TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

Diplomarbeit

Sperrverfahren für kooperative Transaktionsmodelle

vorgelegt von

Francis Gropengießer
Matrikel-Nr: 35278

Betreuer:

Prof. Dr.–Ing. Kai-Uwe Sattler
Dipl.–Inf. Katja Hose

Ilmenau, den 24. Januar 2008

Zusammenfassung

In vielen Anwendungsbereichen, wie z.B. dem Design und dem Medienproduktionsprozess, ist das kooperative Arbeiten mehrerer Nutzer auf einem zentralen Datenbestand unerlässlich geworden. Ein spezieller Anwendungsfall aus dem Bereich der Medienproduktion ist der Ausgangspunkt für die Betrachtungen in dieser Arbeit. Aus ihm lässt sich eine Reihe von Anforderungen, u.a. die Unterstützung des Kooperationsprinzips, an den Aufbau eines Transaktionsmodells, ein Sperrprotokoll sowie ein Fehlerbehandlungsmodell ableiten. Es wird für eine ausgewählte Menge existierender Transaktions-/Aktionsmodelle gezeigt, dass diese die Anforderungen nicht im vollen Maße erfüllen. In der Arbeit wird daher ein neues Aktionsmodell, welches auf den geschachtelten dynamischen Aktionen für kooperative Anwendungen aufbaut, entwickelt und vorgestellt. Dabei wird sowohl auf den Aufbau des Modells, das Sperrprotokoll sowie das Fehlerbehandlungsmodell eingegangen. Des Weiteren werden Eigenschaften des Aktionsmodells erläutert und nachgewiesen. Eine Bewertung des Modells zeigt, dass es alle Anforderungen, vor allem die Unterstützung des Kooperationsprinzips, im vollen Umfang erfüllt und dabei noch einige Besonderheiten, wie z.B. das Reset-Repeat-Prinzip und die Möglichkeit, auf den Einsatz eines CVS zu verzichten, bietet. Anschließend wird eine mögliche Systemarchitektur für das entwickelte Gesamtmodell beschrieben. Diese umfasst den UML-Entwurf eines *Actionmanagers*, *Lockmanagers*, *Objectversionmanagers* und *Logbookmanagers* sowie die Einbettung dieser Komponenten in die Client-Server-Architektur.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.1.1 Ein spezieller Anwendungsfall . . . 2

1.2 Gegenstand der Arbeit . . . 3

1.3 Aufbau der Arbeit . . . 4

2 Theoretische Grundlagen **5**

2.1 Das Transaktionskonzept . . . 5

2.1.1 Die Transaktion . . . 5

2.1.2 Der Schedule . . . 6

2.2 Das ACID-Prinzip . . . 7

2.3 Das Auftrags- und Kooperationsprinzip . . . 9

2.4 XML . . . 10

2.5 DOM . . . 11

2.6 Die taDOM-Spezifikation . . . 12

2.7 Das DeweyID-Konzept . . . 13

3 Bewertung existierender Transaktionsmodelle **16**

3.1 Die Bewertungskriterien . . . 16

3.2 Die geschachtelten Transaktionen . . . 17

3.2.1 Geschlossen geschachtelte Transaktionen . . . 18

3.2.2 Offen geschachtelte Transaktionen . . . 19

3.2.3 Die Vereinigung des Auftrags- und Kooperationsprinzips . . . 22

3.3 Dynamische Aktionen . . . 24

3.4 Geschachtelte dynamische Aktionen . . . 26

3.5 Geschachtelte dynamische Aktionen für kooperative Anwendungen . . . 27

4 Das Modell **30**

4.1 Die Operationen . . . 30

4.2 Der Aufbau des geschachtelten Aktionsmodells . . . 31

4.3 Das Sperrmodell . . . 37

4.3.1 Untersuchung der Kompatibilität der Operationen . . . 37

4.3.2 Die Sperrtypen . . . 40

4.3.3 Das Sperrprotokoll . . . 41

4.4 Das Fehlerbehandlungsmodell . . . 44

4.4.1 Das Objektversionensystem . . . 45

4.4.2 Das Log-Buch . . . 46

4.4.3 Darstellung der Funktionsweise des Log-Buches anhand eines Beispiels 49

4.4.4 Die Verwaltung und Pflege der Log-Bücher . . . 56

4.5 Die Eigenschaften des Modells 57
4.5.1 Untersuchung des Modells hinsichtlich der ACID-Eigenschaften 58
4.5.2 Die Fortschreibbarkeit der Commit-Linie 61
4.5.3 Die Sichtbarkeit von Objektänderungen 64
4.5.4 Der Grad der Nebenläufigkeit 64
4.5.5 Untersuchung der Kooperativität des Modells 65
4.5.6 Die Deadlockfreiheit des Modells 66
4.5.7 Eigenschaften, die aus dem verwendeten Fehlerbehandlungsmodell resultieren 67
4.6 Bewertung und Vergleich 68

5 Systementwurf **70**
5.1 Objektorientierter Entwurf des Modells 70
5.1.1 Der Actionmanager 70
5.1.2 Der Lockmanager 71
5.1.3 Der Objectversionmanager 75
5.1.4 Der Logbookmanager 75
5.2 Einbettung des Modells in die Client-Server-Architektur 78
5.2.1 Der Server 78
5.2.2 Der Client 80
5.2.3 Beispielhafte Darstellung der Kommunikation zwischen den Klassen . 84

6 Zusammenfassung und Ausblick **88**

Literaturverzeichnis **90**

Abbildungsverzeichnis **92**

Tabellenverzeichnis **93**

A Thesen **94**

B Eidesstattliche Erklärung **95**

1 Einleitung

1.1 Motivation

In den letzten Jahren hat der Aspekt des kooperativen Mehrbenutzerbetriebs einer Datenbank immer mehr an Bedeutung gewonnen. In vielen Anwendungsbereichen, wie z.B. im Design oder dem Medienproduktionsprozess, auf den später noch genauer eingegangen wird, ist ein kooperatives Arbeiten mehrerer Nutzer auf einem gemeinsamen Datenbestand bzw. an einem gemeinsamen Projekt zwingend erforderlich. Nur so können die Ideen und Vorschläge jedes einzelnen Beteiligten berücksichtigt werden und in die Erstellung eines Gesamtergebnisses oder eines Endproduktes mit einfließen.

Kooperativität lässt sich wie folgt erläutern. Ein Nutzer A erstellt ein Objekt oder ein Teilergebnis. Dieses wird durch einen Nutzer B begutachtet und gegebenenfalls weiterbearbeitet. Anschließend greift Nutzer A wieder auf dieses Objekt zu. Die Ermöglichung eines derartigen Arbeitsprozesses stellt enorme Anforderungen an die Entwickler solcher kooperativer Datenbanksysteme. Dies lässt sich dadurch begründen, dass das kooperative Arbeiten einen Verlust der Serialisierbarkeit und damit ein mögliches Eintreten von Mehrbenutzeranomalien bedeutet. Auf diese Aspekte wird im Verlauf der Arbeit noch häufiger eingegangen. Die Schwerpunkte bei der Entwicklung kooperativer Datenbanksysteme sind also die Schaffung geeigneter Synchronisationsmechanismen (z.B. Sperrprotokolle), die den konkurrierenden Zugriff auf Objekte kontrollieren, dabei jedoch die Kooperation ermöglichen, und geeigneter Fehlerbehandlungsmechanismen, die einerseits die Behandlung herkömmlicher Transaktionsabbrüche und andererseits das Beseitigen von Inkonsistenzen, die infolge der Kooperation entstehen können, realisieren.

Der Medienproduktionsprozess ist, wie eingangs erwähnt, einer der Anwendungsbereiche, in dem Kooperation erforderlich ist. Er umfasst die Produktion von Filmen, Hörspielen, Musik usw. Dieser Prozess lässt sich in die Phasen Preproduktion, Produktion, Postproduktion und Distribution unterteilen. Die Preproduktion dient der Inhaltsrecherche und Produktionsplanung. In der Produktionsphase werden die einzelnen Bestandteile (z.B. Bild und Ton) erstellt und an das Vermittlungsmedium (z.B. Internet) angepasst. In der Postproduktionsphase erfolgt das Zusammenfügen aller Bestandteile zu einer finalen Version. In der letzten Phase, der Distribution, wird das fertige Produkt an die Zielgruppe verteilt.

Im nächsten Abschnitt wird auf die Postproduktion von Filmton eingegangen, da sich das Thema dieser Arbeit aus diesem speziellen Anwendungsgebiet heraus ergeben hat.

1.1.1 Ein spezieller Anwendungsfall

Nach dem Aufnehmen aller für einen Film benötigten Sounds müssen diese geschnitten und abgemischt werden. Gemischt wird im professionellen Bereich meist für 5.1-Kanalsysteme. Diese bieten allerdings nur für die sich im so genannten „Sweet-Spot“ befindenden Hörer das volle Raumklangerlebnis. Daher entwickelte das Fraunhofer-Institut für Digitale Medientechnologie (IDMT) auf Basis der Wellenfeldsynthese [WFS] ein neuartiges Soundsystem - IOSONO [IOS], welches diese Einschränkung nicht besitzt. Damit können Schallquellen akustisch im dreidimensionalen Raum positioniert werden, wodurch ein natürlicher und räumlich realitätsgetreuer Klangeindruck entsteht.

Ein Film besteht aus vielen Szenen. Für jede Szene werden die verschiedenen Sounds bzw. Soundobjekte mit dem vom Fraunhofer IDMT entwickelten IOSONO-System im dreidimensionalen Raum positioniert. Die dabei entstehenden Metadaten werden in einem Szenegraph, der nichts anderes als ein XML-Dokument ist, abgelegt. In Abbildung 1.1 ist ein Beispiel für solch einen Szenegraph dargestellt. Gut zu erkennen sind die verschiedenen Arbeitsbereiche einer

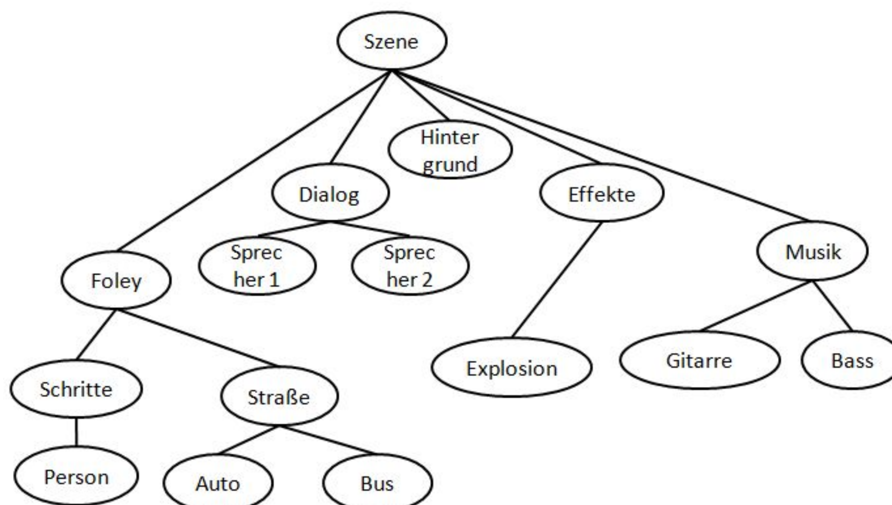

Abbildung 1.1: Beispiel für einen Szenegraph

Sound-Szene: Foley, Dialog, Effekt, Hintergrund und Musik. Für jeden dieser Arbeitsbereiche existiert ein spezielles Team, welches kooperativ an dessen Fertigstellung arbeitet. Allerdings muss auch team- und damit arbeitsbereichsübergreifend gearbeitet werden, um beispielsweise die exakte zeitliche Abfolge der einzelnen Soundobjekte festzulegen. Auch dies erfordert die kooperative Zusammenarbeit verschiedener Designer, da über die endgültige zeitliche Position eines Soundobjekts abgestimmt werden muss. Ein Transaktionsmodell, welches die Arbeiten der Designer an solch einem Szenegraph abbildet, muss also ein kooperatives Arbeiten, sowohl innerhalb der Teams als auch teamübergreifend, ermöglichen.

Das Arbeiten mehrerer Designer an einer gemeinsamen Szene kann als Workgroup realisiert werden. Abbildung 1.2 zeigt beispielhaft die Architektur des IOSONO-Systems im Workgroup-Betrieb. Nachfolgend soll kurz skizziert werden, wie in diesem System gearbeitet wird:

- Jeder Designer lädt zu Beginn seiner Arbeiten eine Kopie der Szene von der zentralen Datenbasis in seinen lokalen Cache.

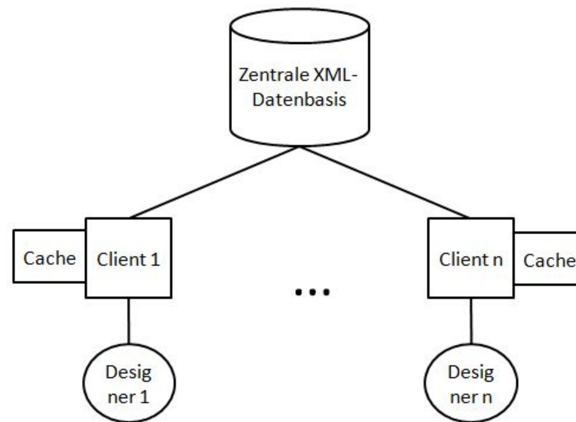

Abbildung 1.2: Architektur des IOSONO-Systems im Workgroup-Betrieb

- Anschließend führen die Designer ihre Tätigkeiten im lokalen Cache aus.
- Nach Abschluss der Arbeiten propagieren sie die neuen Versionen wieder zur zentralen Datenbasis. Dort wird mit Hilfe eines CVS (Concurrent Versions System) aus den verschiedenen Versionen einer Szene eine finale Version gebildet.

Dabei entstehen allerdings aus der Sicht eines Datenbanksystementwicklers folgende Probleme:

- Die Transaktion vom Auschecken der Szene bis zum Wiedereinchecken der bearbeiteten Szene ist sehr lang. Wird sie abgebrochen, so kann die Arbeit mehrerer Stunden verloren gehen. Es werden also erweiterte Transaktionskonzepte, z.B. geschachtelte Transaktionen, und entsprechende Fehlerbehandlungsmodelle benötigt, die diesem Problem begegnen.
- Ein Designer kann die Arbeiten eines anderen frühestens sehen, wenn dieser die betroffenen Szenen wieder beim Server eingecheckt hat. Dies unterbindet natürlich jegliche Form der Kooperation. Es werden also Transaktions- und entsprechende Synchronisationskonzepte benötigt, die eine vorzeitige Sichtbarkeit von (Teil-)Ergebnissen und ein kooperatives Arbeiten mit diesen Ergebnissen ermöglichen.
- Der Einsatz eines CVS kann einen hohen zusätzlichen Arbeitsaufwand bedeuten, da das Verschmelzen der unterschiedlichen Versionen eines Szenegraphs nicht ohne das Einwirken der Nutzer möglich ist.

1.2 Gegenstand der Arbeit

Gegenstand dieser Arbeit ist die Lösung der im vorhergehenden Abschnitt angesprochenen Probleme, die im Workgroup-Betrieb des IOSONO-Systems auftreten. Im Detail bedeutet das:

- Es muss ein geeignetes Transaktionsmodell gefunden werden, welches den Problemen der langen Transaktionen begegnet.
- Dies erfordert auch die Entwicklung eines entsprechenden Sperrprotokolls, welches vorzeitige Objektfreigaben zulässt und kooperativ ist.
- Zur Behandlung von Transaktionsfehlern muss ein geeignetes Fehlerbehandlungsmodell

entwickelt werden. Dies muss möglichst unabhängig vom Anwender sein, da nicht von einem Designer erwartet werden kann, dass er sich mit Datenbankinterna auseinandersetzt.

- Es muss eine Systemarchitektur entworfen werden, die zeigt, welche Systemkomponente welche Aufgaben übernimmt. Dabei stellen sich beispielsweise Fragen wie: Soll die Fehlerbehandlung zentral oder verteilt realisiert werden? Wer übernimmt die Transaktionsverwaltung?

1.3 Aufbau der Arbeit

Der restliche Teil der Arbeit ist wie folgt gegliedert. Im nächsten Kapitel erfolgt zunächst eine Klärung der für das Verständnis der Arbeit notwendigen Begriffe und Konzepte. Im Kapitel 3 werden einige ausgewählte erweiterte Transaktionskonzepte vorgestellt und hinsichtlich einiger Kriterien, die sich aus dem speziellen Anwendungsfall aus Abschnitt 1.1.1 ergeben, verglichen. Gegenstand des darauf folgenden Kapitels 4 ist die Vorstellung des entwickelten Modells. Dabei wird auf das Transaktionskonzept, das dazugehörige Sperrprotokoll sowie auf das Fehlerbehandlungsmodell eingegangen. Weiterhin werden einige wichtige Eigenschaften dieses Modells beschrieben und nachgewiesen. Danach erfolgt eine Bewertung hinsichtlich der im Kapitel 3 formulierten Kriterien und ein Vergleich mit den ausgewählten Transaktionsmodellen. Im Kapitel 5 wird eine Systemarchitektur in Form eines UML-Entwurfs vorgeschlagen. Dieser beinhaltet eine Spezifikation der für die Umsetzung des Modells notwendigen Klassen und Methoden sowie die Einbettung derselbigen in die Client-Server-Architektur. Im Kapitel 6 wird die Arbeit zusammengefasst und ein kurzer Ausblick auf zukünftige Aufgaben gegeben.

2 Theoretische Grundlagen

Gegenstand dieses Kapitels ist die Klärung wichtiger Begriffe und Konzepte, die für das Verständnis der in dieser Arbeit erläuterten Sachverhalte unbedingt erforderlich sind. Zunächst wird dabei das grundlegende Transaktionskonzept vorgestellt, auf dem die im weiteren Verlauf der Arbeit erläuterten erweiterten Transaktionskonzepte und die dynamischen Aktionen aufbauen. Anschließend erfolgt die Beschreibung des ACID-Prinzips, welches sich im Datenbankbereich als Möglichkeit zum Vergleich verschiedener Transaktionssysteme durchgesetzt hat. Ein weiterer Abschnitt widmet sich der Erläuterung des Auftrags- und des Kooperationsprinzips, da vorrangig die Kooperativität eines Transaktionsmodells ein entscheidendes Kriterium im Kontext der verteilten Verarbeitung zentral gehaltener Daten darstellt. Innerhalb der letzten vier Abschnitte wird zunächst auf die XML-Spezifikation eingegangen und anschließend einige spezielle Modelle bzw. Konzepte, die den Umgang mit XML-Daten erheblich erleichtern, vorgestellt.

2.1 Das Transaktionskonzept

In diesem Abschnitt soll das Transaktionskonzept näher erläutert werden.

2.1.1 Die Transaktion

Unter einer Transaktion wird im klassischen Sinne eine Folge von Operationen verstanden, die eine Datenbank von einem konsistenten in einen anderen, gegebenenfalls veränderten, konsistenten Zustand überführt. Dabei hält sie das in Abschnitt 2.2 vorgestellte ACID-Prinzip ein. (Vgl. [SHK05])

Für die klassische Transaktion gilt das in Abbildung 2.1 dargestellte Zustandsmodell. Durch einen Aufruf von *begin()* wird eine Transaktion gestartet und befindet sich danach im Zustand *Active*, in dem die Operationen abgearbeitet werden, die sie beinhaltet. Wurde die Transaktion erfolgreich ausgeführt, wird sie durch einen Aufruf von *commit()* in den Zustand *Committed* versetzt. Trat während der Abarbeitung der Operationen ein Fehler auf oder ist es der explizite Wunsch des Nutzers, so wird die Transaktion mit *abort()* abgebrochen und alle ihre Änderungen rückgängig gemacht. An dieser Stelle soll noch eine Anmerkung zum Zustandsübergang $Committed \rightarrow Aborted$ erfolgen. Für Transaktionen wird die Dauerhaftigkeit der Änderungen nach dem Übergang in den Zustand *Committed* gefordert. Allerdings tritt beispielsweise bei den offen geschachtelten Transaktionen in Abschnitt 3.2.2 das Problem auf, dass ein Rückgängigmachen ihrer Änderungen nach einem *Commit* erforderlich sein kann. Dies wird mit Hilfe eines semantischen *Aborts* unter Nutzung von Kompensationstransaktionen (Abschnitt 2.2) realisiert.

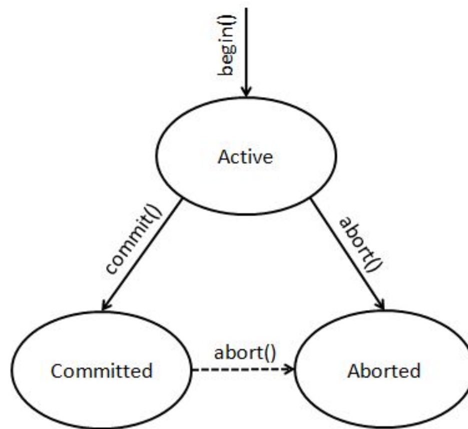

Abbildung 2.1: Zustände einer klassischen Transaktion

Abschließend sollen noch einige wesentliche Ursachen für den Abbruch einer Transaktion genannt werden:

- Es kann der explizite Wunsch eines Nutzers sein, dass die Transaktion abgebrochen wird.
- Der Abbruch kann durch das System erfolgen. Mögliche Gründe dafür könnten sein:
  - Es trat ein Systemfehler auf.
  - Die Transaktion hat eine Integritätsbedingung (Abschnitt 2.2) verletzt.
- Die Transaktion ist das Opfer eines kaskadierenden Abbruchs. Dieser tritt auf, wenn die Transaktion die Änderungen einer anderen Transaktion gelesen hat, die abgebrochen wird. Ein Beispiel stellt der folgende Schedule (Abschnitt 2.1.2) dar:

  $$Schedule = Write_1(a)Read_2(a)Abort_1()Abort_2() \tag{2.1}$$

  Die Transaktion *2* liest das von Transaktion *1* geschriebene Objekt *a*. Transaktion *1* wird durch z.B. einen Nutzer abgebrochen. Nun muss auch Transaktion *2* abgebrochen werden, da sie einen ungültigen Objektzustand gelesen hat.

2.1.2 Der Schedule

In diesem Abschnitt soll der Begriff des Schedules kurz erläutert werden, da er im Verlauf der Arbeit häufig verwendet wird.

Unter einem Schedule versteht man die verschränkte Ausführung mehrerer Transaktionen, wobei die relative Reihenfolge der Operationen einer Transaktion erhalten bleibt. (Vgl. [SHK05])

Seien

$$Transaktion_1 = Begin_1()Read_1(a)Write_1(a)Read_1(b)Write_1(b)Commit_1() \tag{2.2}$$
$$Transaktion_2 = Begin_2()Read_2(c)Write_2(c)Read_2(a)Write_2(a)Commit_2() \tag{2.3}$$

zwei Transaktionen. So stellt

$$\begin{aligned} Schedule \quad = {} & Begin_1()Begin_2()Read_1(a)Write_1(a) \\ & Read_2(c)Write_2(c)Read_1(b)Write_1(b) \\ & Read_2(a)Write_2(a)Commit_1()Commit_2() \end{aligned} \tag{2.4}$$

einen möglichen Schedule dar.

2.2 Das ACID-Prinzip

Die Abkürzung ACID steht für folgende Eigenschaften einer Transaktion.
(Vgl. [SHK05, SST97])

Atomicity

Darunter versteht man, dass entweder alle Operationen einer Transaktion erfolgreich ausgeführt werden oder die gesamte Transaktion scheitert. Eine Transaktion wird also nur vollständig oder gar nicht ausgeführt. Sie hinterlässt somit keine unscharfen Zustände.

Weiterhin soll der Begriff der Fehler-Atomarität eingeführt werden. Diese stellt eine abgeschwächte Form der Atomarität dar. Sie fordert lediglich, dass abgebrochene Transaktionen keine Effekte in der Datenbank hinterlassen. Werden also die Änderungen einer Transaktion nach ihrem Abbruch vollständig rückgängig gemacht, so ist die Fehler-Atomarität gewährleistet.

Die Gewährleistung der Atomarität ist Sache des Fehlerbehandlungsmodells. Dieses kann auf verschiedene Arten realisiert werden. Nachfolgend sollen zwei genannt werden, die im weiteren Verlauf der Arbeit eine Rolle spielen:

- Die Kompensationstransaktion macht die Änderungen einer Transaktion auf semantischer Ebene rückgängig. Beispielsweise würde die Subtraktion eines Wertes a von einer Summe S die Addition von a auf S rückgängig machen.
- Die Verwaltung von Objektversionen stellt eine weitere Möglichkeit der Fehlerbehandlung dar. Jede Änderung führt zu einer neuen Version eines Objektes. Soll die Änderung rückgängig gemacht werden, so wird einfach die entsprechende Objektversion gelöscht. Nähere Informationen dazu werden in Abschnitt 4.4.1 gegebenen.

Consistency

Eine Transaktion überführt eine Datenbank von einem konsistenten in einen anderen, gegebenenfalls veränderten, konsistenten Zustand. Sie hält also alle für die Datenbank definierten Integritätsbedingungen ein. Integritätsbedingungen können z.B. Wertebereichsdefinitionen, Schlüssel- und Fremdschlüsselbeziehungen oder auch die Struktur eines XML-Dokumentes sein.

Isolation

Die Isolation fordert im Mehrbenutzerbetrieb einer Datenbank, dass die Transaktion eines Nutzers die Änderungen einer anderen Transaktion erst nach deren Übergang in den Zustand *Committed* sieht. Dadurch beeinflussen sich die verschiedenen Transaktionen nicht gegenseitig. Jeder Nutzer hat somit den Eindruck, er würde allein auf der Datenbank arbeiten.

	T1	T2
1	$WL(a)$	–
2	–	$WL(b)$
3	$WL(b)$	–
4	–	$WL(a)$

Tabelle 2.1: Beispiel für eine Verklemmung zweier Transaktionen T1 und T2

Die Serialisierbarkeit stellt eine Abschwächung der Isolation dar. Hier wird lediglich gefordert, dass eine verschränkte Ausführung mehrerer Transaktionen das gleiche Ergebnis liefert, wie ihre serielle Ausführung. Es wird nicht gefordert, dass eine Transaktion die Änderungen einer anderen Transaktion erst nach deren Übergang in den Zustand *Committed* lesen darf. Serialisierbarkeit vermeidet das Auftreten so genannter Mehrbenutzeranomalien, wie z.B. das Non-Repeatable-Read oder das Phantom-Problem. (Siehe dazu [SHK05].) Ein Beispiel für einen serialisierbaren Schedule ist in Gleichung 2.4 dargestellt. Er hinterlässt das gleiche Ergebnis wie die serielle Ausführung der beiden Transaktionen aus Gleichung 2.2 und Gleichung 2.3 in der Reihenfolge $Transaktion_1 Transaktion_2$.

Eine Möglichkeit zur Gewährleistung der Isolation bzw. der Serialisierbarkeit ist die Verwendung eines Sperrmodells. Dieses regelt den konkurrierenden Zugriff mehrerer Transaktionen auf gemeinsame Daten. Ein Sperrmodell umfasst im Wesentlichen folgende Aspekte:

- Eine Menge von Sperrtypen für verschiedene Operationen (z.B. eine Lesesperre für eine Leseoperation oder eine Schreibsperre für eine Schreiboperation)
- Eine Kompatibilitätsmatrix, die die Verträglichkeit der Sperren darstellt (z.B. eine Lesesperre und eine Schreibesperre auf dem gleichen Objekt sind nicht zulässig)
- Ein Sperrprotokoll, welches das Setzen und Freigeben der Sperren durch die Transaktionen regelt (z.B. vor der Ausführung einer Schreiboperation muss eine Schreibsperre für das Objekt erlangt werden)

Ein Beispiel für ein Sperrprotokoll ist das Baumprotokoll [SHK05]. Mit diesem kann das Sperren von Datenstrukturen, die in Baumform organisiert sind, realisiert werden. Jede Transaktion darf nach bestimmten Regeln Knoten in dem Baum sperren, wobei es nicht erforderlich ist, ganze Teilbäume zu sperren. In [HS07] wurde beispielsweise ein erweitertes Baumprotokoll zur Synchronisation des konkurrierenden Zugriffs auf XML-Dokumente eingesetzt.

Ein Problem, welches beim Sperren von Objekten auftreten kann, ist, dass es zur Verklemmung (*Deadlock*) von Transaktionen kommen kann. Siehe dazu Tabelle 2.1. Hier wartet Transaktion *T1* auf die Freigabe der Schreibsperre (*WL*) auf *b* durch *T2*. Umgekehrt wartet aber auch *T2* auf die Freigabe der Schreibsperre auf *a* durch *T1*. Keine der beiden Transaktionen kann nun weiterarbeiten. Es gibt zwei Möglichkeiten solchen Deadlocks zu begegnen. Sie können entweder nach ihrem Auftreten erkannt und beseitigt oder von vornherein vermieden werden.

Neben den Sperrverfahren zur Nebenläufigkeitskontrolle gibt es z.B. auch nicht-sperrende Verfahren, optimistische Verfahren und Multiversionen-Synchronisation. (Vgl. [SHK05]) Zu den nicht-sperrenden Verfahren gehört beispielsweise das Zeitstempelverfahren. Dieses erzwingt eine serialisierbare Ausführungsreihenfolge der einzelnen Operationen durch eine Markierung der Transaktionen. Bei den optimistischen Verfahren wird davon ausgegangen, dass Konflikte nur selten auftreten. Daher werden zunächst alle Operationen ohne besondere Vorkehrungen

ausgeführt. Ist bei der Ausführung dennoch ein Konflikt aufgetreten, so wird dieser erkannt und beseitigt. Bei der Multiversionen-Synchronisation gibt es von jedem Objekt mehrere Versionen. Alle Operationen können bei einem Zugriff auf ein Objekt die zur Wahrung der Serialisierbarkeit erforderliche Version aus der Menge der Objektversionen wählen.

Durability

Nach dem erfolgreichen Abschluss einer Transaktion werden ihre Änderungen dauerhaft in der Datenbank gespeichert. Somit darf eine Transaktion nach ihrem Übergang in den Zustand *Committed* nicht mehr abgebrochen werden. Ansonsten müssten ihre Änderungen rückgängig gemacht werden.

2.3 Das Auftrags- und Kooperationsprinzip

Das Auftragsprinzip [Kru97] stellt eine Beziehung zwischen Auftraggeber und Auftragnehmer dar. Ein Nutzer beauftragt einen anderen mit der Bearbeitung einer Teilaufgabe, die dieser eigenständig erfüllt. Anschließend kann der Auftraggeber mit dem ermittelten Ergebnis weiterarbeiten. Dieses Prinzip birgt drei besondere Merkmale in sich, die nachfolgend dargestellt werden sollen:

- Die beteiligten Nutzer sind nicht gleichberechtigt. Der Auftraggeber steht über dem Auftragnehmer und teilt ihm das Stück Information zu, welches er benötigt, um die Aufgabe zu erfüllen. Der Auftragnehmer muss mit diesen Informationen auskommen, d.h. er kann sich keine weiteren von „außen“ beschaffen. Er besitzt somit auch keine Informationen über den Kontext der Aufgabe.
- Zwischen beiden Nutzern findet nur ein gerichteter Informationsfluss statt. Der Auftragnehmer bekommt einmalig eine Aufgabe und die benötigten Informationen zugewiesen und liefert einmalig das Ergebnis nach Beendigung seiner Arbeiten an den Auftraggeber zurück. Ansonsten findet keine Kommunikation zwischen beiden Nutzern statt. Somit hat der Auftragnehmer auch nicht die Möglichkeit, den Auftraggeber zu beeinflussen. Des Weiteren hat der Auftragnehmer nach der Rückgabe der Ergebnisse keine Kontrolle mehr über diese.
- Die Existenz des Auftragnehmers bleibt anderen Nutzern verborgen. Diese können also nicht direkt mit dem Auftragnehmer kommunizieren. Ein vom Auftragnehmer ermitteltes Ergebnis kann erst vom Auftraggeber anderen Nutzern zur Verfügung gestellt werden. Diese haben also den Eindruck, der Auftraggeber habe das Ergebnis selbst erarbeitet. Dieses Prinzip der Verborgenheit hat auch den Vorteil, dass falls der Auftragnehmer an der erteilten Aufgabe scheitert, dies nicht zwangsläufig dazu führt, dass der Auftraggeber kein Ergebnis liefern kann und auch scheitert. Er könnte die Aufgabe selbst lösen oder sie einem anderen Auftragnehmer übergeben, ohne dass andere Nutzer etwas bemerken.

Beim Kooperationsprinzip [Kru97] stehen die Nutzer, im Gegensatz zum Auftragsprinzip, gleichberechtigt in Beziehung. Sie arbeiten gemeinsam an der Erfüllung einer Aufgabe, indem sie die Arbeiten untereinander aufteilen. Es findet dabei ein Informationsfluss in beliebiger Richtung zwischen den Nutzern statt. Jeder Nutzer kann Einfluss auf die Arbeiten des anderen nehmen.

Beispielsweise kann ein Nutzer eine Lösung vorschlagen, die von anderen Nutzern begutachtet wird. Anschließend kann dann eine Verbesserung des Lösungsvorschlags stattfinden. Alle Nutzer stimmen im Prinzip in einer Art Verhandlung über ein finales Ergebnis ab.

Auch die Kombination beider Prinzipien ist möglich. Beispielweise können höher gestellte Nutzer kooperativ an einer Aufgabe arbeiten und über eine finale Lösung verhandeln. Die Lösung bestimmter Teilaufgaben wird jedoch an niedriger gestellte Nutzer übergeben, die keine Informationen über das Gesamtprojekt besitzen und von deren Existenz lediglich der Auftraggeber weiß.

2.4 XML

XML [XML] steht für *Extensible Markup Language* und stellt eine Metasprache, also eine Sprache zur Beschreibung von Sprachen dar. Mit ihr kann also sowohl die Grammatik einer Sprache als auch ihr Inhalt spezifiziert werden. XML wurde entwickelt, um plattformunabhängig Daten zwischen verschiedenen Anwendungen auszutauschen.

In Abbildung 2.2 ist ein Ausschnitt eines XML-Dokumentes dargestellt.

```
<Anschrift>
  <Name Vorname="Ich" Nachname="Bins"/>
  <Strasse>
    Musterstrasse
  </Strasse>
  <Hausnummer>
    1
  </Hausnummer>
  <Ort>
    Musterhausen
  </Ort>
</Anschrift>
```

Abbildung 2.2: Ausschnitt aus einem XML-Dokument

Nachfolgend sollen kurz die Bestandteile erläutert werden:

- Die Anschrift, der Name, die Strasse, die Hausnummer und der Ort sind *XML-Elemente*. Sie beginnen immer mit einem *Start-Tag* (< ... >) und enden mit einem *Ende-Tag* (< /... >). Ein Element kann beliebig geschachtelt werden. Beispielsweise enthält die Anschrift den Namen, die Straße usw. Es können aber auch *Informationen* in Form von *Text* enthalten sein, wie z.B. Musterstrasse, Musterhausen usw. Ist ein Element weder geschachtelt noch enthält es Informationen, kann das öffnende und schließende Tag zusammengefasst werden. (< .../ >) Dies ist beim Name der Fall.
- Der Vorname und Nachname sind *Attribute* des XML-Elementes Name. „Ich“ und „Bins“ sind die entsprechenden *Attributwerte*.

Natürlich ist die XML-Spezifikation noch wesentlich umfangreicher. Allerdings reichen die hier gegebenen Informationen für das Verständnis der Arbeit aus.

2.5 DOM

Das *Document Object Model* [DOM]) ist eine Möglichkeit, ein XML-Dokument als Baumstruktur im Speicher darzustellen. Es bietet auch eine entsprechende Schnittstelle, über die es möglich ist, wahlfrei auf die Objekte des Baumes zuzugreifen und diese zu bearbeiten.

Abbildung 2.3 stellt den Ausschnitt eines XML-Dokumentes aus dem vorhergehenden Abschnitt als DOM-Baum dar.

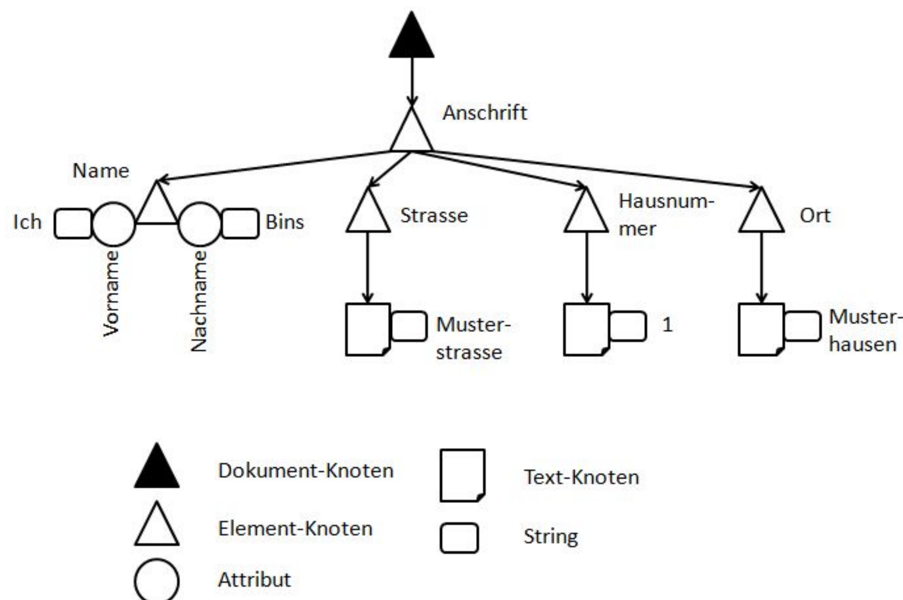

Abbildung 2.3: DOM-Baum des XML-Dokumentausschnitts aus Abbildung 2.2

Nachfolgend soll die Abbildung kurz erläutert werden:

- XML-Elemente werden als *Element-Knoten* abgebildet. Die Wurzel des Baumes bildet der *Dokument-Knoten*. (Das entsprechende Dokument-Element ist in Abbildung 2.2 nicht enthalten.) Auf alle Element-Knoten ist über die Schnittstelle des DOM ein wahlfreier Zugriff möglich.
- Attribute werden als *Attribut(knoten)* dargestellt. Allerdings werden sie im DOM nicht als Knoten behandelt, da kein direkter Zugriff auf sie möglich ist, sondern nur über das entsprechende Element, zu dem sie gehören. Sie stellen also nur eine Eigenschaft des Elementes dar.
- Attributwerte und Informationen werden als *String* im DOM abgebildet. Auch auf sie ist kein direkter Zugriff möglich.
- Informationen, die innerhalb eines Elementes gespeichert sind, werden an einen *Textknoten* angehängt. Auf diesen kann explizit zugegriffen werden.

Natürlich ist die DOM-Spezifikation noch wesentlich umfangreicher. Allerdings reichen die hier gegebenen Informationen für das Verständnis der Arbeit aus.

2.6 Die taDOM-Spezifikation

In [HH03] wurde taDOM (*tailored DOM*) eingeführt, um das feingranulare Sperren von Attributen, ihren Werten und dem reinen Text eines Textknotens zu ermöglichen. Der Grund dafür ist, dass im originalen DOM Attribute nicht als Knoten im XML-Baum betrachtet werden und daher der Zugriff auf ein Attribut nicht ohne den Zugriff auf das einhüllende XML-Element möglich ist. Eine Sperre auf ein Attribut erfordert daher auch eine Sperre auf das entsprechende XML-Element. Genauso verhält es sich mit einem Attributwert bezüglich eines Attributs und dem Text (den Informationen) bezüglich des Textknotens. Auch hier erfordert der Zugriff auf einen Wert (oder den Text) auch einen Zugriff auf das Attribut (oder den Textknoten) und somit Sperren auf beide Elemente. Daher führten die Autoren von [HH03] zwei zusätzliche Knoten im Baum ein. Der *Attribut-Wurzel-Knoten* gruppiert alle Attribute eines XML-Elements als Nachfolgerknoten dieses Elementes. Der *String-Knoten* bildet den Text bzw. den Wert eines Attributs bzw. Textknotens als Nachfolgerknoten dieser Elemente ab. Wie auch in [HS07] wird in dieser Arbeit auf den *String-Knoten* verzichtet, da davon ausgegangen wird, dass der Zugriff auf ein Attribut oder einen Textknoten immer auch einen Zugriff auf den Wert des Attributs bzw. den Text eines Textknotens bedeutet. Der Schwerpunkt liegt also auf der Modellierung von Attributen als Knoten im XML-Baum.

Das nachfolgende Beispiel soll den dargestellten Sachverhalt veranschaulichen. Abbildung 2.4 zeigt zunächst beispielhaft einen XML-Baum, aufgebaut nach der klassischen DOM-Spezifikation. Zu erkennen ist, dass die Attribute keine expliziten Knoten im Baum darstellen, sondern

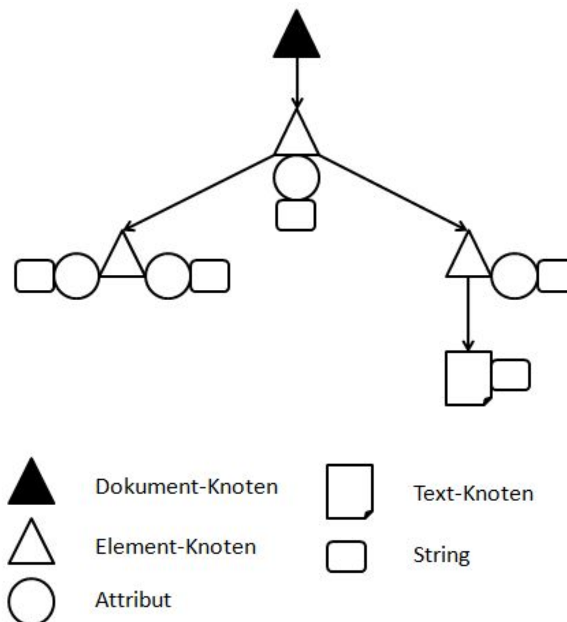

Abbildung 2.4: XML-Baum, aufgebaut nach der DOM-Spezifikation

zum XML-Element-Knoten gehören. Auch die Attributwerte und der Text werden nicht als eigenständige Knoten modelliert.

