Ein kooperatives Sperrverfahren für geschachtelte dynamische Aktionen

Diplomarbeit

Armin Kruse

Rheinische Friedrich-Wilhelms-Universität

Bonn

März 1997

Ein kooperatives Sperrverfahren für geschachtelte dynamische Aktionen

Diplomarbeit

Armin Kruse

Rheinische Friedrich-Wilhelms-Universität

Bonn

März 1997

Vorwort

Die vorliegende Diplomarbeit entstand im Kontext des Forschungsbereichs „Responsive Systeme“ des Instituts für Systementwurfstechnik in der *GMD – Forschungszentrum Informationstechnik*, Sankt Augustin. Sie fügt sich ein in die ausgedehnte Forschung zu Fehlertoleranz, Konsistenz und Kooperation in verteilten Systemen.

Ich möchte den Mitarbeitern dieser Abteilung danken, die stets hilfsbereit waren und mich unterstützt haben. Besonders danke ich meinen Betreuern, Priv.-Doz. Dr. Edgar Nett und Dr. Michael Mock. Sie haben ausgezeichnete Arbeitsbedingungen geschaffen und mir durch ihre vielen kritischen und konstruktiven Hinweise dabei geholfen, meine Ideen zu strukturieren und zu dieser Arbeit zu formen.

Mit dieser Diplomarbeit beende ich mein Studium der Informatik – eine Zeit, in der ich viel gelernt habe. Ich danke allen, die mich dabei unterstützt haben, und möchte in diesem Zusammenhang insbesondere die Freunde und Kommilitonen Michael Frankenberg, Frank Kurth, Roland Rosenfeld und Peter „piet“ Theisohn erwähnen.

Ein besonderer Dank gilt auch Muriel Gerhards sowie Claudia und Roland Conrad, die die Aufgabe des Korrekturlesens übernommen haben.

Bonn, im März 1997 Armin Kruse

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Benutzung verteilter Systeme ist heutzutage zur Realität geworden. Die Vernetzung der Computersysteme macht es möglich, daß unterschiedliche Benutzer und Applikationen auf dieselben Objekte zugreifen. Die Schaffung dieser technischen Voraussetzungen läßt neue Anwendungsgebiete entstehen: Während bisher die Computersysteme hauptsächlich einzelne Benutzer bei ihren Aufgaben unterstützt haben, arbeiten inzwischen immer öfter mehrere Benutzer gleichzeitig mit Computerunterstützung an gemeinsamen Projekten. Den Computern fällt dabei die Aufgabe zu, die einzelnen Benutzer mit den benötigten Informationen zu versorgen und ihre Tätigkeiten so zu koordinieren, daß ein unbehindertes Arbeiten möglich ist.

Eine wichtige Aufgabe bei der Steuerung verteilter Systeme besteht darin, die Konsistenz der gespeicherten Objekte zu erhalten, insbesondere wenn Benutzer zeitgleich auf dieselben Objekte zugreifen. Die Konsistenzerhaltung betrifft dabei zwei Bereiche: Zum einen dürfen mögliche Fehler, wie zum Beispiel der Ausfall eines Rechners oder einer Leitung, nicht zu einem inkonsistenten Zustand der Objekte führen. Zum anderen dürfen auch nebenläufige Berechnungen unterschiedlicher Benutzer keine störenden Auswirkungen aufeinander haben. Transaktionskonzepte haben sich ausgehend von Datenbank-Anwendungen als Mittel der Fehlerbehandlung und der Nebenläufigkeitskontrolle durchgesetzt. In diesen Konzepten finden die Berechnungen einzelner Benutzer jeweils innerhalb von Transaktionen statt, für die systemseitig bestimmte Eigenschaften zugesichert werden. Das klassische Transaktionskonzept hat dabei zum Ziel, jede Transaktion soweit zu isolieren, daß sie andere Transaktionen und auch deren Abbrüche nicht bemerkt. Die Isolation basiert auf dem Konsistenzkriterium der Serialisierbarkeit, welches fordert, daß nebenläufige Transaktionen stets denselben Effekt haben müssen, den sie auch bei serieller Ausführung hätten.

Auch wenn klassische Transaktionen ein ausreichendes Konzept darstellen, um viele Datenbankanwendungen wie zum Beispiel Buchungssysteme zu realisieren, zeigt sich schnell, daß sie aufgrund der geforderten Isolation von Berechnungen ungeeignet für eine Unterstützung von komplexeren Abläufen sind: In vielen Bereichen gibt es Beziehungen zwischen den Berechnungen der einzelnen Anwender. Beispielsweise betraut ein Anwender einen anderen mit der Durchführung einer Teilaufgabe oder mehrere Benutzer führen eine gemeinsame Aufgabe durch und treffen zu diesem Zweck auch gemeinsam Entscheidungen. Diese Beziehungen implizieren in der Regel einen Informationsfluß zwischen den Partnern, da oft Zwischenergebnisse ausgetauscht werden müssen. Zwischen Transaktionen findet dieser Informationsfluß dadurch statt, daß eine Transaktion Objektzustände liest, die eine andere Transaktion geschrieben hat. Eine Transaktionsverwaltung muß also in der Lage sein, diese Beziehungen zu modellieren und in bestimmten Situationen Informationsfluß zwischen laufenden Transaktionen zu erlauben. Die Zielsetzung des klassischen Transaktionskonzepts liegt aber im direkten Gegensatz dazu, weil hier mit der Isolation von Transaktionen jede Art von Informationsfluß zwischen laufenden Transaktionen verhindert werden soll.

Daher wurden erweiterte Transaktionskonzepte entwickelt, die Beziehungen zwischen einzelnen Transaktionen modellieren können. Diese Konzepte haben in der Regel sowohl die Mechanismen zur Fehlerbehandlung als auch die zur Nebenläufigkeitskontrolle danach ausgerichtet, eine bestimmte Art von Beziehungen zwischen Transaktionen zu modellieren. Die Anforderungen, die die Modellierung von Beziehungen an ein Transaktionskonzept stellt, betreffen meist nur die Nebenläufigkeitskontrolle, während die Anforderungen an die Fehlerbehandlung unabhängig von den jeweiligen Beziehungen zwischen den Transaktionen sind. Die starke Verflechtung der Mechanismen zur Fehlerbehandlung und zur Nebenläufigkeitskontrolle bei den erweiterten Transaktionskonzepten führt aber dazu, daß sich der Benutzer bei der Wahl eines Transaktionskonzepts zur Unterstützung von bestimmten Beziehungen auch auf vorgegebene Mechanismen zur Fehlerbehandlung festlegt. Diese Mechanismen stellen im Vergleich zu klassischen Transaktionen oft weitergehende Anforderungen an die Applikationen oder schränken deren zulässiges Verhalten stark ein.

Das in [Nett 1991] eingeführte Konzept der dynamischen Aktionen wählt statt dessen einen flexibleren Ansatz: Im Gegensatz zu den Transaktionskonzepten sind hier Fehlerbehandlung und Nebenläufigkeitskontrolle orthogonal zueinander und es kann daher eine korrekte Fehlerbehandlung sichergestellt werden, ohne bestimmte Eigenschaften der Nebenläufigkeitskontrolle wie beispielsweise eine Isolation der dynamischen Aktionen fordern zu müssen. Im Gegensatz zu den erweiterten Transaktionskonzepten führt die zur Unterstützung komplexerer Anwendungen notwendige Aufgabe der Isolation nicht dazu, daß für eine korrekte Fehlerbehandlung zusätzliche Anforderungen an die Applikation entstehen. Die Orthogonalität macht es somit möglich, bei sichergestellter Fehlerbehandlung durch unterschiedliche Konzepte zur Nebenläufigkeitskontrolle beliebige Arten von Beziehungen zwischen dynamischen Aktionen zu modellieren.

In dieser Arbeit wird ein Konzept auf Basis der dynamischen Aktionen vorgestellt, das die wichtigsten Arten von Beziehungen zwischen den Berechnungen unterschiedlicher Benutzer modellieren kann. Dabei treten in der praktischen Anwendung oft unterschiedliche Beziehungstypen innerhalb einer Anwendung auf, die sich fast alle zwei Hauptprinzipien zuordnen lassen: dem Kooperationsprinzip und dem Auftragsprinzip.

Kooperationsprinzip

Oft wird eine Aufgabe nicht von einem einzelnen Anwender gelöst, sondern an der Lösung sind mehrere Anwender beteiligt. Die Anwender teilen beispielsweise die Aufgabe in einzelne Abschnitte auf, die der jeweilige Anwender bearbeitet, und anschließend wird aus den Abschnitten gemeinsam die Gesamtlösung erstellt.

Ein Beispiel für das Kooperationsprinzip ist das Erstellen eines Textes durch zwei Autoren, wobei Autor A Einleitung und Schluß und Autor B den Mittelteil schreibt. Nachdem Autor A die Einleitung geschrieben hat, soll Autor B sie lesen können, um seinen Mittelteil darauf abzustimmen; das gleiche gilt auch umgekehrt, wenn Autor A zur Erstellung des Schlusses den Mittelteil von Autor B kennen muß. Nachdem alle Teile des Textes geschrieben wurden, sollen beide Autoren, nachdem sie jeweils den gesamten Text gelesen und gegebenenfalls überarbeitet haben, den Text als endgültige Fassung akzeptieren.

Wie dieses Beispiel verdeutlicht, bedeutet das Kooperationsprinzip folgendes: Zwischen den Berechnungen kooperierender Benutzer findet mehrfach Informationsfluß in beliebige Richtungen statt, allerdings gibt es auch Zeiten, in denen kein Informationsfluß möglich sein soll. Autor B soll im obigen Beispiel die Einleitung nicht während ihres gesamten Entstehens sehen, sondern nur dann, wenn der Autor A sie gezielt zur Verfügung stellt. Betrachtet man größere Systeme, wird außerdem deutlich, daß es möglich sein muß, den Informationsfluß auf den Kreis der Kooperationspartner zu beschränken. Falls die späteren Leser des Textes auch Zugriff auf den Objektspeicher haben, muß das System sie daran hindern, Zwischenversionen zu lesen, die nur für den Informationsfluß zwischen den Autoren gedacht sind. Die Leser dürfen den Text erst lesen, nachdem dieser am Ende von beiden Autoren akzeptiert wurde.

Auftragsprinzip

Eine weitere sehr häufig auftretende Beziehung unter Anwendern ist die zwischen Auftraggeber und Auftragnehmer. Dabei beauftragt ein Benutzer einen anderen mit der Lösung einer Teilaufgabe, die dieser eigenständig lösen kann. Der Auftraggeber benutzt anschließend die Ergebnisse des Auftragnehmers, um seine eigene Aufgabe zu erledigen.

Im Beispiel der Texterstellung ist die Einschaltung eines Übersetzungsbüros eine Anwendung des Auftragsprinzips: Der Autor schreibt einen Text in Deutsch, übergibt ihn dem Übersetzungbüro und bekommt einen englischen Text zurück. Der Autor kann anschließend alleine entscheiden, ob er den übersetzten Text benutzt oder verwirft und beispielsweise eine Übersetzung bei einem anderen Büro in Auftrag gibt.

Im Gegensatz zum Kooperationsprinzip ist beim Auftragsprinzip die Beziehung zwischen den Partnern nicht gleichberechtigt: Der Auftragnehmer bekommt eine Aufgabe, die er zu erfüllen hat. Dabei benötigt er keine weiteren Informationen über den Kontext dieser Aufgabe und hat auch keine Möglichkeit der Einflußnahme auf den Auftraggeber. Nachdem der Auftragnehmer die Ergebnisse an den Auftraggeber zurückgegeben hat, liegt die Kontrolle über diese Ergebnisse alleine beim Auftraggeber. Es findet also beim Erteilen des Auftrags einmal ein Informationsfluß von Auftraggeber zum Auftragnehmer statt und bei Beendigung des Auftrags erfolgt ein Informationsfluß in die Gegenrichtung.

Eine andere wichtige Eigenschaft des Auftragsprinzips besteht darin, daß grundsätzlich die Existenz des Auftragnehmers vor fremden Anwendern verborgen werden soll. Im Übersetzungsbeispiel ist es beispielsweise nur wichtig, daß der Autor einen englischen Text abliefert. Ob er den Text selbst in Englisch schreibt oder einen Übersetzer beauftragt, ist für den späteren Leser unerheblich. Insbesondere darf niemand außer dem Autor den vom Übersetzungsbüro zurückgegebenen englischen Text sehen, da der Autor schließlich noch Änderungen am Text machen oder ihn ganz verwerfen könnte und nur die vom Autor endgültig akzeptierte Version für die Allgemeinheit gedacht ist. Selbst wenn das Übersetzungsbüro seine Tätigkeit abbricht und den Auftrag unerledigt zurückgibt, muß dieses nicht unbedingt das Scheitern des Auftraggebers nach sich ziehen, da der Autor seine Aufgabe immer noch erfüllen könnte, indem er alternativ ein anderes Büro beauftragt. Der Abbruch des Auftragnehmers hätte dann keinen Effekt auf das Gesamtergebnis.

Es existieren bereits erweiterte Transaktionskonzepte, die entweder das Kooperations- oder das Auftragsprinzip durchsetzen. Die Konzepte zur Realisierung des Kooperationsprinzips basieren meistens auf den offen geschachtelten Transaktionen, während das Auftragsprinzip meistens durch Konzepte realisiert wird, die auf geschlossen geschachtelten Transaktionen basieren. In der Praxis kommen allerdings oft beide Prinzipien kombiniert vor. Im Texterstellungsbeispiel zum Kooperationsprinzip könnte beispielsweise Autor B seinen Text durch das Übersetzungbüro in das Englische übersetzen lassen. Außerdem könnten im Übersetzungsbüro mehrere Mitarbeiter kooperativ an der Übersetzung arbeiten. Die existierenden erweiterten Transaktionskonzepte eignen sich aufgrund der engen Verflechtung von Fehlerbehandlung und Nebenläufigkeitskontrolle nicht als Basis für ein Konzept, das beide Prinzipien integriert.

Da beide Prinzipien sich auf die Nebenläufigkeitskontrolle beziehen, bieten dynamische Aktionen durch die Orthogonalität von Fehlerbehandlung und Nebenläufigkeitskontrolle einen sinnvollen Rahmen für die Entwicklung eines Konzepts, welches diese beiden Prinzipien integriert: Die existierenden Mechanismen zur Fehlerbehandlung können ohne Änderung genutzt werden, so daß sich die Integration auf den Bereich der Nebenläufigkeitskontrolle beschränkt. Dynamische Aktionen unterstützen bereits Nebenläufigkeitskonzepte zur Realisierung des Auftragsprinzip durch Schachtelung (siehe [Nett 1991], [Nett, Weiler 1994]) und des Kooperationsprinzips (siehe [Mock 1994]), die zu einem Integrationskonzept kombiniert werden können. Eine direkte Kombination der Konzepte ist aber nicht möglich, da beide Prinzipien teilweise entgegengesetzte Anforderungen an die Nebenläufigkeitskontrolle stellen. Statt dessen werden Mechanismen benötigt, die abhängig von der Anwendungssemantik in bestimmten Situationen das Auftrags- oder das Kooperationsprinzip durchsetzen. Die Entwicklung eines Integrationskonzepts, das diese Anforderungen erfüllt, ist Gegenstand dieser Arbeit.

Ergebnisse dieser Arbeit

In dieser Arbeit wird auf der Basis von dynamischen Aktionen das Konzept der *geschachtelten dynamischen Aktionen für kooperative Anwendungen* zur Kombination von Auftrags- und Kooperationsprinzip entwickelt. Es integriert die *geschachtelten dynamischen Aktionen* und die *dynamischen Aktionen für kooperative Anwendungen*. Das Integrationskonzept wird auch unter dem Gesichtspunkt entwickelt, eine möglichst gute Verträglichkeit mit den zugrundeliegenden Konzepten zu bieten. Diese Verträglichkeit wird nachgewiesen, indem das Integrationskonzept in ein formales Modell eingebettet wird, welches bereits für die dynamischen Aktionen für kooperative Anwendungen entwickelt wurde. Neben der Verträglichkeit werden so auch weitere Eigenschaften des Integrationskonzepts formal bewiesen.

Als wichtiger Bestandteil des Integrationskonzepts wird in dieser Arbeit der Mechanismus der *Objektkontrolle* eingeführt. Mit diesem Mechanismus können dynamische Aktionen die Sichtbarkeit von Objektänderungen individuell für einzelne Objekte steuern und dadurch Zugriffe durch andere dynamische Aktionen auf die Objekte kontrollieren. Die Objektkontrolle stellt somit ein adäquates Mittel dar, um sowohl das Auftrags- als auch das Kooperationsprinzip zu unterstützen. Sie bietet den Applikationen die Möglichkeit, in konkreten Situationen abhängig von ihrer Semantik festzulegen, welches der beiden Prinzipien durchgesetzt werden soll.

Zur Realisierung des Integrationskonzepts wird das *kooperative Sperrverfahren für geschachtelte dynamische Aktionen* entwickelt. Analog zur Erstellung des Konzepts werden hier das *geschachtelte Sperrverfahren* und das *flache kooperative Sperrverfahren*, die jeweils eines der beiden zugrundeliegenden Konzepte realisieren, unter Berücksichtigung einer bestmöglichen Verträglichkeit integriert.

Das kooperative Sperrverfahren wird durch eine Systemkomponente zur Sperrenverwaltung durchgesetzt, welche anhand von Regeln überprüft, ob die Anforderung oder Freigabe einer Sperre durch eine dynamische Aktion zulässig ist. In dieser Arbeit werden die für eine solche Überprüfung benötigten Algorithmen und Datenstrukturen detailiert beschrieben.

Schließlich wird die Systemkomponente zur Sperrenverwaltung zum Nachweis der tatsächlichen Einsetzbarkeit des Sperrverfahrens implementiert. Neben der beschriebenen Einbindung in das existierende verteilte Aktionssystem für dynamische Aktionen erfolgt zur Demonstration des kooperativen Sperrverfahrens eine Implementierung als eigenständige Komponente zur Nebenläufigkeitskontrolle mit interaktiver Nutzerschnittstelle.

Aufbau der folgenden Kapitel

Zunächst werden in Kapitel 2 die am weitesten verbreiteten Konzepte zur Realisierung von Auftrags- und Kooperationsprinzip beschrieben und diskutiert. Außerdem erfolgt in diesem Kapitel eine Einführung in die dynamischen Aktionen, wobei insbesondere die beiden Nebenläufigkeitskonzepte zur Realisierung des Auftrags- bzw. des Kooperationsprinzips genauer beschrieben werden.

In Kapitel 3 werden das Integrationskonzept und das zugehörige kooperative Sperrverfahren entwickelt. Die Einbettung in ein formales Modell und der formale Nachweis der Verträglichkeit mit den zugrundeliegenden Konzepten folgt in Kapitel 4. Kapitel 5 beschreibt die Algorithmen und Datenstrukturen zur Durchsetzung des Sperrverfahrens und Kapitel 6 die Implementierung der Sperrenverwaltung. Die Arbeit schließt in Kapitel 7 mit einer Zusammenfassung und einem Ausblick.

Kapitel 2

Erweiterte Transaktionskonzepte

2.1 Einführung

In diesem Kapitel werden erweiterte Transaktionskonzepte vorgestellt und es wird diskutiert, inwieweit sie das Auftrags- und das Kooperationsprinzip realisieren. Nach einer grundlegenden Einführung in das Transaktionskonzept werden geschlossen geschachtelte Transaktionen als Realisierung des Auftragsprinzips und offen geschachtelte Transaktionen als Realisierung des Kooperationsprinzips beschrieben sowie Konzepte, die das Ziel haben, beide Prinzipien zu vereinigen. Es folgt eine Einführung in das Konzept der dynamischen Aktionen. Dieses Konzept hat seinen Schwerpunkt im Gegensatz zu den anderen vorgestellten Transaktionskonzepten nicht auf einem der beiden Prinzipien, sondern es stellt grundlegende Mechanismen der Fehlerbehandlung zur Verfügung, die es erlauben, durch die Nebenläufigkeitskontrolle sowohl das Auftrags- als auch das Kooperationsprinzip zu realisieren. Diese Flexibilität wird durch eine strikte Trennung von Nebenläufigkeitskontrolle und Fehlerbehandlung erreicht. Zwei existierende und für diese Arbeit besonders relevante Nebenläufigkeitskonzepte werden in diesem Kapitel diskutiert: die „geschachtelten dynamischen Aktionen“ zur Realisierung des Auftragsprinzips und die „dynamischen Aktionen für kooperative Anwendungen“ zur Realisierung des Kooperationsprinzips. Dynamische Aktionen bilden auch die Grundlage für das in Kapitel 3 vorgestellte Integrationskonzept.

2.2 ACID-Transaktionseigenschaften

Das klassische Transaktionskonzept, das aus dem Bereich der Datenbanken stammt, basiert auf den *ACID-Eigenschaften* (siehe [Gray 1978] und [Gray, Reuter 1992]). Diese Eigenschaften haben sich zur Grundlage für die Betrachtung von Transaktionen entwickelt und bieten somit ein Vergleichskriterium für verschiedene erweiterte Transaktionskonzepte. ACID steht dabei für die folgenden vier Eigenschaften:

Atomicity: Fehler-Atomarität oder *Alles-oder-Nichts*-Eigenschaft. Es werden entweder alle Zustandsänderungen einer Transaktion an den Objekten durchgeführt oder gar keine. Bei einem fehlerhaften Ende, also einem Abbruch der Transaktion, müssen alle Objekte deshalb einen Zustand bekommen, der so aussieht, als habe die Transaktion nie begonnen.

Consistency: Konsistenz. Von Transaktionen wird gefordert, daß sie, wenn sie alleine abliefen, nur konsistente Zustandsänderungen durchführen. Es liegt in der Verantwortung des Applikationsprogrammierers, daß die Transaktion nur Objektzustände erzeugt, die nach Applikationssemantik zulässig sind.

Isolation: Isolierte Transaktionen dürfen von anderen nebenläufigen Transaktionen und deren möglichen Abbrüchen nicht beeinflußt werden. Dies wird durchgesetzt, indem alle Änderungen einer Transaktion erst bei ihrem erfolgreichen Ende für die anderen Transaktionen sichtbar gemacht werden.

Durability: Permanenz des Effekts. Zustandsänderungen von erfolgreich beendeten Transaktionen bleiben permanent, auch wenn nachher Systemfehler wie zum Beispiel Rechnerausfälle auftreten.

Diese vier Eigenschaften lassen sich in Bezug auf eine Transaktionsverwaltung folgendermaßen klassifizieren:

Konsistenz ist eine Anforderung an die Applikation und nicht an die Transaktionsverwaltung. Bei den weiteren Betrachtungen wird sie deshalb vorausgesetzt und nicht mehr betrachtet.

Fehler-Atomarität und *Permanenz* betreffen die Fehlerbehandlung. Zur Durchsetzung dieser beiden Eigenschaften können Transaktionen bei ihrem Ende in zwei verschiedene Zustände übergehen: Im Zustand *abgebrochen* wurden aufgrund der Fehler-Atomarität keine Änderungen der Transaktion durchgeführt, während der Zustand *committed* besagt, daß alle Änderungen durchgeführt wurden und auch nicht mehr durch Systemfehler verloren gehen können.

Die *Isolation* schließlich betrifft die Nebenläufigkeitskontrolle, hat aber bei Transaktionen auch Einfluß auf die Art der Fehlerbehandlung: Das klassische Transaktionskonzept geht davon aus, daß bei einem Transaktionsabbruch genau die Objektänderungen betroffen sind, die von der abbrechenden Transaktion durchgeführt wurden. Fehler-Atomarität kann daher garantiert werden, indem bei einem Abbruch alle von dieser Transaktion geänderten Objekte auf den Wert zurückgesetzt werden, den diese zu Beginn der Transaktion hatten. Andere Objekte müssen nicht betrachtet werden. Diese Annahme ist aber nur dann gültig, wenn Transaktionen isoliert ablaufen, da sonst die später zurückgesetzten Objektänderungen bereits von anderen Transaktionen gelesen und zu Berechnungen benutzt worden sein könnten. Erst bei dynamischen Aktionen gibt es eine vollständige Trennung von Fehlerbehandlung und Nebenläufigkeitskontrolle. Hier werden Atomarität und Permanenz garantiert, ohne daß Isolation gefordert wird.

