are also a flag `emptyRedoHistoryAfterEdit` that is either set to 1 if the `RedoHistory` shall be deleted or to 0 if the `RedoHistory` shall not be deleted after a removal and a flag `allowNonConsecutiveRemoval` that is either set to 1 if the removal of non consecutive characters is allowed or to 0 if it shall not be allowed.

Since there is the distinction of allowing or not allowing the removal of non consecutive characters we divide the formal depiction in two steps. The first step (section 6.7.1) applies to both the allowing and the not allowing case. In the second step that particularly describes how a character of the character list is removed, it is distinguished between not allowing (section 6.7.2) and allowing (section 6.7.3).

The appropriate flowcharts for this algorithm are given in appendix A.8 and A.9 on pages 187 and 188 depending on the strategy ANCR.

To describe the remove algorithm invoked from undo or redo, some definitions have to be added:

The function *informRemoveOwnAndOthers* informs all clients about the removal of a character list l in session s . This is done by calling the appropriate service routine with the appropriate parameters.

informRemoveOwnAndOthers(l, s) = *inform all clients about removal of l in s*

The function *informInsertOwnAndOthers* informs all clients about the insertion of a character list l in session s . This is done by calling the appropriate service routine with the appropriate parameters.

informInsertOwnAndOthers(l, s) = *inform all clients about insertion of l in s*

The function *addToList* adds a character c to a character list l

addToList(c, l) = *add c to l*

6.7.1 First Step - Allowing and Not Allowing Removal of Consecutive Characters

Thus, the removal of characters triggered by an undo or redo can be defined stepwise as follows: Start a transaction.

startTA

Check the write access on the document session s that has the document session OID s .

checkWriteAccess(s) =
 WriteAccess(documentSessionWrite)
 $\leftarrow \pi_{WriteOption}(\sigma_{ID=s}(CDocumentSession))$

SQL-Statement:

```
select WriteOption into :documentSessionWrite
from CDocumentSession
where ID = :documentSessionId
```

Lock the first character c_1 of the list *characterIds* and continue if successful, else throw a *LockException* e .

If lock(c₁) ≠ 1 throwException(e)

Select the character that is situated before the first character c_1 .

getPrevious(c₁) =
 PrevChar(prevCharId) $\leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar))$

SQL-Statement:

```
select After into :prevCharId
from cache.CChar
where ID = :startCharId
```

Continue, if the selection was successful, else throw a *GeneralException* e .

If prevCharId = ∅ throwException(e)

Acquire the lock for the character that is situated before the first character that shall be removed and continue if successful, else throw a *LockException* e .

If lock(prevCharId) ≠ 1 throwException(e)

Select the character that is situated after the last character c_n that shall be removed.

$getNext(c_n) =$
 $NextChar(nextCharId) \leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

SQL-Statement:

```
select Before into :nextCharId
from cache.CChar
where ID = :endCharId
```

Continue, if the selection was successful, else throw a `GeneralException` e .

If $nextCharId = \emptyset$ *throwException*(e)

Acquire the lock for the character that is situated after the last character that shall be removed and continue if successful, else throw a `LockException` e .

If $lock(nextCharId) \neq 1$ *throwException*(e)

Different to 'normal' invocation:

If the invocation comes from an undo, increment the redoCounter and if it comes from a redo, increment the undoCounter.

If $flag = 1$ *increment*(redoCounter)

If $flag = 2$ *increment*(undoCounter)

At that time, the list *characterIds* contains the characters c_1 to c_n that shall be removed.

$characterIds = \{c_1, \dots, c_n\}$

The characters that are located directly before the first character, the first character itself and the one located directly after the last character of the list of characters that shall be removed are already locked whereupon the removal of each character can take place.

6.7.2 Second Step - Not Allowing Removal of Consecutive Characters

Each character of the string is removed by setting its status to inactive (`IsActive=0`) and each character is filled into the `UndoHistory`.

For each character c_i of *characterIds* :

Select the character before the actual character that shall be removed

$getPrevious(c_i) =$

$PrevChar(after) \leftarrow \pi_{After}(\sigma_{ID=c_i}(CChar))$

Select the character after the actual character that shall be removed

$getNext(c_i) =$

$NextChar(before) \leftarrow \pi_{Before}(\sigma_{ID=c_i}(CChar))$

SQL-Statement:

```
select After, Before, ElementBorder
into :after, :before, :elementBorder
from cache.CChar
where ID = :id
```

If the selection was not successful throw a *GeneralException* e

If $after = \emptyset \vee before = \emptyset$ *throwException*(e)

Continue, if the actual character is consecutive to the preceding one, else throw a *GeneralException* e

If $after \neq previousCharacterId$ *throwException*(e)

If write access is granted, continue, else throw a *GeneralException* e

If $documentSessionWrite \neq 1$ *throwException*(e)

Remove the actual character c_i if it is neither the beginning nor the end of the document, else throw a *GeneralException* e

$checkCharValue(c_i) =$

$GetValue(charValue) \leftarrow \pi_{CharacterValue}(\sigma_{ID=c_i}(CChar))$

If $charValue = \beta \vee charValue = \varepsilon$ *throwException*(e)

Else *removeChar*(c_i)

SQL-Statement:

```
update cache.CChar set IsActive = 0
where ID = :id
and CharacterValue <> 99998 and CharacterValue <> 99999
```

Different to 'normal' invocation:

Now it is distinguished if the removal comes from an undo or a redo.

If the removal comes from an undo (flag=1), the character is marked as not relevant anymore for the undo and the entry is inserted into the RedoHistory.

If the removal comes from a redo (flag=2), the character is marked as relevant again for the undo and the entry is deleted from the RedoHistory. Only continue, if the updates are successful, else throw a GeneralException e .

If flag = 1

status = markNotRelevantForUndo(c_i)

If status = 0 throwException(e)

status = insertInRedoHistory(c_i)

If status = 0 throwException(e)

If flag = 2

status = markRelevantForUndo(c_i)

If status = 0 throwException(e)

status = deleteFromRedoHistory(c_i)

If status = 0 throwException(e)

From undo, SQL-Statement:

```
update cache.CUndoHistory set IsActive = 0
where CharId = :id and ActionId = 1
```

From undo, SQL-Statement:

```
insert into cache.CRedoHistory
(CharId, After, Before, DocumentId, UserId, ActionId,
TimeStamps, CharCounter, Offset)
values (:id, :after, :before, :fileId, :userId, 1,
:stamp, :redoCounter, :where)
```

From redo, SQL-Statement:

```
update cache.CUndoHistory set IsActive = 1
where CharId = :id and ActionId = 2
```

From redo, SQL-Statement:

```
delete from cache.CRedoHistory
where CharId = :id and ActionId = 2
```

Set previousCharacterId to the actual character c_i

previousCharacterId = c_i

Whereas $c_i \in \{c_1, \dots, c_n\}$

Afterwards, it applies:

$$\forall c_i \in characterIds \Rightarrow IsActive = 0$$

Different to 'normal' invocation:

The RedoHistory is not emptied.

Check if the preceding character is active or if it is the beginning of the document and if so continue, else throw a GeneralException e .

```
checkPreceding(prevCharId) =
    IsOK(IsActive, CharacterValue)
    ← πIsActive, CharacterValue(σID=prevCharId(CChar))
    If IsActive ≠ 1 ∧ CharacterValue ≠ β throwException(e)
```

Check if the succeeding character is active or if it is the end of the document and if so continue, else throw a GeneralException e .

```
checkSucceeding(nextCharId) =
    IsOK(IsActive, CharacterValue)
    ← πIsActive, CharacterValue(σID=nextCharId(CChar))
    If IsActive ≠ 1 ∧ CharacterValue ≠ ε throwException(e)
```

Link the preceding and the succeeding characters to each other and continue if successful.

```
status1 = linkBefore(prevCharId, nextCharId)
status2 = linkAfter(nextCharId, prevCharId)
If status1 ≠ 1 ∨ status2 ≠ 1 throwException(e)
```

SQL-Statement:

```
update cache.CChar set Before = :nextCharId
where ID = :prevCharId
and (IsActive=1 or CharacterValue=99998)
```

SQL-Statement:

```
update cache.CChar set After = :prevCharId
where ID = :nextCharId
and (IsActive=1 or CharacterValue=99999)
```

If there exists an exception, release all locks, rollback the transaction and quit with return value 2 (no write access) or 0 (any other exception), else continue.

If $\exists e$:
 releaseLocks
 rollbackTA
 If documentSessionWrite $\neq 1$ quit(2)
 Else quit(0)

Release all locks.

releaseLocks

Commit transaction.

commitTA

Different to 'normal' invocation:

Inform own and other clients about the removal of the entire string.

informRemoveOwnAndOthers(characterIds, s)

Quit algorithm and return 1.

quit(1)

6.7.3 Second Step - Allowing Removal of Consecutive Characters

Different to 'ANCR=0' invocation:

The characters that are directly situated before the first character c_1 and after the last character c_n are stored.

Select the preceding character of the first character that shall be removed

getPrevious(c_1) =
 PrevChar(areaAfter) $\leftarrow \pi_{After}(\sigma_{ID=c_1}(CChar))$

Select the succeeding character of the last character that shall be removed

getNext(c_n) =
 NextChar(areaBefore) $\leftarrow \pi_{Before}(\sigma_{ID=c_n}(CChar))$

SQL-Statement:

```
select After
into :areaAfter
from cache.CChar
where ID = :firstCharacterId
```

SQL-Statement:

```

select Before
    into :areaBefore
from cache.CChar
where ID = :lastCharacterId

```

Different to 'ANCR=0' invocation:

Continue, if the first character is consecutive to the preceding one, else throw a `GeneralException e`

If areaAfter $\neq$ previousCharacterId throwException(e)

Each character of the string is removed by setting its status to inactive (`IsActive=0`) and each character is filled into the `UndoHistory`. If there are characters in between the ones that shall be removed, the characters are separated.

For each character c_i of characterIds :

Select the character before the actual character that shall be removed

getPrevious(c_i) =

PrevChar(after) $\leftarrow \pi_{After}(\sigma_{ID=c_i}(CChar))$

Select the character after the actual character that shall be removed

getNext(c_i) =

NextChar(before) $\leftarrow \pi_{Before}(\sigma_{ID=c_i}(CChar))$

SQL-Statement:

```

select After, Before, ElementBorder
    into :after, :before, :elementBorder
from cache.CChar
where ID = :id

```

If the selection was not successful throw a `RemoveConflictResolutionException e`

If after = $\emptyset \vee$ before = $\emptyset$ throwException(e)

Different to 'ANCR=0' invocation:

A control variable is used to check if the actual character is consecutive with the succeeding one. If c_i is the last character that shall be removed, it has to be consecutive to the next character.