Abbildung 2.5 zeigt den gleichen XML-Baum, nun aufgebaut nach der taDOM-Spezifikation. Zu erkennen ist der neue Knotentyp Attribut-Wurzel-Knoten, unter dem die Attribute des jeweiligen XML-Elementes als Knoten gruppiert werden. Auch der Attributwert und der Text

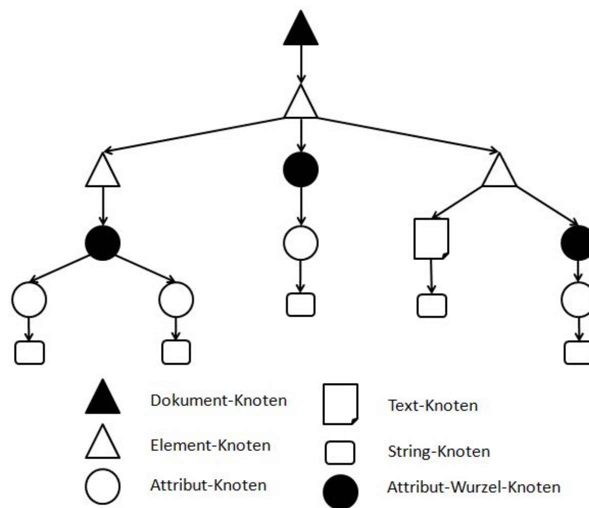

Abbildung 2.5: XML-Baum, aufgebaut nach der taDOM-Spezifikation

werden nach dieser Spezifikation als Knoten (*String-Knoten*) modelliert.

In Abbildung 2.5 schließlich, ist der XML-Baum, aufgebaut nach der in [HS07] vorgeschlagenen und in dieser Arbeit verwendeten Spezifikation, dargestellt. Hier sind die Attribute eben-

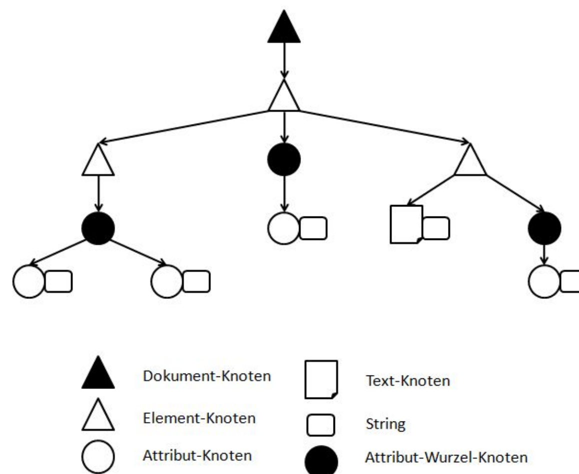

Abbildung 2.6: XML-Baum, aufgebaut nach der in [HS07] vorgeschlagenen Spezifikation

falls als Knoten modelliert, jedoch werden die Attributwerte und Texte als zu den Attribut- bzw. Textknoten gehörig angesehen.

2.7 Das DeweyID-Konzept

Das Konzept der DeweyIDs [OOP+04, Hau05, HS07] wurde eingeführt, um Knoten innerhalb eines XML-Baumes eindeutig zu kennzeichnen. Außerdem sollte es anhand der DeweyIDs verschiedener Knoten möglich sein, deren Beziehung zueinander zu ermitteln. Man sollte also erkennen können, ob z.B. ein Knoten der Vater, der Großvater, der Bruder oder ein Nachkomme eines anderen Knotens ist. Aus diesen Überlegungen ist folgendes Konzept entstanden:

- Die Knoten einer Ebene im Baum werden in ungerader Reihenfolge, mit der *1* beginnend, durchnummeriert. (*1, 3, 5, ...usw.*) Wobei die *1* immer für den Dokument-Knoten und den Attribut-Wurzel-Knoten reserviert ist.
- Die direkten Nachfolger eines Knotens übernehmen dessen ID und hängen ihre Nummer, durch einen Punkt getrennt, an die ID an. Hat z.B. ein Knoten als ID die *1*, so erhält das erste Kind dieses Knotens die ID *1.1*.

In Abbildung 2.7 ist die Vergabe der DeweyIDs nach dem eben beschriebenen Konzept für den XML-Baum aus Abbildung 2.6 dargestellt.

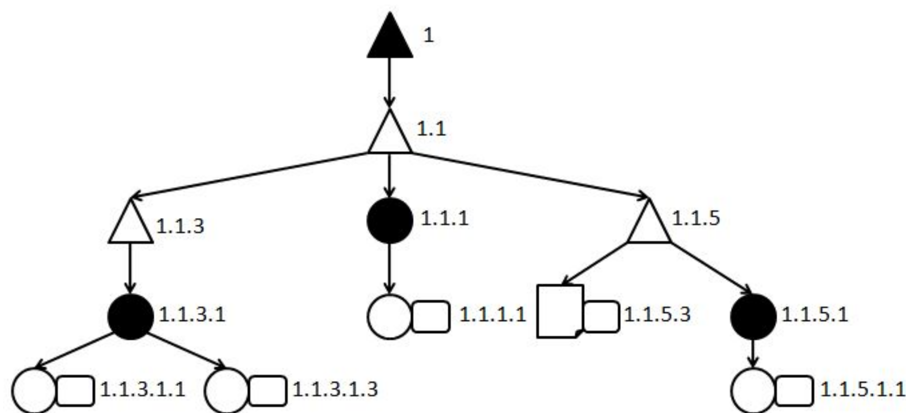

Abbildung 2.7: Vergabe der DeweyIDs für den XML-Baum aus Abbildung 2.6

Das DeweyID-Konzept bietet auch die Möglichkeit, einen bestehenden XML-Baum an beliebiger Stelle zu erweitern:

- Soll der Baum an der rechten Seite erweitert werden, wird einfach die ungerade Nummerierung für den neu eingefügten Knoten fortgeführt.
- Soll der Baum an der linken Seite erweitert werden, wird die ungerade Nummerierung im negativen Zahlenbereich fortgeführt.
- Soll zwischen zwei Knoten einer Ebene ein neuer Knoten eingefügt werden, so wird auf dieser Ebene zunächst ein virtueller Knoten eingefügt. Dieser erhält die gerade Zahl, welche zwischen den beiden ungeraden Zahlen der direkt benachbarten Knoten liegt. Der einzufügende Knoten wird nun an den virtuellen angehängt und wieder ungerade nummeriert. Hier sei angemerkt, dass der virtuelle Knoten nicht als Vorgänger des eingefügten Knotens gewertet wird.

Abbildung 2.8 veranschaulicht die Erweiterung des XML-Baumes aus Abbildung 2.7 an allen eben beschriebenen Stellen. Die gepunktet dargestellten Teile stellen die neu hinzugekommenen Knoten dar:

- Der Knoten mit der DeweyID *1.1.-1* stellt einer Erweiterung des Elementes mit der DeweyID *1.1* um ein Element an der linken Seite des Baumes dar.
- Der Knoten mit der DeweyID *1.1.7* stellt einer Erweiterung des Elementes mit der DeweyID *1.1* um ein Element an der rechten Seite des Baumes dar.
- Der Knoten mit der DeweyID *1.1.2.1* stellt einer Erweiterung des Elementes mit der DeweyID *1.1* um ein Element zwischen den beiden Knoten mit den DeweyIDs *1.1.1* und

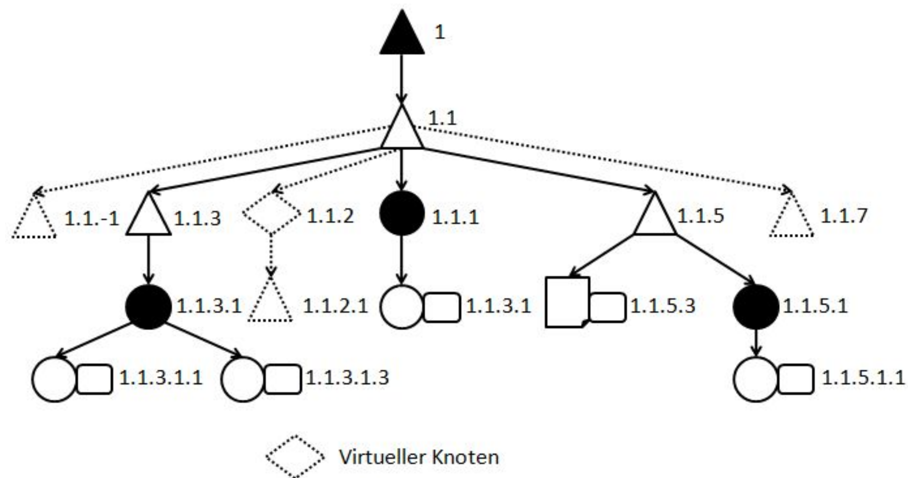

Abbildung 2.8: Erweiterung des XML-Baumes aus Abbildung 2.6

1.1.3 dar. Der Knoten mit der DeweyID *1.1.2* wird nicht ausgewertet. Er ist als virtuell anzusehen. Somit ist der direkte Vorgänger des Elements *1.1.2.1* das Element *1.1*.

Zusammenfassend soll Folgendes festgehalten werden. Besitzt man alle DeweyIDs der Knoten eines XML-Dokumentes, so ist es ohne den Zugriff auf dasselbige möglich, die Struktur dieses Dokumentes (XML-Baumes) zu ermitteln.

3 Bewertung existierender Transaktionsmodelle

In diesem Kapitel sollen einige ausgewählte erweiterte Transaktions- und Aktionskonzepte vorgestellt und anhand spezieller Kriterien bewertet werden. Im nächsten Abschnitt werden dazu zunächst die Bewertungskriterien formuliert. Diese ergeben sich aus dem speziellen Anwendungsgebiet, welches bereits in der Einleitung geschildert wurde. Anschließend werden die Transaktions- und Aktionskonzepte vorgestellt und entsprechend bewertet.

3.1 Die Bewertungskriterien

Gegenstand dieses Abschnitts ist die Formulierung der Anforderungen bzw. Bewertungskriterien, hinsichtlich derer eine Bewertung bereits existierender Transaktionsmodelle und des in dieser Arbeit vorgestellten Aktionsmodells stattfindet.

Die Unterstützung des Kooperationsprinzips

Die Unterstützung des Kooperationsprinzips (Abschnitt 2.3) geht als Kriterium am stärksten in die Bewertung ein, da die Umsetzung dieses Prinzips den Schwerpunkt der Arbeit darstellt. Genügt ein erweitertes Transaktions- oder Aktionskonzept nicht diesem Kriterium, so ist es auf keinen Fall als Lösungsmöglichkeit geeignet. Eine Untersuchung hinsichtlich der nachfolgenden Kriterien ist dann nicht mehr erforderlich.

Der Grad der Nebenläufigkeit

Um ein möglichst verzögerungsfreies Arbeiten mehrerer Nutzer auf einer zentralen Datenbasis zu ermöglichen, ist es erforderlich, deren Arbeitsschritte mit einem möglichst hohen Grad an Nebenläufigkeit auszuführen. Dabei hängt der erreichte Grad der Nebenläufigkeit sehr stark vom verwendeten Transaktionsmodell und dem damit verbundenen Sperrprotokoll ab. Entscheidend dabei ist der Zeitpunkt, an dem eine Transaktion die von ihr gehaltenen Sperren freigibt und die damit verbundenen Objektänderungen für andere Transaktionen sichtbar werden. Je früher eine Sperrfreigabe erfolgt, desto kürzer werden die Wartezeiten anderer Transaktionen bei der Sperranforderung und je höher wird der Grad der Nebenläufigkeit der Transaktionen.

Die Komplexität der Fehlerbehandlung

Die Komplexität des Fehlerbehandlungsmodells stellt einen entscheidenden Kritikpunkt für die Anwendbarkeit eines Transaktionsmodells dar. Je komplexer ein Fehlerbehandlungsmodell ist, je höher ist auch der Aufwand, der betrieben werden muss, um es zu realisieren. Weiterhin ist eine möglichst automatische Fehlerbehandlung, ohne größere Eingriffe des Anwendungsprogrammierers oder gar des Nutzers, erstrebenswert. Dies setzt ebenfalls eine möglichst geringe semantische Komplexität des Fehlerbehandlungsmodells voraus. Betrachtet man beispielsweise Kompensationstransaktionen, so kann deren Komplexität in einigen Fällen so hoch werden, dass ihre Realisierung nur durch Mitwirken des Nutzers möglich ist.

Die Anwenderunabhängigkeit

Im letzten Abschnitt wurde bereits die Anwenderunabhängigkeit des Fehlerbehandlungsmodells angesprochen. Generell sollte das gesamte Transaktionsmodell in jeder Situation ohne das Einwirken des Nutzers funktionieren.

3.2 Die geschachtelten Transaktionen

Die geschachtelten Transaktionen stellen eine Erweiterung des flachen Transaktionskonzepts dar, welches in Abschnitt 2.1 erläutert wurde. (Vgl. [SST97]) Dieses Konzept besagt, dass eine Transaktion wiederum aus Transaktionen, den so genannten *Subtransaktionen*, bestehen kann.

Geschachtelte Transaktionen können als Transaktionsbaum dargestellt werden. Abbildung 3.1 zeigt beispielhaft einen Transaktionsbaum. Die Transaktion *T1* stellt die *Wurzeltransaktion* dar.

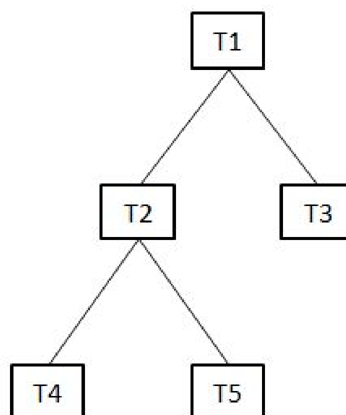

Abbildung 3.1: Ein Beispiel für einen Transaktionsbaum

Der direkte Vorgänger einer Transaktion wird als *Vatertransaktion* und der direkte Nachfolger als *Sohntransaktion* bezeichnet. Somit stehen *T1* und *T2*, *T1* und *T3*, *T2* und *T4* sowie *T2* und *T5* jeweils in einer Vater-Sohn-Beziehung. *T3* ist die *Brudertransaktion* von *T2* und umgekehrt. Gleiches gilt für *T4* und *T5*. Die Transaktionen *T2*, *T4* und *T5* bilden einen *Subtransaktionsbaum* mit der Wurzeltransaktion *T2*. Der dargestellte Transaktionsbaum hat eine Schachtelungstiefe von 3. Die Blätter (*T3*, *T4* und *T5*) sind flache Transaktionen.

Mit der Entwicklung des Konzeptes der geschachtelten Transaktionen wurden folgende zwei Hauptziele verfolgt:

- Lange Transaktionen sollen in kürzere zerlegt werden, um Objektänderungen früher sichtbar zu machen und somit die Nebenläufigkeit der Ausführung von Operationen zu erhöhen. Gerade im Hinblick auf Designarbeiten, wie in dem in Abschnitt 1.1.1 beschriebenen Anwendungsfall, ist dies sinnvoll. Dort dauert eine Transaktion vom Auschecken der XML-Dokumente bis zum Wiedereinchecken beim Server an.
- Die Fehlerbehandlung lässt sich flexibler gestalteten. Nach dem Prinzip der Atomarität müssen alle Änderungen einer Transaktion nach deren Scheitern rückgängig gemacht werden. Bei einer langen Transaktion (z.B. im Design) kann dadurch die Arbeit von mehreren Stunden verloren gehen. Durch die Zerlegung einer Transaktion in Subtransaktionen kann die Arbeit in kleine Abschnitte aufgeteilt werden, die einzeln abgebrochen werden können, ohne dass dies zwangsweise das Scheitern der gesamten Arbeit zur Folge hat.

3.2.1 Geschlossen geschachtelte Transaktionen

Die geschlossen geschachtelten Transaktionen sind eine Realisierungsform der geschachtelten Transaktionen. (Vgl. [Mos81, SST97, SHK05, Kru97]) Im Wesentlichen besagt dieses Prinzip, dass Sperren, die eine Subtransaktion hält, nach ihrer Freigabe an die Vatertransaktion zurückgehen. Diese entscheidet dann, ob sie die Sperren an eine weitere Sohntransaktion übergibt. Erst nach Beendigung der Wurzeltransaktion werden die Sperren für die Allgemeinheit freigegeben und somit die Ergebnisse für alle sichtbar. Es herrscht also Isolation zwischen verschiedenen Transaktionsbäumen und zwischen Geschwistern innerhalb eines Transaktionsbaumes. Lediglich zwischen einer Vatertransaktion und ihrer Sohntransaktion wird die Isolation und sogar die Serialisierbarkeit aufgehoben. Angenommen *T2* aus Abbildung 3.1 arbeitet auf einem Objekt *X*. Danach gibt sie, um eine Teilaufgabe lösen zu lassen, die Sperre auf dieses Objekt an *T4* weiter. Diese Transaktion arbeitet auf *X* und gibt die Sperre auf *X*, und damit auch das Ergebnis, nach Beendigung der Arbeiten an *T2* zurück. *T2* arbeitet danach weiter auf *X*. Der Schedule „$Write_2(X)\ Write_4(X)\ Write_2(X)$“ stellt ein Beispiel für diese Arbeitsabfolge dar. Er ist nicht serialisierbar, da er einen Zyklus enthält. Es lässt sich somit keine serielle Ausführungsreihenfolge der beiden Transaktionen finden.

Bezüglich des Verhaltens beim Abbruch einer Transaktion kann Folgendes festgehalten werden: Der Abbruch einer Sohntransaktion führt nicht zwingend zum Abbruch der Vatertransaktion. Wird jedoch eine Vatertransaktion abgebrochen, so müssen auch alle Sohntransaktionen abgebrochen werden. Dabei sei festgehalten, dass falls der Abbruch einer Sohntransaktion nicht zum Abbruch der Vatertransaktion führt, die Atomarität nicht gewährleistet wird, da, bezogen auf den kompletten Transaktionsbaum, nicht mehr gesagt werden kann, dass Alles oder Nichts ausgeführt wurde. Lediglich die Fehler-Atomarität ist gesichert, da abgebrochene Transaktionen keinen Effekt auf die Datenbank haben. Weiterhin soll an dieser Stelle noch einmal besonders auf die Dauerhaftigkeit hingewiesen werden. Diese gilt bei geschlossen geschachtelten Transaktionen nur für die Wurzeltransaktion. Änderungen einer Subtransaktion werden also erst mit dem erfolgreichen Abschluss der Wurzeltransaktion an die Datenbank übertragen. Dadurch ist kein zusätzlicher Aufwand für das Rücksetzen von Subtransaktionen notwendig, wie das bei den im nächsten Abschnitt beschriebenen offen geschachtelten Transaktionen der Fall ist.

Durch die geschlossen geschachtelten Transaktionen wird das Auftragsprinzip in seiner reinsten Form realisiert. Eine Vatertransaktion, in der Rolle des Auftraggebers, vergibt eine zu erfüllende Teilaufgabe und die erforderlichen Informationen an eine Sohntransaktion, die sich in der Rolle des Auftragnehmers befindet. Diese ermittelt ein Ergebnis und gibt es an die Vatertransaktion zurück. Scheitert sie jedoch, so kann die Vatertransaktion eine weitere Sohntransaktion starten und ihr wiederum die Teilaufgabe übergeben. Die Auftragnehmer bleiben der Allgemeinheit verborgen. Ihre Ergebnisse werden erst mit dem *Commit* der Wurzeltransaktion für alle sichtbar gemacht.

Das Kooperationsprinzip wird durch das Modell der geschlossen geschachtelten Transaktionen nicht realisiert. Ordnet man jedem Nutzer einen vollständigen Transaktionsbaum zu, so kann ein Nutzer die Ergebnisse eines anderen erst sehen, wenn dieser seine Arbeiten beendet hat. Es findet somit nur ein Informationsfluss in einer Richtung statt. Ein Verhandeln der Nutzer über eine finale Version eines Objektes ist nicht möglich. Aus diesem Grund kann schon an dieser Stelle gesagt werden, dass die geschlossen geschachtelten Transaktionen für den speziellen Anwendungsfall aus Abschnitt 1.1.1 nicht geeignet sind. Sie werden daher auch nicht weiter betrachtet.

3.2.2 Offen geschachtelte Transaktionen

Die offen geschachtelten Transaktionen sind eine weitere Realisierungsform der geschachtelten Transaktionen.(Vgl. [SST97, SHK05, Kru97]) In diesem Modell wird die Isolation zwischen Transaktionsbäumen aufgehoben. Ergebnisse einer Subtransaktion werden nach Ihrem Übergang in den Zustand *Committed* für alle Transaktionen freigegeben.

Durch dieses Modell wird das Kooperationsprinzip vollständig realisiert. Ordnet man jedem Nutzer einen vollständigen Transaktionsbaum zu, so kann ein Nutzer die Ergebnisse eines anderen schon sehen, bevor dieser seine Arbeiten beendet hat. Dadurch ist es ihm möglich, die Ergebnisse zu bewerten und gegebenenfalls Änderungen vorzunehmen, die dann wieder an den anderen Nutzer übergeben werden können. Es findet also eine Kommunikation in beliebiger Richtung zwischen den Nutzern statt. Sie können gemeinsam über eine finale Version eines Objektes verhandeln.

Weiterhin wird durch das Konzept der offenen Schachtelung ein hoher Grad der Nebenläufigkeit erreicht. Die Sperren werden am Ende jeder Subtransaktion für alle Transaktionen freigegeben. Dadurch stehen Objekte schnell wieder zur allgemeinen Verfügung. Je höher der Grad der Schachtelung ist und je feiner eine Folge von Operationen in Subtransaktionen zerlegt wird, desto kürzer werden die Wartezeiten auf Objektfreigaben.

Wie bei den geschlossen geschachtelten Transaktionen führt der Abbruch einer Sohntransaktion nicht zwingend zum Abbruch einer Vatertransaktion. Wird jedoch eine Vatertransaktion abgebrochen, so führt dies immer zum Abbruch aller Kindtransaktionen. Da bei den offen geschachtelten Transaktionen die Dauerhaftigkeit auch für die Subtransaktionen gefordert wird, führt dies, im Gegensatz zu den geschlossen geschachtelten, zu nachfolgend erläutertem Problem: Kommt es zum Abbruch einer Vatertransaktion, müssen alle Sohntransaktionen, und zwar auch die, die sich im Zustand *Committed* befinden, rückgesetzt werden. Dies wiederspricht aber nun der Forderung nach Dauerhaftigkeit. Aus diesem Grund werden Kompensationstransaktionen (Siehe Abschnitt 2.2.) genutzt, die die Änderungen einer rückzusetzenden Transaktion seman-

tisch rückgängig machen und damit auch Fehler-Atomarität gewährleisten. Allerdings müssen diese Kompensationstransaktionen für jede mögliche Operation (und den Konkatenationen von Operationen) vom Anwendungsentwickler definiert werden. Dies kann unter Umständen sehr komplex werden, vor allem im Falle von kaskadierenden Abbrüchen. Hat nämlich eine Transaktion B von einer anderen sich bereits im Zustand *Committed* befindenden Transaktion A gelesen, und A wird aufgrund des Abbruchs der Vatertransaktion abgebrochen, so muss auch B abgebrochen werden, da ein nun ungültiger Objektzustand gelesen wurde. Manchmal ist das Finden einer korrekten Kompensationstransaktion auch nicht möglich. Quadriert z.B. eine Transaktion einen Wert, so müsste die Kompensationstransaktion einfach die Quadratwurzel dieses Wertes ziehen. Allerdings hat die Quadratwurzel als Ergebnis zwei Werte, nämlich den positiven und den negativen. Liegt vom Datenbanksystem keine Einschränkung auf entweder den positiven oder negativen Wertebereich vor, so müsste in diesem Fall der Nutzer bestimmen, welcher Werte in Frage kommt. (Auf Probleme mit kompensierenden Transaktionen wird u.a. auch in [GFJK03] hingewiesen.) Somit lässt sich also zusammenfassend feststellen, dass eine automatisierte Fehlerbehandlung nicht in jedem Fall möglich ist.

In den nachfolgenden zwei Abschnitten wird auf zwei konkrete Realisierungen des offen geschachtelten Transaktionskonzepts eingegangen.

Sagas

Sagas stellen eine Möglichkeit dar, der problematischen Fehlerbehandlung der offen geschachtelten Transaktionen zu begegnen. (Vgl. [GMS87, Moc95, Kru97]) Um dies zu verstehen, soll zunächst der Aufbau der Sagas nachfolgend dargestellt werden:

- Ein Saga besteht aus einer Wurzeltransaktion und einer Schicht Subtransaktionen, die ihrerseits flache Transaktionen sind. Die Änderungen einer Subtransaktion werden nach ihrem Übergang in den Zustand *Committed* für alle Transaktionen sichtbar.
- Operationen dürfen nur innerhalb der Subtransaktionen, nicht aber innerhalb der Wurzeltransaktion, ausgeführt werden.
- Alle Subtransaktionen einer Saga werden streng nacheinander ausgeführt.
- Subtransaktionen unterschiedlicher Sagas müssen untereinander kommutieren. Dies bedeutet Folgendes: Greifen zwei Transaktionen auf dasselbe Objekt zu, so ist jeder der beiden Transaktionen gleichgültig, ob vor ihr schon die andere Transaktion auf das Objekt zugegriffen hat. Der Erfolg einer Transaktion hängt also nicht vom Zustand eines Objektes ab. (Subtransaktionen derselben Saga müssen selbstverständlich nicht kommutieren, da ihre Ausführungsreihenfolge festgelegt ist.)
- Für jede Subtransaktion muss eine entsprechende Kompensationstransaktion existieren. Diese wird benötigt, wenn die persistent gespeicherten Änderungen einer sich im Zustand *Committed* befindenden Subtransaktion rückgesetzt werden müssen. Dies tritt, wie bereits erwähnt, ein, wenn die Wurzeltransaktion abbricht. Dann müssen alle Subtransaktionen abgebrochen werden, und zwar auch die, die sich bereits im Zustand *Committed* befinden.

Durch die Forderung nach Kommutativität der Subtransaktionen verschiedener Sagas findet keine Schadensausbreitung über die Grenzen einer Saga hinaus statt. Das heißt, der Abbruch einer Subtransaktion innerhalb einer Saga führt nie zum Abbruch einer Subtransaktion innerhalb einer anderen Saga. Dadurch sinkt die Komplexität der Fehlerbehandlung, allerdings auf Kosten

der Kooperativität. Die Forderung nach Kommutativität unterbindet jegliche „echte“ Form der Kommunikation zwischen verschiedenen Sagas. Wenn einer Transaktion gleichgültig ist, welchen Zustand ein Objekt hat, so findet auch kein Informationsgewinn statt, auf dessen Basis eine Entscheidung gefällt wird. Anders ausgedrückt, bedeutet das, dass jeder Nutzer „blind“ auf die Objekte zugreift, ohne deren Eigenschaften oder Zustände zu berücksichtigen. Für den Einsatz in einer kooperativen Designumgebung sind die Sagas daher völlig ungeeignet und werden nicht weiter betrachtet.

Das ConTract-Modell

Das ConTract-Modell stellt eine weitere Realisierungsform der offen geschachtelten Transaktionen dar. (Vgl. [WR92, Kru97]) Ein ConTract ist wie folgt aufgebaut:

- Es besteht aus einer Menge von *Schritten*. Diese Schritte entsprechen flachen Transaktionen. Die Änderungen eines Schrittes werden nach dessen Übergang in den Zustand *Committed* für alle Schritte sichtbar.
- Es enthält ein *Skript*, welches die Ausführungsreihenfolge der einzelnen Schritte festlegt. Dabei sind z.B. auch Schleifen und Verzweigungen möglich.
- Für jeden Schritt muss eine entsprechende Kompensationstransaktion definiert werden. (Hinweis: Der Abbruch eines ConTracts hat den Abbruch aller Schritte, die er beinhaltet, zur Folge.)
- Anders als bei den Sagas wird keine Kommutativität von Schritten unterschiedlicher ConTracts gefordert. Hier werden zur Synchronisierung der Zugriffe auf Objekte Vor- und Nachbedingungen für jeden Schritt festgelegt. Beispielsweise kann für ein Attribut ein Wertebereich festgelegt werden. Der Schritt muss nun zu Beginn seiner Arbeit prüfen, ob sich der Wert des Attributs in dem definierten Wertebereich befindet und muss danach sicherstellen, dass durch seine Arbeiten keine Verletzung dieser Festlegung eingetreten ist. Konnte eine Vor- oder Nachbedingung nicht erfüllt werden, so darf der Schritt nicht ausgeführt oder muss abgebrochen werden.

Das ConTract-Modell erfüllt das Kooperationsprinzip im vollen Umfang. Zwischen den Schritten verschiedener ConTracts wird weder Kommutativität noch Serialisierbarkeit verlangt. Lediglich die Einhaltung festgelegter Bedingungen ist erforderlich. Ordnet man jedem Nutzer einen ConTract zu, so kann gesagt werden, dass zwischen den Nutzern ein Informationsaustausch in beliebiger Richtung möglich ist.

Durch das Modell wird ein hoher Grad der Nebenläufigkeit gewährleistet. Dies ist einerseits darauf zurückzuführen, dass innerhalb eines ConTracts die nebenläufige Ausführung von Schritten festgelegt werden kann. Andererseits werden durch das Prinzip der offenen Schachtelung Objektänderungen eines Schrittes sehr früh für alle anderen Schritte sichtbar.

Die Fehlerbehandlung im ConTract-Modell ist als sehr komplex zu bewerten. Dies liegt wiederum daran, dass für jede Transaktion (jeden Schritt) eine entsprechende Kompensationstransaktion definiert werden muss. Dies bedeutet einerseits einen hohen Aufwand für den Anwendungsprogrammierer und schließt andererseits ein Eingreifen des Nutzers nicht in jedem Fall aus. Außerdem ist es nicht in jedem Fall möglich, eine geeignete Kompensationstransaktion zu finden.(Vgl. [GFJK03]) Wie bei den klassischen offen geschachtelten Modellen kann es auch im ConTract-Modell zu kaskadierenden Abbrüchen kommen, die ebenfalls mit geeigneten Kom-

pensationstransaktionen abgefangen werden müssen. Wird nämlich ein ConTract abgebrochen, so müssen alle seine Schritte kompensiert werden. Dadurch können aber die Vorbedingungen der Schritte anderer ConTracts verletzt werden, die dann ebenfalls kompensiert werden müssen.

3.2.3 Die Vereinigung des Auftrags- und Kooperationsprinzips

In diesem Abschnitt werden zwei konkrete Modelle vorgestellt, die versuchen, die Eigenschaften der offen und geschlossen geschachtelten Transaktionen miteinander zu verbinden. Anders ausgedrückt, wurde mit der Entwicklung beider Modelle beabsichtigt, dass Auftrags- und Kooperationsprinzip zu vereinigen. (Vgl. [Kru97])

Das DOM-Transaktionsmodell

Das DOM (*Distributed Object Management*)-Transaktionsmodell stellt einen Ansatz dar, die Eigenschaften der offen und geschlossen geschachtelten Transaktionen in einem Transaktionskonzept zu integrieren. (Vgl. [BOH+92, Kru97]) Es ist wie folgt aufgebaut:

- Es existieren zwei Transaktionstypen in diesem Modell:
  - Die *Top-Transaktion* entspricht entweder einer flachen Transaktion oder einer geschlossen geschachtelten Transaktion. Änderungen von erfolgreich beendeten Subtransaktionen einer Top-Transaktion werden erst mit dem Übergang der Top-Transaktion in den Zustand *Committed* für alle Transaktionen sichtbar. Operationen dürfen nur innerhalb von Top-Transaktionen ausgeführt werden.
  - Die *Multi-Transaktionen* stellen offen geschachtelte Transaktionen dar, die entweder andere Multi- oder Top-Transaktionen beinhalten. Für sie gelten also die Eigenschaften der offen geschachtelten Transaktionen. Allerdings darf innerhalb einer Multi-Transaktion keine Operation ausgeführt werden. Diese müssen, wie bereits erwähnt, in Top-Transaktionen gehüllt werden.
- Für die Top-Transaktionen müssen entsprechende Kompensationstransaktionen bereitgestellt werden. (Für die Multi-Transaktionen ist das nicht erforderlich, da sie selbst keine Operationen ausführen dürfen.)

Das DOM-Transaktionsmodell realisiert sowohl das Auftrags- als auch das Kooperationsprinzip. Innerhalb von Top-Transaktionen können Auftraggeber- und Auftragnehmerbeziehungen aufgebaut werden, wobei die Auftragnehmer, wie gefordert, der Allgemeinheit verborgen bleiben. Das Kooperationsprinzip wird durch die Multi-Transaktionen realisiert. Ordnet man jedem Nutzer eine Multi-Transaktion zu, so kann gesagt werden, das zwischen den Nutzern ein Informationsaustausch in beliebiger Richtung erfolgen kann.

Auch der Grad der Nebenläufigkeit kann als hoch eingestuft werden. Dies ist auf die offene Schachtelung der Multi-Transaktionen zurückzuführen.

Bei der Fehlerbehandlung treten jedoch wieder die bereits bekannten Probleme der offen geschachtelten Transaktionen auf. Für alle Top-Transaktionen müssen entsprechende Kompensationstransaktionen bereitgestellt werden. Dies kann hier besonders kompliziert werden, da eine Top-Transaktion aus vielen Subtransaktionen besteht, deren Änderungen implizit rückgängig gemacht werden müssen. (Implizit, da die Dauerhaftigkeit nur für die Top-Transaktion erfüllt

sein muss, siehe auch Abschnitt 3.2.1.) Außerdem kann es auch in diesem Modell zu kaskadierenden Abbrüchen kommen, wenn eine Top-Transaktion einer Multi-Transaktion die durch einen Abbruch ungültig gewordenen Änderungen einer Top-Transaktion einer anderen Multi-Transaktion gelesen hat.

Das CONCORD-Modell

Das CONCORD (*Controlling Cooperation in Design Environments*)-Modell stellt einen weiteren Ansatz zur Integration des Auftrags- und Kooperationsprinzips dar. Es wurde explizit für Designanwendungen, wie z.B. CAD, entwickelt. (Vgl. [RMH+94, Kru97]) Das CONCORD-Modell ist wie folgt aufgebaut:

- Es existieren so genannte *Designaktivitäten*, die die Arbeitsabläufe eines Designers modellieren. Diese beinhalten eine Menge von *Designoperationen*, ein *Skript*, welches deren Ausführungsreihenfolge festlegt, eine *Designspezifikation* und einen *Objektpool*.
- Die Designoperationen stellen einzelne Arbeitsschritte dar und können als flache Transaktionen angesehen werden.
- Der Objektpool enthält alle von den Designoperationen benutzten und erzeugten Objektversionen. Beim Start einer Designaktivität werden die zu bearbeitenden Objekte als lokale Kopien in dem Objektpool angelegt. Jede Änderungsoperation, in Form einer Designoperation, führt zum Anlegen einer neuen Objektversion. Nach Abschluss der Arbeiten (Beendigung einer Designaktivität) werden die lokalen Objektversionen in die Datenbank eingelagert.
- Die Fehlerbehandlung erfolgt über den Objektpool, der ein Objektversionensystem darstellt. Muss eine Designoperation rückgesetzt werden, so wird einfach nur die von ihr erzeugte Objektversion gelöscht. Ist das Rücksetzen einer ganzen Designaktivität erforderlich, so braucht lediglich der entsprechende Objektpool entfernt werden.
- Die Designspezifikation beschreibt das Ziel einer Designaktivität. Ist es erreicht, so kann die Designaktivität beendet werden.
- Prinzipiell läuft jede Designaktivität isoliert von allen anderen ab. Nur über definierte Beziehungen findet eine Kommunikation zwischen den Designoperationen verschiedener Designaktivitäten statt. Nachfolgend werden diese drei verschiedenen Beziehungstypen spezifiziert:
  - Die *Delegation*-Beziehung stellt eine Auftraggeber-Auftragnehmer-Beziehung dar. Dabei erstellt der Auftraggeber für den Auftragnehmer eine Designspezifikation und übergibt ihm die zur Erfüllung der Aufgabe benötigten Informationen und Objekte. Anschließend bekommt er vom Auftragnehmer das Ergebnis, in Form von Objektversionen, zurück. Diese Objektversionen muss er dann mit seinem eigenen Objektpool verschmelzen.
  - Die *Usage*-Beziehung dient der Kooperation zwischen Designaktivitäten. Dabei findet eine Übertragung von Objektversionen, in Form von Kopien, von einer Designaktivität zu einer anderen statt. Hierbei kann auch eine Bewertung einer erhaltenen Objektversion durch eine Designaktivität stattfinden, indem ihr Zustand mit der Designspezifikation verglichen wird. Soll ein bidirektionaler Objektversionenaus-

tausch stattfinden, so müssen entsprechend zwei *Usage*-Beziehungen zwischen den Kommunikationspartnern aufgebaut werden.

– Die *Negotiation*-Beziehung stellt eine andere Form der Kooperation dar. Hierbei können Auftragnehmer, die zu ein- und demselben Auftraggeber gehören müssen, ihre Designspezifikationen untereinander verhandeln und dadurch Aufgaben neu verteilen.

Das CONCORD-Modell realisiert vollständig das Auftrags- und Kooperationsprinzip. Durch die *Delegation*-Beziehung wird dabei das Auftragsprinzip und durch die *Usage*-Beziehung das Kooperationsprinzip umgesetzt.

Der Grad der Nebenläufigkeit kann als sehr hoch eingestuft werden. Lediglich beim Start von Designaktivitäten, bei dem die benötigten Objekte in den lokalen Objektpool jeder Einheit kopiert werden, sind Sperren auf diese Objekte in der Datenbank erforderlich, um den konkurrierenden Zugriff zu synchronisieren. Danach arbeiten die Designoperationen innerhalb einer Designaktivität auf ihren lokalen Objektpools. Dadurch ist das gleichzeitige Arbeiten auf einem Objekt möglich. Diese Isolation der einzelnen Designaktivitäten wird nur durch das bewusste Eingehen einer *Delegation*- oder *Usage*-Beziehung aufgebrochen.

Der hohe Grad der Nebenläufigkeit geht jedoch auf Kosten der Anwenderunabhängigkeit. Dies lässt sich wie folgt begründen: Durch die isolierte Arbeit auf unterschiedlichen Kopien eines Objektes ergeben sich entsprechend unterschiedliche finale Versionen dieses Objektes. Beim Propagieren dieser Objektversionen zur Datenbank muss daher ein Verschmelzen dieser unterschiedlichen Versionen zu einer endgültigen stattfinden. Dieser Vorgang ist ohne Intervention der Designer kaum realisierbar. Gleiches gilt für die *Usage*- und die *Delegation*-Beziehung. Auch hier müssen verschiedene Objektversionen zu einer finalen zusammengeführt werden.

Die Fehlerbehandlung gestaltet sich im CONCORD-Modell durch die Verwendung eines Objektversionensystems relativ unkompliziert. Das Rücksetzen einzelner Designoperationen oder ganzer Designaktivitäten entspricht dem Entfernen der entsprechenden Objektversionen aus dem Objektpool oder dem kompletten Verwerfen desselbigen. Auch die Zahl der kaskadierenden Abbrüche wird durch die wohl definierten Beziehungen erheblich eingeschränkt. Der Abbruch einer Designoperation infolge des Abbruchs einer anderen ist höchstens in den Fällen nötig, in denen eine Beziehung (z.B. *Usage*) zwischen den beiden Designoperationen besteht. Problematisch ist allerdings eine Situation, in der eine Designoperation infolge eines kaskadierenden Abbruchs rückgesetzt werden soll, deren Änderungen bereits in die Datenbank übertragen wurden. Für diesen Fall ist im CONCORD-Modell keine Möglichkeit vorgesehen, um die ungültigen Einträge aus der Datenbank zu entfernen. Die Datenbank wird also gegebenenfalls in einem inkonsistenten Zustand hinterlassen.

3.3 Dynamische Aktionen

Die dynamischen Aktionen stellen eine Erweiterung der klassischen flachen Transaktionen dar. (Vgl. [NS92, Moc95]) Sie unterscheiden sich von ihnen dadurch, dass sie einen zusätzlichen Zustand *Completed* besitzen.

In Abbildung 3.2 ist das vollständige Zustandsmodell einer dynamischen Aktion dargestellt. Auffällig an diesem Modell sind folgende zwei Aspekte:

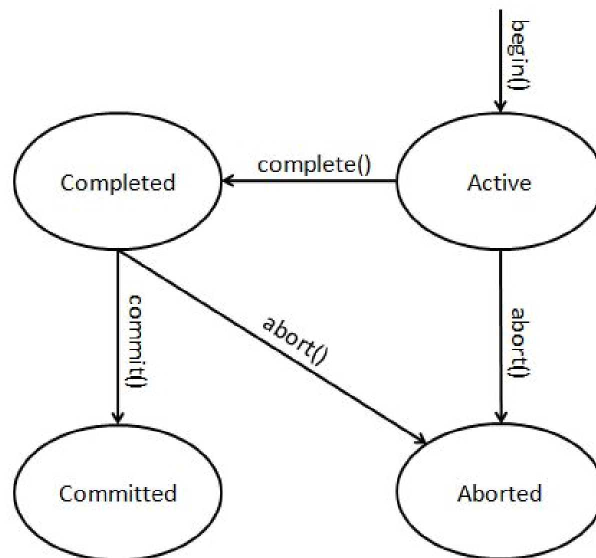

Abbildung 3.2: Zustände einer dynamischen Aktion

- Bevor eine dynamische Aktion in den Zustand *Committed* übergehen kann, muss sie sich im Zustand *Completed* befinden. Dieser Zustand besagt, dass eine dynamische Aktion all ihre Operationen ausgeführt hat und bereit ist, in den Zustand *Committed* überzugehen. Dies bedeutet auch, dass sie aus eigener Kraft nicht mehr abbrechen will und nur noch durch einen kaskadierenden Abbruch oder einen Systemausfall abgebrochen werden kann.
- Befindet sich eine dynamische Aktion im Zustand *Committed*, so kann sie auf keinen Fall mehr abgebrochen werden. Dadurch ist die Dauerhaftigkeit in jedem Fall gewährleistet.

