Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MICHAEL HAASE

MASTER THESIS

INTEGRATION UND ERWEITERUNG EINES VISUELLEN EDITORS IN SWEBLE HUB

Eingereicht am 1. Dezember 2016

Betreuer: Dipl.-Inf. Hannes Dohrn, Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 1. Dezember 2016

License

Erlangen, 1. Dezember 2016

Abstract

Sweble Hub is a collaboration platform which provides application-independent management and editing of wiki-content on the basis of the *Wiki Object Model* (*WOM*). Content editing has to be supported through the use of a platform-independent visual editing component within a web browser environment. The selection of a suitable basis for the visual editing component has been done in previous work through the evaluation of different approaches.

This master thesis presents the extension of the visual editing component to support the whole *Wiki Object Model*. Additionally the foundation is laid to assist a prospective extension towards a collaborative real-time editor.

For this, a literature research in the field of computer-supported collaborative work has been done to present and evaluate the current state of research and technology for the collaborative editing in groupware systems in order to select a suitable method.

Zusammenfassung

Sweble Hub ist eine Kollaborationsplattform, die eine anwendungsneutrale Verwaltung und Bearbeitung von Wiki-Inhalten auf Basis des *Wiki Object Models* (WOM) ermöglichen soll.

Die Bearbeitung von Inhalten mithilfe eines visuellen Editors soll dabei plattformunabhängig innerhalb eines Webbrowsers erfolgen können. Hierfür wurden bereits im Vorfeld verschiedene Basistechnologien evaluiert und dabei der *VisualEditor* der Wikimedia Foundation ausgewählt, sowie prototypisch umgesetzt.

Im Rahmen dieser Arbeit wird der visuelle Editor um die Unterstützung des gesamten *Wiki Object Models* erweitert. Außerdem wird das Fundament für eine spätere Erweiterung hin zu einem kollaborativen Editor gelegt.

Hierzu wird der aktuelle Stand der Forschung und Technik im Rahmen einer Literaturrecherche zu den verschiedenen Möglichkeiten der Bearbeitung von Inhalten innerhalb von Groupware Systemen untersucht und eine geeignete Methode ausgewählt.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Fragestellung . . . 2
1.2 Anforderungen . . . 2

2 Stand der Forschung **4**
2.1 Kollaborationsarten . . . 4
2.1.1 Asynchrone Groupware . . . 5
2.1.2 Multi-Synchrone Groupware . . . 5
2.1.3 Synchrone Groupware . . . 6
2.2 Verfahren zur Konsistenzerhaltung . . . 14
2.2.1 Differentielle Synchronisation . . . 14
2.2.2 Commutative Replicated Data Types . . . 17
2.2.3 Operational Transformation . . . 24
2.2.4 Evaluation . . . 45

3 Architektur und Design **47**
3.1 Stakeholder . . . 47
3.2 Randbedingungen . . . 48
3.2.1 Organisatorische Randbedingungen . . . 48
3.2.2 Technische Randbedingungen . . . 48
3.3 Basistechnologien . . . 49
3.3.1 Wiki Object Model . . . 49
3.3.2 Visueller Editor . . . 51
3.4 Softwarearchitektur . . . 55
3.4.1 Kontextabgrenzung . . . 55
3.4.2 Bausteinsicht . . . 57
3.4.3 Laufzeitsicht . . . 57
3.4.4 Verteilungssicht . . . 59
3.5 Ergebnisse . . . 60
3.5.1 Backend . . . 60
3.5.2 Frontend . . . 61

4 Implementierung **63**
4.1 Backend 63
4.1.1 Sweble Backend 63
4.1.2 Editor Backend 65
4.2 Frontend 67
4.2.1 React Frontend 67
4.2.2 Wom3VisualEditor 69

5 Evaluation **83**

Anhänge **86**
Anhang A Bill Of Materials 86
A.1 Editor Backend 86
A.2 Visual Editor 86
A.3 Editor Frontend 87
Anhang B Entwickler Dokumentation 89
B.1 Projektstruktur 89
B.2 VisualEditor Dokumentation 89
B.3 Aktualisierung des VisualEditors 92

Literaturverzeichnis **93**

1 Einführung

Mit *Sweble Hub* soll eine Kollaborationsplattform zur Verwaltung und Bearbeitung von Dokumenten auf Basis des *Wiki Object Models* (*WOM*) (Dohrn & Riehle, 2011) entstehen. *WOM* ermöglicht die anwendungsneutrale Interaktion mit Wiki-Inhalten, unabhängig ihrer Auszeichnungssprache, um so ein standardisiertes und maschinenlesbares Format zur Bearbeitung und zum Austausch von Dokumenten zu etablieren.

Die Bearbeitung eines *WOM* Dokuments erfolgt bisher rein textuell, in der jeweiligen Auszeichnungssprache der verwendeten Wiki-Software (bspw. Wikitext). Für die effektive Bearbeitung und auch im Hinblick auf Einstiegshürden, die es möglichst niedrig zu halten gilt, soll eine visuelle Editorkomponente zur Bearbeitung von *WOM* Dokumenten implementiert werden. Der Anwender soll dabei den gewohnten Komfort eines *WYSIWYG*[1] Editors geboten bekommen, der für den Großteil der Anwendungsfälle keinerlei Kenntnisse über das zugrunde liegende *WOM* erfordert.

In Haase (2014) wurden bereits die Anforderungen, die an einen visuellen Editor für das *WOM* gestellt werden, ausgearbeitet und verschiedene technologische Ansätze auf ihre Tauglichkeit hin evaluiert. Im Rahmen der Arbeit wurde der *Wikimedia VisualEditor*[2] als geeigneter Kandidat ausgewählt und die Bearbeitung von *WOM* Dokumenten prototypisch für einen Teil der im *WOM* spezifizierten Elemente umgesetzt.

[1] What You See Is What You Get
[2] https://www.mediawiki.org/wiki/VisualEditor

1.1 Fragestellung

Im Rahmen dieser Arbeit soll eine visuelle Editorkomponente implementiert werden, die die gesamte *WOM* Spezifikation unterstützt und zukünftig in die *Sweble Hub* Plattform integriert werden kann. Daher gilt es, die folgende Fragestellung zu beantworten:

> Mit welcher Architektur kann ein visueller Editor zur Bearbeitung von Dokumenten im *Wiki Object Model* auf Basis des *Wikimedia VisualEditors* bestmöglich in das *Sweble Hub* Web-Frontend integriert werden und darüber hinaus die Erweiterung von kontextabhängigen Menüs unterstützen?

1.2 Anforderungen

Die an den Editor gestellten Anforderungen werden im Folgenden aufgeführt, um auf Basis dieser in Kapitel 5 eine Evaluation der implementierten Editorkomponente vornehmen zu können.

VisualEditor
: Ein beliebiges *WOM* Dokument kann plattformübergreifend im Browser in einem visuellen Editor auf Basis des *Wikimedia VisualEditors* bearbeitet und anschließend verlustfrei wieder exportiert werden.

Erweiterung
: Ein Erweiterungsmechanismus für den visuellen Editor unterstützt das Hinzufügen kontextabhängiger Menüs an spezifische Elemente, um darauf Operationen ausführen zu können.

WOM
: Die gesamte *WOM* Spezifikation wird durch den visuellen Editor unterstützt.

Sync
: Durch den Benutzer durchgeführte Änderungen am Dokument werden an eine Serverkomponente übermittelt und somit synchronisiert, um den Bearbeitungszustand zu persistieren.

Zur Vorbereitung für eine zukünftige Erweiterung des Editors hin zur Unterstützung der gleichzeitigen Bearbeitung eines Dokuments durch mehrere Benutzer, soll darüber hinaus der aktuelle Stand der Forschung und Technik aus dem Bereich interaktiver Groupwaresysteme im Rahmen einer Literaturrecherche ausgearbeitet werden.

Die weitere Arbeit ist wie folgt strukturiert. In Kapitel 2 werden der Stand der Forschung aus dem Bereich der interaktiven Groupwaresysteme aufgezeigt und die infrage kommenden Algorithmen und Datenstrukturen vorgestellt. In Kapitel 3 wird zunächst die Architektur und das Design der visuellen Editorkomponente beschrieben, woraufhin in Kapitel 4 tiefer auf die Aspekte der Implementierung eingegangen wird. Abschließend wird die entwickelte Lösung in Kapitel 5 auf Basis der zuvor definierten Anforderungen evaluiert.

2 Stand der Forschung

Groupware Systeme ermöglichen die computergestützte Bearbeitung einer gemeinsamen Aufgabe durch mehrere Benutzer, wie etwa die Bearbeitung eines Textdokuments. Die Bearbeitung erfolgt dabei über eine bereitgestellte Schnittstelle, die unterschiedlichen Benutzern, auf verschiedenen Systemen verteilt, den Zugriff auf eine gemeinsame Arbeitsumgebung ermöglicht (C. A. Ellis & Gibbs, 1989).

Die Kommunikation verschiedener Parteien über ein Netzwerk zählt zu dem Forschungsbereich der verteilten Systeme. Dort vorhandene Algorithmen zum Datenmanagement und der Sicherstellung starker Konsistenz sind für die speziellen Anforderungen von Groupwaresystemen jedoch meist nur bedingt geeignet.

Das Datenmanagement, wie man es bspw. in klassischen Datenbanksystemen vorfindet, zielt meist darauf ab, Inkonsistenzen von vornherein zu vermeiden, indem etwa durch Locking der exklusive Zugriff auf ein Datenelement gewährleistet werden kann. Groupwaresysteme befassen sich dagegen häufig mit der nachträglichen Wiederherstellug der Konsistenz, falls durch verschiedene in Konflikt stehende Operationen Inkonsistenzen geschaffen wurden (Dourish, 1995).

2.1 Kollaborationsarten

Grundsätzlich lassen sich Systeme im Bereich der computergestützten Gruppenarbeit (CSCW) in drei Kategorien unterteilen: (1) asynchron, (2) multi-synchron und (3) synchron (Molli, Skaf-Molli, Oster & Jourdain, 2002).

2.1.1 Asynchrone Groupware

Asynchrone Bearbeitung erlaubt es einem Benutzer, die innerhalb einer Groupware vorhandenen Dokumente zu bearbeiten, indem diesem exklusiver Zugriff zugesprochen wird und nachfolgende Bearbeitungsanfragen abgewiesen werden, bis die Bearbeitung durch den Benutzer abgeschlossen und damit eine neue Dokumentenversion erzeugt wurde.

Aufgrund der gleichzeitigen Bearbeitung durch nur einen Benutzer kann die Konsistenz des in Bearbeitung befindlichen Dokuments zu jedem Zeitpunkt sichergestellt werden. Ziel dabei ist, einen einzelnen, globalen Aktivitätsstrom über den Zustandsraum zu definieren.

Dieser Ansatz entspricht in etwa der Bearbeitung von Dokumenten auf einem gemeinsamen Netzlaufwerk innerhalb einer klassischen Software zur Textverarbeitung, wie etwa LibreOffice. Dort wird beim Öffnen einer Datei ein Lockfile erzeugt, woraufhin anderen Benutzern beim gleichzeitigen Versuch die Datei zu öffnen ein Hinweis angezeigt wird, dass die Datei sich bereits in Bearbeitung befindet. Der finale Stand der Bearbeitung wird erst mit abschließender Speicherung auf das Netzlaufwerk zurück geschrieben und damit für andere Benutzer sichtbar.

2.1.2 Multi-Synchrone Groupware

Ebenso wie die im vorangegangenen Abschnitt beschriebene asynchrone Bearbeitung basieren multi-synchrone Systeme auf der exklusiven Bearbeitung eines Dokuments, jedoch mit dem Unterschied, dass mehreren Benutzern das Recht auf Bearbeitung zugesprochen werden kann.

Der jeweilige Bearbeitungsfortschritt wird dabei für jeden Benutzer separat vorgehalten und kann zu einem späteren Zeitpunkt mit dem aktuellen, globalen Zustand synchronisiert werden. Aufgetretene Konflikte müssen dabei jedoch meist manuell durch den Benutzer aufgelöst werden.

Die Bearbeitung erfolgt somit parallel in mehreren Aktivitätsströmen, innerhalb derer die einzelnen Benutzer nicht zwingend mit dem System verbunden sein müssen. Daher können Änderungen unabhängig voneinander vorgenommen und somit potentiell in Konflikt stehende Dokumentenzustände erzeugt werden. Um einen Gruppenfortschritt zu erzielen, werden die jeweiligen Änderungen nach Bedarf mit dem gerade aktuellen, globalen Zustand synchronisiert.

Diese Form der Kollaboration kommt unter anderem in verteilten Versionsverwaltungssystemen, wie etwa Git, zum Einsatz.

2.1.3 Synchrone Groupware

Synchrone (oder auch Echtzeit-) Groupware zeichnet sich dadurch aus, dass verschiedene Benutzer zeitgleich auf dem selben Dokument aktiv mit der Bearbeitung betraut sind und eine Bearbeitung durch jeden teilnehmenden Benutzer erfolgen kann. Die Änderungen der einzelnen Benutzer werden dabei möglichst latenzarm an die anderen propagiert, um die verschiedenen Sichten idealerweise synchron zu halten (C. A. Ellis & Gibbs, 1989).

Basierend auf den an die *Sweble Hub* Plattform gerichteten Anforderungen, werde ich mich im Rahmen dieser Arbeit im Folgenden nur auf den Bereich der synchronen Groupware konzentrieren.

Systemmodell

Ein verbreitetes Modell zur Beschreibung von synchronen Groupware Systemen wird in C. A. Ellis und Gibbs (1989) definiert.

Mit der Benutzung eines Groupware Systems wird eine Sitzung beliebiger Dauer gestartet, der eine Gruppe aktiver Benutzer, die Teilnehmer einer Sitzung, angehören. Die Sitzung stellt jedem Teilnehmer eine Schnittstelle zu einem gemeinsamen Kontext bereit, der den aktuellen Zustand widerspiegelt und mithilfe dessen Änderungen an dem gemeinsam in Bearbeitung befindlichen Dokument vorgenommen werden können.

Das Groupware-System besteht aus einer Menge von Teilnehmersystemen (Instanzen), die über ein Kommunikationsnetzwerk miteinander verbunden sind. Eine Instanz besteht aus einem Prozess, passivem Datenobjekt, auf dem die lokale Bearbeitung erfolgt, und eindeutigem Instanzidentifikator.

Der Prozess einer Instanz ist dafür zuständig, die Intentionen zur Bearbeitung des Dokuments durch den eigenen Teilnehmer als Bearbeitungsoperationen zu kapseln, an die anderen Teilnehmer der Sitzung zu übermitteln und sowohl lokal erzeugte, als auch von anderen Teilnehmern empfangene Operationen auf das eigene Datenobjekt anzuwenden, mit dem Ziel, die Datenobjekte aller Instanzen synchron zu halten.

Systemeigenschaften

Systeme zum kollaborativen Schreiben in Echtzeit[1], als Teilbereich der computergestützten Gruppenarbeit[2], zeichnen sich nach C. A. Ellis und Gibbs (1989) durch die folgenden Eigenschaften aus:

Hochgradig interaktiv
: Die Antwortzeit[3] sollte möglichst gering sein und mit der einer lokalen Anwendung vergleichbar sein

Echtzeit
: Die Benachrichtigungszeit[4] sollte vergleichbar mit der Antwortzeit sein, um den globalen Zustand möglichst konsistent zu halten und die durch Änderungen potentiell auftretenden Konflikte zu reduzieren

Verteilt
: Teilnehmer einer Sitzung können potentiell über das gesamte Internet verteilt sein, mit entsprechend unterschiedlichen Latenzzeiten und Leitungskapazitäten

Unbeständig
: Die Anzahl der Teilnehmer einer Sitzung kann sich beliebig ändern

Ad hoc
: Der durch die verschiedenen Teilnehmer entstehende Bearbeitungsverlauf kann nicht vorhergesagt werden

Fokussiert
: Es besteht eine hohe Wahrscheinlichkeit, dass Teilnehmer an ähnlichen Stellen innerhalb des Dokuments arbeiten

Externer Kanal
: Die Teilnehmer kommunizieren während der Bearbeitungstätigkeit häufig zusätzlich über externe Kanäle

Um den möglichst geringen Antwortzeiten gerecht zu werden, hält jeder Teilnehmer eine lokale Kopie des Dokuments vor. Auf dieser werden die eigenen Aktionen unverzüglich ausgeführt und sind daraufhin lokal direkt sichtbar. Anschließend wird die Aktion an die anderen Teilnehmer übermittelt, um die dadurch entstehenden Änderungen auf die jeweiligen lokalen Kopien des Dokuments anwenden zu können und den Zustand global zu synchronisieren.

[1] Real-time cooperative editing systems
[2] Computer Supported Cooperative Work (CSCW)
[3] Zeit bis die Aktion eines Teilnehmers in seiner Benutzerschnittstelle sichtbar wird
[4] Zeit bis eine Aktion für alle anderen Teilnehmer sichtbar wird

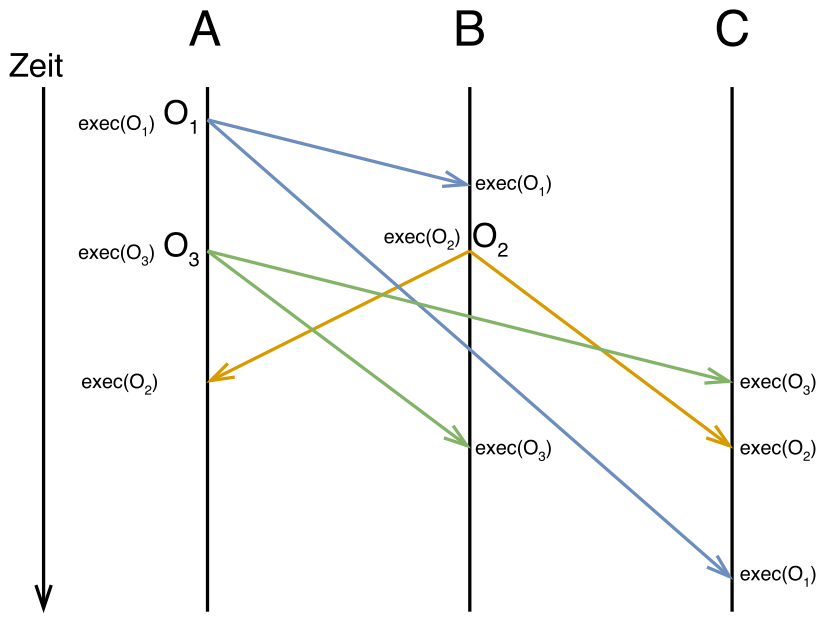

Abbildung 2.1: Beispielhaftes Szenario einer Sitzung mit drei Instanzen adaptiert von C. Sun, Yang, Zhang und Chen (1996)

Konsistenzmodell

Das Konsistenzmodell beschreibt die grundlegenden Probleme, die bei nebenläufiger Ausführung von Operationen auftreten können. Diese bilden die Grundlage für die Anforderungen, die an Verfahren zur Konsistenzerhaltung gestellt werden (Chengzheng Sun & Ellis, 1998).

Definition 1. *Kausalzusammenhang*

Gegeben sind zwei Operationen O_1 und O_2, erzeugt in den Instanzen i und j. O_2 ist kausal abhängig von O_1, bezeichnet mit $O_1 \rightarrow O_2$, genau dann, wenn:

(1) $i = j$ und O_2 nach O_1 erzeugt wurde

(2) $i \neq j$ und O_2 erzeugt wurde, nachdem O_1 in j ausgeführt wurde

Definition 2. *Nebenläufigkeit*

Die beiden Operationen O_1 und O_2 sind nebenläufig, bezeichnet mit $O_1 \parallel O_2$, genau dann, wenn weder $O_1 \rightarrow O_2$ noch $O_2 \rightarrow O_1$

Bei der Betrachtung von Nebenläufigkeitsszenarien wurden von C. Sun, Yang, Zhang und Chen (1996) drei fundamentale Inkonsistenzprobleme identifiziert.

Diese werden im Folgenden anhand des in Abbildung 2.1 dargestellten Szenarios erläutert, das eine beispielhafte Sitzung mit drei Teilnehmern bei der Bearbeitung eines Textdokuments darstellt.

Die beiden Operationen O_1 und O_3 werden in Instanz A generiert, Operation O_2 in Instanz B. Diese werden anschließend an die anderen Teilnehmer der Sitzung propagiert. Ohne Verfahren zur Nebenläufigkeitsbehandlung werden die Operationen nach Empfang direkt auf dem lokalen Datenobjekt zur Ausführung gebracht, was aufgrund von nichtdeterministischen Übertragungslatenzen zu den nachfolgend aufgeführten Problemen führen kann:

Divergenz
: Die Ausführungsreihenfolge der Operationen unterscheidet sich innerhalb jeder Instanz: $A : O_1, O_3, O_2$, $B : O_1, O_2, O_3$, $C : O_3, O_2, O_1$. Da die Operationen im Allgemeinen von dem zum Zeitpunkt der Generierung bestehenden Zustand abhängig, also nicht kommutativ sind, unterscheidet sich der Zustand nach Ausführung aller Operationen von Instanz zu Instanz.

  Angenommen O_1 fügt den Buchstaben `Y` an Position 1 ein, O_2 den Buchstaben `X` an Position 1 und O_3 entfernt den Buchstaben an Position 1, so ergeben sich nach Ausführung aller Operationen folgende inkonsistente Dokumentenzustände in den verschiedenen Instanzen: $A : X$, $B : Y$, $C : YX$

Kausalitätsverletzung
: Die Operation O_2 steht in kausalem Zusammenhang zu O_1, da sich diese auf den durch O_1 resultierenden Dokumentenzustand bezieht. Instanz C führt O_1 jedoch basierend auf dem durch O_2 geänderten Dokument aus und verletzt damit die Kausalitätsbeziehung.

Absichtsverletzung
: Die Absicht, die ein Teilnehmer mit einer Operation ausdrücken will, kann durch eine geänderte Ausführungsreihenfolge verloren gehen, da diese wiederum auf dem zum Zeitpunkt der Generierung der Operation vorhandenen Dokumentenzustand basiert.

  Angenommen A möchte mit O_3 die Absicht ausdrücken, den durch O_1 an Position 1 hinzugefügten Buchstaben wieder zu löschen. Diese Absicht kann bei B verletzt werden, wenn etwa O_2 ebenfalls an Position 1 einen neuen Buchstaben einfügt und dieser durch die Ausführung von O_3 wieder gelöscht wird. Damit entspricht das Ergebnis der Ausführung von Operation O_3 nicht mehr der ursprünglichen Absicht von A.

Hieraus lassen sich die drei folgenden Eigenschaften ableiten, die ein Groupware System zu jedem Zeitpunkt erfüllen muss, damit dieses als konsistent angesehen werden kann (C. Sun u. a., 1996):

Konvergenz
: Sobald alle Operationen innerhalb einer Sitzung verarbeitet wurden, sind die lokalen Datenobjekte aller Instanzen identisch

Kausalitätserhaltung
: Für alle Operationenpaare O_a und O_b gilt, sofern $O_a \rightarrow O_b$, muss O_a innerhalb aller Instanzen vor O_b ausgeführt werden

Absichtserhaltung
: Für alle Operationenpaare O_a und O_b gilt, sofern $O_a \parallel O_b$, muss die Ausführung von O_a und O_b in beliebiger Reihenfolge die Absicht der jeweils anderen Operation erhalten

Nachdem verschiedene Verfahren auf Basis dieses Konsistenzmodells entwickelt wurden, zeigte sich jedoch, dass sich einzelne Anwendungsfälle, die in Groupware Systemen vorzufinden sind, nicht beschreiben lassen. In X. Wang, Bu und Chen (2002) wurde daher ein erweitertes, dreischichtiges Konsistenzmodell vorgestellt (siehe Abbildung 2.2), das zusätzlich noch externe Faktoren der Gruppenarbeit berücksichtigt und das Modell dafür um eine semantische Komponente erweitert.

Abbildung 2.2: Erweiterung des Konsistenzmodells zu drei aufeinander aufbauenden Konsistenzebenen nach X. Wang, Bu und Chen (2002)

Operationenkonsistenz
: Entspricht der Konvergenz im vorhergehenden Konsistenzmodell

Inhaltskonsistenz
: Für jede Operation O sind die Effekte der Ausführung von O innerhalb aller Instanzen identisch mit der ursprünglichen Absicht von O

Semantische Konsistenz
: Für jede Operation O befolgen die Effekte der Ausführung von O innerhalb aller Instanzen die Regeln, die vor oder während der gemeinsamen Arbeit aufgestellt wurden (bspw. Grammatik)

Die bisher vorgestellten Konsistenzmodelle befassen sich hauptsächlich mit der technischen Seite der Konsistenz. Diese Fokussierung wird jedoch von Xue, Orgun und Zhang (2002) in Frage gestellt, da insbesondere in synchronen Groupware Systemen die menschlichen Benutzer und deren wechselseitige Interaktionen eine wichtige Komponente darstellen und bisher nicht in ausreichendem Maße in die Betrachtungen einbezogen wurden. So wird die Wandlung hin zu einem benutzerzentrierten Konsistenzmodell propagiert, in dem die Wahrung sowohl der individuellen Absichten einzelner Benutzer, als auch die durch Interaktion verschiedener Benutzer emergenten Absichten im Mittelpunkt stehen, erweitert um die Förderung von Konfliktlösungsmechanismen.

Nebenläufigkeitssteuerung

Konflikte bei der Bearbeitung eines Dokuments können nur durch nebenläufiges Bearbeiten entstehen, indem zwei Teilnehmer auf dem gleichen Stand eines Dokuments Operationen ausführen und die Änderungen aufgrund der Übertragungs- und Verarbeitungslatenz noch nicht auf dem jeweils anderen System verfügbar sind. Dies macht es erforderlich, sich mit der Behandlung von Problemen auseinanderzusetzen, die durch Nebenläufigkeit entstehen können. Die Verfahren dazu lassen sich grundsätzlich in zwei Kategorien aufteilen: pessimistisch und optimistisch (Nichols, Curtis, Dixon & Lamping, 1995)

Pessimistische Verfahren versuchen, eine global eindeutige Reihenfolge der Operationen sicherzustellen. Dies kann etwa mithilfe eines zentralen Koordinators oder eines verteilten Konsensusverfahrens (bspw. Paxos (Lamport, 1998)) realisiert werden. Insbesondere mit steigender Interaktionsrate durch eine Vielzahl von Teilnehmern tritt hierbei jedoch schnell ein Flaschenhals in Erscheinung, da für jede auszuführende Aktion zunächst eine Kommunikation stattfinden muss und sich damit die Antwortzeit verschlechtert.

Optimistische Verfahren gehen dagegen davon aus, dass nebenläufige Operationen im Allgemeinen nicht in Konflikt stehen. Tritt dennoch ein Konflikt auf, so wird versucht, diesen aufzulösen, indem die Operationen entsprechend angepasst werden, um die nebenläufig durchgeführten Änderungen zu kompensieren und damit den globalen Zustand zu synchronisieren. Verfahren dieser Art bieten sich insbesondere in Umgebungen mit hoher Übertragungslatenz, wie etwa dem Internet, an, da lokale Änderungen direkt ausgeführt werden können und die Antwortzeit des Systems damit nicht beeinflusst wird.

Für eine Webplattform wie *Sweble Hub*, die potentiell über das gesamte Internet erreichbar ist, sollten daher nur optimistische Verfahren zur Nebenläufigkeitssteuerung in Betracht gezogen werden.

Softwarearchitektur

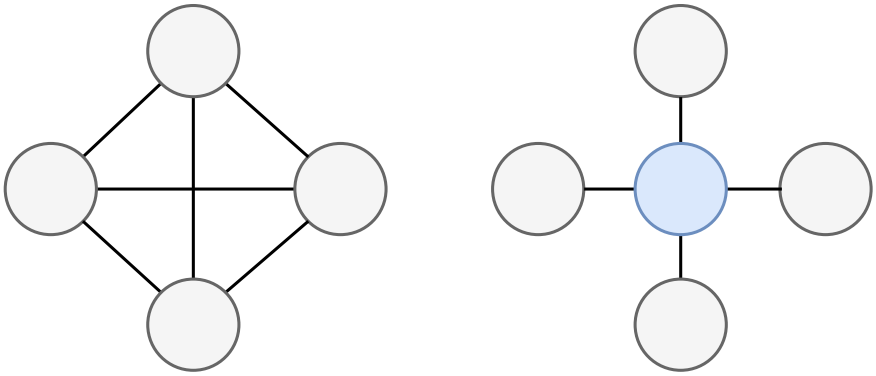

Abbildung 2.3: Schematische Darstellung der Kommunikationskanäle in einer Peer-To-Peer bzw. Client-Server Architektur

Bei der Topologie der Softwarearchitektur lassen sich grundsätzlich zwei Ansätze unterscheiden (siehe Abbildung 2.3). Zum einen eine Peer-To-Peer (P2P) Architektur, in der ein Teilnehmer einen direkten Kommunikationskanal zu jedem anderen Teilnehmer derselben Sitzung aufrecht erhält, um die eigenen Aktionen global bekannt zu machen. Zum Anderen kann eine zentrale Serverinstanz eingesetzt werden, die als Relay fungiert und die Nachrichten eines Teilnehmers entgegen nimmt, bei Bedarf verarbeitet und an die anderen Teilnehmer weiterleitet.

Eine P2P Architektur bietet den Vorteil, einerseits die durch das Bedienen einer potentiell großen Anzahl an Benutzern bereitzustellenden Serverressourcen zu reduzieren, damit Kosten zu senken und zum Anderen die Ausfallsicherheit zu erhöhen, da ein *Single Point of Failure* beseitigt wird. Diesem können jedoch auch praktische Probleme gegenüber stehen, wenn das System etwa nicht nur lokal, sondern im Internet eingesetzt wird, mit entsprechend hohen Kommunikationslatenzen (Weiss, Urso & Molli, 2009).