If $c_i \neq c_n$

```

    nextCharacterId = ci+1
Else
    nextCharacterId = before

```

At that time, the list *before* holds the OID of the next character and *nextCharacterId* holds the OID of the next character that shall be removed.

Different to 'ANCR=0' invocation:

If the actual and next character that shall be removed are not consecutive in the document, separate all characters that are in between and store them in the list *charactersInBetween*.

```

If before ≠ nextCharacterId
    charInBetween = before
    repeat
        addToList(charInBetween, charactersInBetween)
        Select the character after charInBetween and continue, if the
        selection was successful, else throw a
        RemoveConflictResolutionException e
        getNext(charInBetween) =
            NextChar(charInBetween)
            ← πBefore(σID=charInBetween(CChar))
        If charInBetween = ∅ throwException(e)

    until charInBetween = nextCharacterId

```

SQL-Statement:

```

select Before
    into :nextOfInBetweenId
from cache.CChar
where ID = :inBetweenId

```

If write access is granted, continue, else throw a `GeneralException e`

```

If documentSessionWrite ≠ 1 throwException(e)

```

Remove the actual character c_i if it is neither the beginning nor the end of the document, else throw a `GeneralException e`

```

checkCharValue(ci) =

```

$GetValue(charValue) \leftarrow \pi_{CharacterValue}(\sigma_{ID=c_i}(CChar))$
If $charValue = \beta \vee charValue = \varepsilon$ *throwException*(e)
Else *removeChar*(c_i)

SQL-Statement:

```

update cache.CChar set IsActive = 0
where ID = :id
and CharacterValue <> 99998 and CharacterValue <> 99999
  
```

Different to 'normal' invocation:

Now it is distinguished if the removal comes from an undo or a redo.

If the removal comes from an undo (flag=1), the character is marked as not relevant anymore for the undo and the entry is inserted into the RedoHistory.

If the removal comes from a redo (flag=2), the character is marked as relevant again for the undo and the entry is deleted from the RedoHistory.

Only continue, if the updates are successful, else throw a *GeneralException* e .

If $flag = 1$

$status = markNotRelevantForUndo(c_i)$
If $status = 0$ *throwException*(e)
 $status = insertInRedoHistory(c_i)$
If $status = 0$ *throwException*(e)

If $flag = 2$

$status = markRelevantForUndo(c_i)$
If $status = 0$ *throwException*(e)
 $status = deleteFromRedoHistory(c_i)$
If $status = 0$ *throwException*(e)

From undo, SQL-Statement:

```

update cache.CUndoHistory set IsActive = 0
where CharId = :id and ActionId = 1
  
```

From redo, SQL-Statement:

```

insert into cache.CRedoHistory
(CharId, After, Before, DocumentId, UserId, ActionId,
 TimeStamps, CharCounter, Offset)
values (:id, :after, :before, :fileId, :userId, 1,
 :stamp, :redoCounter, :where)
  
```

From redo, SQL-Statement:

```
update cache.CUndoHistory set IsActive = 1
where CharId = :id and ActionId = 2
```

From redo, SQL-Statement:

```
delete from cache.CRedoHistory
where CharId = :id and ActionId = 2
```

Set previousCharacterId to the actual character c_i

$previousCharacterId = c_i$

Whereas $c_i \in \{c_1, \dots, c_n\}$

Afterwards, it applies:

$\forall c_i \in characterIds \Rightarrow IsActive = 0$

If there are characters in between the character string that is removed, link all characters in order to maintain a valid ring of double linked fields.

Different to 'ANCR=0' invocation:

If $charactersInBetween \neq \emptyset$

Update the characters that are not removed and continue if successful

For each character c_i of $charactersInBetween$:

If $c_i = c_1$

$status = updateCharacter(c_i, areaAfter, c_{i+1})$

Else If $c_i = c_n$

$status = updateCharacter(c_i, c_{i-1}, areaBefore)$

Else

$status = updateCharacter(c_i, c_{i-1}, c_{i+1})$

If $status \neq 1$ throwException(e)

Update the characters that are removed and continue if successful

For each character c_i of $characterIds$:

If $c_i = c_1$

$status = linkBefore(c_i, c_{i+1})$

Else If $c_i = c_n$

$status = linkAfter(c_i, c_{i-1})$

Else

$status = updateCharacter(c_i, c_{i-1}, c_{i+1})$

If $status \neq 1$ throwException(e)

Different to 'normal' invocation:

The RedoHistory is not emptied.

Check if the preceding character is active or if it is the beginning of the document and if so continue, else throw a GeneralException e .

```

checkPreceding(prevCharId) =
    IsOK(IsActive, CharacterValue)
    ←  $\pi_{IsActive, CharacterValue}(\sigma_{ID=prevCharId}(CChar))$ 
    If  $IsActive \neq 1 \wedge CharacterValue \neq \beta$  throwException( $e$ )

```

Check if the succeeding character is active or if it is the end of the document and if so continue, else throw a GeneralException e .

```

checkSucceeding(nextCharId) =
    IsOK(IsActive, CharacterValue)
    ←  $\pi_{IsActive, CharacterValue}(\sigma_{ID=nextCharId}(CChar))$ 
    If  $IsActive \neq 1 \wedge CharacterValue \neq \varepsilon$  throwException( $e$ )

```

If there are no characters in between, link the preceding and the succeeding characters to each other, else link them to the first, respectively last character that is in between and continue if successful.

Different to 'ANCR=0' invocation:

```

If charactersInBetween =  $\emptyset$ 
    status1 = linkBefore(prevCharId, nextCharId)
    status2 = linkAfter(nextCharId, prevCharId)
Else
    status1 = linkBefore(prevCharId, charactersInBetween.c1)
    status2 = linkAfter(nextCharId, charactersInBetween.cn)
If status1  $\neq 1 \vee$  status2  $\neq 1$  throwException( $e$ )

```

SQL-Statement:

```

update cache.CChar set Before = :nextCharId
where ID = :prevCharId
and (IsActive=1 or CharacterValue=99998)

```

SQL-Statement:

```

update cache.CChar set After = :prevCharId
where ID = :nextCharId
and (IsActive=1 or CharacterValue=99999)

```

SQL-Statement:

```
update cache.CChar set Before = :firstInBetweenId
where ID = :prevCharId
and (IsActive=1 or CharacterValue=99998)
```

SQL-Statement:

```
update cache.CChar set After = :lastInBetweenId
where ID = :nextCharId
and (IsActive=1 or CharacterValue=99999)
```

If there exists an exception, release all locks, rollback the transaction and quit with return value 2 (no write access) or 0 (any other exception), else continue.

If $\exists e$:

releaseLocks

rollbackTA

If documentSessionWrite $\neq 1$ quit(2)

Else quit(0)

Release all locks.

releaseLocks

Commit transaction.

commitTA

Inform own and other clients about the removal of the entire string and if there are characters in between inform also of the reinsertion of the characters in between.

Different to 'ANCR=0' invocation:

If charactersInBetween = $\emptyset$

informRemoveOwnAndOthers(characterIds, s)

Else

informRemoveOwnAndOthers(characterIds, s)

informInsertOwnAndOthers(charactersInBetween, s)

Quit algorithm and return 1.

quit(1)

6.8 Summary

In this chapter, the complete formal depiction of all algorithms used for the unDOit algorithm was presented. Moreover, the 'normal' insertion and removal algorithms (not invoked from an undo/redo) have been described as well in order to point out the differences. The following listing gives a brief overview of the depicted algorithms:

- General insert algorithm
- General remove algorithm
- Undo algorithm
- Redo algorithm
- Insert algorithm invoked from an undo/redo
- Remove algorithm invoked from an undo/redo

Part III

Documentation

Chapter 7

Implementation

The presented undo/redo framework is embedded in the TeNDaX system. A detailed description of the implemented concurrency control and the formal depiction of all algorithms that are used, is given in chapter 5 on pages 38 et seqq. and chapter 6 on pages 97 et seqq. The framework is an enhancement of the former framework with its underlying GDU algorithm. Although the single class methods were completely rewritten, the architecture of the implementation, i.e. the classes, methods and attributes, which are used, stay almost the same. This architecture is well described in [Jost, 2004, pages 96-107] and therefore, we present in this chapter the differences of the two architectures and the responsibility of the classes.

As described in section 4.2 on pages 25 et seqq., the TeNDaX system consists of a database and a client. As database the product InterSystems Caché is used while the client is programmed in JavaTM¹.

7.1 Technology

The implementation of the undo/redo framework is implemented with InterSystems' Caché Post-Relational Database in Caché Objectscript .

Caché offers three integrated possibilities to access the data in the database: an object-based access, the Caché SQL dialect and a multidimensional access [InterSystems, 2004]. For the implementation, embedded SQL and object-based access is generally used to access and manipulate the data.

¹<http://java.sun.com>

7.1.1 Caché Objectscript

Caché Objectscript is an object-oriented programming language designed for building database applications. It offers object-based access to the database and is well-suited for a variety of applications including:

- Business logic
- Application integration
- Data processing

7.2 Overview

In this section, the data model is presented using ORM². Furthermore, the classes are listed that were changed and extended during this thesis.

Figure 7.1 shows an overview of the TeNDaX data model, in which the most important classes of the undo/redo framework are highlighted.

In the following listing the changes in the undo/redo framework are pointed out:

CChar The class methods **CUndo**, **CRedo**, **InsertCChars** and **RemoveCChars** are extensively changed to fulfill the demand on the new concurrency control and the extended functionality.

CFileNode The class method **NewCDocument** is changed in order to create for each document three new Globals that store the strategy preferences for DIP, ANCR and ERAE.

CUndoHistory and **CRedoHistory** In these classes, a new index is set on the attribute **CharCounter** to achieve a higher performance.

The class **CUndoRedoHistoryPrinter** was newly created to serve only developing purposes. Its class method **WriteContentAsHTML** is able to print out the document and/or history content as HTML³.

²ORM: **O**bject **R**ole **M**odeling is a semantic modeling approach, which is used to describe the world in terms of objects and roles.