Nachfolgend sollen diese Aspekte näher erläutert werden.

Das Konzept der dynamischen Aktionen fordert nicht die Durchsetzung der Isolation. Dies bedeutet, Objekte können schon vor dem Übergang einer Aktion in den Zustand *Committed* freigegeben werden. Dies hat allerdings zur Folge, dass sich andere Aktionen, die auf diese Objektfreigaben zugreifen, von dieser Aktion abhängig machen. Wird also eine Aktion abgebrochen, so müssen auch alle Aktionen, die von dieser Aktion gelesen haben, mit abgebrochen werden. Dies ist als kaskadierender Abbruch bekannt. Besäßen dynamische Aktionen nicht den Zustand *Completed*, so könnte es vorkommen, dass infolge eines kaskadierenden Abbruchs eine sich bereits im Zustand *Committed* befindende Aktion rückgängig gemacht werden müsste, was der Dauerhaftigkeit wiederspräche. Die Einführung des zusätzlichen Zustands *Completed* erlaubt es jedoch, eine Aktion als fertig ausgeführt zu markieren, sie aber erst dann persistent zu machen, wenn sie garantiert nicht mehr abgebrochen werden kann. Somit verharrt eine Aktion solange im Zustand *Completed*, bis alle Aktionen, von denen sie abhängig ist, in den Zustand *Committed* übergegangen sind. Dann kann ausgeschlossen werden, dass diese Aktion noch durch einen kaskadierenden Abbruch rückgesetzt werden muss. Diese Eigenschaft wurde in [Moc95, NM95] als *Commit-Korrektheit* eingeführt und bewiesen.

Wichtig ist allerdings, dass garantiert werden kann, dass überhaupt ein *Commit* von dynamischen Aktionen stattfinden kann. Dies ist notwendig, um einen erlangten Systemfortschritt, also eine Menge von beendeten Operationen, persistent zu sichern. Dies wird als Fortschreibbarkeit der *Commit-Linie* bezeichnet. (Vgl. [Moc95, NM95]) Die *Commit-Linie* umfasst daher immer

die aktuellsten, durch *Commit*-Ereignisse persistent gespeicherten Objektänderungen. Probleme dabei können auftreten, wenn zyklische Abhängigkeiten zwischen dynamischen Aktionen bestehen. Dies kann auftreten, wenn das Kriterium der Serialisierbarkeit nicht erfüllt wird. Der Schedule 3.1 veranschaulicht diesen Sachverhalt.

$$Schedule = Write_1(X)Read_2(X)Write_2(X)Read_1(X)Complete_1Complete_2 \tag{3.1}$$

Dieser Schedule ist nicht serialisierbar, da er einen Zyklus enthält. Weiterhin kommt es auch zu einer zyklischen Abhängigkeit der einzelnen zugehörigen Aktionen. Aktion *2* liest und überschreibt einen von Aktion *1* erzeugten Objektzustand. Danach liest Aktion *1* den von Aktion *2* erzeugten Objektzustand. Ein Abbruch von Aktion *1* würde zu einem Abbruch der Aktion *2* führen und umgekehrt. Beide Aktionen befinden sich nun im Zustand *Completed* und wollen in den Zustand *Committed* übergehen. Nun wartet die Aktion *1* auf das *Commit* der Aktion *2*. Gleichzeitig wartet aber auch die Aktion *1* auf das *Commit* der Aktion *2*, da Aktion *1* nach der Schreiboperation von Aktion *2* nochmals lesend auf *X* zugegriffen hat. Somit kann keine der beiden Aktionen in den Zustand *Committed* übergehen. Die einzige Lösung für dieses Problem ist, dass beide Aktionen gleichzeitig in den Zustand *Committed* übergehen. Dies wird auch als *Gruppen-Commit* bezeichnet und wurde in [Moc95] eingeführt und bewiesen.

Der entscheidende Vorteil der dynamischen Operationen, um ihn noch einmal besonders hervorzuheben, liegt darin, dass die Dauerhaftigkeit garantiert werden kann. Dies vereinfacht die Fehlerbehandlung deutlich, da auf das Konzept der Kompensationstransaktionen verzichtet werden kann.

Die dynamischen Aktionen allein genügen jedoch nicht den gestellten Anforderungen an ein Transaktionskonzept, da unter anderem keine Behandlung der Problematik langer Transaktionen stattfindet. Es wird also ein aufbauendes Modell benötigt, welches im folgenden Abschnitt vorgestellt wird.

3.4 Geschachtelte dynamische Aktionen

Die geschachtelten dynamischen Aktionen besitzen eine Baumstruktur. Diese gleicht der Struktur der in Abschnitt 3.2.1 beschriebenen geschlossen geschachtelten Transaktionen. (Vgl. [NW94, Kru97]) Allerdings fordern die geschachtelten dynamischen Aktionen nicht die vollständige Isolation von (Teil-)Aktionsbäumen. Hier wird lediglich das Kriterium der Serialisierbarkeit für Aktionsbäume gefordert. Im Detail bedeutet das:

- Vollständige Aktionsbäume müssen serialisierbar sein.
- Teilbäume eines Transaktionsbaumes müssen serialisierbar sein.
- Zwischen Vätern und Söhnen wird jedoch keine Serialisierbarkeit gefordert.
- Wie auch bei den geschachtelten Transaktionen gilt, ein Abbruch der Vateraktion führt zum Abbruch aller Sohnaktionen. Ein Abbruch einer Sohnaktion führt jedoch nicht zwingend zum Abbruch der Vateraktion.
- Für die dynamischen Aktionen wird die ACID-Eigenschaft *Dauerhaftigkeit* garantiert. Bei geschachtelten dynamischen Aktionen tritt dabei jedoch ein Problem auf. Eine Sohnaktion kann noch solange abgebrochen werden, wie die Vateraktion noch nicht im Zustand *Committed* ist. Daher verweilt sie im Zustand *Completed*. Andererseits kann auch

die Vateraktion kein *Commit* ausführen, ohne dass sichergestellt ist, dass alle Söhne, die nicht abgebrochen wurden, garantiert in den Zustand *Committed* übergehen. Daher muss auch sie im Zustand *Completed* verweilen. Um dennoch in den Zustand *Committed* zu gelangen, gehen die Vateraktion und die Sohnaktionen gleichzeitig in diesen Zustand über, wenn sichgestellt ist, dass keine Aktion mehr rückgesetzt werden muss. Dies wird, wie bereits im letzten Abschnitt erwähnt, als *Gruppen-Commit* bezeichnet.

Durch die geschachtelten dynamischen Aktionen wird das Auftragsprinzip verwirklicht, da sich zwischen Vätern und Söhnen eine Auftraggeber-Auftragnehmer-Beziehung spezifizieren lässt. Das Kooperationsprinzip wird hingegen nicht realisiert. Ordnet man jedem Nutzer einen vollständigen Aktionsbaum zu, so wird eine serialisierbare Ausführung dieser Aktionsbäume erzwungen. Dadurch wird ein „verhandelndes“ Arbeiten an einem Objekt unterbunden. Aus diesem Grund eignen sich die geschachtelten dynamischen Aktionen nicht für den speziellen Anwendungsfall aus Abschnitt 1.1.1 und werden daher auch nicht näher betrachtet.

3.5 Geschachtelte dynamische Aktionen für kooperative Anwendungen

Die geschachtelten dynamischen Aktionen für kooperative Anwendungen stellen einen Ansatz dar, das Auftrags- und Kooperationsprinzip zu vereinigen. (Vgl. [Kru97]) Sie lassen sich wie folgt beschreiben:

- Zur Verwirklichung des Auftragsprinzips werden Aktionsbäume, die im vorhergehenden Abschnitt beschrieben wurden, verwendet.
- Zur Realisierung des Kooperationsprinzips wird das in [Moc95] eingeführte Prinzip der Kooperationsgruppen von dynamischen Aktionen verwendet. Nutzer, die miteinander kooperieren wollen, treten einer Kooperationsgruppe bei. Eine Kooperationsgruppe lässt sich wie folgt charakterisieren:
  - Eine Kooperationsgruppe besteht aus mindestens einer Aktion. Hier wird weiterhin gefordert, dass Aktionsbäume immer vollständig zu einer Kooperationsgruppe gehören müssen. Ein kooperationsgruppenübergreifender Aktionsbaum ist nicht gestattet.
  - Zwischen Kooperationsgruppen wird Serialisierbarkeit gefordert. Innerhalb der Kooperationsgruppen wird jedoch keine Serialisierbarkeit gefordert. Dadurch kann es zu zyklischen Abhängigkeiten zwischen Aktionen kommen, wie das bereits in Abschnitt 3.3 geschildert wurde. In diesem Fall wird wieder das *Gruppen-Commit* eingesetzt, um ein Fortschreiben der *Commit-Linie* zu gewährleisten.
- Das Auftragsprinzip fordert das Verbergen der Auftragnehmer vor jeder anderen Aktion außer der Vateraktion. Das Kooperationsprinzip hingegen verlangt nach frühen Objektfreigaben, wodurch Auftragnehmer automatisch sichtbar werden. Um diesem Problem zu begegnen, wird das Prinzip der *Objektkontrolle* eingeführt, wodurch sich die Sichtbarkeit von Objekten durch die Applikation steuern lässt. Es kann also z.B. festgelegt werden, ob ein Objekt nur für den Auftragnehmer oder für alle Aktionen innerhalb einer Kooperationsgruppe sichtbar gemacht werden soll. Des Weiteren kann durch dieses Prinzip auch

festgelegt werden, wann Objekte, die gerade innerhalb einer Kooperationsgruppe bearbeitet werden, anderen Kooperationsgruppen zugänglich gemacht werden.

- Durch die fehlende Serialisierbarkeit innerhalb einer Kooperationsgruppe kann es zu Nebenläufigkeitsanomalien kommen, wie bereits in Abschnitt 2.2 erwähnt. Es können also Inkonsistenzen auftreten. Mit Hilfe der Objektkontrolle kann jedoch der kooperative Zugriff für bestimmte Objekte, bei denen dieser garantiert zu Inkonsistenzen führen würde, gezielt verboten werden.

Durch das hier beschriebene Modell wird sowohl das Kooperations- als auch das Auftragsprinzip realisiert. Durch die Verwendung der Objektkontrolle kann sogar bestimmt werden, wie stark jedes der beiden Prinzipien verwirklicht werden soll, indem die Sichtbarkeit von Objekten bzw. Objektänderungen festgelegt wird.

Auch der Grad der Nebenläufigkeit ist als hoch einzustufen. Dies lässt sich dadurch begründen, dass keine Isolation und innerhalb von Kooperationsgruppen nicht einmal Serialisierbarkeit gefordert wird.

Ein besonderer Vorteil dynamischer Aktion ist, dass stets die Dauerhaftigkeit gesichert ist. Daher ist es nicht erforderlich, Kompensationsaktionen zu spezifizieren. Es besteht dadurch die Möglichkeit, das zu verwendende Fehlerbehandlungsmodell frei zu wählen. Da aus diesem Grund kein Fehlerbehandlungsmodell in [Kru97] vorgeschlagen wurde, kann auch über dessen Komplexität keine Aussage getroffen werden.

Die Anwenderunabhängigkeit kann in diesem Modell nicht garantiert werden. Dies lässt sich dadurch begründen, dass Inkonsistenzen, die während der kooperativen Arbeit entstehen können, nur vermieden werden können, indem der Anwender über die Objektkontrolle spezifiziert, für welche Objekte Kooperation bedenkenlos möglich ist und für welche nicht.

Ein weiterer Kritikpunkt ist das Prinzip der Kooperationsgruppen. Nachdem ein Nutzer einer Kooperationsgruppe beigetreten ist, so ist er dieser fest zugeordnet. Will er mit anderen Nutzern, die nicht Teil seiner Kooperationsgruppe sind, in beliebiger Richtung kommunizieren, so muss die Gruppe erweitert werden. Dies ist als umständlich zu bewerten, falls eine derartige Kommunikation nur einmalig innerhalb einer Arbeitssitzung auftritt. Daher kann gesagt werden, dass das Prinzip der Kooperationsgruppen recht unflexibel ist.

Allerdings könnten geschachtelten dynamischen Aktionen für kooperative Anwendungen als Ausgangspunkt für weitere Betrachtungen dienen. Durch ihren geschachtelten Aufbau begegnen sie dem Problem der langen Transaktionen. Ein Abbruch einer Teiloperation führt nicht zum Scheitern des gesamten Arbeitsfortschritts. Ein weiterer Punkt ist, dass sie auf den dynamischen Aktionen aufbauen, die ihrerseits die ACID-Eigenschaft Dauerhaftigkeit garantieren und so den Einsatz von Kompensationstransaktionen vermeiden. Lediglich das Prinzip der Kooperationsgruppen gilt es zu überdenken. Dieses Prinzip ist für den speziellen Anwendungsfall nicht geeignet, da Kooperation zwischen allen Designer möglich sein soll. Weiterhin sollte auch ein anderes Konzept zur Bekämpfung von Konflikten, die infolge der fehlenden Serialisierbarkeit auftreten können, entwickelt werden. Ein Spezifikation von Objekten, für die ein kooperativer Zugriff verboten wird, ist als unpraktisch einzustufen.

In der Tabelle 3.1 ist die Bewertung der in diesem Kapitel vorgestellten Modelle noch einmal zusammengefasst dargestellt.

	Unterstützung des Kooperations-prinzips	Grad der Nebenläufigkeit	Komplexität der Fehlerbehandlung	Anwender-unabhängig?
Geschlossen geschachtelte Transaktionen	nein	–	–	–
Offen geschachtelte Transaktionen	ja	hoch	sehr hoch	nein
Sagas	nein	–	–	–
Das ConTract-Modell	ja	hoch	sehr hoch	nein
Das DOM-Transaktions-modell	ja	hoch	sehr hoch	nein
Das CONCORD-Modell	ja	sehr hoch	gering	nein
Geschachtelte dynamische Aktionen	nein	–	–	–
Geschachtelte dynamische Aktionen f. koop. Anwendungen	ja	hoch	–	nein

Tabelle 3.1: Bewertung der existierenden Modelle im Überblick

4 Das Modell

Nachdem im vorhergehenden Kapitel verschiedene Transaktionskonzepte beschrieben und hinsichtlich der in Abschnitt 3.1 gestellten Anforderungen bewertet wurden, soll in diesem Kapitel ein neu entwickeltes Aktionsmodell, welches auf den geschachtelten dynamischen Aktionen für kooperative Anwendungen basiert, detailliert vorgestellt werden. Dieses Modell ist speziell auf die Anforderungen der verteilten Verarbeitung XML-basierter Dokumente zugeschnitten. Ein besonderer Schwerpunkt liegt hierbei auf einem möglichst hohen Grad der Kooperation und Nebenläufigkeit bei der gemeinsamen Arbeit auf den Daten. Als konkreter Anwendungsfall sei hier der in Abschnitt 1.1.1 beschriebene Workgroup-Betrieb des IOSONO-Systems genannt.

4.1 Die Operationen

In diesem Abschnitt sollen zunächst alle Operationen vorgestellt und grundlegend beschrieben werden, die zur Arbeit auf XML-Dokumenten benötigt werden. Dies ist an dieser Stelle wichtig, da das im darauffolgenden Abschnitt beschriebene Aktionskonzept exakt auf die Operationen zugeschnitten ist.

Read: Diese Operation dient dem Auslesen einzelner Attribut-/Textknoten, Elemente oder ganzer Teilbäume. Jedoch müssen drei unterschiedliche Arten des Lesens unterschieden werden. Zum einen kann der Inhalt eines Knotens, z.B. ein Attributwert, ermittelt werden. Zum anderen kann das Lesen lediglich zur Erfassung der Struktur, z.B. der eines Teilbaums, dienen. Diese Art des Lesens ermöglicht u.a., dass ein Knoten gesehen werden kann, der gerade editiert oder verschoben wird. Eine dritte Möglichkeit ist das holografische Lesen von Knoten. Damit kann das Hologramm eines Knotens oder Teilbaumes gesehen werden, der bereits gelöscht wurde. Dies ist notwendig, um z.B. das Löschen des Knotens oder Teilbaumes rückgängig zu machen. Allerdings setzt es die Verwendung des in Abschnitt 4.4 beschriebenen Fehlerbehandlungsmodells voraus. Mindestens eine Leseoperation sollte Bestandteil jeder Operationsfolge sein, da das „blinde“ Ändern von Daten nicht sinnvoll ist.

Edit: Mit Hilfe einer Editieroperation lassen sich die Werte einzelner Attribut- oder Textknoten ändern.

Delete: Die Ausführung dieser Operation führt zum Löschen einzelner Knoten oder ganzer Teilbäume. Dabei ist zu berücksichtigen, dass es nicht möglich ist, ein Vaterelement ohne seine Nachkommen zu entfernen, da ansonsten die Struktur des XML-Dokumentes zerstört würde.

Insert: Mit Hilfe dieser Operation lassen sich einzelne Knoten oder ganze Teilbäume an ein Element des XML-Dokumentes anhängen. Dieses Element wird dann zum Vaterknoten des eingefügten Objekts.

	A 1	A 2	A 3	A 4	
1	$x1 = Edit(x)$	–			
2	–	$x = Reset$			
3	–	–	$x1 = Edit(x)$		
4	–	–		$x1 = Repeat$	$\lightning$

Tabelle 4.1: Beispiel für ein nicht ausführbares Repeat

Move: Zum Verschieben einzelner Knoten oder ganzer Teilbäume innerhalb oder zwischen Dokumenten wird diese Funktion benutzt. Es sei darauf hingewiesen, dass sie sich nicht auf das Löschen des Objektes an einer Stelle und das Einfügen an einer anderen reduzieren lässt. Dies resultiert daraus, dass das Verschieben eines Blattes oder Teilbaumes nicht die Arbeiten beeinträchtigen darf, die sich innerhalb dieses Objekts ereignen. Es soll also möglich sein, während einer Verschiebung eines Objektes Lese-, Editier-, Einfüge-, Lösch- und Verschiebeoperationen innerhalb desselbigen auszuführen. Im Abschnitt 4.3 wird detailliert auf diesen Sachverhalt eingegangen. (Hinweis: Das Anfügen der verschobenen Objekte an ihr neues Vaterelement wird trotzdem weiterhin auf das Insert zurückgeführt.)

Reset: Reset wird benutzt, um ausgeführte Operationen rückgängig zu machen. Dies ist erforderlich, um die durch die fehlende Serialisierbarkeit entstehenden Konflikte und Inkonsistenzen behandeln und das Kooperationsprinzip verwirklichen zu können. Näheres zu dieser Operation folgt in Abschnitt 4.4 .

Repeat: Repeat dient dem Wiederholen der durch ein Reset rückgängig gemachten Operationen. Allerdings ist die Ausführung eines Repeats nicht in jedem Falle möglich. Grafik 4.1 zeigt ein vereinfachtes Beispiel. Zwei Designer führen jeweils zwei Aktionen aus. Die Aktionen $A1$ und $A3$ gehören zu Designer *1*, $A2$ und $A4$ zu Designer *2*. Jede Operation innerhalb einer Aktion erzeugt eine neue Version des Objektes x, wobei die Aktionen immer die aktuelle als Eingabe nutzen. ($A2$ nutzt $x1$ usw.) Durch das Reset der zweiten Aktion wird die Editieroperation von $A1$ rückgängig gemacht und x zur aktuellen Objektversion deklariert. Designer *1* ändert nun den Wert von x und erzeugt damit eine neue Version $x1$. Durch das Repeat der Aktion *4* wird das zuvor erfolgt Reset rückgängig gemacht. Dies hat zur Folge, dass zu einem Zeitpunkt zwei unterschiedliche Versionen $x1$ des Objektes x existieren. Dies führt zum Konflikt mit dem eingesetzten Fehlerbehandlungsmodell.(Siehe Abschnitt 4.4.) Eine Lösung des Problems wäre nur durch einen Eingriff der Bearbeiter möglich.

Diese knappe Darstellung soll zum Grundverständnis ausreichen. Das Zusammenspiel der einzelnen Operationen und ihre Eigenschaften werden in nachfolgenden Abschnitten genauer erläutert.

4.2 Der Aufbau des geschachtelten Aktionsmodells

Die geschachtelten dynamischen Aktionen für kooperative Anwendungen wurden bereits im Abschnitt 3.5 vorgestellt. An dieser Stelle geht es nun darum, dieses Konzept auf die bereits vorgestellten Operationen so zu zuschneiden, dass alle gestellten Anforderungen erfüllt werden.

Die Wurzelaktion: Diese Aktion beginnt mit dem Auschecken der Daten beim Server und endet, sofern sie nicht abgebrochen wird, wenn der Nutzer seine Arbeiten abgeschlossen hat. (Ein Wiedereinchecken ist aufgrund des in Abschnitt 4.4 vorgestellten Fehlerbehandlungsmodells nicht notwendig.) Innerhalb dieser Toplevel-Aktion werden keine Operationen ausgeführt. Diese sind alle in Subaktionen darunterliegender Ebenen gehüllt. Alle Subaktionen (Ebene 1-3) werden innerhalb der Wurzelaktion sequenziell, d.h. in der Reihenfolge, in der die Befehle der Bearbeiter vom System entgegengenommen werden, ausgeführt. Wie für geschachtelte dynamische Aktionen üblich, gilt auch hier: Ein Abbruch der Wurzelaktion führt immer zum Abbruch aller Subaktionen. Der umgekehrte Fall gilt jedoch nicht. Die Wurzelaktion ist vom Grundprinzip her offen geschachtelt, d.h. die Ergebnisse der Subaktionen erster Ebene sind nach deren Übergang in den Zustand *Completed* für alle sichtbar.

Die Subaktion erster Ebene: Der Aufbau dieser Aktion ist abhängig von den Operationen, die darin ausgeführt werden sollen. Sie besteht entweder aus exakt einer Leseoperation auf genau einem Knoten oder aus einer Folge von Subaktionen der zweiten Ebene. Im Fehlerfall gilt für die Subaktion erster Ebene:

- Ihr Abbruch führt zum Abbruch aller Kindaktionen, jedoch nicht zu dem der Wurzelaktion.
- Der Abbruch einer Kindaktion führt nur dann zum Abbruch der Subaktion erster Ebene, wenn diese sich noch nicht im Zustand *Completed* befindet.

Die Ergebnisse der Subaktion erster Ebene sind nach dem Übergang dieser Aktion in den Zustand *Completed* für alle Aktionen sichtbar. (Hinweis: Die Subaktion der ersten Ebene wird im Folgenden häufig auch einfach als Subaktion bezeichnet.)

Die Subaktion zweiter Ebene: Diese beinhaltet entweder genau eine beliebige Operation auf genau einem Knoten oder eine Folge von Subaktionen der dritten Ebene. Im Fehlerfall gelten dieselben Regeln wie für die Subaktion der ersten Ebene. Für die Sichtbarkeit gilt Folgendes: Die Ergebnisse werden mit dem Übergang in den Zustand *Completed* für die einhüllende Subaktion der ersten Ebene sichtbar. Diese kann sie an die Geschwister der Subaktion der zweiten Ebene weitergeben. (Hinweis: Die Subaktion der zweiten Ebene wird im Folgenden häufig auch einfach als SubSubaktion bezeichnet.)

Die Subaktion dritter Ebene: Sie enthält exakt eine beliebige Operation auf genau einem Knoten. Ihr Abbruch führt nur dann zu dem der einhüllenden Subaktion, sofern sich diese noch nicht im Zustand *Completed* befindet. Die Ergebnisse sind nur für die einhüllende Subaktion der zweiten Ebene sichtbar. (Hinweis: Die Subaktion der dritten Ebene wird im Folgenden häufig auch einfach als SubSubSubaktion bezeichnet.)

Zur Vervollständigung der Darstellung des Aufbaus soll dieser in der erweiterten Backus-Naur-Form formal spezifiziert werden. („{...}“ bedeutet in diesem Fall, dass das Terminal-/Nichtterminalsymbol mindestens einmal vorhanden ist.) Jede Operation setzt sich hier aus dem Namen der Operation und dem Bezugsobjekt zusammen. *ReadNode* ist z.B. eine Leseoperation

auf einem Knoten.

$$Wurzelaktion = \{Subaktion\} \tag{4.1}$$
$$Subaktion = \text{“}ReadNode\text{”}|\{SubSubaktion\} \tag{4.2}$$
$$SubSubaktion = \text{“}ReadNode\text{”}|\text{“}EditNode\text{”}|\text{“}DeleteNode\text{”}| \text{“}InsertNode\text{”}|\text{“}MoveNode\text{”}|\text{“}ResetNode\text{”}| \text{“}RepeatNode\text{”}|\{SubSubSubaktion\} \tag{4.3}$$
$$SubSubSubaktion = \text{“}ReadNode\text{”}|\text{“}EditNode\text{”}|\text{“}DeleteNode\text{”}| \text{“}InsertNode\text{”}|\text{“}MoveNode\text{”}|\text{“}ResetNode\text{”}| \text{“}RepeatNode\text{”} \tag{4.4}$$

Besonders auffällig ist, dass sich die Regeln für die SubSubaktion und die SubSubSubaktion sehr stark ähneln. Das hat folgenden Hintergrund. Wie bereits erwähnt, können alle Operationen, bis auf das Editieren, sowohl auf einzelnen Knoten als auch auf Teilbäumen ausgeführt werden. Um z.B. Teile einer Operation auf einem Teilbaum rücksetzen zu können, müssen die Operationen auf Teilbäumen in Operationen auf Einzelknoten zerlegt werden. Allerdings müssen diese atomaren Aktionen in eine Subaktion der nächst höheren Ebene eingehüllt werden, um eine atomare Sperranforderung zu ermöglichen, die die Deadlockfreiheit garantiert. (Siehe Abschnitt 4.5.6)

Nachdem nun der prinzipielle Aufbau des Modells festgelegt wurde, muss spezifiziert werden, welche Operationsfolgen mit Hilfe dieses Modells ausgeführt werden sollen.

- Es sei $A = \{Edit\,(Z)\,, Delete\,(X)\,, Insert\,(Y)\,, Move\,(X,Y)\,, Reset\,(X)\,,$ $Repeat\,(X)\}$ die Menge der Änderungsoperationen. Dabei ist X ein XML-Knoten oder Teilbaum, Y ein einzelnes XML-Element, das das Ziel einer Verschiebung oder einen Einfügepunkt darstellt und Z ein Attribut- oder Textknoten.
- $Read_i\,(L_i)\,, i \in \mathbb{N}$, bezeichnet eine Leseoperation auf L_i, wobei L_i ein einzelner XML-Knoten oder ein zusammenhängender XML-Baum ist. Das i ist ein Index zur Unterscheidung verschiedener Leseoperationen.
- $S_i = \{S_{i_1}, ..., S_{i_k}\}, k \in \mathbb{N}$, ist die Menge aller Teilmengen von L_i für die gilt: S_{i_k} ist zusammenhängend und unterliegt einer Verschiebung, $S_{i_r} \cap S_{i_s} = \emptyset, r, s \in \mathbb{N}, r \neq s$. S_{i_k} ist vollständig, d.h. es gibt keine Knoten außerhalb dieser Menge, die zur gleichen Verschiebung gehören.
- $F_j = Read_{j_1}\,(L_{j_1})\,Read_{j_2}\,(L_{j_2})\,...Read_{j_{n-1}}\left(L_{j_{n-1}}\right)Read_{j_n}\,(L_{j_n})$, $i, j \in \mathbb{N}$ ist eine Folge von Leseoperationen wobei gilt: $L_{j_i} \subseteq S_{j_{i-1_k}}$, $k \in \mathbb{N}$, $i \in \{2, ..., n\}$, $n \in \mathbb{N}$. Solch eine Leseoperationsfolge hat folgende Bedeutung. Beim Auslesen eines XML-Baumes A kann es vorkommen, dass ein Teilbaum B von A in diesem Augenblick verschoben wird. B erscheint dem Betrachter allerdings nur als Struktur, da vom gegenwärtigen Gesichtspunkt keine Bearbeitung auf B möglich sein soll. Durch eine zweite Leseoperation ist es ihm allerdings möglich, in den Teilbaum B hineinzuspringen. Dadurch ändert er seinen Fokus. Ihn interessiert jetzt nur noch das Innere von B und keine äußeren Strukturen. Es ist ihm folglich auch egal, dass B gerade verschoben wird. Durch diese Standpunktänderung ist es nun möglich, weitere Operationen auf B auszuführen.
- Eine Operationsfolge OF besteht entweder aus

Fall 1:

$$OF = \text{“}F_j\text{”}\,[\text{“}O\text{”}] \tag{4.5}$$

(Die Operationsfolge besteht also aus genau einer Leseoperationsfolge und höchstens einer Änderungsoperation.) Es muss gelten:

- Wenn $O \in \{Reset\,(X)\,, Repeat\,(X)\}$, dann $X \subseteq L_{j_{|F_j|}}$. Ein Reset oder Repeat kann also auch auf einer sich in einer Verschiebung befindenden Knotenmenge ausgeführt werden, da diese in jedem Fall abgebrochen wird, wie später noch zu sehen ist.
- Wenn $O \in \{Delete\,(X)\,, Move\,(X,Y)\}$, dann muss gelten:
  * Wenn $S_j \subsetneq L_{j_{|F_j|}}$, dann $X \subseteq \left(L_{j_{|F_j|}} \backslash S_j\right)$und $Y \in \left(L_{j_{|F_j|}} \backslash S_j\right)$. Das bedeutet, dass sich die Operationen nicht auf Teile der Lesemenge erstrecken dürfen, die sich vom augenblicklichen Standpunkt gesehen in einer Verschiebung befinden.
  * Wenn $L_{j_{|F_j|}} = S_{j_l}, l \in \mathbb{N}$, dann $X \subsetneq L_{j_{|F_j|}}$ und $Y \in L_{j_{|F_j|}}$. Befindet sich also die komplette Lesemenge in einer Verschiebung, so darf sich eine Verschiebung oder Löschoperation nicht auf das Wurzelelement erstrecken. Der Grund wird im nächsten Abschnitt deutlich. Es sei noch darauf hingewiesen, dass der lesende Designer die bereits laufende Verschiebung nicht bemerkt, da ihn die Strukturen außerhalb seiner Lesemenge nicht interessieren.
- Wenn $O \in \{Edit\,(Z)\,, Insert\,(Y)\}$, dann muss gelten:
  * Wenn $S_j \subsetneq L_{j_{|F_j|}}$, dann $Z, Y \in \left(L_{j_{|F_j|}} \backslash S_j\right)$. Das bedeutet, dass die Operationen keine Elemente der Lesemenge betreffen dürfen, die sich vom augenblicklichen Standpunkt gesehen in einer Verschiebung befinden.
  * Wenn $L_{j_{|F_j|}} = S_{j,}, l \in \mathbb{N}$, dann $Z, Y \in L_{j_{|F_j|}}$. Befindet sich also die komplette Lesemenge in einer Verschiebung, so darf das Einfügen oder Editieren auch ein Wurzelelement betreffen.
- $X \cap Y = \emptyset$. Der Einfügepunkt darf also nie Teil der zu verschiebenden Menge sein. Dies ist rein intuitiv klar.

oder aus

Fall 2:

$$OF = \text{“}F_j\text{”“}F_k\text{”“}Move\,(X,Y)\,\text{”} \tag{4.6}$$

(Die Operationsfolge besteht also aus genau zwei verschiedenen Leseoperationsfolgen und genau einer Verschiebe-Operation.) Diese Operationsfolge ist dafür notwendig, um z.B. einen Teilbaum eines Dokumentes an ein Element in einem anderen Dokument anzuhängen. Es muss gelten:

- $L_{j_1} \cap L_{k_1} = \emptyset$, $j \neq k$. Die initialen Lesesets der beiden Lesefolgen sind also völlig unterschiedlich. (Es könnte sich also z.B. um zwei unterschiedliche Dokumente handeln.)
- Wenn $S_j \subsetneq L_{j_{|F_j|}}$, dann $X \subseteq \left(L_{j_{|F_j|}} \backslash S_j\right)$. Das bedeutet, dass sich die Operationen nicht auf Teile der Lesemenge erstrecken dürfen, die sich vom augenblicklichen Standpunkt gesehen in einer Verschiebung befinden.
- Wenn $L_{j_{|F_j|}} = S_{j_k}$, dann $X \subsetneq L_{j_{|F_j|}}$. Befindet sich also die komplette Lesemenge in einer Verschiebung, so darf sich eine weitere Verschiebung nicht auf

das Wurzelelement erstrecken.
- Wenn $S_k \subsetneq L_{k_{|F_k|}}$, dann $Y \in \left(L_{k_{|F_k|}} \backslash S_k\right)$. Das bedeutet, dass die Operation keine Elemente der Lesemenge betreffen darf, die sich vom augenblicklichen Standpunkt gesehen in einer Verschiebung befinden.
- Wenn $L_{k_{|F_k|}} = S_{k_l}, l \in \mathbb{N}$, dann $Y \in L_{k_{|F_k|}}$. Befindet sich also die komplette Lesemenge in einer Verschiebung, so darf das Einfügen auch ein Wurzelelement betreffen.

Andere Operationsfolgen außer die unter Fall 1 und Fall 2 spezifizierten sind nicht zulässig. Allerdings wird die Menge der Operationsfolgen durch das in Abschnitt 4.3 definierte Sperrmodell weiter eingeschränkt.

Die in den Gleichungen 4.5 und 4.6 definierten Operationsfolgen werden auf das Modell übertragen. Dabei wird immer genau eine Operationsfolge auf eine Subaktion erster Ebene abgebildet. Die dabei angewandte Vorgehensweise ist im Algorithmus 1 angedeutet. Jede Operation auf einem Teilbaum wird dabei in elementare Operationen zerlegt. Jede resultierende elementare Operation wird in eine Subaktion der dritten Ebene gehüllt, sofern es mehrere Operationen in der Operationsfolge gibt. Besteht die Operationsfolge nur aus einer Leseoperation auf einem Teilbaum, so werden die elementaren Leseoperationen in Subaktionen der zweiten Ebene gehüllt. Beziehen sich Operationen von vornherein auf einzelne Knoten, so werden sie ebenfalls in Subaktionen der zweiten Ebene gehüllt.

Zum Schluss soll ein kleines Beispiel zum Aufbau des Modells gegeben werden. In Abbildung 4.1 ist ein vereinfachter XML-Baum skizziert. Die gestrichelt umrandeten Elemente befinden

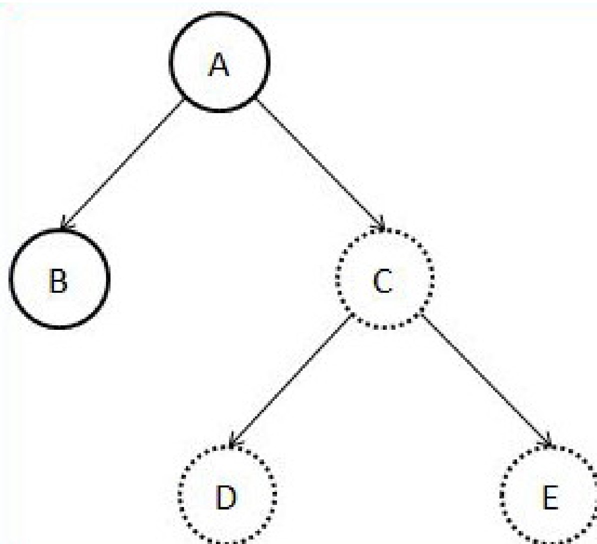

Abbildung 4.1: Ein Beispiel-XML-Baum

sich gerade in einer Verschiebung. Ein Designer will den Knoten *E* löschen, weiß aber nicht, wo er sich befindet. Er führt nun zunächst eine Leseoperation auf den Teilbaum *A* (*A* ist das Wurzelelement des Teilbaums.) aus. Dabei findet er den Knoten *E*, stellt aber fest, dass sich der Teilbaum *C*, dem *E* angehört, in einer Verschiebung befindet. Mit einer zweiten Leseoperation springt er nun in den Teilbaum *C* und führt danach eine Löschoperation auf dem Knoten *E* aus. Die dazugehörige Operationsfolge sieht wie folgt aus:

$$OF = Read(A, B, C, D, E)Read(C, D, E)Delete(E) \tag{4.7}$$

Algorithm 1 Abbildung der Operationsfolgen auf das Modell

```
1: if (OF == ReadNode) then
2:   OF einhüllen in Subaktion;
3: else
4:   if (OF == ReadTree) then
5:     for (i = 1, i < n, i + +) do {Zerlegung von OF in n ReadNode unter
       Berücksichtigung der Reihenfolge}
6:       OF[i]einhüllen in SubSubaktion[i];
7:       SubSubaktion[i] einhüllen in Subaktion;
8:     end for
9:   else
10:    for (i = 1, i < n, i + +) do {Zerlegung von OF in n Operationen unter
       Berücksichtigung der Reihenfolge}
11:      if (OF[i] == OperationNode) then
12:        OF[i]einhüllen in SubSubaktion[i];
13:      else
14:        if (OF[i] == OperationTree) then
15:          for (j = 1, j < o, j + +) do {Zerlegung einer Operation auf einem Baum in o
             Operationen auf einem Knoten unter Berücksichtigung der Reihenfolge}
16:            OF[i][j] einhüllen in SubSubSubaktion[j];
17:            SubSubSubaktion[j] einhüllen in SubSubaktion[i];
18:          end for
19:        end if
20:      end if
21:      SubSubaktion[i] einhüllen in Subaktion;
22:    end for
23:  end if
24: end if
```

Diese Operationsfolge muss nun gemäß dem Abbildungsalgorithmus auf das Modell übertragen werden. Abbildung 4.2 zeigt das resultierende Aktionsmodell. Gut zu erkennen ist, dass die Leseoperationen auf Teilbäumen in Operationen auf Knoten zerlegt werden, sie aber durch die Einhüllung in eine SubSubaktion trotzdem den Charakter einer semantischen Einheit behalten. Weiterhin ist der Wechsel des Fokusses gut zu sehen. Beim Lesen auf dem Teilbaum *A* erscheinen die sich in der Verschiebung befindenden Elemente lediglich als Struktur. Durch das Ausführen einer zweiten Leseoperation auf dem Teilbaum *C* kann dann auch ein inhaltliches Lesen der Elemente und die Ausführung einer Änderungsoperation, hier das Löschen von Knoten *E*, erfolgen.

Zusammenfassend soll an dieser Stelle Folgendes festgehalten werden. Alle Aktivitäten auf einem XML-Dokument sind in Operationsfolgen gehüllt, deren Aufbau speziellen Regeln unterliegt. Sie bestehen aus mindestens einer Leseoperation, da das „blinde" Ändern von Daten oder Strukturen nicht sinnvoll ist, und höchstens einer Änderungsoperation. Die Subaktion erster Ebene stellt die wichtigste Einheit dar. Sie bildet semantisch gesehen einen Container für die Aktivitäten. Alle Operationsfolgen werden auf die starre Struktur der Subaktionen abgebildet. Die Ursachen für den strengen Aufbau der Operationsfolgen sowie des Aktionsmodells liegen darin begründet, dass gewisse Eigenschaften garantiert werden sollen, wie z.B. eine erhöhte

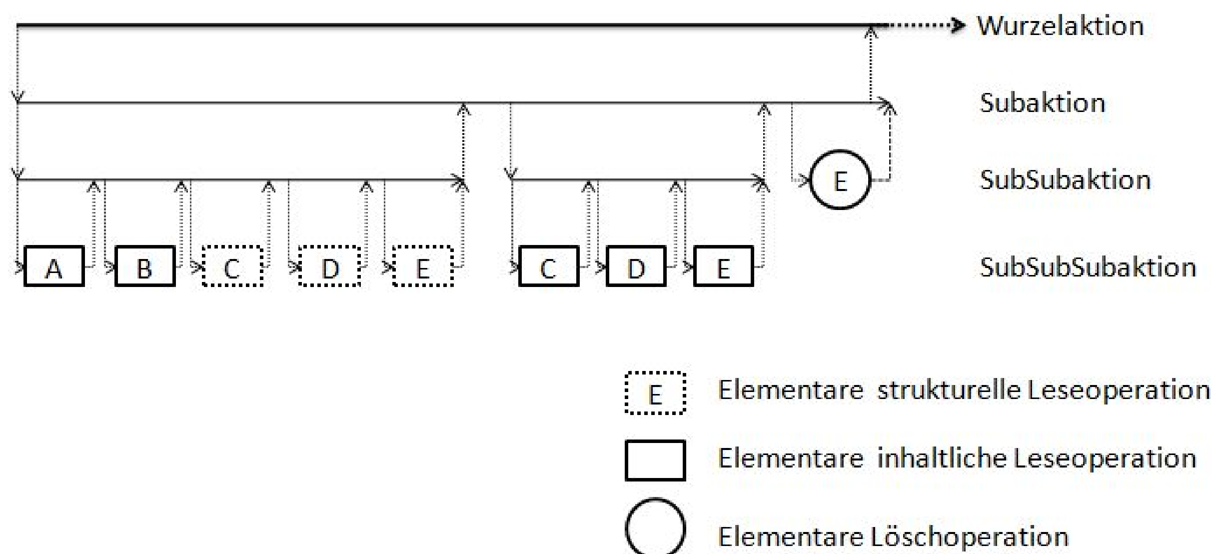

Abbildung 4.2: Die Abbildung der Beispiel-Operationsfolge auf das Modell

Nebenläufigkeit von Aktionen oder die Deadlockfreiheit. Eine genaue Erörterung aller Eigenschaften ist Gegenstand von Abschnitt 4.5.