Mit einer Client-Server Architektur gibt es dagegen eine zentrale Instanz, die zu jedem Zeitpunkt den globalen Zustand des in Bearbeitung befindlichen Dokuments bereit hält, auch wenn alle Teilnehmer die aktive Sitzung verlassen haben. Darüber hinaus erlaubt das Client-Server Modell auch eine Vielzahl von Annahmen, die zur Reduzierung der Komplexität in den in nachfolgenden Kapiteln behandelten Algorithmen beitragen. So ist bspw. die Gesamtzahl der an einer Sitzung beteiligten Teilnehmer für den einzelnen irrelevant, da aus seiner Sicht die durch die anderen Teilnehmer durchgeführten Aktionen ausschließlich auf Serverseite stattfinden (Nichols u. a., 1995).

Sweble Hub verfügt bereits über eine zentralisierte Client-Server-Architektur. Somit bietet es sich an, die Groupware Komponente in gleicher Weise zu realisieren, weshalb der P2P Ansatz im Folgenden nicht weiter betrachtet wird.

2.2 Verfahren zur Konsistenzerhaltung

Um die Konsistenz verteilter Dokumente zu erreichen, stehen verschiedene Verfahren zur Verfügung, die je nach Anwendungsfall unterschiedliche Vorteile bieten. Im Folgenden werden daher die drei Verfahren *Differentielle Synchronisation* (Kapitel 2.2.1), *Commutative Replicated Data Types* (Kapitel 2.2.2) und *Operational Transformation* (Kapitel 2.2.3) vorgestellt, jeweils der aktuelle Stand der Forschung beleuchtet und im Anschluss daran evaluiert (Kapitel 2.2.4), um daraus ein Verfahren auswählen zu können, das zukünftig für die Editorkomponente von *Sweble Hub* am geeignetsten erscheint.

2.2.1 Differentielle Synchronisation

Mit differentieller Synchronisation stellt Fraser (2009a) ein vergleichsweise einfaches Verfahren zur Synchronisierung von Dokumenten vor. Dieses basiert auf der klassischen *diff-patch* Methode und erweitert diese, um sie für das kontinuierliche Synchronisieren von Dokumenten einsetzen zu können.

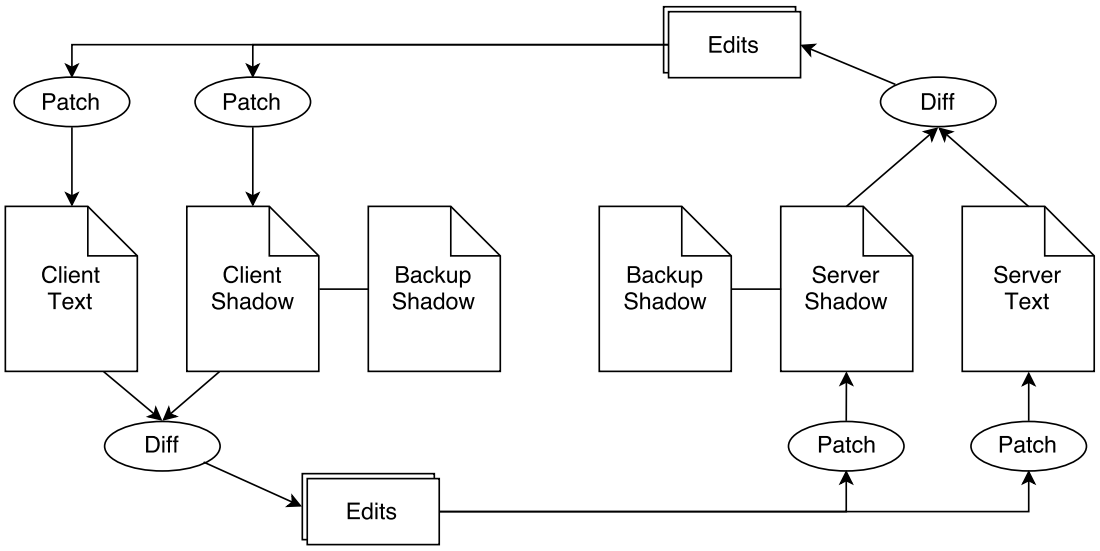

Abbildung 2.4: Differentielle Synchronisation mit einer Client-Server Architektur nach Fraser (2009a)

Der in Abbildung 2.4 skizzierte Algorithmus ist ein ständiger Zyklus von Diff- und Patch-Operationen, der die lokalen Änderungen, die seit der letzten Iteration vollzogen wurden, extrahiert und anschließend auf die entfernten Dokumente anwendet, um diese zu synchronisieren.

Zu Beginn enthalten alle Dokumentenspeicher (*Client Text*, *Client Shadow*, *Server Text*, *Server Shadow* und *Backup Shadow*) den identischen Inhalt. Führt ein Benutzer Änderungen auf Clientseite durch, so wird der Inhalt von *Client Text* entsprechend verändert.

In regelmäßigen Abständen wird der Zustand von *Client Text* und *Client Shadow* verglichen und dabei die seit der letzten Iteration vorgenommenen Änderungen extrahiert. Anschließend wird *Client Shadow* durch den gerade verglichenen Inhalt von *Client Text* ersetzt und mit einer aufsteigenden und eindeutigen Versionsnummer versehen.

Bei der Übertragung werden die Änderungen mitsamt ihren zugewiesenen Versionsnummer zum Server geschickt. Zusätzlich enthält diese Anfrage die Versionsnummer des zuletzt vom Client erhaltenen *Server Shadow* Dokuments, was dessen Erhalt bestätigen soll.

Die vom Client empfangenen Änderungen werden auf Serverseite zunächst auf die *Server Shadow* angewendet und die aktuelle Versionsnummer der *Client Shadow* gespeichert. Zusätzlich wird die *Backup Shadow* durch die gerade aktualisierte *Server Shadow* ersetzt und die aktuelle Versionsnummer der *Server Shadow* festgehalten.

Zur Behandlung von Verbindungsabbrüchen wird ein Stapel von Änderungs-Operationen vorgehalten, deren Erhalt durch die jeweilige Gegenseite mithilfe der Versionsnummern bestätigt werden muss. Im Falle von divergierten Dokumentenzuständen, bedingt etwa durch Paketverlust, kann das in *Backup Shadow* vorgehaltene Dokument dazu verwendet werden, die Operationen, basierend auf der vorhergehenden Dokumentenversion, erneut einzuspielen (Fraser, 2009b).

Nun werden die empfangenen Änderungen auf den *Server Text* angewendet.

Beide *Shadow* Dokumente sind i.d.R. identisch, weshalb sich die Änderungen mit einem Standard Patchverfahren anwenden lassen. Für das Patchen von *Client-* bzw. *Server Text* wird dagegen ein *fuzzy patch* Algorithmus benötigt, der mit *best effort* versucht die, Änderungen auf ein potentiell geändertes Dokument anzuwenden. Dieser Versuch kann fehlschlagen, wenn etwa nicht genügend Kontextinformationen verfügbar sind, weil die beiden Dokumentenzustände zu sehr divergiert sind. In diesem Fall werden die vorgenommenen Änderungen beim Vergleich von *Server Shadow* und *-Text* und den daraus generierten Diffs auf Clientseite wieder rückgängig gemacht.

Damit wird sichergestellt, dass *Server-* und *Client-Text* konsistent bleiben, auch wenn dies das Rückgängigmachen einzelner Änderungen zur Folge hat. Bei entsprechend niedrig gewählten Iterationszeiten halten sich die Auswirkungen, je nach Anwendungsfall, jedoch in Grenzen.

Das Verfahren zeichnet sich dadurch aus, dass dieses auf Client- wie auch Server-

seite identisch ist. Für den Fall, dass die Anfragen nur vom Client aus initiiert werden, wie etwa bei klassischen HTTP Anfragen, so erweist sich die clientseitige *Backup Shadow* als obsolet.

Der zuvor dargestellte Ablauf behandelt die Kommunikation zwischen Server und einem Client. Zur Erweiterung auf mehrere Clients kann das identische Verfahren angewendet werden. Hierbei wird, wie in Abbildung 2.5 dargestellt, für jeden Client ein eigener Kontext (*Client Text*, *Client Shadow* und *Server Shadow*) vorgehalten und der *Server Text* als gemeinsame Ressource zum Austausch der Daten verwendet.

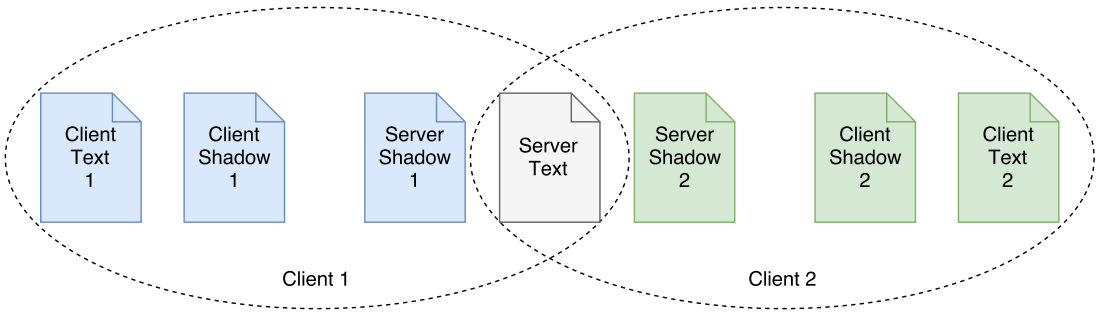

Abbildung 2.5: Kontexte bei der Behandlung mehrerer Teilnehmer adaptiert von Fraser (2009a)

Grundsätzlich lassen sich mit differentieller Synchronisation Dokumente mit beliebigen Inhaltsformaten synchronisieren. Die einzige Voraussetzung dafür ist ein geeigneter fuzzy diff/patch Algorithmus, welcher in der Lage ist, Änderungen auch dann anzuwenden, wenn das zu patchende Zieldokument sich von dem Quelldokument, das als Basis des Diffs verwendet wurde, unterscheidet.

Je nach Diff-Algorithmus lassen sich mit inhaltsbasierten Verfahren wie diesem die Absicht eines Benutzers besser festhalten, als wenn jede vom Benutzer getätigte Operation isoliert betrachtet wird. Daher bietet es sich an, nicht den minimalen Unterschied zu berechnen, sondern auf semantischer Ebene zu arbeiten, also etwa dem Ersetzen ganzer Wörter, im Vergleich zu einzelnen Buchstaben (Fraser, 2009b).

Anwendung findet differentielle Synchronisierung etwa in *CoRED*, einem browserbasierten Java Editor (Lautamäki u. a., 2012). Hierbei wird zur Feststellung der Unterschiede Myers Algorithmus (Myers, 1986) eingesetzt und mit dem Bitap Verfahren (Wu & Manber, 1992) ein Patch erzeugt. Darüber hinaus wurde dieser Ansatz ehemals in *Google Docs* verfolgt, bis er 2009 durch *Operational Transformation* (siehe Kapitel 2.2.3) ersetzt wurde[5].

[5]https://www.heise.de/developer/artikel/Synchronisationsalgorithmen-verstehen-1562877.html?artikelseite=3

2.2.2 Commutative Replicated Data Types

Ein weiterer optimistischer Ansatz, also unter Annahme von *Eventual Consistency*, sind Verfahren aus dem Bereich der *Commutative Replicated Data Types* (CRDT). Die Behandlung von nebenläufigen Operationen unter Berücksichtigung des Konsistenzmodells gestaltet sich im Allgemeinen als schwieriges Unterfangen, was durch Sicherstellung der Benutzerabsichten noch verschärft wird. Insbesondere in mobilen Szenarien kommt dazu außerdem das Problem von potentiellen Unterbrechungen der Kommunikationsverbindungen, was den Synchronisierungsprozess noch aufwändiger gestaltet.

Das Hauptproblem liegt im Sicherstellen des Kausalzusammenhangs der einzelnen Operationen, da eine im falschen Kontext angewendete Operation im Allgemeinen eine Divergenz in den Teilnehmerdokumenten zur Folge hat. CRDTs nehmen sich dieses Problems an, indem sie eine Klasse von Datentypen bilden, auf denen nebenläufige Operationen in beliebiger Reihenfolge angewendet werden können und dennoch garantiert ist, dass nach Anwendung aller Operationen alle verteilten Kopien sich in dem selben konsistenten Zustand befinden (N. Preguiça, Shapiro & Legatheaux Martins, 2008).

Neben Datentypen zur Bearbeitung von linearen Strukturen (bspw. Texte), beschreiben Shapiro, Preguiça, Baquero und Zawirski (2011) weitere Datentypen, wie etwa Zähler, Register, Mengen und Graphen. Zur Veranschaulichung werden im Folgenden beispielhaft die Funktionsweise eines einfachen Zählers und LWW-Registers[6] aufgezeigt.

Im Vergleich zu indexbasierten Verfahren, wie etwa *Operational Transformation* (siehe Kapitel 2.2.3), wird hierbei jedem Atom (je nach Granularität ein Paragraph, Satz, Wort oder Zeichen) ein eindeutiger Bezeichner zugewiesen.

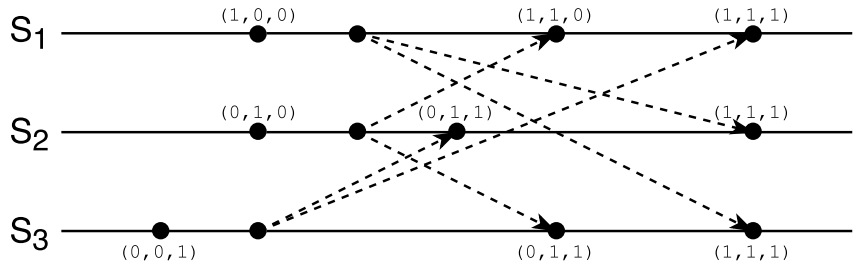

Abbildung 2.6: Beispielhafte Sitzung eines G-Zählers adaptiert Shapiro, Preguiça, Baquero und Zawirski (2011)

[6] Last-Writer-Wins Register

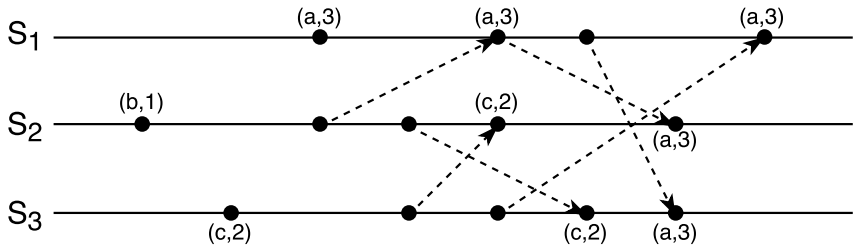

Abbildung 2.7: Beispielhafte Sitzung eines Last-Writer-Wins Registers adaptiert Shapiro, Preguiça, Baquero und Zawirski (2011). Datenpakete enthalten den geschriebenen Wert, sowie den zugehörigen Zeitstempel

G-Zähler

Ein aufsteigender Zähler kann mit einer Datenstruktur ähnlich zu Vektoruhren[7] realisiert werden. Jeder Teilnehmer besitzt einen, mit 0 initialisierten, Eintrag im Vektor. Beim Inkrementieren des Zählers wird der eigene Eintrag um 1 erhöht und der daraus resultierende Vektor an die anderen Teilnehmer übermittelt (siehe Abbildung 2.6).

Bei Empfang eines Vektors von einem anderen Teilnehmer wird dieser mit dem lokalen Vektor verglichen, indem jeder Eintrag mit dem korrespondieren lokalen Eintrag verglichen wird und der Eintrag im lokalen Vektor mit dem Maximum der beiden Werte aktualisiert wird. Der aktuelle Wert des Zählers ist die Summe aller seiner Einträge.

Zustandsbasierter PN-Zähler

Für einen Zähler, der zusätzlich auch Subtraktion unterstützen soll, können zwei *G-Zähler* in Kombination eingesetzt werden. Jeweils zur getrennten Behandlung aller Additions- und Subtraktionsoperationen. Den aktuellen Wert der Zähler stellt die Differenz der Werte der beiden *G-Zähler* dar.

LWW-Register

Ein *Last-Writer-Wins Register* hält den Wert der letzten Schreiboperation vor, wie in Abbildung 2.7 dargestellt.

Dazu wird beim Schreiben eines neuen Wertes ein eindeutiger Zeitstempel übermittelt, wie etwa die Konkatenation der aktuellen Zeit mit dem eindeutigen Instanzidentifikator. Jede Instanz hält den zuletzt erhaltenen Wert sowie den zugehörigen Zeitstempel vor und aktualisiert das Wertepaar bei Erhalt einer Schreiboperation, sofern der Vergleich der Zeitstempel ergibt, dass das empfangene Ereignis neuer ist.

[7] vector clocks

Zur Behandlung linearer Strukturen stellten erstmals Gérald Oster, Urso, Molli und Imine (2005) das *WOOT*[8] Framework vor, mit dem Ziel, das als zu komplex angesehene *Operational Transformation* Verfahren abzulösen.

Bei indexbasierten Verfahren, wie etwa *OT*, enthält eine Einfügeoperation die Position als Index, was eine erneute Berechnung der Ausführungsreihenfolge beim Empfang zur Folge hat. Im Gegensatz dazu wird bei *WOOT* der Kontext des einzufügenden Elements mitgegeben, welcher bei Erzeugung der Operation direkt verfügbar ist.

Das in Kapitel 2.1.3 vorgestellte Konsistenzmodell wird hierbei modifiziert, sodass ein Groupwaresystem innerhalb des *WOOT* Frameworks als konsistent anzusehen ist, genau dann, wenn Konsistenz, Absichtserhaltung sowie die Einhaltung der Vorbedingungen aller Operationen gegeben sind. Eine Operation darf also nur angewendet werden, wenn ihre Vorbedingung erfüllt ist. Die ursprünglich vorhandene Kausalitätserhaltung ist nicht erforderlich.

Das Datenmodell sieht bei jeder Instanz eine logische Uhr sowie eine Menge noch nicht angewendeter Operationen vor. Das Datenobjekt besteht aus einer Folge von Elementen. Ein Element wiederum ist ein Tupel $< id, v, \alpha, id_p, id_n >$, bestehend aus einem eindeutigen Identifikator id, einem Sichtbarkeitsflag v, seinem Inhalt α, sowie Referenzen in Form von Identifikatoren auf das jeweils vorhergehende id_p bzw. nachfolgende Element id_n innerhalb der Folge.

Der Elementidentifikator besteht im Falle von *WOOT* aus einem Tupel aus Instanzidentifikator und einer lokal eindeutigen Nummer, die mithilfe der logischen Uhr erzeugt wird.

Zum Bearbeiten des Datenobjekts stehen zwei Operationen zur Verfügung, die jeweils ein Element e als Parameter erwarten, das eine Referenz auf ein vorhergehendes Element p sowie ein nachfolgendes Element n enthält.

insert(e)
: Fügt das Element e zwischen p und n ein, mit der Vorbedingung, dass sowohl p als auch n im Datenobjekt existieren

delete(e)
: Entfernt das Element e, indem dessen Sichtbarkeitsflag auf `false` gesetzt wird, mit der Vorbedingung, dass e im Datenobjekt existiert

In Abbildung 2.8 ist eine beispielhafte Sitzung mit zwei Instanzen dargestellt. Zu Beginn beinhaltet das zu bearbeitende Dokument die Zeichenkette *abc*. Im weiteren Verlauf fügt S_1 zunächst den Buchstaben x zwischen a und b ein und löscht anschließend "c". Nach Ausführung aller Operationen auf beiden Instanzen befinden sich beide Dokumente konsistent im Zustand *axb*.

[8] WithOut Operational Transformation

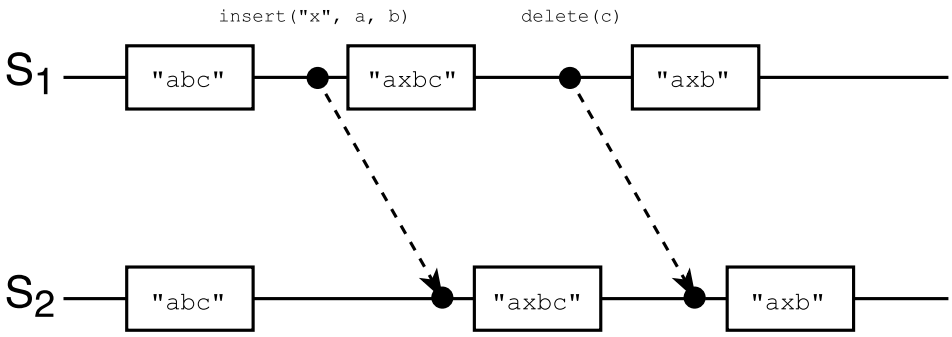

Abbildung 2.8: WOOT Sitzung ohne nebenläufige Operationen adaptiert von Gérald Oster, Urso, Molli und Imine (2005)

Bei der Behandlung von nebenläufigen Operationen kann es aufgrund der vorhandenen Halbordnung der Operationen und der daraus auf den verschiedenen Instanzen zu erzeugende Totalordnung zu unterschiedlichen Ergebnissen kommen, da diese mehrere korrekte Lösungen haben kann, wenn die Operationen in unterschiedlicher Reihenfolge empfangen werden. Es muss also sichergestellt werden, dass das Konvergenzkriterium erfüllt wird.

Dies zeigt sich bspw. in dem in Abbildung 2.9 dargestellten Szenario, in dem sowohl S_1 als auch S_2, ausgehend vom ursprünglichen Dokumentenzustand, einen Buchstaben an der Position zwischen a und b einfügen. Bis die Operation an die jeweils andere Instanz übermittelt und dort ausgeführt wurde, wurde die lokal erzeugte Operation bereits auf das eigene Datenobjekt angewendet. Damit ergeben sich die beiden Totalordnungen *axybc* und *ayxbc* als mögliche Lösungen, was es erforderlich macht, sicherzustellen, dass alle Instanzen in einen identischen Endzustand konvergieren. Ein entsprechendes Verfahren wird in Gérald Oster u. a. (2005) erläutert.

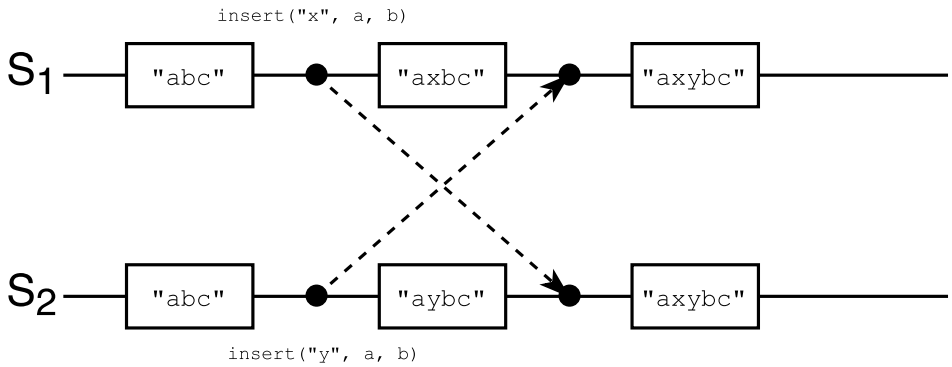

Abbildung 2.9: WOOT Sitzung mit nebenläufigen Operationen adaptiert von Gérald Oster, Urso, Molli und Imine (2005)

Die einzelnen Datenelemente beanspruchen durch die darin notwendigerweise enthaltenen Metadaten viel Speicherplatz, insbesondere wenn ein Element nur ein einzelnes Zeichen darstellt. Zusätzlich aktualisiert die Löschoperation nur das Sichtbarkeitsflag eines Elements, da ein zuvor eingefügtes Element nicht mehr aus dem Dokument entfernt werden darf, was ein potentiell unbegrenztes Wachstum des Dokuments zur Folge hat.

Mit *treedoc* stellen Shapiro und Preguiça (2007) daher ein weiteres Framework vor, dass sich dieser Probleme annimmt. Hierbei wird für die Generierung der eindeutigen Elementidentifikatoren ein Binärbaum (siehe Abbildung 2.10) eingesetzt, womit auch das Ermitteln der korrekten Position einer Einfügeoperation erleichtert wird.

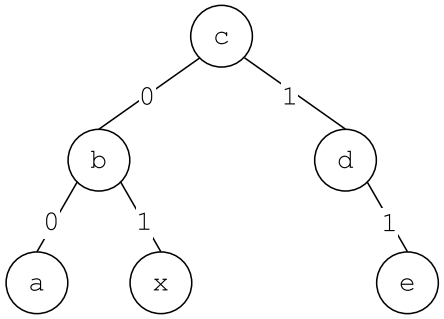

Abbildung 2.10: Ein Binärbaum im *treedoc* Framework für das Dokument *abxcde*. $id(a) = 00$, $id(x) = 01$, $id(b) = 0$, ... adaptiert von Shapiro und Preguiça (2007)

Zur Erzeugung eines Elementidentifikators für das zwischen p und n einzufügende Element e, muss die folgende Bedingung erfüllt sein: $id(p) < id(e) < id(n)$. Dazu wird in den in Infix-Darstellung ausgewerteten Baum an entsprechender Stelle ein neuer Knoten für e erzeugt. Der Identifikator entspricht dabei der Konkatenation der Kantenwerte, die bei der Traversierung vom Wurzelknoten bis zum neu eingefügten Knoten besucht werden.

Zur Anwendung einer Löschoperation wird der entsprechende Knoten im Baum entfernt und durch ein *Null-Objekt* ersetzt.

Bei der Behandlung von nebenläufigen Einfügeoperationen an der gleichen Position kann es in diesem Modell weiterhin zu Konflikten kommen, da der Identifikator beider Operationen identisch und damit nicht mehr eindeutig wäre. Daher wird der Binärbaum um sog. *Mini-Nodes* erweitert, die Teil eines Knotens im Baum sind und darin wiederum einen eindeutigen Identifikator besitzen, wie in Abbildung 2.11 dargestellt. Für diesen Zweck findet in *treedoc* bspw. der Instanzidentifikator Anwendung.

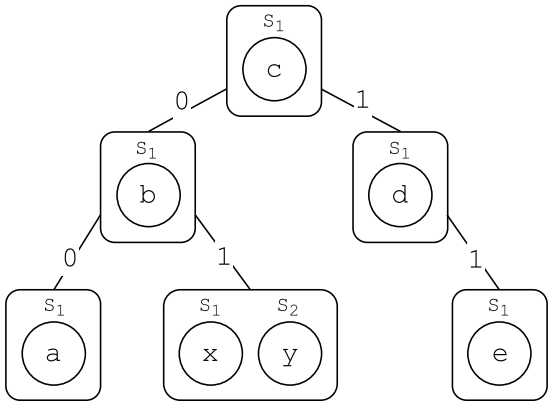

Abbildung 2.11: Ein Binärbaum im *treedoc* Framework mit mehreren *Mini-Nodes* unterhalb eines Knotens durch nebenläufiges Einfügen von x und y adaptiert von Shapiro und Preguiça (2007)

Weiterhin wird ein Verfahren erläutert, womit der Baum bereinigt, also von *Null-Objekten* befreit und neu balanciert wird. Dies wird durch Auslesen und erneuter Generierung des gesamten Baumes bewerkstelligt. Dieser Vorgang unterstützt jedoch keine nebenläufigen Operationen und muss auf allen Instanzen ausgeführt werden, da sich im Zuge der erneuten Generierung die Elementidentifikatoren ändern und sich zuvor erstellte Operationen nicht mehr auf die richtigen Elemente beziehen würden.

Mit *Logoot* (Weiss u. a., 2009) steht ein weiteres, auf *CRDT* basierendes Framework zur Verfügung, das außerdem das Rückgängigmachen von bereits angewendeten Operationen unterstützt (Urso, Molli & Weiss, 2009). Im Gegensatz zu den zuvor behandelten Verfahren, ist *Logoot* zeilenbasiert, betrachtet also Zeilen als atomare Datenelemente. Jede Zeile $\langle pid, content \rangle$ besitzt einen eindeutigen Identifikator *pid* zur Bestimmung der korrekten Position im Dokument, sowie ihren Inhalt *content*. Dabei ist *pid* ein Paar aus Position *pos*, sowie dem Wert der lokalen Vektoruhr zum Zeitpunkt der Erstellung der Operation. *pos* ist eine Liste aus Identifikatoren, bestehend aus der Zeilennummer, sowie dem Instanzidentifikator *site*. Zur Veranschaulichung ist in Abbildung 2.1 ein exemplarisches *Logoot* Dokument aufgezeigt.

```
1 ⟨⟨0, 0⟩, nil, l_s⟩
2 ⟨⟨1, 1⟩, 0, "First line by site 1" ⟩
3 ⟨⟨1, 5⟩, 0, "First line by site 5" ⟩
4 ⟨⟨2, 1⟩, 1, "Second line by site 1" ⟩
5 ⟨⟨∞, 0⟩, nil, l_e⟩
```

Listing 2.1: Interne Darstellung eines *Logoot* Dokuments adaptiert von Weiss u. a. (2009)

Einen gänzlich anderen Ansatz für die Behandlung verteilter, linearer Strukturen stellen Roh, Jeon, Kim und Lee (2011) mit *Replicated Growable Arrays* (*RGA*) vor. Dies stellt eine verkettete Liste dar, die von einer beliebigen Anzahl an Teilnehmern bearbeitet werden kann. Dazu besteht die in Abbildung 2.12 dargestellte, interne Datenstruktur aus verketteten Elementen, die eine verkettete Liste bilden, sowie einer Hash-Map, die einen direkten Elementzugriff über die Elementidentifikatoren ermöglicht.

Zur nachträglichen Sicherstellung der kausalen Ordnung von Operationen werden Vektoruhren eingesetzt, deren aktueller Wert in jeder Operation enthalten ist. Bei der Verarbeitung einer Operation wird zur effizienteren Bestimmung der kausalen Ordnung von Operationenfolgen jedoch eine aus der Vektoruhr transformierte Form verwendet, ein sog. *S4Vector* $\langle ssn, sid, sum, seq \rangle$. Dieser besteht aus einer global aufsteigenden Sitzungsnummer ssn, dem Instanzidentifikator sid, der Summe der Einträge der Vektoruhr sum, sowie dem der Instanz zugeordneten Wert innerhalb der Vektoruhr seq.

Ein Element besteht aus einem zur Identifikation verwendeten *S4Vector*, seinem Inhalt, einem Zeiger auf das nächste Element in der Liste, sowie einem Zeiger auf ein potentiell vorhandenes weiteres Element innerhalb der Hash-Map, das auf dem gleichen Schlüssel abgebildet wurde. Zusätzlich existiert ein Zeiger *head* auf das erste Element der Liste.