³HTML = **H**yper**T**ext **M**arkup **L**anguage

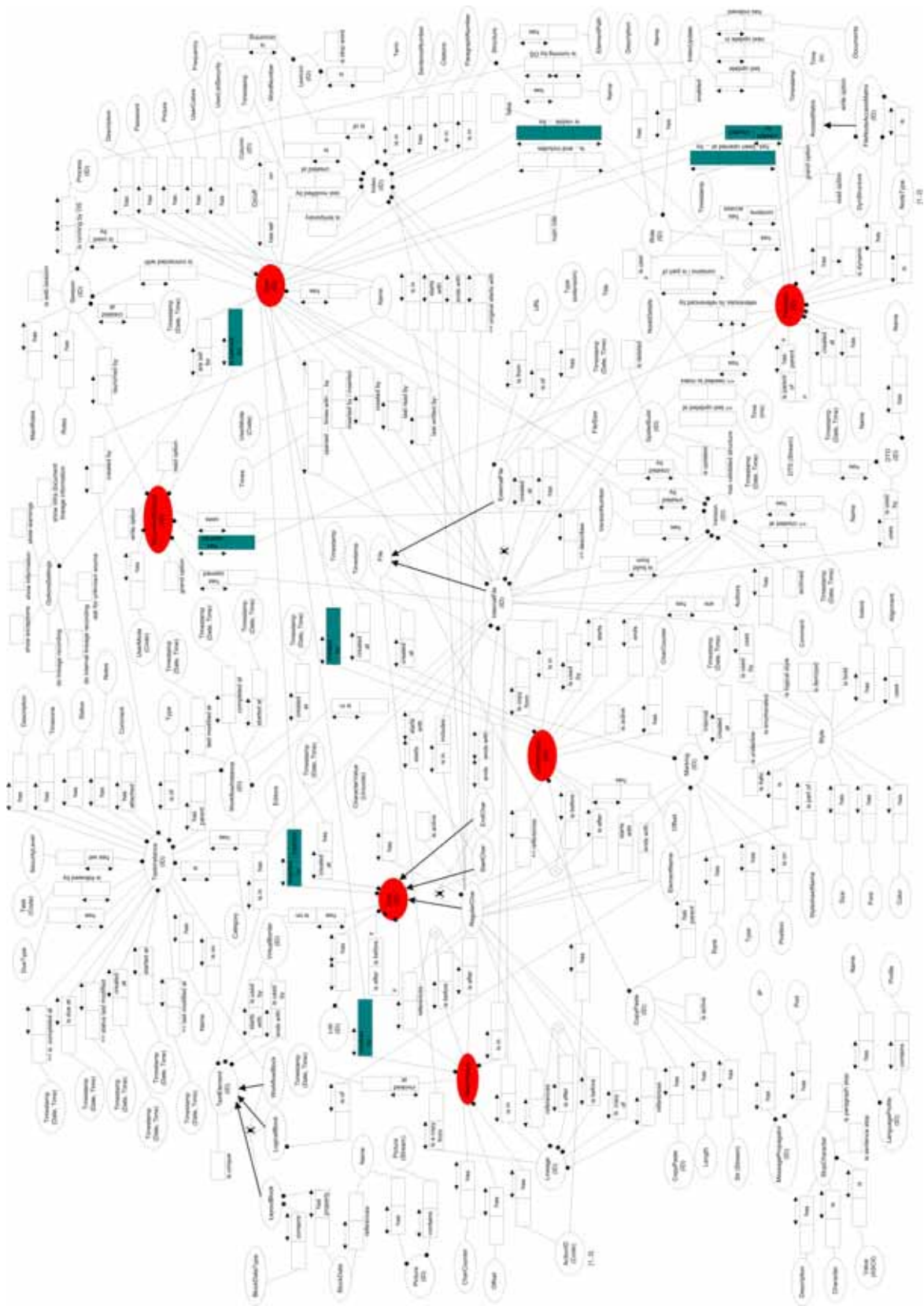


Figure 7.1: ORM - Overview

7.3 Responsibility of the Participating Classes

This section gives a short overview of the most important classes and their basic responsibility in the unDOit algorithm.

As described in section 4.2 the TeNDaX system consists of one central database and an arbitrary number of clients. To trigger an undo or redo the user has the possibility to click on one of the four undo/redo buttons in the graphical user interface of a client. Given that in the implemented undo/redo there is the distinction between the undoing/redoin of the last operation of any user (global undo/redo) and the undoing/redoin of the last operation of each individual user (local undo/redo). Consequently, there exist four according buttons in the user interface: global undo, local undo, local redo and global redo cf. figure 7.2).

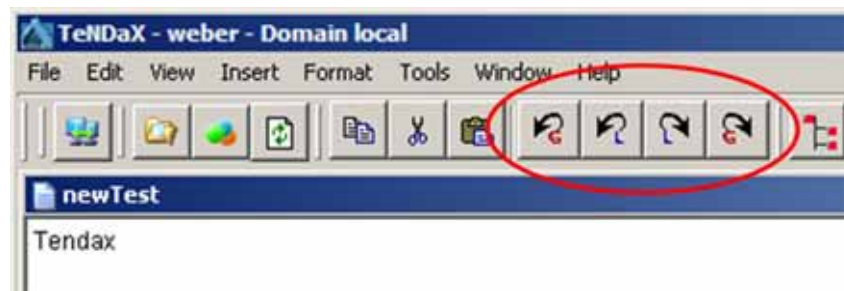


Figure 7.2: Undo/Redo Buttons (own contribution)

If one of those buttons is pressed, the appropriate *ActionListener*⁴ triggers the correct action in the client. The following classes on the client side are involved in this process: *CEToolbarCreator*, *CEEEventController*, *CEManager*, *CacheExchange*, *TDBExchange*, *CacheDB* and *CChar*.

The method invocations in these classes have the almost exclusive purpose to forward the request of an operation to the database. Almost the complete implementation of the undo/redo algorithm is realized in the database. Therefore, the corresponding methods in the database class `cache.CChar` are invoked through the database interface of the client. If a global undo is triggered by the client, the class method `CUndo` in the class `cache.CChar` is invoked with a flag set to true (global undo, 1). If a global redo is triggered, the class method `CRedo` in the class `cache.CChar` is invoked with a flag set to true (global redo, 1). Whenever a local undo or redo shall be executed, the same methods as mentioned above are triggered, but with a flag set to false (no global undo/redo, 0).

⁴ActionListener is a listener interface for receiving action events in JavaTM.

One could distinguish different parts of the undo/redo algorithm, an overall control flow that manages the sequence of actions if an operation is triggered and a special control to manage occurring conflicts. The conflict resolution is not completely separated but rather embedded in the overall control flow. In this section, the overall control flow is briefly depicted since the complex conflict resolution is already discussed in section 5 on pages 38 et seqq.

7.3.1 Undo / Redo

As mentioned above, the method `CUndo` respectively `CRedo` in the class `cache.CChar` are invoked, if a user triggers an undo/redo. The distinction between global and local undo/redo is handled within the appropriate method, but most of the control flow is the same. The structure of those two methods `CUndo` and `CRedo` is very alike. Generally it can be said that the methods are responsible for:

- starting a transaction
- checking the write access for the document
- evaluating the appropriate operation that shall be undone/redone (e.g. the deletion of a string)
- gathering the information about that operation (e.g. the characters of the string and the adjacent reference characters)
- invoking the appropriate method with the necessary information (e.g. `cache.CChar.InsertCChars`)
- checking the success of the undo/redo
- handling the entries to the history files (`CUndoHistory` and `CRedoHistory`)
- committing a transaction or performing a rollback

Only a minor part of the conflict resolution is implemented in those methods but an essential one for the conflict resolution of a removal. If an undo of an insertion or a redo of a removal is triggered, it is checked if at least one of the characters is already deleted.

7.3.2 Insert / Remove

As described in section 4.3.1 on page 34 and in section 4.3.2 on page 35 the insertion of characters is implemented in the method `InsertCChars` of the class `cache.CChar` while the removal of characters is implemented in the method `RemoveCChars` of that class. These methods have been changed as well, in order to satisfy the increased demand on undo/redo. Most of the conflict resolution is implemented in those two methods.

The control flow of those methods, not invoked from an undo or redo, differs from the control flow of those methods if they are invoked from an undo or redo. If `InsertCChars` is triggered by a normal insert operation, it is generally responsible for:

- starting a transaction
- checking the write access for the document
- locking the adjacent reference characters of the string that shall be inserted
- inserting the string at the appropriate position
- handling the entries to the history files (`CUndoHistory` and `CRedoHistory`)
- updating the reference characters of the string that shall be inserted in order to ensure the validity of the chain list
- releasing all locks and committing a transaction or performing a rollback
- informing of all other clients about the insertion

If `InsertCChars` is triggered by an undo or redo, the insertion is replaced by a reactivation and the control flow is extended with a conflict resolution that analyzes the circumstances of the insertion and decides, where to insert the string. Since the document state probably has changed in the elapsed time between the original operation and the undo or redo respectively, the decision is possibly depending on the strategy DIP (cf. section 5.3.2.1 on page 48).

Like the control flow of `InsertCChars` the one of `RemoveCChars` is extended with a conflict resolution if triggered by an undo or redo. Indeed, the conflict resolutions of those two methods are almost not comparable because the conflicts that can occur during an insertion or a removal are very different.

If `RemoveCChars` is triggered by a normal remove operation, it is generally responsible for:

- starting a transaction
- locking the adjacent reference characters and the first character of the string that shall be removed
- checking the write access for the document
- performing a 'soft delete' for each character of the string that shall be removed
- handling the entries to the history files (`CUndoHistory` and `CRedoHistory`)
- updating the reference characters of the string that shall be removed in order to ensure the validity of the chain list
- releasing all locks and committing a transaction or performing a rollback
- informing of all other clients about the removal

If `RemoveCChars` is triggered by an undo or redo, it depends on the strategy ANCR (cf. section 5.3.2.1 on page 48) whether non consecutive characters are removed or not.

7.4 Summary

The undo/redo framework with its underlying unDOit algorithm is implemented as a part of the TeNDaX system. It is based on the architecture of a former framework but the algorithm has changed extensively. Table 7.1 gives an overview of the changed and newly created classes.

	<i>Class</i>	<i>I(ndex)/M(ethod)</i>	<i>Purpose</i>
Changed Classes	CChar	CUndo (M)	Undo Control Flow
		CRedo (M)	Redo Control Flow
		InsertCChars (M)	Insert Conflict Resolution
		RemoveCChars (M)	Remove Conflict Resolution
	CFileNode	NewCDocument (M)	New Globals to store preferences for DIP, ANCR and ERAE
	CUndoHistory	CharCounter (I)	Index on CharCounter
	CRedoHistory	CharCounter (I)	Index on CharCounter
Newly Created Class	CUndoRedo-Printer	WriteContent-AsHTML (M)	Writes out Document, UndoHistory and RedoHistory Content as HTML

Table 7.1: Changed and Newly Created Classes (own contribution)

Part IV

Evaluation

Chapter 8

Conclusion

In this chapter it is summarized what has been achieved in this thesis. It is described how the TeNDaX system was improved and which results can be used for other systems as well. First, we review the chronological order of how the system was enhanced. Then, we present the main achievements in matters of stability, usability and performance.

8.1 Retrospection

In the following listing the development of the new undo/redo framework is presented. It is mentioned what measures were taken and what kind of result has been reached.