Unter dem Gesichtspunkt der Nebenläufigkeitskontrolle stellt die Isolationseigenschaft eine verschärfte Form der *Serialisierbarkeit* dar. Diese grundlegende Eigenschaft besagt, daß nebenläufige Transaktionen den gleichen Effekt haben müssen, wie in einem Ablauf, in dem sie seriell nacheinander ausgeführt werden. Dadurch verhindert Serialisierbarkeit gegenseitige Einflüsse von Transaktionen und garantiert im fehlerfreien Fall die Unabhängigkeit der einzelnen Transaktionen, selbst wenn sie nebenläufig auftreten. Im Gegensatz zur Isolation, die die Unabhängigkeit der Transaktionen auch im Fehlerfall garantiert, fordert die Serialisierbarkeit nicht, daß Objektänderungen erst bei Ende einer Transaktion sichtbar gemacht werden dürfen. Beispielsweise erlaubt das bei geschachtelten dynamischen Aktionen benutzte *nicht-strikte Zwei-Phasen-Sperrprotokoll* (siehe Abschnitt 2.6.2) bei garantierter Serialisierbarkeit, Objektänderungen frühzeitig sichtbar zu machen.

Serialisierbarkeit erlaubt einen Ablauf der Transaktionen, der frei von sogenannten *Nebenläufigkeitsanomalien* ist (siehe [Gray, Reuter 1992]). Diese klassifizieren solche unerwünschten Effekte, in denen eine Berechnung den Ausgang einer anderen dadurch beeinflußt, daß sie nebenläufig auftritt und dieselben Objekte benutzt. Da die Berechnungen in der Regel nichts von der Existenz anderer Berechnungen wissen und diese daher auch nicht berücksichtigen, würden Nebenläufigkeitsanomalien oft zu unvorhersehbaren Störungen der Berechnung führen.

Allerdings ist es auch öfters unerwünscht, gegenseitige Einflüsse durch die erzwungene Serialisierbarkeit systemseitig auszuschließen, da so auch jede Art von Kooperation von Transaktionen verhindert wird. Oft heben deshalb erweiterte Transaktionskonzepte zumindest teilweise die Serialisierbarkeit auf. Die dabei auftretenden Effekte sollen hier genauer untersucht werden.

Nebenläufige (erweiterte) Transaktionen können sich gegenseitig beeinflussen, indem sie zumindest teilweise dieselben Objekte benutzen. Dabei können Zwischenergebnisse der einzelnen Transaktionen zu Argumenten der jeweils anderen Transaktion werden. Abbildung 2.1 zeigt ein dafür typisches Szenario.

Transaktion T_1	Transaktion T_2
$\vdots$	$\vdots$
read o_1	$\vert$
write o_1	$\vert$
$\vert$	read o_1
$\vert$	write o_1
read o_1	$\vert$
write o_1	$\vert$
$\vdots$	$\vdots$

Abbildung 2.1: sich beeinflussende nebenläufige Transaktionen

Je nach Kontext ist eine solche Beeinflussung unterschiedlich zu bewerten: Zum einen könnte sie gewollt sein. In diesem Beispiel könnte die Transaktion T_1 in dem Objekt o_1 ein Zwischenergebnis für Transaktion T_2 ablegen. Diese liest es, führt weitere Berechnungen durch und speichert ihr Ergebnis wieder in o_1, um es T_1 zur Verfügung zu stellen. Der gleiche Ablauf kann aber auch eine ganz andere Bedeutung haben: Die Transaktionen T_1 und T_2 sind unabhängig voneinander und arbeiten nur zufällig gleichzeitig auf demselben Objekt. T_1 besteht dabei aus zwei Modulen, die jeweils den Wert von Objekt o_1 lesen, verändern und wieder zurückschreiben. Transaktion T_1 geht dabei davon aus, daß das zweite Modul aus o_1 den Wert liest, den das erste Modul vorher geschrieben hat. Transaktion T_2 würde durch die zwischenzeitliche Veränderung von o_1 das Ergebnis von T_1 verfälschen.

Dieses Beispiel macht deutlich, daß es ausschließlich von der Anwendungssemantik abhängt, ob eine gegenseitige Beeinflussung gewollt und damit zulässig ist, oder ob sie verhindert werden sollte. Diese Beeinflussungen werden in einer Transaktionsverwaltung mit Hilfe der *Nebenläufigkeitskontrolle* überwacht und gegebenenfalls eingeschränkt. Je nach Ziel des jeweiligen Transaktionskonzepts sind die Einschränkungen dabei unterschiedlich stark. Das Isolationskonzept der klassischen Transaktionen hat zum Beispiel als Ziel, nebenläufige Transaktionen unabhängig und isoliert voneinander ablaufen zu lassen. Daher wird dort jegliche Art der gegenseitigen Einflußnahme unterbunden. Die geforderte Serialisierbarkeit verhindert alle Arten von Nebenläufigkeitsanomalien, aber auch jegliche Kommunikation zwischen den Transaktionen. Erweiterte Transaktionskonzepte haben oft zum Ziel, die strenge Serialisierbarkeit und damit auch die Isolation aufzugeben und in unterschiedlichem Maße Kommunikation zwischen Transaktionen zuzulassen. Da unerwünschte Nebenläufigkeitsanomalien schwer zu handhaben sind, wird in der Regel versucht, Serialisierbarkeit generell beizubehalten, und diese nur zwischen definierten Kommunikationspartnern oder zu bestimmten Zeitpunkten aufzuheben. Die erweiterten Transaktionskonzepte unterscheiden sich stark im Umfang der erlaubten Kommunikation und in der Flexibilität, mit der Kommunikationspartner festgelegt werden.

2.3 Geschlossen geschachtelte Transaktionen

Die in [Moss 1981] eingeführten *geschlossen geschachtelten Transaktionen* heben die Isolation von klassischen Transaktionen für einen speziellen Fall auf: Sie erlauben zur Realisierung des Auftragsprinzips, daß Transaktionen Teilaufträge an andere Transaktionen vergeben und deren Ergebnisse weiterverwenden. Abläufe wie in Abbildung 2.1 sind also erlaubt, sofern zwischen T_1 und T_2 eine Auftraggeber-Auftragnehmer-Beziehung besteht.

Geschlossen geschachtelte Transaktionen bilden eine Unterstrukturierung von klassischen Transaktionen in Form eines Baumes. Die Wurzel des Baums bildet die sogenannte *Toplevel*-Transaktion. Diese kann Sohn-Transaktionen erzeugen, die selbst wieder Söhne erzeugen können. So entsteht eine Baumstruktur, die die Vater-Sohn-Beziehungen zwischen den einzelnen (Sub-) Transaktionen widerspiegelt.

Für einen kompletten Transaktionsbaum, den die Toplevel-Transaktion repräsentiert, werden alle ACID-Eigenschaften garantiert. Für Teilbäume, repräsentiert durch die Sub-Transaktionen an der Wurzel des Teilbaums, werden Fehler-Atomarität und bezüglich der Brüder auch Isolation zugesichert. Die Toplevel-Transaktion kann nur dann in den Zustand *committed* übergehen, wenn alle Nachfahren bereits beendet sind. Wenn eine Vater-Transaktion abbricht, werden auch alle ihre Nachfahren abgebrochen. Sohn-Transaktionen hingegen können abbrechen, ohne daß der Vater davon betroffen ist. Da also eine bereits beendete Sohn-Transaktion bei einem Abbruch ihres Vaters auch nachträglich zurückgesetzt werden kann, erfüllen Sub-Transaktionen nicht die Permanenz-Eigenschaft.

Da komplette Aktionsbäume und auch Teilbäume untereinander die Eigenschaft der Isolation erfüllen, werden Änderungen, die ihre Söhne gemacht haben, bei deren Ende nicht direkt allgemein sichtbar, sondern sie stehen zuerst nur dem Vater zur Verfügung. Nur der Vater kann die geänderten Objektzustände lesen und weiterverarbeiten oder nachfolgend erzeugten Söhnen zur Verfügung stellen. Objektänderungen von Sohn-Transaktionen werden allgemein also erst mit dem *Commit* der Toplevel-Transaktion sichtbar. Dadurch können eventuell Objekte sehr lange vor dem Zugriff durch andere Transaktionen gesperrt sein, obwohl die ändernde Transaktion schon längst beendet ist.

Geschlossen geschachtelte Transaktionen sind bezüglich der Nebenläufigkeitskontrolle sehr restriktiv: Toplevel-Transaktionen erfüllen wie klassische Transaktionen die Isolations-Eigenschaft und Brüder sind untereinander isoliert. Deshalb ist zwischen nebenläufigen (Sub-) Transaktionen keine Kommunikation möglich. Da Vor- und Nachfahren untereinander nicht serialisierbar sein müssen, kann es zwischen ihnen Kommunikation geben.

Das Konzept der klassischen Transaktionen wurde durch die geschlossen geschachtelten Transaktionen in zwei Punkten erweitert: Einerseits werden Transaktionen vor kompletten Abbrüchen geschützt. Tritt in einem Teil einer Transaktion ein Fehler auf, wird im klassischen Transaktionsmodell die komplette Transaktion zurückgesetzt. Insbesondere bei längeren Transaktionen können dadurch automatisch umfangreiche Berechnungen zurückgesetzt werden, obwohl der eigentliche Fehler von der Applikation auch anders hätte behandelt werden können. Die Schachtelung schafft bei diesem Problem Abhilfe, indem sie es erlaubt, solche kritischen Teile in Sohn-Transaktionen zu verlagern, die ohne Einfluß auf den Vater abbrechen können.

Zusätzlich bieten geschlossen geschachtelte Transaktionen auch Systemunterstützung von Parallelität innerhalb von Transaktionen. Zwei unabhängige Berechnungen, zum Beispiel in verschiedenen Prozessen innerhalb einer Transaktion, können in verschiedenen Söhnen ablaufen. Das System garantiert aufgrund der Isolationseigenschaft, daß diese Berechnungen unabhängig voneinander ablaufen und sich nicht gegenseitig stören.

Geschlossen geschachtelte Transaktionen sind eine Möglichkeit, das Auftragsprinzip zu realisieren, wobei die Vater-Sohn-Beziehung dem Verhältnis Auftraggeber-Auftragnehmer entspricht. Durch das Erzeugen einer Sohn-Transaktion delegiert der Vater als Auftraggeber einen Teil seiner Tätigkeiten. Bei Beendigung der Sohn-Transaktion stehen die Ergebnisse des Auftragnehmers zuerst nur dem Auftraggeber zur Verfügung, genau wie es das Auftragskonzept vorsieht. Der Auftraggeber kann über die weitere Verwendung der Ergebnisse entscheiden und der Auftragnehmer bleibt anderen Transaktionen ganz verborgen. Geschlossen geschachtelte Transaktionen sind nicht in der Lage, das Kooperationsprinzip zu unterstützen, da sie mit der Durchsetzung der ACID-Eigenschaften Kommunikation untereinander verbieten.

2.4 Offen geschachtelte Transaktionen

Offen geschachtelte Transaktionen ([Gray 1981] und [Traiger 1983]) erweitern klassische Transaktionen, indem sie Kommunikation zwischen verschiedenen Transaktionsbäumen erlauben und zu diesem Zweck die Isolationsanforderung an Transaktionsbäume aufheben. Bei offen geschachtelten Transaktionen werden im Gegensatz zu geschlossen geschachtelten Transaktionen die Objektänderungen der Sub-Transaktionen bei deren Ende allgemein sichtbar. Zu diesem Zeitpunkt können beliebige andere Transaktion diese Ergebnisse lesen und weiterverarbeiten. Es existiert also eine Beschränkung der Zeitpunkte, an denen Kommunikation und damit auch Nebenläufigkeitsanomalien auftauchen können, auf Beginn und Ende von Sub-Transaktionen. Die möglichen Kommunikationspartner hingegen können nicht spezifiziert werden, da *jede* andere Transaktion die Ergebnisse einer Sub-Transaktion lesen kann.

Da Objekte bereits bei Beendigung einer Sub-Transaktion wieder für andere Transaktionen zugreifbar sind, werden die Möglichkeiten der Nebenläufigkeit stark erhöht. Operationen auf Objekten, die innerhalb einer länger laufenden Transaktion nur kurz benutzt werden, können in einer Sub-Transaktion durchgeführt werden. Der Zugriff durch andere Transaktionen ist sofort nach Ende der Sub-Transaktion möglich, wohingegen bei geschlossen geschachtelten Transaktionen das Objekt bis zum Ende der Toplevel-Transaktion gesperrt bleibt.

Wenn Sub-Transaktionen enden und ihre Ergebnisse allgemein freigeben, gehen sie in den Zustand *committed* über, d. h. die geänderten Objektzustände werden permanent gespeichert. Trotzdem sollen auch offen geschachtelte Transaktionen die Eigenschaft der Fehler-Atomarität erfüllen. Dies führt zu Problemen, da bei einem Abbruch einer Transaktion auch die Effekte aller ihrer Söhne rückgängig gemacht werden müssen, einschließlich derer, die bereits beendet sind. Es widerspräche dem Transaktionskonzept und der Eigenschaft der Permanenz, wenn systemseitig die Möglichkeit bestünde, committete Objektzustände wieder zurückzusetzen. Zur Lösung dieses Problems sieht das Konzept der offen geschachtelten Transaktionen sogenannte *kompensierende Transaktionen* vor.

Kompensierende Transaktion

> Mit einer *kompensierenden Transaktion* wird die Wirkung einer bereits erfolgreich beendeten anderen Transaktion semantisch aufgehoben. Kompensierende Transaktionen müssen auch dann funktionieren, wenn zwischen dem Ende der zu kompensierenden Transaktion und dem Beginn der kompensierenden Transaktion fremde Transaktionen die betroffenen Objekte geändert haben. Diese fremden Änderungen müssen auch nach der Kompensation gültig bleiben.
>
> Beispielsweise kompensiert die Auszahlung eines bestimmten Betrages von einem Konto die Einzahlung desselben Betrages auf dieses Konto.

Semantisch kompensierende Transaktionen können nicht systemseitig automatisch erzeugt werden, sondern müssen vom Applikationsprogrammierer gemeinsam mit der eigentlichen Transaktion zur Verfügung gestellt werden.

Die Anforderung, daß zwischen Transaktion und Kompensation fremde Transaktionen die benutzten Objekte bearbeiten dürfen, wirft weitere Probleme auf: In dem obigen Beispiel des Kontos ist die Auszahlung schon dann nicht mehr kompensierend, wenn zwischen Ein- und Auszahlung eine Zinsberechnung und -gutschrift stattgefunden hat. Auf dem Konto befinden sich weiterhin die Zinsen für den eingezahlten Betrag, obwohl es ihn eigentlich auf dem Konto nie gegeben hat.

Dieses Beispiel zeigt, daß vom Abbruch einer Transaktion nicht nur sie selbst betroffen ist, sondern daß der Abbruch auch Auswirkungen auf andere Transaktionen hat: Es findet eine *Schadensausbreitung* statt. Es gäbe jetzt die Möglichkeit, wie bei den weiter unten in Abschnitt 2.6 beschriebenen dynamischen Aktionen, die Menge der Transaktionen zu bestimmen, die aufgrund der Schadensausbreitung von dem Abbruch betroffen sind, und diese auch abzubrechen bzw. zu kompensieren. Eine weitere Möglichkeit besteht darin, eine Schadensausbreitung von vornherein zu verhindern, indem man den Anwendungsbereich der offen geschachtelten Transaktionen auf solche Transaktionen einschränkt, die untereinander kommutieren. Bei kommutierenden Transaktionen hängt der Erfolg einer Transaktion in keiner Weise von den Zuständen ab, die die benutzten Objekte zu ihrem Beginn haben. Es ist also unerheblich, ob eine andere Transaktion vorher stattgefunden hat oder nicht, und es findet daher auch keine Schadensausbreitung statt. Obwohl die Beschränkung auf kommutierende Transaktionen den Anwendungsbereich eines Transaktionskonzepts stark einschränkt, wird dieser Weg von vielen auf offen geschachtelten Transaktionen aufbauenden Konzepten gewählt. Eine weitere Diskussion des Kommutativitätsbegriffs findet sich zum Beispiel in [Weihl 1988] und [Badrinath, Ramamritham 1992].

Im Konzept der offen geschachtelten Transaktionen wird die enge Verflechtung von Nebenläufigkeitskontrolle und Fehlerbehandlung besonders deutlich: Eine korrekte Fehlerbehandlung mittels Kompensation kann nur dann erfolgreich durchgeführt werden, wenn im Bereich die Nebenläufigkeit die starke Einschränkung der Kommutativität garantiert werden kann. Da im Gegensatz zu den dynamischen Aktion die Orthogonalität dieser beiden Bereiche nicht gegeben ist, führt eine Abweichung von der Kommutativitätsforderung direkt zu massiven Problemen in der Fehlerbehandlung. Beispielsweise fordert das weiter unten in Unterabschnitt 2.4.2 beschriebene ConTract Modell nicht zwingend die Kommutativität und nimmt dafür eine unkontrollierte Schadensausbreitung für den Fall in Kauf, daß durch die Kompensation die Bedingungen für die Ausführung einer anderen Transaktion nachträglich nicht mehr gegeben sind.

Ein zusätzliches Problem bei kompensierenden Transaktionen liegt darin, daß sie auch Abbrüche zurücksetzen können müssen, die durch den Komplettausfall einer Stelle ausgelöst wurden, d. h. die kompensierende Transaktion muß dem System bei einem Neustart zur Verfügung stehen, sie muß also permanent gespeichert sein.

Offen geschachtelte Transaktionen können in gewissem Rahmen das Kooperationsprinzip realisieren, da sie Informationsfluß zwischen Aktionsbäumen zulassen. Allerdings gibt es keine Möglichkeit, die Kooperationspartner genauer zu spezifizieren, sondern Zwischenergebnisse sind für alle Transaktionen sichtbar. Die Festlegung der Kooperationspartner, also der Menge von Transaktionen, die das Zwischenergebnis lesen dürfen, ist aber wesentlich für eine praktische Realisierung des Kooperationsprinzips. Die fehlende Beschränkung der Sichtbarkeit von Objektänderungen führt auch dazu, daß das Auftragsprinzip von offen geschachtelten Transaktionen nicht realisiert werden kann, da es für dieses Prinzip grundlegend ist, daß Objektänderungen von Auftragnehmern zuerst nur für den Auftraggeber sichtbar sind.

2.4.1 Sagas

Die in [Garcia-Molina, Salem 1987] und [Chrysanthis, Ramamritham 1992] vorgestellten Sagas basieren auf offen geschachtelten Transaktionen, haben aber eine starr vorgegebene Art der Schachtelung. Eine Saga ist eine Toplevel-Transaktion, die eine Ebene von Söhnen hat, die sogenannten Komponenten-Transaktionen, die selbst keine weiteren Söhne haben dürfen und somit normale ACID-Transaktionen sind. Außerdem darf eine Saga selbst keine Operationen auf Objekten ausführen, dies dürfen nur die Komponenten, für die daher auch kompensierende Transaktionen existieren müssen. Komponenten unterschiedlicher Sagas müssen untereinander kommutieren, um eine Schadensausbreitung auszuschließen. Da alle Komponenten *einer* Saga seriell ablaufen und daher ihre Ausführungsreihenfolge festgelegt ist, müssen sie untereinander nicht kommutieren.

Wenn, wie es bei Sagas der Fall ist, die Kommutativität von Komponenten unterschiedlicher Sagas zwingende Voraussetzung ist, kann trotz früher Freigabe von Objektänderungen keine echte Kooperation stattfinden: Kooperation beinhaltet, daß ein Informationsfluß über die Objekte von einer Transaktion zu einer anderen stattfindet. Aufgrund des Objektzustands, den die eine Transaktion geschrieben hat, fällt die lesende Transaktion dann bestimmte Entscheidungen. Genau dieses wird aber durch die geforderte Kommutativität verboten, da alle Operationen, die die lesende Transaktion durchführt, unabhängig von dem gerade gelesenen Objektzustand sein müssen.

Sagas sind also keine Realisierung des Kooperationsprinzips, sondern haben das Ziel, insbesondere bei langlebigen Transaktionen die Nebenläufigkeit zu erhöhen, indem Objekte bereits lange vor dem Ende der Transaktion wieder freigegeben und von anderen Transaktionen benutzt werden können. Voraussetzung ist allerdings die Existenz von kompensierenden Transaktionen und Kommutativität aller Komponenten aus unterschiedlichen Sagas.

2.4.2 Das ConTract Modell

In [Reuter 1989], [Waechter, Reuter 1990] und [Waechter, Reuter 1992] wurde mit den ConTracts ein weiteres auf offen geschachtelten Transaktionen basierendes Transaktionskonzept vorgestellt. Ähnlich wie bei Sagas gibt es auch hier eine Schachtelung auf zwei Ebenen: Die *ConTract* genannte Toplevel-Transaktion hat eine Schicht an Söhnen, die *Schritte* genannt werden und bei denen es sich um ACID-Transaktionen handelt. Ein ConTract selbst kann keine Operation ausführen, sondern besteht aus einem *Skript*, in dem festgelegt wird, in welcher Reihenfolge die einzelnen Schritte ausgeführt werden. Im Gegensatz zu den Sagas, in denen nur eine lineare Ausführung der Komponenten möglich ist, können Skripte auch Verzweigungen, Schleifen und die Möglichkeit zur nebenläufigen Ausführung von Schritten beinhalten.

Da es sich bei ConTracts um offen geschachtelte Transaktionen handelt, werden für alle Schritte kompensierende Transaktionen benötigt, um die Fehler-Atomarität der ConTracts zu gewährleisten. Die bei Sagas noch geltende Forderung, daß alle Komponenten unterschiedlicher Sagas kommutieren müssen, um sicher zu stellen, daß eine kompensierte Komponente keine Auswirkungen auf andere Sagas hat, ist im ConTract Modell in dieser Stärke aufgehoben: Der Applikationsprogrammierer kann für jeden Schritt Eingangsbedingungen, sogenannte *Invarianten*, angeben. Ein Schritt S_1 ist nur dann von der Kompensation eines anderen Schritts S_2 betroffen, wenn nach dessen Kompensation die angegebene Invariante nicht mehr erfüllt ist. Falls dies aber der Fall ist, muß auch der ConTract, zu dem der Schritt S_1 gehört, kompensiert werden. Dieses Verhalten, bei dem eine Kompensation eine weitere auslöst, die ihrerseits wiederum Kompensationen auslösen kann, wird *kaskadierende Kompensation* genannt und kann schnell dazu führen, daß eine Vielzahl von ConTracts aufgrund eines einzelnen Abbruchs kompensiert werden müssen. Im ConTract Modell ist kein Mechanismus vorgesehen, kaskadierende Kompensationen zu beschränken oder auch nur bei einem Abbruch die Menge der betroffenen ConTracts zu bestimmen.

Durch die Aufhebung der Kommutativitätsforderung erlaubt das ConTract Modell echte Kooperation zwischen ConTracts, da hier ein Schritt wirkliche Entscheidungen aufgrund von Objektänderungen eines Schritts aus einem anderen noch laufenden ConTract fällen kann. Wie schon bei offen geschachtelten Transaktionen beschrieben, ist auch hier keine Spezifikation der Kooperationspartner möglich.

Das ConTract Modell bietet die Möglichkeit der Kooperation zwischen ConTracts, besitzt aber einige Nachteile, die von dem ihm zugrundeliegenden Konzept der offen geschachtelten Transaktionen herrühren: die fehlende Spezifikation von Kooperationspartnern und die notwendige Existenz von Kompensationen für jeden Schritt. Außerdem ist die Behandlung von kaskadierenden Kompensationen nicht umfassend gelöst.

2.5 Ansätze zur Kombination von Auftrags- und Kooperationsprinzip

Nachdem in den beiden vorherigen Abschnitten mit den geschlossen und offen geschachtelten Transaktionen die bekanntesten Konzepte vorgestellt wurden, die jeweils *nur* das Auftrags- bzw. Kooperationsprinzip realisieren, werden in diesem Abschnitt einige Konzepte diskutiert, die das Ziel haben, beide Konzepte zu integrieren.

2.5.1 Das DOM Transaktionsmodell

Eine naheliegende Möglichkeit der Kombination von Auftrags- und Kooperationsprinzip besteht darin, ein Transaktionskonzept zu entwickeln, das Eigenschaften der offen und der geschlossen geschachtelten Transaktionen integriert, da diese für sich jeweils das Kooperations- bzw. das Auftragsprinzip realisieren. Eine solche Integration wurde beispielsweise in dem in [Buchmann et al. 1992] vorgestellten DOM Transaktionsmodell („**D**istributed **O**bject **M**anagement“) durchgeführt und wird hier kurz beschrieben:

Im DOM Transaktionsmodell werden Transaktionen wie in den schon beschriebenen Konzepten hierarchisch in Bäumen strukturiert. Dabei werden mit den *Multitransaktionen* und den *Toptransaktionen* zwei Typen von (Sub-) Transaktionen unterschieden.