4.3 Das Sperrmodell

Im vorhergehenden Abschnitt wurde sowohl der Aufbau des geschachtelten Aktionsmodells als auch die allgemeine Menge der zulässigen Operationsfolgen spezifiziert. Nun ist es wichtig, ein Sperrmodell zu entwickeln, welches die korrekte nebenläufige Ausführung von Operationsfolgen verschiedener Bearbeiter gewährleistet. Es muss festgestellt werden, welche Operationen gleichzeitig auf den Daten ausgeführt werden können. Daraus ergeben sich die benötigten Sperren. Ein Sperrprotokoll regelt das korrekte Setzen und Freigeben der Sperren und schränkt dabei die Menge der zulässigen Operationsfolgen weiter ein.

4.3.1 Untersuchung der Kompatibilität der Operationen

Zunächst wird die Kompatibilität der einzelnen Operationen zueinander untersucht. Bezugspunkt ist dabei ein gemeinsamer XML-Knoten. Tabelle 4.2 stellt diesen Sachverhalt zunächst ohne die Operation Move dar. Dabei bedeutet $\surd$, dass die beiden Operationen vollständig kompatibel sind, $+$, dass sie bedingt kompatibel sind und $-$, dass sie gar nicht kompatibel sind. Im Folgenden soll die Tabelle kurz erläutert werden.

Beim gleichzeitigen inhaltlichen wie auch strukturellen Lesen eines XML-Knotens treten keine Konflikte auf. Daher sind beide Operationen vollständig zueinander kompatibel.

Das inhaltliche Lesen und gleichzeitige Edit, Delete, Reset oder Repeat ist nicht möglich. Das strukturelle Lesen wird jedoch auch auf einem Knoten zugelassen, der editiert wird. Dies hat folgende Ursachen. Das Editieren führt zu einer Veränderung des Inhalts. Ein anderer Bearbei-

	Read (struk.)	Read (inh.)	Read (holo.)	Edit	Delete	Insert	Reset	Repeat
Read (struk.)	$\surd$	$\surd$	$\surd$	$\surd$	–	$\surd$	–	–
Read (inh.)	$\surd$	$\surd$	$\surd$	–	–	$\surd$	–	–
Read (holo.)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Edit	$\surd$	–	$\surd$	–	–	–	–	–
Delete	–	–	$\surd$	–	–	–	–	–
Insert	$\surd$	$\surd$	$\surd$	–	–	$\surd$	–	–
Reset	–	–	$\surd$	–	–	–	–	–
Repeat	–	–	$\surd$	–	–	–	–	–

Tabelle 4.2: Konfliktmatrix ohne die Operation Move

ter sollte diese Änderungen erst nach ihrer Vollendung sehen können, weshalb ihm der betroffene Knoten lediglich als Strukturelement ohne Inhalt dargestellt wird. Beim Löschen tritt ein Sonderfall ein. Hier wird zwar sowohl der Inhalt als auch die Struktur (Knoten wird entfernt) verändert, jedoch sollte das betroffene Element zumindest als eine Art „Hologramm“ dargestellt werden. Daher ist ein holografisches Lesen auf diesem Element zulässig. Ansonsten wäre das Reset einer Löschoperation nicht möglich. (Auf einem nicht vorhandenen Knoten kann keine Operation ausgeführt werden.) Beim Einfügen und gleichzeitigen Lesen tritt ebenfalls eine Besonderheit auf. Prinzipiell ist das inhaltliche Lesen eines XML-Knotens und das gleichzeitige Anfügen von Knoten an eben diesen uneingeschränkt möglich. Es kann jedoch weder ein inhaltliches, noch strukturelles oder holografisches Lesen auf den anzufügenden Knoten ausgeführt werden, da diese zu diesem Zeitpunkt noch nicht vollständig bekannt sein müssen. Beim Reset und Repeat ist ein holografisches Lesen uneingeschränkt möglich. Diese Operationen stellen semantisch gesehen eine Konkatenation von Änderungsoperationen dar, mit der Besonderheit, dass die Struktur von Objekten immer bekannt ist, und zwar auch im Falle des Einfügens. Diese Operation stellt nämlich in diesem Zusammenhang die Kompensationsaktion des Löschens dar. Ein strukturelles Lesen muss beim Reset oder Repeat verboten werden, da diese Operationen das Löschen eines Knotens beinhalten können. Gleiches gilt für das inhaltliche Lesen.

Das gleichzeitige Editieren zweier Aktionen auf ein- und demselben Objekt ist selbstverständlich verboten, da diese Operationen eine Veränderung des Inhalts bedeuten und so zu einer neuen Version des Objektes führen. Zwei unterschiedliche Versionen eines Objekts sind aufgrund des eingesetzten Fehlerbehandlungsmodells nicht erlaubt. Das gleichzeitige Editieren und Löschen schließt sich ebenfalls auf. Es darf weder einer ändernden Aktion das Objekt entrissen werden (außer beim Reset und Repeat), noch darf ein Objekt editiert werden, welches (zumindest inhaltlich) nicht mehr vorhanden ist. Das gleichzeitige Editieren und Einfügen auf einem Knoten wäre prinzipiell schon möglich, jedoch schließt es sich aufgrund des zugrundeliegenden XML-Modells (taDOM) aus. Eingefügt werden kann nur an XML-Elemente. Editiert werden dürfen nur Attribut- oder Textknoten. Dem zufolge beziehen sich beide Operationen immer auf unterschiedliche Knotentypen.

Das gleichzeitige Löschen eines Knotens durch zwei unterschiedliche Aktionen schließt sich aus, da es semantisch nicht sinnvoll ist, ein Objekt zweimal zu entfernen. Das Löschen eines Knotens und das gleichzeitige Einfügen von Objekten an diesen ist ebenfalls verboten, da einerseits an einen gelöschten Knoten nichts angefügt werden kann und andererseits einer Einfügeoperation nicht das Zielelement entrissen werden darf.

Das gleichzeitige Einfügen von Elementen (Knoten oder Teilbäume) an einen Knoten durch unterschiedliche Aktionen stellt kein Problem dar, da das Zielelement, unter Voraussetzung, dass das taDOM-Modell genutzt wird, nicht verändert wird.

Reset und Repeat nehmen eine besondere Stellung ein. Sie dienen bekanntlich dazu, Änderungsoperationen rückgängig bzw. wiederholen zu können, um den Problemen, die aus der nicht gewährleisteten Serialisierbarkeit entstehen, zu begegnen. Aufgrund dieser Tatsache bekommen diese Operationen ein gewisses Sonderrecht zugesprochen. Wird auf einen Knoten ein Reset oder Repeat ausgeführt, so führt dies zum Abbruch aller Änderungsoperationen, außer einem anderen Reset oder Repeat, die auf diesem Knoten in diesem Moment ausgeführt werden. Das gleichzeitige Ausführen zweier Repeats oder Resets auf dem gleichen Objekt ist verboten. Das bedeutet also, der Bearbeiter, der das Reset oder Repeat zuerst für den Knoten aufgerufen hat, hat den Vorrang. Ein anderer Nutzer kann dann kein Reset oder Repeat auf dem Knoten ausführen. Eine Anmerkung soll an dieser Stelle noch zu der Knotenmenge gemacht werden, die bei einem Reset oder Repeat betroffen ist. Der Bearbeiter wählt innerhalb seines Lesesets die Knoten aus, für die ein älterer bzw. neuerer Zustand wiederhergestellt werden soll. Allerdings kann es erforderlich sein, dass diese Menge dynamisch erweitert wird. Dieser Fall tritt z.B. ein, wenn das Einfügen eines Elementes, welches Kinder hat, rückgängig gemacht werden soll. Das ist nur möglich, wenn auch die Kinder mit entfernt werden, da ansonsten die Struktur des XML-Dokumentes nicht erhalten wird. Eine derartige dynamische Erweiterung der betroffenen Knotenmenge wird mit Hilfe des Fehlerbehandlungsmodells realisiert.

Zum Schluss soll an dieser Stelle die Kompatibilität der Verschiebung mit allen anderen Operationen untersucht werden. Dabei ist jedoch eine besondere Vorgehensweise erforderlich, da zwei unterschiedliche Fokusse des Bearbeiters unterschieden werden müssen. Zum einen kann er Teil einer Verschiebung sein, zum anderen kann er innerhalb eines XML-Teilbaumes auf eine darunterliegende Verschiebung blicken. Aus diesem Grund werden jetzt die Kompatibilitätsmatrizen für diese beiden Fälle getrennt dargestellt. Bezugspunkt ist wieder ein einzelner XML-Knoten, wobei jedoch gerade beim Löschen & Verschieben und Verschieben & Verschieben unterschieden werden muss, ob dieser Knoten das Wurzelelement einer Verschiebung ist.

In Tabelle 4.3 wird der erste Fall aufgegriffen. Der Fokus des Bearbeiters beschränkt sich zunächst einmal nur auf den Knoten, der sich in der Verschiebung befindet. Alle Strukturen außerhalb dieses Objektes interessieren ihn zunächst nicht. In diesem Fall ist das Lesen, sofern es nicht mit anderen Operationen gemäß 4.2 kollidiert, uneingeschränkt möglich. Auch das Editieren stellt dann kein Problem dar. Beim Löschen ist jedoch eine Einschränkung zu machen. Ist der Knoten das Wurzelelement eines sich in der Verschiebung befindenden Objektes (Knoten oder Teilbaum), so darf dieser Knoten nicht gelöscht werden. Der Grund dafür ist, dass ein derartiges Vorgehen zur Zerstörung des kompletten Objekts führen würde. Der Bearbeiter, der die Verschiebung durchführt, hätte in diesem Moment kein Element mehr zur Verfügung, was er bewegen könnte. In einer Umgebung in der das Verschieben per Drag&Drop realisiert wird, würde das bedeuten, das dem Bearbeiter der „Anfasspunkt", also das Wurzelelement des zu verschiebenden Objektes, entzogen wird. Auch das Verschieben eines Wurzelelementes eines sich in der Verschiebung befindenden Objektes ist aus dem eben genannten Grund nicht zulässig. Ein Reset oder Repeat auf einen Knoten, der verschoben wird, führt in jedem Fall zum Abbruch der Verschiebung, da eine sofortige Behandlung eines Serialisierbarkeitskonfliktes uneingeschränkt ermöglicht werden soll.

Tabelle 4.4 stellt die Kompatibilitätsmatrix für den Fall dar, dass der Bearbeiter eine Verschie-

	Read (struk.)	Read (inh.)	Read (holo.)	Edit	Delete	Insert	Move	Reset	Repeat
Move	$\surd$	$\surd$	$\surd$	$\surd$	+	$\surd$	+	–	–

Tabelle 4.3: Konfliktmatrix für die Verschiebung - der Bearbeiter ist Teil der Verschiebung

bung in seinem Blickfeld hat, also z.B. in einem Teilbaum auf einen Attributknoten schaut, der gerade verschoben wird. Auf den sich in der Verschiebung befindenden Knoten darf lediglich

	Read (struk.)	Read (inh.)	Read (holo.)	Edit	Delete	Insert	Move	Reset	Repeat
Move	$\surd$	–	–	–	–	–	–	–	–

Tabelle 4.4: Konfliktmatrix für die Verschiebung - der Bearbeiter hat die Verschiebung im Blickfeld

ein strukturelles Lesen ausgeführt werden, da sich dieser in diesem Moment nicht mehr an dem ursprünglichen Platz befindet. Gemäß den Gleichungen 4.5 und 4.6 kann der Bearbeiter jedoch in die Verschiebung hineinspringen. Er verliert dabei zwar seinen bisherigen Fokus, kann aber dann auch inhaltlich und holografisch lesend auf den Knoten zugreifen. Editierend kann auf einen sich in der Verschiebung befindenden Knoten von außen ebenfalls nicht zugegriffen werden. Dies erfordert wiederum ein Hineinspringen in die Verschiebung mittels einer Leseoperation. Gleiches gilt für das Löschen, Einfügen und Verschieben. Reset und Repeat nehmen wiederum eine Sonderstellung ein. Sie führen aus den bereits mehrfach erwähnten Gründen in jedem Fall zum Abbruch der Verschiebung.

4.3.2 Die Sperrtypen

Aus den Konfliktmatrizen, die im vorhergehenden Abschnitt hergeleitet wurden, lassen sich folgende Sperrtypen für die einzelnen Operationen ableiten:

- Die Leseoperation bekommt aufgrund ihrer drei unterschiedlichen Ausführungsmodi drei Sperrtypen zugeordnet. Die CRL (*Content-Read-Lock*) ist die Sperre für das inhaltliche Lesen, die SRL (*Structure-Read-Lock*) ist die Sperre für das strukturelle Lesen und die HRL (*Holographical-Read-Lock*) ist die Sperre für das holografische Lesen.
- Für das Editieren wird der Sperrtyp EL (*Edit-Lock*) eingeführt.
- Das Löschen eines Knotens wird durch den Sperrtyp DL (*Delete-Lock*) signalisiert.
- Dem Einfügen wird der Sperrtyp IL (*Insert-Lock*) zugeordnet.
- Die Verschiebung bekommt den Sperrtyp ML (*Move-Lock*).
- Für das Reset und Repeat ist ein gemeinsamer Sperrtyp RRL (*Reset-/Repeat-Lock*) ausreichend.

Die Verträglichkeit der Sperren auf einem Knoten entspricht der der dazugehörenden Operationen. Dem entsprechend bezieht sich Tabelle 4.5 auf Tabelle 4.2, Tabelle 4.6 auf Tabelle 4.3 und Tabelle 4.7 auf Tabelle 4.4.

	SRL	CRL	HRL	EL	DL	IL	RRL
SRL	√	√	√	√	–	√	–
CRL	√	√	√	–	–	√	–
HRL	√	√	√	√	√	√	√
EL	√	–	√	–	–	–	–
DL	–	–	√	–	–	–	–
IL	√	√	√	–	–	√	–
RRL	–	–	√	–	–	–	–

Tabelle 4.5: Sperren-Konfliktmatrix ohne die ML

	SRL	CRL	HRL	EL	DL	IL	ML	RRL
ML	√	√	√	√	+	√	+	–

Tabelle 4.6: Sperren-Konfliktmatrix für die Verschiebung - der Bearbeiter ist Teil der Verschiebung

	SRL	CRL	HRL	EL	DL	IL	ML	RRL
ML	√	–	–	–	–	–	–	–

Tabelle 4.7: Konfliktmatrix für die Verschiebung - der Bearbeiter hat die Verschiebung im Blickfeld

Nachdem nun die Kompatibilität der einzelnen Operationen bzw. Sperren auf einem Knoten untersucht wurde, muss geklärt werden, welche Sperren bei einer Operation hinsichtlich der betroffenen Knotenmenge angefordert werden müssen. Beim Löschen eines Elementes, welches Kinder hat, ist es beispielsweise erforderlich, dass diese Kinder ebenfalls gelöscht werden, da ansonsten die Struktur des XML-Dokumentes nicht erhalten bleibt. Ebenso muss eine Verschiebung auf einem Element auch eine Verschiebung der Kinder beinhalten. Bei der Anforderung der Sperren müssen dann natürlich für jeden Knoten die Sperren-Kompatibilitätsmatrizen berücksichtigt werden. Tabelle 4.8 stellt zusammenfassend dar, welche Sperren bei welcher Ausführungsart (hinsichtlich der betroffenen Objekte) gesetzt werden müssen. Das $+$ beim Löschen und Verschieben eines einzelnen Elementknotens bedeutet, dass dies nur möglich ist, wenn dieser Knoten keine Nachkommen hat. Beim Reset und Repeat muss beachtet werden, dass die Sperrmenge durch das Fehlerbehandlungsmodell in Abhängigkeit von den zurückzusetzenden bzw. zu wiederholenden Operationen noch erweitert werden kann.

4.3.3 Das Sperrprotokoll

Für das korrekte Sperren und Entsperren ist ein Protokoll notwendig. Jede Operationsfolge muss diesem Protokoll folgen. Zunächst einige Festlegungen:

- Jede Operationsfolge OF ist gemäß Gleichung 4.5 oder 4.6 aufgebaut.
- F entspricht der ersten Leseoperationsfolge in Gleichung 4.5 oder 4.6.
- G entspricht der zweiten Leseoperationsfolge in Gleichung 4.6.
- V ist die zusammenhängende und vollständige Knotenmenge einer Operation.

	Attribut-/Textknoten	Elementknoten	Teilbaum
Read (struk.)	SRL auf Knoten	SRL auf Knoten	SRL auf alle Knoten
Read (inh.)	CRL auf Knoten	CRL auf Knoten	CRL auf alle Knoten
Read (holo.)	HRL auf Knoten	HRL auf Knoten	HRL auf alle Knoten
Edit	EL auf Knoten	–	–
Delete	DL auf Knoten	DL auf Knoten +	DL auf alle Knoten
Insert	–	IL auf Knoten	–
Move	ML auf Knoten IL auf Zielelement	ML auf Knoten + IL auf Zielelement	ML auf alle Knoten IL auf Zielelement
Reset	RRL auf Knoten +	RRL auf Knoten +	RRL auf alle Knoten +
Repeat	RRL auf Knoten +	RRL auf Knoten +	RRL auf alle Knoten +

Tabelle 4.8: Zu setzende Sperren bei unterschiedlichen Ausführungsarten der Operationen

- $X \in V$.

Das Protokoll besteht aus folgenden Regeln:

1. Für alle $X \in V$ der ersten Leseoperation aus F oder G werden für die Anforderung der Sperren zwei Fälle unterschieden:
   a) Auf X liegt keine ML einer anderen Operationsfolge:
      i. Liegt auf X keine RRL, DL oder EL und ist der Knoten nicht gelöscht, dann erhält OF eine CRL auf X.
      ii. Liegt auf X eine RRL oder DL oder ist der Knoten gelöscht, dann erhält OF eine HRL auf X.
      iii. Liegt auf X keine RRL und keine DL aber eine EL und ist der Knoten nicht gelöscht, so erhält OF eine SLR auf X.
   b) Auf X liegen $r \geq 1$ MLs von r verschiedenen Operationsfolgen:
      i. Haben alle Knoten aus $\{V \setminus X\}$ der ersten Leseoperation mindestens r MLs, dann prüfe Bedingung 1(a)i - 1(a)iii, ansonsten erhält OF nur eine SRL auf X.
2. Für jede weitere Leseoperation aus F oder G gilt:
   a) Für die erste zu setzende Sperre muss gelten:
      i. OF muss auf X eine SRL besitzen.
      ii. Auf X müssen mindestens $r+1$ MLs von $r+1$ verschiedenen Operationsfolgen liegen, wobei r die minimale Anzahl von MLs ist, die auf irgendeinem Knoten in V liegt.
      iii. Ist eine der beiden Bedingungen 2(a)i oder 2(a)ii nicht erfüllt, so führt dies zum Abbruch der Operationsfolge. Andernfalls ergibt sich V aus $\{$ X, Alle Nachkommen von X, die gelesen werden sollen $\}$. OF verliert alle Lesesperren außerhalb von V und alle Lesesperren innerhalb von V werden gemäß 1b neu vergeben.
3. Für eine Änderungsoperation gilt:
   a) Für das Editieren eines Knotens X müssen folgende Bedingungen geprüft werden:
      i. V entspricht dem V der letzten Leseoperation aus F.
      ii. X ist ein Attribut-/Textknoten.

iii. Die Operationsfolge OF hält eine CRL auf X und auf keinen Knoten aus V eine EL.

iv. Ist eine der Bedingungen 3(a)i - 3(a)iii nicht erfüllt, führt das zum Abbruch der Operationsfolge. Andernfalls erhält OF auf X eine EL und verliert alle CRLs, SRLs und HRLs. Die CRLs anderer Operationsfolgen auf X werden in SRLs umgewandelt.

b) Für das Löschen eines Knotens müssen folgende Bedingungen geprüft werden:

i. V entspricht dem V der letzten Leseoperation aus F.

ii. Die Operationsfolge OF hält eine CRL auf X.

iii. Liegen auf X r MLs, dann müssen auf dem Vaterknoten von X auch r MLs liegen.

iv. Auf X darf keine IL liegen.

v. Ist eine der Bedingungen 3(b)i - 3(b)iii nicht erfüllt, so führt das zum Abbruch der Operationsfolge. Andernfalls erhält OF auf X eine DL und verliert alle CRLs, SRLs und HRLs auf $V \backslash \{\text{Alle Nachkommen von } X\}$. Die CRLs anderer Operationsfolgen auf X werden in HRLs umgewandelt.

vi. Hat X Nachkommen, so müssen für diese auch DLs angefordert werden. Dabei müssen für jeden Knoten die Bedingungen 3(b)i - 3(b)v überprüft werden.

c) Für das Einfügen beliebiger Objekte an einem Knoten X müssen folgende Bedingungen geprüft werden:

i. V entspricht dem V der letzten Leseoperation aus F für reines Einfügen oder Verschieben. V entspricht dem V der letzten Leseoperation aus G für reines Verschieben.

ii. X muss ein XML-Element sein.

iii. OF muss eine CRL auf X besitzen.

iv. Ist eine der Bedingungen 3(c)i - 3(c)iii nicht erfüllt, so führt dies zum Abbruch der Operationsfolge. Andernfalls erhält OF auf X eine IL und verliert alle CRLs, SRLs und HRLs.

d) Für das Verschieben eines Knotens X müssen folgende Bedingungen geprüft werden:

i. V entspricht dem V der letzten Leseoperation aus F.

ii. Ist X der erste Knoten, auf den OF eine ML setzten möchte, so muss OF mindestens eine SRL auf X halten.

iii. Hat X r MLs, so muss auch der Vorgänger von X r MLs besitzen. Wobei nur die MLs der von OF verschiedenen Operationsfolgen gezählt werden.

iv. Auf X darf keine RRL liegen.

v. Ist eine der Bedingungen 3(d)i - 3(d)iv nicht erfüllt, so führt das zum Abbruch der Operationsfolge. Andernfalls erhält OF auf X eine ML. Die CRLs oder HRLs auf X von den Operationen, die auch eine CRL auf den Vorgängerknoten des ersten Knotens, auf dem OF eine ML hält, besitzen, werden in SRLs abgeschwächt.

vi. Hat X Nachfolgerknoten, so müssen auch für diese MLs angefordert werden. Dabei müssen für jeden Knoten die Bedingungen 3(d)iii - 3(d)v erfüllt sein.

vii. Die Einfügesperre wird gemäß Regel 3c vergeben.

e) Für das Reset oder Repeat eines Knotens X müssen folgende Bedingungen geprüft werden.

i. V entspricht dem V der letzten Leseoperation aus F.
ii. OF muss mindestens eine HRL auf X halten.
iii. Auf X darf keine RRL einer anderen Operationsfolge liegen.
iv. Ist eine der Bedingungen 3(e)i oder 3(e)iii nicht erfüllt, führt dies zum Abbruch der Operationsfolge. Andernfalls erhält OF eine RRL auf X. Dabei verliert OF alle CRLs, SRLs und HRLs auf $V \setminus \{$Nachfolgeknoten von X, auf die ein Reset oder Repeat ausgeführt werden soll$\}$. Die CRLs und SRLs anderer Operationsfolgen auf X werden in HRLs abgeschwächt. Laufende Änderungsoperationen anderer Operationsfolgen auf X werden abgebrochen.
v. Soll auf Nachkommen von X ein Reset oder Repeat ausgeführt werden, so müssen für diese Knoten gemäß Bedingung 3(e)iii ebenfalls RRLs angefordert werden. Wurden die Sperren erlangt, verliert OF auch für diese Knoten die CRLs, SRLs und HRLs.

Für die Anforderung und Freigabe der Sperren müssen nun noch die Zeitpunkte und die Ebenen im Modell bestimmt werden.

- Bis zum Ende einer Subaktion der ersten Ebene sind alle Sperren freigegeben.
- Besteht die Subaktion nur aus einer Ebene, so wird die entsprechende Lesesperre zu Beginn der Subaktion angefordert und mit dem *Complete* freigegeben.
- Besteht die Subaktion nur aus zwei Ebenen, so wird mit Beginn der Subaktion der zweiten Ebene die Sperre angefordert und mit dem *Complete* der Subaktion erster Ebene freigegeben. Dies lässt sich wie folgt begründen. Die erste Operation innerhalb einer Subaktion mit dem gegebenen Aufbau ist das Lesen exakt eines Knotens. Dafür wird zu Beginn der SubSubaktion die Lesesperre angefordert. Nach dem *Complete* dieser Aktion geht die Sperre an die Subaktion und kann durch eine Änderungsaktion innerhalb einer zweiten SubSubaktion verschärft werden. Nach dem Ende dieser SubSubaktion geht die Sperre wieder an die Subaktion erster Ebene zurück und wird mit deren *Complete* freigegeben.
- Besteht die Subaktion aus drei Ebenen, so werden alle in der untersten Ebene benötigten Sperren zu Beginn der einhüllenden Subaktion der zweiten Ebene atomar angefordert und an die Subaktionen der dritten Ebene weitergegeben. Mit dem *Complete* dieser SubSubSubaktionen gehen die Sperren an die SubSubaktion und mit deren *Complete* an die Subaktion. Diese gibt dann die zur Verschärfung benötigten Sperren an die nächste Subaktion der zweiten Ebene.(Diese fordert die Sperren atomar an.) Alle nicht mehr benötigten Sperren werden sofort freigegeben. Mit dem *Complete* der Subaktion der ersten Ebene werden alle Sperren freigegeben.

Gegenstand dieses Abschnitts war die Erörterung des Sperrmodells. Dabei wurden zunächst die einzelnen Operationen auf ihre Verträglichkeit untereinander untersucht. Daraus ergaben sich die benötigten Sperrtypen. Am Ende dieses Abschnitts wurde ein Protokoll entworfen, welches die korrekte nebenläufige Ausführung von Aktionen gewährleistet.

4.4 Das Fehlerbehandlungsmodell

Innerhalb dieses Abschnitts soll das Fehlerbehandlungsmodell vorgestellt und beschrieben werden. Es handelt sich bei diesem Modell um ein Objektversionensystem in Kombination mit

einem Log-Buch.

4.4.1 Das Objektversionensystem

Ein Objektversionensystem ist sehr leicht zu erklären. Jede Operation auf einem Objekt erzeugt eine neue Version dieses Objektes. Grafik 4.3 veranschaulicht dies an einem Beispiel. Das Ob-

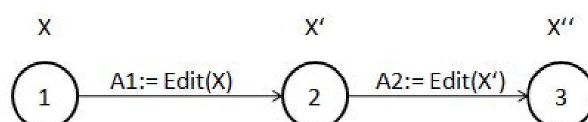

Abbildung 4.3: Ein Beispiel für ein Objektversionensystem

jekt X stellt einen initialen XML-Textknoten dar, der die Zahl *1* enthält. Die Aktion *A1* erzeugt durch das Editieren dieses Knotens eine neue Kopie (Version) X', die nun die Zahl *2* enthält. Ebenso verhält es sich mit der Aktion *A2*, die aus der Objektversion X' die Version X'' erzeugt. Was sich ergibt, ist eine sequenzielle Historie ohne Verzweigungen. Zu einem Zeitpunkt gibt es immer nur genau eine Objektversion. Dies ist ein wichtiger Unterschied zu einem Objektversionensystem, das die Serialisierbarkeit gewährleisten sollen, wie das bei der Multiversionen-Synchronisation [SHK05] der Fall ist. In diesem System kommt es nämlich genau zu solchen Verzweigungen, da die Aktionen oder Transaktionen die zur Erhaltung der Serialisierbarkeit benötigte Objektversion wählen können. Folgendes Beispiel soll dies verdeutlichen:

$$Schedule = Read_1(X), Edit_1(X), Read_2(X), Edit_2(X), Read_1(X) \tag{4.8}$$

Gleichung 4.8 zeigt einen nicht serialisierbaren Schedule. (Die Indizes an den Operationen bezeichnen die Aktionen, zu denen die Operationen gehören.) Durch den Einsatz eines Objektversionensystems, in dem Verzweigungen erlaubt sind, kann die Serialisierbarkeit jedoch wiederhergestellt werden. Dies geschieht dadurch, dass die letzte Operation $Read_1(X)$ auf die von $Edit_1(X)$ erzeugte Objektversion zugreift. Das große Problem dabei ist jedoch, dass durch diese Sicherstellung der Serialisierbarkeit jegliche Form der Kooperation unterbunden wird. Die Aktion *1* bekommt von der Änderung der Aktion *2* nichts mit. Es wird also nur ein unidirektionaler Informationsfluss ermöglicht. Aber die Kooperation ist die wichtigste Eigenschaft, die durch das Modell gewährleistet werden soll. Die Serialisierbarkeit nimmt eine nebensächliche Rolle ein. Daher ist die rein sequenzielle Variante des Objektversionensystems die bessere Wahl. Sie ermöglicht einen bidirektionalen Informationsfluss, indem sie den Schedule in Gleichung 4.8, unter Inkaufnahme der Nicht-Serialisierbarkeit, zulässt. Auf eintretende Konflikte kann mit Hilfe des Resets und Repeats reagiert werden. Die Vorgehensweise dabei wird weiter unten erläutert.

Die Frage, warum überhaupt ein Objektversionensystem anstatt eines Kompensationsaktionssystems eingesetzt wird, ist leicht zu beantworten. Der Aufwand beim Rücksetzen einer oder mehrerer Aktionen ist bei einem Objektversionensystem wesentlich geringer als bei einem Kompensationsaktionssystem. Das Rücksetzen der infolge von Aktionsabbrüchen ungültig gewordenen Änderungen wird dadurch erreicht, dass einfach die durch die abgebrochenen Aktionen erzeugten Objektversionen gelöscht werden. Bei dem Kompensationsaktionssystem müssten Kompensationsaktionen, die die Änderungen abgebrochener Aktionen rückgängig machen, erst ermittelt und dann nacheinander als Aktionen ausgeführt werden.

4.4.2 Das Log-Buch

Die Historie innerhalb dieses Objektversionensystems wird mit Hilfe eines Log-Buches festgehalten. Es dient der Verwaltung der Objekte und Objektversionen und ermöglicht u.a., dass jede Aktion immer auf die aktuelle Objektversion zugreift. Der genau Aufbau dieses Log-Buches soll im Folgenden hergeleitet werden.

Zunächst muss untersucht werden, welche Informationen zu einer Objektversion festgehalten werden müssen. Dies erfordert eine nähere Betrachtung aller möglicher Operationen, wobei Reset und Repeat zunächst außen vorgelassen werden, da diese Operationen semantisch gesehen lediglich eine Konkatenation aller sonstiger Änderungsoperationen darstellen. Auch das Read wird zunächst nicht näher betrachtet, da mit Hilfe dieser Operation lediglich ein Objektzustand betrachtet, jedoch kein neuer erzeugt werden kann. Interessant sind an dieser Stelle also das Edit, Insert, Delete und Move:

- Edit sorgt für eine rein inhaltliche Veränderung eines Knotens, indem die Operation einem Attribut-/Textknoten einen neuen Wert zuweist.
- Insert führt dazu, dass ein Knoten „erscheint", er also materialisiert wird. Dies führt implizit natürlich neben der strukturellen auch zu einer inhaltlichen Materialisierung. Wobei durchaus gesagt werden kann, dass der Operation der Inhalt eines Knotens gleichgültig ist.
- Delete stellt das Gegenstück zum Insert dar. Es führt zum Verschwinden eines Knotens und damit auch dessen Inhalts. Wobei wiederum festgehalten werden kann, dass dieser Operation der Inhalt des Knotens völlig gleichgültig ist.
- Move führt zur Veränderung des Aufenthaltsortes eines Knotens. Z.B. kann ein Knoten von einem Teilbaum in einen anderen verschoben werden. Dieser Operation ist völlig egal, was innerhalb von ihr passiert, bis auf eine Einschränkung: Es darf kein Löschen oder Verschieben des Wurzelelementes der Verschiebung erfolgen. Eine Bemerkung soll hinsichtlich des Inserts gemacht werden. Das Anfügen der Elemente an ihren neuen Platz wird auf der operationalen Ebene auf das Insert zurückgeführt. Auf semantischer Ebene unterscheidet es sich jedoch von der Operation Insert, da hier keine Materialisierung der Elemente im eigentlichen Sinne stattfindet. Diese existieren bereits. Daher kann das Anfügen innerhalb der Operation Move als reine Zuweisung eines neuen Platzes angesehen werden.

Aus dieser Untersuchung kann Folgendes geschlossen werden. Es existieren drei Dimensionen, die eine Objektversion kennzeichnen:

- Der Inhalt, der durch Edit verändert wird. Er ist das Objekt bzw. die Objektversion im eigentlichen Sinne. Der Inhalt kann aber auch leer sein, so wie es bei einem XML-Element im taDOM-Modell der Fall ist. Allerdings gibt es von diesen Elementen nur eine Version, nämlich das Original. Nur von Attribut-/Textknoten können verschiedene Versionen durch das Edit erzeugt werden.
- Der Ort, der durch Move verändert wird. Er gibt den Platz des Objektes innerhalb der XML-Struktur an und bestimmt es somit näher.
- Die Materialisierung, welche durch Insert und Delete verändert wird. Wobei beim Insert natürlich ein initialer Inhalt und Ort vorgegeben werden muss. Die Materialisierung

bestimmt, ob ein Objekt vorhanden ist oder nicht. Es dient also auch einer näheren Bestimmung des Objektes.

Zwischen den einzelnen Dimensionen ergeben sich gewisse Abhängigkeiten, die aber schon ausreichend bei der Beschreibung des Sperrmodells erläutert worden sind. Beispielsweise kann eine Änderung des Inhalts nur erfolgen, wenn der Knoten materialisiert, sprich eingefügt ist. Bei der Veränderung des Ortes gibt es die Bedingung, dass das Wurzelelement der Verschiebung vorhanden ist und nicht gelöscht wird.

Für jeden Knoten eines XML-Dokumentes wird nun ein Log-Buch geführt, welches die eben beschriebenen Dimensionen enthält. Die Abbildung 4.4 zeigt beispielhaft, wie ein Log-Buch aussehen kann. Diese Grafik zeigt eine Abbildung der drei Dimensionen auf eine Tabelle. Diese

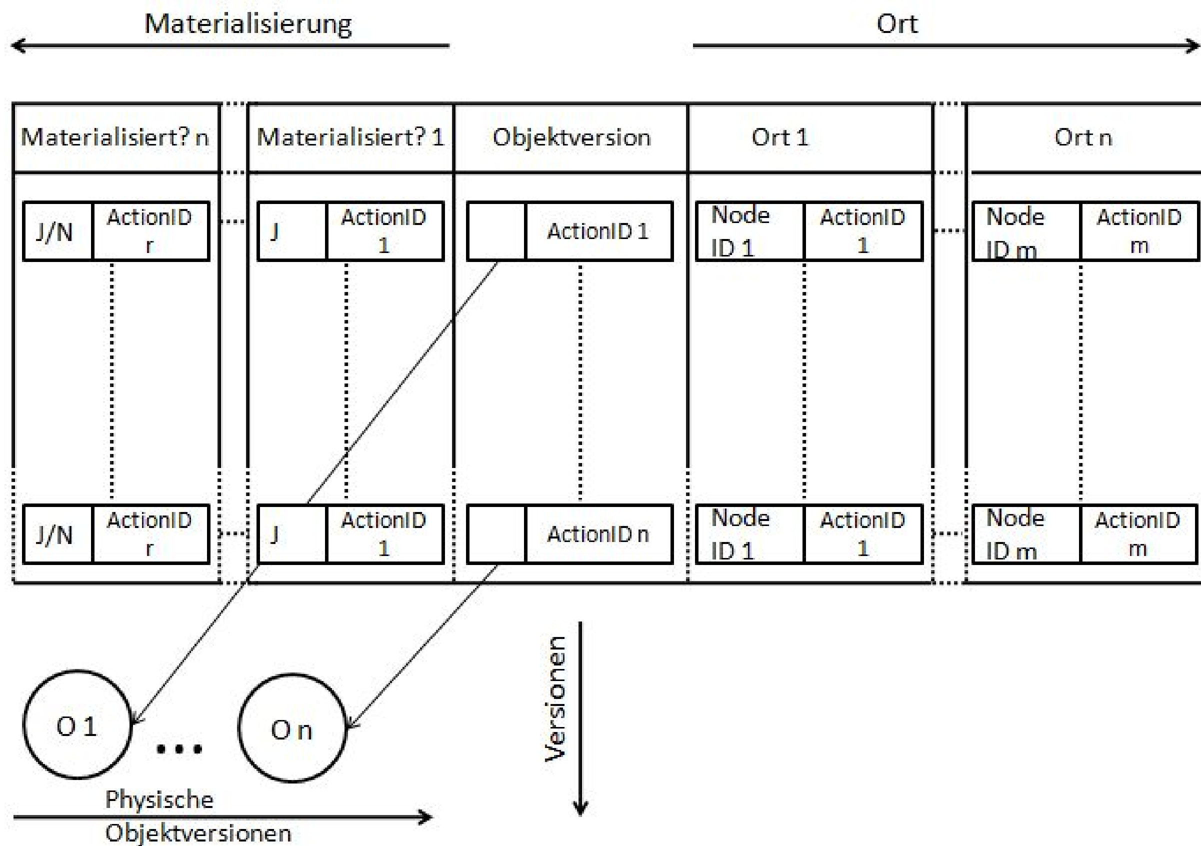

Abbildung 4.4: Beispielhafter Aufbau des Log-Buches

wächst mit jeder Verschiebung nach rechts, mit jeder Erzeugung einer neuen Version durch ein Edit nach unten und mit jeder Änderung der Materialisierung nach links. Warum die Tabelle auch nach links wachsen kann, lässt sich wie folgt erklären. Ein Element kann durch ein Insert materialisiert werden. Durch ein darauf folgendes Delete kann es wieder entfernt werden. Nun könnte mit einem Reset das Delete rückgängig gemacht werden. Anschließend könnte ein Repeat das Reset rückgängig machen usw. Jede dieser Operationen muss festgehalten werden, um eine lückenlose Historie der Eigenschaften der Objektversion zu gewährleisten. Dies gilt für alle drei Dimension. Daher wächst die Tabelle mit jeder Änderungsoperation in die jeweilige Richtung. Weiterhin sind im Bild die Tabelleneinträge zu erkennen. In der Spalte *Objektversion* besteht jeder Eintrag aus einem Zeiger auf die zugehörige Objektversion, die „physisch"

als Datenelement im Speicher vorliegt, und einer *ActionID*, die eine Kennzeichnung der Aktion darstellt, die diese Objektversion erzeugt hat. In der Spalte *Ort* besteht jeder Eintrag aus einer *NodeID*, die den Platz der Objektversion im XML-Dokument eindeutig bestimmt, und der *ActionID*, die die Aktion eindeutig bestimmt, welche die Objektversion an diese Stelle gesetzt hat. Die *NodeID* ist dabei eine Konkatenation einer *DeweyID* (Siehe Abschnitt 2.7) und der *DocumentID* des zugehörigen XML-Dokumentes. Dies ist notwendig, um einen Knoten eindeutig zu identifizieren, da eine *DeweyID* nur innerhalb eines XML-Dokumentes eindeutig ist. (Knoten können auch zwischen XML-Dokumenten verschoben werden.) Die Einträge in der Spalte *Materialisiert?* bestehen aus einem *Bool*-Wert, der angibt, ob die Objektversion materialisiert ist oder nicht, und der *ActionID*, die die Aktion eindeutig bestimmt, welche die Objektversion in diesen Zustand versetzt hat.

Es fällt weiterhin auf, dass innerhalb der Spalten des Ortes und der Materialisierung jede Zeile den gleichen Eintrag enthält. Dies hat folgende Ursache. Durch das Verschieben eines Knotens werden auch alle Versionen dieses Knotens an die neue Stelle verschoben. Dadurch ist es z.B. möglich, alte Objektversionen an einer neuen Stelle wiederherzustellen. Beim Delete und Insert ist es so, dass entweder alle Objektversionen materialisiert sind oder keine. Dies ist sinnvoll, da durch das Löschen nicht nur der Inhalt eines Elementes gelöscht, sondern auch das Element als Struktur entfernt wird. Ein Element, welches keine Struktur hat, sollte auch keinen Inhalt haben. Deshalb werden alle Objektversionen als gelöscht markiert. Ein Insert kehrt ein Delete um, wodurch alle Objektversionen wieder verfügbar gemacht werden. Das Insert kann einmal das initiale Insert sein, was ein Objekt erst erstellt. Es kann aber auch einfach als semantische Kompensationsaktion innerhalb eines Resets oder Repeats aufgefasst werden.

Ein letzte Besonderheit, die in der Grafik auffällt, betrifft die Spalte *Materialisiert? 1*, die Spalte *Ort 1* und die erste Zeile in der Spalte *Objektversion*. Ein Log-Buch wird bekanntlich für jeden Knoten angelegt. Es bleibt dabei noch zu klären, wann und wie dies geschehen soll. Es müssen dabei zwei Fälle unterschieden werden:

1. Fall: Der Knoten existiert zu Beginn der Arbeiten aller Clients noch nicht und wird erst durch ein Insert erzeugt.
2. Fall: Der Knoten existiert bereits vor dem Beginn der Arbeiten aller Clients.

Im ersten Fall wird das Log-Buch mit dem Insert angelegt. Wie bereits weiter oben erwähnt, muss bei einem Insert natürlich ein initialer Ort und Inhalt vorgegeben werden. Dadurch haben die Einträge in der Spalte *Materialisiert? 1* und *Ort 1* sowie in der ersten Zeile der Spalte *Objektversion* die gleiche *ActionID*. Diese kennzeichnet die Aktion, welche die initiale Insert-Operation beinhaltet. Im zweiten Fall kann das Anlegen auch mit einer Insert-Operation erfolgen, jedoch sollte dabei eine reservierte *ActionID* verwendet werden, die sonst nicht benutzt werden darf. Diese kennzeichnet dann eine fiktive Aktion, die auch nie abgebrochen werden kann. Das Übernehmen der *ActionID* der letzten Insert-Operation für diesen Knoten aus einer anderen Arbeitssitzung ist ungünstig, da es in Abhängigkeit von der jeweiligen Realisierung des Systems zu einer Neuvergabe der *ActionIDs* aus der vorhergehenden Sitzung kommen kann.