- Studied new/other solutions
→ No better solution found
- Analyzed and improved the concurrency control
→ Stability improved
- Research for multi-user test cases
→ no information found
- Test cases created, system tested and improved
→ New insights for concurrency control
→ Stability improved
- Introduced exception handling
→ New insights for concurrency control
→ Stability improved

- Performance measured, analyzed and improved
→ Performance improved
- Introduced configurable concurrency control
→ Flexible concurrency control
→ Usability improved
- New test cases created, system tested and improved
→ Complete system tested with three users
→ Stability improved
- Detailed description of the system
→ Detailed description of the concurrency control
→ Collection of test cases
→ Detailed formal depiction of all algorithms (insert, remove, undo, redo)

8.2 Stability

One of the main issues of the former solution was the poor stability. The undo/redo framework of the TeNDaX system has been analyzed in detail and the necessary corrections are made. The insights for the improvement of stability were reached not just by analyzing the system but with specific testing. The completely tested concurrency control that is enhanced with numerous special cases and a collection of test cases is well described in chapter 5 on pages 38 et seqq. Furthermore, the framework is now implemented with exception handling, i.e. it is checked, if database queries and other critical statements were successfully executed or not. If an exception should be thrown, the system would not just crash but would be able to react on the unattended situation. Moreover, all exceptions are clearly distinguishable what facilitates the exception handling in the system.

The testing of the TeNDaX system with the created test cases was a main task to ensure system stability. Thus, we briefly summarize how the system now handles the different puzzles (cf. section 5.1.1 on page 38):

Ambiguous reference and Do-undo-pair trap These puzzles are analyzed in detail and the solution is enhanced to an absolute deterministic behavior that covers all known special cases.

Insert-delete-undo dilemma The new framework is enhanced to a totally different conflict resolution that does not skip a character if it was just edited

by another user. The new conflict resolution checks if a character is already deleted by another user. Therefore, undo operations that affect characters, which were edited by other users are now generally possible.

Copy-paste-insert dilemma The new algorithms are enhanced to a totally different conflict resolution that is configurable and allows the removal of non consecutive characters and therefore the undoing of an area that was edited in the meantime.

Reference Character Not Existent Anymore The new solution is enhanced to a more powerful, stable and deterministic conflict resolution that is even configurable.

Table 8.1 presents the summarized puzzle enhancements.

<i>Status</i>	<i>Puzzle</i>	<i>Enhancement</i>
Solved because of the database based approach	Insert-insert-tie dilemma	
	Overlapping deletes puzzle	
Solved with the GDU algorithm, but although enhanced	Insert-delete-undo dilemma	new conflict resolution
	Copy-paste-insert dilemma	new configurable conflict resolution
Enhanced because not properly solved with the GDU algorithm	Ambiguous reference	deterministic and stable
	Do-undo-pair trap	deterministic and stable
	Reference character not existent anymore	deterministic and stable

Table 8.1: Puzzle Enhancements (own contribution)

8.3 Usability

The new undo/redo framework uses more meta data of the TeNDaX system for the concurrency control. This meta data is used to find the best conflict resolution that is possible. Thus, the algorithm is more powerful what increases the probability to match the user's intention. Furthermore, the concurrency control is now configurable and consequently improves the usability as well.

8.4 Performance

To achieve a higher performance, a time measuring system was implemented after the stability of the concurrency control was ensured. The results identified the code snippets that were responsible for the low performance. The new indexing is extended in the classes `CUndoHistory` and `CRedoHistory` where an index is set on the attribute `CharCounter`.

A significantly higher performance was achieved but after this improvement the implementation was changed in order to enhance the system to a configurable concurrency control. Consequently, the former results of the measuring system are not valid anymore and a new measuring system should be implemented to get new results that can be presented.

8.5 Summary

The goal of this diploma thesis was to enhance the existing TeNDaX undo/redo framework in matters of stability, usability and performance and to ensure the availability of a working prototype (embedded in the TeNDaX research project). Therefore, a new concept for the concurrency control has been developed and implemented in the TeNDaX system.

The new undo/redo framework:

- is more powerful because the concurrency control uses more meta data to find the best conflict resolution,
- is more stable because the concurrency control was completely tested, enhanced with several special cases and extended with exception handling,
- has an improved usability because the concurrency control is now configurable and
- has a higher performance because of the new indexing method.

The main results of this thesis are the enhanced TeNDaX system as described above, the collection of test cases that can be used for any system and the exact formal depiction of the algorithms used.

Chapter 9

Summary

This thesis presents an undo/redo framework for a native text database system, realized in the TeNDaX research project. The framework, with its algorithms presented in chapters 5 and 6, is a state-of-the-art solution for an undo/redo in a collaborative environment.

In contrast to many other multi-user undo concepts the framework is embedded in a native text database system and is not based on operational transformation. Due to the database based collaborative system, the framework advantages, since all relevant data is stored persistently in the database what offers the possibility to access the document history even if the document has not been edited for weeks. Thus, there exists no limitation for logged operations and undo/redo operations of former sessions.

The TeNDaX undo/redo solution stands out due to its usability. The user can choose at any time between a local and a global undo/redo while the concurrency control is configurable and consequently can be adjusted to react differently to occurring collaborative conflicts.

The main results of this thesis are the undo/redo framework with its advanced concurrency control embedded in the TeNDaX system, the collection of test cases that can be used for any system and the exact formal depiction of the algorithms used.

Chapter 10

Future Investigations

The aim of a sophisticated undo algorithm is to enable on the one hand the undoing of the entire document after it is finished and on the other hand to redo the entire document when it is undone until its beginnings. Thus, in any moment it should be possible to go one step forth and one step back.

The framework presented in this thesis builds a reliable basis for an effective collaborative undo/redo. Although, it can still be improved particularly in matters of functionality and usability to fulfill the requirement stated above.

A future extension could build upon the unDOit algorithm without concerning about 'low-level' conflicts. To give an idea what can be realized, we provide in this chapter an outlook of possible further enhancements.

10.1 Possible Extended Functionality

The implemented undo/redo framework handles the actions insertion and removal of characters but in TeNDaX there are a lot more possibilities to edit a document, like editing the layout, specifying zones and so forth. Thus, an advanced collaborative undo solution should provide not only text editing but also advanced undo mechanisms like layout undo, selective undo and others.

This could be achieved by expanding the action log types in the existing solution. At the moment, there is an `ActionId` stored for each log entry in `CUndoHistory`. As far as implemented, these `ActionId`'s are defined as follows:

- `ActionId = 1` is an insert operation
- `ActionId = 2` is a delete operation

By defining a new type of log entry it is possible to extend the types of operations that are logged and that can be undone.

In the following subsections is described what kind of extensions could be implemented.

10.1.1 Selective Undo

A selective undo would allow a user to undo any operation in any order. E.g., the selective undo is implemented in the collaborative editors DistEdit of [Prakash and Knister, 1994] and REDUCE of [Sun, 2000, Sun, 2002]. In the context of TeNDaX and the existing undo/redo framework, the editor would have to be extended with a graphical user interface that allows the user to choose several operations from a list of all logged operations that can be undone afterwards.

In combination with the local undo, the list would only show the operations, which the user executed by itself. In combination with the global undo, the list would show all the operations from the history of the appropriate document.

One could imagine some other criteria to show lists of operations with certain properties, like all properties that concern layout operations.

10.1.2 Regional Undo

The regional undo would allow the user to mark a certain zone in the document and to execute a global or local undo in this area. If there would be other types of operations that are logged, these operations could be applied as well to any region of the document.

Consequently, different users, who are responsible for different zones in a document, could retrace all their executed operations in a certain area and could undo them, if necessary.

10.1.3 Intelligent Undo

The undo mechanism of most collaborative editors only undo the last logged operation. Other word processing systems such as Microsoft Word¹ uses an intelligent undo that for example undoes several operations in just one transaction due to the typing rate of the user.

It could also be possible to implement an undo mechanism that undoes words, parts of sentences or whole paragraphs in one step.

¹<http://office.microsoft.com>

10.1.4 Time Slot Based Undo

Another possibility could be a time slot based undo that undoes all operations, which have been executed in a certain time range. In this case the user should be provided with a graphical user interface that allows him to enter the desired time range, in which the operations are undone.

10.1.5 User-specific Undo

With the aid of a user specific undo an administrator could undo all operations of a certain user in the appropriate document. Such an undo would be parametrized with the identifier of the document and the user.

10.1.6 Combination of Different Undo Types

Naturally, a combination of the described undo types would be possible. Thus, a user could undo all operations of another user that are executed in the last two weeks in the introduction of a certain document.

10.1.7 Undo of Layout

It would be very useful to have an undo that undoes layout operations (font, font style, size). This approach could be implemented in TeNDaX by logging such operations in the history.

10.1.8 Undo of Access Rights

The system could be extended with other functionalities that undo the granting of access rights or the creation of new users in the TeNDaX security system.

10.2 Possible Enhancements

As described above, there exist numerous possibilities for additional functionality. In this section, enhancements are presented that do not extend the functionality but nevertheless would improve the present framework.

10.2.1 Usability

The unDOit algorithm with its configurable concurrency control builds a solid basis for a flexible interaction between the user and the conflict resolution system. At

the present time, the preferences of the concurrency control are set in the database if a document is created. It would be useful to have a graphical user interface that provides at least an administrator with the editable configuration preferences. Furthermore, the user should have the possibility to decide, which strategy has to be applied for a conflict by setting either the own conflict resolution preferences in advance or by choosing one of several options, which the system could present if a user decision is advisable. In the first case, the present solution would be extended with the appropriate user interface and the storage of user preferences. In the second case, where a user is asked what his intention or decision is, the conflict resolution would have to be adapted to handle such situations. I.e., if a situation occurs where the concurrency control has to ask the user for a decision, the system would have to send a message to the user and react later on his decision. Probably, the document state would have already changed in the meantime, which could cause additional conflicts.

10.2.2 Performance

With the measuring system that was implemented during this thesis, the code snippets, which were responsible for a low performance, were successfully identified. After the performance improvement the code was extensively changed due to the realization of the configurable concurrency control.

A significantly higher performance was achieved but the former results of the measuring system are not valid anymore. Consequently, a new measuring system should be implemented to get new results that can be used for another performance enhancement.

10.3 Summary

In this chapter, we have proposed different enhancements of the present undo/redo framework in the TeNDaX system, which are summarized in table 10.1.

	<i>Extension/Enhancement</i>
Possible Extended Functionality	Selective Undo
	Regional Undo
	Intelligent Undo
	Time Slot Based Undo
	User-specific Undo
	Combination of Different Undo Types
	Undo of Layout
	Undo of Access Rights
Possible Enhancements	Usability
	Performance

Table 10.1: Outlook Overview (own contribution)

Appendix A

Algorithm Flow Charts

The following flowcharts describe all algorithms used for the TeNDaX undo/redo framework.