Multitransaktionen sind Wurzeln von offen geschachtelten (Teil-) Bäumen, deren Söhne selbst wieder Multitransaktionen oder auch Toptransaktionen sein können. Dabei können alle Söhne einer Multitransaktion nebenläufig gestartet werden, es sei denn, es werden Regeln angegeben, die die Ausführungsreihenfolge der Söhne spezifizieren. Für Multitransaktionen wird Fehler-Atomarität aber nicht Isolation gefordert und entsprechend dem Konzept der offen geschachtelten Transaktionen müssen daher für alle Söhne kompensierende Transaktionen existieren. Da nur Toptransaktionen Objekte verändern dürfen, muß der Applikationsprogrammierer nur für diese Kompensationen angeben und komplette Multitransaktionen werden kompensiert, indem alle ihre bereits erfolgreich beendeten Söhne in der umgekehrten Reihenfolge der Commit-Ereignisse kompensiert werden. Söhne von Multitransaktionen werden auch danach unterschieden, ob sie vital oder nicht-vital sind: Eine Vatertransaktionen darf nur dann in den Zustand committed übergehen, wenn alle vitalen Söhne bereits in diesem Zustand sind. Nicht-vitale Söhne hingegen können abbrechen, ohne den Abbruch des Vaters nach sich zu ziehen.

Toptransaktionen sind Wurzeln von geschlossen geschachtelten Transaktionsbäumen, die entsprechend [Moss 1981] die Eigenschaften der Fehler-Atomarität und Isolation (bezüglich Brüdern) erfüllen. Toptransaktionen dürfen nur weitere Toptransaktionen als Nachfahren haben, wobei flache Transaktionen auch als Toptransaktionen gelten.

Auftrags- und Kooperationsprinzip

Das Auftragsprinzip wird im DOM Transaktionsmodell durch die Toptransaktionen realisiert. Da Toptransaktionen geschlossen geschachtelte Transaktionen sind, gelten für sie alle in Abschnitt 2.3 gemachten Aussagen zur Realisierung von Auftrags- und Kooperationsprinzip entsprechend.

Das Kooperationsprinzip wird durch Multitransaktionen gelöst, allerdings gelten auch hier die von den offen geschachtelten Transaktionen bekannten Einschränkungen: Kooperationspartner können nicht festgelegt werden und um wirkliche Kooperation realisieren zu können, muß auf die Kommutativität zwischen den Söhnen unterschiedlicher Multitransaktionen verzichtet werden. Die daraus resultierende Schadensausbreitung im Falle eines Abbruchs wird im DOM Transaktionsmodell nicht betrachtet.

Ferner ist die Integration beider Prinzipien nur unzureichend gelöst: Eine Transaktion kann entweder als Toptransaktion Unteraufträge an andere Transaktionen vergeben oder als Multitransaktion mit anderen Multitransaktionen kooperieren. Es ist nicht möglich, daß eine Transaktion sowohl Unteraufträge vergibt als auch mit anderen Transaktionen kooperiert. Auftragnehmer sind immer Toptransaktionen und können deshalb in keinem Fall kooperieren.

Im DOM Transaktionskonzept ist die Hauptintention von Multitransaktionen aufgrund dieser Probleme nicht die Kooperation, sondern die Erhöhung der Nebenläufigkeit bei langen Transaktionen durch frühzeitige Freigabe von Objektänderungen. Außerdem erleichtern Multitransaktionen wie in [Buchmann et al. 1992] beschrieben die Einbindung externer Datenbanken, deren Transaktionssteuerung nicht komplett von DOM übernommen werden kann.

Zusammenfassung

Das DOM Transaktionsmodell zeigt ein grundsätzliches Problem, das auftritt, wenn zur Integration von Auftrags- und Kooperationsprinzip versucht wird, geschlossen und offen geschachtelte Transaktionen miteinander zu kombinieren:

Geschlossen und offen geschachtelte Transaktionen sind zwei Konzepte, die sich sowohl in ihrer Art der Nebenläufigkeitskontrolle als auch in der Art der Fehlerbehandlung unterscheiden. Auf der anderen Seite trennen beide Konzepte in ihren Mechanismen aber nicht zwischen Fehlerbehandlung und Nebenläufigkeitskontrolle. So funktioniert die Fehlerbehandlung der geschlossen geschachtelten Transaktionen, die eine Schadensausbreitung außerhalb des Aktionsbaums nicht betrachtet, nur deshalb, weil seitens der Nebenläufigkeitskontrolle die Isolation durchgesetzt wird. Die Aufgabe der Isolation hängt bei den offen geschachtelten Transaktionen direkt mit der Fehlerbehandlung auf der Grundlage der kompensierenden Transaktionen zusammen.

Auftrags- und Kooperationsprinzip sind Anforderungen, die nur die Nebenläufigkeitskontrolle betreffen. Bei der Integration dieser Prinzipien geht es also darum, zwei verschiedene *Nebenläufigkeitskonzepte* zu kombinieren. Das zugrundeliegende Konzept zur Fehlerbehandlung sollte von der Integration gar nicht betroffen sein und statt dessen unabhängig von der jeweiligen Nebenläufigkeitskontrolle die Fehler-Atomarität durchsetzen. Genau dieses ist aber bei der Integration der beiden geschachtelten Transaktionskonzepte nicht möglich. Daher kann kein auf offen geschachtelten Transaktionen basierendes Konzept ohne kompensierende Transaktionen auskommen, die den Anwendungsbereich des Konzepts wie oben gezeigt stark einschränken.

2.5.2 Das CONCORD Modell

Ein anderer Ansatz zur Integration des Auftrags- und des Kooperationsprinzips wurde mit dem CONCORD Modell („**Con**trolling **Co**ope**R**atation in **D**esign Environments“) in [Ritter et al. 1993] und [Ritter et al. 1994] vorgestellt. CONCORD wurde zur Systemunterstützung von CAD Anwendungen in Arbeitsgruppen entwickelt, wobei die Schwerpunkte sowohl auf der Kooperation von Benutzern als auch auf der Delegation von Unteraufträgen liegen.

Das Konzept basiert auf den sogenannten *Design Activities*. Diese modellieren die Arbeitsabläufe, die zur Erledigung eines Auftrags nötig sind und haben eine ähnliche Struktur wie die in Abschnitt 2.4.2 vorgestellten ConTracts: Eine Design Activity besteht aus einem Skript zur Ausführung von Komponenten, den *DOPs* (Design Operations). Die DOPs sind wie die Schritte der ConTracts ACID-Transaktionen. Im Gegensatz zu ConTracts sind Design Activities aber *keine* Transaktionen, d. h. für sie wird weder Fehler-Atomarität noch Isolation oder Serialisierbarkeit zwingend gefordert. Allerdings stellt das System Mechanismen zur Verfügung, mit denen alle Berechnungen einer Design Activity zurückgesetzt oder Zwischenergebnisse von DOPs vor Zugriff von außen geschützt werden können.

CONCORD arbeitet mit einem *versionierten Objektsystem*. Zu Beginn einer Design Activity werden lokale Versionen der zu bearbeitenden Objekte angelegt. Jede DOP kann Objekte ändern und legt dadurch neue lokale Versionen dieser Objekte an. Wenn eine Design Activity erfolgreich endet, werden die letzten Objektversionen in den globalen Objektspeicher zurückgeschrieben. Dieses Zurückschreiben findet in der Regel aber nicht „blind“ statt, sondern die Design Activity bildet in einer sogenannten *Konfigurierung* aus der aktuellen globalen und der lokalen Version eine neue globale Version, wobei der Programmierer der Design Activity dafür verantwortlich ist, daß die Konsistenz des globalen Objektspeichers erhalten bleibt. Alle lokalen Objektversionen bilden den *Scope*, also die Menge der zugreifbaren Objekte einer Design Activity.

Eine Design Activity enthält neben dem Ausführungsskript der DOPs und den lokalen Objektversionen noch eine weitere Information: Mit einer *Designspezifikation* wird das Ziel einer Design Activity beschrieben. Sobald diese Spezifikation erfüllt ist, hat die Design Activity ihre Aufgabe erfüllt. Außerdem kann anhand einer Bewertungsfunktion jederzeit bestimmt werden, zu welchem Grad diese Spezifikation erfüllt ist.

Zwischen Design Activities können drei Typen von Beziehungen bestehen:

Delegation: Der Typ *Delegation* repräsentiert eine Auftraggeber-Auftragnehmer-Beziehung.

Der Auftraggeber erstellt für den Auftragnehmer eine Designspezifikation und kann ihm als initiale Objektversion Kopien einiger Objekte aus seinem eigenen Scope übergeben. Nach dem Ende des Auftragnehmers muß der Auftraggeber aus den endgültigen Objektversionen des Auftragnehmers und seinen eigenen aktuellen Versionen in einer Konfigurierungsphase einen neuen Objektzustand erzeugen.

Usage: Der Beziehungstyp *Usage* dient der Kooperation. Eine Usage-Beziehung entsteht, wenn eine Design Activity von einer anderen Design Activity Zwischenversionen von Objekten anfordert und diese auch bekommt. In dem Fall erweitert sich der Scope der anfordernden Design Activity um Kopien der angeforderten Objektversionen. Bei einer Anforderung einer Zwischenversion kann mit der Bewertungsfunktion aus der Designspezifikation bestimmt werden, wie verläßlich diese Version ist und der Anforderer kann daraufhin entscheiden, ob er diese Version benutzen will.

Negotiation: Wenn zwei Design Activities in einer *Negotiation*-Beziehung stehen, findet eine andere Art der Kooperation statt: Die beteiligten Partner, die beide Auftragnehmer desselben Auftraggebers sein müssen, können ihre Designspezifikationen neu „verhandeln“, indem zum

Beispiel ein Partner noch Aufgaben des anderen übernimmt. Die Negotiation-Beziehung erlaubt nicht, daß Objektversionen ausgetauscht werden.

Fehlerbehandlung

Bei einem Abbruch einer DOP werden systemseitig alle ihre Änderungen zurückgesetzt, da es sich bei DOPs um normale ACID-Transaktionen handelt. (Es gibt auch die Möglichkeit Zwischenstände einer DOP vor Systemausfällen mittels Checkpointing zu schützen. Darauf wird hier aber nicht weiter eingegangen.) Schon beendete DOPs oder auch ganze Design Activities können zurückgesetzt werden, indem alle Objektversionen gelöscht werden, die von zurückgesetzten DOPs erzeugt wurden. Normalerweise kann dieses Zurücksetzen lokal durchgeführt werden, da alle Objektänderungen in lokalen Versionen durchgeführt wurden und deshalb keine Schadensausbreitung stattgefunden haben kann. Allerdings gibt es zwei Ausnahmen: Eine Schadensausbreitung kann entlang der Delegation- und der Usage-Abhängigkeiten stattgefunden haben. In diesem Fall informiert das System die abhängigen Design Activities von der Ungültigkeit der Objektversionen, so daß diese dann entscheiden können, wie sie darauf reagieren. Die abhängige Design Activity kann zum Beispiel als Reaktion selbst abbrechen, einzelne DOPs zurücksetzen oder auch ihre Arbeit fortsetzen.

Die Fehler-Atomarität wird also solange garantiert, wie von einer Design Activity keine Delegation- oder Usage-Abhängigkeiten abgehen. Falls dies aber der Fall ist, kann die abhängige Design Activity entscheiden, inwieweit ein ausgebreiteter Fehler zurückgesetzt wird.

Eine weitere wichtige Eigenschaft aus dem Bereich der Fehlerbehandlung ist die sogenannte *Commit-Korrektheit*, die im Rahmen der dynamischen Aktionen bei [Nett, Mock 1995] definiert wurde. Diese Eigenschaft besagt, daß Änderungen von Transaktionen, die sich im Zustand *committed* befinden, unter keinen Umständen mehr zurückgesetzt werden müssen. Zum einen schließt das die ACID-Eigenschaft der Permanenz ein und zum anderen dürfen die geänderten Objektzustände auch nicht mehr aufgrund einer Schadensausbreitung zurückgesetzt werden müssen. Auf der Ebene der DOPs ist die Commit-Korrektheit nicht gewährleistet, da bei einem Abbruch der Design Activity die committeten Objektzustände durch die Invalidierung dieser Objektversion zurückgesetzt werden können. Dies führt allerdings zu keinen Problemen, da diese Zwischenversionen ohnehin nur lokal sichtbar waren und das System eventuelle Delegation- oder Usage-Beziehungen registriert und die abhängigen Design Activities informiert.

Globale Objektversionen, die aus einer Konfigurierung als Endergebnisse einer Design Activity entstanden sind, können grundsätzlich nicht mehr zurückgesetzt werden. Daher besteht nie die Gefahr, daß sich eine Design Activity unbeabsichtigt von einer anderen noch nicht beendeten Design Activity abhängig macht. Abhängigkeiten können nur durch gezielt eingegangene Delegation- oder Usage-Beziehungen entstehen. Da die globalen Objektversionen nicht mehr zurückgesetzt werden können, kann die Konsistenz der globalen Objektversionen nur dann gewährleistet werden, wenn jede Design Activity ihre Ergebnisse erst dann durch Konfigurierung in den globalen Objektspeicher zurückschreibt, wenn sie von keiner anderen aktiven Design Activity mehr abhängt. Dies wird aber nicht vom System garantiert, sondern jede Design Activity ist selbst für die Überwachung der Abhängigkeiten verantwortlich. Falls eine Design Activity ihre Ergebnisse „zu früh“ zurückschreibt, riskiert sie einen inkonsistenten globalen Objektzustand, falls eine Design Activity, von der sie abhängt, anschließend noch abbricht. Die Commit-Korrektheit wird für Design Activities also nicht garantiert.

Nebenläufigkeitskontrolle

Grundsätzlich ist systemseitige Nebenläufigkeitskontrolle nur in einem sehr geringen Umfang notwendig, da aufgrund der Versionierung jede DOP auf unterschiedlichen Objekten, bzw. genauer auf unterschiedlichen Versionen desselben Objekts arbeitet. Lediglich beim Erstellen einer initialen lokalen Version und bei der Konfigurierung müssen globale Objekte gesperrt werden. Das lokale Halten von Objektversionen garantiert auch eine Isolation der Design Activities, die nur dann aufgebrochen werden kann, wenn eine Design Activity bewußt eine Delegation- oder Usage-Abhängigkeit aufbaut. Mehrere Design Activities können nebenläufig dieselben Objekte bearbeiten, indem jede Design Activity eine lokale Version dieses Objekt erstellt und dieses bearbeitet. Es können auch sogenannte *Ableitungssperren* erworben werden. Diese verhindern, daß andere Design Activities nebenläufig lokale Versionen der Objekte erstellen und garantieren somit eine ausschließliche Bearbeitung eines Objekts durch eine einzige Design Activity.

Diese einfache Nebenläufigkeitskontrolle beinhaltet allerdings die Notwendigkeit von Konfigurierungen, also der Zusammenführung unterschiedlicher Versionen. Konfigurierungsprozeduren können in der Regel nicht systemseitig vorgegeben werden, sondern müssen vom Applikationsprogrammierer zur Verfügung gestellt oder vom Benutzer interaktiv durchgeführt werden. Ein Großteil der Nebenläufigkeitskontrolle wird im CONCORD Modell also nicht systemseitig gelöst, sondern durch die Konfigurierung in die Hand des Anwenders gelegt. Konfigurierungen, die die Konsistenz der globalen Objekte erhalten und zum Beispiel auch Nebenläufigkeitsanomalien ausschließen, sind im allgemeinen aber nicht trivial durchzuführen.

Auftrags- und Kooperationsprinzip

Sowohl das Auftrags- als auch das Kooperationsprinzip werden im CONCORD Modell realisiert. Eine Delegation-Abhängigkeit zwischen zwei Design Activities entspricht einer Auftraggeber-Auftragnehmer-Beziehung. Dabei werden Objektänderungen, also neue Versionen, des Auftragnehmers bei dessen Ende nur für den Auftraggeber sichtbar, wie es das Auftragsprinzip fordert. Das Kooperationsprinzip wird über die Usage-Beziehung realisiert. Usage-Beziehungen können zwischen beliebigen Design Activities aufgebaut werden und erlauben den vor anderen Design Activities geschützten Informationsfluß in eine Richtung. Beidseitiger Informationsfluß ist möglich, indem eine zweite Usage-Beziehung in Gegenrichtung aufgebaut wird.

Zusammenfassung

Das CONCORD Modell realisiert zwar das Auftrags- und das Kooperationsprinzip, allerdings macht das zugrundeliegende versionierte Objektsystem dieses Konzept für viele Anwendungsbereiche nur eingeschränkt geeignet: Entscheidende Teile der Nebenläufigkeitskontrolle werden nicht vom System gelöst, sondern durch die notwendige Konfigurierung der Applikation oder dem Anwender überlassen. Bei nebenläufiger Objektbearbeitung kann die Konfigurierung schnell so komplex werden, daß sie von einer programmierten Prozedur nicht geleistet werden kann, sondern von einem Anwender manuell durchgeführt werden muß. Auch liegt es in der Verantwortung des Anwenders, daß eine Konfigurierung erst dann durchgeführt wird, wenn keine Gefahr mehr besteht, daß die Objektzustände aufgrund von Abbrüchen *anderer* Design Activities zurückgesetzt werden müssen. Der einfache Mechanismus zur Fehlerbehandlung durch Reaktivierung alter Objektversionen wird nur dadurch möglich, daß der Anwender durch die Konfigurierung im Bereich der Nebenläufigkeitskontrolle einen erheblichen Aufwand leistet. Wie bei anderen Transaktionsmodellen zeigen sich auch hier die Nachteile einer fehlenden Orthogonalität von Fehlerbehandlung und Nebenläufigkeitskontrolle.

CONCORD erfordert zur Konsistenz-Erhaltung der Objekte oft Eingriffe durch den Benutzer, die in der Regel so komplex sind und soviel applikationsspezifisches Wissen benötigen, daß sie nicht vom Computer übernommen werden können. Daher eignet sich CONCORD eher für stark interaktive Arbeitsvorgänge einer beschränkten Anzahl Benutzer, aber nicht für umfangreiche verteilte Systeme, in denen die Konsistenz der Objekte in großem Maße durch die Systemverwaltung sichergestellt werden soll. Für ein sinnvolles und effektives Arbeiten mit einem solchen System müssen die Anforderungen bezüglich der Konsistenzerhaltung an den Anwender so gering wie möglich gehalten werden.

2.6 Dynamische Aktionen

Im letzten Abschnitt wurde am Beispiel des DOM Transaktionsmodells gezeigt, daß eine Integration von Auftrags- und Kooperationsprinzip nicht umfassend möglich ist, solange die Konzepte zur Nebenläufigkeitskontrolle und Fehlerbehandlung nicht unabhängig voneinander gewählt werden können. Da bei offen und geschlossen geschachtelten Transaktionen diese Unabhängigkeit nicht existiert, sind diese als Basis eines Integrationskonzepts ungeeignet. Das CONCORD Modell, das weder auf geschlossen noch auf offen geschachtelten Transaktionen basiert und statt dessen Fehlerbehandlung und Nebenläufigkeitskontrolle mit Hilfe eines versionierten Objektsystems unterstützt, erreicht zwar eine Integration der beiden Prinzipien, kann aber wichtige Teile der Nebenläufigkeitskontrolle und der Konsistenzerhaltung bei Fehlern nicht systemseitig lösen, sondern legt sie in großem Maße in die Verantwortung der Anwender.

Die in [Nett 1991] eingeführten *dynamischen Aktionen* setzen die Orthogonalität von Fehlerbehandlung und Nebenläufigkeitskontrolle durch. Unabhängig vom gewählten Fehlerbehandlungskonzept lassen sich so verschiedene Konzepte der Nebenläufigkeitskontrolle realisieren, wie zum Beispiel die unten beschriebenen „geschachtelten dynamischen Aktionen“ und die „dynamischen Aktionen für kooperative Anwendungen“.

Die dynamischen Aktionen setzen von den ACID-Eigenschaften (siehe Abschnitt 2.2) die Punkte der Fehler-Atomarität und der Permanenz des Effektes durch, ohne allerdings irgendwelche Anforderungen an die Isolation zu stellen. Hier liegt der grundlegende Unterschied zum klassischen Transaktionsmodell, in dem die Fehler-Atomarität nur bei Einhaltung der Isolations-Eigenschaft gewährleistet werden kann. Durch das Aufheben der Isolationsforderung bietet das Konzept der dynamischen Aktionen über verschiedene Mechanismen zur Nebenläufigkeitskontrolle die Möglichkeiten der vorzeitigen Freigabe von Ergebnissen und zur Kommunikation zwischen dynamischen Aktionen. Diese Flexibilisierung führt aber dazu, daß die Durchsetzung der Fehler-Atomarität für die Aktionsverwaltung mehr Aufwand bedeutet, als dies beim klassischem Transaktionskonzept der Fall ist. Allerdings werden bei dynamischen Aktionen keine weiteren Anforderungen an die Applikationen gestellt, im Gegensatz zu den in Abschnitt 2.4.1 vorgestellten SAGAs, bei denen zusätzlich zu den Transaktionen auch kompensierende Transaktionen existieren müssen.

2.6.1 Fehlerbehandlung bei dynamischen Aktionen

Durch die vorzeitige Freigabe von Änderungen ist es möglich, daß dynamische Aktionen Ergebnisse von anderen *laufenden* dynamischen Aktionen lesen und aufgrund dieser Ergebnisse Entscheidungen fällen oder diese Ergebnisse anders weiterverarbeiten. Im Falle des Abbruchs der schreibenden dynamischen Aktion ist der lesenden dynamischen Aktion die Grundlage für ihre Berechnungen entzogen, ihre Ergebnisse sind also ungültig und auch die lesende dynamische Aktion muß abgebrochen werden. Der Erfolg der lesenden dynamischen Aktion hängt also von dem des Schreibers ab. Umgekehrt darf deshalb eine dynamische Aktion erst dann in den Zustand *committed* übergehen, wenn alle dynamischen Aktionen, von denen sie direkt oder indirekt abhängt, bereits im Zustand *committed* sind.

Zwischen aktiven dynamischen Aktionen können also Abhängigkeiten entstehen, die für die Durchsetzung der Fehler-Atomarität sehr wichtig sind. Dies sind die sogenannten Informationsabhängigkeiten.

Informationsabhängigkeit

> Eine dynamische Aktion B ist von einer dynamischen Aktion A *informationsabhängig*, wenn B einen von A geschriebenen Objektzustand liest und A noch nicht committed oder abgebrochen ist.

Das Entstehen von Informationsabhängigkeiten führt zu einem *Fehlermodell*, welches zwischen drei Arten von Fehlern unterscheidet. Dabei treten die Fehler *Aktionsfehler* und *Stellenfehler* bereits beim klassischen Transaktionsmodell auf, während der *kaskadierende Aktionsfehler* erst bei dynamischen Aktionen betrachtet wird, da hier durch Informationsabhängigkeiten der Umfang einer Schadensausbreitung bestimmt werden kann.

Aktionsfehler: Eine einzelne dynamische Aktion wird abgebrochen. Dies kann zum einen durch die zugehörige Applikation geschehen, wenn diese zum Beispiel erkennt, daß die Konsistenz-Eigenschaft verletzt würde. Andererseits kann ein Aktionsfehler auch vom System zum Beispiel zur Auflösung eines Deadlocks ausgelöst werden.

Stellenfehler: In einem verteilten Aktionssystem fällt eine Stelle aus. Dies geschieht z. B. beim Ausfall des entsprechenden Rechners. Dabei wird das sogenannte *fail-stop*-Verhalten angenommen: Der Rechner hört sofort und komplett auf zu arbeiten. Permanent gespeicherte Informationen sind nach einem Neustart wieder vorhanden, flüchtige Informationen, insbesondere von laufenden dynamischen Aktionen gehen verloren. Ein Stellenfehler zieht Aktionsfehler aller im Moment des Ausfalls auf dieser Stelle laufenden dynamischen Aktionen nach sich.

Kaskadierender Aktionsfehler: Eine dynamische Aktion, die von einer anderen dynamischen Aktion informationsabhängig ist, muß bei einem Abbruch dieser anderen dynamischen Aktion auch abgebrochen werden. Dieser Abbruch heißt *kaskadierender Aktionsfehler*.