Nachdem das Log-Buch für die Verwaltung der Objektversionen aufgebaut wurde, muss dieses nun auch für Read-Operationen erweitert werden. Dazu sollte zunächst untersucht werden, in welchem Fall der Abbruch einer Leseoperation überhaupt erforderlich ist. Wie bereits bekannt, ist eine Subaktion der ersten Ebene die zentrale Einheit zur Ausführung von Operationen. Eine Subaktion kann in zwei unterschiedlichen Ausprägungen vorkommen. Sie kann nur Leseopera-

tionen beinhalten oder Leseoperationen in Verbindung mit einer Änderungsoperation. Im ersten Fall dient die Operationsfolge dem reinen Informationsgewinn. Im zweiten Fall jedoch wird der Informationsgewinn für einen Bearbeitungsschritt genutzt. Nun stellt sich die Frage, ob in beiden Fällen ein Abbruch der Leseoperationen aufgrund des Lesens eines nicht mehr gültigen Objektzustandes erfolgen sollte oder nur im zweiten Fall. Die Beantwortung dieser Frage ist abhängig davon, ob ein kaskadierender Abbruch soweit zurückreicht, dass der Originalknoten verloren geht. (Das heißt, dass die initiale Insert-Operation abgebrochen wurde.) Dann wurde nämlich von einem Knoten gelesen, der gar nicht existiert. Tritt dies ein, so müssen in beiden Fällen die Leseoperationen abgebrochen werden. Bleibt bei einem Abbruch der Originalknoten erhalten, so müssen lediglich im zweiten Fall die Leseoperationen abgebrochen werden. Dies lässt sich wie folgt begründen. Wird eine reine Leseoperationsfolge durchgeführt, so kann nach deren Abschluss eine beliebige Folge von Änderungen durchgeführt werden. D.h., der Bearbeiter kann in diesem Fall nie sicher sein, dass der in diesem Moment gelesene Objektzustand nach Beendigung der reinen Leseoperationsfolge nicht verändert wird. Ob nun diese Veränderung durch einen anderen Bearbeiter vorgenommen oder durch einen Abbruch verursacht wird, ist im Prinzip völlig gleichgültig. Im zweiten Fall ist die Situation etwas anders. Dort hat der Bearbeiter auf Basis eines Objektzustandes einen Bearbeitungsschritt durchgeführt. Diesen hätte er, wäre der Objektzustand ein anderer gewesen, vielleicht nie vorgenommen.

Abbildung 4.5 stellt nun einmal dar, um welche Teile das Log-Buch erweitert werden muss, um eine korrekte Fehlerbehandlung zu ermöglichen. Oben im Bild ist noch einmal der bisherige Teil des Log-Buches zu sehen. Die Tabelle unten rechts im Bild speichert zu jeder *ActionID* aus der oberen Tabelle alle elementaren Leseoperationen, die innerhalb einer Subaktion der Änderungsoperation vorausgegangen sind. Im Falle eines Abbruchs einer Änderungsoperation wird so gewährleistet, dass rückwirkend auch die dazugehörigen Leseoperationen rückgesetzt werden. Somit wird für Operationsfolgen, die eine Änderungsoperation beinhalten, sichergestellt, dass die Leseoperationen rückgesetzt werden, wenn sie einen nicht mehr gültigen Objektzustand gelesen haben. Die Tabelle unten links im Bild enthält alle *ActionIDs* der Leseoperationen, bei denen die Operationsfolgen, zu denen sie gehören, nur aus Leseoperationen bestehen. Wird nun die erste Operation, also ein Insert, in der Tabelle oben im Bild, abgebrochen, so führt dies auch zum Abbruch der Leseoperationen. Dadurch wird sichergestellt, dass Leseoperationen, die keine Änderungsoperationen nach sich ziehen, nur dann rückgesetzt werden, wenn der Knoten, auf den sie sich beziehen, nicht mehr existiert. Hat ein Knoten schon vor Beginn der Arbeiten existiert, dann bedeutet das, dass Nur-Lese-Operationsfolgen nie abgebrochen werden.

4.4.3 Darstellung der Funktionsweise des Log-Buches anhand eines Beispiels

Nachdem im letzten Abschnitt der prinzipielle Aufbau des Log-Buches geklärt wurde, soll die Funktionsweise zunächst allgemein verdeutlicht werden. Wie bereits erwähnt, führt jede Änderungsoperation zu einem neuen Eintrag im Log-Buch. Was passiert aber nun, wenn durch einen Fehler ein Abbruch erfolgt? Dann wird das Log-Buch wie folgt modifiziert:

- Wird ein Edit oder das initiale Insert eines Knotens abgebrochen, so führt das zum Entfernen der Zeile, die den entsprechenden Eintrag enthält und aller darauffolgenden Zeilen.
- Wird eine Löschoperation abgebrochen, so führt dies zum Entfernen der entsprechenden

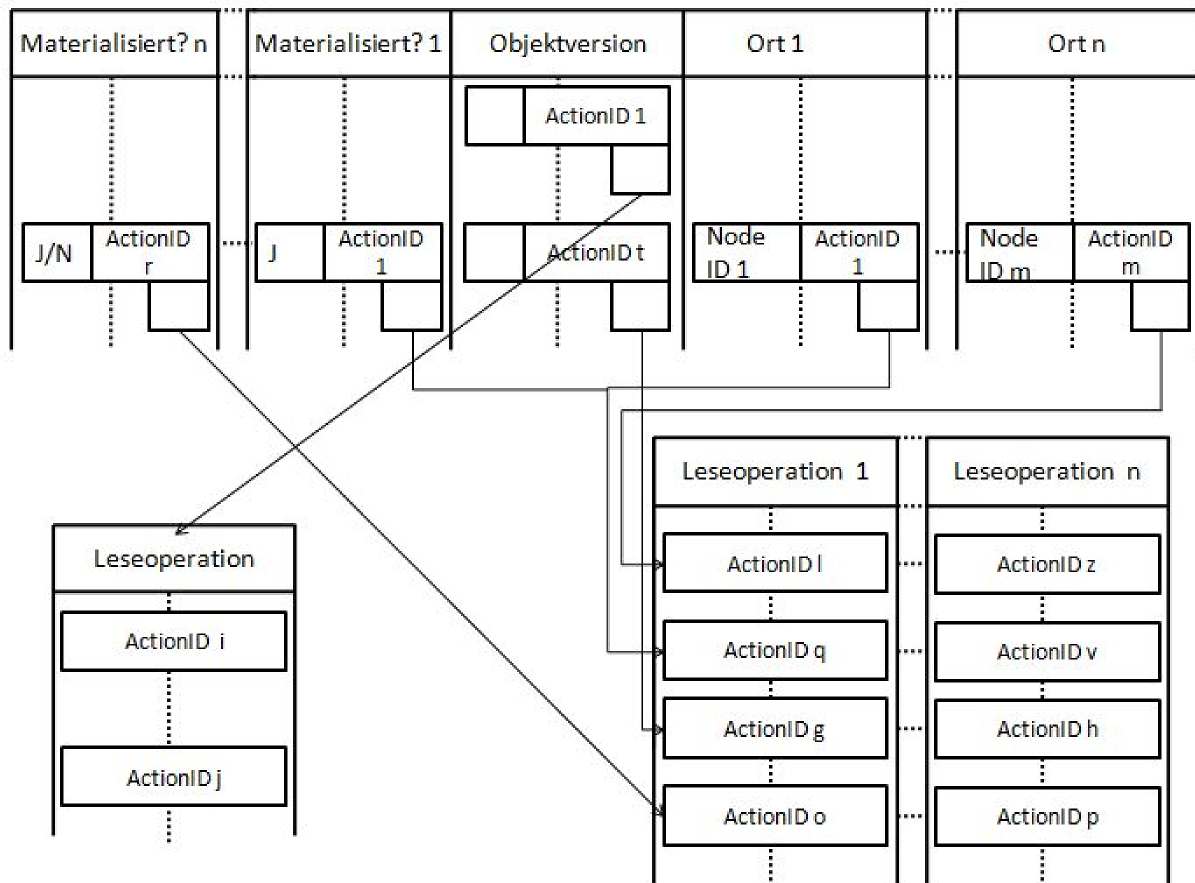

Abbildung 4.5: Erweiterung des Log-Buches zur Unterstützung des Reads

Spalte in der Dimension Materialisierung und aller Spalten links von ihr.

- Wird eine Verschiebung abgebrochen, so wird nur die entsprechende Spalte in der Dimension Ort entfernt. Die Ursache wird später deutlich.
- Muss eine Reset oder Repeat abgebrochen werden, so müssen die Einträge aus den betroffenen Dimensionen entfernt werden. Mit allen darauf folgenden Einträgen wird abhängig von der Dimension verfahren.

Anhand eines Beispiels soll nun der Umgang mit dem Log-Buch näher erläutert werden und auf Besonderheiten bei der Fehlerbehandlung hingewiesen werden. Abbildung 4.6 zeigt zunächst eine Folge von Operationen eines Designers auf einem XML-Teilbaum. Zur Vereinfachung wird davon ausgegangen, dass sich alle Operationen auf ein XML-Dokument beschränken. Der Ort eines Knotens wird daher nur durch seine *DeweyID* bestimmt.

1. Teilbaum *B* befindet sich in einer Verschiebung vom Knoten *A* zum Knoten *X*. Eine mögliche dazugehörige Operationsfolge könnte lauten:
   $OF_1 = Read(A, B, C, D, E, F, G)Read(X)Move((B, C, D, E, F, G), X)$
   Während dieser Verschiebung werden die Operationen 2 - 5 ausgeführt.
2. Der Wert des Attributknotens *E* wird editiert. Eine mögliche dazugehörige Operationsfolge könnte lauten: $OF_2 = Read(E)Edit(E)$

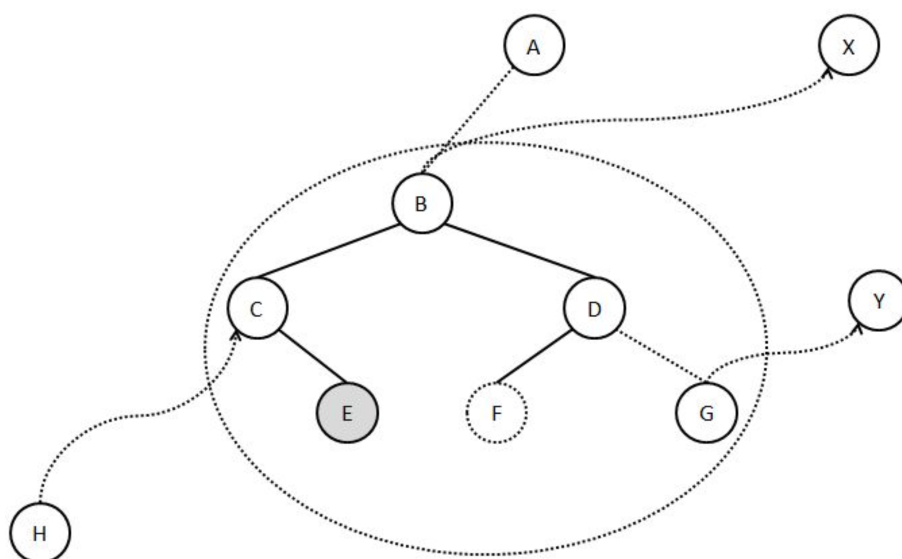

Abbildung 4.6: Folge von Operationen auf einem XML-Teilbaum

3. Ein neuer Knoten *H* wird an das XML-Element *C* eingefügt. Eine mögliche dazugehörige Operationsfolge könnte lauten: $OF_3 = Read(C)Insert(C)$
4. Der Knoten *F* wird gelöscht. Eine mögliche dazugehörige Operationsfolge könnte lauten: $OF_4 = Read(F)Delete(F)$
5. Der Knoten *G* wird verschoben und an ein XML-Element *Y* angehängt. Eine mögliche dazugehörige Operationsfolge könnte lauten: $OF_5 = Read(G)Read(Y)Move(G,Y)$

Alle Operationen auf Teilbäumen werden bekanntlich in elementare Operationen auf einem Knoten zerlegt. Diese werden dann in eine Subaktion eingehüllt. Die Ebene ist dabei variabel. (Z.B. werden die elementaren Move-Operationen von OF_1 in Subaktionen der dritten Ebene gehüllt.) Jede Subaktion, egal welcher Ebene, hat eine eindeutige *ActionID*. Diese wird im Log-Buch vermerkt und zwar immer dann, wenn die Operationsfolge vollständig ausgeführt wurde, d.h. sich die Subaktion der ersten Ebene im Zustand *Completed* befindet, und die Sperren freigegeben wurden. Dadurch wird verhindert, dass Änderungen von laufenden Operationen, die noch abgebrochen werden können, in das Log-Buch gelangen. Die Log-Bücher werden für jeden Knoten einzeln geführt.

Für einige ausgewählte Knoten aus Abbildung 4.6 sollen nun die Log-Bücher vor und nach der Ausführung der obigen Operationen dargestellt werden. Die *ActionIDs* wurden dabei beliebig gewählt. Abbildung 4.7 zeigt das Log-Buch vor den Änderungen am Knoten *E*. Es ist die initiale Insert-Operation mit der *ActionID 10* zu sehen. Dieser Aktion ist eine Leseoperation mit der *ActionID 9* auf dem Vaterknoten von *E*, also *C*, vorangegangen. In Abbildung 4.8 ist das Log-Buch nach der Verschiebung und der Editierung des Knotens *E* (Operationen 1 und 2) zu sehen. Durch das Editieren, gekennzeichnet mit der *ActionID 46*, wurde eine neue Objektversion (Knotenversion) *E 1* des Knotens *E* erzeugt. Für diesen neuen Eintrag im Log-Buch werden die Einträge in der Spalte *Materialisiert? 1* und *Ort 1* übernommen, da das Edit an beiden Dimensionen keine Änderungen vorgenommen hat. Durch die Operation Move, gekennzeichnet durch die *ActionID 42*, auf dem Knoten *E* wird eine neue Spalte *Ort 2* hinzugefügt, die in jeder Zeile den gleichen Eintrag enthält. Dies lässt sich wie folgt begründen: Ein Move verändert weder den Inhalt eines Objektes noch seine Materialisierung. Ihr ist es folglich auch egal, welche

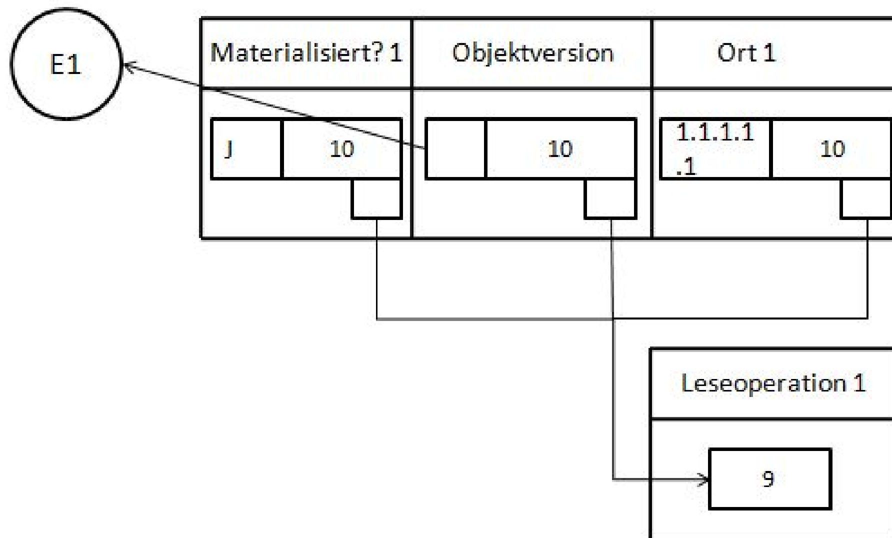

Abbildung 4.7: Log-Buch des Knotens E vor den Änderungen

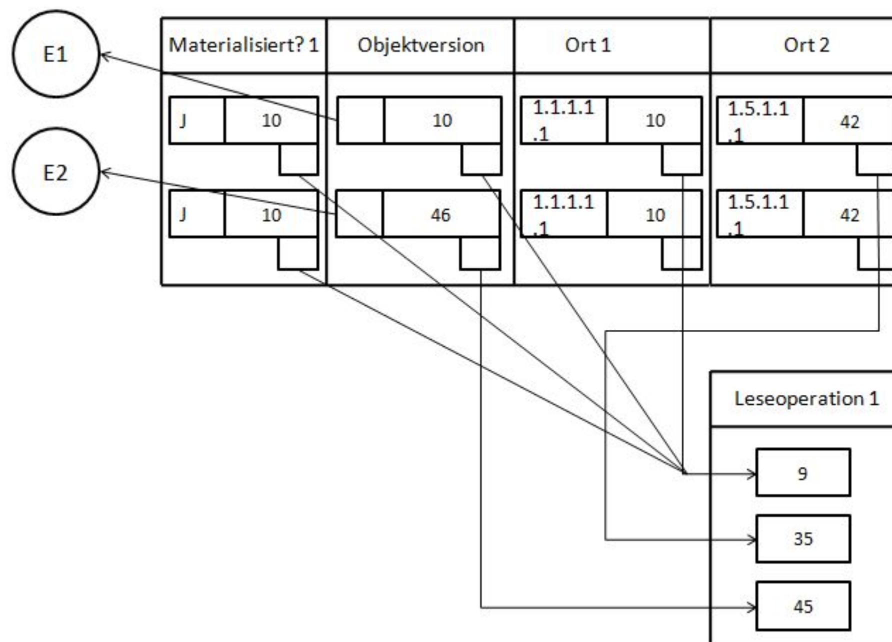

Abbildung 4.8: Log-Buch des Knotens E nach den Änderungen

Objektversion sie verschiebt. Daher muss auch im Falle eines Rücksetzens der Aktion *46* die Objektversion *E 1* an dem neuen Ort verfügbar sein.

Als nächstes soll der Knoten *H* betrachtet werden. Dieser wird während der Verschiebung des Teilbaumes *B* an *C* angefügt. Das Log-Buch für *H* sieht nun wie in Abbildung 4.9 dargestellt aus. Die Aktion, die den Knoten *H* einfügt, trägt die *ActionID 48*. Dem Insert geht eine Leseoperation auf dem Knoten *C* mit der *ActionID 47* voraus. Durch die Verschiebung (Operation 1) während dem Einfügen (Operation 3) tritt ein Spezialfall auf, der durch das Fehlerbehandlungsmodell abgefangen werden muss, da das Aktionsmodell diesen nicht handhaben kann. Zu Beginn der Verschiebung des Teilbaums *B* existierte der Knoten *H* noch nicht. Daher konnte

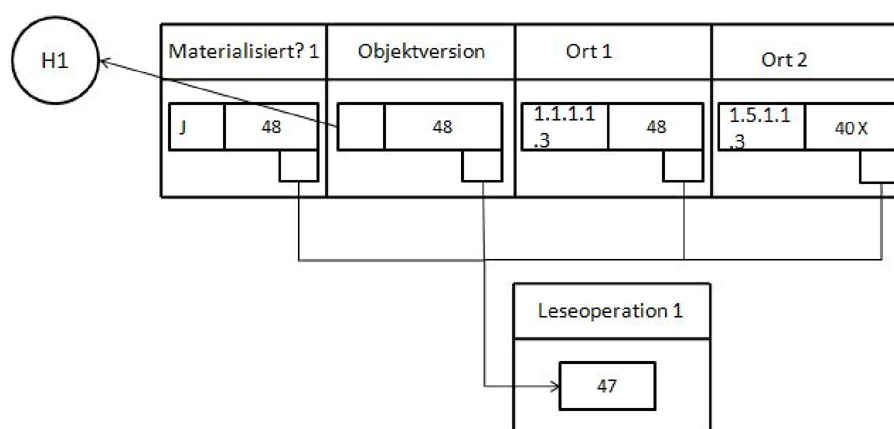

Abbildung 4.9: Log-Buch des Knotens H nach dem Einfügen und Verschieben

auch keine Aktion zu dessen Verschiebung erstellt werden. Trotzdem muss eine Möglichkeit gefunden werden, dass dieser Knoten sowohl eine *DeweyID* für den Platz vor der Verschiebung als auch für den nach der Verschiebung erhält, damit er nach der Verschiebung am neuen und im Falle eines Abbruchs der Verschiebung des Teilbaums *B* auch am alten Platz verfügbar ist. Um dies zu realisieren, wird für die Aktion, die *H* verschiebt, die *ActionID 40X* vergeben. Dies hat folgenden Hintergrund. Die *ActionID 40* kennzeichnet die Aktion, die die Verschiebung des Knotens *C* realisiert. Da für die Verschiebung des Knotens *H* keine Aktion existiert, wird eine virtuelle mit der besagten ID *40X* generiert. Dadurch soll ermöglicht werden, dass im Falle eines Abbruchs der Einfügeoperation *48*, die Aktion *40* nicht mit abgebrochen wird. Wird jedoch Aktion *40* abgebrochen, so soll Aktion *40X* ebenfalls abgebrochen werden. Insgesamt heißt das, wenn das Einfügen von *H* rückgängig gemacht wird, so soll das keine Auswirkungen auf die Verschiebung des Teilbaumes *B* haben. Wird jedoch die Verschiebung des Teilbaumes *B* rückgängig gemacht, so soll auch die Verschiebung von *H* rückgängig gemacht werden. Um diese speziellen Abhängigkeiten zu realisieren, ist eine Verwaltung von Log-Büchern notwendig. Diese wird weiter unten etwas näher betrachtet. Ein kleiner, allgemeiner Hinweis sei an dieser Stelle noch zum Einfügen gegeben. Er bezieht sich auf ein Beispielszenario, in dem ein Knoten R an ein XML-Element T, welches vor dem Einfügen von R an seinen aktuellen Platz verschoben wurde, angehängt werden soll. Auch hier sollte für R nicht nur der aktuelle Ort im Log-Buch festgehalten werden, sondern auch aus allen Orten von T vor dessen Verschiebung die Orte für R generiert werden. Auf diese Weise ist im Falle des Rücksetzens der Verschiebung von T das Rückgängigmachen des Einfügens von R nicht erforderlich, sondern R wird einfach an T, welches sich nun an seinem Platz vor der Verschiebung befindet, angehängt. Dieses Vorgehen lässt sich jedoch wiederum nur durch eine intelligente Log-Buch-Verwaltung realisieren.

Die Abbildung 4.10 zeigt das Log-Buch des Knotens *F* nach dessen Verschiebung und Entfernung (Operationen 1 und 4). *F* wurde zu irgend einem vergangenen Zeitpunkt von einer Aktion mit der Kennung *8* eingefügt. Dieser Aktion ist die Leseoperation mit der Kennung *7* vorangegangen. Durch das Löschen und gleichzeitige Verschieben wird die Tabelle in der Horizontalen in beide Richtungen um eine Spalte erweitert. Dabei realisiert die Aktion mit der Kennung *43* die Verschiebung an den neuen Platz des Knotens und die Aktion mit der Kennung *50* die Änderung des Materialisierungszustandes. Hier ist wieder gut zu erkennen, dass die Operationen Löschen und Verschieben unabhängig voneinander rückgesetzt werden können. Allerdings sind in dem neuen Materialisierungszustand nur noch das implizite Ver-

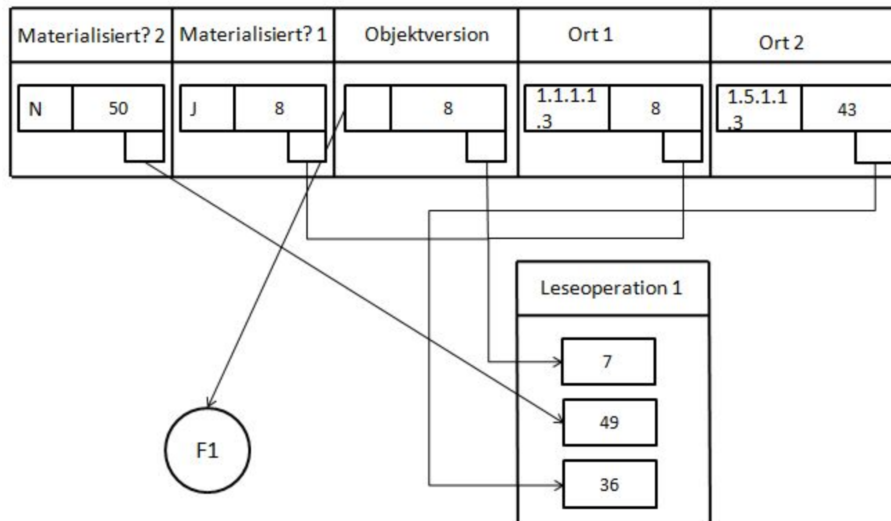

Abbildung 4.10: Log-Buch des Knotens F nach dem Löschen und Verschieben

schieben (indem der Vorgängerknoten *D* verschoben wird) und das Reset möglich. Alle anderen Änderungsoperationen können aufgrund der Abhängigkeiten der Dimensionen, die durch das Sperrmodell realisiert werden, nicht mehr ausgeführt werden.

Zum Schluss soll der Knoten G betrachtet werden. Sein Log-Buch nach den beiden Verschiebungen (Operationen 1 und 5) ist in Abbildung 4.11 dargestellt. Die *ActionID 12* kennzeichnet

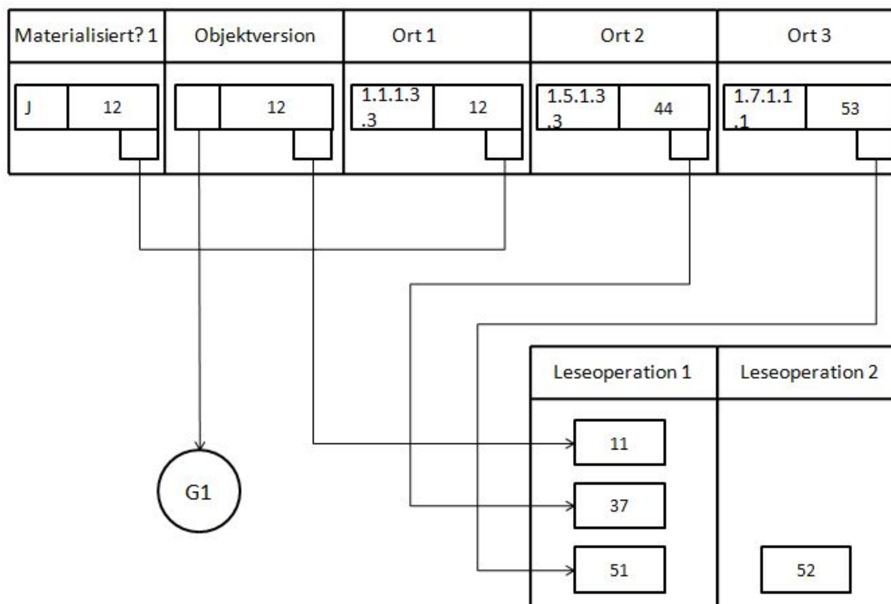

Abbildung 4.11: Log-Buch des Knotens G nach den beiden Verschiebungen

hier wieder die initiale Einfügeoperation des Knotens *G*. Durch die beiden Verschiebeoperationen wird das Log-Buch um zwei Spalten nach rechts erweitert. Die Reihenfolge der Einträge ist festgelegt. Als erstes erfolgt die äußere Verschiebung, gekennzeichnet durch die *ActionID 44*. Danach erfolgt die zweite Verschiebung des Knotens *G* zum Zielknoten *Y*, die durch die *ActionID 53* gekennzeichnet ist. Dadurch wird gewährleistet, dass im Falle des Rücksetzens

der Aktion *53 G* wieder unter *D* erscheint. Eine besondere Situation ergibt sich, falls die Aktion *44* abgebrochen wird. In diesem Fall ist es möglich, auf einen Abbruch der Aktion *53* zu verzichten, da aus semantischer Sicht die verschiedenen Lokalisationen eines Knotens als unabhängig voneinander angesehen werden können. Somit müsste lediglich die Spalte *Ort 2* mit der dazugehörigen Leseoperation entfernt werden. Eine weitere Besonderheit, die in Abbildung 4.11 zu sehen ist, ist die Leseoperation mit der *ActionID 52*. Sie dient dem Lesen des Zielknotens *Y*. Sollte es durch einen Fehler zur Dematerialisierung oder zum endgültigen Entfernen (durch das Abbrechen einer Insert-Operation) des Knotens *Y* kommen, so wird die Verschiebung des Knotens *G* nach *Y* ebenfalls abgebrochen und rückwirkend auch die Leseoperation mit der *ActionID 52*. Der Grund, warum bei den anderen im Beispiel betrachteten Knoten, die ebenfalls einer Verschiebung unterliegen, nicht die Leseoperation des Zielknotens *X* mit in den Log-Büchern verzeichnet wird, ist der folgende: Die betrachteten Knoten *E*, *F* und *H* sind nicht die Wurzelknoten der Verschiebung und daher sollte der Abbruch ihrer Verschiebung nicht automatisch zum Abbruch der Leseoperation auf *X* führen. Nur der Abbruch der Verschiebung von *B* sollte auch zum Rücksetzen der Leseoperation auf *X* führen, da in diesem Fall die ganze Verschiebung abgebrochen werden muss.(Andernfalls würde die Struktur des Dokumentes zerstört.)

Als nächstes soll anhand von Abbildung 4.8 die Funktionsweise der Resets und Repeats bzw. die Führung des Log-Buches für diese beiden Funktionen erläutert werden. Angenommen ein Benutzer möchte die Änderungen der Aktionen *46* und *42* am Knoten *E* rückgängig machen. Dazu ruft er die Funktion Reset auf, die zu der in Abbildung 4.12 dargestellten Modifikation des Log-Buches von *E* führt. Gut zu erkennen ist, dass die Operation Reset (*ActionID 70*) nicht zum

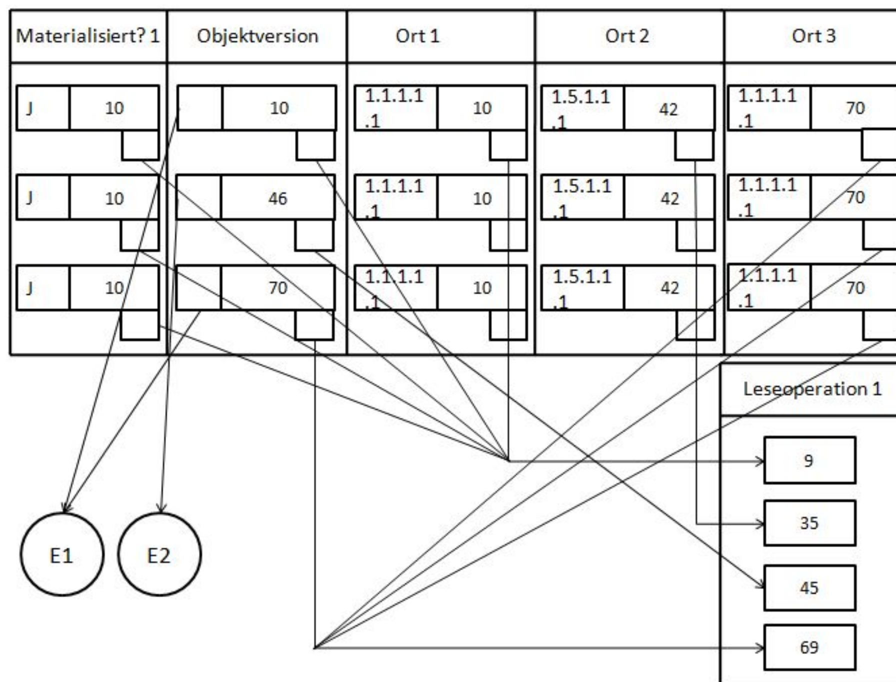

Abbildung 4.12: Log-Buch des Knotens E nach dem Reset der Aktionen 46 und 42

Abbruch der rückgängig zu machenden Operationen führt. Das Reset stellt lediglich eine Konkatenation eines Edits und eines Moves dar und führt zum Erstellen einer neuen Zeile und einer neuen Spalte *Ort 3*. Es werden also mit einem Reset mehrere Änderungen an verschiedenen

Dimensionen in einer Operation ausgeführt. Dadurch, dass im Log-Buch mit Zeigern gearbeitet wird, ist ein physisches Erstellen einer Version *E 3*, die identisch wäre mit der Version *E 1*, nicht erforderlich. Mit einem Repeat könnte nun wiederum das Reset rückgängig gemacht werden, allerdings nur, wenn es die direkte Folgeoperation auf *E* ist. Würde nämlich zwischen Reset und Repeat eine weitere Änderungsoperation durchgeführt, so hätten deren Auswirkungen nach einem Repeat jegliche Bedeutung verloren. Es ist so, als wäre sie nie ausgeführt worden. Dies soll vermieden werden. Tritt eine derartige Konstellation auf, so würde die Repeat-Operation abgebrochen. Das Repeat wird genauso ausgeführt wie das Reset. Es führt wiederum zum Anfügen einer neuen Spalte und einer neuen Zeile. Der Zeiger des Eintrags in der neuen Zeile würde auf die Objektversion *E 2* zeigen, der Eintrag in der neuen Spalte hätte die *DeweyID 1.5.1.1.1*.

4.4.4 Die Verwaltung und Pflege der Log-Bücher

Ein weiterer Aspekt, der betrachtet werden soll, ist die Verwaltung der Log-Bücher. Die Notwendigkeit dieser Verwaltung wurde bereits mehrmals in den vorherigen Abschnitten angedeutet. Auch für das zuletzt erklärte Reset der Operationen auf dem Knoten *E* des Beispiels 4.6 ist eine Verwaltung der Log-Bücher unumgänglich, da das Rückgängigmachen der Verschiebung des Knotens *E* das vollständige Rücksetzen der Verschiebung des gesamten Teilbaums *B* erfordert, da ansonsten die Struktur des Dokumentes nicht mehr erhalten bleibt. Eine Log-Buch-Verwaltung muss in diesem Fall alle Knoten ermitteln, auf denen das Reset ebenfalls ausgeführt werden muss, und die entsprechenden Eintragungen in den dazugehörigen Log-Büchern vornehmen. Hier ist eine Besonderheit für den Knoten *G* zu beachten. Während bei allen anderen Knoten des Teilbaumes *B* die Einträge des Resets bezüglich der Verschiebung durch eine neue Spalte ganz rechts außen realisiert werden, muss bei dem Log-Buch des Knotens *G* die neue Spalte zwischen *Ort 2* und *3* eingefügt werden, da dieser Knoten zum Zeitpunkt des Resets (bei einem darauffolgenden Repeat natürlich auch) nicht mehr Teil des Teilbaumes *B* ist. Dies ist notwendig, um ein korrektes Rücksetzen des Moves des Knotens *G* nach *Y* zu gewährleisten. Auch beim Abbruch einer Löschoperation auf einem Knoten müssen die Löschoperationen auf den Vorgängerknoten mit abgebrochen werden, da ansonsten die Struktur des Dokumentes zerstört wird. Auch in diesem Fall muss die Menge der betroffenen Knoten ermittelt und die notwendigen Eintragungen in den Log-Büchern durchgeführt werden.

Die Realisierung der Log-Buch-Verwaltung kann durch eine Funktionen erfolgen, die unter Berücksichtigung der Struktur des XML-Dokumentes immer die minimale Menge von Knoten bestimmt, für die ein Rücksetzen von Operationen (egal ob durch ein Reset, Repeat oder einen Fehler verursacht) erforderlich ist. Dies ist mit Hilfe der *NodeIDs*, die auf *DeweyIDs* aufbauen, möglich, denn über die *DeweyIDs* kann bekanntlich die Struktur des XML-Dokumentes ermittelt werden. Durch die Bestimmung der minimalen Menge werden nur die notwendigen Operationen rückgesetzt. Wurde z.B. ein Teilbaum, bestehend aus einem XML-Element und zwei Kindern, an ein XML-Element durch ein Insert angefügt und die zuständige Subaktion erster Ebene befindet sich im Zustand *Completed*, so muss bei einem Rücksetzen des Inserts eines Kindes weder das Rücksetzen des Vaters noch das Rücksetzen des Bruders erfolgen. Dadurch wird auch die Anzahl der Folgeabbrüche (kaskadierende Abbrüche) gering gehalten. So werden nur die Folgeoperationen, die Änderungen auf Basis des nun ungültigen Zustand des Kindes, dessen Insert-Operation rückgesetzt wurde, abgebrochen. Die Folgeoperationen auf dem Vater oder dem Bruder müssen nicht abgebrochen werden.

Zum Schluss sollen einige Worte zur Pflege der Log-Bücher folgen. Für die Erzeugung eines Log-Buches existieren zwei Möglichkeiten. Zum einen kann es durch eine fiktive Insert-Operation beim Auschecken eines Knotens vom Server für diesen Knoten angelegt werden. Dies ist ja, wie bereits im Abschnitt 4.4.2 erwähnt, der Fall, wenn der Knoten schon vor Beginn einer Arbeitssitzung existiert hat. Zum anderen kann es beim Erstellen eines Knotens durch ein initiales Insert für diesen Knoten angelegt werden. Dies ist der Fall, wenn der Knoten zu Beginn einer Arbeitssitzung noch nicht existiert hat. Nachdem ein Log-Buch für einen Knoten erzeugt wurde, wächst es kontinuierlich, da jede Operation auf diesem Knoten zu einem oder mehreren neuen Einträgen führt. Das Entfernen von Einträgen tritt nur auf, falls Operationen abgebrochen werden. Kommt es zum Abbruch der initialen Insert-Operation, so führt dies zum Entfernen des gesamten Log-Buches, da der Knoten in dem Fall nie existiert hat. (Dies kann nur der Fall sein, wenn der Knoten zu Beginn der Arbeitssitzung noch nicht existiert hat.) Sind alle Arbeiten der Designer beendet, so befinden sich die dazugehörigen Wurzelaktionen im Zustand *Completed*. Mit einem darauf folgenden *Commit* werden alle zuvor ausgecheckten und bearbeiteten Knoten (inklusive der in der Sitzung neu erzeugten Knoten) in einer endgültigen Version persistent gespeichert. Ist dies geschehen, so werden die Log-Bücher der einzelnen Knoten nicht mehr benötigt, da in einer neuen Arbeitssitzung auch wieder neue Log-Bücher für die Knoten erzeugt werden. Sie können nun entfernt werden.

Tabelle 4.9 gibt noch einmal einen Überblick über den Lebenszyklus eines Log-Buches.

Ereignis	Auswirkung auf das Log-Buch
Initiales Insert	Anlegen des Log-Buches
Jede Folgeoperation	Wachstum des Log-Buches
Abbruch einer Aktion (nicht des initialen Inserts)	Schrumpfen des Log-Buches
Abbruch des initialen Inserts	Entfernen des Log-Buches
Commit aller Aktionen im Log-Buch	Entfernen des Log-Buches

Tabelle 4.9: Der Lebenszyklus des Log-Buches

Zusammenfassend soll für den gesamten Abschnitt Folgendes festgehalten werden. Ausgehend von einer Analyse der Operationen bezüglich der Änderungen, die sie an den Eigenschaften eines Objektes vornehmen, wurde ein dreidimensionales Log-Buch entworfen. Dieses wird für jeden Knoten separat geführt. Danach wurde der Umgang mit dem Log-Buch, sowohl das Hinzufügen von Einträgen als auch das Entfernen von Einträgen im Fehlerfall, zunächst allgemein und dann anhand eines Beispiels erläutert. Abschließend wurde darauf hingewiesen, dass zusätzlich zu den Log-Büchern eine Log-Buch-Verwaltung notwendig ist, die es im Falle eines Resets/Repeats oder eines Abbruchs ermöglicht, die minimale Menge der betroffenen Knoten zu ermitteln und in deren Log-Büchern die notwendigen Änderungen vorzunehmen.

4.5 Die Eigenschaften des Modells

In diesem Abschnitt werden die Eigenschaften des vorgestellten Aktionsmodells und des dazugehörigen Fehlerbehandlungsmodells untersucht. Dabei wird zunächst geschaut, in wieweit das

Modell die ACID-Eigenschaften erfüllt. Anschließend wird nachgewiesen, dass ein Fortschreiben der *Commit-Linie* sichergestellt ist. Danach wird untersucht, wann Objektänderungen für andere Nutzer sichtbar werden. Weiterhin wird untersucht, welcher Grad der Nebenläufigkeit der Operationen durch das Modell erzielt wird. Danach wird der kooperative Charakter des Modells analysiert. Des Weiteren wird nachgewiesen, dass das vorgestellte Aktionsmodell deadlockfrei ist. Zum Schluss folgen einige Eigenschaften, die speziell etwas mit der Fehlerbehandlung des Modells zu tun haben.

4.5.1 Untersuchung des Modells hinsichtlich der ACID-Eigenschaften

Gegenstand dieses Abschnitts ist die Untersuchung, in wieweit das vorgestellte Modell die ACID-Eigenschaften (Siehe Abschnitt 2.2.) erfüllt. Im Folgenden werden die Ergebnisse dieser Untersuchung für die Atomarität, die Konsistenz, die Isolation und die Dauerhaftigkeit getrennt dargestellt.