- Global / Local Undo (page 181)
- Global / Local Redo (page 182)
- Insertion not from Undo/Redo (page 183)
- Insertion from Undo/Redo (page 184)
- Detail Insert Conflict Resolution from Undo/Redo with Update (page 185)
- Removal Overview (page 186)
- Detail Removal not from Undo/Redo or with ANCR=0 (page 187)
- Detail Removal from Undo/Redo and with ANCR=1 (page 188)
- Detail Update for Removal from Undo/Redo and with ANCR=1 (page 189)

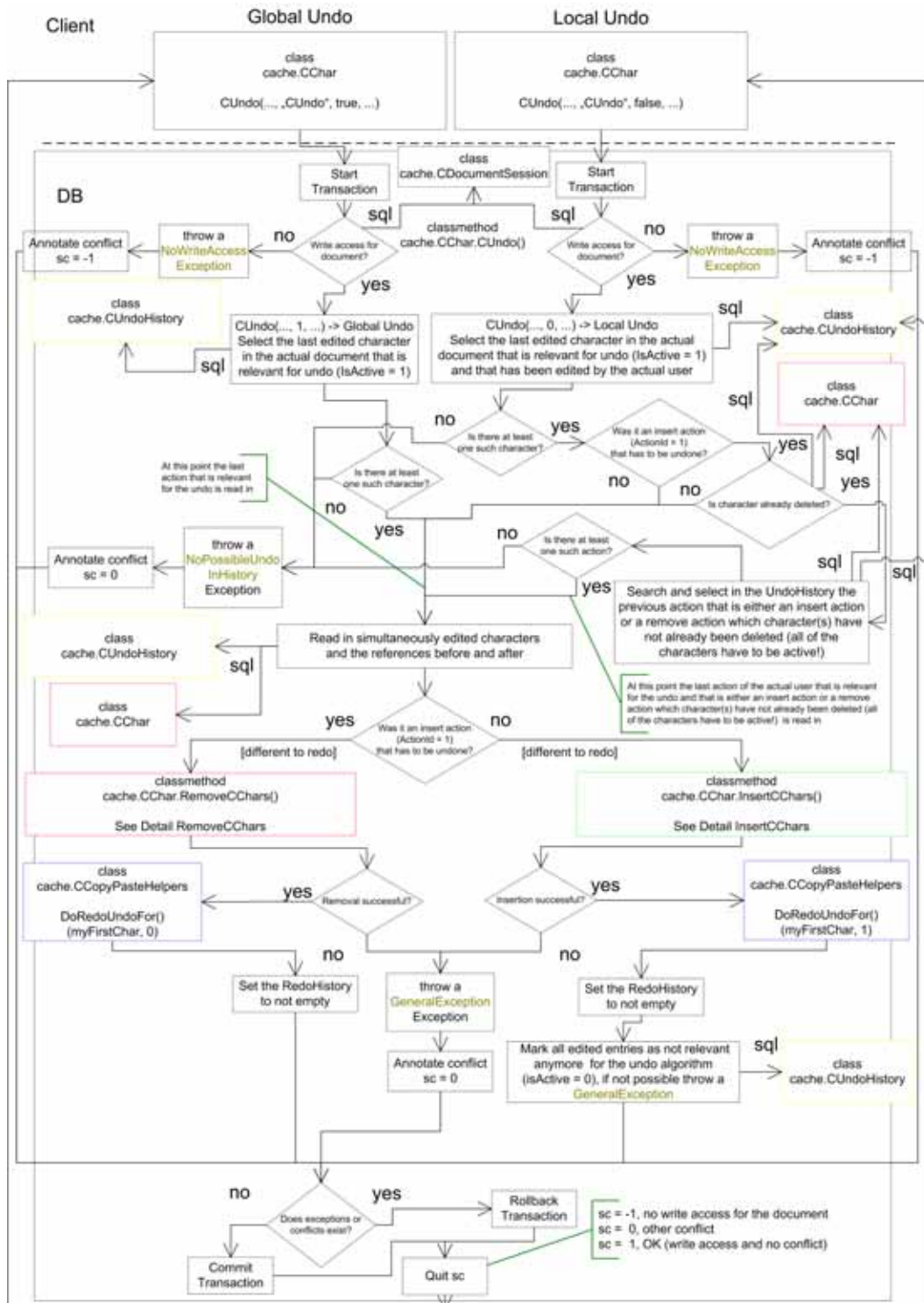


Figure A.1: Flowchart Undo (own contribution)

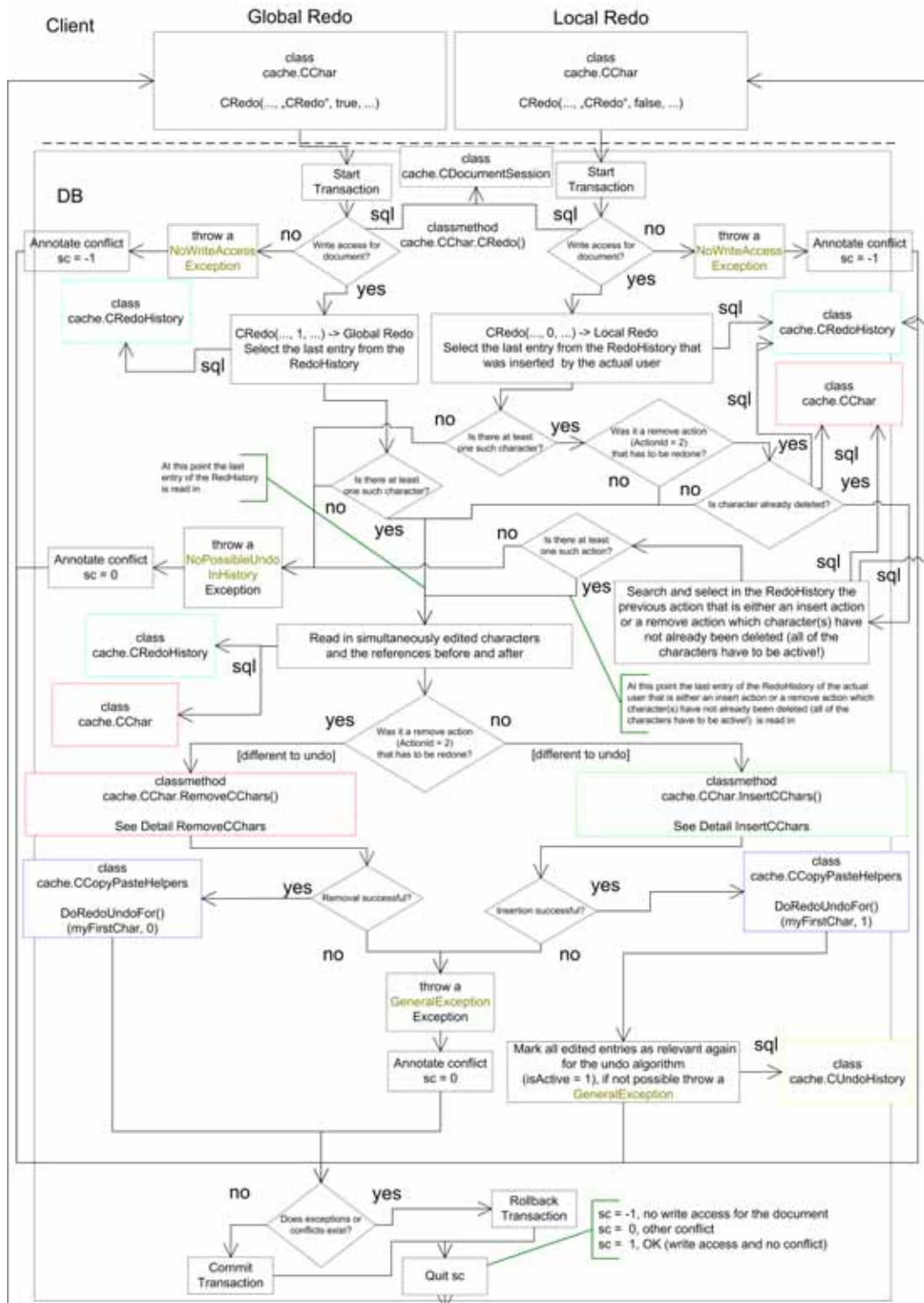


Figure A.2: Flowchart Redo (own contribution)