Um die Fehler-Atomarität zu gewährleisten, muß die Aktionsverwaltung also bei einem Aktions- oder Stellenfehler die Menge der dynamischen Aktionen bestimmen, die von einem kaskadierenden Aktionsfehler betroffen sind, und diese auch abbrechen. Eine erfolgreiche Bestimmung setzt voraus, daß die Aktionsverwaltung entstandene Informationsabhängigkeiten speichern und bei Bedarf auswerten kann. Zu diesem Zweck enthält die Aktionsverwaltung ein Modul zur Abhängigkeitsverwaltung. Die Abhängigkeiten werden dort bei ihrer Entstehung aufgenommen und wieder entfernt, wenn die beteiligten dynamischen Aktionen committen oder abbrechen. Diese Abhängigkeitsverwaltung ist in der Lage, Abhängigkeitsmengen, wie zum Beispiel die von einem Abbruch betroffenen dynamischen Aktionen, auch in verteilten Systemen konsistent zu bestimmen. In [Theisohn 1996] wurde ein erweitertes generisches Modul zur Abhängigkeitsverwaltung vorgestellt, das neben den Informationsabhängigkeiten auch beliebige andere Abhängigkeitstypen verwalten kann. Dadurch läßt sich dasselbe Modul auch in der Nebenläufigkeitskontrolle einsetzen, um zum Beispiel wie bei den unten beschriebenen dynamischen Aktionen für kooperative Anwendungen einen Serialisierungsgraphen aufzubauen und zu überprüfen.

Sicherung des Berechnungsfortschritts

Ein Ziel vieler Transaktionsmodelle ist die Vermeidung des Domino-Effektes. Unter dem Domino-Effekt versteht man das möglicherweise unbegrenzte Zurücksetzen von Berechnungen aufgrund eines einzigen Fehlers. Im klassischen Transaktionsmodell wird dieser Effekt durch die Isolations-Eigenschaft verhindert: Bei einem Abbruch einer Transaktion sind nur Berechnungen dieser Transaktion betroffen und diese werden auf einen definierten Wert zurückgesetzt, nämlich auf den Wert zu Beginn der Transaktion.

In [Nett 1991] und [Nett, Mock 1995] wurde gezeigt, daß der Domino-Effekt auch bei den dynamischen Aktionen trotz Aufhebung der Isolations-Eigenschaft und den deshalb möglichen kaskadierenden Aktionsfehlern nicht auftreten kann: Dynamische Aktionen garantieren die Existenz und Fortschreibbarkeit der sogenannten *Commitlinie*. Die Commitlinie repräsentiert dabei für alle Objekte den jeweils letzten Zustand, der nicht mehr aufgrund irgendeines Fehlers invalidiert werden kann, und beschreibt damit den gesicherten Berechnungsfortschritt. Durch das Commit-Ereignis einer dynamischen Aktion wird die Commitlinie für alle von ihr geänderten Objekte fortgeschrieben, da dynamische Aktionen die Eigenschaft der in [Nett, Mock 1995] eingeführten *Commit-Korrektheit* erfüllen, die besagt, daß keine dynamische Aktion nach ihrem Commit-Ereignis noch invalidiert werden kann.

Um die Commit-Korrektheit sicherzustellen, darf eine dynamische Aktion im Zustand *committed* nie von anderen, noch nicht committeten dynamischen Aktionen informationsabhängig sein. Dieses wird dadurch garantiert, daß ein Commit-Ereignis nur dann erfolgreich ist, wenn alle dynamischen Aktionen, von der die committende dynamische Aktion abhängt, bereits im Zustand *committed* sind, oder gleichzeitig mit ihr in diesen Zustand übergehen.

Zustände dynamischer Aktionen

Die Commit-Korrektheit führt also dazu, daß eine dynamische Aktion nicht autonom bei Ende ihrer Berechnungen in den Zustand *committed* übergehen kann, sondern gegebenenfalls darauf warten muß, daß dynamische Aktionen, von denen sie abhängt, bereit sind, auch in den Zustand *committed* zu wechseln. Bis zum Commit-Ereignis durchläuft eine dynamische Aktion also zwei Abschnitte: Die aktive Zeit, in der sie Berechnungen durchführt und auch selbständig abbrechen kann, und die Zeit, in der sie keine eigenen Berechnungen mehr durchführt und daher nur noch durch Stellen- und kaskadierende Aktionsfehler abgebrochen werden kann. Das Ende der eigenen Berechnungen wird durch den Wechsel in den Zwischenzustand *completed* markiert. Dieser in [Nett, Grosspietsch et al. 1985] eingeführte Zustand besagt, daß die dynamische Aktion keine Berechnungen mehr durchführt und bereit ist, in den Zustand *committed* zu wechseln. Dieser Wechsel findet aber erst dann statt, wenn die dynamische Aktion nicht mehr von einem kaskadierenden Aktionsfehler betroffen werden kann und wenn alle Objektänderungen zum Schutz vor Stellenfehlern permanent gespeichert sind. Es ergeben sich für dynamische Aktionen also die in Abbildung 2.2 gezeigten Zustände.

Nach der Beschreibung der Fehlerbehandlung sollen jetzt zwei wichtige Nebenläufigkeitskonzepte im Rahmen der dynamischen Aktionen dargestellt werden: die *geschachtelten dynamischen Aktionen* und die *dynamischen Aktionen für kooperative Anwendungen.*

2.6.2 Geschachtelte dynamische Aktionen

Die *geschachtelten dynamischen Aktionen* sind eine Möglichkeit der Nebenläufigkeitskontrolle innerhalb von dynamischen Aktionen. Mit ihnen wird das Auftragsprinzip realisiert, wobei eine Vater-

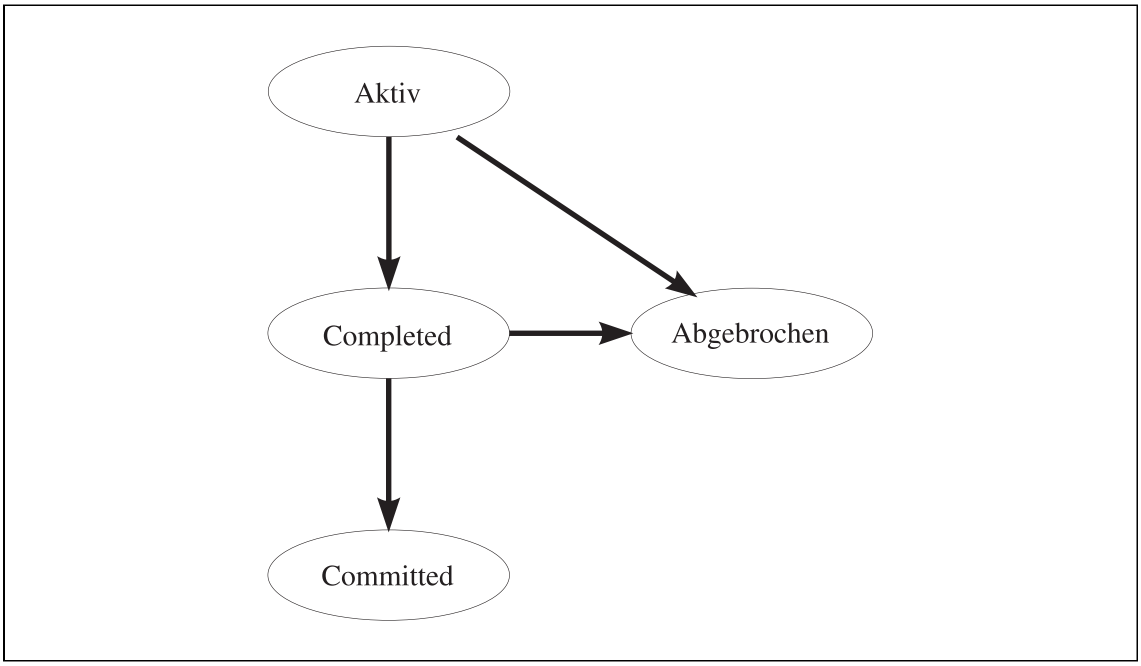


Abbildung 2.2: Zustände dynamischer Aktionen

Sohn-Beziehung die Auftraggeber-Auftragnehmer-Beziehung widerspiegelt. Eine ausführliche Beschreibung und Diskussion findet sich in [Nett 1991], [Nett, Weiler 1994] und [Bröhl 1995].

Geschachtelte dynamische Aktionen haben eine Baumstruktur, wie sie schon in [Moss 1981] für die geschlossen geschachtelten Transaktionen vorgeschlagen und in Abschnitt 2.3 beschrieben wurde. Im Gegensatz zu diesen, bei denen für jeden Transaktionsbaum weiterhin die Isolation garantiert wird, erlauben geschachtelte dynamische Aktionen die vorzeitige Freigabe von Ergebnissen. Hier wird lediglich zur Vermeidung von Nebenläufigkeitsanomalien das Konsistenzkriterium der *Serialisierbarkeit* von Aktionsbäumen gefordert.

Die geschlossen geschachtelten Transaktionen wurden eingeführt, um innerhalb von Transaktionen die Parallelität zu erhöhen und Transaktionen vor vollständigen Abbrüchen zu schützen. Dieser Schutz vor vollständigem Abbruch kommt dann zum Tragen, wenn bei verteilten Transaktionen eine Stelle ausfällt oder wenn die Applikation gezielt einen Teil der Berechnung abbrechen will. Zusätzlich hierzu bietet die Schachtelung von dynamischen Aktionen auch den Schutz vor kaskadierenden Aktionsfehlern, eine sehr wichtige Eigenschaft. Durch die vorzeitige Freigabe kann sich eine dynamische Aktion von einer anderen aktiven dynamischen Aktion abhängig machen. Wenn diese nun abgebrochen wird, muß aufgrund der Fehler-Atomarität auch die abhängige dynamische Aktion abgebrochen werden, wodurch möglicherweise lange Berechnungen, die gar nicht direkt von der abgebrochenen dynamischen Aktion abhängen, zurückgesetzt werden. Wurde aber von einem Sohn auf die fraglichen Objekte zugegriffen, so wird nur dieser abgebrochen, die anderen Berechnungen bleiben bestehen und die Applikation kann geeignet auf den Abbruch des Sohnes reagieren.

Serialisierbarkeit

Das Konsistenzkriterium für geschachtelte dynamische Aktionen ist die Serialisierbarkeit, wie sie in [Beeri, Bernstein et al. 1989] eingeführt wurde. Bei den geschlossen geschachtelten Transaktionen wurde noch die Isolation gefordert, die dazu führte, daß Objektänderungen erst am Ende der Toplevel-Transaktion allgemein sichtbar wurden. Serialisierbarkeit hingegen erlaubt, daß Objektänderungen schon früher sichtbar gemacht werden können, wodurch die Nebenläufigkeit von dynamischen Aktionen stark erhöht werden kann. Serialisierbarkeit wird garantiert, wenn sich dynamische Aktionen zweiphasig verhalten, wie es weiter unten bei der Diskussion des Sperrverfahrens beschrieben wird.

Bei geschachtelten dynamischen Aktionen müssen Aktionsbäume und Aktionsteilbäume, bei denen nicht einer den anderen enthält, untereinander serialisierbar sein. Das bedeutet im einzelnen:

Toplevel-Aktionen einschließlich aller ihrer Nachfahren, also vollständige Aktionsbäume, müssen serialisierbar sein.

Brüder einschließlich aller ihrer Nachfahren müssen serialisierbar sein.

Väter und Söhne müssen nicht miteinander serialisierbar sein.

Genau wie Toplevel-Aktionen können Brüder unabhängig parallel voneinander ablaufen, sofern sie nicht auf gemeinsame Objekte zugreifen.

Aktionsbäume müssen nach außen weiterhin wie eine einzige flache dynamische Aktion serialisierbar sein. Aus diesem Grund dürfen Zwischenergebnisse von Sohnaktionen nicht direkt für alle dynamischen Aktionen sichtbar werden, sondern zuerst nur für den Vater. Dieser kann dann die Änderungen entweder weiter verbergen, um zum Beispiel diese durch einen weiteren Sohn bearbeiten zu lassen, oder in der Schachtelungshierarchie für weitere Vorfahren sichtbar machen. Erst wenn eine Toplevel-Aktion eine Änderung sichtbar macht, ist sie für alle anderen dynamischen Aktionen auch sichtbar.

Auftrags- und Kooperationsprinzip

Durch die eben beschriebenen Sichtbarkeitseigenschaften realisieren geschachtelte dynamische Aktionen das Auftragsprinzip. Das Kooperationsprinzip kann durch geschachtelte dynamische Aktionen nicht realisiert werden, da die durchgesetzte Serialisierbarkeit von dynamischen Aktionen einer Ebene Informationsfluß nur in eine Richtung zuläßt. Das Kooperationsprinzip fordert aber zwischen Kooperationspartnern Informationsfluß in beide Richtungen.

Fehlerbehandlung

Für Subaktionen wird die Fehler-Atomarität garantiert. Um auch die Permanenz garantieren zu können, darf eine dynamische Aktion erst dann in den Zustand *committed* übergehen, wenn sie garantiert nicht mehr abgebrochen werden kann. Eine Sohnaktion kann aber solange noch abgebrochen werden, wie ihr Vater noch nicht committed hat. Der Sohn geht bei seinem Ende deshalb zuerst in den Zustand *completed* über und wechselt erst zusammen mit dem Vater nach *committed*.

Bei der Behandlung von kaskadierenden Aktionsfehlern muß zusätzlich zum oben beschriebenen Verfahren für Väter und Söhne folgende besondere Beziehung beachtet werden: Der Abbruch eines Vaters führt immer zum Abbruch aller Söhne, wohingegen der Abbruch eines Sohnes nicht erzwingt, daß der Vater auch abbricht.

Geschachteltes Sperrverfahren

Die oben beschriebenen Regeln zur Serialisierbarkeit und Sichtbarkeit werden durch Sperren (engl. „locks“) realisiert. Es gibt Schreib- und Lesesperren, wobei im allgemeinen eine Schreibsperre nicht gleichzeitig mit einer Schreib- oder Lesesperre für eine andere dynamische Aktion auf dasselbe Objekt vergeben werden darf: Eine Schreibsperre steht mit anderen Sperren im *Konflikt*. Falls eine dynamische Aktion A_1 also bereits eine Sperre auf ein Objekt hält und A_2 auch eine Sperre für dieses Objekt anfordert, wird sie ihr entsprechend der Kompatibilitätsmatrix in Abbildung 2.3 gewährt. Falls A_1 und A_2 in einem Vorfahre-Nachfahre-Verhältnis stehen, gelten allerdings die unten beschriebenen erweiterten Regeln.

		dyn. Aktion A_2 fordert an	
		Schreib-Sperre	Lese-Sperre
dynamische Aktion A_1 hält	Schreib-Sperre	abgelehnt	abgelehnt
	Lese-Sperre	abgelehnt	gewährt

Abbildung 2.3: Kompatibilitätsmatrix für die Gewährung von Sperren

Serialisierbarkeit läßt sich nach [Eswaran 1976] erreichen, indem Sperren nach dem sogenannten *nicht-strikten Zwei-Phasen-Sperrprotokoll* vergeben werden. Die aktive Zeit einer dynamischen Aktion läßt sich hiernach in zwei Phasen einteilen: die wachsende Phase, in der eine dynamische Aktion Sperren nur anfordern, und die schrumpfende Phase, in der sie die Sperren nur freigegeben darf. Der Wechsel zwischen beiden Phasen wird *Lockpunkt* genannt. Sowohl Toplevel- als auch Subaktionen müssen sich konform zum nicht-strikten Zwei-Phasen-Sperrprotokoll verhalten.

Um zu verhindern, daß Zwischenergebnisse zu früh „nach außen“ sichtbar werden, werden Sperren bei der Freigabe durch eine Sohnaktion nicht tatsächlich freigegeben, sondern an den nächsten Vorfahren vererbt, der noch nicht den Lockpunkt erreicht hat (siehe hierzu [Nett, Weiler 1994]).

Es ist möglich, daß Väter und Söhne gleichzeitig in Konflikt stehende Sperren auf dasselbe Objekt besitzen. Zur Synchronisation zwischen den dynamischen Aktionen und auch bei mehreren parallelen Prozessen innerhalb einer dynamischen Aktion wird eine zweite Ebene von Sperren eingeführt, die *Benutzungssperren*, für die die gleichen Konfliktregeln gelten.

Sperren werden nach folgenden Regeln gewährt und freigegeben:

1. **(a)** Eine dynamische Aktion darf nur dann für ein Objekt eine Schreib-Sperre bekommen, wenn sie sich in der ersten Phase befindet und alle anderen Besitzer von Lese- oder Schreib-Sperren auf dieses Objekt Vorfahren dieser dynamischen Aktion sind.

(b) Eine dynamische Aktion darf nur dann für ein Objekt einen Lese-Sperre bekommen, wenn sie sich in der ersten Phase befindet und alle anderen Besitzer von Schreib-Sperren auf dieses Objekt Vorfahren dieser dynamischen Aktion sind.

2. Eine dynamische Aktion darf eine gehaltene Sperre freigeben, wenn sie sich in der zweiten Phase befindet und sie keine Benutzungssperre auf dieses Objekt besitzt.

Sofern die dynamische Aktion einen Vorfahren hat, der sich noch in der ersten Phase befindet, erbt der nächste Vorfahre die Sperre. Hält dieser Vorfahre bereits eine Sperre, besitzt er dann die stärkere. (Eine Schreibsperre ist stärker als eine Lesesperre.)

3. **(a)** Eine dynamische Aktion darf nur dann für ein Objekt eine Schreib-Benutzungssperre bekommen, wenn sie für dieses Objekt bereits eine Schreib-Sperre aber kein Nachfahre eine Schreib- oder Lesesperre hält und keine andere dynamische Aktion darauf eine Schreib- oder Lese-Benutzungssperre hält.

(b) Eine dynamische Aktion darf nur dann für ein Objekt einen Lese-Benutzungssperre bekommen, wenn sie für dieses Objekt bereits eine Schreib- oder Lese-Sperre aber kein Nachfahre eine Schreibsperre hält und keine andere dynamische Aktion darauf eine Schreib-Benutzungssperre hält.

4. Gehaltene Benutzungssperren dürfen immer freigegeben werden.

5. Wenn eine dynamische Aktion abbricht, werden alle ihre Sperren und Benutzungssperren gelöscht und es werden keine Sperren vererbt.

Ferner wird vorausgesetzt, daß Applikationen sich konform zu den Sperren verhalten: Zugriffe auf die Objekte dürfen nur stattfinden, wenn die entsprechende dynamische Aktion die entsprechenden Sperren besitzt.

2.6.3 Dynamische Aktionen für kooperative Anwendungen

Eine weitere Möglichkeit des Einsatzes von dynamischen Aktionen sind die in [Mock 1994] eingeführten *dynamischen Aktionen für kooperative Anwendungen.* Mit ihnen wird das Kooperationsprinzip sinnvoll verwirklicht: Serialisierbarkeit kann gezielt zwischen den dynamischen Aktionen aufgehoben werden, die von der Anwendungssemantik her kooperieren sollen.

Dynamische kooperative Serialisierbarkeit

Serialisierbarkeit ist eine wichtige Eigenschaft, die unerwünschte Effekte wie die Nebenläufigkeitsanomalien vermeidet. Allerdings verhindert sie auch Kooperation, die typischerweise zu folgenden nicht-serialisierbaren Abläufen führt: Aktion A ändert ein Objekt, Aktion B liest die Änderung und schreibt erneut einen Wert, den Aktion A wieder liest. Die Aufhebung der Serialisierbarkeit ist für Kooperation also notwendig, erfordert aber, daß der Applikationsprogrammierer das konsistente Verhalten der dynamischen Aktion bezüglich anderer, nicht-serialisierbarer dynamischer Aktionen sicherstellen muß. Da dieses zwar für bekannte Kooperationspartner möglich, im allgemeinen aber nicht zu leisten ist, muß die Serialisierbarkeit so weit wie möglich durch die Aktionsverwaltung garantiert und nur gezielt für kooperierende dynamische Aktionen aufgehoben werden. Kooperierende dynamische Aktionen werden der Aktionsverwaltung durch *Kooperationsgruppen* mitgeteilt. In [Mock 1994] wird auch die Möglichkeit individueller Kooperationsfunktionen besprochen, die das Kooperationsverhalten jeder einzelnen dynamische Aktion festlegen. Da Kooperationsgruppen in der Praxis leichter zu realisieren sind und meistens das gewünschte Kooperationsverhalten ausreichend genau beschreiben, wird die dynamische kooperative Serialisierbarkeit hier auf Basis der Kooperationsgruppen eingeführt.

Kooperationsgruppe

Eine Kooperationsgruppe ist eine Menge von dynamischen Aktionen, die kooperieren, und zwischen denen deshalb keine Serialisierbarkeit gefordert wird. Jede dynamische Aktion muß vor ihrem Commit einer Kooperationsgruppe beitreten, wobei auch einelementige Kooperationsgruppen zugelassen sind. Kooperationsgruppen und damit auch dynamischen Aktionen aus unterschiedlichen Kooperationsgruppen sind untereinander serialisierbar.

Serialisierbarkeit wird hier nicht mehr mit Hilfe von Zweiphasigkeit durchgesetzt, sondern dynamisch überprüft und durch Einschränkung der zugreifbaren Objekte in bestimmten Phasen sichergestellt. Zur Überprüfung der Serialisierbarkeit wird in der ohnehin zur Verwaltung von dynamischen

Aktionen vorhandenen Abhängigkeitsverwaltung (siehe [Theisohn 1996]) mit Hilfe der sogenannten *Konfliktabhängigkeiten* der Serialisierungsgraph aufgebaut.

Konfliktabhängigkeit

Zwei Operationen auf einem Objekt stehen dann miteinander in Konflikt, wenn ihre Ergebnisse abhängig von ihrer Reihenfolge sind. Als Ergebnisse gelten sowohl die Rückgabewerte (zum Beispiel von Leseoperationen) als auch der Zustand des Objektes. In einer Schreib-/Lese-Semantik stehen zwei Operationen auf einem Objekt genau dann in Konflikt, wenn mindestens eine von ihnen eine Schreiboperation ist.

Zwischen zwei dynamischen Aktionen existiert eine Konfliktabhängigkeit, wenn sie jeweils Operationen ausführen, die in Konflikt stehen. Die Abhängigkeit läuft in Richtung des zeitlichen Auftretens der Operationen.

Nach [Bernstein, Hadzilacos et al. 1987] ist eine Ausführungsfolge genau dann serialisierbar, wenn der Serialisierungsgraph zyklenfrei ist. Um die gewünschte Serialisierbarkeit von Kooperationsgruppen zu gewährleisten, dürfen Zyklen nur zwischen dynamischen Aktionen derselben Kooperationsgruppe auftreten.

Die dynamische kooperative Serialisierbarkeit wird durchgesetzt, indem eine dynamische Aktion *drei Phasen* durchläuft, die durch die Ereignisse *Bildung der Kooperationsgruppe* und *Validierung* getrennt sind:

1. Phase

Zu Anfang ist eine dynamische Aktion noch nicht Mitglied einer Kooperationsgruppe. Sie darf auf alle Objekte zugreifen, entstehende Konfliktabhängigkeiten werden vom System registriert.

Ereignis: Bildung der Kooperationsgruppe

Dynamische Aktionen, die miteinander kooperieren wollen, bilden eine Kooperationsgruppe. Die Aktionsverwaltung überprüft, ob alle Zyklen, an denen dynamische Aktionen dieser Kooperationsgruppe beteiligt sind, komplett innerhalb der Kooperationsgruppe liegen. Ist dies nicht der Fall, muß die Kooperationsgruppe erweitert, oder einige an den Zyklen beteiligte dynamische Aktionen müssen abgebrochen werden.

2. Phase

In dieser Phase dürfen dynamische Aktionen Operationen auf denjenigen Objekten ausführen, bei denen die letzte zu dieser im Konflikt stehende Operation von einer dynamischen Aktion ausgeführt wurde, die entweder aus derselben Kooperationsgruppe stammt oder bereits validiert ist.

Ereignis: Validierung

Wenn eine Kooperationsgruppe validiert, d. h. alle dynamischen Aktionen dieser Kooperationsgruppe validieren, überprüft die Aktionsverwaltung, ob alle anderen dynamischen Aktionen, von denen sie abhängen, bereits validiert sind. Findet sich eine nicht-validierte dynamische Aktion, schlägt die Validierung fehl.

3. Phase

In dieser Phase dürfen dynamischen Aktionen Operationen auf denjenigen Objekten ausführen, bei denen die letzte zu dieser im Konflikt stehende Operation von einer dynamischen Aktion aus derselben Kooperationsgruppe ausgeführt wurde.