Atomarität

Die Atomarität hängt vom Aufbau des Modells und von der verwendeten Fehlerbehandlung ab. Das Modell wird nun Ebene für Ebene auf die Gewährleistung der Atomarität untersucht:

- Für die Wurzelaktion kann die Atomarität nur gewährleistet werden, wenn diese Aktion direkt abgebrochen wird. Dies lässt sich wie folgt begründen. In Abschnitt 4.2 wurde für die Wurzelaktion definiert, dass der Abbruch einer Subaktion nie zum Abbruch der Wurzelaktion führt. Wird jedoch die Wurzelaktion abgebrochen, so führt dies zum Abbruch aller Subaktionen. Bricht z.B. ein Nutzer seine Arbeiten ab, so werden alle Operationen, die er bis dahin getätigt hat, zurückgesetzt. Wird jedoch eine ausgeführte Operation als ungültig erklärt, z.B. durch einen kaskadierenden Abbruch, so kann der Nutzer trotzdem weiterarbeiten und die nicht vom Abbruch betroffenen Operationen werden weiter als gültig angesehen. Das Alles-oder-Nichts-Prinzip, welches die Atomarität fordert, wird also nur verwirklicht, wenn es zum direkten Abbruch der Wurzelaktion durch den Nutzer oder einen Systemfehler kommt.
- Für die Subaktionen der ersten Ebene kann die Atomarität nur so lange garantiert werden, bis diese den Zustand *Completed* erreicht haben. Das hat folgende Ursache: Bricht eine Subaktion erster Ebene aus eigener Kraft ab, so führt das zum Rücksetzen aller Subaktion der zweiten Ebene, die innerhalb von ihr durchgeführt wurden. Wird eine Subaktion zweiter Ebene abgebrochen, so führt das zum Abbruch der Subaktion erster Ebene, sofern diese noch nicht den Zustand *Completed* erreicht hat. In beiden Fällen ist die Atomarität garantiert, d.h. es werden entweder alle Operationen durchgeführt oder keine. Hat jedoch die Subaktion erster Ebene den Zustand *Completed* erreicht, so führt ein Abbruch einer Subaktion der zweiten Ebene nicht mehr zum Abbruch der Subaktion erster Ebene. Dadurch bleiben die nicht direkt vom Abbruch betroffenen Operationen gültig. Gleiches gilt im Übrigen auch für die Subaktionen der zweiten Ebene
- Für die Subaktionen der dritten Ebene kann die Atomarität hingegen vollständig garantiert werden, da diese aus genau einer Operation bestehen. Wird also eine Subaktion der

dritten Ebene abgebrochen, egal ob aus eigener Kraft, durch eine Schadensausbreitung oder einen Systemfehler, so wird in jedem Fall die enthaltene Operation rückgesetzt.

Aus diesen Betrachtungen kann folgendes geschlossen werden:

- Für elementare Operationen ist die Atomarität stets gewährleistet.
- Für eine Operation auf einem Teilbaum, als semantische Einheit betrachtet, ist die Atomarität nur gewährleistet, wenn sich die Subaktion erster Ebene noch nicht im Zustand *Completed* befindet. Andernfalls ist es möglich, Teiloperationen innerhalb der semantischen Einheit rückzusetzen. Dadurch wird gewährleistet, dass Operationen auf Teilbäumen zunächst immer vollständig ausgeführt werden. Nach ihrer vollständigen Ausführung ist es jedoch möglich, sie teilweise abzubrechen und somit die Anzahl der dadurch ausgelösten kaskadierenden Abbrüche gering zu halten. Ein entsprechendes Beispiel wurde am Ende des zweiten Absatzes von Abschnitt 4.4.4 beschrieben.
- Fehler-Atomarität kann jedoch in jedem Falle garantiert werden, da abgebrochene Aktionen keine Effekte in der Datenbank hinterlassen. Dies wird durch das vorgeschlagene Fehlerbehandlungsmodell (Abschnitt 4.4) sichergestellt.

Zusammenfassend kann festgehalten werden, dass das vorgestellte Modell die Atomarität stark aufweicht.

Konsistenz

Unter Konsistenz versteht man, wie bereits erwähnt, die Wahrung von Integritätsbedingungen, wie z.B. Schlüssel- und Fremdschlüsselbeziehungen. Derartige Bedingungen wurden nicht definiert und müssen daher auch nicht näher betrachtet werden. Die einzige implizite Integritätsbedingung, die durch das Modell gewährleistet werden muss, ist die Erhaltung der Struktur des XML-Dokumentes. Daher müssen die strukturverändernden Operationen dahingehend geprüft werden, ob sie immer, auch im Falle eines Abbruchs, eine korrekte Struktur des XML-Dokumentes erzeugen bzw. hinterlassen.

- **Insert:** Dieses darf laut Sperrprotokoll nur auf XML-Elemente ausgeführt werden. Für die korrekte Struktur der einzufügenden Knoten ist der Nutzer verantwortlich. (Beim Verschieben garantiert das Sperrprotokoll eine korrekte Struktur der einzufügenden Knoten.) Beim Abbruch eines Inserts (auch bei teilweisem) wird durch die Log-Buch-Verwaltung sichergestellt, dass die Struktur des Dokumentes erhalten bleibt. Dies geschieht zum einen dadurch, dass alle Folgeoperationen mit abgebrochen werden. Zum anderen werden bei einem teilweisen Abbruch einer Insertoperation durch die Log-Buch-Verwaltung alle Knoten ermittelt, für die das Insert mit abgebrochen werden muss.
- **Delete:** Beim Delete auf einem Knoten wird durch das Sperrprotokoll immer geprüft, ob dieser Nachfolger hat (Diese werden dann mit entfernt.), damit niemals einem Knoten der Vater entzogen wird, was zum Verlust der Struktur führen würde. Beim Abbruch eines Deletes (auch bei teilweisem) wird durch die Log-Buch-Verwaltung sichergestellt, dass die Struktur des Dokumentes erhalten bleibt.
- **Move:** Bei dieser Operation wird durch das Sperrprotokoll immer sichergestellt, dass auch alle Nachfolger eines Knotens mit verschoben werden und so keine Knoten ohne Vater zurückbleiben. Beim Abbruch eines Moves (auch bei teilweisem) wird durch die

Log-Buch-Verwaltung sichergestellt, dass die Struktur des Dokumentes erhalten bleibt.

- **Reset und Repeat:** Beide Operationen stellen lediglich eine Konkatenation von Änderungsoperationen dar, die alle die Struktur eines XML-Dokumentes nicht verletzen. Bei der Ausführung und beim Abbruch eines Resets oder Repeats (auch bei teilweisem) wird durch die Log-Buch-Verwaltung sichergestellt, dass die Struktur des Dokumentes erhalten bleibt.

Isolation

Die Isolation von Aktionen vermittelt dem Nutzer, wie bereits in Abschnitt 2.2 erläutert, den Eindruck, als würde er allein auf der Datenbank arbeiten. Die Aufweichung der Isolation im Mehrbenutzerbetrieb kann zu Konsistenzproblemen führen, die entsprechend behandelt werden müssen. Der Grad der Isolation wird durch das Sperrprotokoll bestimmt. Es soll nun untersucht werden, in wieweit das vorgestellte Sperrprotokoll die Isolation von Aktionen im beschriebenen Modell durchsetzt.

Isolation im strengen Sinne wird durch das vorgeschlagene Modell nicht gewährleistet, da Sperren schon vor dem *Commit* der Aktionen für andere Aktionen freigegeben werden. Daher soll hier die Isolation nicht bezüglich des *Commits* sondern des *Completes* einer Aktion betrachtet werden. Diese Art der Isolation erzwingt zumindest eine serialisierbare Ausführung der Aktionen. (Alle Operationen, bis auf die *Commits*, werden seriell ausgeführt.) Nachfolgend wird das Modell Ebene für Ebene durchschritten und auf diese abgeschwächte Form der Isolation untersucht:

- Zwischen verschiedenen Wurzelaktionen existiert keine Isolation. Dies liegt daran, dass die Sperren einer Subaktion erster Ebene nach deren *Complete* nicht an die Wurzelaktion zurückgehen, sondern für alle Subaktionen erster Ebene freigegeben werden. So können auch Subaktionen erster Ebene anderer Benutzer danach Sperren auf die Objekte anfordern. Somit herrscht zwischen Wurzelaktionen keine Serialisierbarkeit. Diese ist aber eine Grundvoraussetzung für die Isolation.
- Zwischen verschiedenen Subaktionen erster Ebene herrscht Isolation, da die Ergebnisse der Operationen, die in einer Subaktion ausgeführt werden, erst am Ende einer Subaktion erster Ebene für andere Subaktionen erster Ebene sichtbar werden. Dies wird dadurch gewährleistet, dass die Sperren von Änderungsoperationen bis zum *Complete* der Subaktion erster Ebene gehalten werden. Lediglich Lesesperren dürfen vorzeitig freigegeben werden.
- Zwischen Subaktionen der zweiten Ebene innerhalb einer Subaktion der ersten Ebene herrscht ebenfalls Isolation. Alle innerhalb einer SubSubaktion benutzten Sperren gehen mit ihrem Übergang in den Zustand *Completed* zunächst an die Subaktion erster Ebene. Diese kann dann die Sperren an eine weitere SubSubaktion neu vergeben bzw. die nicht mehr benötigten Sperren für alle freigeben.
- Zwischen Subaktionen der dritten Ebene innerhalb einer Subaktion der zweiten Ebene herrscht auch Isolation. Dies ist aber dadurch zu begründen, dass diese Aktionen sich alle auf unterschiedliche Objekte beziehen. Diese Aktionen stellen bekannterweise immer eine Zerlegung einer Operation auf einem Teilbaum in elementare Operationen auf einzelnen Knoten dieses Teilbaums dar. Dadurch bezieht sich jede SubSubSubaktion innerhalb

einer SubSubaktion auf einen unterschiedlichen Knoten.

- Zwischen Subaktionen der dritten Ebene und der einhüllenden Subaktion der zweiten Ebene herrscht auch Isolation, da beim Vorhandensein von SubSubSubaktionen innerhalb der SubSubaktion keine Operationen ausgeführt werden. Gleiches gilt auch für die Beziehung zwischen SubSubaktionen und der einhüllenden Subaktion.

Zusammenfassend ist also zu sagen, dass erst ab den Subaktionen der ersten Ebene Isolation im abgeschwächten Sinne herrscht. Zwischen Wurzelaktionen ist die Isolation vollständig aufgeweicht.

Dauerhaftigkeit

Die Dauerhaftigkeit besagt, wie bereits in Abschnitt 2.2 beschrieben, dass die Änderungen einer Aktion mit deren *Commit* persistent in der Datenbank festgehalten werden sollen. Diese Eigenschaft wird durch das vorgestellte Modell garantiert. Alle Änderungen, die ein Benutzer an den Objekten getätigt hat, werden mit dem *Commit* der zugehörigen Wurzelaktion in die zentrale Datenbasis eingelagert. Bis zu diesem *Commit* werden die Änderungen, wie in Abschnitt 4.4 beschrieben, durch das Log-Buch verwaltet. Die Dauerhaftigkeit wird durch das vorgestellte Modell auch nicht verletzt, da es auf den dynamischen Aktionen aufbaut. Diese garantieren, durch die Gewährleistung der *Commit-Korrektheit* von dynamischen Aktionen, dass persistent gespeicherte Änderungen nie mehr rückgesetzt werden müssen. Für nähere Informationen sei auf Abschnitt 3.3 verwiesen, in dem eine detaillierte Behandlung der dynamischen Aktionen stattgefunden hat.

4.5.2 Die Fortschreibbarkeit der Commit-Linie

Gegenstand dieses Abschnitts ist es, darzustellen, wie die Fortschreibbarkeit der *Commit-Linie* für das vorgeschlagene Modell gewährleistet werden kann. Die Grundlagen dafür wurden bereits in Abschnitt 3.3 und 3.4 vorgestellt und sind ausführlich in [Moc95] enthalten.

Zunächst soll folgende Einschränkung gemacht werden. Systemausfälle, die nach dem *Complete* einer Aktion noch zu deren Abbruch führen können, werden hier nicht betrachtet. Dies liegt einerseits daran, dass ihr Eintreten unvorhersehbar ist und andererseits ihre Behandlung eine Recovery-Komponente erfordert, deren Entwicklung nicht Gegenstand dieser Arbeit ist. Aktionen können nach dem *Complete* also nur noch durch kaskadierende Abbrüche bzw. durch den Abbruch einer Vateraktion abgebrochen werden. Abbrüche aus eigener Kraft erfolgen immer vor dem Übergang der Aktion in den Zustand *Completed*.

Wie schon bei den geschachtelten dynamischen Aktionen in Abschnitt 3.4 erläutert, besteht auch bei diesem Aktionsmodell das *Commit*-Problem. Eine Vateraktion kann erst in den Zustand *Committed* übergehen, wenn sichergestellt ist, dass alle Söhne, die nicht abgebrochen wurden, auch tatsächlich ein *Commit* ausführen. Andererseits darf ein Sohn erst in den Zustand *Committed* übergehen, wenn garantiert ist, dass die Vateraktion nicht mehr abbricht. Daher wechseln auch in diesem Modell Väter und Söhne gleichzeitig in den Zustand *Committed*. Allerdings muss die Menge der gleichzeitig in den Zustand *Committed* wechselnden Aktionen auf mehrere Aktionsbäume ausgeweitet werden, da zwischen den Blattaktionen verschiedener Wurzelaktionen Kommunikation stattfindet und sich dadurch auch Abhängigkeiten zwischen

den Blättern verschiedener Aktionsbäume ergeben. Abbildung 4.13 veranschaulicht dieses Problem. In dieser Grafik sind drei Aktionsbäume (*A1*, *A4* und *A6*) zu sehen. Die Kanten zwischen

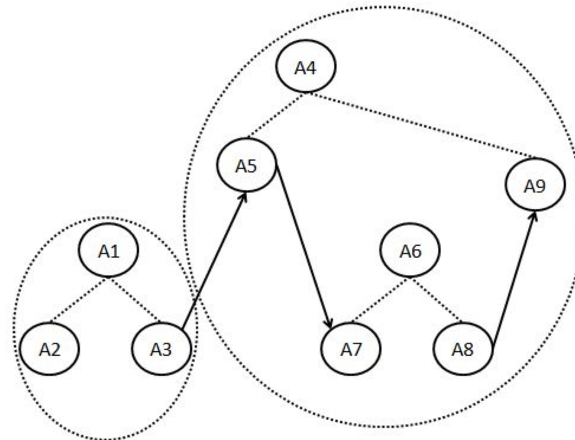

Abbildung 4.13: Beispiel für einen Abhängigkeitsgraphen

den einzelnen Aktionen wurden deshalb gestrichelt gezeichnet, da zwischen den Blattaktionen und der Wurzelaktionen noch weitere Aktionen existieren können. Diese können jedoch für die nachfolgenden Betrachtungen in einer Wurzelaktion zusammengefasst werden, da in dem vorgeschlagenen Modell nur Blätter Operationen ausführen und Vorfahren keine Abhängigkeiten zu anderen Aktionen aufbauen. Für alle Vorfahren gilt also nur die eingangs angesprochene *Vater-Sohn-Commit-Beziehung*. Die Bögen zwischen den Aktionen geben an, welche Aktion von welcher gelesen hat. Beispielsweise hat *A5* ein Ergebnis von *A3* gelesen. Dadurch, dass *A5* von *A3* gelesen hat, macht sich *A5* von *A3* abhängig und muss daher abgebrochen werden, wenn *A3* abgebrochen wird. (Kaskadierender Abbruch) Da jedoch für das Eintreten eines *Commit*-Ereignisses für eine Aktion gefordert wird, dass sie nicht mehr infolge eines kaskadierenden Abbruchs rückgängig gemacht werden muss, darf *A5* erst in den Zustand *Committed* übergehen, wenn *A3* sich in diesem Zustand befindet. In Abbildung 4.14 ist dargestellt, welche Aktion auf das *Commit* einer anderen warten muss, um selbst in den Zustand *Committed* übergehen zu können. In dieser Abbildung sind folgende Aspekte gut zu erkennen:

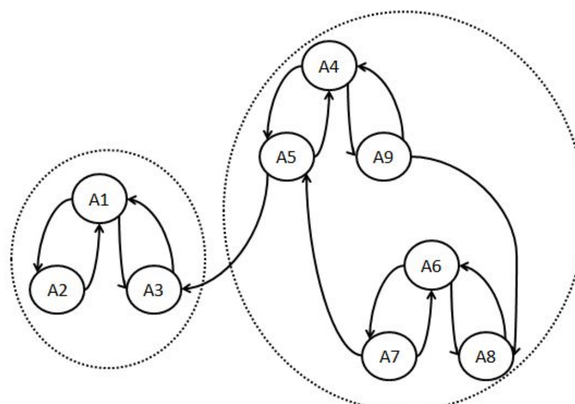

Abbildung 4.14: Der Wartegraph zu Abbildung 4.13

- Eine Vateraktion kann nicht ohne seine Sohnaktionen in den Zustand *Committed* übergehen und umgekehrt.
- Zwischen den Aktionsbäumen *A4* und *A6* kommt es zu einer zyklischen Abhängigkeit. Keiner der Aktionen der beiden Aktionsbäume *A4* und *A6* kann in den Zustand *Committed* übergehen. Daher ist ein *Gruppen-Commit* erforderlich. Das bedeutet, alle Aktionen in der Gruppe wechseln gleichzeitig in den Zustand *Committed*.
- Der Aktionsbaum *A1* bildet eine eigene *Commit-Gruppe*. Hier müssen nur Väter und Söhne gemeinsam in den Zustand *Committed* übergehen. *A1* ist nicht von einer Aktion der anderen beiden Aktionsbäume abhängig.

Nachdem das Problem geschildert wurde, soll untersucht werden, wann ein *Gruppen-Commit* möglich ist. Werden Systemausfälle nicht berücksichtigt, so können folgende Punkte festgelegt werden, die erfüllt sein müssen, damit ein *Gruppen-Commit* erfolgen kann:

- Alle Aktionen innerhalb der *Commit-Gruppe* dürfen nur noch von anderen Aktionen der Gruppe oder von sich bereits im Zustand *Committed* befindenden Aktionen abhängen. Im Beispiel bedeutet das, dass der Aktionsbaum *A1* als erster in den Zustand *Committed* übergehen muss, damit ein *Commit* der anderen beiden Aktionsbäume überhaupt möglich ist.
- Alle Aktionen innerhalb der *Commit-Gruppe* müssen sich im Zustand *Completed* befinden. Ist dies nicht der Fall, so kann es zum einen passieren, dass diese aktiven Aktionen weitere Abhängigkeiten zu gruppenfremden Aktionen aufbauen, wodurch zyklische Abhängigkeiten zu anderen Aktionsbäumen entstehen können. In diesem Fall muss die Gruppe um diese Aktionsbäume erweitert werden. Zum anderen können aktive Aktionen immer noch abbrechen und die Abbrüche anderer Aktionen nach sich ziehen.

Befinden sich also alle Aktionen vom Aktionsbaum *A1* im Zustand *Committed* und sind alle Aktionen der beiden Aktionsbäume *A4* und *A6* im Zustand *Completed*, so kann ein *Gruppen-Commit* dieser beiden Aktionsbäume stattfinden, da keine der Aktionen mehr abgebrochen werden muss. Dies lässt sich dadurch begründen, dass eine sich im Zustand *Completed* befindende Aktion nur noch durch einen kaskadierenden Abbruch oder durch einen Systemausfall abgebrochen werden kann, jedoch nicht mehr aus eigener Kraft abbrechen darf. Da ein Systemausfall vernachlässigt wird und ein kaskadierender Abbruch nicht mehr möglich ist, da alle Aktionen in der Gruppe ein *Commit* durchführen wollen bzw. von sich bereits im Zustand *Committed* befindenden Aktionen abhängen, kann das *Gruppen-Commit* garantiert stattfinden und ein Fortschreiben der *Commit-Linie* ist somit sichergestellt.

Für das vorgeschlagene Modell bedeutet das, dass frühstens ein *Commit* stattfinden kann, wenn ein Designer seine Arbeiten beendet und die dazugehörige Wurzelaktion in den Zustand *Completed* übergeht. Dann müssen sich auch per Definition des Aufbaus des Modells alle Subaktion, die nicht abgebrochen wurden, im Zustand *Completed* befinden. Somit sind die Abhängigkeiten, die der Designer während seiner Arbeiten aufgebaut hat, endgültig. Wenn dies Aktionen, zu denen der Designer Abhängigkeiten aufgebaut hat, in den Zustand *Committed* übergehen, so steht einem *Gruppen-Commit* des kompletten Aktionsbaumes des Designers nichts mehr im Wege. Ein *Commit* findet spätestens, aber dann garantiert, statt, wenn alle Designer ihre Arbeiten abgeschlossen haben. Dieser Fall tritt ein, wenn sich infolge der Kooperation ein Wartegraph konstruieren lässt, in dem sich eine zyklische Abhängigkeit über alle Aktionsbäume aller Designer erstreckt. Dann gehören sie alle zu einer *Commit-Gruppe* und hängen auch nicht mehr

von noch laufenden (oder gruppenfremden) Aktionen ab. Somit können sie gemeinsam bzw. gleichzeitig in den Zustand *Committed* übergehen.

Wichtig ist, dass eine Ermittlung der Abhängigkeiten stattfinden kann, denn nur so können die Aktionen, die gemeinsam in den Zustand *Committed* übergehen müssen, gefunden werden. Dies wird durch das in Abschnitt 4.4 vorgestellte Fehlerbehandlungsmodell ermöglicht. Mit Hilfe des dort vorgeschlagenen Log-Buches kann genau festgestellt werden, welche Aktionen noch von einem kaskadierenden Abbruch betroffen sein könnten. Betrachtet man beispielsweise die Dimension Inhalt, die durch die Ausführung von Edit-Operationen wächst, so können dort alle Aktionen bestimmt werden, die von einer durch die Aktion X erstellten Objektversion gelesen bzw. diese weiterbearbeitet haben. (Die Aktion X ist zwar im Zustand *Completed*, denn sonst wäre ihr Ergebnis nicht im Log-Buch, sie kann aber noch durch den Abbruch der Vateraktion rückgesetzt werden.) Weiterhin können über die Log-Buch-Verwaltung auch alle Aktionen ermittelt werden, die infolge des Abbruchs einer Aktion rückgesetzt werden müssen, damit die Struktur des XML-Dokumentes erhalten bleibt. Wird beispielsweise das Löschen eines Blattknotens rückgängig gemacht, so müssen auch die Löschoperationen auf den Vorfahren dieses Knotens rückgängig gemacht werden. Zusammenfassend kann also gesagt werden, dass das Fehlerbehandlungsmodell die Ermittlung jeglicher Abhängigkeiten zwischen Aktionen ermöglicht und somit die Gruppen von Aktionen ermittelt werden können, die gleichzeitig in den Zustand *Committed* übergehen müssen.

Zusammenfassend kann also gesagt werden, dass das Fortschreiben der *Commit-Linie* in jedem Fall gesichert ist, wenn Abbrüche in Folge von Systemausfällen nicht betrachtet werden.

4.5.3 Die Sichtbarkeit von Objektänderungen

Eine der Hauptanforderungen an das zu entwerfende Modell war, dass die Änderungen eines Bearbeiters so früh wie möglich für andere Bearbeiter zugänglich bzw. sichtbar gemacht werden. Diese Anforderung kann das System im vollen Umfang erfüllen. Der Aufbau des Modells ist so konzipiert, dass jede Subaktion der ersten Ebene höchstens eine Änderungsoperation auf einem einzelnen Knoten oder einem Teilbaum enthält. Durch die offene Schachtelung der Wurzelaktionen werden die Ergebnisse einer Subaktion sofort nach deren Übergang in den Zustand *Completed* für andere Nutzer sichtbar. Ein noch früherer Zeitpunkt der Sichtbarmachung von Ergebnissen wäre nur möglich, wenn die atomare Ausführung von Änderungsoperationen vernachlässigt würde. Dann könnten z.B. schon Änderungen an Knoten, deren Bearbeitung innerhalb einer Änderungsoperation auf einem Teilbaum abgeschlossen ist, während der Ausführung der Änderungsoperation auf den übrigen Knoten des Teilbaums gesehen werden. Dabei besteht jedoch ein erhöhtes Risiko vieler kaskadierender Abbrüche (von den Aktionen, die die Ergebnisse der abgebrochenen Aktionen gelesen haben), da ja bekanntlich eine Subaktion erster Ebene komplett zurückgesetzt wird, falls eine Subaktion tieferer Ebene abbricht und sich die Subaktion erster Ebene noch nicht im Zustand *Completed* befindet.

4.5.4 Der Grad der Nebenläufigkeit

Das vorgestellte Modell, insbesondere das damit verbundene Sperrprotokoll, ermöglicht einen sehr hohen Grad der Nebenläufigkeit. Dies ist zum einen vom Aufbau des Modells abhängig.

Eine Subaktion erster Ebene, als die wichtigste Einheit im Modell, realisiert immer eine Operationsfolge gemäß den Gleichungen 4.5 und 4.6. Nach vollständiger Ausführung (oder einem Abbruch) einer dieser Operationsfolgen auf einer Knotenmenge werden die Sperren für diese Knotenmenge wieder frei gegeben und stehen anderen Benutzern mit ihren Operationsfolgen zur Verfügung. Für die Operationsfolgen lässt sich eine maximale Anzahl von Operationen angeben:

- Es sei F die erste Leseoperationsfolge gemäß den Gleichungen 4.5 und 4.6.
- Es sei G die zweite Leseoperationsfolge gemäß der Gleichung 4.6.
- V ist die initiale zusammenhängende Knotenmenge der ersten Leseoperation von F.
- W ist die initiale zusammenhängende Knotenmenge der ersten Leseoperation von G.
- Im Worst-Case-Fall liegen auf V $|V|$ verschiedene Move-Operationen von $|V|$ verschiedenen Operationsfolgen. Auf einem Baum A,B,C mit A als Wurzel können z.B. 3 verschiedene Move-Operationen liegen. Die erste gilt dem Baum mit der Wurzel A, die zweite betrifft den Teilbaum mit der Wurzel B und die dritte das Blatt C. Das Gleiche gilt im Übrigen auch für W. Auf dieser Menge können $|W|$ verschiedene Move-Operationen von $|W|$ verschiedenen Operationsfolgen liegen.
- Somit besteht im Worst-Case-Fall eine Operationsfolge für Gleichung 4.5 aus $|V|$ Leseoperationen und einer Änderungsoperation.
- Für Gleichung 4.6 gilt im Worst-Case-Fall für die Gesamtzahl der Operationen: $|V| + |W| + 1$.

Diese Obergrenze der Anzahl von Operationen in einer Operationsfolge kann als Maß genutzt werden, um den Zeitpunkt zu bestimmen, an dem ein Knoten oder eine Knotenmenge spätestens wieder benutzt werden kann. Hinzu kommt noch, dass sich bei jeder Folgeleseoperation die Knotenmenge weiter einschränkt und die nicht mehr benötigten Lesesperren freigegeben werden und an andere Nutzer vergeben werden können. Dadurch lässt sich durchaus behaupten, dass der erreichte Grad der Nebenläufigkeit sehr hoch ist. Ein weiterer Grund für die hohe Nebenläufigkeit ist das Sperrprotokoll. Es ermöglicht die gleichzeitige Ausführung von sich gegenseitig nicht beeinflussenden Operationen auf einem Knoten, wie z.B. das gleichzeitige Editieren und Verschieben eines Attribut-/Textknotens.

4.5.5 Untersuchung der Kooperativität des Modells

Da die Kooperativität eine wesentliche Anforderung an das Modell darstellt, soll in diesem Abschnitt noch einmal detailliert darauf eingegangen werden, wie und in wieweit dieses Prinzip verwirklicht wird. Von Kooperation kann, wie bereits in Abschnitt 2.3 erläutert wurde, gesprochen werden, wenn zwischen zwei Benutzern ein bidirektionaler Informationsfluss ermöglicht wird. Dann haben diese beiden Nutzer die Möglichkeit, „verhandelnd“ an einer Aufgabe zu arbeiten. Für das in Abschnitt 1.1.1 dargestellte Anwendungsszenario bedeutet das, dass ein Nutzer Veränderungen an einem Objekt vornimmt, ein anderer Benutzer diese Änderungen analysiert und gegebenenfalls weitere Änderungen ausführt. Diese können nun wiederum vom ersten Benutzer betrachtet werden usw. Die Nutzer verhandeln im Prinzip über einen finalen Objektzustand. Ein derartiges Arbeitskonzept bedeutet einen Verzicht auf Serialisierbarkeit, da diese, wie bereits mehrfach erwähnt, lediglich einen unidirektionalen Informationsfluss zwischen verschiedenen Nutzern zulässt. Hat Nutzer A eine Änderung an einem Objekt ausgeführt

und Nutzer B hat diese Änderung gelesen und gegebenenfalls weitere Veränderungen an dem Objekt vorgenommen, so darf A danach nicht mehr auf das Objekt zugreifen. Da aber die Serialisierbarkeit ein wichtiges Kriterium für die Erhaltung der Konsistenz in einer Datenbank ist, wird sie von den meisten Sperrmodellen erzwungen. So auch von dem sehr häufig eingesetzten 2-Phasen-Sperrprotokoll [[SHK05]]. Es verhindert das Auftreten von Nicht-Serialisierbarkeit eines Schedules dadurch, dass es fordert, dass nach der ersten Freigabe einer Sperre durch eine Aktion, diese nicht erneut eine Sperre anfordern darf. Somit wird verhindert, dass eine Aktion zweimal auf das gleiche Objekt zugreift. (Dies könnte ja zur Nicht-Serialisierbarkeit des Schedules führen, da z.B. zwischen einem ersten und zweiten lesenden Zugriff einer Aktion auf ein Objekt, eine andere Aktion das Objekt z.B. verändert haben könnte.) Würde dieses Sperrprotokoll in dem vorgestellten Modell angewendet, so würde das bedeuten, dass innerhalb der Wurzelaktion nie zweimal auf das gleiche Objekt zugegriffen werden darf, sofern die Sperre auf das Objekt nach dem ersten Zugriff freigegeben wurde. Dies bedeutet natürlich den kompletten Verlust der Kooperativität. Das vorgestellte Sperrprotokoll erlaubt hingegen innerhalb der Wurzelaktion das mehrmalige Zugreifen auf ein Objekt, wodurch das kooperative Arbeiten mehrerer Designer möglich ist.

Die nachfolgende Gleichung zeigt eine vereinfachte Darstellung einer Folge von Operationen innerhalb einer Wurzelaktion, die durch ein 2-Phasen-Sperrprotokoll nicht erlaubt wäre.

$$\begin{aligned} Wurzelaktion = \; & CRL(X)Read(X)EL(X)Edit(X)Unlock(X) \\ & CRL(Y)Read(Y)EL(Y)Edit(Y)Unlock(Y) \\ & CRL(X)Read(X)EL(X)Edit(X)Unlock(X) \end{aligned} \tag{4.9}$$

X und Y sind Attribut-/Textknoten. $Unlock()$ zeigt den Punkt an, an dem die Edit-Lock ($EL()$) freigegeben wird. Die Content-Read-Lock ($CRL()$) wird durch die Anforderung der EL verschärft und muss daher nicht explizit freigegeben werden. Die Aktionen vom $CRL()$ bis zum $Unlock()$ stellen jeweils eine Operationsfolge dar, die bekanntlich einer Subaktion zugeordnet wird. Hier lässt sich eine gewisse Zweiphasigkeit erkennen. Die Wurzelaktion hingegen verhält sich nicht zweiphasig, da nach der Freigabe einer Sperre das erneute Anfordern einer Sperre gestattet wird. Gut zu erkennen ist in der Gleichung auch, dass mehrmals innerhalb der Wurzelaktion lesend und editierend auf den Knoten X zugegriffen werden darf. Dies wäre mit einem klassischen 2-Phasen-Sperrprotokoll, wie bereits erwähnt, nicht möglich. Aber genau dies ermöglicht die Kooperation, wie bereits erläutert, zwischen mehreren Nutzern. Die Konsistenzprobleme, die durch die fehlende Serialisierbarkeit entstehen können, werden dadurch erkannt, dass mindestens eine Leseoperation Bestandteil jeder Operationsfolge ist, und können durch die Operationen Reset und Repeat entsprechend behandelt werden.

4.5.6 Die Deadlockfreiheit des Modells

Deadlocks entstehen bekanntlich dadurch, dass im Falle von zwei Aktionen, jeweils die eine auf eine Sperrfreigabe der anderen wartet. Diesem Phänomen wird durch das vorgestellte Sperrprotokoll wie folgt begegnet:

- Die Anforderung der Sperren für die Leseoperationen innerhalb der beiden Leseoperationsfolgen gemäß Gleichung 4.5 und 4.6 erfolgt atomar. Sollte für einen Knoten die Sperre in diesem Moment nicht anforderbar sein, so führt dies zum Abbruch der Operationsfolge.

- Die Verschärfung der Lesesperren für die Ausführung einer Änderungsoperation erfolgt ebenfalls atomar. Sollte für einen Knoten die Sperre in diesem Moment nicht anforderbar sein, so führt dies zum Abbruch der Operationsfolge.

Durch dieses Vorgehen bei der Sperranforderung werden Deadlocks vermieden. Ein Warten einer Aktion auf die Freigabe einer Sperre durch eine andere Aktion gibt es in diesem Sinne nicht, da bei Nichterhalt einer angeforderten Sperre sofort die zugehörige Operationsfolge abgebrochen wird.

4.5.7 Eigenschaften, die aus dem verwendeten Fehlerbehandlungsmodell resultieren

Das in Abschnitt 4.4 vorgestellte Fehlerbehandlungsmodell bietet ein Reihe von Eigenschaften, die den praktischen Nutzen des Gesamtmodells erhöhen. Diese Eigenschaften sollen nachfolgend genannt werden:

- Durch die Verwendung des Objektversionensystems in Verbindung mit dem lückenlos geführten Log-Buch kann die Fehlerbehandlung sehr einfach und ohne Eingriffe des Nutzers erfolgen. Alle Einträge von ungültig gewordenen Objektversionen werden einfach aus den Log-Büchern entfernt. Es ist keine Definition von semantisch komplexen Kompensationsaktionen notwendig, die gegebenenfalls auch durch den Nutzer erfolgen müsste.
- Ebenso ist die Ausführung eines Resets oder Repeats problemlos und ohne die Definition komplexer Kompensationsaktionen möglich. Der Nutzer wählt einfach einen Objektzustand, welcher wiederhergestellt werden soll. Diese Wiederherstellung bedarf dann lediglich eines weiteren Eintrags in den jeweiligen betroffenen Dimensionen des Log-Buches, der durch die Log-Buch-Verwaltung durchgeführt wird. Diese nimmt auch automatisch Änderungen an den Log-Büchern anderer Knoten vor, die indirekt durch ein Reset oder Repeat betroffen sind. (Stichwort: Das Löschen eines Knotens erfordert auch das Löschen der Nachfolger.)
- Weiterhin bedarf es keines CVS in der zentralen Datenbasis. Dies lässt sich wie folgt begründen. Das verwendete Objektversionensystem ist so konzipiert, dass es immer nur eine aktuelle Version gibt. (Siehe dazu Abschnitt 4.4.1) Durch die Kooperativität des Aktionsmodells besteht für die Nutzer die Möglichkeit, solange über einen Objektzustand zu verhandeln, bis man sich auf eine finale Version geeinigt hat. Da es aufgrund des verwendeten Objektversionensystems nur genau eine finale Version geben kann, kann diese dann ohne weitere Prüfungen bei der zentralen Datenbasis eingecheckt werden.
- Eine letzte Eigenschaft des verwendeten Fehlerbehandlungsmodells ist, dass es die Anzahl der kaskadierenden Abbrüche reduziert, indem es auf eine Atomarität von Operationen auf Teilbäumen nach dem *Complete* der dazugehörigen Subaktion der ersten Ebene verzichtet. (Siehe dazu 4.5.1.). Wird nämlich nach dem *Complete* einer Subaktion der ersten Ebene eine Subaktion tieferer Ebene, die eine Teiloperation einer Operation auf einem Teilbaum enthält, abgebrochen, so werden nicht alle Teiloperationen dieser Operation auf einem Teilbaum mit abgebrochen, sondern nur die, die unbedingt notwendig sind, damit die Struktur des XML-Dokumentes erhalten bleibt. Genauso wird im Übrigen auch beim Reset einer Operation auf einem Knoten verfahren. Ist nämlich diese Operation

eine Teiloperation einer Operation auf einem Teilbaum, so wird auch hier nicht auf allen Knoten des Teilbaumes ein Reset der Operation durchgeführt, sondern nur auf denen, die zur Erhaltung der Struktur des XML-Dokumentes notwendig sind.

4.6 Bewertung und Vergleich

Gegenstand dieses Abschnitts ist eine abschließende Bewertung des vorgeschlagenen Modells hinsichtlich der in Abschnitt 3.1 vorgestellten Kriterien. Dabei wird auch ein Vergleich mit den existierenden Modellen vorgenommen.

Nachfolgend werden zunächst jene Transaktions-/Aktionsmodelle genannt, mit denen kein Vergleich stattfindet, da sie aufgrund ihrer fehlenden Kooperativität nicht als Lösungsmöglichkeit für den in Abschnitt 1.1.1 beschriebenen Anwendungsfall geeignet sind:

- Die Geschlossen geschachtelten Transaktionen realisieren nur das Auftragsprinzip und verbieten durch ihre strenge Forderung nach Isolation jegliche Form der Kooperation.
- Die Sagas stellen eine Realisierung der offen geschachtelten Transaktionen, die ihrerseits Kooperation ermöglichen, dar. Sie unterbinden aber durch die Forderung nach Kommutativität der Subtransaktionen verschiedener Sagas jegliche Kooperation.
- Die geschachtelten dynamischen Aktionen realisieren nur das Auftragsprinzip und unterbinden durch ihre strenge Forderung nach Serialisierbarkeit jegliche Form der Kooperation.

Das Kooperationsprinzip, als das wichtigste der vier Kriterien, wird durch das vorgeschlagene Modell im vollen Umfang erfüllt. Der Beweis dafür wurde bereits in Abschnitt 4.5.5 erbracht. Dabei bietet dieses Modell zwei entscheidende Vorteil, die nachfolgend aufgezeigt werden:

- Es müssen keine Kooperationspartner oder Kooperationsgruppen spezifiziert werden, wie das beispielsweise beim CONCORD-Modell oder den geschachtelten dynamischen Aktionen für kooperative Anwendungen der Fall ist. Dadurch ist es den Nutzern möglich, ihre Interaktion äußerst flexibel zu gestalten.
- Die Behandlung der durch die fehlende Serialisierbarkeit eintretenden Inkonsistenzen ist durch das Reset-Repeat-Prinzip sehr einfach. Sobald ein Nutzer mit einem durch einen anderen Nutzer erzeugten Objektzustand nicht einverstanden ist, kann er einen älteren Objektzustand wiederherstellen, ohne die genaue Semantik der durchgeführten Operationen zu kennen. Es müssen keine Objekte spezifiziert werden, für die ein kooperativer Zugriff nicht erlaubt ist, wie das beim geschachtelten dynamischen Aktionsmodell für kooperative Anwendungen der Fall ist. Weiterhin müssen auch keine Vor-und Nachbedingungen zur Wahrung der Konsistenz definiert werden, wie das beim ConTract-Modell erforderlich ist. Im DOM-Modell werden Konsistenz-Probleme durch die fehlende Serialisierbarkeit nicht betrachtet, daher kann an dieser Stelle keine Aussage über die Komplexität der Behandlungsmethode getroffen werden. Auch bei den offen geschachtelten Transaktionen wird es der Anwendung überlassen, wie sie den Konsistenzproblemen, die durch die fehlende Serialisierbarkeit entstehen, begegnet. Im CONCORD-Modell werden Konsistenzprobleme dadurch gelöst, dass jeder Nutzer lokale Objektkopien hat, mit denen er arbeitet. Allerdings müssen am Ende alle unterschiedlichen Objektzustände zu einer finalen Objektversion vereinigt werden, die dann persistent gesichert wird. Dies erfordert

in den meisten Fällen das Eingreifen der Nutzer und kann sich als äußerst aufwendig gestalten.

Insgesamt lässt sich also festhalten, dass das vorgeschlagene Modell eine äußerst flexible und einfach zu handhabende Form der Kooperation ermöglicht.

Der Grad der Nebenläufigkeit ist bei dem vorgeschlagenen Modell als hoch einzustufen. Dies liegt daran, dass benutzte Objekte sehr schnell wieder freigeben werden. Ein Nutzer darf maximal eine Änderungsoperation ausführen. Danach steht das Objekt wieder zur freien Verfügung. In Abschnitt 4.5.4 wurde sogar ein Maß für die Nebenläufigkeit angegeben. Ein direkter Vergleich mit den anderen vorgestellten kooperativen Transaktions-/Aktionsmodellen ist nicht möglich, da diese sehr allgemein formuliert sind und keine Operationsfolgen festlegen.

Die Komplexität des vorgeschlagenen Fehlerbehandlungsmodells ist als gering einzustufen. Es müssen weder semantisch komplexe Kompensationsaktionen spezifiziert, noch semantische Vor- und Nachbedingungen festgelegt und überwacht werden. Die Fehlerbehandlung kann vollständig durch das System ausgeführt werden und bedarf keiner Eingriffe des Nutzers. Dies stellt einen entscheidenden Vorteil gegenüber den offen geschachtelten Transaktionen, dem DOM-Modell und dem ConTract-Modell dar. Lediglich das CONCORD-Modell bietet ebenfalls ein Objektversionensystem zur Fehlerbehandlung an. Allerdings besteht dort das Problem, dass kein Mechanismus vorgesehen ist, persistent gesicherte Änderungen aus der Datenbank zu entfernen. (Dies kann bekanntlich bei einem kaskadierenden Abbruch erforderlich sein.) In dem in dieser Arbeit vorgeschlagenen Modell ist eine solche Situation gar nicht möglich, da es auf den dynamischen Aktionen aufbaut, die die Dauerhaftigkeit gewährleisten. Ein direkter Vergleich mit den geschachtelten dynamischen Aktionen für kooperative Anwendungen kann nicht stattfinden, da in [Kru97] keine Betrachtungen hinsichtlich des zu verwendenden Fehlerbehandlungsmodells durchgeführt wurden.

Abschließend sollen noch einige Bemerkungen zur Anwenderunabhängigkeit folgen. Das in dieser Arbeit vorgestellte Modell ist als sehr anwenderunabhängig einzustufen. Dies hat folgende Ursachen:

- Die Fehlerbehandlung kann ohne Intervention des Nutzers erfolgen, wie bereits geschildert. Dabei stellt auch das Reset-Repeat-Prinzip, zur Behandlung von Konsistenzproblemen infolge des Verzichts auf Serialisierbarkeit, keine Einschränkung dar. Reset und Repeat stellen sich dem Nutzer als ganz normale Operationen dar, für die er kein bestimmtes Vorwissen braucht. Er kann einfach einen Objektzustand wiederherstellen, ohne zu wissen, wessen oder welche Änderungen er dabei rückgängig macht.
- Es muss trotz Verwendung eines Objektversionensystems kein Verschmelzen von Objektversionen wie beim CONCORD-Modell stattfinden. Am Ende der Arbeiten steht, wie aus Abschnitt 4.4 ersichtlich, immer genau eine finale Objektversion fest. Dies macht auch die Verwendung eines CVS überflüssig.