Beim Löschen eines Eintrags wird dessen zugehöriges Element nicht aus der Datenstruktur entfernt, sondern durch einen *Tombstone* ersetzt, indem die Referenz zu seinem Inhalt entfernt wird. Dadurch kann für jede Operation die korrekte Position ermittelt werden, selbst wenn das Element nebenläufig entfernt wurde.

Beim Verarbeiten einer Einfügeoperation wird zunächst ein neues Element n erstellt, der zugehörige *S4Vector* generiert und das in der Operation referenzierte Vorgängerelement p aus der Hash-Map gelesen. Nun werden die Identifikatoren der Nachfolger von p mit dem von n verglichen, bis die korrekte Einfügeposition gefunden wurde, n in die Hash-Map sowie an der ermittelten Position in die verkettete Liste eingefügt.

Auch *RGAs* sehen einzelne Zeichen als atomare Inhalte der Datenelemente an, die verfügbaren Operationen sind also zeichenbasiert. Unabhängig voneinander erweitern Lv, He, Cai und Cheng (2016) mit *RGA Supporting String* (*RGASS*) und Briot, Urso und Shapiro (2016) mit *RGATreeSplit* das *RGA* Konzept dahin, auch zeichenketten- bzw. blockbasierte Operationen zu unterstützen, mit dem Ziel, die Absichten der Benutzer besser widerspiegeln zu können und nebenbei die Effizienz zu verbessern.

Das vor dem Erscheinen von *CRDTs* dominierende Verfahren *OT* wurde in Ahmed-Nacer, Ignat, Oster, Roh und Urso (2011) mit *CRDTs* verglichen und

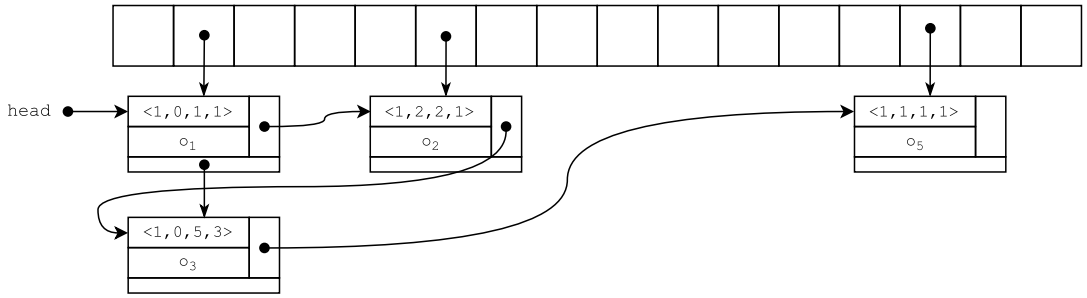

Abbildung 2.12: Beispielhafte Darstellung der internen Datenstruktur eines *Replicated Growable Arrays* adaptiert von Roh, Jeon, Kim und Lee (2011)

auf die Anwendbarkeit für synchrone Groupware Systeme hin untersucht, indem Bearbeitungsverläufe unter anderem von Wikipedia Artikeln extrahiert und mithilfe der Verfahren reproduziert wurden. Hierbei zeigte *RGA* die beste Gesamtleistung, auch im Vergleich zu Verfahren aus dem Bereich *Operational Transformation*.

Den bisher vorgestellten Konzepten lagen linear strukturierte Dokumente zugrunde, wie etwa Zeichenketten. Für die Behandlung semistrukturierter Inhalte in Form von XML Dokumenten schlagen Martin, Urso und Weiss (2010) eine Kombination der im Vorfeld erläuterten *CRDTs* vor. Dazu bedarf es der Lösung zur Darstellung von Kindern sowie Attributen eines Elements. Da die Reihenfolge der Kindelemente von Bedeutung ist, findet hierfür erneut ein Verfahren zur Behandlung linearer Strukturen Anwendung, wie etwa *WOOT*. Für die einzelnen Attribute eines Elements werden zusätzlich *Last-Writer-Wins Register* eingesetzt.

2.2.3 Operational Transformation

Mit dem in C. A. Ellis und Gibbs (1989) vorgestellten *Operational Transformation* (OT) steht ein weiteres Verfahren zur Konsistenzerhaltung zur Verfügung, das seither einer stetigen Weiterentwicklung unterliegt. OT findet aktuell eine breite Anwendung in kommerziellen, sowie Open-Source Echtzeit-Kollaborationslösungen, wie etwa *Google Docs*[9] oder *Etherpad*[10].

Im Verlauf dieses Kapitels wird zunächst die grundlegende Funktionalität von OT erläutert, um im Anschluss daran die bedeutendsten Weiterentwicklungen des ursprünglichen Konzepts zu verfolgen und den aktuellen Stand der Forschung aufzuzeigen.

Ein *Operational Transformation* Algorithmus besteht aus zwei Komponenten.

[9] https://docs.google.com/
[10] https://github.com/ether/etherpad-lite

Einerseits Transformationsfunktionen zur Transformation zweier gegebener nebenläufiger Operationen und andererseits einem Kontrollalgorithmus. Dieser hat die Aufgabe, lokale Operationen auszuführen, Operationen an entfernte Instanzen zu verteilen und von diesen entgegen zu nehmen, sowie die Auswahl erforderlicher Transformationsfunktionen zur Anwendung auf empfangenen Operationen zu treffen (Randolph, Boucheneb, Imine & Quintero, 2015).

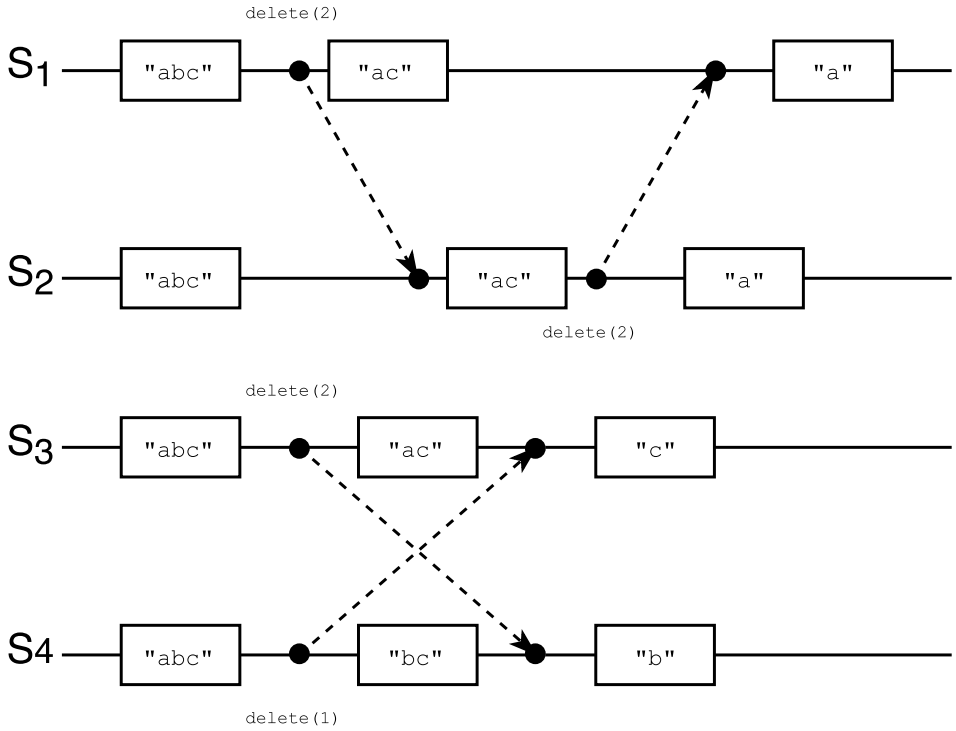

Abbildung 2.13: Anwendung von entfernt erzeugten Operationen ohne *Operational Transformation*

Das Grundkonzept zeichnet sich dadurch aus, dass die vom Benutzer erzeugten Operationen lokal direkt auf das sich in Bearbeitung befindliche Dokument angewendet und zusätzlich die Operationen, anstelle des geänderten Zustands, an die anderen Instanzen übermittelt werden. Beim Empfang einer Operation von einer anderen Instanz wird diese zunächst transformiert, sodass die Operation, angewendet auf dem gerade aktuellen lokalen Dokument, das unter Umständen durch nebenläufige Operationen verändert wurde, die ursprüngliche Intention des entfernten Benutzers widerspiegelt. Operationen beziehen sich im Allgemeinen auf Zeichenindizes. Um das dritte Zeichen innerhalb eines Dokuments zu entfernen muss also die Operation *delete*(3) angewendet werden.

Für die Ausführung von Operationen mit kausalem Zusammenhang bedarf es keiner Transformation der Operationen, wie Abbildung 2.13 zu entnehmen ist.

Die Dokumente von S_1 und S_2 befinden sich nach Ausführung der beiden Operationen im selben Endzustand. Demgegenüber unterscheiden sich die Zustände der Dokumente von S_3 und S_4, da die nebenläufig erzeugten Operationen direkt auf das Dokument angewendet wurden, obwohl diese in Konflikt zueinander stehen.

Der ursprüngliche *dOPT*[11]-Algorithmus (C. A. Ellis & Gibbs, 1989) ist *Peer-to-Peer*-basiert, mit einer beliebigen, jedoch festen Anzahl an Instanzen. Jede Instanz besitzt eine Vektoruhr s_i, deren Einträge die Anzahl der erzeugten Operationen der jeweiligen Instanzen beinhaltet und den Operationen beigefügt wird, um den Kausalitätszusammehang daraus ableiten zu können, sowie ein Protokoll L_i aller bereits angewendeten Operationen (*HB*[12]).

Vor der Anwendung einer Operation o wird deren Vektoruhr s_o mit der Vektoruhr der Instanz s_i verglichen. Sofern diese identisch sind, kann die Operation direkt angewendet werden. Für den Fall $s_o > s_i$ wird die Operation zur späteren Ausführung zurückgestellt, da die lokale Instanz noch nicht alle kausal vorhergehenden Operationen angewendet hat, weil diese etwa noch nicht empfangen wurden. Im Falle von $s_o < s_i$ bezieht sich die Operation auf einen älteren Dokumentenzustand, die lokale Instanz hat also bereits nebenläufig andere Operationen angewendet, was eine Transformation der Operation erforderlich macht. Dazu wird L_i nach bereits angewendeten Operationen durchsucht, die nebenläufig zu o erzeugt wurden, um im Folgenden o dementsprechend zu transformieren.

Allgemein müssen alle Operationenpaare o_i, o_j mithilfe von Transformationsfunktionen gegeneinander transformiert werden können. Im Falle von *dOPT* sind dies nur die beiden Operationen $insert(i, c)$ und $delete(i)$, was eine 2×2 Transformationsmatrix ergibt. Eine Transformationsfunktion t muss die gegebenen Operationen o_i und o_j entsprechend umwandeln in die beiden neuen Operationen o_i' und o_j', unter Einhaltung der Bedingung $o_j' \circ o_i = o_i' \circ o_j$, sodass sowohl die Anwendung von o_i, gefolgt von o_j', als auch von o_j, gefolgt von o_i', in einem identischen Dokumentenzustand mündet.

```
1 t(delete(i), delete(j)) {
2     if (i < j) o_i' = delete(i)
3     if (i > j) o_i' = delete(i - 1)
4     else o_i' = ∅
5 }
```

Listing 2.2: Transformationsfunktion im *dOPT* Algorithmus

Zur Veranschaulichung muss die für den in Abbildung 2.13 dargestellten Fall von zwei nebenläufigen Löschoperationen erforderliche Transformationsfunktion (siehe Listing 2.2) beispielsweise das bereits entfernte Zeichen in Betracht ziehen

[11] distributed operational transformation
[12] History Buffer

und die Position der zweiten Löschoperation entsprechend anpassen. Beziehen sich beide Löschoperationen auf dieselbe Position, so wurde das Zeichen durch die erste Operation entfernt, sodass die zweite nicht mehr angewendet werden muss.

Grundsätzliche werden zwei Arten von Transformationsfunktionen unterschieden, die in den Kontrollalgorithmen Verwendung finden (Chengzheng Sun, Jia, Zhang, Yang & Chen, 1998). *Inclusion Transformation* Funktionen werden dabei von jedem Algorithmus benötigt, *Exclusion Transformation* Funktionen dagegen nur, wenn der Kontrollalgorithmus es erfordert die Auswirkung einer Operation aus einer anderen heraus zu rechnen.

Inclusion Transformation (IT)
: Entspricht der Transformationsfunktion in *dOPT*, indem die Operationen o_i und o_j entsprechend transformiert werden, sodass die Auswirkung von o_j in der neuen Operation eingebettet ist

Exclusion Transformation (ET)
: Die Operationen o_i und o_j werden transformiert, sodass die Auswirkung von o_j aus o_i ausgenommen wird

Allen *OT* Algorithmen liegt ein bestimmtes *OT* Framework zugrunde, das zum Einen das eingesetzte Konsistenzmodell definiert und außerdem das Vorgehen aufzeigt, um mit dem Modell übereinstimmende Algorithmen zu entwickeln (D. Li & Li, 2007). In der Literatur finden sich eine Vielzahl, teils aufeinander aufbauender Frameworks, die im Folgenden kurz vorgestellt werden.

CC Das auf dem ursprünglichen Konsistenzmodell von C. A. Ellis und Gibbs (1989) basierende *CC* Framework (Ressel, Nitsche-Ruhland & Gunzenhäuser, 1996) fordert die beiden Konsistenzkriterien (1) Kausalitätserhaltung[13] und (2) Konvergenz[14]. Transformationsfunktionen müssen die beiden Eigenschaften *TP1* und *TP2* erfüllen, um Konvergenz unter Verwendung beliebiger Transformationspfade sicherzustellen.

Transformation Property 1 (TP1)
: $o_i \circ o'_j = o_j \circ o'_i$

Transformation Property 2 (TP2)
: Für jede Operation o gilt: $t(t(o, o_i), o'_j) = t(t(o, o_j), o'_i)$

TP1 entspricht der aus *dOPT* bekannten Bedingung. *TP2* fordert darüber hinaus, dass die Transformation einer beliebigen Operation mit zwei nebenläufigen Operationen in beliebiger Reihenfolge identische Operationen ergibt.

[13] Causality Preservation
[14] Convergence

CCI

Das *CCI* Framework erweitert das Konsistenzmodell des *CC* Frameworks um Absichtserhaltung[15] (Chengzheng Sun u. a., 1998). Die Absicht, die ein Benutzer durch eine Operation ausdrücken will, muss auch bei der Anwendung der Operation auf allen anderen Instanzen erhalten bleiben.

CSM

Mit dem *CSM* Framework führen D. Li und Li (2008) ein neues, erstmals verifizierbares Konsistenzmodell ein, um die Korrektheit des Modells im Hinblick auf *TP1* und *TP2* beweisen zu können.

Der Ansatz basiert auf dem Konzept der Operationeneffektrelation, aus der sich eine Totalordnung auf den durch Operationen an der Effektposition erzeugten Elementen ableiten lässt. Das Konsistenzmodell fordert drei Bedingungen, um Konsistenz gewährleisten zu können: (1) Operationen müssen entsprechend ihrer Kausalitätsreihenfolge angewendet werden, (2) die Anwendung einer Operation muss auf dem aktuellen Zustand des Dokuments den gleichen Effekt erzielen wie auf dem Zustand, der bei Erstellung der Operation vorlag, und (3) gleiches gilt für die Anwendung zweier beliebiger Operationen in beliebigem Zustand des Dokuments.

CR Als eine Alternative zu *TP2*, mit dem Ziel diese Bedingung zu ersetzen und damit formal beweisbar zu machen, wurde das *CSM* Konsistenzmodell im *CR* Framework umformuliert (R. Li & Li, 2005). Dieses besteht nur noch aus den beiden Bedingungen (1) Kausalitätserhaltung und (2) Erhaltung der Operationeneffektrelation.

Der eigentliche Beitrag des *CR* Frameworks besteht jedoch in der Einführung einer neuen Methodik zur Entwicklung von geeigneten *OT* Algorithmen, indem zuerst geeignete Bedingungen identifiziert und diese für die grundlegenden Transformationsfunktionen bewiesen werden, um daraufhin im Kontrollalgorithmus bestimmte Transformationspfade zu generieren, die die zuvor identifizierten Bedingungen sicherstellen. Im Gegensatz zu vorhergehenden Ansätzen, in denen ein generischer Kontrollalgorithmus Verwendung findet, spielt die schwierig zu beweisende Bedingung *TP2* durch die Verwendung spezieller Transformationspfade keine Rolle mehr. Dagegen fordert das Konsistenzmodell jedoch eine applikationsspezifische Totalordnung der Elemente, die nur schwer zu definieren und zu beweisen sind (D. Li, 2011).

CA Basierend auf den Erkenntnissen aus dem *CR* Framework besteht das Konsistenzmodell im *CA* Framework aus den zwei formalisierten Bedingungen (1) Kausalitätserhaltung und (2) Zulässigkeitserhaltung[16] (D. Li & Li,

[15] Intention Preservation
[16] Admissibility Preservation

2010). Zulässigkeitserhaltung bedeutet zum Einen, dass eine Operation in ihrem Definitionszustand zulässig sein muss, und zum Anderen, dass die Anwendung einer jeden Operation in einem beliebigen Zustand die durch die vorhergehende Anwendung zulässiger Operationen erzeugte Elementreihenfolge nicht verletzen darf.

Anstatt einer erforderlichen Totalordnung wie im *CR* Framework wird die Ordnung der Elemente inkrementell durch Anwendung zulässiger Operationen erzeugt und bildet eine Halbordnung, was den Korrektheitsbeweis signifikant vereinfacht (D. Li, 2011).

Es wurde jedoch gezeigt, dass die von *dOPT* geforderten Konsistenzkriterien Konvergenz und Kausalitätserhaltung nicht ausreichend dafür sind, Konsistenz zu gewährleisten (C. A. Ellis & Gibbs, 1989).

Um die Konsistenz innerhalb eines *OT* Systems zu gewährleisten, existieren beim Entwurf von Transformationsfunktionen zwei Ansätze (Xu, Sun & Li, 2014): Zum Einen der Entwurf applikationsspezifischer Transformationsfunktionen, die dazu fähig sind, sowohl *TP1* als auch *TP2* zu gewährleisten. Und zum Anderen *TP1* durch generische Transformationsfunktionen sicher zu stellen und *TP2* durch entsprechende Kontrollalgorithmen zu vermeiden.

Der Entwurf von *TP2* erfüllenden Transformationsfunktionen gestaltet sich jedoch als überaus schwierig (Imine, Molli, Oster & Rusinowitch, 2003), weshalb für eine Vielzahl von als geeignet vorgestellten Funktionen Konstellationen gefunden wurden, für die *TP2* nicht erfüllt wird.

Mit den *Tombstone Transformationsfunktionen* (*TTF*) (G. Oster, Imine, Molli & Urso, 2006) wurde jedoch eine Menge sowohl *TP1* als auch *TP2* erfüllender Funktionen veröffentlicht, die das Konzept der *Tombstones* aus dem *WOOT*-Framework übernehmen und innerhalb von *OT* anwenden. Dementsprechend werden Elemente durch Löschoperationen nicht mehr entfernt, sondern durch *Tombstones* ersetzt, also nur vor dem Benutzer verborgen.

In der Literatur wurden verschiedene Verfahren zur Lösung dieser Probleme veröffentlicht. Eine Auswahl von *OT*-Kontrollalgorithmen daraus wird in der folgenden Auflistung kurz vorgestellt. Im Anschluss daran werden die wesentlichen Eigenschaften der Algorithmen in Tabelle 2.1 gegenübergestellt.

Jupiter

Eine Weiterentwicklung von *dOPT* findet sich in *Jupiter* (Nichols u. a., 1995), einem synchronen Groupware Framework, das im Gegensatz zum ursprünglich *Peer-To-Peer* Verfahren auf einer zentralisierten Kommunikation über eine Serverkomponente basiert. Der Einsatz eines zentralen Servers kann sich nicht nur positiv auf die Reaktionszeit auswirken, sondern ermöglicht auch eine wesentliche Vereinfachung des *dOPT* Verfahrens, da

die Kommunikation nur noch zwischen zwei Instanzen erfolgt, einem Client und dem Server. Operationen, die von anderen Clients erzeugt werden, erscheinen dabei für den eigenen Client, als wären diese auf Serverseite erzeugt worden. Durch die Serialisierung der Operationen auf Serverseite entsteht eine Totalordnung und erfüllt die Kriterien zur Vermeidung von *TP2* (Xu u. a., 2014).

Wave

Der von Google entwickelte *OT* Algorithmus, der in *Google Wave* und später in *Google Docs* implementiert wurde, basiert auf dem *Jupiter* System. Dieses wurde jedoch um verschiedene Konzepte erweitert, die vor allem auf Serverseite eine verbesserte Effizienz erbringen sollen, um das System möglichst einfach skalieren zu können (D. Wang, Mah & Lassen, 2010). *Google Wave* wurde 2012 als eigenständiges Projekt eingestellt[17] und wird seither als *Apache Wave*[18] in Form eines Apache Incubator Projekts fortgeführt.

GOT

Generic Operation Transformation (*GOT*) führt das zusätzliche Konsistenzkriterium der Absichtserhaltung und den Begriff des Operationenkontexts (*DC*) ein, den Dokumentenzustand, auf den sich eine Operation bezieht, sowie den Anwendungskontext (*EC*), dem Zustand der Anwendung einer Operation (Chengzheng Sun u. a., 1998).

Zur Sicherstellung der Konvergenz und Absichtserhaltung wird ein *undo/-do/redo* Schema eingesetzt, das ein zu einer Totalordnung von Operationen identisches Resultat liefert. Dazu werden die für die Anwendung einer empfangenen und unter Berücksichtigung der Kausalität anwendbaren Operation O_{new} zunächst alle Operationen O_p im Operationenprotokoll O_L identifiziert, die kausal nach O_{new} erzeugt wurden. Die in O_p enthaltenen Operationen werden daraufhin rückgängig gemacht. O_{new} wird entsprechend transformiert, sodass *EC* dem aktuellen Zustand entspricht und angewendet werden kann. Gleiches wird mit den in O_p enthaltenen Operationen vollzogen, wodurch diese nun sukzessiv im jeweils neu enstandenen *EC* angewendet werden können.

Darauf aufbauend steht mit *GOTO*[19] (Chengzheng Sun & Ellis, 1998) eine optimierte Variante zur Verfügung, die die Anzahl der notwendigen Transformationen reduziert, indem sowohl *DC* als auch *EC* einer Operation transformiert werden, mit dem Ziel äquivalente Kontexte zu erhalten.

[17] https://support.google.com/answer/1083134
[18] http://incubator.apache.org/wave/
[19] *GOT* Optimized

AnyUndo

AnyUndo (Chengzheng Sun, 2000) stellt ein generisches *Undo*-Framework dar, das es ermöglicht, eine beliebige Operation zu jedem beliebigen Zeitpunkt rückgängig machen zu können. Die Basis dafür bildet GOT, welches um zwei zusätzliche Komponenten erweitert wurde, zum Einen dem *Undo*-Algorithmus, der entscheidet, wie eine Operation rückgängig gemacht wird, und zum Anderen der *Undo*-Strategie zur Auswahl der rückgängig zu machenden Operation.

Um eine Operation O rückgängig zu machen, wird dazu eine neue, zu O inverse Operation $\overline{O}$ direkt nachfolgend zu O erzeugt, die die Auswirkungen von O aufhebt. Das Paar $O \circ \overline{O}$ kann somit als Identitätsoperation angesehen werden, die keine Auswirkung auf den Zustand des Dokuments hat und deren Operationen Vektoruhren identischen Inhalts zugewiesen werden.

Für Strategien kann jeweils zwischen globalem und lokalem Wirkungsbereich unterschieden werden. Eine globale Strategie zieht bei der Auswahl die Operationen aller Teilnehmer in Betracht, wohingegen eine lokale Strategie nur die lokal erzeugten Operationen berücksichtigt. Zusätzlich basiert eine Strategie auf einem der im folgenden aufgeführten Modelle zur Auswahl der rückgängig zu machenden Operation:

Einzelschritt Jeweils die aktuellste Operation

Chronologisch Eine Folge von Operationen in ihrer chronologischen Reihenfolge

Selektiv Eine beliebige Operation in beliebiger Reihenfolge

COT

Context-Based OT (COT) baut auf dem Konzept des Operationenkontexts auf, der im Gegensatz zur Verwendung in GOT jedoch nicht die Menge der transformierten, sondern originalen Operationen enthält, die, auf das ursprüngliche Dokument angewendet, dieses in den Zustand überführen, auf dem eine Operation definiert ist (D. Sun & Sun, 2006). Wie bei *AnyUndo* besteht die Möglichkeit, eine beliebige angewendete Operation zu beliebigem Zeitpunkt rückgängig zu machen.

Auf Basis des Operationenkontexts werden sechs neue kontextbasierte Bedingungen (CC[20]) eingeführt, die ein OT basiertes Groupwaresystem erfüllen muss. Diese sind, im Gegensatz zur Darstellung der Abhängigkeiten zwischen Operationen mittels Kausalitätsbeziehungen, in der Lage, auch inverse Operationen darzustellen.

[20] Context-Based Conditions

TIBOT

Der auf Zeitintervallen basierende *OT* Algorithmus *TIBOT*[21] führt verschiedene Restriktionen bei der Kommunikation zwischen Instanzen ein (R. Li, Li & Sun, 2004).

Die Kommunikation zwischen den Instanzen wird dabei in logische Zeitintervalle aufgeteilt. Jede Operation O, die innerhalb eines Zeitintervalls t erzeugt wird, besitzt eine Referenz auf t. Anstatt eine lokal erzeugte Operation unverzüglich an die anderen Instanzen zu verteilen, werden Operationen erst mit dem Beginn des nachfolgenden Zeitintervalls übertragen, wenn diese gegen alle Operationen aus vorhergehenden Intervallen transformiert wurden. Dies setzt den Erhalt der Operationen aller teilnehmenden Instanzen voraus oder alternativ einer *Null-Operationen Nachricht*, die das Ende eines Zeitintervalls ohne Operationen markiert. Alle Operationen einer Instanz, sowie eines Zeitintervalls werden als Einheit betrachtet und immer gemeinsam behandelt. Durch die Synchronisierung an Zeitintervallen und dadurch, dass die Anwendung von Operationen ähnlich dem aus *GOT* bekannten *undo/do/redo* Schemas erfolgt, um eine Totalordnung zu erhalten, kann Konvergenz sichergestellt werden.

Darauf aufbauend stellen Xu und Sun (2016) eine neue Variante *TIBOT 2.0* vor, die bei der Behandlung von empfangenen Operationen durch Anwendung symmetrischer Transformationen auf das *undo/do/redo* Schema verzichtet, da dieses unter Umständen zu unerwarteten Ergebnissen in der Benutzerschnittstelle führen können.

SOCT

SOCT[22] setzt auf eine globale Serialisierungsreihenfolge, die durch einen zentralen Dienst zur Bereitstellung von eindeutigen Zeitstempeln festgelegt wird (Vidot, Cart, Ferrié & Suleiman, 2000). *SOCT3* und *SOCT4* unterscheiden sich dahingehend, welche Instanz für die Transformation nebenläufiger Operationen zuständig ist. *SOCT3* übermittelt eine Operation direkt nach Erhalt des Zeitstempels an die anderen Teilnehmer, woraufhin diese lokale Transformationen durchführen müssen. *SOCT4* transformiert die Operation dagegen nach Zuweisung des Zeitstempels zunächst lokal und verteilt die Operation in transformierter Form.

MOT

Auf *SOCT4* aufbauend, präsentieren Cart und Ferrié (2006) den Synchronisierer *MOT*[23] zur paarweisen Synchronisierung zweier Dokumente durch Abgleich der angewendeten Operationen in den jeweiligen Operationenpro-

[21] Time Interval Based Operational Transformation
[22] Sérialisation des Opérations Concurrentes par Transposition
[23] Merge based on Operational Transformation

tokollen. Dies kann wahlweise über einen zentralen Server erfolgen (*MOT1*) oder verteilt in einer *P2P*-Architektur (*MOT2*).

SLOT

Der in Shen und Sun (2002) vorgestellte Kontrollalgorithmus *SLOT*[24] basiert auf einem flexiblen Framework zur Beschreibung von Benachrichtigungsarten und erlaubt daher die Anpassung von Frequenz und Granularität der Benachrichtigung. Es werden je Instanz zwei Warteschlangen *OB* (*Outgoing Buffer*) und *IB* (*Ingoing Buffer*) eingesetzt, die jeweils bestimmen zu welchem Zeitpunkt welche Operationen wie an andere Instanzen übermittelt bzw. empfangen und verarbeitet werden. Operationen in *OB* sind ihrem kausalen Zusammenhang entsprechend geordnet, in *IB* kontextuell serialisiert.

Zur Übermittlung einer Folge von Operationen O aus *OB* muss O mittels Transformationsfunktionen hinter die Position der zuletzt übermittelten Operation verschoben werden, um sicherzustellen, dass sich beim Empfang von O keine kausal vorausgehenden Operationen in *IB* befinden. Zusätzlich muss O gegen alle empfangenen Operationen in *IB* transformiert werden, um O beim Empfang an *IB* anhängen zu können. Beides erfolgt, indem die Operationen aus *IB* und *OB* symmetrisch gegeneinander transformiert werden.

Für die Anwendung einer Operationenfolge O in *IB* muss O zum Einen unter Verwendung von Transformationsfunktionen an den Beginn von *IB* verschoben werden und zum Anderen gegen noch nicht übermittelte Operationen aus *OB* transformiert werden, um die darin enthaltenen Absichten zu erhalten.

SDT

Der auf dem *CSM* Framework basierende *SDT*[25] Algorithmus erzielt Konvergenz unter Zuhilfenahme der Effektrelation $\prec$ zwischen Zeichen $'a' \prec' b' \prec' c'$ bzw. Operationen $O_a \prec O_b \prec O_c$ und der daraus abgeleiteten Effektpositionen $P(O)$ (D. Li & Li, 2008). Die Transformationsfunktionen früherer *OT* Algorithmen beruhen dagegen auf einem Vergleich der in den Operationen enthaltenen Positionsangaben.