Von Phase zu Phase wird die Menge der zugreifbaren Objekte mehr und mehr eingeschränkt, wie Abbildung 2.4 veranschaulicht. Dies geschieht, um die bei der Bildung von Kooperationsgruppen sichergestellte Zyklenfreiheit mit dynamischen Aktionen außerhalb der Kooperationsgruppe weiter garantieren zu können.

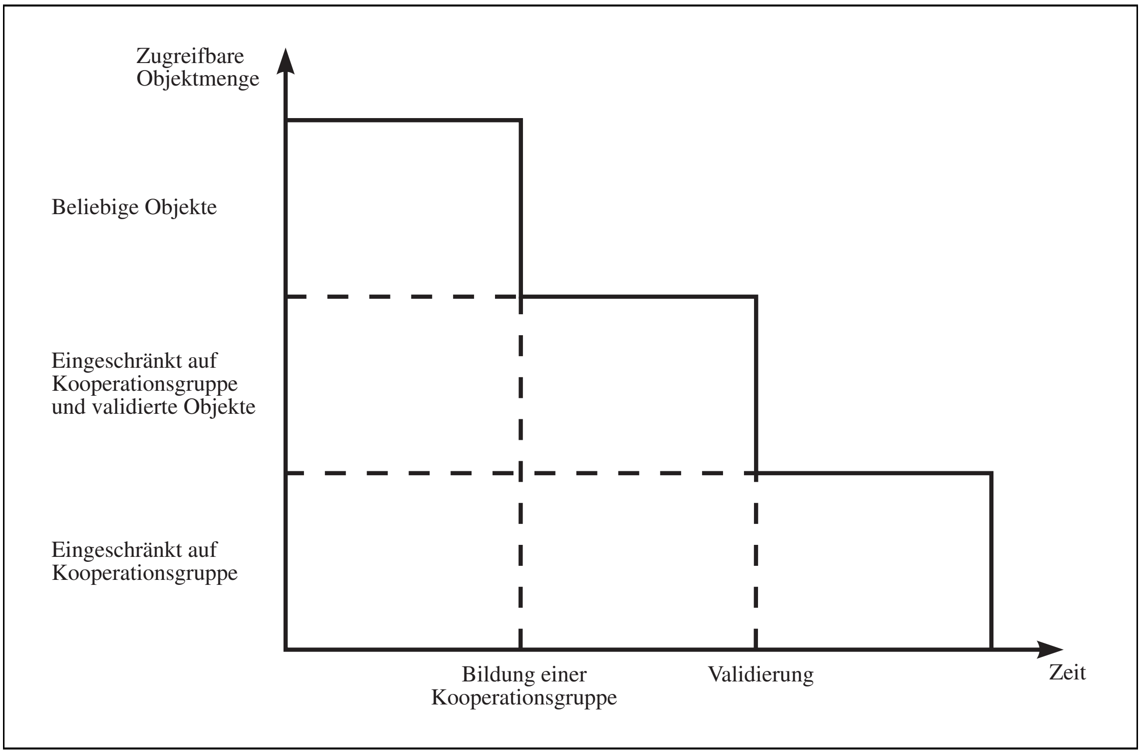


Abbildung 2.4: Phasenmodell des kooperativen Sperrverfahrens

Wenn eine dynamische Aktion in die zweite Phase gelangt, liegen garantiert alle Zyklen, an denen sie beteiligt ist, innerhalb ihrer Kooperationsgruppe. In der zweiten Phase darf eine dynamische Aktion sich nur noch von dynamischen Aktionen abhängig machen, die in der eigenen Kooperationsgruppe liegen oder schon validiert sind. Validierte dynamischen Aktionen hängen aber nur noch von anderen validierten dynamischen Aktionen ab und werden die Menge der dynamischen Aktionen, von denen sie abhängen, auch nicht mehr vergrößern. Da die dynamische Aktion selbst noch nicht validiert ist, kann so kein Zyklus mehr entstehen. Durch die Festlegung der Abhängigkeitsmenge bei der Validierung ist auch das Entstehen weiterer Zyklen in der dritten Phase ausgeschlossen.

Auftrags- und Kooperationsprinzip

Das Kooperationsprinzip wird durch die dynamischen Aktionen für kooperative Anwendungen realisiert, da die Eigenschaft der dynamischen kooperativen Serialisierbarkeit garantiert, daß beidseitiger Informationsfluß zwischen dynamischen Aktionen innerhalb einer Kooperationsgruppe möglich ist. Zwischen dynamischen Aktionen aus unterschiedlichen Kooperationsgruppen ist der Informationsfluß nur in eine Richtung möglich. Das Auftragsprinzip kann nicht realisiert werden, da eine Auftraggeber-Auftragnehmer-Beziehung zwischen dynamischen Aktionen erfordert, daß Ergebnisse von Auftragnehmern nur für den Auftraggeber sichtbar sind. Im Konzept der dynamischen Aktionen für kooperative Anwendungen sind Ergebnisse einer dynamischen Aktion aber immer für alle anderen dynamischen Aktionen sichtbar.

Fehlerbehandlung

Die oben vorgestellten Mechanismen zur Fehlerbehandlung bei dynamischen Aktionen reichen vollkommen aus, um diese auch im Fall von Kooperation adäquat durchzuführen.

Da parallel zu den Konflikt- oft auch Informationsabhängigkeiten entstehen, kann es auch dort zu Zyklen kommen. Die Eigenschaft der Commit-Korrektheit verhindert, daß in einem Zyklus eine dynamische Aktion vor den anderen in den Zustand *committed* wechselt, da sie noch von einem kaskadierenden Aktionsfehler betroffen werden kann. Dynamische Aktionen innerhalb von Zyklen können nur in den Zustand *committed* übergehen, indem sie alle einen sogenannten *Gruppen-Commit* durchführen. Beim Gruppen-Commit wechseln alle dynamischen Aktionen gleichzeitig ihren Zustand, wodurch die Commit-Korrektheit eingehalten werden kann.

Kooperatives Sperrverfahren

Das Konzept der dynamischen Aktionen für kooperative Anwendungen wird durch das folgende kooperative Sperrverfahren realisiert, das auf der bekannten Schreib-/Lese-Semantik basiert. Applikationen dürfen nur auf Objekte zugreifen, wenn die zugehörige dynamische Aktion eine entsprechende Sperre besitzt, d. h. die Applikation muß sich zu den Sperren konform verhalten.

Den Wechsel der Phasen muß die dynamische Aktion der Aktionsverwaltung durch zwei entsprechende Operationen mitteilen. Diese Operationen haben nicht nur reinen Informationscharakter, wie zum Beispiel das Setzen des Lockpunktes beim geschachtelten Sperrverfahren (Abschnitt 2.6.2), sondern können in der eben beschriebenen Situation auch fehlschlagen. Für eine genaue Beschreibung beispielsweise des Validierungsprotokolls siehe [Mock 1994].

Die Sperren werden nach folgenden Regeln gewährt und freigegeben:

1. **(a)** Eine dynamische Aktion darf nur dann für ein Objekt eine Schreib-Sperre bekommen, wenn keine andere dynamische Aktion eine Lese- oder Schreib-Sperre auf dieses Objekt besitzt.

  (b) Eine dynamische Aktion darf nur dann für ein Objekt einen Lese-Sperre bekommen, wenn keine andere dynamische Aktion eine Schreib-Sperre auf dieses Objekt besitzt.

2. Sperren werden in Abhängigkeit von der Phase, in der sich die anfordernde dynamische Aktion befindet, vergeben:

  (a) Sperren dürfen in der ersten Phase beliebig angefordert werden.

  (b1) In der zweiten Phase dürfen nur solche Lese-Sperren angefordert werden, bei denen der letzte Schreiber in derselben Kooperationsgruppe liegt oder bereits validiert ist.

  (b2) In der zweiten Phase dürfen nur solche Schreib-Sperren angefordert werden, bei denen der letzte Halter einer Schreib- oder Lese-Sperre in derselben Kooperationsgruppe liegt oder bereits validiert ist.

  (c1) In der dritten Phase dürfen nur noch Lese-Sperren angefordert werden, bei denen der letzte Schreiber oder der letzte Halter einer Lese-Sperre in derselben Kooperationsgruppe liegt.

  (c2) In der dritten Phase dürfen nur noch Schreib-Sperren angefordert werden, bei denen der letzte Halter einer Schreib- oder Lese-Sperre in derselben Kooperationsgruppe liegt.

3. Eine dynamische Aktion darf eine gehaltene Sperre immer freigeben.

Kapitel 3

Geschachtelte dynamische Aktionen für kooperative Anwendungen

3.1 Einleitung

Im vorherigen Kapitel wurden zwei auf dynamischen Aktionen basierende Nebenläufigkeitskonzepte diskutiert: Die *geschachtelten dynamischen Aktionen* (siehe Abschnitt 2.6.2) zur Realisierung des Auftragsprinzips und die *dynamischen Aktionen für kooperative Anwendungen* (Abschnitt 2.6.3), mit denen das Kooperationsprinzip realisiert wird. In diesem Kapitel wird ein neues, auch auf dynamischen Aktionen basierendes Konzept vorgestellt, daß diese beiden Prinzipien verbindet. Dynamische Aktionen eignen sich für eine Integration dieser zwei Konzepte besonders gut, da sie strikt zwischen Fehlerbehandlung und Nebenläufigkeitskontrolle trennen. Zur Fehlerbehandlung werden durch die dynamische Aktion schon alle nötigen Mechanismen zur Verfügung gestellt. Dadurch ist es möglich, sich bei der Integration ganz auf die Behandlung der Nebenläufigkeitskontrolle zu konzentrieren, ohne daß Seiteneffekte auf die Fehlerbehandlung zu befürchten sind. Bei der Diskussion des hier neu entwickelten und der beiden zugrundeliegenden Aktionskonzepte brauchen daher immer nur die entsprechenden Nebenläufigkeitskonzepte betrachtet zu werden.

Die Integration findet auf drei Ebenen statt, von denen zwei in diesem Kapitel behandelt werden: Die konzeptuelle Ebene und die Ebene der Sperren. Die Implementierungsebene wird in den Kapiteln 5 und 6 besprochen.

Zuerst wird untersucht, welche Probleme auftauchen, wenn man versucht, beide Konzepte ohne Modifizierung zu kombinieren, und es wird gezeigt, daß eine triviale Kombination nicht möglich ist.

Um ein neues Konzept zu entwickeln, welches Kooperations- und Auftragsprinzip vereinigt, wird dann betrachtet, welche Eigenschaften der beiden Nebenläufigkeitskonzepte der geschachtelten dynamischen Aktionen und der dynamischen Aktionen für kooperative Anwendungen bei der Durchsetzung der zugrundeliegenden Prinzipien benötigt werden. Ausgehend von diesen Eigenschaften wird das Integrationskonzept der *geschachtelten dynamischen Aktionen für kooperative Anwendungen* vorgestellt. Es zeigt sich, daß entscheidend für die Anwendung des jeweiligen Konzepts ist, zu welchen Zeitpunkten und für welche dynamischen Aktionen Objektänderungen sichtbar werden. Zu diesem Zweck enthält das Integrationskonzept den Mechanismus der Objektkontrolle, mit dem die Sichtbarkeit von Objektänderungen individuell für jedes Objekt gesteuert werden kann.

Das kooperative Sperrverfahren für geschachtelte dynamische Aktionen realisiert das Konzept der geschachtelten dynamischen Aktionen für kooperative Anwendung. Es basiert auf dem geschachtelten Sperrverfahren (siehe Abschnitt 2.6.2) und dem kooperativen Sperrverfahren (siehe Abschnitt 2.6.3).

Das in dieser Arbeit eingeführte kooperative Sperrverfahren für geschachtelte dynamische Aktionen stellt somit eine Erweiterung des kooperativen Sperrverfahrens aus [Mock 1994] (Abschnitt 2.6.3) dar. Deshalb wird es genau wie das Verfahren aus [Mock 1994] kurz „kooperatives Sperrverfahren“ genannt. Um Verwechslungen mit dem kooperativen Sperrverfahren für flache dynamische Aktionen aus [Mock 1994] zu vermeiden, wird auf letzteres mit dem Begriff „flaches kooperatives Sperrverfahren“ Bezug genommen.

Nachdem die Integration beider Prinzipien auf den unterschiedlichen Ebenen gezeigt wurde, geht es am Ende dieses Kapitels darum zu zeigen, daß das kooperative Sperrverfahren sowohl mit dem flachen kooperativen Sperrverfahren als auch mit dem geschachtelten Sperrverfahren in großem Maße verträglich ist. Es wird angegeben, wie diese beiden Verfahren im kooperativen Sperrverfahren simuliert werden können.

3.2 Probleme bei der Kombination von geschachtelten dynamischen Aktionen und dynamischen Aktionen für kooperative Anwendungen

Konzepte zur Realisierung des Auftrags- und des Kooperationsprinzips zu integrieren ist nicht trivial, da beide Prinzipien unterschiedliche – teilweise sogar gegensätzliche – Eigenschaften der Konzepte fordern. Nachdem zuerst gezeigt wird, warum sich die Konzepte der geschachtelten dynamischen Aktionen und der dynamischen Aktionen für kooperative Anwendungen nicht einfach durch Kombination aller Eigenschaften integrieren lassen, wird genauer untersucht, wie die Konzepte jeweils durchgesetzt werden. Dazu wird nach einer genauen Betrachtung der Charakteristika beider Prinzipien herausgearbeitet, welche Eigenschaften der beiden Konzepte *grundlegend* für die Durchsetzung des jeweiligen Prinzips sind, und welche Eigenschaften gebraucht werden, um eine nötige Konsistenz des Konzepts zu gewährleisten. Die herausgearbeiteten Eigenschaften bilden die Grundlage für das im nächsten Abschnitt beschriebene Integrationskonzept.

3.2.1 „Naiver“ Versuch der Kombination

Die dynamischen Aktionen für kooperative Anwendungen realisieren das Kooperationsprinzip und die geschachtelten dynamischen Aktionen das Auftragsprinzip. Um beide Prinzipien zu integrieren, wäre der direkteste Weg, ein Aktionskonzept zu entwickeln, welches alle Eigenschaften beider Konzepte in sich vereinigt. Dies ist allerdings nicht möglich, da sich einige Eigenschaften beider Konzepte gegenseitig ausschließen.

Strukturell lassen sich beide Konzepte miteinander kombinieren: Das integrierte Konzept besitzt sowohl eine Baumstruktur der dynamischen Aktionen als auch eine Aufteilung der dynamischen Aktionen in Kooperationsgruppen. Wie das Verhältnis von Aktionsbäumen und Kooperationsgruppen ist, muß eine genauere Betrachtung der weiteren Anforderungen ergeben. Folgende Alternativen sind denkbar: Aktionsbäume könnten zum Beispiel komplett in Kooperationsgruppen enthalten sein müssen, sie könnten eine Eins-zu-eins-Beziehung haben oder sie könnten auch orthogonal zueinander sein.

Bezüglich anderer Eigenschaften treten hingegen bei der Integration Probleme auf:

Serialisierbarkeit

Bei den geschachtelten dynamischen Aktionen sind Aktionsbäume und auch Teilbäume einer Ebene untereinander serialisierbar. Vor- und Nachfahren müssen nicht serialisierbar sein. Bei dynamischen Aktionen für kooperative Anwendungen wird Serialisierbarkeit zwischen Kooperationsgruppen gefordert. Wenn ein Aktionsbaum einer Kooperationsgruppe entspräche, wäre die Serialisierbarkeit von vollständigen Aktionsbäumen gewährleistet. Allerdings könnte die Serialisierbarkeit von Teilbäumen nicht garantiert werden, da innerhalb von Kooperationsgruppen Kommunikation jeder Art möglich ist. Es ist auch nicht möglich, die Granularität der Kooperationsgruppen feiner zu wählen und für einzelne Subaktionen getrennte Kooperationsgruppen zu bilden, da so die Kommunikation zwischen Vater und Sohn unterbunden würde. Außerdem ist das Ziel der dynamischen Aktionen für kooperative Anwendungen, Kooperation zwischen beliebig wählbaren Partnern zu ermöglichen. Diese Partner dürfen in einem kombinierten Konzept nicht auf dynamische Aktionen desselben Aktionsbaums beschränkt sein, so daß es auch möglich sein muß, daß mehrere Aktionsbäume in einer Kooperationsgruppe liegen.

Die unterschiedlichen Serialisierbarkeitsanforderungen in beiden Konzepten lassen sich nicht trivial kombinieren.

Beide Konzepte unterscheiden sich auch in der Art, wie die Serialisierbarkeit durchgesetzt wird. Bei geschachtelten dynamischen Aktionen geschieht dieses durch das *nicht-strikte Zweiphasen-Sperrprotokoll*. Halten sich dynamische Aktionen an dieses Protokoll, ist Serialisierbarkeit gesichert. Bei den dynamischen Aktionen für kooperative Anwendungen wird Serialisierbarkeit hingegen dynamisch überprüft und zur ihrer Gewährleistung müssen sich die dynamischen Aktionen entsprechend einem *dreiphasigen Schema* verhalten. Dabei wird die Menge der zugreifbaren Objekte von Phase zu Phase so eingeschränkt, daß die bei den Phasenwechseln überprüfte Serialisierbarkeit der Kooperationsgruppen nicht durch neu entstehende unzulässige Konfliktabhängigkeiten verletzt wird.

Sichtbarkeit von Objektänderungen

Im geschachtelten Konzept werden alle Objektänderungen einer Subaktion zunächst nur für ihren Vater bzw. den nächsten Vorfahren in der ersten Phase sichtbar. Zum einen wird durch diese Einschränkung der Sichtbarkeit die Existenz des Auftragnehmers nach außen verborgen und damit das Auftragsprinzip durchgesetzt. Zum anderen könnte die Serialisierbarkeit von kompletten Aktionsbäumen nicht mehr garantiert werden, falls Objektänderungen von Subaktionen zu früh von dynamischen Aktionen aus anderen Aktionsbäumen gelesen werden könnten.

Im kooperativen Konzept werden Objektänderungen von dynamischen Aktionen allgemein sichtbar. Es gibt keine Mechanismen, die die Menge der dynamischen Aktionen, die auf geänderte Objekte zugreifen können, einschränken. Zur Durchsetzung des Kooperationsprinzips ist es auch zwingend notwendig, daß Objektänderungen früh sichtbar werden, da nur so ein beidseitiger Informationsfluß zwischen laufenden dynamischen Aktionen möglich ist.

Auch bezüglich der Sichtbarkeitsregeln ist keine direkte Kombination möglich. Abhängig davon, ob eine dynamische Aktion gerade einen Auftrag ausführt oder ob sie mit einer anderen dynamischen Aktion kooperieren will, werden unterschiedliche Anforderungen an die Sichtbarkeit gestellt.

3.2.2 Für das Kooperationsprinzip relevante Eigenschaften von dynamischen Aktionen für kooperative Anwendungen

Das Konzept der dynamischen Aktionen für kooperative Anwendungen wird im folgenden danach untersucht, welche seiner Eigenschaften für die Durchsetzung des Kooperationsprinzip relevant sind und deshalb in das integrierte Konzept übernommen werden sollten. Das Kooperationsprinzip besagt dabei folgendes:

Kooperationsprinzip

> Mehrere dynamische Aktionen bearbeiten *kooperativ* Objekte. Dies bedeutet, daß Objektänderungen aufgrund von gemeinsamen Entscheidungen der dynamischen Aktionen zustande kommen können. Um diese Entscheidungen treffen zu können, müssen zwischen den beteiligten dynamischen Aktionen Informationen ausgetauscht werden. Dies geschieht über die Objekte, indem zum Beispiel solche Abläufe auftreten: Eine laufende dynamische Aktion liest Objektänderungen einer anderen laufenden dynamischen Aktion und macht auch selber Änderungen, die diese dynamische Aktion wieder liest. Kooperierende dynamische Aktionen sind in der Regel also nicht serialisierbar. Dynamische Aktionen, die nicht miteinander kooperieren wollen, müssen serialisierbar bleiben.

Zur Durchsetzung des Kooperationsprinzips setzen die dynamischen Aktionen für kooperative Anwendungen folgende Forderungen durch:

- Es muß eine Möglichkeit geben, dem System mitzuteilen, welche dynamischen Aktionen kooperieren wollen und welche nicht. Dies geschieht durch die Zugehörigkeit der dynamischen Aktionen zu *Kooperationsgruppen*.
- Änderungen einer dynamischen Aktion müssen frühzeitig freigegeben werden können, damit nebenläufige Kooperationspartner diese Änderungen lesen können und somit ein Informationsfluß stattfinden kann.
- Zwischen Kooperationspartnern wird keine Serialisierbarkeit gefordert. Dynamische Aktionen, die nicht miteinander kooperieren wollen, müssen serialisierbar sein, um keine unerwünschten Effekte aufeinander zu haben.

Die Dreiphasigkeit der dynamischen kooperativen Serialisierbarkeit wird dazu benötigt, die geforderte Serialisierbarkeit von Kooperationsgruppen durchzusetzen. Dieser Mechanismus eignet sich sehr gut dazu, dieses Konsistenzkriterium durchzusetzen, allerdings gäbe es hierzu auch andere Möglichkeiten. Dreiphasigkeit ist also nicht *zwingend* zur Durchsetzung des Kooperationsprinzips notwendig.

Das Kooperationsprinzip fordert, daß kooperierende dynamische Aktionen nicht serialisierbar zu sein brauchen, während nicht-kooperierende dynamische Aktionen dieses sein müssen. Um diese Anforderungen systemseitig überwachen zu können, müssen die Kooperationspartner dem System mitgeteilt werden, welches bei den dynamischen Aktionen für kooperative Anwendung automatisch beim Zutritt zu einer Kooperationsgruppe geschieht. Da die Kooperationspartner einer dynamischen Aktion aber nicht immer schon bei ihrem Start feststehen, bietet die dynamische kooperative Serialisierbarkeit den dynamischen Aktionen die Möglichkeit, in der ersten Phase ohne Serialisierbarkeitsanforderungen auf Objekte zuzugreifen und so ihre Kooperationspartner zu finden. Erst bei der Bildung der Kooperationsgruppen, also beim Wechsel in die zweite Phase, wird nachträglich überprüft, ob die Objektzugriffe der ersten Phase bezüglich der jetzt gebildeten Kooperationsgruppen erlaubt waren.

3.2.3 Für das Auftragsprinzip relevante Eigenschaften von geschachtelten dynamischen Aktionen

Analog zum vorherigen Abschnitt wird jetzt das Konzept der geschachtelten dynamischen Aktionen betrachtet.

Auftragsprinzip

> Das Auftragsprinzip besagt, daß eine dynamische Aktion nicht alle ihre Aufgaben selbst erledigen muß, sondern daß sie Teilaufgaben an andere dynamische Aktionen delegieren kann. Außenstehenden dynamischen Aktionen muß der Auftragnehmer verborgen bleiben, damit es für sie so aussieht, als ob alle Tätigkeiten von einer einzigen dynamischen Aktion ausgeführt worden wären.

Das Auftragsprinzip wird durch folgende Eigenschaften der geschachtelten dynamischen Aktionen realisiert:

- Die Auftraggeber-Auftragnehmer-Beziehungen müssen in der Struktur der dynamischen Aktionen widergespiegelt werden. Dies entspricht der Strukturierung der dynamischen Aktionen in Aktionsbäume.

- Ergebnisse von Auftragnehmern bleiben für alle dynamischen Aktionen außer dem Auftraggeber verborgen, bis dieser sie freigibt.

Die Anforderungen der Serialisierbarkeit von Aktionsbäumen aus dem Konzept der geschachtelten dynamischen Aktionen wird nicht zur Durchsetzung des Auftragsprinzips benötigt. Das Verbergen der Auftragnehmer vor äußeren dynamischen Aktionen wird nur durch die Sichtbarkeitsregeln realisiert. Serialisierbarkeit bewirkt, daß Aktionsbäume konsistent frei von Nebenläufigkeitsanomalien ablaufen können.

Das Auftragsprinzip beinhaltet auch eine Forderung an die Fehlerbehandlung: Ein Abbruch des Auftragnehmers darf nicht zum Abbruch des Auftraggebers führen. Diese Forderung ist bei den geschachtelten dynamischen Aktionen schon umfassend gelöst worden (siehe Abschnitt 2.6.2) und betrifft wegen der Orthogonalität von Fehlerbehandlung und Nebenläufigkeitskontrolle nicht die Entwicklung eines integrierten Nebenläufigkeitskonzepts.

3.3 Integrationskonzept

Im vorherigen Abschnitt wurde aufgezeigt, welche Eigenschaften ein auf dynamischen Aktionen basierendes Konzept zur Integration von Auftrags- und Kooperationsprinzip haben muß. Davon ausgehend wird jetzt das Konzept der *geschachtelten dynamischen Aktionen für kooperative Anwendungen* vorgestellt.