Zusammenfassend kann gesagt werden, dass das in dieser Arbeit entwickelte Modell die gestellten Bewertungskriterien im vollen Umfang erfüllt. Dabei bietet es auch einige Vorteile gegenüber bereits existierenden Transaktions-/Aktionskonzepten.

5 Systementwurf

Gegenstand dieses Kapitels ist der objektorientierte Entwurf des in Kapitel 4 vorgestellten Modells. Dieses Modell umfasst bekanntlich die geschachtelten dynamischen Aktionen, das Sperrmodell und das Fehlerbehandlungsmodell. Jedes dieser Teilmodelle wird in Form von Managern realisiert. Daher gibt es einen Actionmanager für die dynamischen Aktionen, einen Lockmanager, der das Sperrmodell realisiert und einen Objectversionmanager und Logbookmanager, die zusammen das Fehlerbehandlungsmodell umsetzen. Zunächst werden diese Komponenten erläutert. Hierbei soll jedoch festgehalten werden, dass die Beschreibungen nur die für das Verständnis unbedingt erforderlichen Elemente (Membervariablen und Funktionen) der Komponenten umfassen, da eine vollständige Komponentenbeschreibung den Rahmen der Arbeit sprengen würde. Anschließend werden die vorgestellten Klassen in die Client-Server-Architektur eingebettet.

5.1 Objektorientierter Entwurf des Modells

5.1.1 Der Actionmanager

Der Actionmanager dient der Verwaltung der Aktionen. Seine Schnittstelle orientiert sich an den Zuständen des Aktionsmodells. Folglich bietet der Actionmanager eine Funktion für das *Begin*, das *Complete*, das *Commit* und das *Abort* einer Aktion an.

Abbildung 5.1 zeigt zunächst das Klassendiagramm des Actionmanagers. Nachfolgend soll die

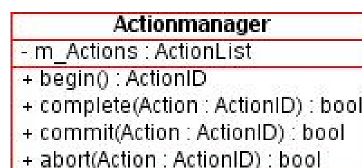

Abbildung 5.1: Klassendiagramm des Actionmanagers

Klasse näher erläutert werden.

Die Membervariable *m_Actions* speichert alle laufenden Aktionen in Listenform. Die Aktionen werden dabei über ihre *ActionID* identifiziert. Weiterhin wird zu jeder Aktion der Zustand ihrer Ausführung festgehalten. Dabei bedarf es allerdings nur eines *Running* und eines *Completed*, um anzuzeigen, dass die Aktion läuft bzw. *completed* ist. Beim *Commit* oder *Abort* einer Aktion wird diese aus der Liste entfernt, wodurch kein Vermerk des Zustandes erforderlich ist.

Die Funktion *begin()* dient dazu, eine neue Aktion zu starten. Sie führt zum Anfügen einer neuen Aktion (*ActionID, Running*) an *m_Actions* und gibt die vergebene *ActionID* an den Aufrufer zurück.

Mit dem Aufruf von *complete(Action: ActionID)* wird die durch den Parameter *Action* identifizierte Aktion in den Zustand *Completed* versetzt. Intern erfolgt dabei eine Änderung des Zustandes in der Aktionsliste *m_Actions* bei der entsprechenden *ActionID*. Bei erfolgreicher Zustandsänderung liefert die Funktion *true* zurück.

Die Funktion *commit(Action: ActionID)* führt zum *Commit* der entsprechenden Aktion. Diese wird dann aus der Liste *m_Actions* entfernt. Bei Erfolg liefert die Funktion *true*, ansonsten *false* zurück.

Durch einen Aufruf von *abort(Action: ActionID)* wird die entsprechende Aktion abgebrochen und aus der Liste der Aktionen *m_Actions* entfernt. Bei erfolgreichem Abbruch liefert die Funktion *true*, ansonsten *false* zurück.

Nun sollen noch einige Worte zu dem Datentyp *ActionID* folgen. Dieser stellt einen *String* mit folgendem besonderen Aufbau dar:

$$ActionID = Int1.Int2.Int3.Int4 \tag{5.1}$$

Die *ActionID* besteht also aus einer Konkatenation von maximal vier *Integerwerten*, jeweils getrennt durch einen Punkt. $Int1$ ist dabei der Wurzelaktion zugeordnet, $Int2$ einer Subaktion, $Int3$ einer SubSubaktion und $Int4$ einer SubSubSubaktion. Der Wertebereich der *Int*s beginnt mit *1* und endet mit dem maximalen *Integerwert*, der systemseitig möglich ist. Vorteilhaft ist dieser Aufbau deswegen, weil er einerseits Rückschlüsse darauf erlaubt, welche Subaktion z.B. zu welcher Wurzelaktion gehört, und er andererseits einen viel größeren ID-Pool zur Verfügung stellt, als wenn jede Aktion mit einem *Integerwert* identifiziert würde. Die Größe des ID-Pools ist sehr wichtig, da jede Aktion eindeutig identifiziert werden muss. Innerhalb einer Arbeitssitzung darf eine *ActionID* nur einmal vergeben werden, es sei denn, eine andere Aktion mit der selben ID wurde abgebrochen oder befindet sich im Zustand *Committed*. In diesen beiden Fällen wird bekanntlich die Aktion und damit auch die *ActionID* aus der Aktionsliste *m_Actions* entfernt. Wird also z.B. eine Subaktion mit der *ActionID 1.4* abgebrochen, so kann diese ID neu vergeben werden.

5.1.2 Der Lockmanager

Der Lockmanager dient der Verwaltung der Sperren, die von den Aktionen auf XML-Knoten angefordert werden. Intern realisiert er dabei das in Abschnitt 4.3.3 vorgestellte Sperrprotokoll.

Abbildung 5.2 zeigt das Klassendiagramm des Lockmanagers. Nachfolgend wird die Klasse

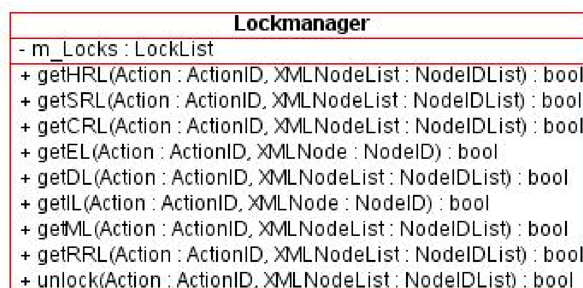

Abbildung 5.2: Klassendiagramm des Lockmanagers

kurz beschrieben.

Die Membervariable *m_Locks* speichert in Listenform alle vergebenen Sperren. Sie ist vom Datentyp *LockList*. *LockList* ist eine Liste von Einträgen der Form: *ActionID*, *NodeID*, *Locktype*. Die *ActionID* kennzeichnet, wie bereits bekannt, die Aktion, welche die Sperre angefordert hat. Die *NodeID* stellt eine Konkatenation einer *DocumentID* und einer *DeweyID* dar. Nur durch die beiden IDs kann der XML-Knoten, für den eine Sperre angefordert wurde, widerspruchsfrei bestimmt werden, da *DeweyIDs* nur innerhalb eines XML-Dokumentes eindeutig sind. Der *Locktype* gibt die Art der Sperre an, die auf dem XML-Knoten liegt. Die verschiedenen Sperrtypen wurden in Abschnitt 4.3.3 eingeführt.

Die Funktion *getHRL(Action: ActionID, XMLNodeList: NodeIDList)* dient dazu, holografische Lesesperren auf eine Knotenmenge anzufordern. Sie übernimmt die *ActionID* der anfordernden Aktion und alle Knoten, für die die Sperre erlangt werden soll, in Form einer Liste von *NodeIDs*. Die Funktion überprüft anhand des Sperrprotokolls, ob für die gesamte Knotenmenge die holografischen Lesesperren erlangt werden können. Zu dieser Prüfung gehört auch eine Suche innerhalb von *m_Locks*, ob es eine direkte Vorgängeraktion innerhalb der gleichen Subaktion gibt, die bereits Lesesperren auf die Knoten hält. (Dies entspricht dem Hineinspringen in eine Verschiebung gemäß den Gleichungen 4.5 und 4.6) Dies ist aufgrund des besonderen Aufbaus der *ActionIDs* möglich. Fordert z.B. eine SubSubaktion mit der *ActionID 1.4.2* HRL für eine Knotenmenge an, so wird geprüft, ob eine SubSubaktion mit der *ActionID 1.4.1* (Dies ist der direkte Vorgänger von der SubSubaktion *1.4.2*.) SRL auf dieser Knotenmenge hält. Ist dies der Fall, so wird anhand des Sperrprotokolls geprüft, ob eine Sperrvergabe möglich ist, und die Einträge in *m_Locks* dahingehend verändert, dass die alte *ActionID* durch die neue und der alte Sperrtyp SRL durch den neuen, HRL, ersetzt wird. Da eine Folgeleseoperation in diesem Falle immer einschränkend ist, werden die Einträge der Vorgängeraktion, die sich auf Knoten beziehen, die nicht mehr Teil der Knotemenge der aktuellen Aktion sind, aus *m_Locks* entfernt. Gibt es zu der aktuellen Aktion keinen Vorgänger, der zur gleichen Subaktion gehört, so können auch keine Einträge von diesem Vorgänger in *m_Locks* existieren. In diesem Fall führen die vergebenen Sperren zum Anfügen neuer Einträge an *m_Locks*. Die Einträge enthalten dann jeweils die *ActionID* der anfordernden Aktion, die *NodeID* des Knotens, für den die Sperre vergeben wurde, und den *Locktype* HRL. Können nicht für alle Knoten die Sperren erlangt werden, so scheitert die gesamte Sperranforderung. In diesem Fall wird kein Eintrag an *m_Locks* angefügt und die Funktion liefert *false* zurück. Ansonsten wird *true* zurückgegeben.

Mit der Funktion *getSRL(Action: ActionID, XMLNodeList: NodeIDList)* werden die Strukturlesesperren für eine Knotenmenge angefordert. Jeder Knoten aus der Knotenliste *XMLNodeList* wird gemäß dem Sperrprotokoll überprüft. Dabei wird auch überprüft, ob ein direkter Vorgänger strukturelle Lesesperren auf der Knotenmenge hält. Ist es für einen Knoten nicht möglich eine SRL zu erlangen, so wird versucht, mit *getHRL(...)* zumindest eine holografische Lesesperre für diesen Knoten zu bekommen. Ist auch das nicht möglich, so führt dies zum Scheitern der gesamten Funktion *getSRL(...)*, welche dann *false* zurück gibt. Ansonsten werden die vergebenen Sperren an *m_Locks* angehängt oder die existierenden Einträge geändert (Falls eine direkte Vorgängeraktion innerhalb derselben Subaktion existiert.), wobei der Sperrtyp hier SRL oder HRL sein kann, und die Funktion liefert *true* zurück.

Die Funktion *getCRL(Action: ActionID, XMLNodeList: NodeIDList)* dient der Anforderung der inhaltlichen Lesesperren für eine Knotenmenge. Kann für einen Knoten keine CRL vergeben werden, so wird mit *getSRL(...)* versucht, eine SRL zu erlangen. Ist auch das nicht möglich,

so wird mit *getHRL(...)* geprüft, ob zumindest eine HRL für diesen Knoten vergeben werden kann. Scheitert auch diese Funktion, so wird für die gesamte Knotenmenge keine Sperre erlangt und *getCRL(...)* liefert *false* zurück. Ansonsten werden die Einträge mit den entsprechenden Sperrtypen, also entweder CRL, SRL oder HRL, an *m_Locks* angehängt oder die existierenden Einträge geändert und die Funktion liefert *true* zurück.

Mit *getEL(Action: ActionID, XMLNode: NodeID)* wird die Editiersperre für einen Knoten *XML-Node* angefordert. Dabei erfolgt anhand des vorgestellten Sperrprotokolls eine Prüfung, ob eine Sperrvergabe überhaupt möglich ist. Zu dieser Prüfung gehört auch, dass die Funktion in der Sperrliste *m_LockList* sucht, ob die Vorgängeraktion (innerhalb der selben Subaktion) von der jetzt anfordernden Aktion eine CRL auf diesen Knoten hält. Dazu muss einfach über die Liste iteriert werden und der entsprechende Eintrag gesucht werden. Ist nun eine Sperrvergabe möglich, so wird der gefundene Eintrag dahingehend verändert, dass die alte *ActionID* durch die neue und der alte Sperrtyp, CRL, durch EL ersetzt wird. Alle anderen Einträge der direkten Vorgängeraktion werden aus *m_Locks* gelöscht. (Der Aufbau schreibt bekanntlich vor, dass einer Editiertoperation mindestens eine Leseoperation voranzugehen hat. Des Weiteren darf einer Editieroperation keine weitere Operation innerhalb der selben Subaktion folgen.) Die Einträge von anderen Aktionen anderer Designer, die eine CRL auf diesen Knoten haben, müssen gemäß dem Sperrprotokoll dahingehend geändert werden, dass der Sperrtyp in SRL umgewandelt wird. Wurde die EL erfolgreich an die anfordernde Aktion vergeben, so gibt die Funktion *true*, ansonsten *false* zurück.

Die Funktion *getDL(Action: ActionID, XMLNodeList: NodeIDList)* dient der Anforderung von Löschsperren auf eine Knotenmenge. Dabei muss natürlich überprüft werden, ob die Knotenmenge ein vollständiger Teilbaum ist, da ansonsten die Struktur des XML-Dokumentes zerstört wird. Ob die Sperren für die Knotemenge vergeben werden können, wird anhand des vorgestellten Sperrprotokolls überprüft. Dies Prüfung beinhaltet wiederum ein Durchsuchen von *m_Locks*, ob die Vorgängeraktion dieser Aktion (innerhalb der gleichen Subaktion) CRLs auf diese Knotemenge hält. Ist dies nicht der Fall, so liefert die Funktion *false* zurück. Ansonsten wird die Knotenmenge anhand des Sperrprotokolls weiter darauf überprüft, ob es zu Kollisionen mit Sperren anderer Aktionen kommt. Können für alle Knoten der Menge DLs erlangt werden, so führt dies zur Änderung der bestehenden Einträge der Vorgängeraktion in *m_Locks* und die Funktion liefert *true* zurück. (Die CRLs werden durch DLs und die *ActionID* der Vorgängeraktion durch die aktuelle *ActionID* ersetzt.) Die Einträge der direkten Vorgängeraktion, die sich auf Knoten beziehen, die nicht Teil der aktuellen Knotenmenge, für die DLs angefordert werden,sind, werden aus *m_Locks* gelöscht. Die Einträge von anderen Aktionen anderer Designer, die eine CRL auf diesen Knoten haben, müssen gemäß dem Sperrprotokoll dahingehend geändert werden, dass der Sperrtyp in HRL umgewandelt wird. Kann auch nur für einen Knoten der Menge keine DL erlangt werden, so führt dies zum kompletten Scheitern der Funktion und sie liefert *false* zurück.

Mit *getIL(Action: ActionID, XMLNode: NodeID)* wird eine Einfügesperre für einen Knoten angefordert. Dabei muss, wie bei den vorhergehenden Funktionen auch, sichergestellt werden, dass es zu keinen Kollisionen mit anderen Sperren kommt und dass ein Eintrag einer direkten Vorgängeraktion in *m_Locks* existiert, der dieser Vorgängeraktion eine CRL oder SRL auf diesen Knoten bestätigt. Kann die IL für den Knoten erlangt werden, so wird der bestehende Eintrag der Vorgängeraktion geändert (Die *ActionID* der Vorgängeraktion wird durch die *ActionID* der aktuellen Aktion und die CRL oder SRL wird durch die IL ersetzt.) und die Funktion liefert *true* zurück. Außerdem erfolgt wieder ein Entfernen der restlichen Einträge der Vorgängeraktion und

eine Anpassung der Einträge der anderen Aktionen anderer Designer gemäß dem Sperrprotokoll.

Mit der Funktion *getML(Action: ActionID, XMLNodeList: NodeIDList)* werden die Sperren für die zu verschiebende Knotenmenge angefordert. Dabei muss wieder überprüft werden, ob die Knotenmenge ein vollständiger Teilbaum ist, da ansonsten die Struktur des XML-Dokumentes zerstört wird. Ob die Sperren für die Knotemenge vergeben werden können, wird anhand des Sperrprotokolls überprüft. Diese Prüfung beinhaltet wiederum ein Durchsuchen von *m_Locks*, ob die Vorgängeraktion dieser Aktion (innerhalb der gleichen Subaktion) CRLs, SRLs oder HRLs (Der Knoten darf dann aber nicht die Wurzel der Verschiebung sein.) auf dieser Knotemenge hält. Ist dies nicht der Fall, so liefert die Funktion *false* zurück. Ansonsten wird die Knotenmenge anhand des Sperrprotokolls weiter darauf überprüft, ob es zu Kollisionen mit Sperren anderer Aktionen kommt. Können für alle Knoten der Menge MLs erlangt werden, so führt dies zur Änderung der bestehenden Einträge der Vorgängeraktion in *m_Locks* und die Funktion liefert *true* zurück. (Die CRLs, SRLs oder HRLs werden durch MLs und die *ActionID* der Vorgängeraktion durch die aktuelle *ActionID* ersetzt.) Die Einträge der direkten Vorgängeraktion, die sich auf Knoten beziehen, die nicht Teil der aktuellen Knotenmenge, für die MLs angefordert werden,sind, werden aus *m_Locks* gelöscht. Die Einträge von anderen Aktionen anderer Designer, die eine CRL oder HRL auf diesen Knoten haben, müssen gemäß dem Sperrprotokoll dahingehend geändert werden, dass der Sperrtyp in SRL umgewandelt wird. Kann auch nur für einen Knoten der Menge keine ML erlangt werden, so führt dies zum kompletten Scheitern der Funktion und sie liefert *false* zurück.

Die Funktion *getRRL(Action: ActionID, XMLNodeList: NodeIDList)* dient der Anforderung der RRLs für eine Knotenmenge. Dabei muss wieder auf eventuelle Kollisionen mit anderen RRLs geprüft werden und es muss sichergestellt werden, dass die direkte Vorgängeraktion dieser Aktion auf allen Knoten der Menge CRLs, SRLs, oder HRLs und somit die entsprechenden Einträge in *m_Locks* hat. Diese werden dann entsprechend abgeändert. Die restlichen Einträge der Vorgängeraktion werden entfernt. Für die Einträge der anderen Aktionen anderer Designer, die sich auf die Knotenmenge beziehen, gilt:

- Haben die anderen Aktionen CRLs oder SRLs auf den Knoten, so werden in den Einträgen diese Sperrtypen in HRLs umgewandelt.
- Haben die anderen Aktionen ILs, MLs, ELs oder DLs auf dieser Knotenmenge, so werden ihre Einträge entfernt, da ein Reset oder Repeat immer zum Abbruch aller Änderungsoperationen führt.

Wurden für alle Knoten aus *XMLNodeList* die RRLs erlangt, so liefert die Funktion *true*, ansonsten *false* zurück.

Die Funktion *unlock(Action: ActionID, XMLNodeList: NodeIDList)* führt zur Freigabe der Sperren auf den angegebenen Knoten. Dabei werden die Einträge aus *m_Locks* entfernt, die die *ActionID* der aufrufenden Aktion und die *NodeID* aus der Liste *XMLNodeList* tragen. Konnten für alle angegebenen Knoten die Einträge entfernt werden, so gibt die Funktion *true*, ansonsten *false* zurück.

5.1.3 Der Objectversionmanager

Der Objectversionmanager dient der Speicherung der Objektversionen. Abbildung 5.3 zeigt das Klassendiagramm des Objectversionmanagers.

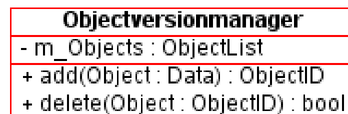

Abbildung 5.3: Klassendiagramm des Objectversionmanagers

Die Membervariable *m_Objects* speichert Einträge der Form (*ObjectID, Data*) als Liste. Die *ObjectID* dient der Identifikation einer Objektversion, die vom Typ *Data* ist. Die *ObjectID* kann einfach als eine fortlaufende *Integerzahl* angesehen werden.

Mit der Funktion *add(Object: Data)* wird die durch z.B. ein Edit erzeugt Objektversion an *m_Objects* angehängt. Dabei erhält diese eine *ObjectID*, die auch an die aufrufende Funktion zurückgegeben wird.

Die Funktion *delete(Object: ObjectID)* dient dem Entfernen eines über den Parameter *Object* spezifizierten Eintrags aus *m_Objects*. Bei Erfolg liefert die Funktion *true*, ansonsten *false* zurück.

5.1.4 Der Logbookmanager

Der Logbookmanager dient der Führung, Verwaltung und Pflege der Logbücher. Auf diese Aspekte wurde bereits in den Abschnitten 4.4.3 und 4.4.4 eingegangen.

Zunächst soll die Datenstruktur Logbook dargestellt werden, die die Historie eines Knotens festhält. Abbildung 5.4 zeigt zunächst das Klassendiagramm des Logbooks. Nachfolgend soll

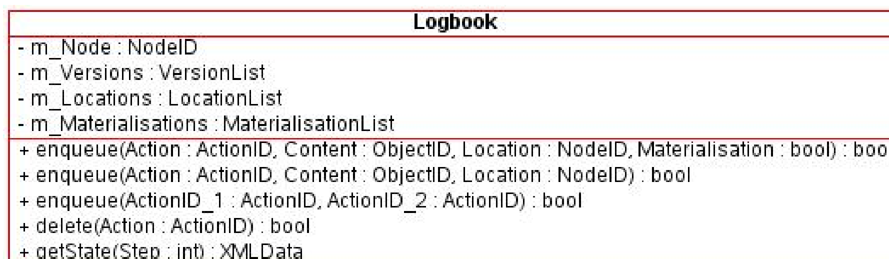

Abbildung 5.4: Klassendiagramm des Logbooks

die Klasse näher erläutert werden.

Die Membervariable *m_Node* dient der eindeutigen Identifikation des Knotens, für den das Log-Buch geführt wird. Sie ist vom Datentyp *NodeID*. Dieser stellt, wie bereits mehrfach erwähnt, eine Konkatenation einer *DocumentID* und einer *DeweyID* dar. Diese ID wird beim Anlegen des Log-Buches gebildet. Dabei wird die *DocumentID* und die *DeweyID* verwendet, die den Knoten zu Beginn der Arbeitssitzung eindeutig identifiziert haben. Die Membervariable wird danach nicht mehr verändert.

Die Membervariable *m_Versions* bildet die Dimension Inhalt des Log-Buches ab. Die Variable ist vom Datentyp *VersionList*. Dieser stellt eine Liste von Einträgen dar. Die Einträge enthalten folgende Informationen:

- Die *ActionID* der Aktion, die die Objektversion erzeugt hat,
- Die ID der Objektversion (Die Objektversion selbst wird im Objectversionmanager gespeichert),
- Eine Liste von *ActionIDs*, die die Leseoperationen identifizieren. Für das erste Listenelement in *m_Versions* sind dies die Nachfolgeleseoperationen und für jedes weitere Listenelement die Vorgängerleseoperationen. (Siehe dazu Abschnitt 4.4.2)

Die Membervariable *m_Locations* bildet die Dimension Ort des Log-Buches ab. Sie ist vom Datentyp *LocationList*. Dieser stellt eine Liste von Einträgen dar. Diese beinhalten folgende Informationen:

- Die *ActionID* der Aktion, die den Knoten an den neuen Ort verschoben hat,
- Die neue *NodeID* des Knotens,
- Eine Liste von *ActionIDs*, die die Vorgängerleseoperationen dieser Aktion identifizieren.

Die Membervariable *m_Materialisations* bildet die Dimension Materialisierung des Log-Buches ab. Sie ist vom Datentyp *MaterialisationList*. Dieser stellte eine Liste von Einträgen dar. Diese beinhalten folgende Informationen:

- Die *ActionID* der Aktion, die zur Änderung des Materialisierungszustandes geführt hat,
- Einen *Bool*-Wert, der angibt, ob der Knoten materialisiert ist oder nicht,
- Eine Liste von *ActionIDs*, die die Vorgängerleseoperationen dieser Aktion identifizieren.

Die Funktion *enqueue(Action: ActionID, Content: ObjectID, Location: NodeID, Materialisation: bool)* dient dem Anfügen jeweils eines Eintrags in *m_Versions*, *m_Locations* und *m_Materialisations*. Der Parameter *Action* übernimmt dabei die *ActionID* der Operation, *Content* die ID der Objektversion, *Location* die neue *NodeID* des Knotens und *Materialisation* den neuen Materialisierungszustand des Knotens. Soll keine neue Objektversion angelegt werden, so wird beim Parameter *Content* die *0* übergeben. Soll kein neuer Eintrag in der Dimension Ort erfolgen, so wird für *Location* die *0* übergeben. Der Parameter *Materialisation* muss jedoch immer gesetzt werden. Bei erfolgreichem Anhängen der Einträge gibt die Funktion *true*, ansonsten *false* zurück.

Die Funktion *enqueue(Action: ActionID, Content: ObjectID, Location: NodeID)* dient dem Anfügen jeweils eines Eintrags in *m_Versions* und *m_Locations*. Soll keine neue Objektversion angelegt werden, so wird beim Parameter *Content* die *0* übergeben. Soll kein neuer Eintrag in der Dimension Ort erfolgen, so wird für *Location* die *0* übergeben. Bei erfolgreichem Anhängen der Einträge gibt die Funktion *true*, ansonsten *false* zurück.

Mit *enqueue(Action_1: ActionID, Action_2: ActionID)* wird die *ActionID* einer Leseoperation (ActionID_1) an die Liste der Leseoperationen eines Eintrags in *m_Versions*, *m_Locations* oder *m_Materialisations* angehängt, der durch die *ActionID_2* gekennzeichnet ist. Bei erfolgreichem Anhängen der *ActionID* gibt die Funktion *true*, ansonsten *false* zurück.

Die Funktion *delete(Action: ActionID)* dient im Falle eines Abbruchs der Aktion dem Entfernen aller Einträge aus dem Log-Buch, die die durch den Parameter *Action* spezifizierte *ActionID*

tragen. Dabei muss abhängig von der Dimension entschieden werden, ob gegebenenfalls Folgeeinträge entfernt werden müssen. Dieser Aspekt wurde zu Beginn des Abschnitts 4.4.3. Bei erfolgreichem Entfernen der Einträge liefert die Funktion *true*, ansonsten *false* zurück.

Die Methode *getState(Step: int)* dient der Ermittlung eines früheren Zustands des Knotens. Über den Parameter *Step* werden dabei die Zeitschritte angegeben, die in der Historie des Knotens zurückgegangen werden soll. Der Zustand wird dann in Form von XML-Daten, die aus der Objektversion, der Materialisierung und dem Ort des Knotens gewonnen werden, an den Aufrufer zurückgegeben.

Zur Verwaltung der Log-Bücher der einzelnen Knoten dient der in Abbildung 5.5 dargestellte Logbookmanager. Nachfolgend soll die Klasse kurz erläutert werden.

Logbookmanager
- m_Logbooks : LogbookList
+ enqueue(Node : NodeID, Action : ActionID, Content : ObjectID, Location : NodeID, Materialisation : bool) : bool + enqueue(Node : NodeID, Action : ActionID, Content : ObjectID, Location : NodeID) : bool + enqueue(Node : NodeID, Action_1 : ActionID, Action_2 : ActionID) : bool + delete(Node : NodeID, Action : ActionID) : bool + searchCommitted() : bool + getNodes(Node : NodeID, Content : ObjectID, Location : NodeID, Materialisation : bool) : NodeIDList + getState(Node : NodeID, Step : int) : XMLData + getCommitabelActions() : ActionIDList

Abbildung 5.5: Klassendiagramm des Logbookmanagers

Die Membervariable *m_Logbooks* ist vom Datentyp *LogbookList* und speichert in Listenform die Log-Bücher der einzelnen Knoten.

Die Funktion *enqueue(Node: NodeID, Action: ActionID, Content: ObjectID, Location: NodeID, Materialisation: bool)* ruft die Funktion *enqueue(Action: ActionID, Content: ObjectID, Location: NodeID, Materialisation: bool)* des Log-Buches auf, bei dem die *NodeID* von *m_Node* mit der *NodeID* des Parameters *Node* übereinstimmt. Existiert ein entsprechendes Log-Buch noch nicht in der Liste, so wird es angelegt und der Liste hinzugefügt. Existiert das Log-Buch und werden in der Funktion mehr Parameter als nur *Materialisation* spezifiziert, so handelt es sich bei der zur Aktion gehörenden Operation um eine Reset oder Repeat. Wurde die Funktion erfolgreich ausgeführt, so liefert sie *true*, ansonsten *false* zurück.

Die Funktion *enqueue(Node: NodeID, Action: ActionID, Content: ObjectID, Location: NodeID)* ruft die Funktion *enqueue(Action: ActionID, Content: ObjectID, Location: NodeID)* des Log-Buches auf, bei dem die *NodeID* von *m_Node* mit der *NodeID* des Parameters *Node* übereinstimmt. Beim Aufruf dieser Funktion muss das Log-Buch auf jeden Fall existieren, da hier der Parameter *Materialisation* nicht spezifiziert werden kann. Wird beim Aufruf sowohl der *Content* als auch die *Location* spezifiziert, so handelt es sich bei der zur Aktion gehörenden Operation um ein Reset oder Repeat. Wurde die Funktion erfolgreich ausgeführt, so liefert sie *true*, ansonsten *false* zurück.

Die Funktion *enqueue(Node: NodeID, Action_1: ActionID, Action_2: ActionID)* ruft die Funktion *enqueue(Action_1: ActionID, Action_2: ActionID)* des durch *Node* spezifizierten Log-Buches auf. Wurde die Funktion erfolgreich ausgeführt, so liefert sie *true*, ansonsten *false* zurück.

Die Funktion *delete(Node: NodeID, Action: ActionID)* ruft die Funktion *delete(Action: ActionID)* des durch *Node* spezifizierten Log-Buches auf. Weiterhin durchsucht sie die Liste von Log-Büchern, ob auch für andere Knoten Einträge aufgrund des Abbruchs der Aktion mit der spezifizierten *ActionID* entfernt werden müssen (kaskadierender Abbruch), um die Struktur des

XML-Dokumentes zu erhalten. Wird ein Log-Buch durch das Entfernen eines Eintrags leer, so wird es aus der Liste der Log-Bücher entfernt. Wurde die Funktion erfolgreich ausgeführt, so liefert sie *true*, ansonsten *false* zurück.

Die Funktion *searchCommitted()* durchsucht die Liste der Log-Bücher und prüft für jedes Log-Buch, ob sich alle Aktionen innerhalb eines Log-Buches im Zustand *Committed* befinden. (Dazu muss natürlich auch auf die Aktionsliste im Actionmanager zugegriffen werden. Dies geschieht über die üblichen *get*-Methoden, die nicht explizit aufgeführt sind.) Ist das der Fall, so kann des Log-Buch aus der Liste entfernt werden und die Funktion liefert *true*, ansonsten *false* zurück.

Die Funktion *getNodes(Node: NodeID, Content: ObjectID, Location: NodeID, Materialisation: bool)* dient der Ermittlung der Knoten, für die ein Reset oder Repeat mit ausgeführt werden muss, damit die Struktur des XML-Dokumentes erhalten bleibt. Der Nutzer spezifiziert bekanntlich für einen Knoten einen Zustand, der mit einem Reset oder Repeat erzeugt werden soll. Der Knoten wird über den Parameter *Node* und der Zustand über die Parameter *Content*, *Location* und *Materialisation* spezifiziert. Die Funktion iteriert nun über alle Log-Bücher und ermittelt die Knoten, für die das Reset und Repeat ebenfalls ausgeführt werden muss, und gibt sie in Form einer Liste von *NodeIDs* zurück.

Die Methode *getState(Node: NodeID, Step: int)* ruft die Methode *getState(Step: int)* des über den Parameter *Node* spezifizierten Log-Buches auf.

Die Methode *getCommitableActions()* dient dazu, alle Aktionen zu ermitteln, die nicht mehr abgebrochen werden können. Es werden also alle Log-Bücher durchlaufen und für jede Aktion geschaut, von welchen anderen Aktionen diese Aktion abhängt. Dadurch kann ermittelt werden, ob eine Aktion noch von einem kaskadierenden Abbruch betroffen werden kann. Alle Aktionen, die garantiert nicht mehr infolge eines kaskadierenden Abbruchs rückgesetzt werden müssen, werden in Form einer Liste von *ActionIDs* zurückgeliefert.

5.2 Einbettung des Modells in die Client-Server-Architektur

5.2.1 Der Server

Der Server ist das Herzstück des verteilten XML-Verarbeitungssystems. Er umfasst nicht nur die XML-Datenbank, sondern auf ihm laufen auch alle im letzten Abschnitt vorgestellten Manager. Abbildung 5.6 zeigt das Klassendiagramm des Servers. Nachfolgend soll der Server kurz erläutert werden.

Das Interface ServerConnectInterface

Über das Interface *ServerConnectInterface*, welches die Klasse *Server* implementiert, stellt der Server allen Clients eine Kommunikationsschnittstelle zur Verfügung und bietet damit eine Reihe von Diensten in Form von Funktionen an. Anfragen, die über die bereits im letzten Abschnitt beschriebenen Funktionen an den Server gestellt werden, werden zur Bearbeitung an die ent-

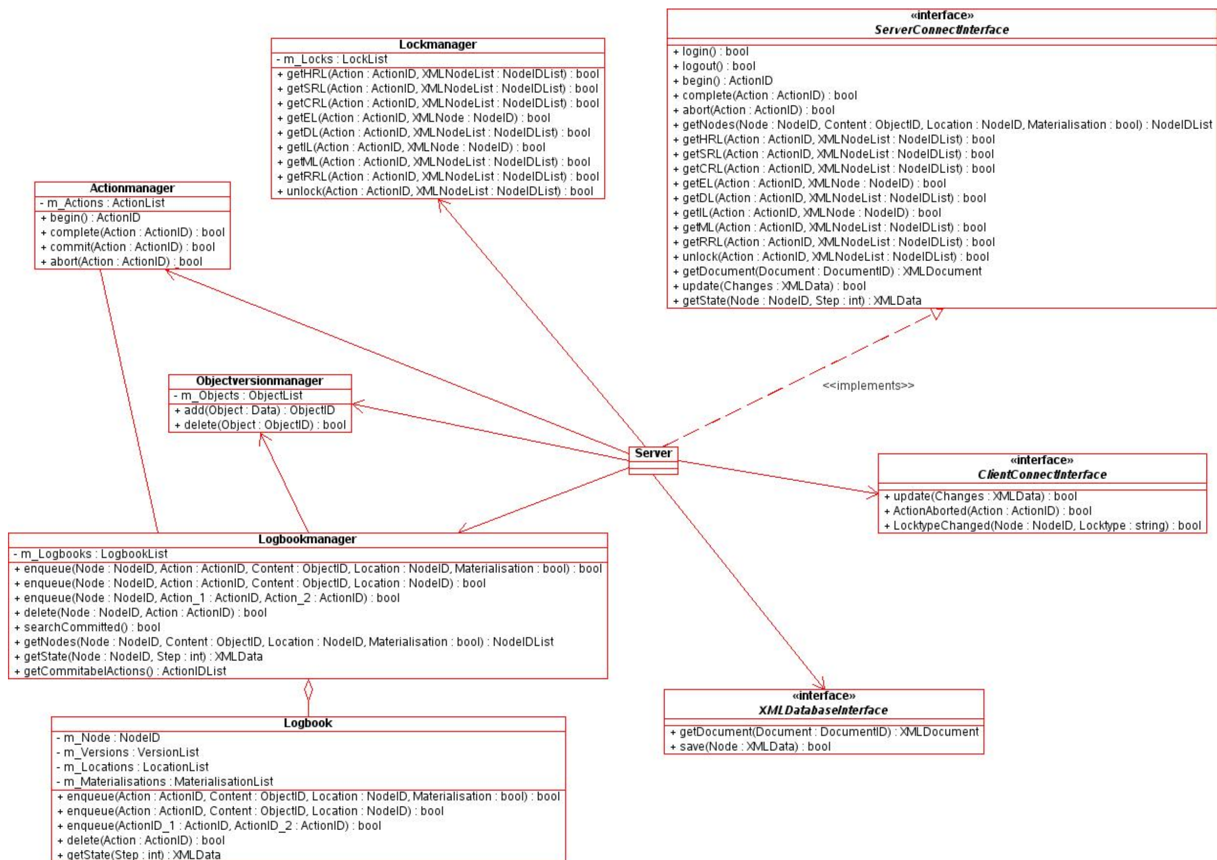

Abbildung 5.6: Klassendiagramm des Servers

sprechenden Klassen bzw. Manager weitergeleitet. Zusätzlich zu den bereits bekannten Funktionen bietet das Interface vier weitere an, auf die nun genauer eingegangen werden soll.

Mit Hilfe der Funktion *login()* kann sich ein Client mit dem Server verbinden. Zum Trennen der Verbindung wird die Funktion *logout()* benutzt. Beide Funktionen liefern bei Erfolg jeweils *true*, ansonsten *false* zurück.

Über die Funktion *getDocument(Document: DocumentID)* kann sich ein Client das XML-Dokument mit der über den Parameter *Document* spezifizierten *DocumentID* von der zentralen Datenbasis herunterladen. Der Server leitet also den Aufruf dieses Dienstes direkt an das Interface *XMLDatabaseInterface* weiter. Gleichzeitig registriert sich der Client mit dem Aufruf der Funktion für den Empfang von Benachrichtigungen über das Interface *ClientConnectInterface*.

Mit Hilfe der Funktion *update(Changes: XMLData)* lädt der Client die Änderungen, die er an einem oder mehreren XML-Dokumenten vorgenommen hat, auf den Server hoch. Ein derartiger Upload erfolgt immer mit dem Übergang einer Subaktion erster Ebene, die eine Änderungsoperation enthält, in den Zustand *Completed*. Aus den übertragenen *Changes* werden dann die Objektversionen und die Einträge für die Log-Bücher gewonnen. Des Weiteren erfolgt eine Übermittlung der *Changes* über die Methode*update(Changes: XMLData)* des Interfaces *ClientConnectInterface* an alle für die betroffenen XML-Dokumente registrierten Clients. Bei erfolgreicher Ausführung der Funktion liefert sie *true*, ansonsten *false* zurück.

Die Klasse Server

Die Klasse *Server* implementiert, wie bereits erwähnt, das Interface *ServerConnectInterface*. Die über diese Schnittstelle kommenden Anfragen leitet sie an die entsprechenden Klassen weiter. Sie kommuniziert mit der XML-Datenbank über das Interface *XMLDatabaseInterface* und initiiert den Nachrichtenversand an die Clients über das Interface *ClientConnectInterface*.

Das Interface XMLDatabaseInterface

Über diese Schnittstelle leitet die Klasse *Server* Anfragen an die XML-Datenbank weiter. Da das Interface abhängig vom verwendeten Datenbanksystem ist, sind hier beispielhaft nur zwei Methoden aufgeführt, die für die Funktionsweise des Gesamtsystems unbedingt erforderlich sind.

Mit Hilfe der Methode *getDocument(Document: DocumentID)* wird eine XML-Dokument, welches die über den Parameter *Document* übergebene *DocumentID* besitzt, aus der Datenbank geholt.

Über die Methode *save(Node: XMLData)* wird ein XML-Knoten dauerhaft in der zentralen Datenbank gespeichert. Die Methode wird immer dann aufgerufen, wenn die Funktion *searchCommitted()* der Klasse *Logbookmanager true* liefert. In diesem Fall befinden sich alle Aktionen, die den Knoten betreffen, im Zustand *Committed*, was wiederum bedeutet, dass sich die Nutzer auf eine finale Version des Knotens geeinigt haben. Nach dem erfolgreichen Datenbankzugriff liefert die Methode *true*, ansonsten *false* zurück.

5.2.2 Der Client

Der Client stellt das Bindeglied zwischen Anwendung und Server dar. Er nimmt über das Interface *ClientInterface* Anfragen vom Anwender entgegen und leitet diese über das *ServerConnectInterface* an den Server weiter. Das Besondere am Client ist jedoch, dass er eine lokale XML-Datenbank besitzt, die über das Interface *XMLDatabaseInterface* angesprochen wird. Änderungen, die ein Nutzer an einem XML-Dokument vornimmt, werden zunächst lokal in dieser Datenbank abgelegt und erst beim *Complete* der dazugehörigen Subaktion erster Ebene an den Server übertragen, der diese dann auch an andere Clients weiterleitet.

In Abbildung 5.7 ist das Klassendiagramm des Clients dargestellt. Nachfolgend soll der Client näher erläutert werden.

Das Interface ClientInterface

Dies ist die Schnittstelle zwischen Anwendung und Client. Sie wird durch die Klasse *Client* implementiert. Das Interface stellt eine Reihe von Methoden zur Verfügung, die nachfolgend näher erläutert werden sollen.