Der Kontrollalgorithmus lehnt sich an die Idee von *GOTO* an. Zur Anwendung einer kausal bereiten Operation O wird das Operationenprotokoll HB zunächst so transponiert, dass eine Sequenz SQ_p von Operationen, die O kausal vorhergehen, gefolgt von einer Sequenz SQ_c zu O nebenläufiger Operationen entsteht. O wird gegen die Sequenz SQ_c inklusiv transformiert,

[24] Symmetric Linear Operation Transformation
[25] State Difference Based Transformation

woraus O' erzeugt wird. O' wird angewendet und anschließend an HB angehängt.

Der formale Beweis des *SDT* Algorithmus stellte sich aufgrund einer zu starken Fokussierung auf den Beweis von *TP2* anstatt des *CSM* Konsistenzmodells im Nachhinein jedoch als fehlerhaft heraus (D. Li, 2011).

LBT

Mit *LBT*[26] stellen R. Li und Li (2005) einen *OT* Algorithmus auf Basis des *CR* Frameworks vor.

Um *TP2* zu vermeiden wird bei der Transformation einer empfangenen Operation O zur Anwendung im aktuellen Dokumentenzustand s ein spezieller Transformationspfad P konstruiert (D. Li & Li, 2007). P setzt sich hierbei aus einem rückwärts gerichteten Pfad P_b, der s zunächst in einen bestimmten Zustand s'' überführt, sowie einem vorwärts gerichteten Pfad P_f zusammen, um von s'' in den finalen Zustand s' zu gelangen (siehe Abbildung 2.14). P_b beinhaltet die Löschoperationen aus dem lokalen Operationenprotokoll HB, die kausal vor o liegen. Ebenso P_f, zusätzlich erweitert um die zu o nebenläufigen Operationen in HB.

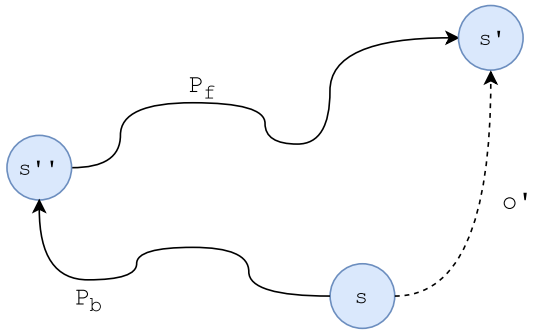

Abbildung 2.14: *LBT* Transformationspfad nach D. Li und Li (2007)

Nach der exklusiven Transformation von o zu o'' mit der Sequenz von Operationen in P_b stellt s'' den Zustand dar, in dem alle zuvor entfernten Zeichen wiederhergestellt wurden. Damit existieren alle für Einfügeoperationen erforderlichen Zeichen (*landmarks*), womit Zeichen an der korrekten Position eingefügt werden können. Abschließend wird o'' mit P_f zu o' inklusiv transformiert, das durch Anwendung s in s' überführt.

ABT

Das *Admissibility-Based-Transformation* Framework (D. Li & Li, 2010) führt das *CA* Konsistenzmodell ein, das aus Kausalitätserhaltung und Zulässigkeit besteht und, im Vergleich zu vorhergehenden Modellen, sowohl

[26] Landmark Based Transformation

formalisierbar als auch beweisbar ist. Zulässig ist die Anwendung von Operationen, wenn diese keine durch zuvor angewendete, zulässige Operationen erzeugte Objektbeziehungen verletzen. Auf Basis der formalisierten Konsistenzbedingungen wird ein Kontrollalgorithmus entwickelt, der diese erfüllt. Eine Trennung von Kontrollalgorithmus und Transformationsfunktionen ist dabei jedoch nicht mehr gegeben.

Das Operationenprotokoll besteht aus einer Liste von Einfügeoperationen, gefolgt von einer Liste der Löschoperationen. Daraus folgt, dass bei der Transformation von nebenläufigen Einfügeoperationen dazwischenliegende Zeichen immer vorhanden sind, sog. *landmarks* also nicht verloren gehen. Dies ermöglicht die Generierung von speziellen Transformationspfaden, die eindeutig sind und eine zulässige Folge von Operationen bilden.

Aufbauend auf dem ursprünglichen *ABT* Framework bzw. den dort eingeführten neuen Konvergenzkriterien wurden in der Folge Algorithmen mit erweiterter Funktionalität vorgestellt. Während *ABT* keine Möglichkeit bereit stellt, zuvor angewendete Operationen wieder rückgängig zu machen, erweitert *ABT-Undo* (*ABTU*) das *ABT* Framework um selektives *Undo* (Shao, Li & Gu, 2010b). Um dem immer mehr an Bedeutung gewinnenden mobilen Anwendungssektor Rechnung zu tragen und den dortigen Restriktionen hinsichtlich Leistungsfähigkeit, Akkulaufzeit und unterbrochenen Verbindungen gerecht zu werden, steht weiterhin der *Admissibility-Based Sequence Transformation* Algorithmus (*ABST*) zur Verfügung (Shao, Li & Gu, 2010a). Dieser optimiert die Behandlung von Operationensequenzen, indem die darin enthaltenen Operationen nicht wie bisher unabhängig voneinander angewendet werden, sondern die gesamte Sequenz transaktionsorientiert betrachtet. Weiterhin optimiert *ABTSO*[27] den *ABST* Algorithmus, indem die Operationen in den beiden Listen des Operationenprotokolls nicht wie bisher nach ihrer Anwendungsreihenfolge, sondern direkt entsprechend ihrer Effektrelation geordnet sind (Shao, Li & Gu, 2009).

POT

Der *Pattern-based OT* Ansatz basiert auf einer Client-Server Architektur (Xu & Sun, 2016). Die Serverkomponente dient als Vermittler sowie der Serialisierung von Operationen durch Zuweisung einer eindeutigen Nummer, um eine Totalordnung der Operationen zu generieren. Die Transformation von Operationen erfolgt nur auf Seite der Clients.

Die zentrale Komponente zur Transformation von Operationen ist die sog. *Transformation Map* (*TM*), in der ein Client alle Transformationspfade

[27] *ABT* - String Optimized

(TP[28]) der entfernten Instanzen beinhaltet. So enthält der lokale TP für Client c alle Operationen, die nicht von c generiert wurden und damit einen Pfad zur Transformation von Operationen, die von c empfangen werden.

pOT

Um der Entwicklung von Mehrkernarchitekturen der letzten Jahre gerecht zu werden, beschreiben Cai, He und Lv (2015) einen parallelisierten OT Algorithmus (pOT). Dieser kann die Anwendung mehrerer Operationen beschleunigen, indem die Transformationen parallelisiert werden. Dazu werden empfangene Operationen in einer nicht-blockierenden Warteschlange *RQueue* vorgehalten, ebenso wie das Operationenprotokoll *HQueue*. Die Einträge in *RQueue* werden parallel gegen *HQueue* transformiert und die transformierte Operation mittels CAS[29] Befehl an das das Ende von *HQueue* angehängt. Bei Konflikten durch nebenläufige Threads wird der Transformationsvorgang wiederholt und ein erneuter Versuch gestartet, die Operation an *HQueue* anzuhängen.

TIPS

Für den Einsatz im *Web 2.0* Kontext stellen (Shao, Li, Lu & Gu, 2011) das $TIPS$[30] OT Protokoll vor, das speziell auf das in $HTTP$ verwendete Anfrage/Antwort Kommunikationsprinzip hin entwickelt wurde. Das grundlegende Konzept basiert auf einer Kombination von Ideen aus den beiden OT Protokollen $ABST$ und $TIBOT$, und kann dadurch für ein breites Spektrum der Zusammenarbeit eingesetzt werden, von nahezu in Echtzeit bis hin zu asynchron.

Das Synchronisationsprotokoll sieht einen zentralisierten Server sowie eine beliebige Anzahl an Clients vor, die zu jedem beliebigen Zeitpunkt einer Sitzung beitreten oder diese wieder verlassen können. Jeder Client c kann unabhängig von anderen entscheiden, in welchem Zeitintervall t_c die Synchronisierung mit dem Server durchgeführt werden soll. Der Server wiederum entscheidet selbst, in welchem Zeitintervall t_s empfangene Operationen angewendet werden sollen.

Während t_c werden lokal erzeugte Operationen in einem Puffer T_c zwischengespeichert und während t_s von c empfangene Operationen in RB_c. Zusätzlich enthält SB_c auf Serverseite Operationen, deren Zustellung an c noch aussteht und die c zu beliebiger Zeit abfragen kann.

Zum Ende von t_s werden alle in den jeweiligen RB Puffern enthaltenen Operationen angewendet und die SB Puffer befüllt. Bei Erreichen von t_c fragt c alle Operationen aus SB_c ab und transformiert den Inhalt gegen T_c,

[28] Transformation Paths

[29] compare-and-swap

[30] Transformation and Time Interval Based Protocol for Synchronization

um SB'_c und T'_c zu erhalten. Die Operationen in SB'_c werden auf die lokale Kopie des Dokuments angewendet und T'_c an den Server übermittelt.

Neben der linearen Darstellung gibt es auch Ansätze, OT auf einer hierarchischen Baumrepräsentation eines Dokuments anzuwenden, wie etwa der *treeOPT* Algorithmus (C. Ignat, 2002). Bei ersterem wird ein globaler Puffer für die Operationenhistorie verwendet, der bei Konfliktauflösung nebenläufiger Operationen stets vollständig durchsucht werden muss, auch wenn Operationen aufgrund ihrer Änderungsposition unmöglich im Konflikt zueinander stehen können. Im hierarchischen Ansatz kann die Historie dagegen auf die verschiedenen Ebenen aufgeteilt werden, sodass ein Knoten nur alle Operationen erfasst, die seine Kindknoten betreffen. Daneben kann die zeichenbasierte Operationensemantik keine semantische Konsistenz erzwingen. Operationen auf höheren Abstraktionsleveln liefern im Allgemeinen auch ein semantisch konsistenteres Ergebnis. So verwendet *treeOPT* für Textdokumente etwa die folgende Granularität: Dokument, Paragraph, Satz, Wort und Zeichen. Darüber hinaus kann *treeOPT* auf der Baumstruktur von XML Dokumenten angewendet werden (C.-L. Ignat & Norrie, 2006).

Ebenfalls basierend auf einer Baumrepräsentation erweitern Davis, Sun und Lu (2002) den *GOTO* Algorithmus für die Bearbeitung von beliebigen Dokumenten auf Basis des SGML Datenmodells, also etwa XML oder HTML.

Im Gegensatz zu den zuvor vorgestellten Algorithmen, die ausschließlich wissenschaftlichen Veröffentlichungen zugrunde liegen, findet das *Wave* Protokoll auch im Produktivumfeld Anwendung. Darüber hinaus weisen das Dokumentenmodell, sowie die darauf zur Verfügung stehenden Operationen eine starke Ähnlichkeit zu der im *VisualEditor* verwendeten Technologie auf (siehe Kapitel 3.3.2). Daher werde ich das *Wave* Protokoll im Folgenden ausführlicher behandeln, um eine mögliche Erweiterung des *VisualEditors* um Kollaborationsfähigkeiten in Echtzeit zu vereinfachen.

Dadurch, dass jeder Client im *Jupiter* System beliebig viele Operationen an den Server übermitteln kann, muss die Serverkomponente die Zustandsräume für jede Verbindung aufbewahren, was zum Einen den Speicherbedarf erhöht und zum Anderen die Transformation der Clientoperationen zwischen den Zustandsräumen komplexer gestaltet. Stattdessen fordert das *Google OT* Protokoll, dass Operationen vom Server bestätigt werden müssen und nur eine einzige Operation gleichzeitig übermittelt werden darf. Die Antworten sind als *commit* Nachrichten anzusehen, die bestätigen, dass die Operation auf dem Zustand des Servers angewendet wurde und die anderen Teilnehmer darüber in Kenntnis gesetzt wurden. Die Bestätigungen enthalten darüber hinaus den aktuellen Zustand des Servers sowie von anderen Instanzen durchgeführte Operationen. Operationen, die auf Clientseite generiert werden während auf die Bestätigung des Servers gewartet wird, werden lokal zwischengespeichert und nach Erhalt der Bestätigung gebündelt in eine einzige Operation überführt, mit potentiell empfangenen

Tabelle 2.1: Übersicht der *Operational Transformation* Algorithmen

Algorithmus	Quelle	Frame-work	Transformations-funktionen	Topologie	Eigenschaften Transformations-funktionen	Eigenschaften Kontrollalgorithmus
dOPT	C. A. Ellis und Gibbs (1989)	CC	IT	P2P	–	TP1, TP2
Jupiter / Wave	Nichols, Curtis, Dixon und Lamping (1995)	CC	IT	Client/Server	TP1	TP2
GOT	Chengzheng Sun, Jia, Zhang, Yang und Chen (1998)	CCI	IT, ET	P2P	–	TP1, TP2
AnyUndo	Chengzheng Sun (2000)	CCI	IT, ET	P2P	TP1, TP2	–
COT	D. Sun und Sun (2006)	CC	IT	P2P	TP1	TP2
TIBOT	R. Li, Li und Sun (2004)	CCI	IT	P2P	–	TP1, TP2
TIBOT2	Xu und Sun (2016)	CCI	IT	P2P	TP1	TP2
SOCT	Vidot, Cart, Ferrié und Suleiman (2000)	CCI	IT, ET	Client/Server	TP1	TP2
MOT	Cart und Ferrié (2006)	CCI	IT, ET	P2P,Client/Server	TP1	TP2
SLOT	Shen und Sun (2002)	CCI	IT	P2P	TP1	TP2
SDT	D. Li und Li (2008)	CSM	IT, ET	P2P	TP1, TP2	–
LBT	R. Li und Li (2005)	CR	IT, ET	P2P	TP1, TP2	–
ABT	D. Li und Li (2010)	CA	IT, ET	P2P	TP1, TP2	–
POT	Xu und Sun (2016)	CCI	IT	P2P,Client/Server	TP1	TP2
TIPS	Shao, Li, Lu und Gu (2011)	CA	IT, ET	Client/Server	TP1, TP2	–

Serveroperationen transformiert und daraufhin an den Server übermittelt.

Ein Dokument im *Wave* Protokoll besteht aus einem wohlgeformten XML-Dokument, das mit Annotationen angereichert werden kann, etwa zur Textauszeichnung, für Links oder auch Cursorpositionen, und zur Bearbeitung in eine interne, als Datenstrom verwendbare Datenstruktur überführt wird. Das Dokument ist eine linear adressierbare Struktur, deren Inhalt über einen Index angesprochen werden kann, wobei jedes Zeichen sowie der Beginn bzw. das Ende eines Elements oder einer Annotation jeweils einer Adresse entspricht (siehe Abbildung 2.15). Die Interaktion mit dem Dokument erfolgt ausschließlich mittels Operationen, die auf dem Dokument angewendet werden. Eine Operation ist wiederum aus Komponententypen zusammengesetzt, die jeweils bestimmte Aktionen auf der Datenstromrepräsentation des Dokuments, relativ zur aktuellen Cursorposition, ausführen. Die datenstromorientierte Darstellung der Operationen ermöglicht es, zwei kompatible Operationen effizient zu einer neuen Operation zusammenzufassen, die die Auswirkungen beider Operationen in sich vereint (Spiewak, 2010).

$_0$H$_1$e$_2$l$_3$l$_4$o$_5$ $_6$$_7W_8o_9r_{10}l_{11}d_{12}$!$_{13}$$_{14}$

Abbildung 2.15: Lineare Repräsentation von `Hello <i>World!</i>` in *Wave*

Die folgenden Komponententypen stehen zur Komposition von Operationen in *Google Wave* zur Verfügung (D. Wang u. a., 2010):

retain
: Verschiebt den Cursor

insertCharacters
: Fügt eine Zeichenkette an der aktuellen Cursorposition ein

insertElementStart
: Fügt den Beginn eines Elements an der aktuellen Cursorposition ein

inserEelementEnd
: Fügt das Ende eines Elements an der aktuellen Cursorposition ein

deleteCharacers
: Entfernt eine Zeichenkette ab der aktuellen Cursorposition

deleteElementStart
: Entfernt den Beginn eines Elements an der aktuellen Cursorposition

deleteElementEnd
: Entfernt das Ende eines Elements an der aktuellen Cursorposition

replaceAttributes
: Ersetzt die Attribute des Elements an der aktuellen Cursorposition

updateAttributes
: Aktualisiert die Attribute des Elements an der aktuellen Cursorposition

annotationBoundary
: Aktualisiert die Annotationen an der aktuellen Cursorposition

So könnte das Beispieldokument `Hello <i>World!</i>` etwa durch die Operation, bestehend aus den in Listing 2.3 aufgeführten Komponententypen, angewendet auf ein leeres Dokument, dargestellt werden.

```
insertCharacters('Hello ')
annotationBoundary(
    startKeys: ['style/font-style'],
    startValues: ['italic']
)
insertCharacters('World!')
annotationBoundary(
    endKeys: ['style/font-style']
)
```

Listing 2.3: Mögliche Operation zur Erstellung eines *Wave*-Dokuments mit dem Inhalt `Hello <i>World!</i>`

Operationen auf einem *Wave*-Dokument müssen stets das gesamte Dokument umfassen, der Cursor nach Ausführung aller Komponententypen also am Ende des Dokuments stehen. Um im vorhergehenden Beispieldokument also bspw. `World` durch `FAU` zu ersetzen, kann die in Listing 2.4 aufgeführte Operation verwendet werden.

```
retain(7)
deleteCharacters('World')
insertCharacters('FAU')
retain(2)
```

Listing 2.4: Mögliche Operation zum Ersetzen von `World` durch `FAU` im Dokument `Hello <i>World!</i>`

Erst durch Interaktion mehrerer Teilnehmer einer Sitzung können nebenläufige Operationen entstehen. In diesem Fall müssen die Operationen transformiert werden, um die Konsistenz der Dokumente auf den verschiedenen Instanzen gewährleisten zu können. Das im Folgenden verwendete Beispiel verwendet zwei an der Bearbeitung beteiligte Instanzen a und b, wobei aus Sicht einer Instanz alle Operationen, die ihren Ursprung nicht bei sich selbst haben, von der zentralen Serverinstanz kommen. Damit ist das Beispiel auch allgemeingültig für eine beliebige Anzahl an Teilnehmern.

Bei der Anwendung von Operationen wird das Dokument durch einen Zustandsraum bewegt, welcher als Hasse Diagramm visualisiert werden kann, wie etwa in Abbildung 2.16 dargestellt. Dabei stellen Punkte mögliche Zustände dar, in denen sich ein Dokument befinden kann, wobei ein Tupel (C_c, C_s) die Anzahl der angewendeten Operationen durch den Client C_c sowie durch den Server C_s beschreibt. Die beiden Linien beschreiben die Pfade, die Client bzw. Server beschritten haben, um in den aktuellen Zustand zu gelangen. Sobald sich beide Linien in einem Punkt treffen, befinden sich die Dokumente auf Client- und Serverseite im gleichen Zustand, die Konsistenz wurde also wiederhergestellt.

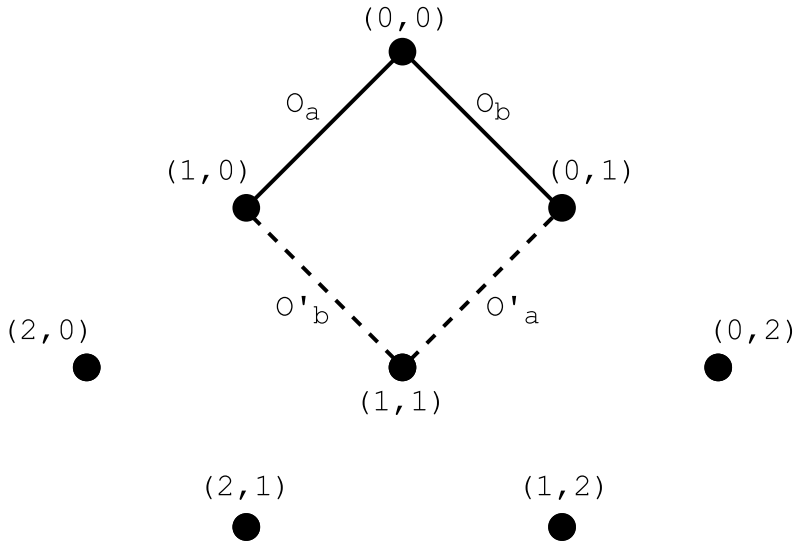

Abbildung 2.16: *Operational Transformation* Zustandsraum von Client und Server adaptiert von Spiewak (2010)

Für Operationen, die ein Client an den Server schickt, gelten verschiedene Bedingungen. Zum Einen muss diese vom Server bestätigt werden, bevor der Client eine weitere Operation senden darf. Zum Anderen muss der Kontext der Operation ein Punkt im Zustandsraum des Servers sein, da dieser, im Gegensatz zu *Jupiter*, nur einen Zustandsraum je Sitzung bereit hält, der für alle Instanzen verwendet wird. Mit der Bestätigung einer Operation durch den Server erklärt dieser, dass der Zustandsraum auf Serverseite durch Anwendung der Operation entsprechend angepasst und die Operation auch an alle anderen Instanzen verteilt wurde. Dadurch, dass ein Client nur eine Operation versenden darf und der Server die Reihenfolge nebenläufiger Operationen festlegt, die gleichzeitig allen Instanzen mitgeteilt wird, entsteht eine eindeutige Totalordnung der Operationen, weshalb *TP2* umgangen wird.

Um die Antwortzeit für einen lokalen Benutzer nicht zu beeinträchtigen können nebenläufig weitere Operationen auf dem Dokument angewendet worden

sein. Dieses Szenario ist in Abbildung 2.16 dargestellt. Das Ursprungsdokument $(0,0)$ enthält die Zeichenkette c. Client a erzeugt eine Operation O_a, die ein Zeichen t an das Ende des Dokuments anfügt ($retain(1); insertCharacters('t')$), und wendet diese auf dem lokalen Dokument an, was das Dokument in den Zustand $(1,0)$, mit dem Inhalt ct, überführt. O_a wird an den Server übertragen. Zwischenzeitlich hat ein zweiter Client b ebenfalls eine Operation O_b erzeugt, die ebenfalls ein Zeichen a an das Ende des Dokuments aus Zustand $(0,0)$ anfügt ($retain(1); insertCharacters('a')$), bereits vom Server bestätigt wurde und dessen Zustand auf $(0,1)$, mit dem Inhalt ca, überführt.

Sobald der Client O_b bzw. der Server O_a empfängt, werden diese auf das jeweils lokale Dokument angewendet. Würden diese Operationen jedoch direkt zur Anwendung kommen, würde sich der Inhalt auf Clientseite (cat) von dem Inhalt auf Serverseite (cta) unterscheiden. Daher müssen die beiden Operationen O_a und O_b mithilfe einer Transformationsfunktion (C. A. Ellis & Gibbs, 1989) zu O'_a ($retain(2); insertCharacters('t')$) bzw. O'_b ($retain(1)$; $insertCharacters('a')$; $retain(1)$) transformiert werden. Nach Anwendung dieser Operationen befinden sich beide Systeme im Zustand $(1,1)$ und dem Dokumenteninhalt cat.

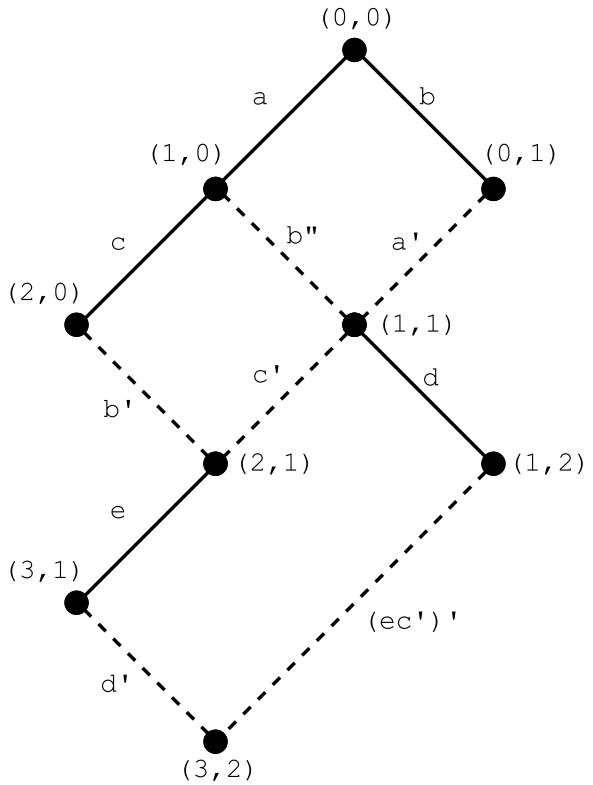

Abbildung 2.17: *Operational Transformation* Zustandsraum von Client und Server adaptiert von Spiewak (2010)

Dies ist der einzige Anwendungsfall, den Transformationsfunktionen selbstständig lösen können. Sobald der Zustand zwischen Client und Server um mehr als einen Schritt divergiert, muss zusätzlich ein Algorithmus zur Steuerung der Nebenläufigkeitsauflösung eingesetzt werden. Im Folgenden wird ein weiteres Szenario aufgezeigt (siehe Abbildung 2.17), bei dem die Zustände um mehrere Schritte divergieren und allgemeingültig für beliebige Situationen im *Wave*-Protokoll die Konflikte nebenläufiger Operationen auflöst (Spiewak, 2010).

Für den Steuerungsalgorithmus werden zwei zusätzliche Datenstrukturen auf dem Client vorgehalten:

Puffer
: Ein zu Beginn leerer Speicher, der lokal erzeugte Operationen zwischenspeichert, während der Client noch auf die Bestätigung einer zuvor an den Server übermittelten Operation wartet.

  Beim Erzeugen einer lokalen Operation wird diese mit den bereits im Puffer vorhandenen Operationen zusammengefasst.

  Bei Erhalt einer neuen Operation vom Server und der damit einhergehenden Aktualisierung des lokal verfolgten Serverzustands muss der Puffer mit der Serveroperation transformiert werden. Dafür müssen sowohl Serveroperation als auch Puffer denselben Anwendungskontext besitzen, was durch erneute Transformation der Serveroperation erreicht werden kann.

Brücke[31]
: Die Brücke stellt eine Operation dar, die das Dokument vom letzten (bekannten) Zustand des Servers in den aktuellen Clientzustand überführt. Diese kann dazu verwendet werden, eine erhaltene Serveroperation direkt zu transformieren und so lokal anwenden zu können. Die zweite, bei der Transformation erzeugte Operation ist gleichzeitig die neue Brücke.

Mit diesen Hilfsstrukturen lässt sich der in Abbildung 2.17 dargestellte Ablauf wie folgt beschreiben. Dabei werden alle durch andere Teilnehmer erzeugten Operationen aus Sicht des lokalen Clients vom Server erzeugt, womit sich das Beispiel als allgemeingültig für eine beliebige Anzahl an Teilnehmern erweist.

1. Sowohl Client und Server befinden sich zu Beginn in Zustand $(0, 0)$, woraufhin die beiden nebenläufigen Operationen O_a und O_b erzeugt und jeweils lokal angewendet werden. Der Client sendet O_a an den Server, sodass die nachfolgende Operation O_c in den lokalen Puffer gelegt wird. Der letzte bekannte Serverzustand ist $(0, 0)$, weshalb sich als Brücke $c \circ a$ ergibt.

[31] Bridge

2. Der Client empfängt O_b und die Serverseite wendet O_a an.

   Zur Anwendung von O_b auf dem Client muss diese zunächst mit der Brücke transformiert werden: $transform(c \circ a, b) = ((c \circ a)', b')$. Damit befindet sich der Client in Zustand $(2, 1)$ und die neue Brücke ist $(c \circ a)'$. Für die Aktualisierung des Puffers muss zunächst der Anwendungskontext von O_b und dem Puffer angeglichen werden, was mit der Transformation von a und b erfolgt und b'' ergibt. Diese Operation, transformiert mit dem aktuellen Puffer c, ergibt den neuen Pufferinhalt: c'.

   Auf Serverseite wird O_a nach Transformation mit O_b angewendet und a' zur Bestätigung an alle Clients verteilt.

3. Sowohl Client als auch Server erzeugen die nebenläufigen Operationen O_e respektive O_d und wenden diese auf den jeweils lokalen Zustand an.

   Der Client aktualisiert Puffer und Brücke, indem die neue Operation angehängt wird. Damit ergibt sich als neuer Pufferinhalt $e \circ c'$ und als Brücke $e \circ (c \circ a)'$.

4. Der Client empfängt a', also die Bestätigung einer zuvor lokal erzeugten Operation. Damit kann der aktuelle Pufferinhalt $e \circ c'$ an den Server übermittelt werden. Gleichzeitig wird die Brücke aktualisiert: $e \circ c'$

5. Der Client empfängt d. Der Puffer ist aktuell leer, weshalb keine Transformation erfolgen muss. Zur lokalen Anwendung von d muss diese zunächst mit der Brücke transformiert werden: $transform((e \circ c'), d) = ((e \circ c')', d')$. Die neue Brücke ist $(e \circ c')'$.

6. Der Server empfängt $e \circ c'$, transformiert diese mit d: $transform((e \circ c'), d) = ((e \circ c')', d')$. $(e \circ c')'$ wird angewendet und an die Clients verteilt.

7. Mit Erhalt von $(e \circ c')'$ befinden sich sowohl Client als auch Server im Zustand $(3, 2)$.

Neben der Effizienzsteigerung, die die Komposition von Operationen bei der Anwendung auf das Dokument mit sich bringt, ermöglicht ein durch paarweise Komposition von benachbarten Operationen erzeugter Kompositionsbaum die Traversierung zu jedem beliebigen Zustand in der Dokumentenhistorie sowie die Ermittlung der Unterschiede zwischen zwei Zuständen, durch Anwendung von $\mathcal{O}(\log n)$ Operationen (Whitelaw, Dan, Mah & Wang, 2009).