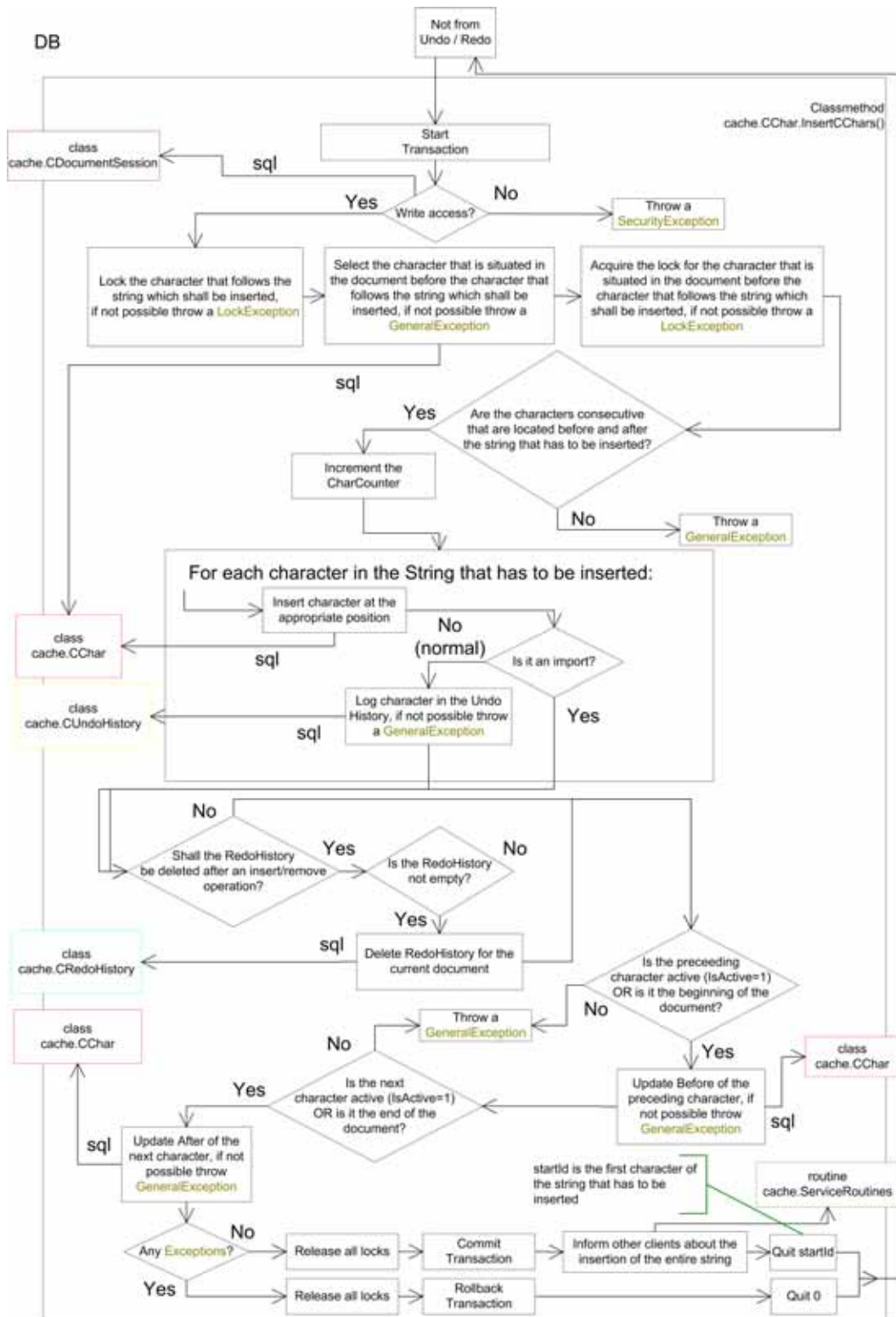


Figure A.3: Flowchart InsertCChars Not From Undo/Redo (own contribution)

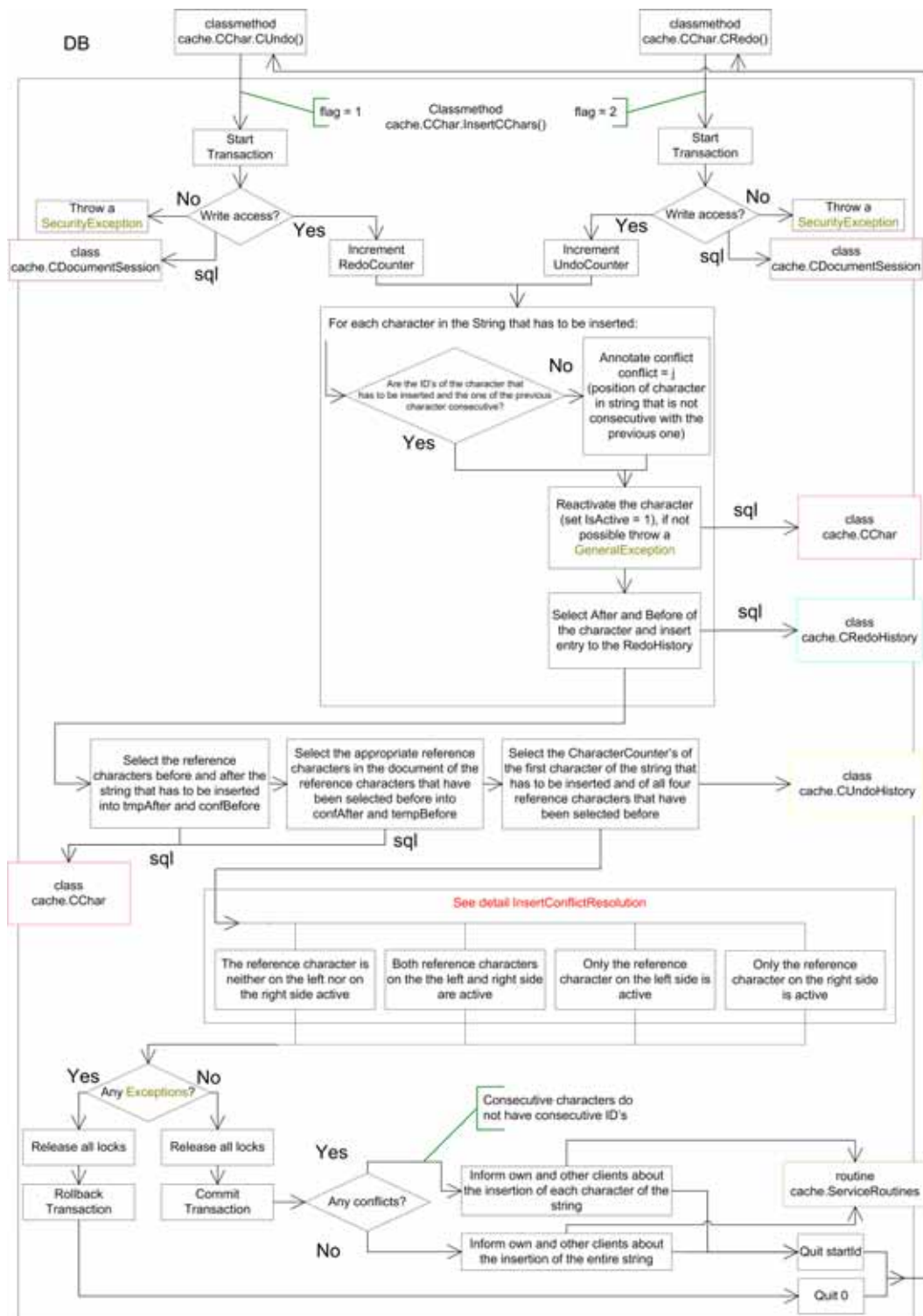


Figure A.4: Flowchart InsertCChars From Undo/Redo Overview (own contribution)

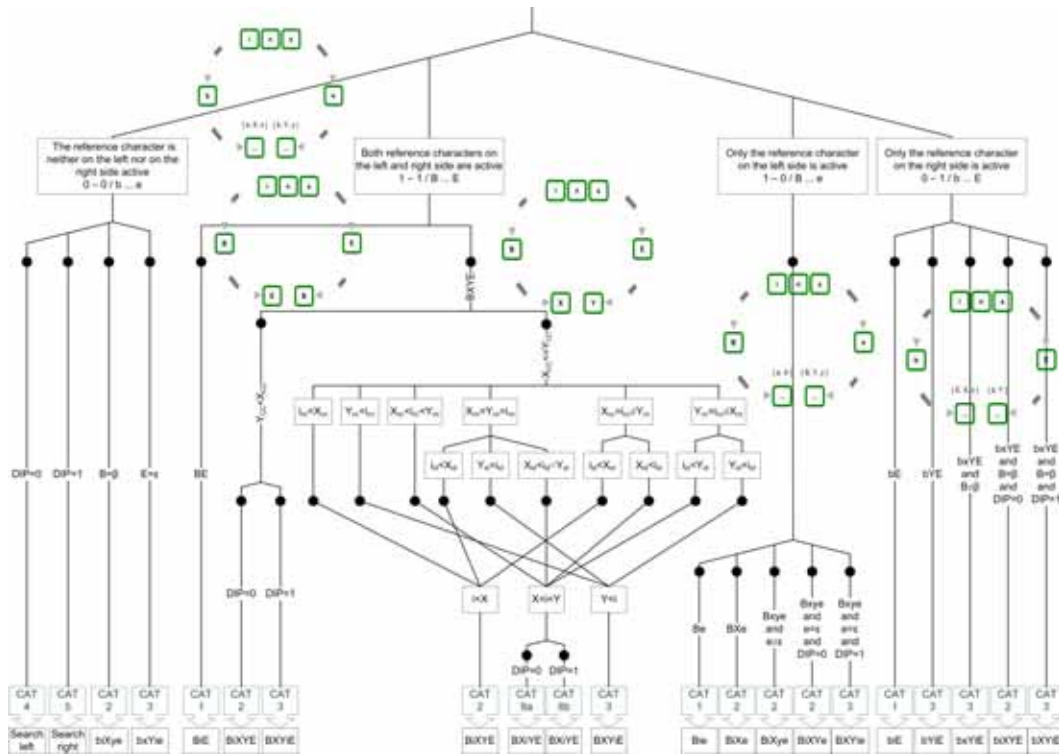


Figure A.5: Flowchart Detail Insert Conflict Resolution From Undo/Redo (own contribution)

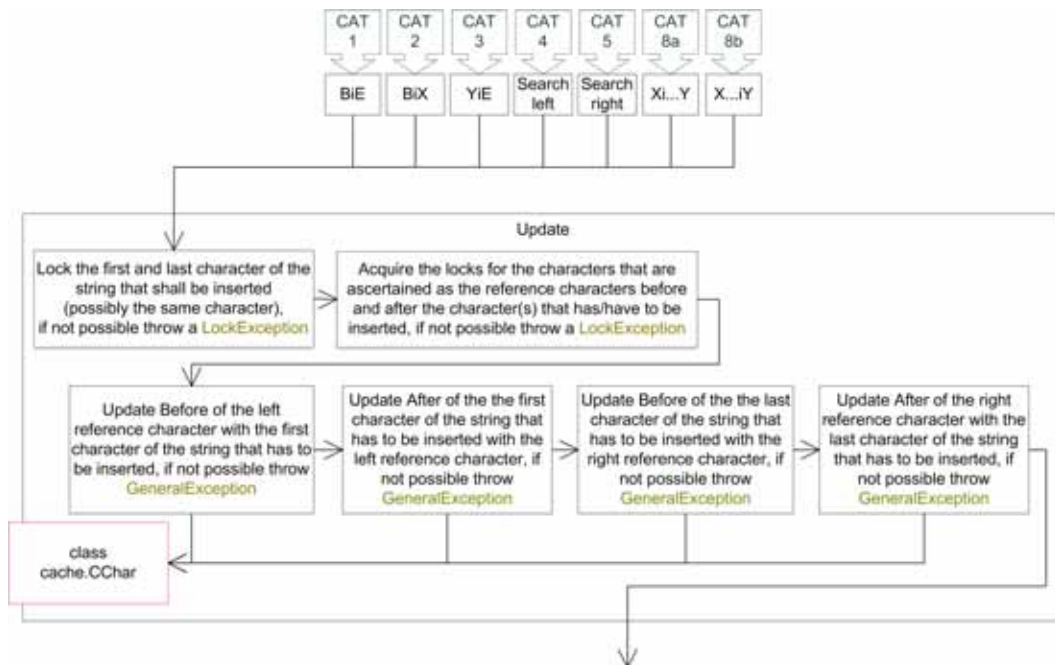


Figure A.6: Flowchart Detail Insert From Undo/Redo Category Update (own contribution)