Struktur

Die Auftraggeber-Auftragnehmer-Beziehungen zwischen dynamischen Aktionen werden wie bei den geschachtelten dynamischen Aktionen durch eine Strukturierung in *Aktionsbäume* realisiert. Die Zusammenfassung von Kooperationspartnern zu *Kooperationsgruppen* wird von den dynamischen Aktionen für kooperative Anwendungen übernommen. Dabei dürfen sich Aktionsbäume nicht über mehrere Kooperationsgruppen erstrecken: Sofern eine (Sub-) Aktion Teil einer Kooperationsgruppe ist, müssen alle anderen (Sub-) Aktionen dieses Aktionsbaums auch dieser Kooperationsgruppe beitreten. In einer Kooperationsgruppe können mehrere Aktionsbäume enthalten sein. Eine genauere Diskussion des Verhältnisses von Aktionsbäumen von Kooperationsgruppen folgt in Unterabschnitt 3.3.1.

Serialisierbarkeit

Die Diskussion im vorigen Abschnitt hat gezeigt, daß zur Durchsetzung des Auftragsprinzips keine Anforderungen bezüglich der Serialisierbarkeit gestellt werden müssen. Das Kooperationsprinzip hingegen fordert Serialisierbarkeit von dynamischen Aktionen aus verschiedenen Kooperationsgruppen und keine Serialisierbarkeit innerhalb einer Kooperationsgruppe. Zur Durchsetzung dieser Anforderungen wird das Kriterium der *dynamischen kooperativen Serialisierbarkeit* aus dem Konzept der dynamischen Aktionen für kooperative Anwendungen übernommen. Wie dort wird sie mit Hilfe des dreiphasigen Verhaltens der dynamischen Aktionen durchgesetzt.

Im Unterschied zu den geschachtelten dynamischen Aktionen wird hier nicht zwingend die Serialisierbarkeit von ganzen Aktionsbäumen oder Teilbäumen gefordert, sofern sie innerhalb einer Kooperationsgruppe liegen. Nebenläufigkeitsanomalien, zu deren Vermeidung die Serialisierbarkeit gefordert wurde, können aber individuell für jedes Objekt mit Hilfe der Mechanismen zur Sichtbarkeit von Objektänderungen vermieden werden.

Sichtbarkeit von Objektänderungen

In Bezug auf die Sichtbarkeit von Objektänderungen stellen Auftrags- und Kooperationsprinzip unterschiedliche Anforderungen: Zur Durchsetzung des Auftragsprinzips dürfen Objektänderungen von Auftragnehmern nur für den jeweiligen Auftraggeber sichtbar werden, wohingegen das Kooperationsprinzip nur dann durchgesetzt werden kann, wenn Objektänderungen zumindest für die Kooperationspartner sichtbar sind. Das Konzept der dynamischen Aktionen für kooperative Anwendungen als Realisierung des Kooperationsprinzip stellt gar keine Mechanismen zur Einschränkung der Sichtbarkeit von Objektänderungen zur Verfügung, d. h. Objektänderungen sind nach Freigabe durch die ändernde dynamische Aktion für alle anderen dynamischen Aktionen sichtbar.

Diese beiden Anforderungen werden integriert, indem die Sichtbarkeit von Objektänderungen individuell durch die Applikationen gesteuert werden kann: Mit der weiter unten in Unterabschnitt 3.3.2 genauer beschriebenen *Objektkontrolle* kann ein Auftraggeber gezielt für jedes Objekt entscheiden, ob die Änderungen des Auftragnehmers an diesem Objekt nur für ihn oder auch für andere dynamische Aktionen sichtbar sind.

Die Objektkontrolle läßt sich nicht nur auf Auftraggeber und Auftragnehmer anwenden, sondern auch auf Kooperationsgruppen: Die Mitglieder einer Kooperationsgruppe können für einzelne Objekte festlegen, wie lange Änderungen nur innerhalb der Kooperationsgruppe sichtbar sind. Diese *Erweiterung der Objektkontrolle auf Kooperationsgruppen* wird in Unterabschnitt 3.3.3 beschrieben.

3.3.1 Schachtelung und Kooperationsgruppen

Dynamische Aktionen werden im Integrationskonzept nach zwei Aspekten strukturiert: Zum einen in Aktionsbäumen und zum anderen in Kooperationsgruppen. Dabei muß jede dynamische Aktion Teil eines Aktionsbaums sein, wobei auch einelementige „Aktionsbäume“ zugelassen sind, die flachen dynamischen Aktionen entsprächen. Außerdem muß auch jede dynamische Aktion einer – gegebenenfalls einelementigen – Kooperationsgruppe beitreten. Diese beiden Strukturierungen sind jeweils Grundlage für die Durchsetzung von Auftrags- und Kooperationsprinzip. Da die dynamische kooperative Serialisierbarkeit beidseitigen Informationsfluß zwischen dynamischen Aktionen auf solche innerhalb derselben Kooperationsgruppe beschränkt und das Auftragsprinzip die Möglichkeit zu beidseitigem Informationsfluß zwischen Auftraggeber und Auftragnehmer fordert, müssen im Integrationskonzept Auftraggeber und -nehmer derselben Kooperationsgruppe beitreten, welches bei rekursiver Anwendung dieses Arguments dazu führt, daß Aktionsbäume immer komplett zu derselben Kooperationsgruppe gehören müssen.

Kooperation soll zwischen (Sub-) Aktionen auch aus unterschiedlichen Aktionsbäumen möglich sein. Da Kooperationspartner sich in derselben Kooperationsgruppe befinden müssen und mit den kooperierenden dynamischen Aktionen die jeweiligen kompletten Aktionsbäume auch zu derselben Kooperationsgruppe gehören müssen, enthält eine Kooperationsgruppe also einen oder mehrere komplette Aktionsbäume, wie exemplarisch in Abbildung 3.1 dargestellt ist.

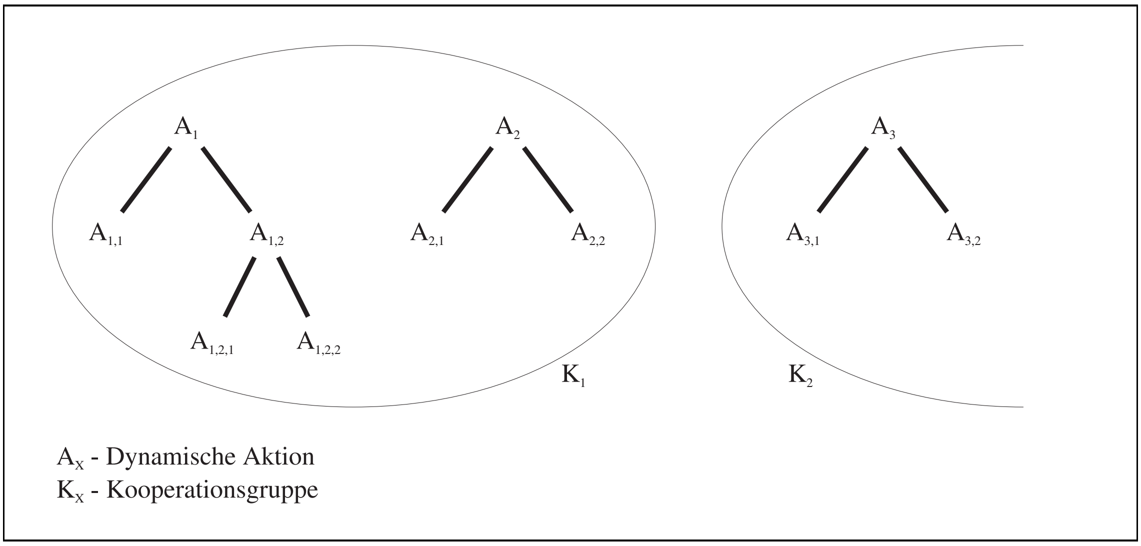


Abbildung 3.1: Aktionsbäume sind komplett in Kooperationsgruppen enthalten

In diesem Beispiel können beliebige (Sub-) Aktionen aus den beiden Aktionsbäume innerhalb von K_1 miteinander kooperieren, während eine Kooperation mit einer Aktion aus K_2 nicht möglich ist. Die dynamischen Aktionen A_1 und A_2 haben mehrere Sohnaktionen zur Erledigung von Unteraufträgen erzeugt. Um diese Aufträge durchzuführen, kann es zum Beispiel nötig sein, daß $A_{1,2}$ und $A_{2,1}$ miteinander kooperieren, wodurch die gesamten Aktionsbäume von A_1 und A_2 derselben Kooperationsgruppe beitreten müssen.

3.3.2 Objektkontrolle

Auftrags- und Kooperationsprinzip stellen unterschiedliche Anforderungen an die Sichtbarkeit von Objektänderungen, die nicht zu einem einheitlichen starren Konzept kombiniert werden können: Im Auftragsprinzip dürfen Objektänderungen von Auftragnehmern zuerst nur für den Auftraggeber sichtbar sein, wohingegen im Kooperationsprinzip Änderungen zumindest für die Kooperationspartner sichtbar sein sollten. Ob eine Objektänderung eines Auftragnehmers als Ergebnis nur für den Auftraggeber oder zur Kooperation für andere dynamische Aktionen sichtbar sein soll, hängt von der Applikationssemantik ab. Im Integrationskonzept gibt es daher den Mechanismus der Objektkontrolle, mit dem seitens der Applikation die Sichtbarkeit von Objektänderungen individuell geregelt werden kann.

Objektkontrolle

Im allgemeinen werden Objektänderungen nach Freigabe durch die ändernde dynamische Aktion für alle anderen dynamischen Aktionen sichtbar. Eine dynamische Aktion kann für einen bestimmten Zeitraum die Objektkontrolle über ein Objekt erwerben und dadurch die Sichtbarkeit aller Änderungen, die während dieser Zeit an diesem Objekt durchgeführt werden, auf sich selbst und ihre Nachfahren einschränken. Außerdem dürfen während dieser Zeit auch nur diese dynamische Aktion selbst und ihre Nachfahren Änderungen an diesem Objekt durchführen.

Eine Objektkontrolle kann nur dann erworben werden, wenn höchstens Vorfahren der anfordernden dynamischen Aktion die Objektkontrolle über das betreffende Objekt besitzen.

Eine dynamische Aktion kann also zu einem beliebigen Zeitpunkt die Objektkontrolle über ein Objekt anfordern. Wenn die Aktionsverwaltung ihr diese gewährt, kann sie die Objektkontrolle solange behalten wie nötig und anschließend wieder abgeben. Während eine dynamische Aktion die Objektkontrolle über ein Objekt hält, kann sie sicher sein, daß das Objekt nur von ihr selbst oder ihren Nachfahren geändert oder gelesen werden kann. Das Objekt existiert während dieser Zeit praktisch nur für dynamische Aktionen innerhalb des Teilbaums, dessen Wurzel die dynamische Aktion ist, die die Objektkontrolle hält.

Es können mehrere dynamische Aktionen gleichzeitig die Objektkontrolle über dasselbe Objekte besitzen, sofern sie alle untereinander in einem Vorfahre-Nachfahre-Verhältnis stehen. Objekte sind dann nur für die dynamischen Aktionen sichtbar, die in dem Teilbaum liegen, dessen Wurzel die tiefste dynamische Aktion im Baum ist, die eine Objektkontrolle hält. Wenn in Abbildung 3.1 zum Beispiel die dynamischen Aktionen A_1 und $A_{1,2}$ die Objektkontrolle auf einem Objekt o_1 besitzen, wäre es für die dynamischen Aktionen $A_{1,2}$, $A_{1,2,1}$ und $A_{1,2,2}$ sichtbar.

Der Mechanismus der Objektkontrolle hat Ähnlichkeit mit dem nicht-strikten Zwei-Phasen-Sperrverfahren der geschachtelten dynamischen Aktionen (siehe Abschnitt 2.6.2), wobei Anfordern und Freigeben von Sperren dem Anfordern und Abgeben der Objektkontrolle entsprechen. Zwischen beiden gibt es aber einen entscheidenden Unterschied: Die Sperren müssen *zweiphasig* angefordert und freigegeben werden. Dieses bedeutet zum einen, daß Sperren einer dynamischen Aktion auf die bearbeiteten Objekte nicht unabhängig voneinander angefordert werden können, da sobald eine Sperre auf einem Objekt freigegeben wurde, keine weitere Sperre mehr angefordert werden kann. Zum anderen ist es deshalb auch nicht möglich, daß eine dynamische Aktion auf einem Objekt mehrfach hintereinander eine Sperre anfordert und wieder freigibt. Objektkontrollen hingegen können von einer dynamischen Aktion für jedes Objekt mehrfach angefordert und abgegeben werden. Außerdem gibt es keine Abhängigkeiten zwischen den Objektkontrollen, die eine dynamische Aktion auf unterschiedliche Objekte besitzt. Die Applikation ist dadurch in der Lage, Objektkontrolle für jedes Objekt individuell so anzufordern und abzugeben, wie es ihre Semantik erfordert.

Zur Realisierung des Auftragsprinzips lassen sich mit Hilfe der Objektkontrolle die Ergebnisse und damit die Existenz eines Auftragnehmers nach außen verbergen, indem der Auftraggeber vor dem Start des Auftragnehmers die Objektkontrolle über alle Objekte erwirbt, die der Auftragnehmer ändert.

Die Eigenschaft der dynamischen kooperativen Serialisierbarkeit fordert innerhalb von Kooperationsgruppen keine Serialisierbarkeit von dynamischen Aktionen, wodurch es zwischen diesen dynamischen Aktionen zu Nebenläufigkeitsanomalien kommen könnte. Diese können aber mit Hilfe der Objektkontrollen verhindert werden: Solange eine dynamische Aktion eine Objektkontrolle für ein Objekt besitzt, ist ein Zugriff für alle anderen dynamischen Aktionen außer den Nachfahren verboten, wodurch bei diesem Objekt für die Zeit der gehaltenen Objektkontrolle keine Nebenläufigkeitsanomalien auftreten kann. Während die dynamische kooperative Serialisierbarkeit zwischen nicht-kooperierenden dynamischen Aktionen unabhängig von den jeweils zugegriffenen Objekten Nebenläufigkeitsanomalien ausschließt, können zwischen Kooperationspartnern durch die Objektkontrolle solche Anomalien mit feiner Granularität, nämlich für jedes einzelne Objekt, verhindert werden. Dadurch können die Situationen, in denen die Applikation mögliche Nebenläufigkeitsanomalien berücksichtigen muß, auf solche reduziert werden, an denen tatsächlich Informationsfluß stattfinden soll und aus diesem Grund die Objektkontrolle abgegeben werden muß.

Die Tatsache, daß eine Objektkontrolle nur Objektzugriffe vom Besitzer der Objektkontrolle und dessen Nachfahren erlaubt, läßt sich auch bei der Kooperation nutzen: Angenommen zwei Brüder kooperieren miteinander, dann findet gezielt zwischen diesen beiden ein Informationsfluß über ein oder mehrere Objekte statt. Die Inhalte dieser Objekte sind aber auch für andere dynamische Aktionen les- und änderbar, was dazu führen kann, daß durch unerwartete Objektänderungen unbeteiligter dynamischer Aktionen der Informationsfluß zwischen den Kooperationspartnern unterbrochen wird. Indem die Vater-Aktion der kooperierenden Brüder für die betroffenen Objekte eine Objektkontrolle erwirbt, wird dieser Informationsfluß vor Unterbrechungen geschützt. Lediglich weitere nebenläufige Brüder einschließlich ihrer Nachfahren könnten den Informationsfluß der kooperierenden dynamischen Aktionen noch stören. Da die dynamischen Aktionen, die den Informationsfluß überhaupt noch stören könnten, alle Nachfahren derselben dynamischen Aktion sind, hat die Applikation oft weitere Informationen über deren Verhalten und kann so das Risiko gut einschätzen, ob der Informationsfluß durch nicht-kooperierende dynamische Aktionen gestört und damit unterbrochen werden könnte. Allgemein läßt sich ein Informationsfluß zwischen zwei kooperierenden dynamischen Aktionen schützen, indem ein gemeinsamer Vorfahre, falls er existiert, die Objektkontrolle über die gemeinsam genutzten Objekte erwirbt.

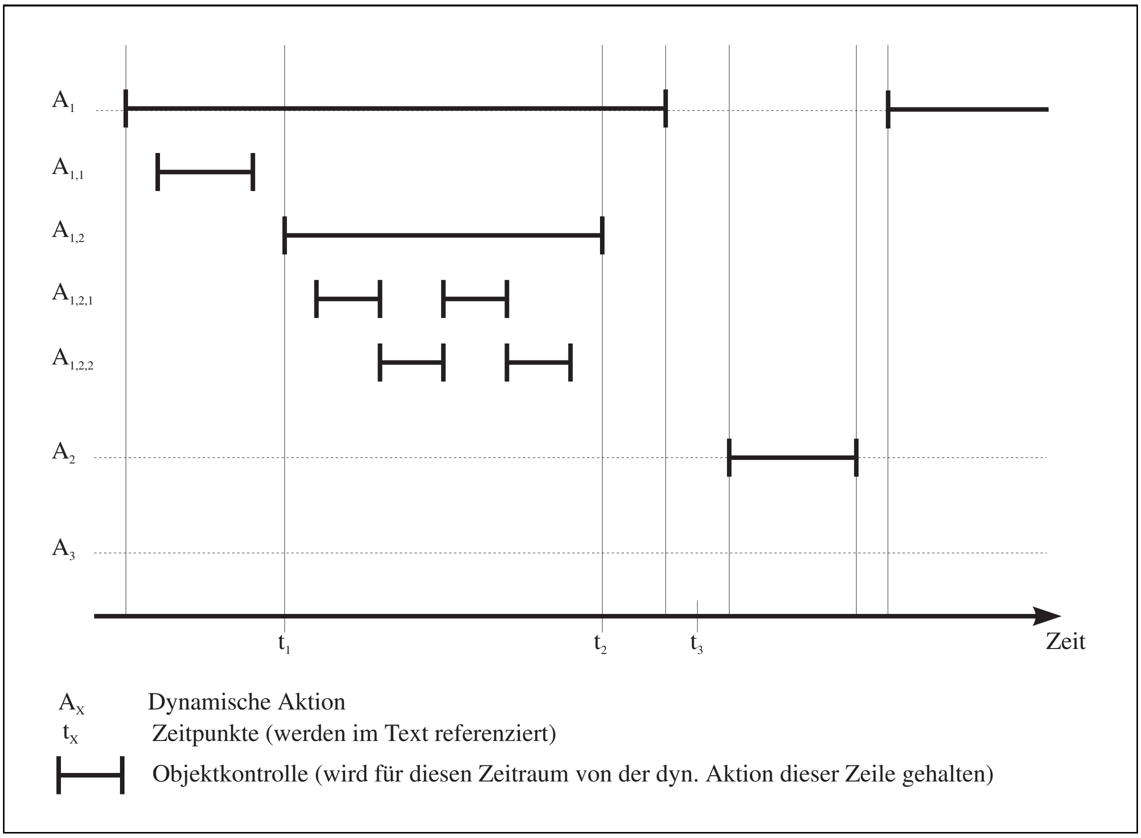


Abbildung 3.2: Beispiel für die Realisierung des Auftragsprinzips und Unterstützung von Kooperation durch Objektkontrolle

Abbildung 3.2 zeigt einen möglichen Ablauf der dynamischen Aktionen aus Abbildung 3.1. In diesem Beispiel ist zu sehen, wann die einzelnen dynamischen Aktionen die Objektkontrolle über ein Objekt o_1 besitzen, und es läßt sich daran veranschaulichen, wie durch die Objektkontrolle das Auftragsprinzip realisiert und Kooperation zwischen dynamischen Aktionen unterstützt werden kann.

Zu Beginn vergibt A_1 einen Auftrag an $A_{1,1}$. Um die Ergebnisse des Auftragnehmers vor fremden dynamischen Aktionen wie zum Beispiel A_2 und A_3 zu schützen, fordert A_1 vor dem Auftrag eine

Objektsperre über das Objekt o_1 an, das das Ergebnis von $A_{1,1}$ enthalten wird. $A_{1,1}$ erwirbt für die Zeit ihrer Berechnung auch eine Objektsperre auf o_1, um eigene Zwischenergebnisse vor anderen dynamischen Aktionen zu verbergen und erst ihr Endergebnis nach außen sichtbar zu machen. Zwischen den Zeitpunkten t_1 und t_2 wollen die dynamischen Aktionen $A_{1,2,1}$ und $A_{1,2,2}$ miteinander kooperieren, indem sie mehrfach über o_1 Informationen austauschen. Um diese Informationen, die nur für diese beiden dynamischen Aktionen bestimmt sind, vor dem Zugriff anderer dynamischer Aktionen zu schützen, erwirbt ihr gemeinsamer Vater $A_{1,2}$ für die Zeit der Kooperation eine Objektsperre auf o_1. Nachdem alle Aufträge von A_1 erledigt worden sind, möchte sie ihr Ergebnis der anderen Toplevel-Aktion derselben Kooperationsgruppe A_2 zur Verfügung stellen. Dafür muß sie die Objektsperre abgeben, damit A_2 anschließend die Möglichkeit hat, selbst eine Objektsperre zu erwerben und dieses Objekt zu bearbeiten.

3.3.3 Erweiterung der Objektkontrolle auf Kooperationsgruppen

Oben wurde beschrieben, wie die Kooperation zweier dynamischer Aktionen vor Unterbrechungen von außen geschützt werden kann, falls sie einen gemeinsamen Vorfahren haben. Es ist aber nicht möglich, die Kooperation zwischen dynamischen Aktionen aus unterschiedlichen Aktionsbäumen innerhalb einer Kooperationsgruppe vor Unterbrechungen durch dynamische Aktionen, die nicht zu dieser Kooperationsgruppe gehören, zu schützen. Daher kann folgender unerwünschte Fall auftreten:

Wenn zwei dynamische Aktionen aus unterschiedlichen Kooperationsgruppen über ein Objekt miteinander kooperieren, gibt es einen Moment, indem die eine dynamische Aktion die Objektkontrolle bereits abgegeben, die andere die Objektkontrolle aber noch nicht angefordert hat. Genau dann kann bei validierten Kooperationspartnern eine noch nicht validierte dynamische Aktion aus einer anderen Kooperationsgruppe auf dieses Objekt zugreifen. (In Abbildung 3.2 ist der Zeitpunkt t_3 zum Beispiel ein solcher Moment, in dem A_3 auf das Objekt zugreifen könnte.) Die dynamische kooperative Serialisierbarkeit verbietet in diesem Fall aber, daß einer der beiden Kooperationspartner erneut auf das Objekt zugreift, da dann die Serialisierbarkeit von dynamischen Aktionen aus unterschiedlichen Kooperationsgruppen nicht mehr gewährleistet werden könnte. Die fremde dynamische Aktion hat also durch ihren Zugriff eine weitere Kooperation über dieses Objekt unmöglich gemacht.

Dieser Fall soll mit Hilfe der Objektkontrolle vermieden werden. Zu diesem Zweck betrachtet man die Kooperationsgruppe selbst als „virtueller“ Vater aller Toplevel-Aktionen dieser Kooperationsgruppe und erlaubt, daß für eine Kooperationsgruppe auch Objektkontrollen erworben werden können. Kooperation über Objekte, für die die Kooperationsgruppe die Objektkontrolle besitzt, sind dann vor Unterbrechungen durch dynamische Aktionen außerhalb der Kooperationsgruppe geschützt, aber für dynamische Aktionen innerhalb der Kooperationsgruppe weiter zugreifbar. Die dynamischen Aktionen aus Abbildung 3.1 bilden mit virtuellen *Kooperationsgruppen-Aktionen* dann die Baumstruktur aus Abbildung 3.3.

Da eine Kooperationsgruppe nur eine Strukturierung von dynamischen Aktionen ist, kann die Kooperationsgruppen-Aktion nicht selbst Objektkontrollen anfordern. Die Objektkontrollen werden stattdessen stellvertretend von den dynamischen Aktionen der Kooperationsgruppe angefordert.