So zeigen die in Abbildung 2.18 hervorgehobenen Operationskompositionen etwa die anzuwendenden Operationen, um in der Historie eines Dokuments zwischen dem Zustand nach Anwendung von O_5, zum Zustand O_{14} zu springen.

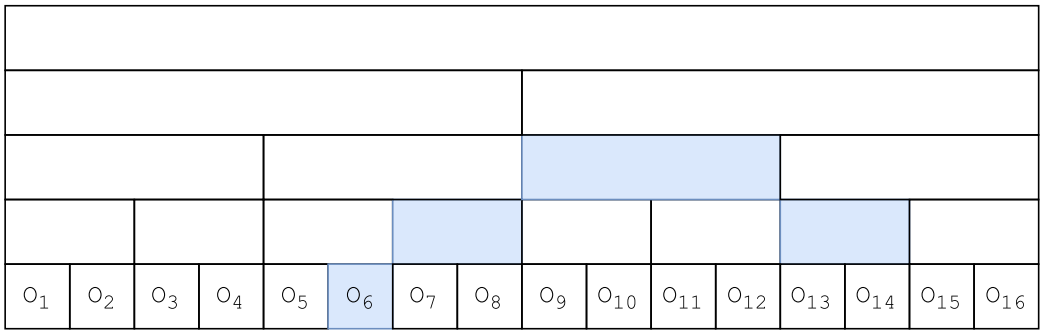

Abbildung 2.18: Kompositionbaum von Operationen in *Wave* zur Ermittlung der Dokumentenunterschiede zwischen O_5 und O_{14} adaptiert von Whitelaw, Dan, Mah und Wang (2009)

2.2.4 Evaluation

Nach der Vorstellung der drei Ansätze zur Konsistenzerhaltung in synchronen Groupwaresystemen in den vorangegangenen Kapiteln soll nun ein geeignetes Verfahren zur Erweiterung des *VisualEditors* um synchrone Kollaborationsfähigkeiten ausgewählt werden.

Die Bearbeitung eines Dokuments im *VisualEditor* erfolgt nicht auf Basis der serialisierten Form des Dokuments, sondern durch Anwendung von Operationen auf einem internen Datenmodell, das einer linearisierten Darstellung des Inhalts entspricht.

Der Einsatz differentieller Synchronisierung erzeugt daher einen erheblichen Mehraufwand, wenn das Dokument zu jedem Synchronisierungzeitpunkt aus dem internen Datenformat heraus exportiert wird und, aufgrund des zustandsbasierten Ansatzes, stets das gesamte Dokument zwischen Client und Server übertragen werden muss. Darüber hinaus müssen die auf der Serialisierung des *WOM* Dokuments basierenden extrahierten Änderungen zusätzlich in auf dem internen Datenmodell anwendbare Operationen übersetzt werden. All dies macht den Ansatz der differentiellen Synchronisation für den gegebenen Anwendungsfall zu einem aufwändigen Unterfangen.

Im Gegensatz dazu liegt den beiden anderen Konzepten, *CRDT* und *OT*, ein operationenbasierter Ansatz zugrunde, bei dem Änderungen am Dokument durch den Austausch von Operationen propagiert werden. Daher besteht weder die Notwendigkeit, den Dokumentenzustand wiederholt zu exportieren, noch die Änderungsoperationen zurück in das interne Datenformat übersetzen zu müssen.

Das in einem Großteil der wissenschaftlichen Arbeiten verwendete Dokumentenmodell sieht ein rein zeichenbasiertes Dokument vor, das zur Modifikation nur eine Einfüge- sowie Löschoperation bereitstellt. Für eine Anwendung im *VisualEditor*

müsste dieses jedoch, ähnlich dem in *Google Wave* verwendeten Modell, erweitert werden, um etwa hierarchische Elementstrukturen auf Basis einer linearisierten Darstellung zu unterstützten und die Modifikation elementspezifischer Attribute zu erlauben.

Dass dies grundsätzlich möglich ist, zeigen sowohl Martin u. a. (2010) durch die Anwendung von *CRDTs* für semistrukturierte XML Dokumente, sowie *Google Wave* (D. Wang u. a., 2010) auf Seite des *Operational Transformation* Ansatzes.

Ein für den Einsatz im *VisualEditor* geeignetes Verfahren stellt also etwa *TIPS* dar, ergänzt um ein ähnlich dem in *Google Wave* verwendeten Dokumentenmodell. Diesem liegt eine Client/Server Topologie zugrunde und vermeidet dadurch *TP2* erfüllende Transformationsfunktionen, was zusätzlich zur Verwendung des *CA* Frameworks des *ABT* Algorithmus (siehe Tabelle 2.1) eine Vereinfachung der Transformationsfunktionen ermöglicht. Zusätzlich bietet *TIPS* durch eine Erweiterung der aus *TIBOT* bekannten Zeitintervalle zur Erzeugung einer Totalordnung von Operationen eine geeignete Basis für den Einsatz in einer verteilten Umgebung wie dem Internet.

3 Architektur und Design

Im Folgenden wird die Softwarearchitektur der im Rahmen dieser Arbeit entwickelten Lösung zur Bearbeitung von *WOM* Dokumenten durch den Wikimedia *VisualEditor* vorgestellt. Dazu werden in Abschnitt 3.1 zunächst die mit dem System in Interaktion stehenden Akteure definiert sowie die Randbedingungen in Abschnitt 3.2 aufgezeigt, die bei der Konzeption beachtet werden mussten. Die für die Entwicklung relevanten Basistechnologien werden daraufhin in Kapitel 3.3 grundlegend vorgestellt, um die für das Design getroffenen Entscheidungen nachvollziehen zu können. Anschließend befasst sich Kapitel 3.4 mit der eigentlichen Architektur, deren Beschreibung durch verschiedene Sichten erfolgt. In Kapitel 3.5 werden abschließend die Ergebnisse dieser Arbeit präsentiert.

Die in diesem Kapitel vorgestellte Softwarearchitektur baut auf der in Haase (2014), der prototypischen Umsetzung eines visuellen Editors für das *Wiki Object Model*, zugrunde liegenden Architektur auf und wird an die neuen Voraussetzungen und Randbedingungen angepasst und erweitert.

3.1 Stakeholder

Bei der Konzeption und Entwicklung von Softwaresystemen gilt es insbesondere auf die Erwartungen und Absichten der verschiedenen, mit dem System in Aktion tretenden Akteure einzugehen und diese bei der Benutzung bestmöglich zu unterstützen. Daher werden im Folgenden die unterschiedlichen Akteure grob vorgestellt:

Benutzer Verwendet den visuellen Editor zum Bearbeiten von Dokumenten auf *Sweble Hub* und besitzt nur Kenntnisse in der Bedienung klassischer Textverarbeitungsprogramme. Kenntnisse über das zugrunde liegende *Wiki Object Model* können nicht vorausgesetzt werden. Das Bedienkonzept sollte sich entsprechend an diesen Erwartungen orientieren, um die Einstiegshürden möglichst gering zu halten.

Entwickler Führt Anpassungen am Programmcode des visuellen Editors durch, um diesen um neue Funktionalitäten zu erweitern, zur Fehlerbehebung oder um den Editor in eine externe Anwendung einzubinden. Dazu muss für die gesamte Architektur eine angemessene Dokumentation vorliegen, um sich ohne technologiespezifische Vorkenntnisse in das System einarbeiten zu können. Um diesen Prozess zu unterstützten sollte sich die Softwarearchitektur und der darauf basierende Programmcode an etablierten Prinzipien orientieren.

3.2 Randbedingungen

Für die Entwicklung der Editorkomponente gilt es die bereits im Vorfeld definierten Randbedingungen einzuhalten, die nachfolgend aufgeführt sind:

3.2.1 Organisatorische Randbedingungen

Zeitplan Die Implementierung des Gesamtsystems muss innerhalb des für eine Masterarbeit angesetzten Stundenkontingents umgesetzt werden.

Open Source Die Quelltexte und zugrundeliegende Software sollen es nach Abschluss der Masterarbeit ermöglichen die Editorkomponente in weiteren Projekten einsetzen zu können und sollen daher unter einer geeigneten Open Source Lizenz veröffentlicht werden.

3.2.2 Technische Randbedingungen

VisualEditor Als Basis für die visuelle Editorkomponente soll der in Haase (2014) nach Evaluation verschiedener Technologien ausgewählte *Wikimedia VisualEditor* dienen.

React Kompatibilität Der Editor soll sich problemlos in das auf dem *React* Framework[1] basierende *Sweble Hub* Web-Frontend einbinden lassen und mit diesem interagieren können.

Drittsoftware Lizenzen Das Sweble Projekt wird unter der *Apache License Version 2.0* veröffentlicht. Bei Verwendung von Drittsoftware muss diese zwingend unter einer dazu kompatiblen Open Source Lizenz stehen[2].

[1] https://facebook.github.io/react/
[2] http://apache.org/legal/resolved.html#category-a

Browserkompatibilität Die Editorkomponente soll in allen gängigen, desktopbasierten Webbrowsern lauffähig sein. Die Unterstützung mobiler Laufzeitumgebungen auf Smartphones oder Tablets ist im Rahmen dieser Arbeit zweitrangig zu betrachten.

3.3 Basistechnologien

Neben der eigentlichen Architektur ist auch ein grundlegendes Verständnis über die zugrunde liegenden Basistechnologien erforderlich, um die bei der Entwicklung getroffenen Entscheidungen nachvollziehen zu können. Daher wird in Abschnitt 3.3.1 die wesentliche Struktur von *WOM* Dokumenten erläutert und in Abschnitt 3.3.2 die Architektur des *VisualEditors* vorgestellt.

3.3.1 Wiki Object Model

Das Document Object Model (*DOM*) definiert eine plattform- und sprachunabhängige Schnittstelle zum Zugriff und zur Änderung des Inhalts und der Struktur von Dokumenten (Le Hors, Arnaud et al., 2004). *DOM* hat besondere Bedeutung erlangt, da es etwa die primäre Methode zur Interaktion mit den Elementen einer im Webbrowser gerenderten HTML (Berjon, Robin et al., 2014) Seite ist. Die HTML Elemente werden darin als Knoten in einem Baum, dem sog. DOM-Baum, repräsentiert, die jeweils einen Eltern- sowie eine beliebige Anzahl an Kindknoten besitzen können. Mit dem Wiki Object Model (*WOM*) wird dieses Konzept für die Interaktion von Wiki-Dokumenten erweitert (Dohrn & Riehle, 2011). Das *WOM* ist dazu ausgelegt, von der zugrunde liegenden Auszeichnungssprache zu abstrahieren und eine generische Schnittstelle anzubieten, unabhängig von der verwendeten Wiki-Software.

In Listing 3.1 wird ein einfaches Wikitext Dokument aufgezeigt, auf dessen Grundlage im Folgenden die weitere Beschreibung des *WOM* erfolgt.

```
1 =Heading =
2 Paragraph
```