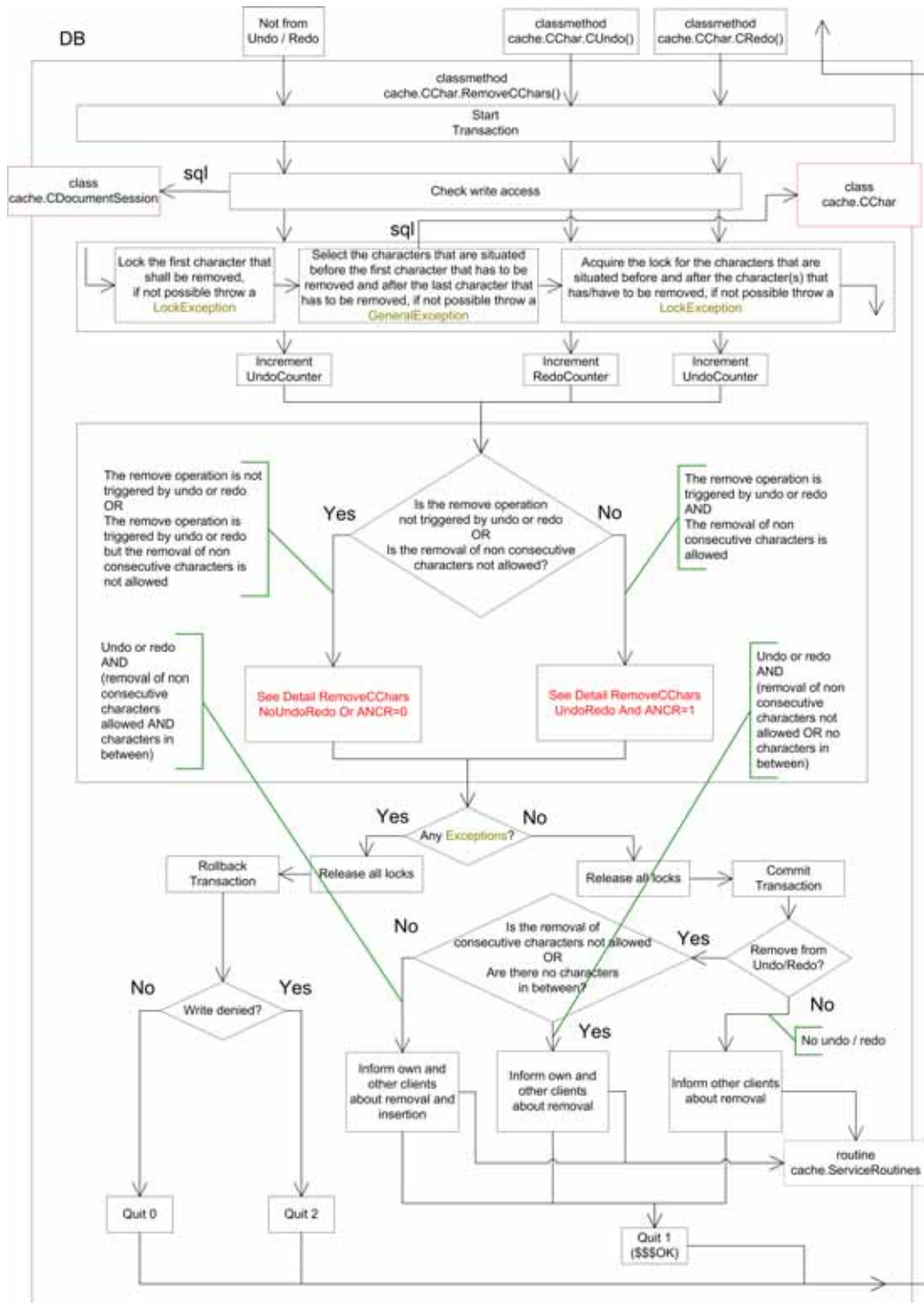


Figure A.7: Flowchart RemoveCChars Overview (own contribution)

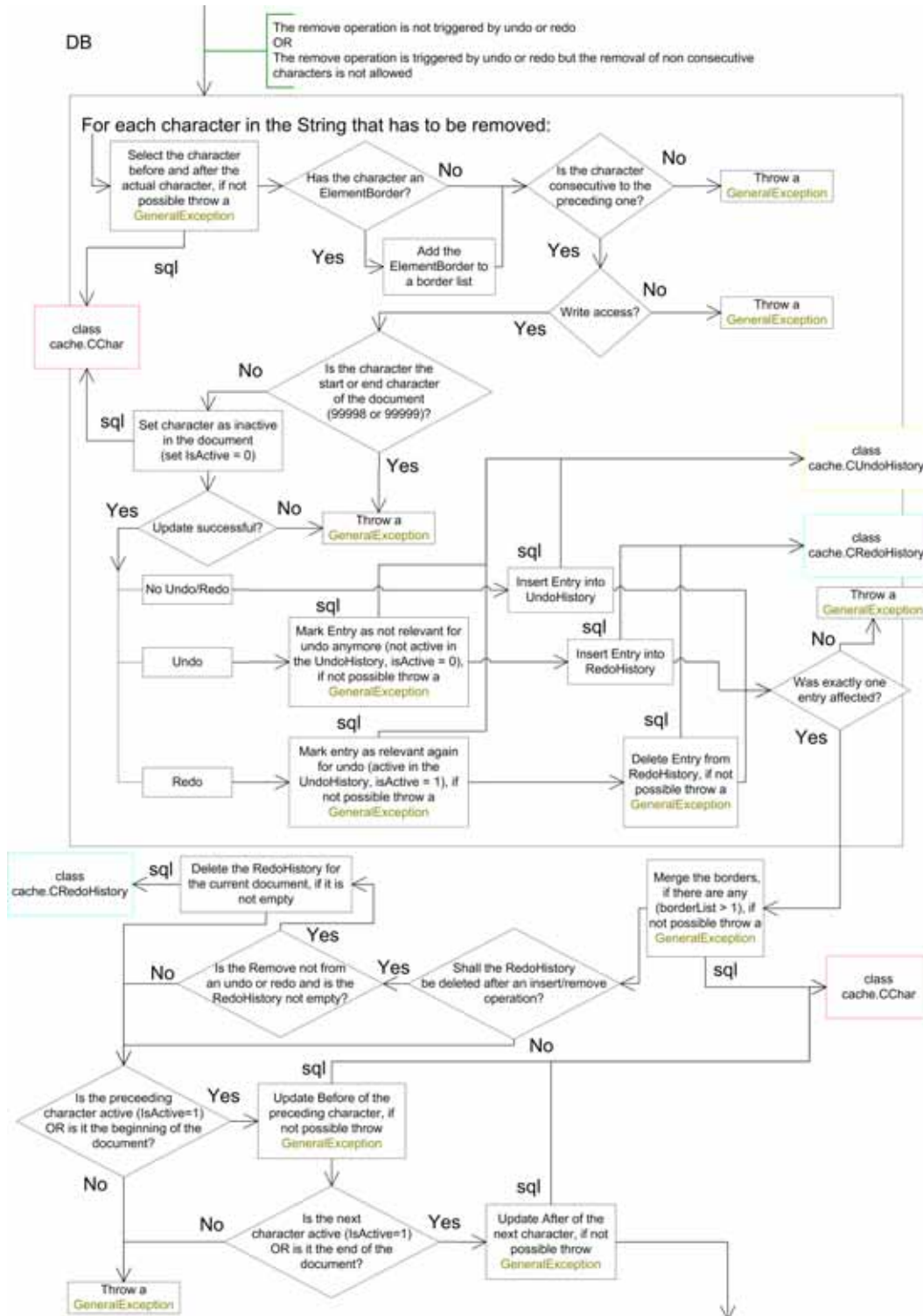


Figure A.8: Flowchart Detail RemoveCChars Not From Undo/Redo or ANCR=0 (own contribution)

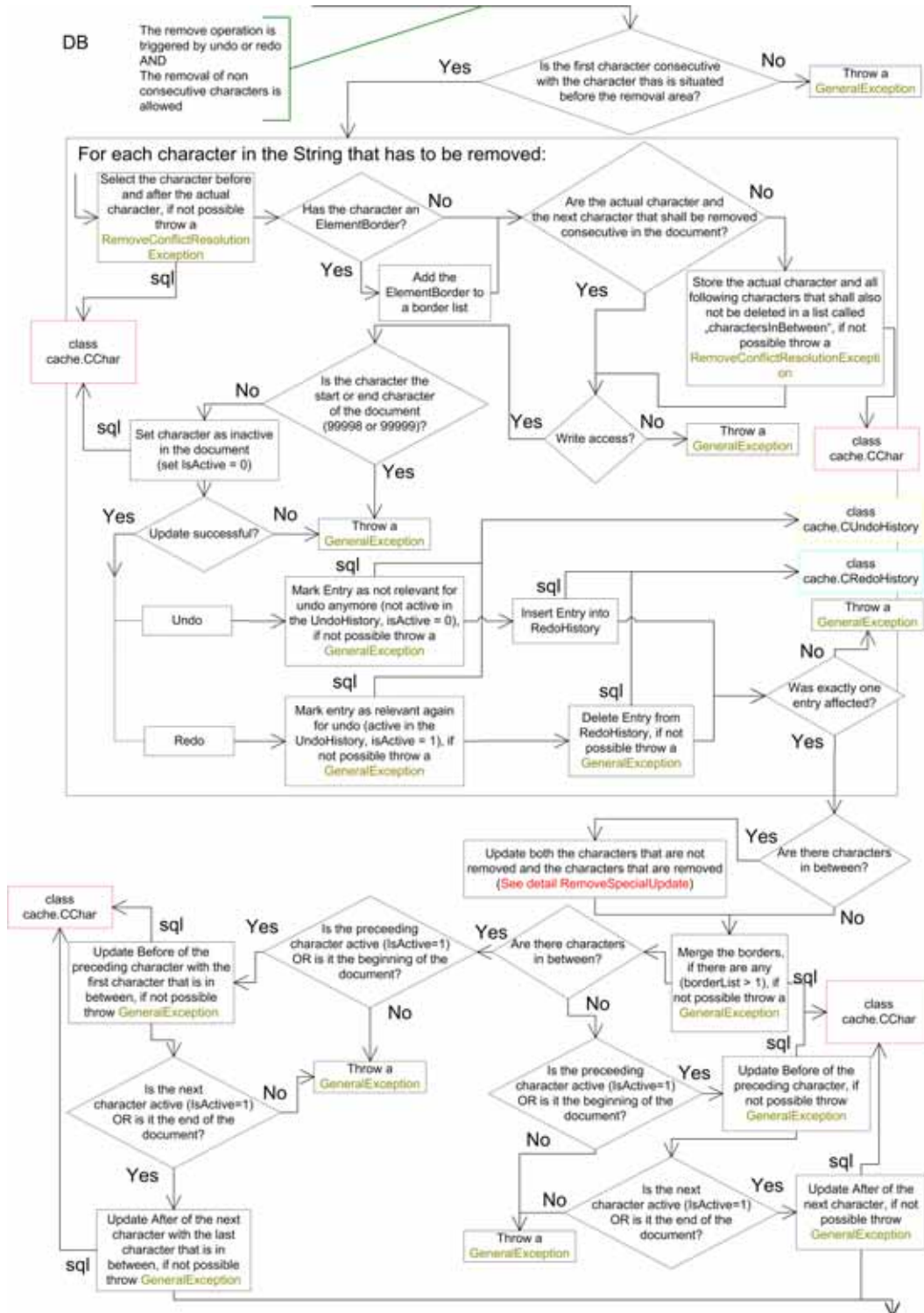


Figure A.9: Flowchart Detail RemoveCChars From Undo/Redo and ANCR=1 (own contribution)