3.4 Das kooperative Sperrverfahren

Das Konzept der geschachtelten dynamischen Aktionen für kooperative Anwendungen wird durch das *kooperative Sperrverfahren* realisiert. Genau wie dieses Konzept mit dem Ziel entwickelt wurde, die beiden Konzepte der geschachtelten dynamischen Aktionen und der dynamischen Aktionen für kooperative Anwendungen soweit wie möglich zu integrieren, sollen auch im kooperativen Sperrver-

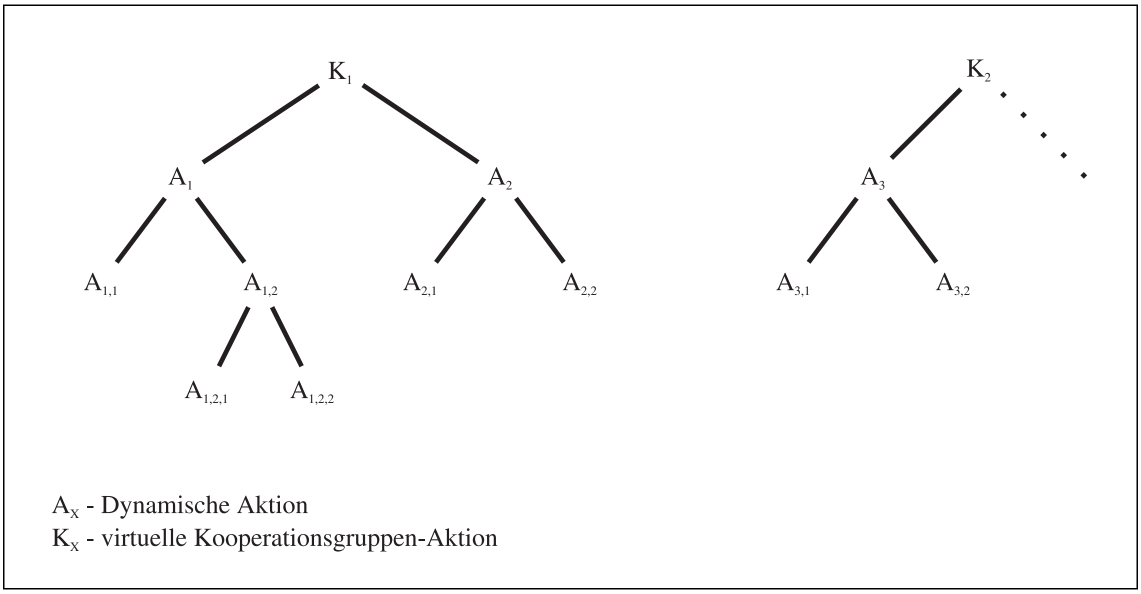


Abbildung 3.3: Aktionsbäume mit Kooperationsgruppen als virtuelle Vater-Aktionen

fahren die zu diesen beiden Konzepten gehörenden Sperrverfahren – das flache kooperative und das geschachtelte Sperrverfahren – soweit wie möglich integriert werden. Dazu werden zuerst die beiden zugrundeliegenden Sperrverfahren verglichen und entstehende Probleme einer direkten Kombination herausgearbeitet. Anhand dieser Analyse werden die benötigten Eigenschaften des kooperativen Sperrverfahrens diskutiert und anschließend wird das kooperative Sperrverfahren anhand der Regeln zur Vergabe und Freigabe von Sperren vorgestellt.

3.4.1 Probleme bei der Kombination des flachen kooperativen und des geschachtelten Sperrverfahrens

Das flache kooperative Sperrverfahren (siehe Abschnitt 2.6.3) und das geschachtelte Sperrverfahren (Abschnitt 2.6.2) basieren beide auf der bekannten Schreib-Lese-Semantik. Das bedeutet, es gibt Schreib- und Lesesperren, wobei Schreibsperren mit anderen Sperren in Konflikt stehen und deshalb außer in den aufgeführten Ausnahmen nicht gleichzeitig mit anderen Sperren für dasselbe Objekt vergeben werden dürfen. Es gilt dabei die schon in Abbildung 2.3 gezeigte Kompatibilitätsmatrix. Schreibende oder lesende Objektzugriffe dürfen grundsätzlich nur durchgeführt werden, wenn die dynamische Aktion die entsprechende Sperre besitzt.

Obwohl beide Sperrverfahren grundsätzlich auf demselben Typ Sperren basieren, gibt es doch erhebliche Unterschiede zwischen ihnen:

Unterschiedliche Ebenen der Sperren

Im flachen kooperativen Sperrverfahren gibt es *eine* Sperrenebene: Eine dynamische Aktion kann auf ein Objekt zugreifen, sofern sie die entsprechende Sperre hält.

Das geschachtelte Sperrverfahren hingegen enthält *zwei* Ebenen von Sperren. Nachdem eine dynamische Aktion eine Sperre für ein Objekt erworben hat, muß sie für dasselbe Objekt noch eine Benutzungssperre erwerben und darf dann auf das Objekt zugreifen. Die zweite Ebene der Benutzungssperren wurde eingeführt, da im geschachtelten Sperrverfahren Vor- und Nachfahren gleichzeitig zueinander in Konflikt stehende Sperren auf dasselbe Objekt bekommen dürfen, trotzdem aber sichergestellt werden soll, daß jeweils nur der letzte Nachfahre auf ein Objekt zugreifen kann. Dadurch ist für den Auftragnehmer gesichert, daß der Auftraggeber trotz gehaltener Sperre keine temporären Objektzustände sehen kann, sondern erst das Ergebnis bei Freigabe der Sperre durch den Auftragnehmer. Außerdem dienen die Benutzungssperren, die nicht gleichzeitig vergeben werden dürfen, wenn sie in Konflikt zueinander stehen, der Synchronisation von mehreren nebenläufigen Prozessen innerhalb einer dynamischen Aktion.

Vergabe und Freigabe der Sperren

Die Sperren im geschachtelten Sperrverfahren werden nach dem nicht-strikten Zweiphasensperrprotokoll vergeben. Das bedeutet, daß eine dynamische Aktion eine Sperre in der ersten Phase einmal anfordern und anschließend in der zweiten Phase wieder freigeben darf. Die Zweiphasigkeit des Sperrverfahrens garantiert die Serialisierbarkeit von Toplevel-Aktionen und Brüdern, ohne daß eine dynamische Überprüfung durchgeführt werden muß. Außerdem verbirgt ein Auftraggeber die Ergebnisse seiner Auftragnehmer nach außen, indem er selbst Sperren für die Objekte hält, die die Auftragnehmer bearbeiten. Sperren von Sub-Aktionen werden bei ihrer Freigabe nicht sofort allgemein freigegeben, sondern an den nächsten Vorfahren vererbt, der sich noch in der ersten Phase befindet, um die Zweiphasigkeit von Aktionsbäumen und das Verbergen aller Objektänderungen des Auftragnehmers zu garantieren. Falls ein Auftragnehmer Objekte ändert, die der Auftraggeber nicht vorher gesperrt hat, würden ohne das Vererben der Sperren diese Änderungen ohne Kontrolle durch den Auftraggeber direkt allgemein sichtbar sein.

Das flache kooperative Sperrverfahren basiert auf dem Kriterium der dynamischen kooperativen Serialisierbarkeit. Hier wird die Serialisierbarkeit von dynamischen Aktionen aus unterschiedlichen Kooperationsgruppen zur Laufzeit dynamisch überprüft, indem Konfliktabhängigkeiten zwischen dynamischen Aktionen von der Aktionsverwaltung erkannt, gespeichert und überprüft werden. In den drei Phasen einer dynamischen Aktion wird die Menge der zugreifbaren Objekte so eingeschränkt, daß die bei Bildung der Kooperationsgruppe überprüfte und zugesicherte Serialisierbarkeit nicht mehr durch neu entstehende Abhängigkeiten verletzt werden kann. Sperren dienen in diesem Verfahren nicht der Durchsetzung von Serialisierbarkeit der dynamischen Aktionen, sondern verhindern die Sichtbarkeit von temporären Objektzuständen und vermeiden dadurch Nebenläufigkeitsanomalien. Sofern es aufgrund der Phase, in der sich die dynamische Aktion befindet, erlaubt ist, können dynamische Aktionen in diesem Sperrverfahren mehrfach Sperren auf dasselbe Objekt anfordern und freigeben.

3.4.2 Eigenschaften des kooperativen Sperrverfahrens

Das kooperative Sperrverfahren basiert wie die zugrundeliegenden Sperrverfahren auch auf der Schreib-Lese-Semantik mit ensprechender Kompatibilitätsmatrix.

Das durch das kooperative Sperrverfahren umgesetzte Konzept der geschachtelten dynamischen Aktionen für kooperative Anwendungen hat als Konsistenzkriterium die dynamische kooperative Serialisierbarkeit. Dieses Konsistenzkriterium wird bereits im flachen kooperativen Sperrverfahren durchgesetzt, indem dynamische Aktionen drei Phasen durchlaufen, in denen die Menge der zugreifbaren Objekte, wie in Abbildung 2.4 dargestellt, entsprechend der Phase eingeschränkt wird, und indem bei den Phasenübergängen die Serialisierbarkeit dynamisch überprüft wird. Dieses Verfahren wird in das kooperative Sperrverfahren übernommen.

Die Betrachtung im vorherigen Abschnitt hat gezeigt, daß das Auftragsprinzip im geschachtelten Sperrverfahren dadurch realisiert wird, daß auch in Konflikt stehende Sperren gleichzeitig an

Auftraggeber und -nehmer vergeben werden können. Ebenso wird durch die zusätzlichen Benutzungssperren eine Unterbrechung des Auftragnehmers durch den Auftraggeber verhindert. Die zwei Ebenen der Sperren werden auch in das kooperative Sperrverfahren übernommen. Da die Sperren aber nicht mehr zweiphasig vergeben werden, um so Serialisierbarkeit durchzusetzen, sondern zur Realisierung von Kooperation mehrfach angefordert und freigegeben werden können, werden sie *kooperative Sperren* genannt. Das kooperative Sperrverfahren hat also zwei verschiedene Arten von Sperren: kooperative Sperren und Benutzungssperren, wobei eine dynamische Aktion eine Benutzungssperre nur dann anfordern darf, wenn sie schon eine kooperative Sperre besitzt. Objektzugriffe sind nur erlaubt, wenn die dynamische Aktion sowohl eine kooperative als auch eine Benutzungssperre besitzt.

Im Gegensatz zum geschachtelten Sperrverfahren werden freigegebene Sperren von Subaktionen nicht vererbt, sondern allgemein freigegeben, sofern nicht ein Vorfahre diese Sperre ohnehin schon besitzt. Das Vererben hat im geschachtelten Sperrverfahren die Aufgabe, Zweiphasigkeit und das Verbergen von Auftragnehmern in jedem Fall zu garantieren. Beide Eigenschaften sind im kooperativen Sperrverfahren nicht mehr zwingend erforderlich.

Das Konzept der geschachtelten dynamischen Aktionen für kooperative Anwendungen enthält zur Steuerung der Sichtbarkeit von Objektänderungen und zur Vermeidung von Unterbrechungen den Mechanismus der Objektkontrolle. Die Objektkontrolle wird im kooperativen Sperrverfahren durch die kooperativen Sperren realisiert, wobei Anfordern und Freigeben von kooperativen Sperren dem Anfordern und Freigeben von Objektkontrollen entspricht. Wenn eine dynamische Aktion die Objektkontrolle über ein Objekt besitzt, bedeutet dies, daß nur sie selbst und ihre Nachfahren auf das Objekt zugreifen dürfen. Bei der Übertragung der Objektkontrolle in das kooperative Sperrverfahren muß beachtet werden, daß eine dynamische Aktion nur dann auf ein Objekt zugreifen darf, wenn sie die passende kooperative Sperre (und Benutzungssperre) für dieses Objekt hält. Die Beschränkung der Objektzugriffe bei der Objektkontrolle läßt sich also durch eine Beschränkung der gewährten kooperativen Sperren realisieren: Wenn eine dynamische Aktion eine kooperative Sperre auf ein Objekt hält, dürfen kooperative Sperren, die zu dieser in Konflikt stehen, nur an Nachfahren der haltenden dynamischen Aktion vergeben werden.

Kooperationsgruppen als Halter von kooperativen Sperren

Der Mechanismus der Objektkontrolle wurde in Abschnitt 3.3.3 auf Kooperationsgruppen ausgedehnt, um Kooperation innerhalb von Kooperationsgruppen vor Unterbrechungen von außerhalb zu schützen. Zu diesem Zweck wurde die virtuelle Kooperationsgruppen-Aktion als Vater aller Toplevel-Aktionen dieser Kooperationsgruppe eingeführt. Die Kooperationsgruppen-Aktion kann die Objektkontrolle über ein Objekt besitzen. Dieses Vorgehen läßt sich auf die kooperativen Sperren, die die Objektkontrolle realisieren, übertragen, indem die Kooperationsgruppen-Aktion kooperative Sperren halten darf.

Da die Kooperationsgruppen-Aktion nicht real existiert, kann sie die gesperrten Objekte nicht wirklich bearbeiten und darf deshalb auch keine Benutzungssperren anfordern. Die Tatsache, daß die Kooperationsgruppen-Aktion nur virtuell ist, hat auch zur Folge, daß sie nicht selbst ihre kooperativen Sperren anfordern kann. Die kooperativen Sperren der Kooperationsgruppen-Aktionen können daher stellvertretend von den dynamischen Aktionen dieser Kooperationsgruppe angefordert werden. Dabei kann jede dynamische Aktion kooperative Sperren für die Kooperationsgruppe anfordern und muß sie auch selbst wieder freigeben. Die Kooperationsgruppen-Aktion hält eine kooperative Sperre auf ein Objekt, wenn mindestens eine dynamische Aktion stellvertretend für sie eine solche Sperre auf dieses Objekt hält.

Unabhängig davon, welche dynamische Aktion die kooperative Sperre für die Kooperationsgruppe anfordert, werden stets die normalen Regeln zur Vergabe von kooperativen Sperren bezüglich der

Kooperationsgruppen-Aktion als Vater aller Toplevel-Aktionen angewandt.

3.4.3 Regeln zur Sperrenvergabe

Die beschriebenen Eigenschaften des kooperativen Sperrverfahrens werden durch folgende Regeln umgesetzt. Dabei wird vorausgesetzt, daß sich dynamische Aktionen zu den Sperren konform verhalten, also nur auf die Objekte zugreifen, für die sie auch eine kooperative und eine Benutzungssperre im passenden Modus besitzen. Außerdem führt die Aktionsverwaltung bei der Bildung von Kooperationsgruppen und der Validierung, welche jeweils bei den Übergängen zwischen den Phasen einer dynamischen Aktion stattfinden, die nötigen Überprüfungen zur Serialisierbarkeit durch, wie sie in [Mock 1994] beschrieben und in Abschnitt 2.6.3 zusammengefaßt worden sind.

1. **(a)** Eine dynamische Aktion darf nur dann für ein Objekt eine kooperative Schreib-Sperre bekommen, wenn alle anderen Besitzer von kooperativen Lese- oder Schreib-Sperren auf dieses Objekt Vorfahren dieser dynamischen Aktion sind.

   (b) Eine dynamische Aktion darf nur dann für ein Objekt eine kooperative Lese-Sperre bekommen, wenn alle anderen Besitzer von kooperativen Schreib-Sperren auf dieses Objekt Vorfahren dieser dynamischen Aktion sind.

2. Kooperative Sperren werden in Abhängigkeit von der Phase, in der sich die anfordernde dynamische Aktion befindet, vergeben:

   (a) Kooperative Sperren dürfen in der ersten Phase beliebig angefordert werden.

   (b1) In der zweiten Phase dürfen nur kooperative Lese-Sperren für solche Objekte angefordert werden, bei denen der letzte Schreiber in derselben Kooperationsgruppe liegt oder bereits validiert ist.

   (b2) In der zweiten Phase dürfen nur kooperative Schreib-Sperren für solche Objekte angefordert werden, bei denen der letzte Halter einer kooperativen Schreib- oder Lese-Sperre in derselben Kooperationsgruppe liegt oder bereits validiert ist.

   (c1) In der dritten Phase dürfen nur noch kooperative Lese-Sperren für Objekte angefordert werden, bei denen der letzte Schreiber oder der letzte Halter einer kooperativen Lese-Sperre in derselben Kooperationsgruppe liegt.

   (c2) In der dritten Phase dürfen nur noch kooperative Schreib-Sperren für Objekte angefordert werden, bei denen der letzte Halter einer kooperativen Schreib- oder Lese-Sperre in derselben Kooperationsgruppe liegt.

3. Eine dynamische Aktion darf nur dann eine gehaltene kooperative Sperre freigeben, wenn sie keine Benutzungssperre auf dieses Objekt besitzt.

4. **(a)** Eine dynamische Aktion darf nur dann für ein Objekt eine Schreib-Benutzungssperre bekommen, wenn sie für dieses Objekt bereits eine kooperative Schreib-Sperre, aber kein Nachfahre eine kooperative Schreib- oder Lesesperre hält und keine andere dynamische Aktion darauf eine Schreib- oder Lese-Benutzungssperre hält.

   (b) Eine dynamische Aktion darf nur dann für ein Objekt eine Lese-Benutzungssperre bekommen, wenn sie für dieses Objekt bereits eine kooperative Schreib- oder Lese-Sperre, aber kein Nachfahre eine kooperative Schreibsperre hält und keine andere dynamische Aktion darauf eine Schreib-Benutzungssperre hält.

5. Gehaltene Benutzungssperren dürfen jederzeit freigegeben werden.

3.4.4 Anwendungsbeispiel

Das kooperative Sperrverfahren soll jetzt anhand des in der Einleitung dieser Arbeit gegebenen Beispiels der kooperativen Texterstellung anschaulich dargelegt werden.

Dabei wird von folgendem Anwendungsszenario ausgegangen: Zwei Autoren, modelliert durch die dynamischen Aktionen A_1 und A_2, erstellen gemeinsam einen englischen Text. Autor A_1 ist in der Lage Englisch zu schreiben, während Autor A_2 zwar Englisch lesen kann, seinen Text aber in Deutsch schreibt und einen Übersetzer, welcher durch die dynamische Aktion $A_{2,1}$ modelliert wird, damit beauftragt, den Text ins Englische zu übersetzen. Der Text besteht aus Einleitung, Mittelteil und Schluß. Diese Teile werden durch die Objekte E, M und S repräsentiert. Die beiden Autoren haben sich die Texterstellung derart aufgeteilt, daß Autor A_1 Einleitung und Schluß und Autor A_2 den Mittelteil schreibt. Zur Abstimmung muß Autor A_2 die Einleitung lesen können, bevor er den Mittelteil schreibt und Autor A_2 benötigt wiederum den Mittelteil für den Schluß. Letztendlich wollen beide Autoren den gesamten Text lesen, bevor sie gemeinsam entscheiden, ob sie ihn freigeben.

In diesem Beispiel treten sowohl das Auftrags- als auch das Kooperationsprinzip auf: Die Einbeziehung des Übersetzers stellt eine Anwendung des Auftragsprinzips dar, wohingegen die gemeinsame Texterstellung und Entscheidung zur Freigabe durch beide Autoren dem Kooperationsprinzip entsprechen.

Die beiden Autoren bilden zu diesem Zweck eine Kooperationsgruppe K, der auch der Übersetzer als Auftragnehmer von Autor A_2 angehört. Es ergibt sich also der in Abbildung 3.4 dargestellte Aufbau der dynamischen Aktionen.

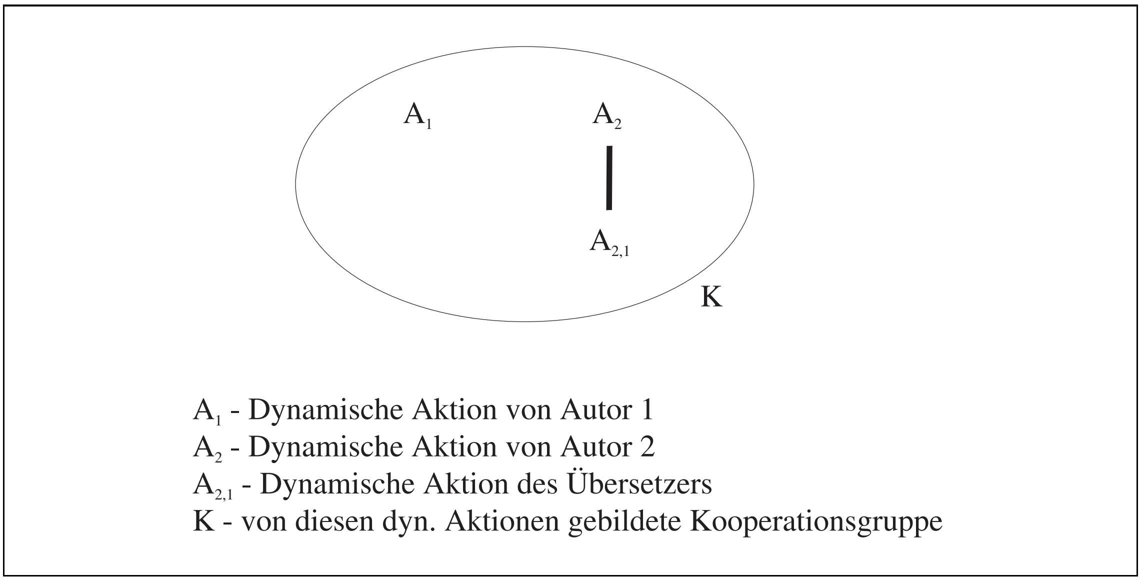


Abbildung 3.4: Modellierung der kooperativen Texterstellung durch dynamische Aktionen

Die Texterstellung läuft dann wie folgt beschrieben ab. In Abbildung 3.5 ist die Vergabe der Sperren an die dynamischen Aktionen zur Veranschaulichung graphisch dargestellt.

- Zu Beginn erwerben beide Autoren für die Kooperationsgruppe K kooperative Schreibsperren auf die drei – noch leeren – Textobjekte. Dadurch wird allen dynamischen Aktionen außerhalb der Kooperationsgruppe der Zugriff auf den Text verboten und es ist ein unterbrechungsfreier Informationsfluß zwischen den Partnern möglich. (In Abbildung 3.5 sind die kooperativen

Sperren der Kooperationsgruppe aus Gründen der Übersichtlichkeit pro Objekt nur einmal dargestellt und nicht für jede stellvertretend erwerbende dynamische Aktion extra.)

- Autor A_1 erwirbt dann eine kooperative Schreibsperre und eine Schreib-Benutzungssperre auf Objekt E und schreibt die Einleitung.

- Nachdem Autor A_1 die Einleitung beendet hat, gibt er beide Schreibsperren auf E ab und erwirbt sofort wieder eine kooperative Lesesperre auf die Einleitung. Dies erlaubt Autor A_2, die Einleitung zu lesen, verhindert aber, daß dieser sie ändert.

  Autor A_2 erwirbt eine kooperative Lesesperre und eine Lese-Benutzungssperre auf die Einleitung, um sie zu lesen. Er erwirbt außerdem eine kooperative Schreibsperre und eine Schreib-Benutzungsperre auf den Mittelteil M und schreibt ihn auf Deutsch. Nachdem Autor A_2 die Einleitung gelesen hat, gibt er beide Lesesperren wieder frei. Über die Einleitung hat somit ein Informationsfluß von Autor A_1 nach Autor A_2 stattgefunden.

- Autor A_2 hat den deutschen Text beendet, gibt die Schreib-Benutzungssperre frei und beauftragt den Übersetzer. Dieser darf für den Mittelteil auch noch eine kooperative Schreibsperre anfordern, da der einzige andere Halter einer Sperre auf M sein Auftraggeber ist. Solange der Übersetzer die kooperative Schreibsperre hält, kann er sicher sein, daß auch sein Auftraggeber den Text nicht ändert, da ihm keine Benutzungssperre gewährt würde. Der Übersetzer kann die Schreib-Benutzungssperre anfordern und Objekt M durch die englische Version des Textes ersetzen.

- Nachdem der Übersetzer seinen Auftrag erfüllt hat, gibt er beide Sperren auf den Mittelteil frei und Autor A_2 kann erneut eine Schreib-Benutzungssperre erwerben, um noch letzte Änderungen in der englischen Fassung durchzuführen. Während der ganzen Zeit hat Autor A_1 keinen Zugriff auf den Mittelteil und somit keine Information von der Existenz eines Übersetzers. So ist das Auftragsprinzip erfüllt.

- Autor A_2 beendet seine Arbeit am Mittelteil, gibt beide Schreib-Sperren frei und erwirbt direkt eine kooperative Lese-Sperre, um Autor A_1 Lesezugriff auf den Mittelteil zu gewähren.

  Dieser erwirbt darauf für die Zeit des Lesens die kooperative Lesesperre und die Lese-Benutzungssperre für den Mittelteil. Zum Schreiben des Schlusses erwirbt er kooperative Schreibsperre und Schreib-Benutzungssperre auf Objekt S. Typisch für das Kooperationsprinzip findet hier nun ein Informationsfluß von A_2 nach A_1 statt, wodurch diese beiden dynamischen Aktionen nicht mehr serialisierbar sind.