Listing 3.1: Wikitext Dokument

Der daraus resultierende WOM-Baum ist in seiner als XML serialisierten Form Listing 3.2 zu entnehmen.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <article xmlns="http://sweble.org/schema/wom30"
3           xmlns:mww="http://sweble.org/schema/mww30"
4           version="3.0"
5           title="Foo">
6     <body>
7        <section level="1">
8           <heading>
9              <rtd>=</rtd>
10             <text>Heading </text>
11             <rtd>= </rtd>
12          </heading>
13          <body>
14             <text>
15 </text>
16             <p topgap="0" bottomgap="0">
17                <text>Paragraph</text>
18             </p>
19             <text>
20 </text>
21          </body>
22       </section>
23    </body>
24 </article>
```

Listing 3.2: XML Serialisierung eines WOM-Baumes

Nach der Bearbeitung eines Dokuments im *WOM* kann es notwendig sein, das Dokument wieder in seine Darstellung in der ursprünglich verwendeten Auszeichnungssprache (z.B. Wikitext) zurück zu übersetzen. Sofern keine Änderungen am *WOM*-Dokument vorgenommen wurden, soll am Ende dieses als *round-trip* bezeichneten Vorgangs idealerweise eine zum Quelldokument identische Ausgabe erzeugt werden. Finden sich dagegen Unterschiede an eigentlich unveränderten Stellen im Dokument, so spricht man von einem *dirty-diff*[3]. Dies wirkt sich zum Einen negativ auf die Akzeptanz aus, erschwert vor allem aber auch den Vergleich zweier Dokumentenzustände.

Das *WOM* bedient sich hierfür sog. *Round-Trip-Data* (*RTD*) Elemente. Diese enthalten die im Quelldokument verwendeten spezifischen Tags in der verwendeten Auszeichnungssprache (vgl. Zeile 9 und 11), um eine identische Rücktransformation gewährleisten zu können.

[3] https://www.mediawiki.org/wiki/VisualEditor/Design/Software_overview#Dirty-diffs

3.3.2 Visueller Editor

Als Basis für den visuellen Editor dient der in Haase (2014) ausgewählte *VisualEditor* der Wikimedia Foundation. Die Entwicklung wurde 2011 begonnen, mit dem Ziel, eine benutzer- und vor allem einsteigerfreundlichere Möglichkeit zur Bearbeitung von Artikeln in der Wikipedia bzw. der zugrunde liegenden Wiki-Software *MediaWiki* anbieten zu können.

Die nicht *MediaWiki* spezifischen Teile des Editors wurden dabei in ein separates Projekt ausgegliedert und unter der MIT Lizenz veröffentlicht. Dieses dient als Grundlage der *MediaWiki* Erweiterung, erlaubt aber auch den Einsatz der Editorkomponente in externen Projekten.

Im Gegensatz zur vorhergehenden Evaluation ist der Editor nun in aktuellen Versionen browserübergreifend einsetzbar und erlaubt inzwischen auch die Verwendung des *Internet Explorers*.[4]

Technologisch baut der Editor auf dem in HTML5 eingeführten *contentEditable* auf, was grundsätzlich die direkte Bearbeitung von Teilen einer Website im Browser durch den Benutzer erlaubt. Aufgrund verschiedener Defizite und Inkompatibilitäten zwischen den verschiedenen Webbrowsern schränkt der *VisualEditor* die direkte Bearbeitung jedoch teils stark ein oder greift stattdessen selbst korrigierend ein[5].

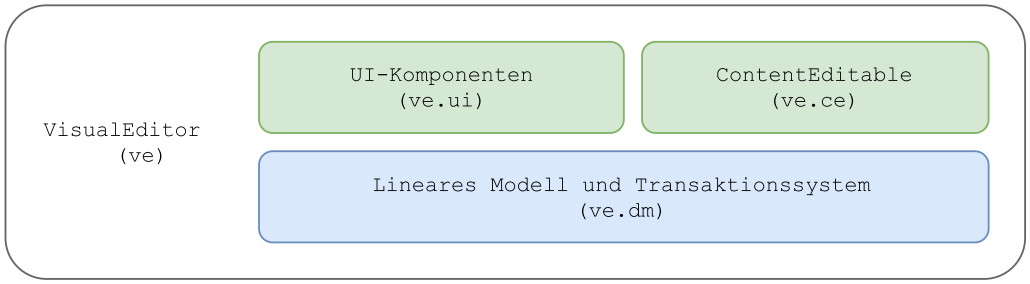

Abbildung 3.1: Architektur des *VisualEditors*

Die Architektur des *VisualEditors* lässt sich, wie in Abbildung 3.1 dargestellt, in die drei folgenden Komponenten unterteilen[6]:

[4] https://www.mediawiki.org/wiki/VisualEditor/Target_browser_matrix

[5] https://www.mediawiki.org/wiki/VisualEditor/Design/Software_overview#Data_Structures

[6] https://www.mediawiki.org/wiki/VisualEditor/Design/Software_overview#Architecture

ve.dm Das Dokumentenmodell des *VisualEditors* unterscheidet grundsätzlich zwischen Elementen und Inhalt[7]. Elemente dienen zur Strukturierung des Dokuments und können entweder weitere Elemente oder Inhalt enthalten. Inhalt kann wiederum keine weiteren Kindknoten enthalten. Hierbei spielt insbesondere das Konzept von Annotationen ein besondere Rolle, die dazu dienen, textuellen Inhalt zu beschreiben oder dessen Erscheinungsbild zu verändern (bspw. **bold**, *italics*).

Alle Operationen arbeiten auf der Grundlage einer linearisierten Darstellung des zu bearbeitenden Dokuments. Beim Laden eines Dokuments wird dieses zunächst in ein lineares Modell überführt, das aus einer Liste besteht. Die Einträge stellen einzelne Zeichen dar, oder, im Falle von Elementen, Markierungen für den Beginn bzw. das Ende eines Elements. Dies ermöglicht es, jedem Eintrag in der Liste einen eindeutigen Index zuzuweisen.

Neben dem linearen Modell wird ein korrespondierender Baum aus Knoten gepflegt, der der ursprünglichen Baumstruktur des Dokuments entspricht. Jeder Knoten in diesem Baum enthält als Wert seine Länge im linearen Modell, um die Position des durch ihn repräsentierten Elements in der linearen Darstellung berechnen zu können.

Sowohl das lineare Modell als auch der zugehörige Baum implementieren das Observer Pattern, um extern über Änderungen benachrichtigt werden zu können.

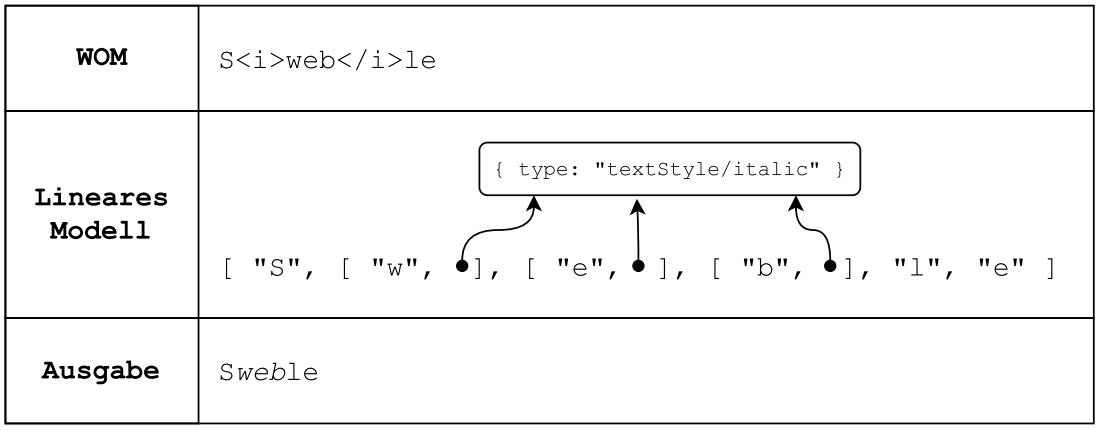

Abbildung 3.2: Lineares Datenmodell des *VisualEditors*

Ein einzelnes Zeichen kann entweder direkt in die Liste eingetragen werden, oder als Array, wobei das erste Element darin das Zeichen selbst ist, gefolgt von einer beliebigen Anzahl an Referenzen auf Annotationen, die bei der

[7] https://www.mediawiki.org/wiki/VisualEditor/Design/Software_overview#Document_model

Darstellung auf das Zeichen angewendet werden. In Abbildung 3.2 ist eine beispielhafte Gegenüberstellung eines *WOM* Inhalts, seiner korrespondieren linearen Darstellung und der daraus generierten Ausgabe dargestellt.

Zur Modifikation des Dokumentenmodells durch den Benutzer werden dessen Eingaben, etwa mittels Maus oder Tastatur, in Transaktionen gekapselt, die selbst wiederum aus mehreren Operationen bestehen können[8]. Eine Transaktion ist umkehrbar, bezieht sich aber immer auf einen bestimmten Dokumentenzustand, weshalb alle Transaktionen in vorgegebener Reihenfolge und vollständig ausgeführt werden müssen, bevor eine neue Transaktion angewendet werden kann.

Der *VisualEditor* unterscheidet vier verschiedene Transaktionstypen: (1) *Insert*, um Inhalt oder Elemente einzufügen, (2) *Remove*, zum Löschen von Inhalt oder Elementen, (3) *Annotate Content*, um Annotationen auf textuellem Inhalt zu modifizieren, und (4) *Change Element Attributes*, um Attribute eines Elements zu ändern.

Transaktionen können aus den folgenden Operationen zusammengesetzt werden: (1) *Retain*, (2) *Insert*, (3) *Remove*, (4) *Start Annotating*, (5) *Stop Annotating* und (6) *Change Element Attributes*. Eine Transaktion muss dabei stets das gesamte lineare Modell umfassen, der Cursor bei einer sequentiellen Abarbeitung der Liste also nach Abschluss der Transaktion am Ende der Liste stehen. Um Listeneinträge zu überspringen, also nicht zu modifizieren, kann die *Retain* Operation verwendet werden.

ve.ce Im *VisualEditor* ist die *ve.ce* Komponente für die eigentliche Darstellung im Webbrowser des Benutzers zuständig. Diese erzeugt aus dem linearen Modell unter Verwendung des *DOM* eine dem jeweiligen Element entsprechende Darstellung (View). Die verschiedenen Views bilden dabei, dem Composite Pattern entsprechend, einem Baum (Document View). Dieser beobachtet Änderungen des Dokumentenmodells und wird entsprechend synchronisiert. Für die erzeugten Elemente kann das *contentEditable* Feature aktiviert werden, um eine native Bearbeitung im Browser zu ermöglichen. Dabei werden jegliche Änderungen am *DOM* durch entsprechende *EventListener* überwacht, um das Dokumentenmodell im Falle von durchgeführten Änderungen aktualisieren zu können. Daneben wird dieser Mechanismus auch dazu verwendet, um unzulässige Änderungen zu unterbinden bzw. rückgängig machen zu können.

ve.ui *ve.ui* ist für die eigentliche Benutzerschnittstelle zuständig, um überhaupt Änderungen (Texteingabe mittels Tastatur ausgenommen) am Dokument vornehmen zu können. Dazu stellt *ve.ui*, die in Abbildung 3.3 dargestellten Komponenten bereit: erweiterbare *Toolbars*, *ContextItems* und *Inspectors*.

[8] https://www.mediawiki.org/wiki/VisualEditor/Design/Software_overview#Transactions

Eine *Toolbar* ist aus klassischen Textverarbeitungsprogrammen bekannt und enthält bspw. Schaltflächen, um bestimmte Aktionen ausführen zu können. Ein *ContextItem* wird dagegen direkt über dem aktuell markierten Element eingeblendet und kann zusätzliche Informationen zu dem selektieren Element anzeigen, wie bspw. das Linkziel. *Inspectors* bieten darüber hinaus die Möglichkeit ein selektiertes Element direkt bearbeiten zu können, etwa die Änderung des Linkziels. Darüber hinaus können auch beliebig komplexe Dialogfenster erstellt werden.

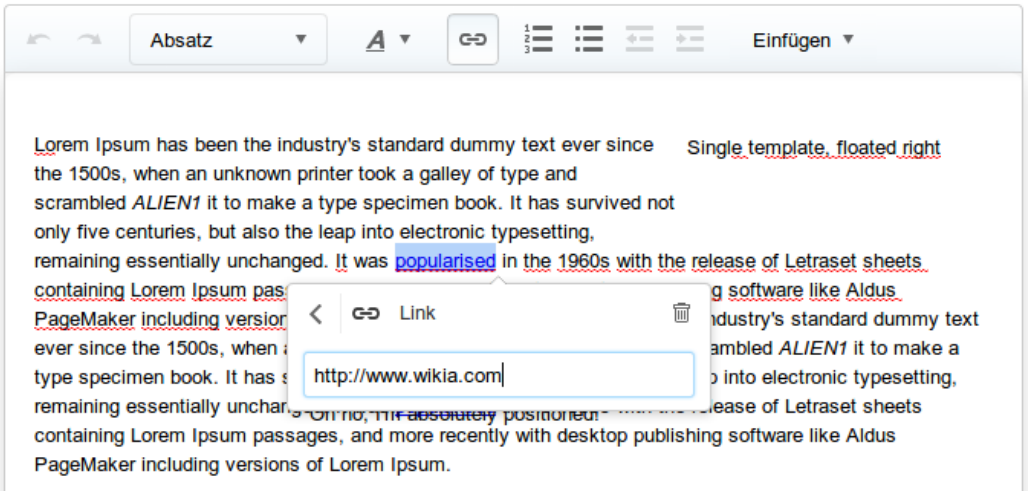

Abbildung 3.3: UI Komponenten im *VisualEditor*

Die Interaktion zwischen den einzelnen Komponenten entspricht dabei dem MVC[9] Pattern. In Abbildung 3.4 ist der Nachrichtenfluss der Komponenten dargestellt. Als *Surface* wird der vom *VisualEditor* verwendete Bereich der Webseite bezeichnet, in den die Editorkomponente gerendert wird. Dieser überwacht die vom Benutzer getätigten Eingaben und fungiert als *Controller*. Sobald ein Eingabeevent registriert wird, wird eine entsprechende Transaktion erzeugt, um das Dokumentenmodell (Model) zu aktualisieren. Dieses wiederum benachrichtigt den *Document View* (View) über die Änderungen, um daraufhin die gerenderten Views durch Modifikationen des *DOM* entsprechend anzupassen.

[9]Model View Controller

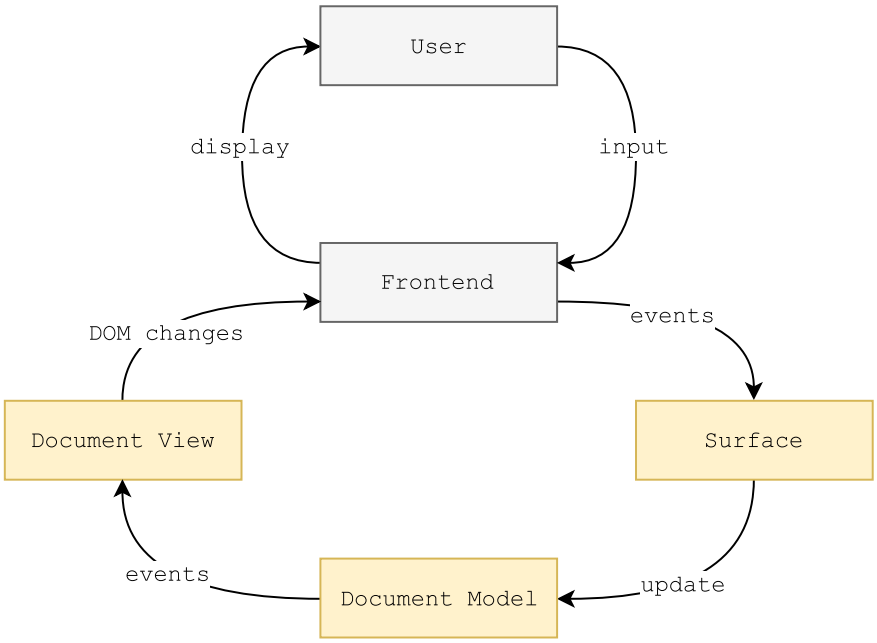

Abbildung 3.4: Interaktionen der verschiedenen Komponenten im *VisualEditor*

3.4 Softwarearchitektur

Die Architektur der entwickelten Lösung wird im Folgenden durch verschiedene Sichten beschrieben, um die jeweils unterschiedlichen Aspekte hervorzuheben.

3.4.1 Kontextabgrenzung

Um einen ersten Überblick über die in Interaktion stehenden Akteure und Systeme zu erhalten, werden zunächst die Zusammenhänge der einzelnen Komponenten vorgestellt.

Fachlicher Kontext

Der visuelle Editor steht, wie in Abbildung 3.5 dargestellt, in direkter Interaktion zum Einen mit dem am Dokument Änderungen vornehmenden *Benutzer*, sowie einem als Dokumentspeicher dienenden *WOM3 Backend*.

Benutzer Eine Instanz entsprechend der in Abschnitt 3.1 aufgeführten Akteure. Dem Benutzer steht eine grafische Benutzeroberfläche innerhalb einer Browserumgebung auf einem Endgerät seiner Wahl zur Verfügung. Die

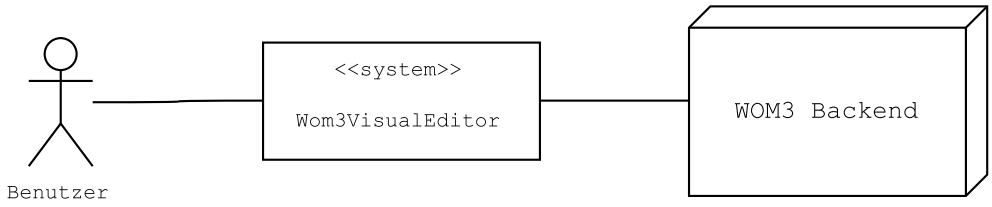

Abbildung 3.5: Fachlicher Kontext

Interaktion mit dem Editor erfolgt über Standardeingabegeräte wie etwa Maus und Tastatur.

WOM3 Backend Das *WOM3 Backend* dient als persistenter Speicher und Datenquelle für zu bearbeitende *WOM* Dokumente.

Technischer Kontext

Die konkreten technischen Systeme, mit denen der visuelle Editor in Verbindung steht, werden in Abbildung 3.6 aufgeführt.

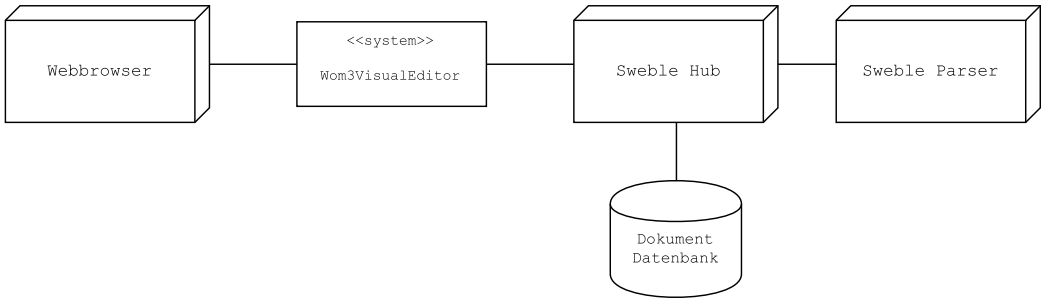

Abbildung 3.6: Technischer Kontext

Webbrowser Der Webbrowser stellt die Laufzeitumgebung für das Frontend dar, der auf einem Endgerät des Benutzers ausgeführt wird und über den die gesamte Interaktion mit dem Editor abläuft. Als Grundvoraussetzungen für den fehlerfreien Betrieb gelten HTML5-Unterstützung sowie aktiviertes JavaScript.

Sweble Parser Der Sweble Parser ist dafür zuständig, ein in einer externen Auszeichnungssprache (bspw. Wikitext[10]) vorliegendes Dokument in ein korrespondierendes *WOM* Dokument zu überführen, auf dem die Bearbeitung innerhalb des Editors erfolgen kann. Darüber hinaus kann das *WOM* Dokument nach abgeschlossener Bearbeitung zurück in die Auszeichnungssprache übersetzt werden.

[10]https://en.wikipedia.org/wiki/Help:Wiki_markup

Sweble Hub *Sweble Hub* dient als Kollaborationsplattform zur Bearbeitung von Dokumenten auf Basis des *Wiki Object Models*, in die die Editorkomponente eingebettet werden soll. Die auf Webtechnologien basierende Plattform stellt dazu Schnittstellen bereit, um etwa *WOM* Dokumente aus dem Editor heraus abfragen und speichern zu können.

3.4.2 Bausteinsicht

Die Architektur des visuellen Editors besteht aus zwei separaten Komponenten:

Frontend

Über das Frontend stehen die eigentlichen Interaktionsmöglichkeiten für den Benutzer mit dem zu bearbeitenden Dokument zur Verfügung. Die durch den Benutzer getätigten Eingaben werden durch das Frontend entgegen genommen und das Dokument entsprechend modifiziert. Die Basis für diese Komponente stellt der *Wikimedia VisualEditor* dar, der für die Unterstützung des *Wiki Object Models* erweitert wird.

Backend

Das Backend stellt eine Schnittstelle in Form einer *REST-API* für die Verwendung durch das Frontend bereit. Hierüber können *WOM3* Dokumente geladen und gespeichert, sowie Snapshots des aktuellen Dokumentenzustands persistiert werden.

Im Rahmen dieser Arbeit wird das Backend als separate Komponente eingesetzt. Bei der späteren Integration des Editors in *Sweble Hub* kann die Funktionalität der *REST-API* auch in eine bereits vorhandene Komponente integriert werden.

3.4.3 Laufzeitsicht

Um eine Vorstellung für das dynamische Zusammenspiel der Komponenten zu erhalten, wird im Folgenden der Ablauf einer standardmäßigen Interaktion mit dem Editor aufgezeigt und in Abbildung 3.7 als Sequenzdiagramm dargestellt.

1. Der Benutzer wählt über die als Website erreichbare *Sweble Hub* Plattform das zu bearbeitende Dokument aus bzw. erstellt ein neues Dokument. Daraufhin wird eine Webseite geladen, die alle von der Editorkomponente

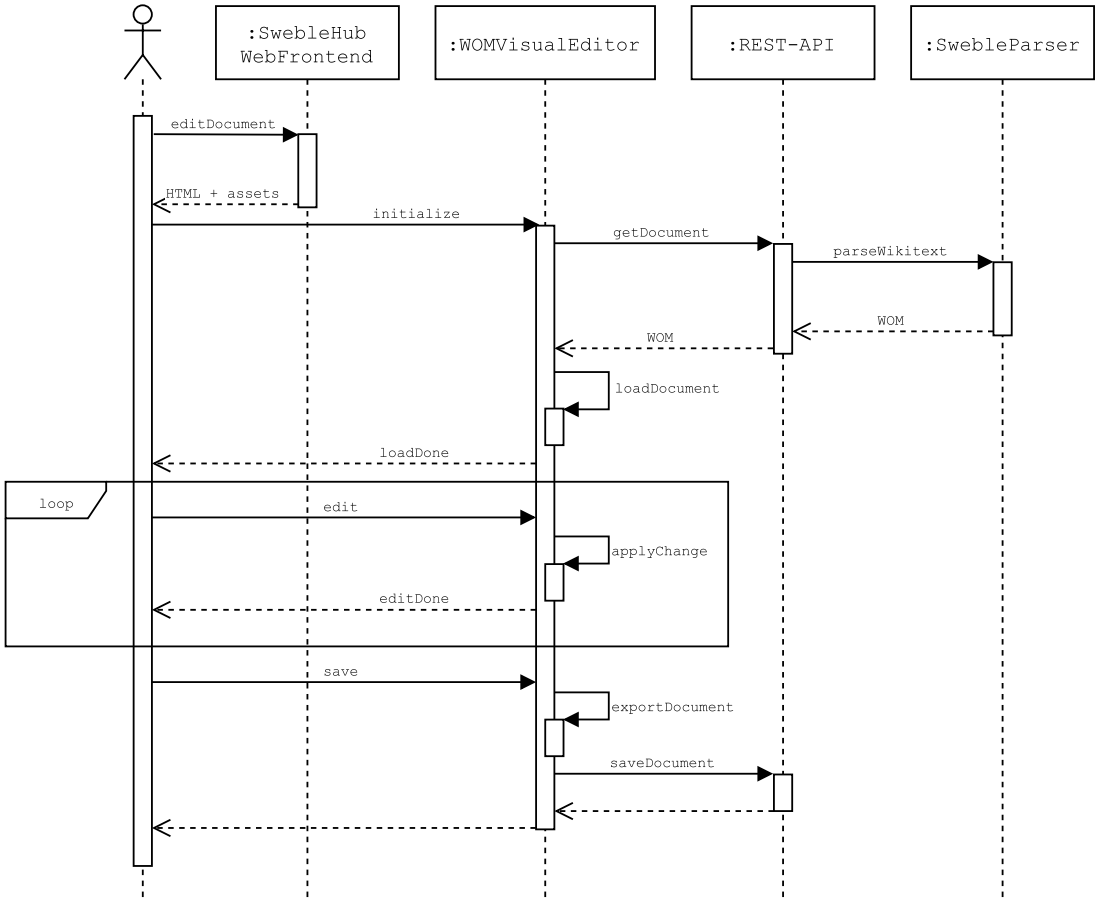

Abbildung 3.7: Sequenzdiagramm einer Interaktion mit dem visuellen Editor

benötigten statischen Inhalte einbindet, wie etwa JS-, CSS- oder Bilddateien. Nachdem der Ladevorgang abgeschlossen ist, wird der JavaScript Programmcode zur Ausführung gebracht und der Editor gestartet

2. Nach abgeschlossener Initialisierungsphase des Editors wird das zu bearbeitende *WOM* Dokument in seiner XML Repräsentation über eine REST-API abgefragt. Falls das Dokument bereits als *WOM* vorliegt, kann dieses direkt ausgeliefert werden. Für den Fall, dass das Dokument aus einer externen Quelle stammt, muss das Dokument vor der Auslieferung zunächst unter Verwendung des *Sweble Parsers* in ein *WOM* Dokument konvertiert werden

3. Das zu bearbeitende Dokument wird in den Editor geladen. Dieser generiert auf Basis des Dokuments eine durch den Benutzer im Browser bearbeitbare HTML-Repräsentation

4. Nach erfolgter Bearbeitung kann das Dokument gespeichert werden. Dazu exportiert der Editor zunächst den Zustand des aktuellen Dokuments in

Form einer *WOM* XML Darstellung. Diese wird daraufhin mittels HTTP-POST Anfrage an die REST-API übermittelt, um anschließend auf Backendseite persistiert zu werden und weiter verarbeitet werden zu können, um bspw. durchgeführte Änderungen zu extrahieren

3.4.4 Verteilungssicht

Die einzelnen Komponenten werden entsprechend der in Abbildung 3.8 aufgeführten Darstellung auf die verschiedenen Systeme verteilt.

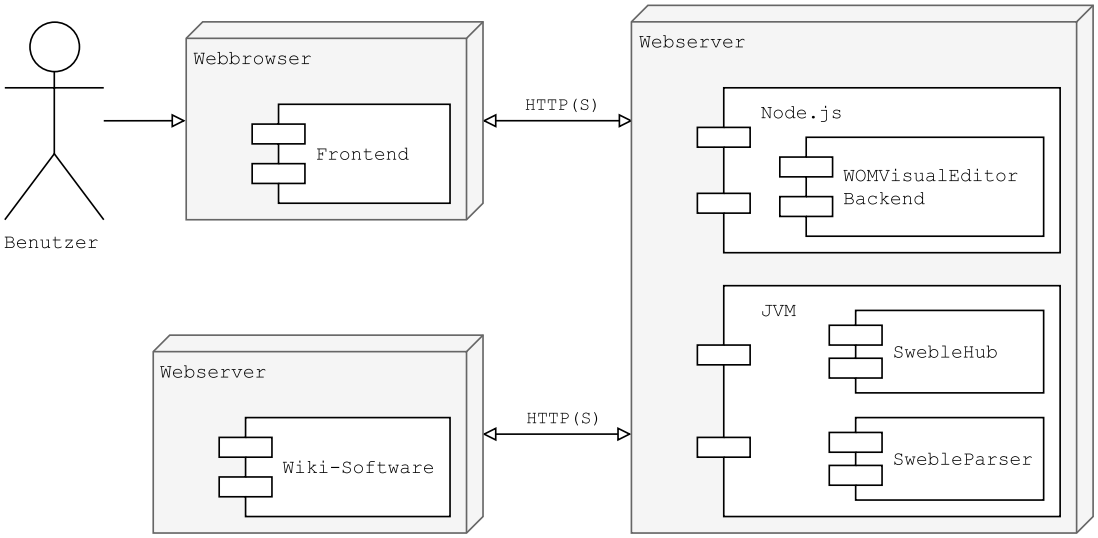

Abbildung 3.8: Verteilungssicht des Gesamtsystems

Auf einem lokalen Endgerät des Benutzers wird ein Webbrowser ausgeführt, der über eine JavaScript Laufzeitumgebung verfügt und für die Ausführung und Interaktion des eigentlichen visuellen Editors verwendet wird.

Auf Serverseite wird eine JVM Laufzeitumgebung zur Ausführung der *Sweble Hub* Komponenten verwendet, die Funktionen mittels einer HTTP(S) erreichbaren Schnittstelle bereit stellen. Daneben wird eine *Node.js*[11] Umgebung eingesetzt, um die im Rahmen dieser Arbeit entwickelte Backendkomponente auszuführen.

Das Abfragen externer Datenquellen, wie etwa Artikeln aus einer externen *MediaWiki* Instanz kann ebenfalls über eine via HTTP(S) erreichbare Schnittstelle erfolgen.

[11] https://nodejs.org/

3.5 Ergebnisse

Im Folgenden werde ich die im Rahmen dieser Arbeit entwickelte Lösung zur Bearbeitung von *WOM* Dokumenten mithilfe des *VisualEditors* vorstellen. Dabei ist zwischen der Frontend- und Backendkomponente zu unterscheiden. Das Backend ist hierbei als *PoC*[12] anzusehen, dessen Funktionalität zukünftig auch direkt im *Sweble Hub* Backend integriert werden kann.

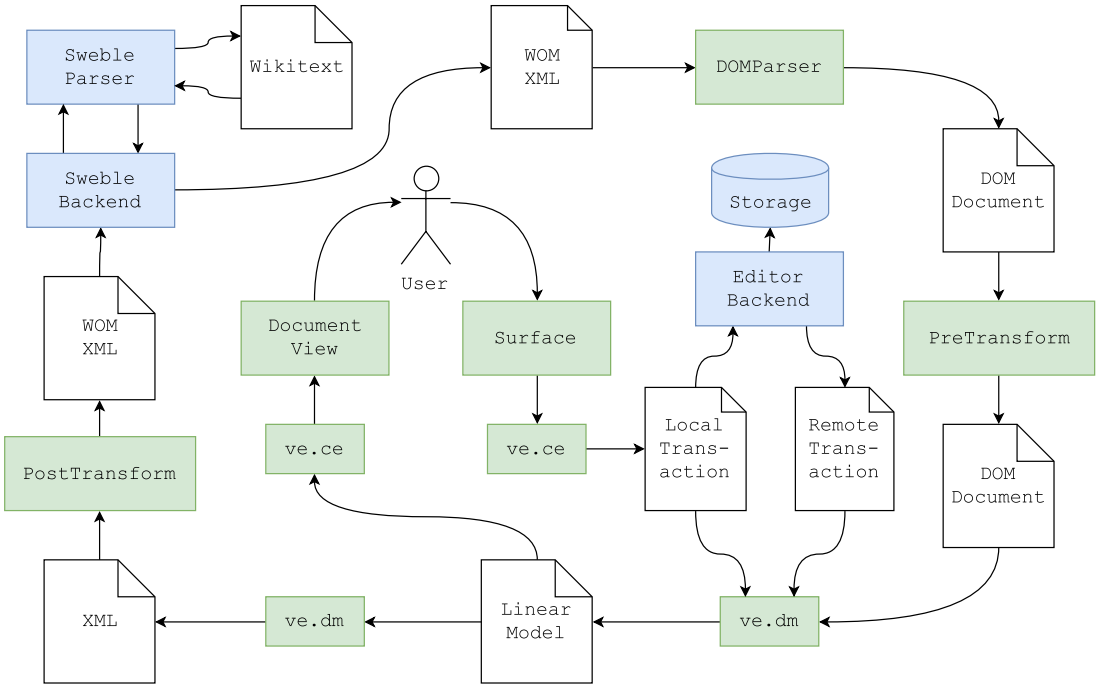

Abbildung 3.9: Datenfluss bei der Bearbeitung von WOM Dokumenten im *VisualEditor*. Komponenten im Frontend sind grün markiert, im Backend blau.

In Abbildung 3.9 wird das Zusammenspiel der verschiedenen Komponenten und ihrer ausgetauschten Daten dargestellt.

3.5.1 Backend

Auf Seite des Backends muss zwischen zwei Komponenten unterschieden werden: (1) *SwebleBackend* und (2) *EditorBackend*. Das *SwebleBackend* stand bereits im Vorfeld dieser Arbeit zu Testzwecken zur Verfügung und stellt eine REST-API zum Abfragen von *WOM* Dokumenten bereit. Als Datenbasis dient ein lokales Git Repository mit Wikitext Dokumenten, die vom *SwebleBackend* und dem darin enthaltenen *Sweble Parser* in ein *WOM* Dokument überführt werden können.

[12] Proof of Concept

Das *EditorBackend* ist dagegen eine im Rahmen dieser Arbeit erstellte Komponente, die für das Empfangen und Weiterleiten von Transaktionen verwendet wird. Darüber hinaus besteht die Möglichkeit, empfangene Transaktionen zu speichern, um die durch einen Benutzer durchgeführten Änderungen nachverfolgen zu können.

3.5.2 Frontend

Auf Clientseite wird das empfangene *WOM-XML* Dokument zunächst durch den im Webbrowser nativ vorhandenen *DOMParser*[13] in ein *DOM* Dokument überführt. Im Gegensatz zu Haase (2014) wird nun nicht mehr die JSON Serialisierung des *WOM*, sondern direkt XML verwendet, was den ehemals serverseitig durchgeführten Konvertierungsschritt von JSON zu HTML erspart.

Das erhaltene *DOM* Dokument könnte nun direkt in die *ve.dm* Komponente des *VisualEditors* geladen werden. Stattdessen wird jedoch ein weiterer *PreTransform* Schritt vorangestellt, der die Struktur des Dokuments modifiziert, um es der vom *VisualEditor* erwarteten *HTML* Struktur anzunähern. Dies bietet den Vorteil, die notwendigen Änderungen bzw. Erweiterungen am *VisualEditor* zu reduzieren und dadurch das spätere Einspielen von Updates zu vereinfachen. Als Beispiel für eine solche Transformation sei hier etwa die Extraktion von *RTD* Elementen zu nennen, da diese während der Bearbeitung keine Rolle spielen. Für eine ausführlichere Beschreibung verweise ich auf Kapitel 4.

Die *ve.dm* Komponente traversiert das *DOM* Dokument und generiert daraus ein entsprechendes lineares Modell zur weiteren Verwendung im *VisualEditor*. Nach einer erfolgten Bearbeitung kann ebenfalls die *ve.dm* Komponente dazu verwendet werden, das lineare Modell als XML Dokument zu exportieren.

Das XML Dokument wird anschließend einem *PostTransform* Schritt unterzogen, analog dem *PreTransform* Schritt zu Beginn, um ein valides *WOM-XML* Dokument zu erhalten, das die zuvor extrahierten *RTD* Elemente beinhaltet.

Mithilfe des *SwebleParsers* kann das *WOM* Dokument bei Bedarf wieder zurück in die ursprüngliche Auszeichnungssprache (z.B. Wikitext) übersetzt werden.

Die Transaktionen des *VisualEditors* werden parallel zur lokalen Ausführung in *ve.dm* auch an das *EditorBackend* übermittelt, um sie dort wahlweise speichern oder an andere Editorinstanzen weiterleiten zu können.

Zur vollständigen Unterstützung des *WOM* im *VisualEditor* mussten alle vorhandenen Elemente erweitert bzw. neue, bisher unbekannte Elemente erstellt werden. Dafür bedarf es zum Einen der Erweiterung von *ve.dm*, um zu beschreiben,

[13] https://developer.mozilla.org/de/docs/Web/API/DOMParser

wie Elemente in das interne Datenformat abgebildet werden sollen, sowie dem umgekehrten Prozess, ein Element im linearen Modell zurück in einen Teilbaum des *DOM* zu überführen. Und zum Anderen einer Erweiterung der *ve.ce* Komponente, um aus dem Inhalt des linearen Modells, im Browser darzustellende bzw. bearbeitbare Elemente zu generieren.

Darüber hinaus wurde die *ve.ui* Komponente erweitert, um alle *WOM* Elemente über Dialogfenster bzw. *Inspectors* modifizieren zu können, etwa zur Bearbeitung von Elementattributen.

4 Implementierung

Im voran gegangenen Kapitel wurde die Architektur der im Rahmen dieser Arbeit entwickelten Lösung vorgestellt. Im Folgenden sollen nun konkrete Implementierungsdetails der einzelnen Komponenten betrachtet werden.

4.1 Backend

Auf Serverseite ist zwischen den beiden Komponenten *Sweble Backend* und *Editor Backend* zu unterscheiden. Das *Sweble Backend* ist eine auf Java basierende Serverkomponente, die eine REST-API zur Abfrage von *WOM* Dokumenten bereit stellt. Diese Komponente stand bereits zu Beginn dieser Arbeit zur Verfügung und wurde um Funktionalitäten erweitert. Bei dem *Editor Backend* handelt es sich um eine auf *Node.js* aufbauende Serverkomponente, die im Rahmen dieser Arbeit entwickelt wurde.

4.1.1 Sweble Backend

Das vorgegebene *Sweble Backend* entsprach bereits weitgehend den Anforderungen, um beliebige in Wikitext vorliegende Dokumente mithilfe des *Sweble Parsers* in *WOM* Dokumente zu überführen und als JSON serialisiert über eine REST-API abfragen zu können.

Da die neue Editorkomponente jedoch *WOM* Dokumente auf Basis ihrer XML Serialisierung bearbeitet, musste ein neuer REST Endpoint in der Klasse `XML-ArticleResource` (siehe Listing 4.1) erstellt werden, der den `WomSerializer` mit angepasstem `SerializationFormat` aufruft (Zeile 7).

Darüber hinaus erfordert der zum späteren Einlesen des *WOM* Dokuments auf Clientseite im Webbrowser Firefox verwendete *DOMParser* ein XML Dokument der Version `1.0`. Das ausgegebene XML Dokument des `WomSerializers` basiert jedoch auf Version `1.1`, was beim Versuch des Einlesens zu einer Exception führt.

Mangels Konfigurationsmöglichkeiten des *WomSerializes* wird die Versionsangabe daher aktuell in der ausgegebenen Zeichenkette einfach ersetzt (Zeile 13), da ohnehin keine XML `1.1` Features verwendet werden.

```
1  @Path("/article_xml")
2  @Produces(MediaType.APPLICATION_XML)
3  public class XMLArticleResource extends ArticleResource {
4      ...
5    protected String serializeWom(Wom3Document wom) throws
         IOException, SerializationException {
6      WomSerializer womSerializer = new WomSerializer();
7      SerializationFormat format = WomSerializer.
           SerializationFormat.XML;
8      boolean compact = true;
9      boolean pretty = true;
10     byte[] bytes = womSerializer.serialize(wom, format, compact,
           pretty);
11
12     String output = new String(bytes, "UTF8");
13     output = output.replace("<?xml version=\"1.1\"", "<?xml
           version=\"1.0\"");
14
15     return output;
16   }
17 }
```

Listing 4.1: *WOM* XML Endpoint (`editor-backend/src/main/java/org/sweble/student_thesis/ve3/XMLArticleResource.java`)

Da die verschiedenen Komponenten in der Entwicklung nicht unterhalb derselben Domain erreichbar sind, sondern sich i.d.R. lokal an unterschiedliche Ports binden, muss aufgrund der *Same Origin Policy*[1] der Zugriff mittels *CORS*[2] (Anne van Kesteren, 2014) explizit erlaubt werden (siehe Listing 4.2).

```
1  private void configureCors(Environment environment) {
2      ...
3      filter.setInitParameter(CrossOriginFilter.
           ALLOWED_METHODS_PARAM, "GET,PUT,POST,DELETE,OPTIONS");
4      filter.setInitParameter(CrossOriginFilter.
           ALLOWED_ORIGINS_PARAM, "*");
5      ...
6  }
```

Listing 4.2: *CORS* Filter Konfiguration (`editor-backend/src/main/java/org/sweble/student_thesis/ve3/EditorBackendApplication.java`)

[1] https://www.w3.org/Security/wiki/Same_Origin_Policy
[2] Cross-Origin Resource Sharing

4.1.2 Editor Backend

Das *Editor Backend* dient dazu, von einer *VisualEditor* Instanz erzeugte Transaktionen bzw. Snapshots des aktuellen Dokumentenzustands zu empfangen und an andere verbundene Instanzen zu verteilen bzw. serverseitig zu persistieren.

Diese Komponente wurde zunächst als eigenständige *Node.js* Anwendung in JavaScript implementiert und anschließend in die vorhandene React Anwendung (siehe Abschnitt 4.2.1) integriert, da für diese ohnehin eine laufende *Node.js* Umgebung erforderlich ist.

Zur Behandlung von Transaktionen wird die *socket.io*[3] Bibliothek eingesetzt, um eine bidirektionale Kommunikation zwischen Client und Server zu ermöglichen. In dem als *PoC* anzusehenden *Editor Backend* wird angenommen, dass alle verbundenen Instanzen innerhalb einer Sitzung zur Bearbeitung desselben Dokuments arbeiten. Listing 4.3 zeigt den zugehörigen Programmcode zur Weiterleitung von empfangenen Transaktionen. In Zeile 6 wird ein *EventListener* für den Empfang von Transaktionen definiert, der diese anschließend an die Teilnehmer der Sitzung weiterleitet (Zeile 10).

```
1  const io = require('socket.io')(server);
2
3  io.on('connection', (socket) => {
4    console.log('a user connected');
5
6    socket.on('transaction', (tx) => {
7      console.log('transaction received');
8      console.dir(tx);
9
10     socket.broadcast.emit('transaction', tx);
11   });
12
13   socket.on('disconnect', () => {
14     console.log('user disconnected');
15   });
16 });
```

Listing 4.3: Empfang und Verteilen von *VisualEditor* Transaktionen (`editor-frontend/app/server/server.js`)

Snapshots werden dagegen mittels HTTP-POST Anfrage an das *Editor Backend* übertragen, wofür ein entsprechender Endpoint in der bereitgestellten REST-API zur Verfügung steht. Zur Serialisierung können Datenbankadapter eingesetzt werden, was einen einfachen Austausch der Persistierungslogik ermöglicht.

[3] http://socket.io/

Zu Testzwecken steht aktuell nur ein *DebugSnapshotAdapter* (siehe Listing 4.4) bereit.

```
export default class DebugSnapshotAdapter {
  constructor() {
    this._snapshots = {};
  }

  getSnapshot(id) {
    return this._snapshots[ id ];
  }

  getSnapshots() {
    return this._snapshots;
  }

  saveSnapshot(content, id) {
    const snapshotId = id || this.getNextId();

    this._snapshots[ snapshotId ] = {
      id: snapshotId,
      content
    };

    return snapshotId;
  }

  getNextId() {
    return Object.keys(this._snapshots).length;
  }
}
```

Listing 4.4: Datenbankadapter zur Persistierung von Dokumentensnapshots (`editor-frontend/app/server/api/DebugSnapshotAdapter.js`)

4.2 Frontend

Auch auf Seite des Frontends sind zwei Komponenten zu unterscheiden: (1) Eine auf dem React Framework basierende Applikation, auf dem zukünftig auch das *Sweble Hub* Web Frontend aufbauen soll, sowie (2) der eigentliche *Wom3VisualEditor*, die Erweiterungen des *Wikimedia VisualEditors* zur Bearbeitung von *WOM* Dokumenten.

4.2.1 React Frontend

Das *React Frontend* basiert auf einem vorgegebenen Grundgerüst einer auf dem *React* Framework[4] basierenden Web Applikation. Diese wurde um Möglichkeiten zur dynamischen Abfrage der verfügbaren Artikel über die REST-API des *Sweble Backends* erweitert sowie einer Integration des *Wom3VisualEditors* (siehe Kapitel 4.2.2), um diese bearbeiten zu können.

Zur Einbindung des auf dem *jQuery*[5] basierenden *Wom3VisualEditors* kann die *React* Komponente *EditorBox* verwendet werden (siehe Listing 4.5). Der zu bearbeitende Artikelname kann dabei über das Attribut `article` des Eingabeparameters `props` übergeben werden (Zeile 28). Der Editor fragt den entsprechenden Artikel nach seiner Initialisierungsphase daraufhin selbstständig von der REST-API des *Sweble Backends* ab (Zeile 9).

Zusätzlichen werden *EventHandler* für das Empfangen bzw. Senden von Transaktionen der *ve.dm* Komponente registriert. Im Falle der lokalen Ausführung einer Transaktion wird diese über ein Socket an das *Editor Backend* übermittelt (Zeile 12). Beim Empfangen einer Transaktion wird diese lokal zur Ausführung gebracht (Zeile 18-20). Die Konfiguration der Callback Funktionen erfolgt außerhalb der Editorkomponente, um sie später einfacher an den konkreten Anwendungsfall anpassen zu können.

[4] https://facebook.github.io/react/
[5] https://jquery.com/

```
export default class EditorBox extends React.Component {
  ...
  render() {
    ...
    $(function () {
      ...
      new ve.init.sa.Platform(ve.messagePaths).initialize().done(
          function () {
        ...
        var page = ve.apiBase + '/article_xml?articleTitle=' +
            props.article;
        var surfaceContainer = new ve.wom3.SurfaceContainer(
            target, page, currentLang, currentDir, function(tx) {
          if (!tx.remote) {
            socket.emit('transaction', tx.getOperations());
          }
        });
        target.$element.append(surfaceContainer.$element);
        surfaceContainer.loadPage(page);

        socket.on('transaction', function(operations) {
          surfaceContainer.handleTransaction(operations);
        });
      });
    });
    ...
  }
}

EditorBox.propTypes = {
  article: React.PropTypes.string.isRequired
};
```

Listing 4.5: *React* Komponente *EditorBox* zur Einbindung des *Wom3VisualEditors* (`editor-frontend/app/client/editor/EditorBox.js`)

4.2.2 Wom3VisualEditor

Die Erweiterung des *VisualEditors* um die Möglichkeit zur Bearbeitung von *WOM* Dokumenten basiert auf der aktuellen, u.a. als GitHub Projekt[6] verfügbaren Codebasis. Um ein auf dem *VisualEditor* basierendes Projekt umzusetzen, wurde die MediaWiki Erweiterung des *VisualEditors*[7] als Vorlage herangezogen, die den Editor ebenfalls, um MediaWiki spezifische Elemente, erweitert.

Im Folgenden werden die verschiedenen Komponenten vorgestellt, die aktuell für eine Bearbeitung von *WOM* Dokumenten erforderlich sind.

Transformationen

Um die erforderlichen Änderungen am *VisualEditor* zu reduzieren, wird ein zu ladendes *WOM* Dokument jeweils vor der Übergabe an die *ve.dm* Komponente sowie nach dem Export daraus syntaktisch transformiert (siehe *Pre-/PostTransform* in Abbildung 3.9), wobei keine semantischen Änderungen vorgenommen werden.

Die nachfolgend aufgeführten Schritte beschreiben die Transformation zu Beginn des Ladeprozesses. Die Rücktransformation erfolgt analog, in umgekehrter Reihenfolge und wird hier nicht weiter ausgeführt.

TextTransformation
: Das *Sweble Backend* liefert *WOM-XML* Dokumente aktuell in einer sog. *pretty-print* Variante aus, in der etwa alle Elemente korrekt eingerückt sind. Dieses Verhalten ist während der Entwicklung sowie zur Fehlerbehebung hilfreich, da die Struktur des Dokuments für den menschlichen Betrachter einfacher zu verstehen ist, hat für den eigentlichen Inhalt des Dokuments jedoch keinerlei Bedeutung. Daher werden im ersten Schritt alle Textknoten entfernt, die nicht Kind eines *WOM* Textelements sind, um zu verhindern, dass diese bspw. als Zeilenumbrüche durch den Editor interpretiert werden.

RtdNodeTransformation
: Ebenso wie die im vorangegangenen Schritt entfernten Textknoten, werden *RTD* Elemente nicht für die Bearbeitung im Editor benötigt, sind jedoch für eine spätere Rücktransformation in die ursprüngliche Auszeichnungssprache essenziell. Aus diesem Grund werden die *RTD* Elemente in einem lokalen *RtdStore* Objekt mit einem eindeutigen Identifikator versehen, bis zur Rücktransformation zwischengespeichert und eine Referenz darauf bspw. im Elternelement abgelegt (vgl. Listing 4.6).

[6] https://github.com/wikimedia/VisualEditor
[7] https://github.com/wikimedia/mediawiki-extensions-VisualEditor

```
// Find RTD position
if (currentElement.previousElementSibling == null) {
  // First element within parent
  parentElement.dataset.rtd.firstElement = rtdId;
} else if (currentElement.nextElementSibling == null) {
  // Last element within parent
  parentElement.dataset.rtd.lastElement = rtdId;
} else {
    ...
}
```

Listing 4.6: Speichern der Referenz eines *RTD* Elements in seinem Elternelement (`visual-editor/modules/wom3-ve/ve.wom3.DocumentTransformation.js`)

Die bereits extrahierten und in den jeweiligen Elternelementen vermerkten *RTD* Elemente werden nun aus dem Dokument entfernt.

Bei der Rücktransformationen werden die *RTD* Elemente aus den zuvor gespeicherten Information rekonstruiert.

LinkNodeTransformation

Im nächsten Schritt werden *WOM* Linkelemente (`intlink` und `extlink`) durch reguläre HTML Linkelemente (`a`) ersetzt und deren *WOM* Typ im `rel` Attribut des Elements vermerkt (siehe Listing 4.7). Zusätzlich wird der Linktitel normalisiert, d.h. sichergestellt, dass der Titel als Kind des Linkelements vorhanden ist, um die vorhandene Linkannotation des *VisualEditors* als Grundlage für die *WOM* Erweiterung verwenden zu können. Im *WOM* Dokument ist der Linktitel nicht zwingend als Kindelement vorhanden, sofern sich dieser nicht vom Linkziel unterscheidet.