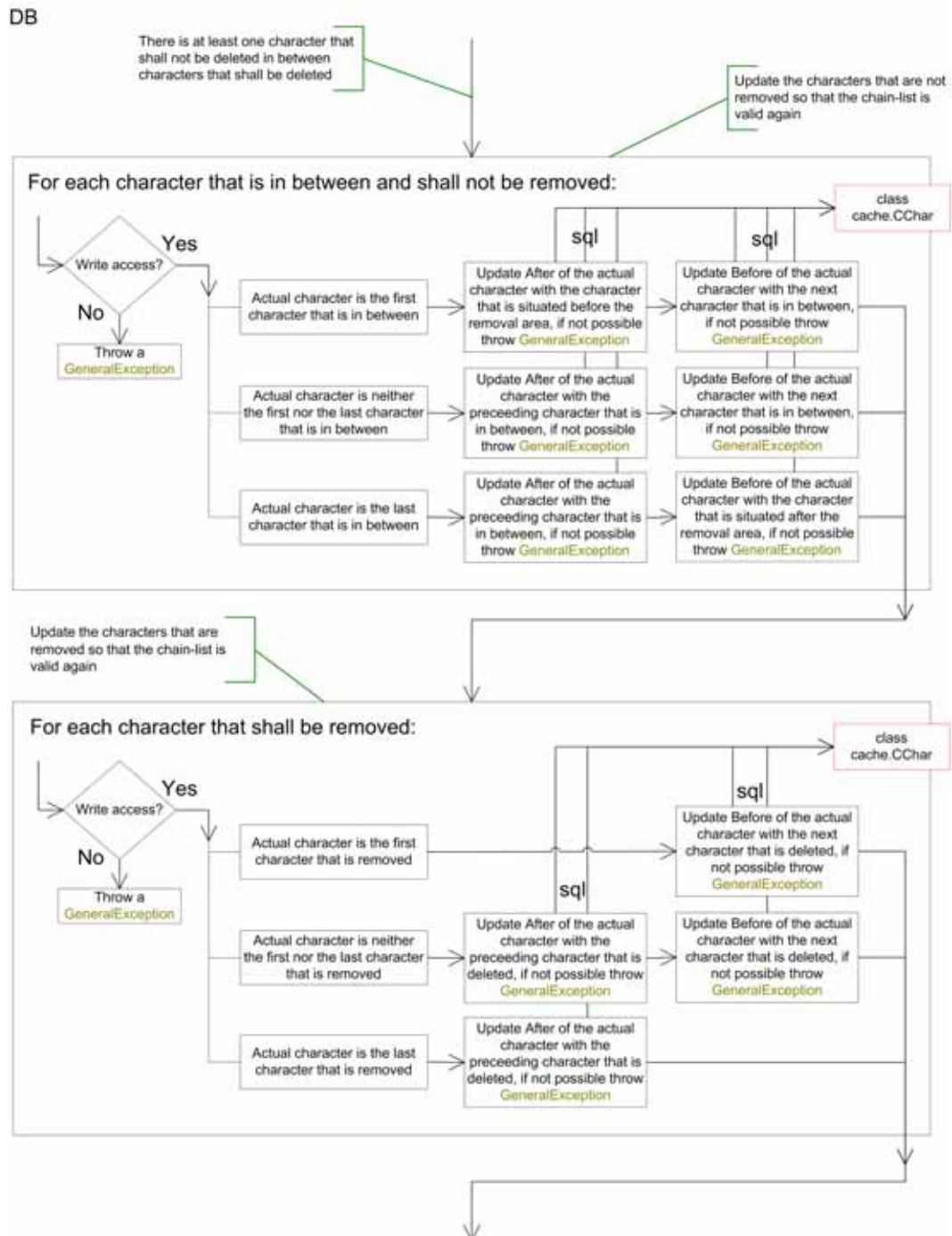


Figure A.10: Flowchart Detail Remove Special Update (own contribution)

Bibliography

- [Abiteboul et al., 2003] Abiteboul, S., Agrawal, R., Bernstein, P., Carey, M., Ceri, S., Croft, B., DeWitt, D., Franklin, M., Molina, H. G., Gawlick, D., Gray, J., Haas, L., Halevy, A., Hellerstein, J., Ioannidis, Y., Kersten, M., Pazzani, M., Lesk, M., Maier, D., Naughton, J., Schek, H., Sellis, T., Silberschatz, A., Stonebraker, M., Snodgrass, R., Ullman, J., Weikum, G., Widom, J., and Zdonik, S. (2003). The Lowell Database Research Self-Assessment Meeting. <http://research.microsoft.com/~gray/lowell>. Last visited: March 30th 2005.
- [Abowd and Dix, 1992] Abowd, G. D. and Dix, A. J. (1992). Giving undo attention. <http://www.comp.lancs.ac.uk/computing/users/dixa/papers/undo92/undo.pdf> Last visited: May 12th 2005. *Interacting with Computers*, 4(3): 317-342.
- [Berlage, 1994] Berlage, T. (1994). A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3):269–294.
- [Berlage and Genau, 1993] Berlage, T. and Genau, A. (1993). A framework for shared applications with a replicated architecture. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 249–257, New York, NY, USA. ACM Press.
- [Choudhary and Dewan, 1995] Choudhary, R. and Dewan, P. (1995). A General Multi-User Undo/Redo Model. In *Proceedings of European Conference on Computer Supported Work*, pages 231–246.
- [Dittrich, 2004] Dittrich, K. (2004). Script of Lecture: 'KV Datenbanksysteme, WS 2004/2005'. University of Zurich. Published in German.
- [Dubacher, 2003] Dubacher, M. (2003). Gleichzeitiges, datenbankbasiertes Bearbeiten von Textdokumenten. Master's thesis, University of Zurich. Published in German.

- [Edwards, 1997] Edwards, W. K. (1997). Flexible conflict detection and management in collaborative applications. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 139–148. ACM Press.
- [Ellis and Gibbs, 1989] Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM Press.
- [Ellis et al., 1988] Ellis, C. A., Gibbs, S. J., and Rein, G. (1988). Design and use of a group editor. Technical report.
- [Ferri et al., 2004] Ferri, J., Vidot, N., and Cart, M. (2004). Concurrent Undo Operations in Collaborative Environments Using Operational Transformation. In *CoopIS/DOA/ODBASE 2004, LNCS 3290*, pages 155–173. Springer-Verlag Berlin Heidelberg.
- [Hacmac, 2003] Hacmac, R. (2003). Dynamische Ordner. Master’s thesis, University of Zurich. Published in German.
- [Hammer et al., 1981] Hammer, M., Ilson, R., Anderson, T., Gilbert, E., Good, M., Niamir, B., Rosentein, L., and Schoichet, S. (1981). The implementation of Etude, an integrated and interactive document production system. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 137–146.
- [Hodel and Dittrich, 2004a] Hodel, T. B. and Dittrich, K. R. (2004a). A collaborative, real-time insert transaction for a native text database system. Information Resources Management Association, IRMA 2004.
- [Hodel and Dittrich, 2004b] Hodel, T. B. and Dittrich, K. R. (2004b). A native text database: What for? Information Resources Management Association, IRMA 2004.
- [Hodel-Widmer and Dittrich, 2005] Hodel-Widmer, T. B. and Dittrich, K. R. (2005). Concept and prototype of a collaborative business process environment for document processing. *Data & Knowledge Engineering*, 52(1):61–120.
- [InterSystems, 2004] InterSystems (2004). Caché On-Line Documentation for Caché 5.0.10. <http://platinum.intersystems.com/csp/docbook/DocBook.UI.Page.cls>. Last visited: March 30th 2005.

- [Jost, 2004] Jost, M. (2004). Undo in einer Kollaborativen Datenbankbasierten Textverarbeitung. Master's thesis, University of Zurich. Published in German.
- [Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.
- [Maendli, 2004] Maendli, M. (2004). TeNDaX Algorithmus Formale Fassung. Master's thesis, University of Zurich. Published in German.
- [Mancini et al., 1996] Mancini, R., Dix, A., and Levialdi, S. (1996). Reflections on Undo. Technical report, University of Hudderseld. <http://www.comp.lancs.ac.uk/computing/users/dixa/papers/undo-techrep-96/tech9611.pdf>. Last visited: May 5th 2005.
- [Müller, 2004] Müller, S. (2004). Data Lineage in Dokument Management Systemen am Beispiel von Copy-Paste Aufzeichnungen. Master's thesis, University of Zurich. Published in German.
- [Prakash and Knister, 1994] Prakash, A. and Knister, M. J. (1994). A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330.
- [Rapino, 2004] Rapino, G. (2004). Versionierung in Kollaborativen Datenbanken-basierten Texteditiersystemen Implementierung im Projekt TeNDaX. Master's thesis, University of Zurich. Published in German.
- [Ressel and Gunzenhäuser, 1999] Ressel, M. and Gunzenhäuser, R. (1999). Reducing the problems of group undo. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139. ACM Press.
- [Ressel et al., 1996] Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, New York, NY, USA. ACM Press.
- [Spenske and Beilken, 1991] Spenske, M. and Beilken, C. (1991). An overview of GINA - the generic interactive application. In *UIMS: Proceedings of the workshop on user interface management systems and environments on User interface*

management and design, pages 273–293, New York, NY, USA. Springer-Verlag New York, Inc.

[Sun, 2000] Sun, C. (2000). Undo any operation at any time in group editors. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 191–200. ACM Press.

[Sun, 2002] Sun, C. (2002). Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361.

[Vitter, 1984] Vitter, J. S. (1984). US&R: A new framework for redoing (Extended Abstract). In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 168–176, New York, NY, USA. ACM Press.

[Yang, 1988] Yang, Y. (1988). Undo support models. *Int. J. Man-Mach. Stud.*, 28(5):457–481.