Die Methode *startSession(Documents: DocumentIDList)* führt zum Start einer Wurzelaktion. Innerhalb dieser Methode werden also zunächst die serverseitig angebotenen Funktionen *login()* und *begin()* aufgerufen. Des Weiteren werden durch das mehrmalige Aufrufen von *get-*

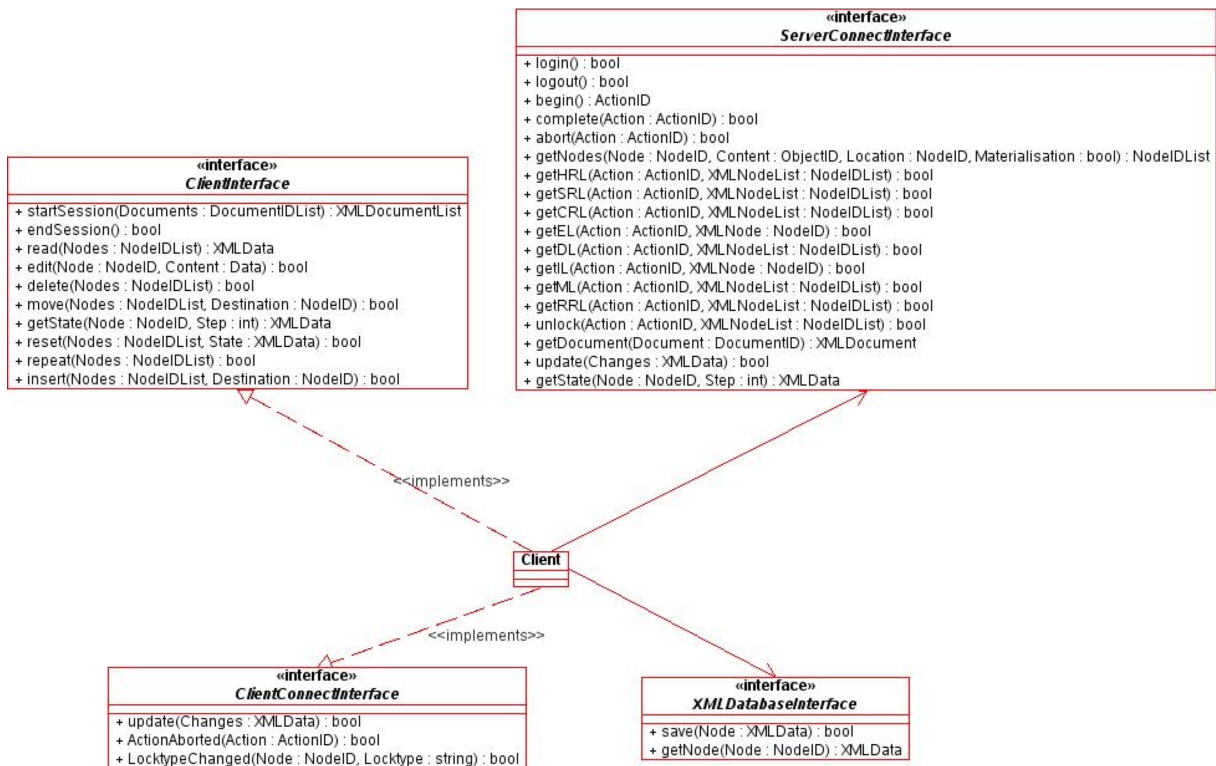

Abbildung 5.7: Klassendiagramm des Clients

Document(Document: DocumentID) die über den Parameter *Documents* spezifizierten XML-Dokumente in Listenform vom Server heruntergeladen.

Die Methode *endSession()* führt zum Aufruf der serverseitig angebotenen Methode *complete(Action: ActionID)*, wodurch die Wurzelaktion in den Zustand *Completed* versetzt wird. Darauf folgt ein Aufruf von *logout()*, wodurch der Client die Verbindung zum Server trennt. Wurden beide Teilaktionen erfolgreich ausgeführt, liefert *endSession() true*, ansonsten *false* zurück.

Die Methode *read(Nodes: NodeIDList)* bildet die in Abschnitt 4.1 erläuterte Operation *Read* auf das Aktionsmodell ab. Dies geschieht unter Berücksichtigung sowohl des strengen Aufbaus einer Operationsfolge (Siehe Gleichungen 4.5 und 4.6.) als auch des strengen Aufbaus des geschachtelten Aktionsmodells (Siehe den Beginn des Abschnitts4.2.). Innerhalb der Funktion werden die serverseitig angebotenen Methoden *begin()*, *complete(Action: ActionID)* und *abort(Action: ActionID)* zur Einhüllung der Operation in das geschachtelte Aktionsmodell benutzt. (Wird die Funktion *read(...)* für eine Knotenmenge aufgerufen, so kommt es zum rekursiven Aufruf der Funktion für die einzelnen Knoten.) Über die Methoden *getHRL(Action: ActionID, XMLNodeList: NodeIDList)*, *getSRL(Action: ActionID, XMLNodeList: NodeIDList)* und *getCRL(Action: ActionID, XMLNodeList: NodeIDList)* werden die benötigten Sperren beim Server angefordert und mit *unlock(Action: ActionID, XMLNodeList: NodeIDList)* die nicht mehr benötigten freigegeben. Das Auslesen der Knoten erfolgt jedoch auf der lokalen Datenbank mit Hilfe der Methode *getNode(Node: NodeID)* des Interfaces *XMLDatabaseInterface*.

Die Methode *edit(Node: NodeID, Content: Data)* bildet die in Abschnitt 4.1 erläuterte Operation *Edit* auf das Aktionsmodell ab. Mit Hilfe dieser Funktion wird der Inhalt des über *Node*

spezifizierten Attribut- oder Textknotens durch den Inhalt, der über *Content* übergeben wird, ersetzt. Hierbei ist zu bemerken, dass diese Operation zunächst auf der lokalen Datenbank ausgeführt wird. (Über die Funktion *getNode(Node: NodeID)* des Interfaces *XMLDatabaseInterface* wird dabei zunächst der Knoten aus der Datenbank geholt und mit *saveNode(Node: XMLData)* der bearbeitete Knoten wieder in der Datenbank abgelegt.) Nach Ausführung der Edit-Operation und dem Übergang der zugehörigen Subaktion erster Ebene in den Zustand *Completed* werden die Änderungen über die Methode *update(Changes: XMLData)* des Interfaces *ServerConnectInterface* an den Server geschickt. Wurde alles erfolgreich abgeschlossen, so liefert die Funktion *edit(Node: NodeID, Content: Data) true*, ansonsten *false* zurück. Selbstverständlich werden bei der Ausführung der Methode auch die serverseitig angebotenen Methoden *begin()*, *complete(Action: ActionID)*, *abort(Action: ActionID)*, *getEL(Action: ActionID, XMLNode: NodeID)* und *unlock(Action: ActionID, XMLNodeList: NodeIDList)* genutzt.

Die Methode *delete(Nodes: NodeIDList)* bildet die in Abschnitt 4.1 erläuterte Operation *Delete* auf das Aktionsmodell ab. Das Löschen der Knoten findet zunächst wieder in der lokalen Datenbank statt. Erst mit dem Übergang der Subaktion erster Ebene in den Zustand *Completed* werden die Änderungen über die Methode *update(Changes: XMLData)* des Interfaces *ServerConnectInterface* an den Server übermittelt. Zur Abbildung der Operation auf das Aktionsmodell werden wieder die Methoden *begin()*, *complete(Action: ActionID)* und *abort(Action: ActionID)* des Interfaces *ServerConnectInterface* genutzt. (Wird der Funktion *delete(...)* eine Knotenmenge übergeben, so wird die Methode für die einzelnen Knoten rekursiv aufgerufen.) Die benötigten Sperren werden über die Methode *getDL(Action: ActionID, XMLNodeList: NodeIDList)* angefordert und mit *unlock(Action: ActionID, XMLNodeList: NodeIDList)* wieder freigegeben. Nach erfolgreicher Ausführung aller Verarbeitungsschritte liefert die Methode *delete(Nodes: NodeIDList) true*, ansonsten *false* zurück.

Die Methode *move(Nodes: NodeIDList, Destination: NodeID)* bildet die in Abschnitt 4.1 erläuterte Operation *Move* auf das Aktionsmodell ab. Auch diese Änderungsoperation wird zunächst in der lokalen Datenbank des Clients ausgeführt. Erst mit dem Übergang der dazugehörigen Subaktion erster Ebene in den Zustand *Completed* werden die Änderungen an den Server übertragen. Zur Abbildung der Operation auf das Aktionsmodell werden wieder die entsprechenden Operationen des Interfaces *ServerConnectInterface* genutzt. Die benötigten Verschiebesperren werden mit *getML(Action: ActionID, XMLNodeList: NodeIDList)* und die Einfügesperre für den über *Destination* spezifizierten Knoten wird mit *getIL(Action: ActionID, XMLNode: NodeID)* angefordert. Wurde die Methode *move(Nodes: NodeIDList, Destination: NodeID)* erfolgreich ausgeführt, liefert sie *true*, ansonsten *false* zurück.

Die Methode *reset(Nodes: NodeIDList, State: XMLData)* bildet die in Abschnitt 4.1 erläuterte Operation *Reset* auf das Aktionsmodell ab. Neben den betroffen Knoten muss über die Methode *getState(Node: NodeID, Step: int)* des Interfaces *ServerConnectInterface* vorher der Zustand spezifizierte werden, auf den die Knoten rückgesetzt werden sollen. Danach wird die Methode zunächst wieder lokal ausgeführt. Hierbei müssen natürlich auch die Knoten ermittelt werden, für die ein Reset ebenfalls erforderlich ist, damit die Struktur des XML-Dokumentes erhalten bleibt, die aber vom Nutzer nicht explizit bestimmt worden sind. Diese Knoten werden mit der Methode *getNodes(Node: NodeID, Content: ObjectID, Location: NodeID, Materialisation: bool)* des Interfaces *ServerConnectInterface* ermittelt. Nach dem Übergang der zur Operation dazugehörigen Subaktion erster Ebene in den Zustand *Completed* werden die Änderungen an den Server übertragen. Zur Abbildung der Operation auf das Aktionsmodell werden wieder die besagten Methoden *begin()*, *complete()* und im Falle eines Abbruchs auch die Metho-

de *abort()* des Interfaces *ServerConnectInterface* genutzt. Die benötigten Sperren werden mit *getRRL(Action: ActionID, XMLNodeList: NodeIDList)* angefordert. Ist alles erfolgreich ausgeführt worden, liefert die Methode *reset(Nodes: NodeIDList, State: XMLData) true*, ansonsten *false* zurück.

Mit Hilfe der Methode *repeat(Nodes: NodeIDList)* wird die in Abschnitt 4.1 erläuterte Operation *Repeat* auf das Aktionsmodell abgebildet. Diese Operation macht bekanntlich ein zuvor erfolgtes *Reset* rückgängig. Auch bei dieser Operation müssen die zwar nicht vom Nutzer spezifizierten jedoch zur Erhaltung der Struktur des XML-Dokumentes notwendigen Knoten durch die Methode *getNodes(Node: NodeID, Content: ObjectID, Location: NodeID, Materialisation: bool)* ermittelt werden. Zur Abbildung der Operation auf das Aktionsmodell und zur Anforderung der Sperren werden die entsprechenden Methoden des Interfaces *ServerConnectInterface* genutzt. Ist alles erfolgreich ausgeführt worden, liefert die Methode *reset(Nodes: NodeIDList, State: XMLData) true*, ansonsten *false* zurück.

Die Methode *insert(Nodes: NodeIDList, Destination: NodeID)* bildet die in Abschnitt 4.1 erläuterte Operation *Insert* auf das Aktionsmodell ab. Auch diese Änderungsoperation wird zunächst in der lokalen Datenbank des Clients ausgeführt. Erst mit dem Übergang der dazugehörigen Subaktion erster Ebene in den Zustand *Completed* werden die Änderungen an den Server übertragen. Zur Abbildung der Operation auf das Aktionsmodell werden wieder die entsprechenden Operationen des Interfaces *ServerConnectInterface* genutzt. Die benötigte Einfügesperre für den über *Destination* spezifizierten Knoten wird mit *getIL(Action: ActionID, XMLNode: NodeID)* angefordert. Wurde die Methode *insert(Nodes: NodeIDList, Destination: NodeID)* erfolgreich ausgeführt, liefert sie *true*, ansonsten *false* zurück.

Das Interface ClientConnectInterface

Über dieses Interface empfängt der Client Nachrichten, die vom Server versendet wurden. Nachfolgend sollen die angebotenen Methoden kurz erläutert werden.

Über die Methode *update(Changes: XMLData)* wird der Client vom Server über Änderungen an einem XML-Dokument informiert. Diese Änderungen werden über den Parameter *Changes* in Form von XML-Daten an die Methode übergeben. Der Client, der diese Funktion implementiert, kann auf die Änderungen entsprechend reagieren. Ist die Benachrichtigung erfolgreich durchgeführt worden, gibt die Methode *true*, ansonsten *false* an den Server zurück.

Mit Hilfe der Methode *ActionAborted(Action: ActionID)* wird der Client, zum dem die über den Parameter *Action* übergebene Aktion gehört, über den serverseitigen Abbruch eben dieser benachrichtigt. Solch eine Situation kann z.B. auftreten, wenn ein anderer Client ein Reset für den Knoten aufruft, an dem die besagte Aktion gerade eine Änderung durchführt. Ist die Benachrichtigung des Clients erfolgreich durchgeführt worden, gibt die Methode *true*, ansonsten *false* an den Server zurück.

Mit Hilfe der Methode *LocktypeChanged(Node: NodeID, Locktype: string)* wird der Client, der eine Sperre auf dem über *Node* übergebenen Knoten hat, darüber informiert, dass der Typ dieser Sperre geändert wurde. Solch eine Situation tritt z.B. auf, wenn zwei Clients jeweils eine CRL auf einem Knoten haben und einer der beiden Clients diese CRL zur EL verschärft. Die CRL des anderen Clients muss dann zur SRL abgeschwächt werden. Über diese Änderungen wird der Client dann mit der besagten Methode informiert. Ist die Benachrichtigung erfolgreich

durchgeführt worden, gibt die Methode *true*, ansonsten *false* an den Server zurück.

Die Klasse Client

Die Klasse *Clients* implementiert, wie bereits erwähnt, die Schnittstellen *ClientConnectInterface* und *ClientInterface*. Dadurch bildet sie einerseits die anwendungsnahen Methoden des Interfaces *ClientInterface* auf systemnahe Funktionen des Interfaces *ServerConnectInterface* ab. Andererseits empfängt sie über das Interface *ClientConnectInterface* Nachrichten vom Server und reagiert entsprechend darauf. Weiterhin ermöglicht sie den Zugriff auf die lokale Datenbank über das Interface *XMLDatabaseInterface*.

Das Interface XMLDatabaseInterface

Über diese Schnittstelle leitet die Klasse *Client* Anfragen an die persistente XML-Datenbasis weiter. Da das Interface abhängig vom verwendeten Datenbanksystem ist, sind hier beispielhaft nur zwei Methoden aufgeführt, die für die Funktionsweise des Gesamtsystems unbedingt erforderlich sind.

Über die Methode *save(Node: XMLData)* wird ein XML-Knoten in der lokalen Datenbank gespeichert. Nach dem erfolgreichen Datenbankzugriff liefert die Methode *true*, ansonsten *false* zurück.

Mit Hilfe der Methode *getNode(Node: NodeID)* wird ein XML-Knoten aus der Datenbank geholt und an den Aufrufer übergeben.

5.2.3 Beispielhafte Darstellung der Kommunikation zwischen den Klassen

In diesem Abschnitt soll die Kommunikation auf Client- und Serverseite, die Interaktion des Nutzers mit dem Client sowie der Nachrichtenaustausch zwischen Client und Server mit Hilfe eines Sequenzdiagramms beispielhaft dargestellt werden. Folgendes Szenario soll als Beispiel dienen:

- Es existieren zwei Clients *A* und *B*, die mit einem Server kommunizieren.
- Die Aktivitäten des Clients *B* sollen nicht berücksichtigt werden. Hier soll lediglich gezeigt werden, welche Nachrichten er vom Server empfängt.
- Der Client *A* führt folgende Operationsfolge aus: $OF = Read(X)Edit(X)$
- Aus Gründen der Übersicht wird auf die Instantiierung der Klassen sowie auf detaillierte Parameterlisten bei den Funktionsaufrufen verzichtet.

Abbildung 5.8 zeigt nun das Sequenzdiagramm dieses Szenarios. Nachfolgend soll es kurz erläutert werden:

- Mit dem Aufruf von *startSession(...)* beginnt ein Nutzer, dargestellt durch die fiktive Klasse *User*, seine Arbeiten. (Das Objekt *A* repräsentiert den zum Nutzer gehörenden *Client*.) Dabei erfolgt zunächst ein *login()* beim Server. Anschließend werden die für die Arbeiten benötigten XML-Dokumente aus der zentralen Datenbasis, dargestellt durch die Klasse

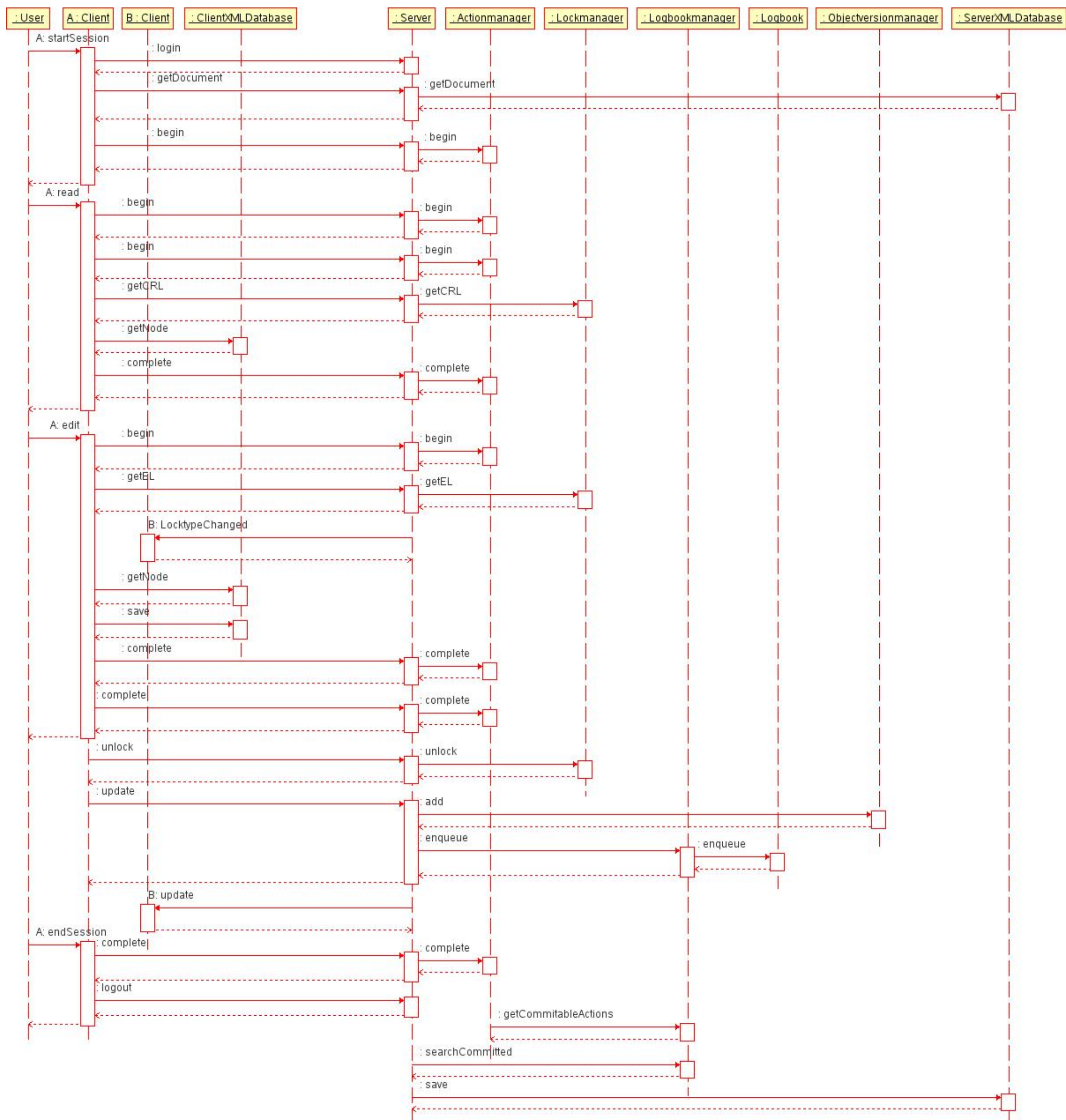

Abbildung 5.8: Sequenzdiagramm für ein Beispielszenario

ServerXMLDatabase, die über das Interface *XMLDatabaseInterface* angesprochen wird, an den *Client* übermittelt. Gleichzeitig registriert sich der Client *A* durch den Aufruf dieser Methode (*getDocument(...)*) für den Nachrichtenempfang. Nun wird die Wurzelaktion durch einen Aufruf von *begin()* gestartet und beim *Actionmanager* registriert.

- Als nächstes führt der Nutzer das besagte $Read(X)$ durch einen Aufruf der über das Interface *ClientInterface* angebotenen Methode *read(...)* aus. Dies führt zu einem zweimaligen serverseitigen Aufruf der Methode *begin()* über das *ServerConnectInterface* durch die Klasse *Client*. Dadurch wird die Subaktion der ersten Ebene und die Subaktion der zweiten Ebene gestartet und beim *Actionmanager* registriert. Danach wird mit Hilfe der Me-

thode *getCRL(...)* die benötigte Lesesperre über das *ServerConnectInterface* beim *Lockmanager* angefordert. Nach Erhalt der Lesesperre wird der zu lesende Knoten *X* mit der *getNode(...)*-Methode des Interfaces *XMLDatabaseInterface* aus der lokalen Datenbank des Clients *A* (*ClientXMLDatabase*) geholt und ausgelesen. Anschließend wird die Subaktion der zweiten Ebene durch einen Aufruf von *complete()* in den Zustand *Completed* überführt.

- Nun führt der Nutzer die Operation $Edit(X)$ durch einen Aufruf der über das Interface *ClientInterface* angebotenen Methode *edit(...)* aus. Dies führt zu einem serverseitigen Aufruf der Methode *begin()* über das *ServerConnectInterface* durch die Klasse *Client*. Dadurch wird die zweite Subaktion der zweiten Ebene gestartet und beim *Actionmanager* registriert. Danach wird mit Hilfe der Methode *getEL(...)* die benötigte Editiersperre über das *ServerConnectInterface* beim *Lockmanager* angefordert. (Dadurch wird die vorher erhaltene Lesesperre verschärft.) An dieser Stelle wird angenommen, dass der Client *B* eine inhaltliche Lesesperre auf dem Knoten *X* besitzt. Diese wird nun in eine strukturelle Lesesperre umgewandelt. Diese Änderung des Sperrtyps wird dem Client *B* über die Methode *LocktypeChanged(...)* des Interfaces *ClientConnectInterface* mitgeteilt. Nach Erhalt der Editiersperre wird der zu lesende Knoten *X* mit der *getNode(...)*-Methode des Interfaces *XMLDatabaseInterface* aus der lokalen Datenbank des Clients *A* (*ClientXMLDatabase*) geholt und bearbeitet. Nach Beendigung der Arbeiten werden die Änderungen in der lokalen Datenbank durch einen Aufruf von *save(...)* gespeichert. Anschließend wird die Subaktion der zweiten Ebene durch einen Aufruf von *complete()* in den Zustand *Completed* überführt. Da nach einer Änderungsoperation keine weitere folgen darf, wird auch die Subaktion erster Ebene durch einen Aufruf von *complete()* in den Zustand *Completed* versetzt.
- Nach dem Übergang der Subaktion der ersten Ebene in den Zustand *Completed* wird durch einen Aufruf von *unlock(...)* die Editiersperre auf *X* freigegeben und die Änderungen durch einen Aufruf der Methode *update(...)* des Interfaces *ServerConnectInterface* an den Server übermittelt. Auf der Seite des Servers führt dieses Update mit Hilfe der Methode *add(...)* zum Anlegen einer neuen Objektversion des Knotens *X* im *Objectversionmanager*. Dabei muss natürlich auch die Historie des Knotens *X* weitergeschrieben werden. Dies geschieht durch einen Aufruf von *enqueue(...)* im *Logbookmanager*, der zu einem Aufruf derselbigen Methode im entsprechenden *Logbook* des Knotens *X* führt.
- Nun wird der Client *B* durch einen Aufruf von *update(...)* über das Interface *ClientConnectInterface* durch den Server über die Änderungen informiert.
- Anschließend beendet der Nutzer seine Arbeiten durch einen Aufruf der Methode *endSession()* des Interfaces *ClientInterface*. Dadurch wird die Wurzelaktion durch einen Aufruf von *complete()* in den Zustand *Completed* versetzt und der Client *A* beim Server durch einen Aufruf von *logout()* abgemeldet.
- Danach ruft der Actionmanager die Methode *getCommitableActions()* des *Logbookmanagers* auf und bekommt von ihm alle IDs der Aktionen zurückgeliefert, die bereit sind, ein *Commit* auszuführen. Für diese Aktionen ruft der Actionmanager dann die Methode *commit()* auf.
- Abschließend ruft der Server die Methode *searchCommitted()* des *Logbookmanagers* auf und speichert durch einen Aufruf von *save(...)* alle durch ein *Commit* bestätigten Objektänderungen persistent.

Zum Abschluss dieses Kapitels soll noch einmal auf den in Abschnitt 1.1.1 beschriebenen speziellen Anwendungsfall reflektiert werden. Die in diesem Kapitel vorgeschlagene Systemarchitektur entspricht vom Kernprinzip her der des Workgroup-Betriebs des IOSONO-Systems. Es existiert ein zentraler Server, der alle notwendigen Dienste zur Verfügung stellt und Halter der zentralen Datenbasis ist. Er übernimmt die Behandlung von Transaktionsfehlern (Logbookmanager), die Synchronisation der Zugriffe auf die gemeinsam genutzten XML-Dokumente (Lockmanager), die Aktionsverwaltung (Actionmanager) und den Nachrichtenversand an die Clients. Daneben existiert eine Vielzahl von Clients, die mit diesem Server verbunden sind. Diese besitzen einen lokalen Cache, der die zu bearbeitenden XML-Dokumente enthält. Jede Operation wird zunächst lokal in diesem Speicher ausgeführt und erst nach ihrer Beendigung (*Complete*) werden die Ergebnisse an Server geschickt. Dieser verteilt sie dann an alle anderen Clients.

Über die Schnittstelle *ClientInterface* werden den Designern die für ihre Arbeit notwendigen Operationen angeboten. Mit *startSession(...)* erfolgt beispielsweise das Auschecken der für die Arbeiten notwendigen XML-Dokumente beim Server und der Start der Wurzelaktion. Mit *endSession()* signalisiert der Designer, dass er sein Arbeiten beendet hat. Dadurch wird die Wurzelaktion in den Zustand *Completed* versetzt. Will der Designer innerhalb eines Dokumentes arbeiten, so benutzt er die Methoden *read(...), edit(...), insert(...), delete(...), move(...), reset(...) und repeat(...)*. Durch diese werden dann der Start der entsprechenden Subaktionen, die Anforderung und die Freigabe der benötigten Sperren, die Ausführung der eigentlichen Operation sowie die Beendigung der entsprechenden Subaktionen vorgenommen. Diese Vorgänge sind in dem Sequenzdiagramm 5.8 exemplarisch für die Ausführung der Operationsfolge $Read(X)$ $Edit(X)$ dargestellt, wobei *X* ein Attribut- oder Textknoten ist.

Gegenüber der in Abschnitt 1.1.1 beschriebenen Architektur bietet die in diesem Kapitel vorgeschlagene jedoch einige Vorteile:

- Die lange Transaktion vom Beginn bis zum Ende der Arbeiten der Designer wird unterteilt. Somit geht beim Scheitern einer Operation nicht der komplette Arbeitsfortschritt verloren.
- Objektänderungen sind schon vor dem Ende der Arbeiten eines Designers sichtbar und werden an die anderen Designer übermittelt. Dadurch ist ein kooperatives Arbeiten möglich.
- Es kann auf ein CVS verzichtet werden. Der Grund wurde bei der Vorstellung des Fehlerbehandlungsmodells und dessen Eigenschaften (Abschnitte 4.4 und 4.5.7) erläutert.

6 Zusammenfassung und Ausblick

Gegenstand dieser Arbeit war die Entwicklung eines kooperativen Transaktionsmodells, eines dazugehörigen Sperrprotokolls und eines Fehlerbehandlungsmodells. Ausgehend von den Problemen eines speziellen Anwendungsfall, dem Workgroup-Betrieb im IOSONO-System, wurden Kriterien formuliert, die ein vollständiges Transaktionskonzept erfüllen muss, um als Lösung für diese Probleme in Frage zu kommen. Neben der Unterstützung langer Transaktionen waren das, die Verwirklichung des Kooperationsprinzips, der Grad der Nebenläufigkeit, die Komplexität des Fehlerbehandlungsmodells sowie die Anwenderunabhängigkeit.

Die Bewertung bereits existierender Transaktionskonzepte hat ergeben, dass keines dieser Modelle alle Kriterien im vollen Umfang erfüllt. Jedoch hat sich ein Konzept, nämlich das der geschachtelten dynamischen Aktionen für kooperative Anwendungen, als Grundlage für weitere Betrachtungen herauskristallisiert. Ein wesentlicher Hauptgrund dafür war, dass dynamische Aktionen die Dauerhaftigkeit garantieren und somit eine Vermeidung des Einsatzes von Kompensationstransaktionen zur Fehlerbehandlung ermöglichen.

Aufbauend auf den geschachtelten dynamischen Aktionen für kooperative Anwendungen wurde ein neues Aktionsmodell entwickelt, was speziell auf die Anforderungen des IOSONO-Systems zugeschnitten ist. Dabei wurden zunächst die möglichen Operationen aufgeschlüsselt und die geschachtelten dynamischen Aktionen für kooperative Anwendungen auf die Ausführung dieser Operationen zugeschnitten. Neben den üblichen Methoden zur Bearbeitung von XML-Dokumenten wurden auch zwei spezielle Operationen, nämlich das Reset und Repeat eingeführt. Mit ihnen ist es den Designern möglich, auf Konflikte, die aufgrund der fehlenden Serialisierbarkeit entstehen können, zu reagieren. Dabei können sie Operationen anderer Designer rückgängig machen bzw. wiederholen, ohne diese Operationen kennen zu müssen. Anschließend wurde ein Sperrprotokoll vorgeschlagen, welches zum einen die nebenläufige Ausführung der Operationen synchronisiert und zum anderen ein kooperatives Arbeiten der Designer ermöglicht.

Im weiteren Verlauf der Arbeit wurde das entwickelte Fehlerbehandlungsmodell zur Behandlung von Aktionsfehlern vorgestellt. Dieses basiert auf einem Objektversionensystem in Kombination mit einem Log-Buch. Neben der relativ einfachen Behandlung von Transaktionsabbrüchen ohne die Eingriffe des Nutzers bietet es den Vorteil, dass nachdem alle Designer ihre Arbeiten abgeschlossen haben, eine finale Version von jedem benutzen Objekt feststeht. Dadurch kann auf ein CVS verzichtet werden.

Die Bewertung des vorgeschlagenen Modells hat ergeben, dass es alle gestellten Anforderungen erfüllt. Durch die Schachtelung begegnet es dem Problem der langen Transaktionen. Es können also einzelne Operationen abgebrochen werden, ohne das dies den Verlust des gesamten Arbeitsfortschritts zur Folge hat. Durch das verwendete Sperrprotokoll und die frühzeitigen Objektfreigaben erfüllt das Modell nicht nur das Kooperationsprinzip im vollen Umfang sondern bietet auch einen hohen Grad der Nebenläufigkeit der Arbeiten verschiedener Designer.

Das entwickelte Fehlerbehandlungsmodell ist von geringer Komplexität, da es u.a. auf den Einsatz von Kompensationstransaktionen verzichtet. Dadurch ist eine automatische Behandlung von Fehlern ohne Eingriffe des Designers möglich.

Neben der Erfüllung der gestellten Anforderungen bietet es noch einige weiter Vorteile, die es besonders attraktiv machen. Zu diesen Vorteilen gehört beispielsweise, dass es nicht erforderlich ist, Kooperationsgruppen zu bilden, wie dies bei den geschachtelten dynamische Aktionen für kooperative Anwendungen erforderlich ist. In dem vorgeschlagenen Modell kann Kooperation zwischen beliebigen Partnern stattfinden. Ein weiterer Vorteil ist, dass eine Möglichkeit (Reset/Repeat) zur Behandlung von Konflikten, die infolge der fehlenden Serialisierbarkeit auftreten können, vorgesehen ist. Außerdem wird dadurch, dass auf ein CVS verzichtet werden kann, viel Aufwand für die Designer eingespart.

Im letzten Teil der Arbeit wurde eine mögliche Systemarchitektur beschrieben. Im Wesentlichen werden alle notwendigen Dienste durch den Server vorgenommen. Auf Clientseite erfolgt lediglich eine Abbildung der Operationen auf das Aktionsmodell, sowie die lokale Ausführung der Operationen. Dies bietet den Vorteil, dass einerseits nur die Ergebnisse abgeschlossener (*completed*) Operationen auf den Server gelangen und andererseits der Ausfall eines Clients keine Auswirkungen auf die Funktionsfähigkeit des Gesamtsystems hat.

In zukünftigen Arbeiten gilt es herauszufinden, wie sich das System in der Praxis bewährt. Es müssen also umfangreiche Tests durchgeführt werden. Ein besonderer Augenmerk sollte dabei auf die Netzwerkauslastung gelegt werden. Die frühe Freigabe von Ergebnissen für alle Designer (nach einer Änderungsoperation) führt zu einem hohen Datenverkehr, da alle Änderungen zu den einzelnen Clients propagiert werden müssen. Ein weiterer Aspekt, der in dieser Arbeit nicht betrachtet wurde, ist der Umgang mit Systemausfällen. In weiteren Arbeiten sollte also die Entwicklung einer Recovery-Komponente für das vorgeschlagene System erfolgen.

Literaturverzeichnis

[BOH^{+}92] Alejandro P. Buchmann, M. Tamer Ozsu, Mark Hornick, Dimitrios Georgakopoulos, and Frank A. Manola. A transaction model for active distributed object systems. In *Database Transaction Models for Advanced Applications*, pages 123–158. 1992.

[DOM] http://www.w3.org/DOM/.

[GFJK03] Paul Greenfield, Alan Fekete, Julian Jang, and Dean Kuo. Compensation is Not Enough. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 232, Washington, DC, USA, 2003. IEEE Computer Society.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 249–259. ACM Press, 1987.

[Hau05] Michael Peter Haustein. Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery. In *BTW*, pages 265–284, 2005.

[HH03] Michael Peter Haustein and Theo Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In *ADBIS*, pages 88–102, 2003.

[HS07] K. Hose and K. Sattler. Cooperative Data Management for XML Data. In *Dexa*, pages 308–318, 2007.

[IOS] http://www.iosono-sound.com/.

[Kru97] Armin Kruse. Ein kooperatives Sperrverfahren für geschachtelte dynamische Aktionen — Diplomarbeit an der Universität Bonn. März 1997.

[Moc95] Michael Mock. *Aktionsunterstützung für verteilte, kooperative Anwendungen — Konzept und Realisierung*. GMD-Bericht Nr. 250, R. Oldenbourg Verlag, 1995.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[NM95] Edgar Nett and Michael Mock. How to Commit Concurrent, Non-Isolated Computations. *ftdcs*, 00:0343, 1995.

[NS92] Edgar Nett and Ralf Schumann. Supporting fault-tolerant distributed computations under real-time requirements. *Comput. Commun.*, 15(4):252–260, 1992.

[NW94] Edgar Nett and Beatrice Weiler. Nested dynamic actions - how to solve the fault containment problem in a cooperative action model. In *Symposium on Reliable Distributed Systems*, pages 106–115, 1994.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908. ACM, 2004.

[RMH+94] Norbert Ritter, Bernhard Mitschang, Theo Härder, Michael Gesmann, and Harald Schöning. Capturing Design Dynamics the Concord Approach. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 440–451, Washington, DC, USA, 1994. IEEE Computer Society.

[SHK05] G. Saake, A. Heuer, and K.Sattler. *Datenbanken: Implementierungstechniken, 2. Auflage*. mitp-Verlag, Januar 2005.

[SST97] G. Saake, I. Schmitt, and C. Türker. *Objektdatenbanken — Konzepte, Sprachen, Architekturen*. International Thomson Publishing, Bonn, 1997.

[WFS] http://www.idmt.fraunhofer.de/de/projekte_themen/wellenfeldsynthese.htm.

[WR92] Helmut Wächter and Andreas Reuter. The ConTract model. pages 219–263, 1992.

[XML] http://www.w3.org/TR/xml/.

Abbildungsverzeichnis

1.1 Beispiel für einen Szenegraph . . . 2
1.2 Architektur des IOSONO-Systems im Workgroup-Betrieb . . . 3

2.1 Zustände einer klassischen Transaktion . . . 6
2.2 Ausschnitt aus einem XML-Dokument . . . 10
2.3 DOM-Baum des XML-Dokumentausschnitts aus Abbildung 2.2 . . . 11
2.4 XML-Baum, aufgebaut nach der DOM-Spezifikation . . . 12
2.5 XML-Baum, aufgebaut nach der taDOM-Spezifikation . . . 13
2.6 XML-Baum, aufgebaut nach der in [HS07] vorgeschlagenen Spezifikation . . . 13
2.7 Vergabe der DeweyIDs für den XML-Baum aus Abbildung 2.6 . . . 14
2.8 Erweiterung des XML-Baumes aus Abbildung 2.6 . . . 15

3.1 Ein Beispiel für einen Transaktionsbaum . . . 17
3.2 Zustände einer dynamischen Aktion . . . 25

4.1 Ein Beispiel-XML-Baum . . . 35
4.2 Die Abbildung der Beispiel-Operationsfolge auf das Modell . . . 37
4.3 Ein Beispiel für ein Objektversionensystem . . . 45
4.4 Beispielhafter Aufbau des Log-Buches . . . 47
4.5 Erweiterung des Log-Buches zur Unterstützung des Reads . . . 50
4.6 Folge von Operationen auf einem XML-Teilbaum . . . 51
4.7 Log-Buch des Knotens E vor den Änderungen . . . 52
4.8 Log-Buch des Knotens E nach den Änderungen . . . 52
4.9 Log-Buch des Knotens H nach dem Einfügen und Verschieben . . . 53
4.10 Log-Buch des Knotens F nach dem Löschen und Verschieben . . . 54
4.11 Log-Buch des Knotens G nach den beiden Verschiebungen . . . 54
4.12 Log-Buch des Knotens E nach dem Reset der Aktionen 46 und 42 . . . 55
4.13 Beispiel für einen Abhängigkeitsgraphen . . . 62
4.14 Der Wartegraph zu Abbildung 4.13 . . . 62

5.1 Klassendiagramm des Actionmanagers . . . 70
5.2 Klassendiagramm des Lockmanagers . . . 71
5.3 Klassendiagramm des Objectversionmanagers . . . 75
5.4 Klassendiagramm des Logbooks . . . 75
5.5 Klassendiagramm des Logbookmanagers . . . 77
5.6 Klassendiagramm des Servers . . . 79
5.7 Klassendiagramm des Clients . . . 81
5.8 Sequenzdiagramm für ein Beispielszenario . . . 85

Tabellenverzeichnis

2.1 Beispiel für eine Verklemmung zweier Transaktionen T1 und T2 8

3.1 Bewertung der existierenden Modelle im Überblick 29

4.1 Beispiel für ein nicht ausführbares Repeat 31
4.2 Konfliktmatrix ohne die Operation Move 38
4.3 Konfliktmatrix für die Verschiebung - der Bearbeiter ist Teil der Verschiebung . 40
4.4 Konfliktmatrix für die Verschiebung - der Bearbeiter hat die Verschiebung im Blickfeld 40
4.5 Sperren-Konfliktmatrix ohne die ML 41
4.6 Sperren-Konfliktmatrix für die Verschiebung - der Bearbeiter ist Teil der Verschiebung 41
4.7 Konfliktmatrix für die Verschiebung - der Bearbeiter hat die Verschiebung im Blickfeld 41
4.8 Zu setzende Sperren bei unterschiedlichen Ausführungsarten der Operationen . 42
4.9 Der Lebenszyklus des Log-Buches 57

A Thesen

1. In vielen Anwendungsbereichen, wie z.B. dem Design oder dem Medienproduktionsprozess, ist das kooperative Arbeiten mehrerer Nutzer auf einem gemeinsamen Datenbestand unerlässlich.
2. Kooperativität bedeutet den Verzicht auf Serialisierbarkeit.
3. Ohne Serialisierbarkeit können Mehrbenutzeranomalien auftreten. Diese müssen behandelt werden.
4. Die Ermöglichung eines kooperativen Arbeitsprozesses stellt daher enorme Anforderungen an die Entwickler kooperativer Datenbanksysteme.
5. Der Workgroup-Betrieb des IOSONO-Systems ist ein spezieller Anwendungsfall des Medienproduktionsprozesses. Aus diesem speziellen Anwendungsfall ergeben sich die Schwerpunkte dieser Arbeit. Zu diesen gehören die Entwicklung eines geeigneten kooperativen Transaktionsmodells, eines kooperativen Sperrprotokolls, eines Fehlerbehandlungsmodells sowie einer Systemarchitektur.
6. Die Entwicklung dieser einzelnen Komponenten ist erforderlich, da gegenwärtige Transaktions-/Aktionskonzepte die Anforderungen, die aus dem Workgroup-Betrieb des IOSONO-Systems resultieren, nicht vollständig erfüllen.
7. Das entwickelte Aktionsmodell basiert auf den geschachtelten dynamischen Aktionen für kooperative Anwendungen. Es begegnet einerseits den Problemen langer Transaktionen und unterstützt andererseits auch das Kooperationsprinzip.
8. Das entwickelte Sperrprotokoll unterstützt das Kooperationsprinzip und ermöglicht einen hohen Grad der Nebenläufigkeit der Aktivitäten.
9. Das entwickelte Fehlerbehandlungsmodell ist von geringer Komplexität. Es erfordert keine Eingriffe des Anwenders. Außerdem ermöglicht es den Verzicht auf ein CVS.
10. Mit dem entwickelten Reset-Repeat-Prinzip werden Konflikte und Inkonsistenzen, die infolge der fehlenden Serialisierbarkeit auftreten können, behandelt.

Ilmenau, den 24. Januar 2008

Francis Gropengießer

B Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, sind von mir kenntlich gemacht worden.

Diese Arbeit war in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung.

Ilmenau, den 24. Januar 2008

Francis Gropengießer