```
<a rel="wom3:' + linkType + '"></a>
```

Listing 4.7: Repräsentation eines *WOM* Linkelements nach der Transformation

ImageNodeTransformation

Bei der Erzeugung von *WOM* Elementen im *DOM* kommt es bei der Behandlung von Bildelementen zu Komplikationen, da diese in *HTML* keine Kindelemente besitzen dürfen, im *WOM* dagegen ein Bildtitelelement enthalten können.

Daher werden für den Im- und Export von Dokumenten alle Bildelemente `<image>` durch `<figure>` ersetzt.

SectionNodeTransformation

Eine weiterer Transformationsschritt betrifft die `Section` Elemente im *WOM*, die entsprechend der Darstellung in Listing 4.8, jeweils zwei Kindelemente besitzen: (1) `Heading` und (2) `Body`.

```
<section level="1">
    <heading>Title</heading>
    <body>Content</body>
</section>
```

Listing 4.8: Struktur von `Section` Elementen im *WOM*

Da die Strukturierung eines Dokuments in HTML nur implizit über die Verschachtelung von Überschriften definiert wird, werden *WOM* `Section` Elemente entsprechend in die in Listing 4.9 dargestellten, semantisch äquivalenten HTML Repräsentation überführt.

```
<h1>Title</h1>
Content
```

Listing 4.9: Repräsentation eines *WOM* Sectionelements nach der Transformation

Um die Erweiterung bzw. Modifikation der auszuführenden Transformationen zu unterstützen steht der `ve.wom3.DocumentTransformer` zur Verfügung, an dem Transformationen dynamisch registriert und in der korrekten Reihenfolge ausgeführt werden können. Das Registrieren (Zeile 2-7) sowie die Ausführung der Transformationen (Zeile 10-11) wird in Listing 4.10 dargestellt.

```
1  /* Registration */
2  ve.wom3.DocumentTransformer.registerTransformation(
3      new ve.wom3.TextTransformation()
4  );
5  ve.wom3.DocumentTransformer.registerTransformation(
6      new ve.wom3.RtdNodeTransformation()
7  );
8
9  /* Transformation */
10 content = ve.wom3.DocumentTransformer.preTransform(body);
11 body = ve.wom3.DocumentTransformer.postTransform(content);
```

Listing 4.10: Registrierung und Ausführung von Transformationen mittels `ve.wom3.DocumentTransformer` (`visual-editor/modules/wom3-ve/ve.wom3.DocumentTransformation.js`)

VisualEditor

Der *VisualEditor* basiert auf dem verbreiteten *jQuery*[8] JavaScript Framework und verwendet darüber hinaus eine eigene Bibliothek *OOjs*[9], um objektorientierte Konzepte in JavaScript verwenden zu können, sowie die UI Bibliothek *OOjs UI*[10] zur browserübergreifenden Erstellung von Webapplikationen.

Die Verwendung dieser Bibliotheken macht es aufwändig, das in *ECMAScript 6* eingeführte, native Klassenkonzept einzusetzen. Daher basiert die Erweiterung des *VisualEditors*, ebenso wie der Editor selbst, auf den zuvor aufgeführten Bibliotheken.

Im Folgenden soll die Beschreibung der Komponenten des *VisualEditors* anhand einer beispielhaften Implementierung eines *WOM* Elements aufgezeigt werden.

Das `Subst` Element im *WOM* dient der Darstellung von Auszeichnungssprachen spezifischen Elementen durch standardmäßig im *WOM* verfügbare Elemente. So bietet etwa Wikitext die in Listing 4.11 abgebildete Möglichkeit, als Linktitel das Linkziel mit einem Suffix zu verwenden.

```
1 [[Cat]]s
```

Listing 4.11: Setzen des Linktitels als Erweiterung des Linkziels um einen Suffix in Wikitext

Die zugehörige Darstellung im *Wiki Object Model* ist Listing 4.12 zu entnehmen. Ein `Subst` Element besitzt die beiden Kindelemente `Repl` und `For`. Das `For` Element enthält dabei den Wikitext spezifischen Inhalt, in diesem Fall ein `mww:intlink` Element (Zeile 10-12), welches in einem *WOM* interpretierenden System jedoch möglicherweise nicht zur Verfügung steht. Daher bietet das `Repl` Element eine Interpretation des Inhalts, ausgedrückt durch die im *WOM* zur Verfügung stehenden Elementen (Zeile 3-7).

Das `Subst` Element soll während der Bearbeitung im *VisualEditor* durch ein nicht direkt bearbeitbares Blockelement dargestellt werden, das den interpretierten Inhalt seines `Repl` Elements enthält. Mit der Fokussierung des Elements soll ein *Inspector* die Möglichkeit bieten, das Element wahlweise zu löschen oder das `Subst` Element durch den Inhalt seines `Repl` Kindelements zu ersetzen. Die dazu notwendigen Schritte werden im Folgenden grob skizziert.

[8] https://jquery.com/
[9] https://www.mediawiki.org/wiki/OOjs
[10] https://www.mediawiki.org/wiki/OOjs_UI

```
<subst>
    <repl>
        <intlink target="Cat">
            <title>
                <text>Cats</text>
            </title>
        </intlink>
    </repl>
    <for>
        <mww:intlink target="Cat" postfix="s">
            <rtd>[[Cat]]s</rtd>
        </mww:intlink>
    </for>
</subst>
```

Listing 4.12: Repräsentation der verkürzten Wikitext Syntax zum Setzen des Linktitels im *WOM*

DataModel (ve.dm)

In der *ve.dm* Komponente wird hierfür eine neue Klasse `ve.dm.Wom3SubstNode` angelegt. Da weder in HTML noch dem *VisualEditor* ein passendes Element existiert, das als Grundlage verwendet werden könnte, um es funktional zu erweitern, wird eine vom *VisualEditor* bereitgestellte Basisklasse verwendet. Listing 4.13 zeigt die Spezifikation der Elternklasse (Zeile 2) sowie die Erweiterung um zusätzliche Funktionalitäten durch die Verwendung von Mixins (Zeile 4-6). Ein `ve.dm.LeafNode` stellt hierbei ein Element im *ModelTree* dar, das keine weiteren Kindelemente besitzen kann. Mithilfe der Mixins `ve.dm.GeneratedContentNode` und `ve.dm.FocusableNode` werden darüber hinaus die Eigenschaften hinzugefügt, dass der während der Bearbeitung dargestellte Inhalt des Elements dynamisch erzeugt sowie das Element bspw. durch einen Mausklick fokussiert werden kann. Zusätzlich wird das `ve.dm.Wom3RtdHandler` Mixin dazu verwendet, die für alle Elemente des *WOM* erforderliche Überführung der *RTD* Referenzen zwischen linearem Modell und *DOM* Dokument bereit zu stellen.

```
/* Inheritance */
OO.inheritClass(ve.dm.Wom3SubstNode, ve.dm.LeafNode);

OO.mixinClass(ve.dm.Wom3SubstNode, ve.dm.GeneratedContentNode);
OO.mixinClass(ve.dm.Wom3SubstNode, ve.dm.FocusableNode);
OO.mixinClass(ve.dm.Wom3SubstNode, ve.dm.Wom3RtdHandler);
```

Listing 4.13: Definition der Klassenhierarchie des `ve.dm.Wom3SubstNode` Elements (`visual-editor/modules/wom3-ve/dm/nodes/ve.dm.Wom3SubstNode.js`)

Die konkrete Konvertierung zwischen *DOM* Element und linearem Modell wird in den beiden Funktionen `toDataElement` und `toDomElements` durchgeführt.

Zur Überführung in das lineare Modell werden, wie in Listing 4.14 dargestellt, im Falle des `Subst` Elements die beiden Kindelemente (Zeile 6-7) durch Verwendung des `XMLSerializers` (Zeile 4) serialisiert und die beiden Zeichenketten anschließend als Attribute auf dem Element im linearen Modell gesetzt (Zeile 12-13). Die Erstellung des grundlegenden linearen Elements wird an eine Elternklasse delegiert (Zeile 10), in das daraufhin die *RTD* Informationen übertragen werden (Zeile 11).

Die Rücktransformation eines Elements im linearen Modell zu einem Teilbaum des *DOM* in Listing 4.15 erfolgt weitestgehend analog. Das grundlegende *DOM* Element wird durch eine Elternklasse erzeugt (Zeile 4) und *RTD* Informationen aus dem linearen Modell extrahiert (Zeile 5). Nun werden die zuvor serialisierten und als Attribute gespeicherte Kindelemente wieder instantiiert (Zeile 8-9) und anschließend als Kinder des *DOM* Elements eingefügt (Zeile 11-12).

```
1  ve.dm.Wom3SubstNode.static.toDataElement = function (domElements,
       converter) {
2      var dataElement, serializer, substElement, replElement,
           forElement;
3
4      serializer = new XMLSerializer();
5      substElement = domElements[0];
6      replElement = (ve.wom3.findChildren(substElement, ['repl']))
           [0];
7      forElement = (ve.wom3.findChildren(substElement, ['for']))
           [0];
8
9      dataElement = ve.dm.Wom3SubstNode.super.static.toDataElement.
           apply(this, arguments);
10     ve.dm.Wom3SubstNode.static.handleDataElementRtd(domElements
           [0], dataElement);
11
12     dataElement.attributes.repl = serializer.serializeToString(
           replElement);
13     dataElement.attributes.for = serializer.serializeToString(
           forElement);
14
15     return dataElement;
16 };
```

Listing 4.14: `toDataElement` Funktion des `ve.dm.Wom3SubstNode` Elements zur Überführung des *DOM* Elements in das lineare Modell (`visual-editor/modules/wom3-ve/dm/nodes/ve.dm.Wom3SubstNode.js`)

```
ve.dm.Wom3SubstNode.static.toDomElements = function (dataElement,
    doc) {
    var domElements, substElement, replElement, forElement;

    domElements = ve.dm.Wom3SubstNode.super.static.toDomElements.
       apply(this, arguments);
    ve.dm.Wom3SubstNode.static.handleDomElementRtd(dataElement,
       domElements[0]);

    substElement = domElements[0];
    replElement = ve.wom3.createElementFromXml(dataElement.
       attributes.repl);
    forElement = ve.wom3.createElementFromXml(dataElement.
       attributes.for);

    substElement.appendChild(replElement);
    substElement.appendChild(forElement);

    return domElements;
};
```

Listing 4.15: `toDomElements` Funktion des `ve.dm.Wom3SubstNode` Elements zur Überführung des Elements im linearen Modell in einen Teilbaum des *DOM* (`visual-editor/modules/wom3-ve/dm/nodes/ve.dm.Wom3SubstNode.js`)

Um dem *VisualEditor* die neue Klasse der *ve.dm* Komponente bekannt zu machen, muss diese abschließend noch registriert werden (siehe Listing 4.16). Daraufhin werden `Subst` Elemente selbstständig beim Einlesen bzw. Exportieren eines Dokuments vom *VisualEditor* umgewandelt.

```
/* Registration */
ve.dm.modelRegistry.register(ve.dm.Wom3SubstNode);
```

Listing 4.16: Registrierung des `ve.dm.Wom3SubstNode` Elements (`visual-editor/modules/wom3-ve/dm/nodes/ve.dm.Wom3SubstNode.js`)

ContentEditable (ve.ce)

Mit der zuvor beschriebenen `ve.dm.Wom3SubstNode` Klasse kann der *VisualEditor* ein `Subst` Element in das lineare Modell überführen. Allerdings verfügt er noch über keine Kenntnis darüber, wie ein solches Element zur Bearbeitung im Webbrowser angezeigt werden soll. Hierfür bedarf es der zusätzlichen Klasse `ve.ce.Wom3SubstNode` innerhalb der *ve.ce* Komponente.

Analog der zuvor definierten Klassenhierarchie innerhalb der *ve.dm* Komponente, erfolgt dies in Listing 4.17 für *ve.ce*.

```
1 /* Inheritance */
2 OO.inheritClass(ve.ce.Wom3SubstNode, ve.ce.LeafNode);
3
4 OO.mixinClass(ve.ce.Wom3SubstNode, ve.ce.GeneratedContentNode);
5 OO.mixinClass(ve.ce.Wom3SubstNode, ve.ce.FocusableNode);
```

Listing 4.17: Definition der Klassenhierarchie des `ve.ce.Wom3SubstNode` Elements (`visual-editor/modules/wom3-ve/ce/nodes/ve.ce.Wom3SubstNode.js`)

Das verwendete Mixin `ve.ce.GeneratedContentNode` erwartet das Überschreiben der `generateContents` Methode, um den innerhalb des Elements darzustellenden Inhalt zu erzeugen. Wie in Listing 4.18 dargestellt, wird für diesen Fall die serialisierte Darstellung des `Repl` Elements dynamisch aus einem Attribut des `Subst` Elements im linearen Modell geladen (Zeile 4).

```
1 ve.ce.Wom3SubstNode.prototype.generateContents = function(config)
      {
2     var deferred = $.Deferred();
3
4     deferred.resolve( $( '<span>' + this.getModel().getAttribute(
          'repl') + '</span>' ).get() );
5
6     return deferred.promise();
7 };
```

Listing 4.18: Generierung des Inhalts eines `ve.ce.Wom3SubstNode` Elements (`visual-editor/modules/wom3-ve/ce/nodes/ve.ce.Wom3SubstNode.js`)

Nach einer abschließenden Registrierung der Klasse an der `ve.ce.nodeFactory` wird ein im linearen Modell vorhandenes `Subst` Element während der Bearbeitung im *VisualEditor* dargestellt.

UserInterface (ve.ui)

In der aktuellen Ausprägung kann mit einem `Subst` Element bereits grundlegend interagiert werden, dieses also etwa gelöscht oder im Dokument verschoben werden. Für zusätzliche Funktionalitäten, um bspw. das `Subst` Element durch sein `Repl` Kindelement zu ersetzen, muss darüber hinaus die *ve.ui* Komponente erweitert werden.

Hierfür können bspw. die beiden Komponenten *ContextItem* und *Inspector* eingesetzt werden. Wird ein entsprechendes `Subst` Element im Editor selektiert, so

öffnet sich das zugehörige *ContextItem*, was verwendet werden kann, um bestimmte Informationen anzuzeigen, etwa den Inhalt des `For` Kindelements, wie in Abbildung 4.1 dargestellt. Um eine weitergehende Interaktion mit dem Element zu beginnen, kann der *Edit* Button gedrückt werden, um entsprechend dem Command Pattern ein beliebiges Kommando abzusetzen. Daraufhin öffnet sich in diesem Fall der in Abbildung 4.2 abgebildete *Inspector*, der zum Einen die Möglichkeit bietet, den Inhalt des `For` Elements textuell zu modifizieren, sowie durch Verwendung des *Dissolve* Buttons, das gesamte `Subst` Element durch den Inhalt seines `Repl` Elements zu ersetzen. Dies führt in diesem Beispiel zu dem in Abbildung 4.3 abgebildeten Ergebnis eines internen Links.

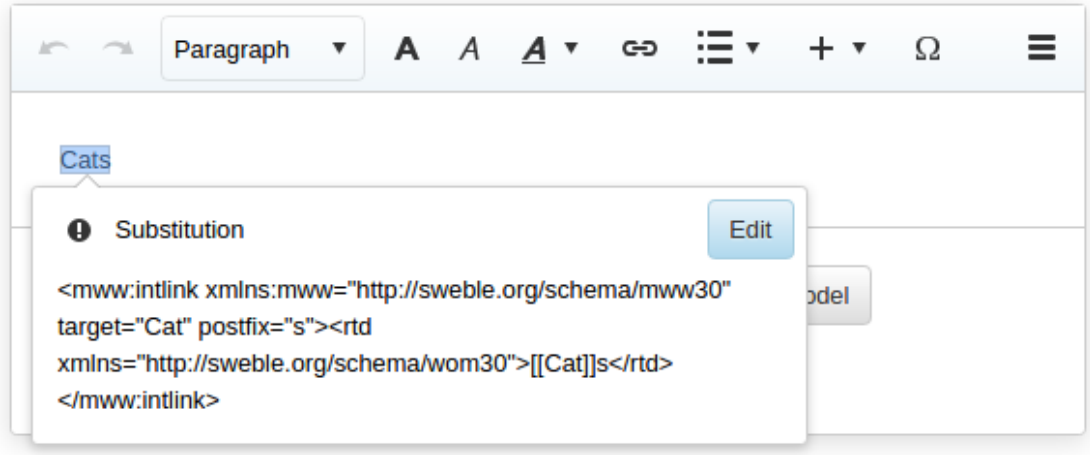

Abbildung 4.1: *ContextItem* eines `Subst` Elements im *VisualEditor*

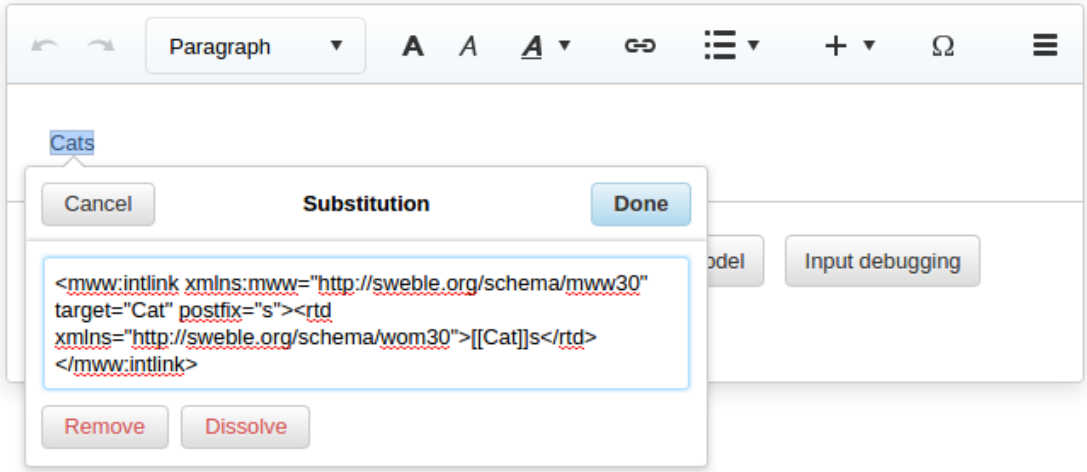

Abbildung 4.2: *Inspector* eines `Subst` Elements im *VisualEditor*

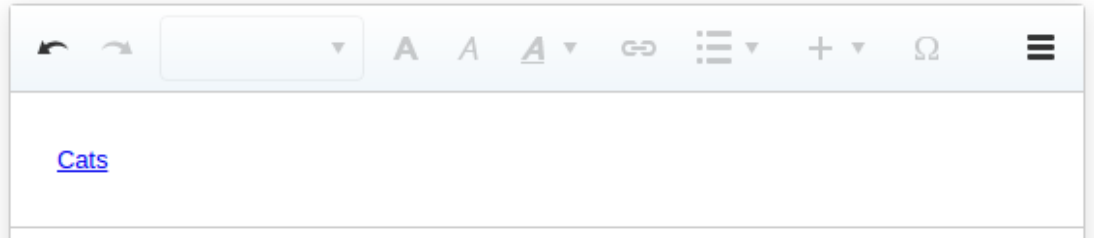

Abbildung 4.3: Ergebnis der Ersetzung eines `Subst` Elements im *VisualEditor* durch seine enthaltene *WOM* Repräsentation

In Listing 4.19 ist der zur Registrierung eines neuen Kommandos, in diesem Fall zum Öffnen eines *Inspectors* bzw. beliebigen Dialogs, aufgeführt. Kommandos können ebenfalls durch ein Element in der *Toolbar* getriggert werden. Da bei der Bearbeitung eines Dokuments jedoch keine neuen `Subst` Elemente erzeugt werden sollen, wurde auf eine entsprechende Erweiterung der *Toolbar* verzichtet.

```
1 ve.ui.commandRegistry.register(
2   new ve.ui.Command(
3     'wom3Subst', 'window', 'open',
4     { args: [ 'wom3Subst' ], supportedSelections: [ 'linear' ] }
5   )
6 );
```

Listing 4.19: *Command* zum Öffnen eines *Inspectors* für `Subst` Elemente (`visual-editor/modules/wom3-ve/ui/commands/ve.ui.Wom3SubstCommand.js`)

Für die Erstellung eines *ContextItems* genügt es in diesem Fall, die anzuzeigende Beschreibung anzupassen. Listing 4.20 zeigt das dafür notwendige Überschreiben der `getDescription` Methode, die die textuelle Repräsentation des `For` Elements aus dem im linearen Modell gespeicherten Attribut ausliest und zurück gibt.

```
1 ve.ui.Wom3SubstContextItem.prototype.getDescription = function ()
      {
2   return "" + this.model.getAttribute('for');
3 };
```

Listing 4.20: *ContextItem* für `Subst` Elemente (`visual-editor/modules/wom3-ve/ui/contextitems/ve.ui.Wom3SubstContextItem.js`)

Der *Inspector* zur Bearbeitung von `Subst` Elementen basiert auf dem `Comment-Inspector` des *VisualEditors*, der bereits über ein Eingabefeld zum Bearbeiten von Text verfügt. Dieser wird um einen zusätzlichen *Dissolve* Button erweitert, um das `Subst` Element durch den Inhalt seines `Repl` Element zu ersetzen.

Listing 4.21 zeigt die zum Erstellen eines zusätzlichen Buttons notwendigen Modifikationen. Zunächst wird die Liste der *actions* um einen weiteren Eintrag ergänzt (Zeile 3-8). Basierend auf dem Namen des Eintrags wird der `TeardownProcess` erweitert, der den auszuführenden Code beim Schließen eines *Inspectors* enthält. Der serialisierte Inhalt des `Repl` Elements wird in ein *DOM* Dokument umgewandelt (Zeile 20-23), was wiederum in das lineare Modell überführt wird (Zeile 19). Daraufhin wird das selektierte *Subst* Element entfernt (Zeile 33) und stattdessen das soeben erzeugte Dokument an der identischen Position eingefügt (Zeile 34-36). Abschließend werden die Änderungen committed (Zeile 37) und somit das lineare Modell des Dokuments aktualisiert.

Transaktionen

Transaktionen (`ve.dm.Transaction`) im *VisualEditor* enthalten eine Liste von Operationen, ein Zustandsflag, ob die Transaktion bereits ausgeführt wurde sowie eine Referenz auf das Dokument, auf das sich die Transaktion bezieht.

Die Dokumentenklasse (`ve.dm.Document`) ermöglicht die Registrierung von transaktionsbezogenen *EventHandlern* auf der Instanz eines Dokuments für die folgenden Ereignisse:

precommit
: Eine neue Transaktion ist verfügbar und soll ausgeführt werden.

presynchronize
: Die Transaktion wurde ausgeführt und das lineare Modell entsprechend modifiziert, allerdings wurde das die zugehörige Baumstruktur noch nicht aktualisiert

transact
: Die Transaktion wurde vollständig ausgeführt.

Die Ereignisse eignen sich dazu, die Transaktionsausführung durch eine externe Komponente zu beobachten und etwa die ausgeführten Transaktionen an das *Editor Backend* zu übermitteln.

```
ve.ui.Wom3SubstNodeInspector.static.actions = [
  ...
  {
    action: 'dissolve',
    label: OO.ui.deferMsg('visualeditor-wom3-substinspector-
       dissolve-tooltip'),
    flags: 'destructive',
    modes: 'edit'
  }
].concat(ve.ui.FragmentInspector.static.actions);
...
ve.ui.Wom3SubstNodeInspector.prototype.getTeardownProcess =
    function (data) {
  data = data || {};
  return ve.ui.Wom3SubstNodeInspector.super.prototype.
      getTeardownProcess.call(this, data)
    .first(function () {
      var surfaceModel = this.getFragment().getSurface();
      var doc = surfaceModel.getDocument();

      if (data.action === 'dissolve') {
        var replDocument = ve.dm.converter.getModelFromDom(
          ve.createDocumentFromXml(
            '<?xml version="1.0" encoding="UTF-8" standalone="no
               "?>' +
            this.substNode.getAttribute('repl')
          )
        );
        var offset = this.selectedNode.getRange().from - 1;
        var range = ve.Range.static.newFromHash({
          // Skip paragraph
          from: 1,
          // Skip /paragraph, internalList, /internalList
          to: replDocument.getLength() - 3
        });
        this.getFragment()
          .removeContent()
          .change(
            ve.dm.Transaction.newFromDocumentInsertion(doc,
               offset, replDocument, range)
          );
        surfaceModel.applyStaging();
      }
      ...
    }, this);
};
```

Listing 4.21: *Inspector* für `Subst` Elemente (`visual-editor/modules/wom3-ve/ui/inspectors/ve.ui.Wom3SubstNodeInspector.js`)

Akzeptanztests

Um die Korrektheit der Transformationen von Elementen sicherzustellen und Regressionen zu vermeiden, sowohl bei der Transformation vom eingelesenen *DOM* Baum in das für die Bearbeitung verwendete lineare Modell, als auch zurück in einen *DOM* Baum, stehen Testfälle bereit. Diese stehen für jedes im *VisualEditor* eingebaute *WOM* Element zur Verfügung, testen aktuell einen einfachen Transformationszyklus, beinhalten jedoch keine Bearbeitung des Dokuments.

Zur Implementierung der Testfälle wurde das bereits im *VisualEditor* verwendete *Karma*-Framework[11] zur Ausführung von Testfällen sowie *Jasmine*-Framework[12] zur Spezifikation der eigentlichen Testfälle verwendet.

In Listing 4.22 wird beispielhaft der Testfall für einen grundlegenden Transformationszyklus eines *Paragraph* Elements aufgezeigt. Hierbei wird die *XML* Serialisierung eines *WOM* Dokuments verwendet (Zeile 7), um dieses in einer Editorinstanz zu öffnen und daraufhin wieder zu exportieren (Zeile 8). Anschließend werden die *DOM* Bäume des ursprünglichen sowie des exportierten Dokuments miteinander verglichen (Zeile 10). Für den Vergleich von *DOM* Bäumen wurde das *Jasmine* Framework um einen *CustomEqualityTester* erweitert (Zeile 3), der den Baum rekursiv absteigt und die einzelnen Knoten sowie deren Attribute miteinander vergleicht.

```
1  describe("Wom3ParagraphNode", function () {
2      beforeEach(function () {
3          jasmine.addCustomEqualityTester(ve.wom3.domTreeEquality);
4      });
5
6      it("can parse and export a basic WOM3 paragraph node",
           function () {
7          var originalXml = '<p topgap="5" bottomgap="10"><text>
               Lorem Ipsum</text></p>';
8          var result = ve.wom3.initJasmineEnvironment(originalXml);
9
10         expect(result.resultDocument).toEqual(result.
               originalDocument);
11     });
12 });
```

Listing 4.22: Testfall zur Überprüfung des Transformationszyklusses anhand eines *Paragraph* Elements (`visual-editor/tests/nodes/Wom3ParagraphNode.test.js`)

[11] https://karma-runner.github.io
[12] https://jasmine.github.io/

Die Ausführung der Testfälle erfolgt aktuell im *Firefox* Browser. Um die Testfälle auch in einer Umgebung ausführen zu können, in der keine grafische Benutzeroberfläche zur Verfügung steht, wie etwa zur automatischen Ausführung innerhalb einer *Continuous Integration* Umgebung, bietet es sich an auch dafür eine Alternative anzubieten.

Die hierfür zur Ausführung von *JavaScript* angedachte Browserumgebung ohne die Verwendung einer grafischen Benutzeroberfläche *PhantomJS*[13] bietet jedoch keine ausreichende Unterstützung des *JavaScript*-Standards. So steht bei der Ausführung bspw. das innerhalb des *VisualEditors* verwendete `children` Attribut eines Elements nicht zur Verfügung, woraufhin `undefined` zurückgeliefert und anschließend die Ausführung abgebrochen wird.

Eine weitere Alternative bietet die Fernsteuerung eines Webbrowsers mithilfe des *Selenium* Frameworks[14]. Hierfür kann bspw. eine Serverinstanz mitsamt enthaltenem Webbrowser innerhalb eines *Docker*[15] Containers ausgeführt werden, zu dem sich das Testframework verbindet und daraufhin dort die Testfälle zur Ausführung bringt.

[13] http://phantomjs.org/
[14] http://www.seleniumhq.org/
[15] https://www.docker.com/

5 Evaluation

Nach der Beschreibung der Architektur sowie der Implementierung in den vorangegangenen Kapiteln soll nun eine Evaluation der entwickelten Lösung gegeben werden. Die Grundlage hierfür bilden die in Kapitel 1.2 aufgelisteten Anforderungen.

VisualEditor

Auf Basis der aktuellen Version des *Wikimedia VisualEditors* wurde eine visuelle Editorkomponente zur Bearbeitung von *WOM* Dokumenten entwickelt, die zukünftig in die *Sweble Hub* Plattform integriert werden kann. Hierfür wurde der gesamte Transformationsprozess für die Umwandlung eines *WOM* Dokuments in eine durch den *VisualEditor* interpretierbare Form neu konzipiert. Die Verwendung der *XML* Serialisierung eines *WOM* Dokuments leistet eine signifikante Reduzierung der Komplexität der Lösung im Gegensatz zu der in Haase (2014) vorgestellten Architektur, die eine *JSON* Serialisierung als Grundlage der Bearbeitung verwendet. Die dort serverseitig eingesetzten *Wom3Tools* zur Konvertierung zwischen JSON und HTML Darstellung eines *WOM* Dokuments beinhalten eine komplette Implementierung des *WOM* in JavaScript, was neben der Komplexität der Implementierung einen erhöhten Wartungsaufwand mit sich bringt.

Bei der Implementierung wurde besonderen Wert darauf gelegt, die als Grundlage dienende Codebasis des *VisualEditors* nicht zu verändern, um eine spätere Aktualisierung zur Fehlerbehebung bzw. zur Erweiterung der Funktionalität möglichst einfach zu gestalten. Alle zur Unterstützung des *Wiki Object Models* durchgeführten Modifikationen wurden in einem separaten Projekt vorgenommen, das die Codebasis des *VisualEditors* nur erweitert. Eine Auflistung der verwendeten Fremdsoftware sowie deren Lizenzen sind der Auflistung in Anhang A zu entnehmen.

Erweiterung

Zur Erweiterung des *VisualEditors* um kontextabhängig dynamische Menüeinträge anzeigen zu lassen, um bspw. bestimmte Aktionen ausführen zu können, kann jedem *WOM* Element etwa ein *Inspector* zugewiesen werden.

Die Beschreibung der Vorgehensweise zur Implementierung eines solchen kontextabhängigen Menüs ist in Kapitel 4 beispielhaft aufgeführt. Darüber hinaus findet sich eine ausführlichere Dokumentation in Anhang B.

WOM

Die Erweiterung des *VisualEditors* unterstützt alle im *Wiki Object Model* enthaltenen Elemente und ermöglicht einen verlustfreien Transformationszyklus eines beliebigen *WOM* Dokuments in das lineare Datenmodell des Editors und zurück in ein *WOM* Dokument.

Die Korrektheit der grundlegenden Transformationen für alle im Rahmen dieser Arbeit implementierten *WOM* Elemente werden durch Akzeptanztests sichergestellt. Für eine Beschreibung des Vorgehens ist auf Kapitel 4 zu verweisen, um die Testfälle bei Bedarf erweitern zu können.

Die vorhandenen Testfälle decken nur die grundlegendsten Fälle ab und sollten zukünftig noch weiter ausgebaut werden, etwa um die beliebige Verschachtelung von Elementen.

Sync

Zur Synchronisierung des Dokumentenzustands mit dem Server stehen in der im Rahmen dieser Arbeit entwickelten Lösung zwei Möglichkeiten zur Verwendung bereit.

Der Editor erlaubt es in regelmäßigen Zeitintervallen, den aktuellen Inhalt des Dokuments als Ganzes im *WOM-XML* Format zu exportieren und dieses daraufhin als HTTP-POST an das *EditorBackend* zu übermitteln. Serverseitig können die exportierten Zustände vorgehalten werden, um den Bearbeitungsfortschritt bei Bedarf wiederherstellen zu können. Außerdem können Unterschiede zwischen zwei Zuständen problemlos durch Verwendung eines Diff-Tools ermittelt werden (z.B. HDDiff[1]).

Der zweite Ansatz zur Synchronisierung verwendet die Transaktionen der *ve.dm* Komponente zur Anwendung auf dem linearen Modell des *VisualEditors*. Hierbei werden die Transaktionen zusätzlich zu ihrer lokalen Ausführung auch an an das *EditorBackend* übertragen. Zur Wiederherstellung eines Bearbeitungszustands wird das ursprüngliche Dokument im Editor geladen, anschließend alle Transaktionen vom *EditorBackend* abgefragt und chronologisch lokal zur Ausführung gebracht. Dies bietet den Vorteil, dass der gesamte Bearbeitungsverlauf zur Verfügung steht, also auch wieder rückgängig gemacht werden kann, im Gegensatz zum direkten Laden eines *WOM* Dokuments.

Um mit diesem Ansatz das *WOM* Dokument auf Serverseite aktualisieren zu können, bedarf es zusätzlich noch einer Übersetzung des linearen Mo-

[1] https://github.com/sweble/hddiff

dells des Editors in *WOM*. Dies könnte etwa durch eine serverseitige Ausführung der *ve.dm* Komponente innerhalb einer *Node.js* Umgebung relativ einfach implementiert werden. Die Voraussetzung dafür ist jedoch, dass die für *ve.dm* erforderlichen *DOM* Funktionalitäten, die in *Node.js* nur durch die Verwendung einer zusätzlichen *DOM* Bibliothek (z.B. jsdom[2]) verfügbar sind, eine ausreichende Unterstützung bieten.

Im Hinblick auf eine mögliche Erweiterung des Editors um synchrone Kollaborationsfähigkeiten sowie der geringeren Synchronisationslatenz sollte der zweite Ansatz präferiert werden.

[2]https://github.com/tmpvar/jsdom

Anhang A Bill Of Materials

Im Folgenden wird die eingesetzte Fremdsoftware sowie ihrer Lizenz aufgeführt, aufgeteilt anhand der verschiedenen Komponenten.

A.1 Editor Backend

Software	Lizenz	URL
Guava	Apache 2.0	https://github.com/google/guava
SLF4J	MIT	http://www.slf4j.org/
Dropwizard	Apache 2.0	http://www.dropwizard.io/
RESTful Services API	CDDL	https://jax-rs-spec.java.net/
JGit	EDL	https://eclipse.org/jgit/
Joda Time	Apache 2.0	http://www.joda.org/joda-time/
Rats!	LGPL	http://cs.nyu.edu/rgrimm/xtc/
Saxon HE	MPL	http://saxon.sourceforge.net/
Parser Toolkit	Apache 2.0	http://sweble.org
Sweble Wikitext Components	Apache 2.0	http://sweble.org
Sweble WOM v3	Apache 2.0	http://sweble.org
Sweble Engine	Apache 2.0	http://sweble.org

A.2 Visual Editor

Software	Lizenz	URL
grunt	MIT	github.com/gruntjs/grunt
grunt-contrib-cssmin	MIT	github.com/gruntjs/grunt-contrib-cssmin
grunt-contrib-uglify	MIT	github.com/gruntjs/grunt-contrib-uglify
grunt-data-merge	MIT	github.com/rd5/grunt-data-merge
grunt-http-server	MIT	github.com/divhide/node-grunt-http-server
grunt-karma	MIT	github.com/karma-runner/grunt-karma
jasmine-core	MIT	github.com/jasmine/jasmine
karma	MIT	github.com/karma-runner/karma
karma-firefox-launcher	MIT	github.com/karma-runner/karma-firefox-launcher
karma-jasmine	MIT	github.com/karma-runner/karma-jasmine

A.3 Editor Frontend

Software	Lizenz	URL
Foundation for Sites	MIT	foundation.zurb.com/sites.html
React	BSD3	facebook.github.io/react/
babel-preset-es2015	MIT	github.com/babel/babel
babel-preset-react	MIT	github.com/babel/babel
babel-register	MIT	github.com/babel/babel
babelify	MIT	github.com/babel/babelify
body-parser	MIT	github.com/expressjs/body-parser
browserify	MIT	github.com/substack/node-browserify
browserify-global-shim	MIT	github.com/rluba/browserify-global-shim
chai	MIT	github.com/chaijs/chai
compression	MIT	github.com/expressjs/compression
del	MIT	github.com/sindresorhus/del
eslint	MIT	github.com/eslint/eslint
eslint-config-airbnb	MIT	github.com/airbnb/javascript
eslint-plugin-import	MIT	github.com/benmosher/eslint-plugin-import
eslint-plugin-jsx-a11y	MIT	github.com/evcohen/eslint-plugin-jsx-a11y
eslint-plugin-react	MIT	github.com/yannickcr/eslint-plugin-react
express	MIT	github.com/expressjs/express
gulp	MIT	github.com/gulpjs/gulp
gulp-autoprefixer	MIT	github.com/sindresorhus/gulp-autoprefixer
gulp-babel	MIT	github.com/babel/gulp-babel
gulp-babel-istanbul	MIT	github.com/cb1kenobi/gulp-babel-istanbul
gulp-eslint	MIT	github.com/adametry/gulp-eslint
gulp-inject-modules	MIT	github.com/cb1kenobi/gulp-inject-modules
gulp-mocha	MIT	github.com/sindresorhus/gulp-mocha
gulp-nodemon	MIT	github.com/JacksonGariety/gulp-nodemon
gulp-print	MIT	github.com/alexgorbatchev/gulp-print
gulp-rename	MIT	github.com/hparra/gulp-rename
gulp-sass	MIT	github.com/dlmanning/gulp-sass
gulp-sourcemaps	ISC	github.com/floridoo/gulp-sourcemaps
gulp-uglify	MIT	github.com/terinjokes/gulp-uglify
jade	MIT	github.com/pugjs/pug
morgan	MIT	github.com/expressjs/morgan
react	BSD3	github.com/facebook/react
react-dom	BSD3	github.com/facebook/react
sinon	BSD3	github.com/sinonjs/sinon

Software	**Lizenz**	**URL**
socket.io	MIT	github.com/socketio/socket.io
socket.io-client	MIT	github.com/socketio/socket.io-client
stream-combiner2	MIT	github.com/substack/stream-combiner2
superagent	MIT	github.com/visionmedia/superagent
vinyl-buffer	MIT	github.com/hughsk/vinyl-buffer
vinyl-source-stream	MIT	github.com/hughsk/vinyl-source-stream
watchify	MIT	github.com/substack/watchify
winston	MIT	github.com/winstonjs/winston

Anhang B Entwickler Dokumentation

Im Folgenden werden zusätzliche Informationen für die Erweiterung bzw. Modifikation des visuellen Editors aufgeführt. Zunächst wird in Abschnitt B.1 die Ordnerstruktur des Projektrepositories beschrieben. Anschließend beinhaltet Abschnitt B.2 zusätzliche Informationen für den eigentlichen *VisualEditor*, sowie Abschnitt B.3 Informationen zu dessen Aktualisierung.

B.1 Projektstruktur

Das Projektrepository beinhaltet die drei Komponenten *Sweble Backend*, *Editor Frontend* und *VisualEditor*.

Das *Sweble Backend* im Ordner `editor-backend/` stellt die vom Editor verwendete REST-API zum Abfragen von *WOM* Dokumenten bereit. Dazu muss im Ordner `editor-backend/wiki/` ein Git Repository erstellt werden, das Dateien mit der Endung `.wikitext` enthält und daraufhin als Datenbasis dient.

Die Komponente des *Editor Frontends* besteht aus einer *React* Web Applikation, die unterhalb von `editor-frontend/` zu finden ist. Zusätzlich zur Bereitstellung einer Website enthält diese das vom Editor zur Persistierung verwendete *Editor Backend*.

Der eigentliche visuelle Editor zur Bearbeitung von *WOM* Dokumenten liegt im Ordner `visual-editor/`. Der als Grundlage fungierende *VisualEditor* ist als Git Submodul unterhalb von `visual-editor/lib/ve/` eingebunden.

B.2 VisualEditor Dokumentation

Zunächst findet sich in Abschnitt B.2.1 eine Beschreibung zum Aufsetzen der Entwicklungsumgebung, gefolgt von einer Sammlung an Informationen über den Wikimedia *VisualEditor* in Abschnitt B.2.2. Abschnitt B.2.3 beschreibt abschließend die Ausführung der Testfälle.

B.2.1 Entwicklungsumgebung

Für die Entwicklung des visuellen Editors sollten die beiden Komponenten *VisualEditor* und *Sweble Backend* aufgesetzt und gestartet werden. Die dazu notwendigen Schritte werden in Listing 5.1 für das *Sweble Backend* sowie in Listing 5.2

für den *VisualEditor* aufgezeigt. Weitere Informationen können der im Projektrepository enthaltenen Datei `README.md` entnommen werden.

```
cd editor-backend
mkdir wiki && cd wiki && git init && echo "Hello World" > test.wikitext && git add . && git commit -am "Initial commit" && cd ..
mvn package
./bin/run
```

Listing 5.1: Aufsetzen und Starten der *Sweble Backend* Komponente

```
cd visual-editor
git submodule update --init
npm install
grunt build serve
```

Listing 5.2: Aufsetzen und Starten der *VisualEditor* Komponente

Anschließend steht ein lokaler Webserver zur Verfügung, der den visuellen Editor unter http://0.0.0.0:9001/ bereitstellt. Das *Sweble Backend* stellt die beiden REST-API Endpoints http://0.0.0.0:8080/articles für eine Übersicht der verfügbaren Dokumente sowie http://0.0.0.0:8080/article_xml?articleTitle=test zur Abfrage eines einzelnen *WOM* Dokuments zur Verfügung.

Innerhalb der Entwicklungsumgebung werden alle für den *VisualEditor* benötigten JavaScript sowie CSS Dateien separat geladen, was im Entwicklungsprozess etwa bei der Fehlersuche hilfreich ist, jedoch zu Lasten der Performanz geht. Für den Produktivbetrieb können die Dateiinhalte daher auch mit dem in Listing 5.3 aufgeführten Befehle kombiniert und so die Anzahl der zu ladenden Dateien reduziert werden. Die daraus resultierenden Dateien befinden sich daraufhin im Ordner `visual-editor/dist/`.

```
cd visual-editor
grunt deploy
```

Listing 5.3: Generierung statischer Assets für den Produktivbetrieb des *VisualEditors*

Die zuvor erzeugten Dateien finden auch innerhalb der *Editor Frontend* Komponente Verwendung und müssen daher zuvor erzeugt werden. Im Anschluss daran kann das Frontend gestartet werden, wie in Listing 5.4 beschrieben.

```
cd editor-frontend
npm install
bower install
gulp build:wom3-ve default
```

Listing 5.4: Aufsetzen und Starten der *Editor Frontend* Komponente

Das *Editor Frontend* ist nun unter http://0.0.0.0:6001 erreichbar.

B.2.2 Wikimedia VisualEditor

Nachfolgend werden zusätzliche Ressourcen für Informationen bezüglich des eigentlichen Wikimedia *VisualEditors* aufgelistet.

https://github.com/wikimedia/VisualEditor
: Das Git Repository des *VisualEditors*. Das im Projektverzeichnis eingebundene Git Submodul unter `visual-editor/lib/ve/` verwendet ebenfalls dieses Repository

https://www.mediawiki.org/wiki/VisualEditor/Design
: Beschreibung der dem *VisualEditor* zugrunde liegenden Designkonzepte sowie der Softwarearchitektur

https://doc.wikimedia.org/VisualEditor/master/
: API Dokumentation der im *VisualEditor* verwendeten Klassen

https://www.mediawiki.org/wiki/VisualEditor/changelog
: Changelog des *VisualEditors*, das die wesentlichen Änderungen für jedes Release beinhaltet. Darüber hinaus ist auf die Git Historie zu verweisen

https://www.mediawiki.org/wiki/OOjs
: Dokumentation der dem *VisualEditor* zugrundeliegenden Bibliothek zur Verwendung objektorientierter Konzepte in JavaScript

https://www.mediawiki.org/wiki/OOjs_UI
: Dokumentation der im *VisualEditor* verwendeten Bibliothek zum Erstellen von Benutzeroberflächen

https://github.com/wikimedia/mediawiki-extensions-VisualEditor
: *MediaWiki* Erweiterung des *VisualEditors*, das den Editor um *MediaWiki* spezifische Elemente erweitert und Codebeispiele zur Verwendung bzw. Erweiterung des *VisualEditors* bietet

B.2.3 Akzeptanztests

Die Testfälle zur Überprüfung des Transformationszyklus befinden sich unterhalb von `visual-editor/tests/`. Zur Ausführung der Testfälle muss der in Listing 5.5 aufgeführte Befehl ausgeführt werden.

```
1 cd visual-editor
2 grunt test
```

Listing 5.5: Befehl zum Ausführen der Testfälle

B.3 Aktualisierung des VisualEditors

Die Aktualisierung der dem visuellen Editor zur Bearbeitung von WOM Dokumenten zugrundeliegenden *VisualEditor* der Wikimedia Foundation wird durch eine Aktualisierung des Git Submoduls erreicht, das im Projektrepository auf die aktuell in Verwendung befindliche Version verweist (`visual-editor/lib/ve/`). Hierbei sollte jedoch stets auf die im Changelog[3] bzw. der Commit Historie aufgeführten Änderungen geachtet werden, da es teilweise noch zu grundlegenden Veränderungen kommen kann, die eine Anpassung der *WOM* Erweiterung erforderlich machen. Die zur Aktualisierung notwendigen Schritte werden in Listing 5.6 aufgeführt.

```
1 git submodule update --remote --merge
2 git commit visual-editor/lib/ve
```

Listing 5.6: Aktualisierung des *VisualEditor* Git Submoduls

[3] https://www.mediawiki.org/wiki/VisualEditor/changelog

Literaturverzeichnis

Ahmed-Nacer, M., Ignat, C.-L., Oster, G., Roh, H.-G. & Urso, P. (2011). Evaluating crdts for real-time document editing. In *Proceedings of the 11th acm symposium on document engineering* (S. 103–112). DocEng '11. Mountain View, California, USA: ACM. doi:10.1145/2034691.2034717

Anne van Kesteren. (2014). *CORS*. W3C. https://www.w3.org/TR/2014/REC-cors-20140116/. Neueste Version verfügbar unter https://www.w3.org/TR/cors/.

Berjon, Robin et al. (2014). *HTML5*. W3C. https://www.w3.org/TR/2014/REC-html5-20141028/. Neueste Version verfügbar unter https://www.w3.org/TR/html5/.

Briot, L., Urso, P. & Shapiro, M. (2016 November). High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*. Sanibel Island, FL, USA. doi:10.1145/2957276.2957300

Cai, W., He, F. & Lv, X. (2015). Multi-core accelerated operational transformation for collaborative editing. In *Collaborative computing: networking, applications, and worksharing - 11th international conference, collaboratecom 2015, wuhan, china, november 10-11, 2015. proceedings* (S. 121–128). doi:10.1007/978-3-319-28910-6_11

Cart, M. & Ferrié, J. (2006). Synchronizer based on operational transformation for p2p environments.

Davis, A. H., Sun, C. & Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 acm conference on computer supported cooperative work* (S. 58–67). CSCW '02. New Orleans, Louisiana, USA: ACM. doi:10.1145/587078.587088

Dohrn, H. & Riehle, D. (2011). *Wom: an object model for wikitext* (Techn. Ber. Nr. CS-2011-05). Technische Fakultät.

Dourish, P. (1995). The parting of the ways: divergence, data management and collaborative work. In *Proceedings of the fourth conference on european conference on computer-supported cooperative work* (S. 215–230). ECSCW'95. Stockholm, Sweden: Kluwer Academic Publishers. Zugriff unter http://dl.acm.org/citation.cfm?id=1241958.1241972

Ellis, C. A. & Gibbs, S. J. (1989 Juni). Concurrency control in groupware systems. *SIGMOD Rec. 18*(2), 399–407. doi:10.1145/66926.66963

Fraser, N. (2009a). Differential synchronization. In *Doceng'09, proceedings of the 2009 acm symposium on document engineering* (S. 13–20). 2 Penn Plaza, Suite 701, New York, New York 10121-0701. Zugriff unter http://neil.fraser.name/writing/sync/eng047-fraser.pdf

Fraser, N. (2009b). Googletechtalks - differential synchronization. last checked on 15/11/2016. Zugriff unter http://www.youtube.com/watch?v=S2Hp_1jqpY8

Haase, M. (2014). *A visual editor for the wiki object model.*

Ignat, C. (2002). Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems.

Ignat, C.-L. & Norrie, M. C. (2006). Flexible collaboration over xml documents. In *Proceedings of the third international conference on cooperative design, visualization, and engineering* (S. 267–274). CDVE'06. Mallorca, Spain: Springer-Verlag. doi:10.1007/11863649_33

Imine, A., Molli, P., Oster, G. & Rusinowitch, M. (2003). Proving correctness of transformation functions in real-time groupware. In *Ecscw'03: proceedings of the eighth conference on european conference on computer supported cooperative work* (S. 277–293). Norwell, MA, USA: Kluwer Academic Publishers. doi:10.1145/587078.587088

Lamport, L. (1998 Mai). The part-time parliament. *ACM Trans. Comput. Syst. 16*(2), 133–169. doi:10.1145/279227.279229

Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T. & Englund, M. (2012). Cored: browser-based collaborative real-time editor for java web applications. In *Proceedings of the acm 2012 conference on computer supported cooperative work* (S. 1307–1316). CSCW '12. Seattle, Washington, USA: ACM. doi:10.1145/2145204.2145399

Le Hors, Arnaud et al. (2004). *Document Object Model (DOM) Level 3 Core Specification.* W3C. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/. Neueste Version verfügbar unter http://www.w3.org/TR/DOM-Level-3-Core/.

Li, D. (2011). A tale of the abt family of ot algorithms: from correctness to performance toward applications.

Li, D. & Li, R. (2007). A new operational transformation framework for real-time group editors. *IEEE Transactions on Parallel & Distributed Systems*, *18*(undefined), 307–319. doi:doi.ieeecomputersociety.org/10.1109/TPDS.2007.35

Li, D. & Li, R. (2008 Dezember). An approach to ensuring consistency in peer-to-peer real-time group editors. *Comput. Supported Coop. Work*, *17*(5-6), 553–611. doi:10.1007/s10606-005-9009-5

Li, D. & Li, R. (2010 Februar). An admissibility-based operational transformation framework for collaborative editing systems. *Comput. Supported Coop. Work*, *19*(1), 1–43. doi:10.1007/s10606-009-9103-1

Li, R. & Li, D. (2005). A landmark-based transformation approach to concurrency control in group editors. In *Proceedings of the 2005 international acm siggroup conference on supporting group work* (S. 284–293). GROUP '05. Sanibel Island, Florida, USA: ACM. doi:10.1145/1099203.1099252

Li, R., Li, D. & Sun, C. (2004). A time interval based consistency control algorithm for interactive groupware applications. In *Proceedings of the parallel and distributed systems, tenth international conference* (S. 429–). ICPADS '04. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICPADS.2004.12

Lv, X., He, F., Cai, W. & Cheng, Y. (2016). An efficient collaborative editing algorithm supporting string-based operations. In *20th IEEE international conference on computer supported cooperative work in design, CSCWD 2016, nanchang, china, may 4-6, 2016* (S. 45–50). doi:10.1109/CSCWD.2016.7565961

Martin, S., Urso, P. & Weiss, S. (2010). Scalable XML collaborative editing with undo short paper. *CoRR*, *abs/1010.3615*. Zugriff unter http://arxiv.org/abs/1010.3615

Molli, P., Skaf-Molli, H., Oster, G. & Jourdain, S. (2002). SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments. In *Seventh International Conference on Computer Supported Cooperative Work in Design - CSCWD'02* (5 p). Colloque avec actes et comité de lecture. internationale. Rio de Janeiro, Brasil. Zugriff unter https://hal.inria.fr/inria-00107571

Myers, E. W. (1986). An o(nd) difference algorithm and its variations. *Algorithmica*, *1*, 251–266.

Nichols, D. A., Curtis, P., Dixon, M. & Lamping, J. (1995). High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual acm symposium on user interface and software technology* (S. 111–120). UIST '95. Pittsburgh, Pennsylvania, USA: ACM. doi:10.1145/215585.215706

Oster, G. [G.], Imine, A., Molli, P. & Urso, P. (2006). Tombstone transformation functions for ensuring consistency in collaborative editing systems. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, *00*, 38. doi:doi.ieeecomputersociety.org/10.1109/COLCOM.2006.361867

Oster, G. [Gérald], Urso, P., Molli, P. & Imine, A. (2005). *Real time group editors without Operational transformation* (Research Report Nr. RR-5580). INRIA. Zugriff unter https://hal.inria.fr/inria-00071240

Preguiça, N., Shapiro, M. & Legatheaux Martins, J. (2008). *Designing a commutative replicated data type for cooperative editing systems.* Universidade

Nova de Lisboa, Dep. Informatica, FCT. LIP6 REGAL. Universidade Nova de Lisboa, Dep. Informatica, FCT.

Randolph, A., Boucheneb, H., Imine, A. & Quintero, A. (2015). On synthesizing a consistent operational transformation approach. *IEEE Trans. Computers*, *64*(4), 1074–1089. Zugriff unter http://dblp.uni-trier.de/db/journals/tc/tc64.html#RandolphBIQ15

Ressel, M., Nitsche-Ruhland, D. & Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 acm conference on computer supported cooperative work* (S. 288–297). CSCW '96. Boston, Massachusetts, USA: ACM. doi:10.1145/240080.240305

Roh, H.-G., Jeon, M., Kim, J.-S. & Lee, J. (2011 März). Replicated abstract data types: building blocks for collaborative applications. *J. Parallel Distrib. Comput. 71*(3), 354–368. doi:10.1016/j.jpdc.2010.12.006

Shao, B., Li, D. & Gu, N. (2009). An optimized string transformation algorithm for real-time group editors. In *Proceedings of the 2009 15th international conference on parallel and distributed systems* (S. 376–383). ICPADS '09. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICPADS.2009.72

Shao, B., Li, D. & Gu, N. (2010a). A sequence transformation algorithm for supporting cooperative work on mobile devices. In *Proceedings of the 2010 acm conference on computer supported cooperative work* (S. 159–168). CSCW '10. Savannah, Georgia, USA: ACM. doi:10.1145/1718918.1718949

Shao, B., Li, D. & Gu, N. (2010b). An algorithm for selective undo of any operation in collaborative applications. In *Proceedings of the 16th acm international conference on supporting group work* (S. 131–140). GROUP '10. Sanibel Island, Florida, USA: ACM. doi:10.1145/1880071.1880093

Shao, B., Li, D., Lu, T. & Gu, N. (2011). An operational transformation based synchronization protocol for web 2.0 applications. In *Proceedings of the acm 2011 conference on computer supported cooperative work* (S. 563–572). CSCW '11. Hangzhou, China: ACM. doi:10.1145/1958824.1958910

Shapiro, M. & Preguiça, N. M. (2007). Designing a commutative replicated data type. *CoRR*, *abs/0710.1784*. Zugriff unter http://dblp.uni-trier.de/db/journals/corr/corr0710.html#abs-0710-1784

Shapiro, M., Preguiça, N., Baquero, C. & Zawirski, M. (2011 Januar). *A comprehensive study of Convergent and Commutative Replicated Data Types* (rr Nr. 7506). inria. rocq.

Shen, H. & Sun, C. [Chengzheng]. (2002). Flexible notification for collaborative systems. In *Proceedings of the 2002 acm conference on computer supported cooperative work* (S. 77–86). CSCW '02. New Orleans, Louisiana, USA: ACM. doi:10.1145/587078.587090

Spiewak, D. (2010). Understanding and applying operational transformation. Zugriff unter http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation

Sun, C. [C.], Yang, Y., Zhang, Y. & Chen, D. (1996). A consistency model and supporting schemes for real-time cooperative editing systems.

Sun, C. [Chengzheng]. (2000). Undo any operation at any time in group editors. In *Proceedings of the 2000 acm conference on computer supported cooperative work* (S. 191–200). CSCW '00. Philadelphia, Pennsylvania, USA: ACM. doi:10.1145/358916.358990

Sun, C. [Chengzheng] & Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 acm conference on computer supported cooperative work* (S. 59–68). CSCW '98. Seattle, Washington, USA: ACM. doi:10.1145/289444.289469

Sun, C. [Chengzheng], Jia, X., Zhang, Y., Yang, Y. & Chen, D. (1998 März). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact. 5*(1), 63–108. doi:10.1145/274444.274447

Sun, D. & Sun, C. [Chengzheng]. (2006). Operation context and context-based operational transformation. In *Proceedings of the 2006 20th anniversary conference on computer supported cooperative work* (S. 279–288). CSCW '06. Banff, Alberta, Canada: ACM. doi:10.1145/1180875.1180918

Urso, P., Molli, P. & Weiss, S. (2009). Logoot-undo: distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel & Distributed Systems, 21*(undefined), 1162–1174. doi:doi.ieeecomputersociety.org/10.1109/TPDS.2009.173

Vidot, N., Cart, M., Ferrié, J. & Suleiman, M. (2000). Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 acm conference on computer supported cooperative work* (S. 171–180). CSCW '00. Philadelphia, Pennsylvania, USA: ACM. doi:10.1145/358916.358988

Wang, D., Mah, A. & Lassen, S. (2010). Google wave operational transformation. Zugriff unter https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html

Wang, X., Bu, J. & Chen, C. (2002). A new consistency model in collaborative editing systems. In *Proceedings of the 4 th international workshop on collaborative editing.*

Weiss, S., Urso, P. & Molli, P. (2009). Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proceedings of the 2009 29th ieee international conference on distributed computing systems* (S. 404–412). ICDCS '09. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICDCS.2009.75

Whitelaw, C., Dan, D., Mah, A. & Wang, D. (2009). Google wave: under the hood. Zugriff unter http://www.youtube.com/watch?v=uOFzWZrsPV0

Wu, S. & Manber, U. (1992 Oktober). Fast text searching: allowing errors. *Commun. ACM*, *35*(10), 83–91. doi:10.1145/135239.135244

Xu, Y. & Sun, C. [Chengzheng]. (2016 März). Conditions and patterns for achieving convergence in ot-based co-editors. *IEEE Trans. Parallel Distrib. Syst.* *27*(3), 695–709. doi:10.1109/TPDS.2015.2412938

Xu, Y., Sun, C. & Li, M. (2014). Achieving convergence in operational transformation: conditions, mechanisms and systems. In *Proceedings of the 17th acm conference on computer supported cooperative work & social computing* (S. 505–518). CSCW '14. Baltimore, Maryland, USA: ACM. doi:10.1145/2531602.2531629

Xue, L., Orgun, M. A. & Zhang, K. (2002). A user-centred consistency model in real-time collaborative editing systems. In *Revised papers from the 4th international workshop on distributed communities on the web* (S. 138–150). DCW '02. London, UK, UK: Springer-Verlag. Zugriff unter http://dl.acm.org/citation.cfm?id=647805.760766