- Nachdem Autor A_1 seine Arbeit am Schluß abgeschlossen hat, gibt er die Schreibsperren frei und beide Autoren erwerben kooperative Lesesperren und Lese-Benutzungssperren auf alle Objekte, um den gesamten Text zu lesen. Falls noch Änderungen gemacht werden müßten, würde jetzt ein Autor seine Lesesperren auf dem betroffenen Objekt freigeben, um dem anderen Gelegenheit zur Änderung zu geben.

- Da beide Autoren den Text akzeptieren, geben sie alle ihre Lese-Sperren frei und durch Freigabe der kooperativen Schreib-Sperren für die Kooperationsgruppe machen sie den Text für alle anderen Benutzer lesbar.

In diesem Beispiel findet die Bildung der Kooperationgruppe direkt zu Beginn statt, da davon ausgegangen wird, daß alle Partner inklusive des beauftragten Übersetzers bereits feststehen. Die Validierung kann hier zu einem beliebigen Zeitpunkt stattfinden, nachdem die kooperativen Sperren für die Kooperationsgruppe angefordert worden sind, da hierdurch für die dynamischen Aktionen der letzten Halter einer kooperativen Sperre garantiert innerhalb derselben Kooperationsgruppe liegt, wie es für Objektzugriffe nach der Validierung gefordert ist.

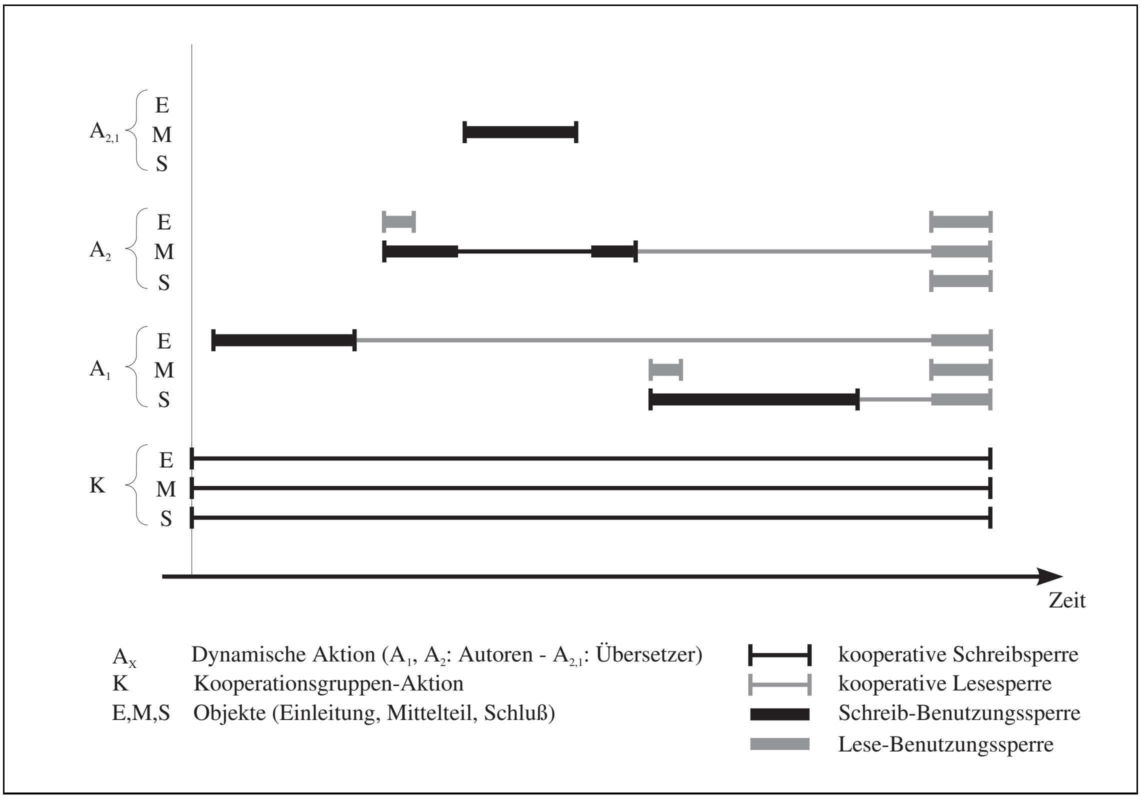


Abbildung 3.5: Sperrenvergabe bei der kooperativen Texterstellung

Dieses Beispiel zeigt neben der Realisierung von Kooperations- und Auftragsprinzip auch deutlich die Einsatzmöglichkeiten der Objektkontrolle, die durch die kooperativen Sperren realisiert ist: Beim Aufruf des Übersetzers wird mittels Objektkontrolle dessen Existenz durch den Auftraggeber vor anderen dynamischen Aktionen verborgen. Die Objektkontrolle auf Kooperationsgruppen-Ebene ermöglicht eine unterbrechungsfreie Kooperation zwischen den Partnern, indem dynamischen Aktionen außerhalb der Kooperationsgruppe der Zugriff auf die gesperrten Objekte verweigert wird.

3.5 Verträglichkeit mit den zugrundeliegenden Konzepten

Das Konzept der geschachtelten dynamischen Aktionen für kooperative Anwendungen wurde in diesem Kapitel als Integrationskonzept aus den beiden zugrundeliegenden Konzepten der geschachtelten dynamischen Aktionen und der dynamischen Aktionen für kooperative Anwendungen entwickelt. Ebenso basiert auch das kooperative Sperrverfahren zur Realisierung des Integrationskonzepts auf dem geschachtelten Sperrverfahren und dem flachen kooperativen Sperrverfahren. Das Integrationskonzept und das geschachtelte Sperrverfahren wurden beide mit dem Ziel entwickelt, neben der geleisteten Integration von Auftrags- und Kooperationsprinzip auch mit den zugrundeliegenden Konzepten und Sperrverfahren verträglich zu sein. Dieser Aspekt wird in diesem Abschnitt untersucht. Dazu werden die Eigenschaften des Integrationskonzepts und der beiden zugrundeliegenden Konzepte verglichen und darauf aufbauend wird die Verträglichkeit des kooperativen Sperrverfahrens gezeigt, indem angegeben wird, wie das Verhalten der beiden zugrundeliegenden Sperrverfahren simuliert werden kann. Im anschließenden Kapitel 4 wird die hier dargelegte Verträglichkeit formal durch Einbettung der geschachtelten dynamischen Aktionen für kooperative Anwendungen in das in [Mock 1994] entwickelte formale Modell für dynamische Aktionen nachgewiesen.

3.5.1 Dynamische Aktionen für kooperative Anwendungen

Die dynamischen Aktionen für kooperative Anwendungen sind dem Integrationskonzept sehr ähnlich, allerdings besteht ein zentraler Unterschied: Die Schachtelungsstruktur der dynamischen Aktionen fehlt, da das Auftragsprinzip nicht realisiert werden soll. Die Strukturierung der Kooperationspartner in Kooperationsgruppen ist in beiden Konzepten vorhanden.

Aufgrund der fehlenden Schachtelungsstruktur ist auch der Mechanismus der Objektkontrolle nicht vorhanden, dessen Hauptaufgabe ist, Ergebnisse von Auftragnehmern zu verbergen. Die Erweiterung der Objektkontrolle auf Kooperationsgruppen hingegen würde auch bei flachen dynamischen Aktionen die Kooperation vor Unterbrechungen von außerhalb der Kooperationsgruppe schützen. Diese Möglichkeit des Unterbrechungsschutzes fehlt im Konzept der dynamischen Aktionen für kooperative Anwendungen.

Beide Konzepte besitzen als Konsistenzkriterium die dynamische kooperative Serialisierbarkeit und setzen daher auch das gleiche dreiphasige Verhalten der dynamischen Aktionen voraus. Aufgrund des gleichen Serialisierbarkeitskriteriums ist in beiden Konzepten für die einzelnen dynamischen Aktionen die Menge der zugreifbaren Objekte in den einzelnen Phasen gleich, und Kooperation ist im gleichen Umfang möglich.

Im Integrationskonzept gibt es also zusätzlich zum Konzept der dynamischen Aktionen für kooperative Anwendungen drei weitere Eigenschaften: eine Schachtelungsstruktur der dynamischen Aktionen, die Mechanismen der Objektkontrolle für „normale“ dynamische Aktionen und die Objektkontrolle für Kooperationsgruppen. Dabei ist die Objektkontrolle für „normale“ dynamische Aktionen nur gemeinsam mit der Schachtelung sinnvoll, während die Objektkontrolle für Kooperationsgruppen auch unabhängig von der Schachtelung eingesetzt werden kann.

Sperrverfahren

Ein Vergleich des kooperativen Sperrverfahrens und des flachen kooperativen Sperrverfahrens zeigt, daß das kooperative Sperrverfahren dem flachen kooperativen Sperrverfahren entspricht und einige weitere Mechanismen für die Durchsetzung der zusätzlichen Eigenschaften des Integrationskonzepts aufweist. Dies sind die Möglichkeit, Sohnaktionen zu erzeugen, die zwei Ebenen der Sperren, Vergaberegeln, die die Schachtelung berücksichtigen, und die Möglichkeit, kooperative Sperren auch für Kooperationsgruppen zu erwerben. Das flache kooperative Sperrverfahren läßt sich im kooperativen Sperrverfahren simulieren, indem genau diese Mechanismen nicht oder nur nach festen Regeln benutzt werden. Im einzelnen müssen sich für eine Simulation alle dynamischen Aktionen zu folgenden Einschränkungen konform verhalten:

- Da das flache kooperative Sperrverfahren keine Schachtelung zuläßt, dürfen keine Sohnaktionen erzeugt werden.

  Da es dadurch keine Vor- oder Nachfahren geben kann, die Sperren halten könnten, vereinfachen sich die Regeln des kooperativen Sperrverfahrens zur Sperrenvergabe derart, daß die Regeln zur Vergabe der kooperativen Sperren genau denen zur Vergabe von Sperren im flachen kooperativen Sperrverfahren entsprechen.

- Im flachen kooperativen Sperrverfahren gibt es nur eine Ebene von Sperren, wohingegen im kooperativen Sperrverfahren eine dynamische Aktion sowohl eine kooperative Sperre als auch eine Benutzungssperre halten muß, um auf ein Objekt zugreifen zu können. Benutzungssperren wurden eingeführt, da in kooperativen Sperrverfahren Vor- und Nachfahren gleichzeitig in Konflikt stehende kooperative Sperren halten dürfen, und zwischen diesen daher eine weitere Synchronisation notwendig ist. Da Schachtelung in der Simulation nicht erlaubt ist, können in Konflikt stehende kooperative Sperren nie gleichzeitig vergeben werden und die Regeln zur Vergabe der Benutzungssperren vereinfachen sich derart, daß jede dynamische Aktion, die eine kooperative Sperre hält, garantiert eine Benutzungsperre erwerben kann.

Statt im flachen kooperativen Sperrverfahren eine Sperre anzufordern, muß in der Simulation die dynamische Aktion zuerst eine kooperative und direkt danach eine Benutzungsperre anfordern. Der Freigabe einer Sperre entspricht die aufeinanderfolgende Freigabe von Benutzungssperre und kooperativer Sperre.

- In der Simulation dürfen keine kooperativen Sperren für Kooperationsgruppen angefordert werden, da ein Äquivalent im flachen kooperativen Sperrverfahren nicht existiert.

3.5.2 Geschachtelte dynamische Aktionen

Die Verträglichkeit der Konzepte der geschachtelten dynamischen Aktionen und der dynamischen Aktionen für kooperative Anwendungen ist schwieriger aufzuzeigen, da diese auf unterschiedlichen Konsistenzbegriffen aufbauen. Das Integrationskonzept basiert auf der dynamischen kooperativen Serialisierbarkeit, während die geschachtelten dynamischen Aktionen auf der Serialisierbarkeit von Aktionsbäumen basieren und Kooperationsgruppen dort gar nicht zur Strukturierung der dynamischen Aktionen benötigt werden. Läßt man nur einen Aktionsbaum pro Kooperationsgruppe zu, garantiert auch das Integrationskonzept die Serialisierbarkeit von kompletten Aktionsbäumen. Allerdings wird im Konzept der geschachtelten dynamischen Aktionen auch die Serialisierbarkeit von Teilbäumen gefordert, deren Wurzeln Brüder sind. Hierzu gibt es im Integrationskonzept keine direkte Entsprechung, da dort keine weiteren Serialisierbarkeitsforderungen innerhalb von Kooperationsgruppen aufgestellt werden können. Wie gleich beschrieben, läßt sich die Serialisierbarkeit von Brüdern aber mit Hilfe der Objektkontrolle realisieren.

Das Konzept der geschachtelten dynamischen Aktionen erfordert, daß *alle* Änderungen, die eine Sohn-Aktion durchführt, zuerst nur für den Vater sichtbar sind. Der Mechanismus der Objektkontrolle im Integrationskonzept kann dieses leisten, ist aber deutlich flexibler, indem er zum Beispiel auch nur einige Änderungen des Sohnes verbergen kann. Mit der Objektkontrolle können auch alle Änderungen, die von dynamischen Aktionen innerhalb eines Teilbaums gemacht werden, nach außen verborgen werden. Wenn die Objektkontrollen zweiphasig vergeben werden, d. h. wenn bis zu einem bestimmten Zeitpunkt nur Objektkontrollen angefordert und danach nur freigegeben werden, ist dadurch die Serialisierbarkeit von Brüdern gewährleistet.

Sperrverfahren

Zwischen dem kooperativen und dem geschachtelten Sperrverfahren gibt es einige Unterschiede, die überwunden werden müssen, um das geschachtelte Sperrverfahren im kooperativen Sperrverfahren zu simulieren. Dabei zeigt sich auch, daß eine komplette Simulation nicht möglich ist, da für das Vererben von Sperren im geschachtelten Sperrverfahren kein Äquivalent im kooperativen Sperrverfahren existiert.

Folgende Punkte müssen bei der Simulation beachtet werden:

- Um die Serialisierbarkeit von Aktionsbäumen zu gewährleisten, befindet sich in jeder Kooperationsgruppe genau ein Aktionsbaum.
- Das geschachtelte Sperrverfahren benutzt das Zwei-Phasen-Sperrprotokoll, während das kooperative Sperrverfahren die drei Phasen der dynamischen kooperativen Serialisierbarkeit realisiert.

Eine Simulation des Zwei-Phasen-Sperrprotokolls ist im kooperativen Sperrverfahren wie folgt möglich: Jede dynamische Aktion tritt direkt zu Beginn der Kooperationsgruppe bei, die ihrem Aktionsbaum entspricht. Ein Verhalten, in dem alle dynamischen Aktionen in der zweiten Phase starten, wird in [Mock 1994] als *statische Kooperationsgruppen* beschrieben. Der Lockpunkt der Toplevel-Aktion im geschachtelten Sperrverfahren entspricht der Validierung und damit dem Wechsel in die dritte Phase. Im Gegensatz zum Setzen des Lockpunkts kann die Validierung normalerweise fehlschlagen, in [Mock 1994] wurde jedoch gezeigt, daß im Falle von statischen Kooperationsgruppen die Validierung immer erfolgreich ist.

Im geschachtelten Sperrverfahren erzwingt die Aktionsverwaltung das zweiphasige Verhalten der dynamischen Aktionen und damit deren Serialisierbarkeit. Das kooperative Sperrverfahren kann diese Zweiphasigkeit nicht erzwingen. Stattdessen wird bezüglich der Anforderung und Freigabe der kooperative Sperren ein Wohlverhalten der einzelnen dynamischen Aktionen vorausgesetzt. Es wird also erwartet, daß sich dynamische Aktionen freiwillig zweiphasig verhalten. In diesem Fall wird nach [Eswaran 1976] die Serialisierbarkeit von Brüdern gewährleistet.

Die Serialisierbarkeit von kompletten Aktionsbäumen ist aufgrund der dynamischen kooperativen Serialisierbarkeit in jedem Fall gewährleistet, selbst wenn die Zweiphasigkeit nicht eingehalten wird. Die Vergaberegeln für kooperative Sperren garantieren, daß Objekte, die bereits vor der Validierung freigegeben worden sein sollten, für dynamische Aktionen aus anderen Aktionsbäumen erst nach der Validierung des ursprünglichen Halters zugreifbar sind, da alle dynamischen Aktionen in der zweiten Phase starten und ihnen deshalb nur der Zugriff auf bereits validierte Objekte erlaubt ist (siehe die Regeln 2.(b1) und 2.(b2) auf Seite 42).

Nach der Validierung ist dynamischen Aktionen nur noch der Zugriff auf Objekte erlaubt, deren letzter Halter aus derselben Kooperationsgruppealso demselben Aktionsbaum kommt. Eine dynamische Aktion kann nach der Validierung also nur Sperren auf solche Objekte anfordern, die bereits schon vor der Validierung von einer dynamischen Aktionen aus demselben Aktionsbaum und in der Zwischenzeit nicht durch dynamische Aktionen aus anderen Aktionsbäumen gesperrt wurden. Die vorübergehende Freigabe des Objekts wurde also außerhalb des Aktionsbaums nicht bemerkt, so daß der Aktionsbaum weiterhin den Anschein eines zweiphasigen Verhaltens macht.

Da keine Änderungen vor der Validierung außerhalb einer Kooperationsgruppe sichtbar werden und nach der Validierung keine Sperren auf zusätzliche Objekte mehr angefordert werden dürfen, erzwingt die dynamische kooperative Serialisierbarkeit von kompletten Aktionsbäumen ein Verhalten, das von dynamischen Aktionen aus anderen Aktionsbäumen nicht von einem nicht-strikten zweiphasigen Verhalten zu unterscheiden ist.

- Im geschachtelten Sperrverfahren werden Sperren von Subaktionen bei deren Ende an den Vater vererbt, um die Zweiphasigkeit von Teilbäumen zu garantieren. Im kooperativen Sperrverfahren ist dieses Vererben nicht mehr vorgesehen, da keine Serialisierbarkeit von Teilbäumen vorgeschrieben ist, und daher auch die Zweiphasigkeit nicht mehr gefordert wird.

  Da die Vererbung fehlt, läßt sich das Verhalten von geschachtelten dynamischen Aktionen nicht exakt simulieren. Die einzige Möglichkeit, die Zweiphasigkeit von Teilbäumen zu garantieren besteht darin, daß dynamische Aktionen vor Aufruf eines Sohnes kooperative Sperren für sämtliche Objekte erwerben, die dieser Sohn und dessen Nachfahren bearbeiten werden. Dies setzt allerdings voraus, daß dem Vater alle diese Objekte bereits vor Aufruf des Sohns bekannt sind, welches in der Praxis oft nicht der Fall sein wird.

Zusammengefaßt läßt sich das geschachtelte Sperrverfahren im kooperativen Sperrverfahren unter folgende Einschränkungen simulieren:

- In jeder Kooperationsgruppe befindet sich genau ein Aktionsbaum und alle dynamischen Aktionen treten sofort bei ihrem Beginn der entsprechenden Kooperationsgruppe bei.
- Alle dynamischen Aktionen verhalten sich zweiphasig und die Validierung der Kooperationsgruppe entspricht dem Setzen des Lockpunkts durch die Toplevel-Aktion.
- Eine dynamische Aktion muß vor dem Aufruf eines Sohns kooperative Sperren auf alle Objekte erwerben, die dieser Sohn und dessen Nachfahren bearbeiten.

Kapitel 4

Formales Modell

4.1 Einleitung

Im vorherigen Kapitel wurde das Konzept der *geschachtelten dynamischen Aktionen für kooperative Anwendungen* als Integrationskonzept der *geschachtelten dynamischen Aktionen* und der *dynamischen Aktionen für kooperative Anwendungen* entwickelt. Viele Eigenschaften des Integrationskonzepts wurden dabei von einem der zugrundeliegenden Konzepte übernommen und die Betrachtungen zur Verträglichkeit haben deutlich gemacht, daß diese Eigenschaften tatsächlich vom Integrationskonzept erfüllt werden. Diese Argumentation wird in diesem Kapitel nun durch den formalen Nachweis der Eigenschaften bestätigt.

Zur Beschreibung von dynamischen Aktionen wurde bereits in [Mock 1994] ein umfangreiches formales Modell entwickelt. Dort werden flache dynamische Aktionen formal beschrieben und es werden unter anderem Eigenschaften der Fehlerbehandlung nachgewiesen. Insbesondere wird gezeigt, daß dynamische Aktionen *abbruch-* und *commit-korrekt* sind. Außerdem erfolgt eine formale Spezifizierung des Kriteriums der *dynamischen kooperativen Serialisierbarkeit* und der Nachweis, daß dynamische Aktionen dieses Kriterium erfüllen, wenn sie sich gemäß dem in Abschnitt 2.6.3 eingeführten und auch im Integrationskonzept benutzten drei-phasigen Schema verhalten.

Da das Integrationskonzept auf dem in [Mock 1994] vorgestellten Konzept der dynamischen Aktionen für kooperative Anwendungen basiert, erweist es sich für den Nachweis der Verträglichkeit als sinnvoll, den dort benutzten Formalismus so zu erweitern, daß auch Struktur und Eigenschaften des Integrationskonzepts formalisiert werden können. Ziel dieses Kapitels ist es deshalb nicht, ein eigenes vollständiges formales Modell zu entwickeln, sondern es wird statt dessen die Verträglichkeit des Integrationskonzepts mit dem Konzept aus [Mock 1994] gezeigt, indem bewiesen wird, daß die dort geforderten und gezeigten Eigenschaften auch im erweiterten formalen Modell gelten. Außerdem wird versucht, die Notation weitestgehend beizubehalten, um einen direkten Bezug zu Definitionen und Sätzen in [Mock 1994] möglich zu machen.

Im einzelnen hat dieses Kapitel folgenden Aufbau:

Zuerst wird, ausgehend von den Definitionen der dynamischen Aktion und des vollständigen Schedules in [Mock 1994], die Baumstruktur der geschachtelten dynamischen Aktionen formal modelliert. Daraufhin wird gezeigt, daß die so modellierten geschachtelten dynamischen Aktionen die in [Mock 1994] definierten Eigenschaften der Abbruch- und Commit-Korrektheit erfüllen. Dieser Nachweis ist aufgrund der Orthogonalität von Fehlerbehandlung und Nebenläufigkeitskontrolle bei dynamischen Aktionen sowohl unabhängig vom zugrundeliegenden Konsistenzkriterium als auch von der Steuerung der Sichtbarkeit von Objektänderungen mittels Objektkontrolle. Daher gilt dieser Nachweis nicht nur für das Integrationskonzept sondern auch für das zugrundeliegende Konzept der geschachtelten dynamischen Aktionen. Anschließend wird ein Modell für geschachtelte dynamische Aktionen aufgestellt, das auch den Mechanismus der Objektkontrolle beinhaltet und somit das Integrationskonzept umfassend modelliert. Für dieses Modell wird zum Nachweis der Verträglichkeit gezeigt, daß es trotz erweiterter Möglichkeiten zur Steuerung der Nebenläufigkeit das in [Mock 1994] definierte Kriterium der dynamischen kooperativen Serialisierbarkeit erfüllt.

4.2 Grundlagen und Basisdefinitionen

Der Formalismus in diesem Kapitel basiert auf einer in [Hadzilacos 1988] entwickelten Theorie, der die Herbrand-Semantik von *Schedules* zugrundeliegt. Mit Hilfe dieser Theorie lassen sich sowohl Eigenschaften der Fehlertoleranz als auch der Nebenläufigkeit gut modellieren. Die bereits in [Mock 1994] beschriebenen Grundlagen der Theorie werden im folgenden kurz zusammengefaßt.

Ein Schedule beschreibt die Beziehung von *Ereignissen*, den sogenannten *Schritten* des Schedules, in Form einer *partiellen Ordnung*. Grundsätzlich gibt es dabei zwei unterschiedliche Arten von Schritten: die Objektoperationen, die den Zugriff auf den Objektraum realisieren, und die Ereignisse, die etwas über den Zustand der dynamischen Aktionen aussagen. Diese Ereignisse werden im Laufe des Kapitels eingeführt.

Definition 1 Objektraum oder Datenbank

Ein Objektraum $D = \{x, y, z, \ldots\}$ ist eine endliche Menge von Objekten. Er wird auch Datenbank genannt. ∎

Auf diesem Objektraum werden die Objektoperation definiert. Dabei wird von dem klassischen Schreib-/Lesemodell ausgegangen:

Definition 2 Objektoperation

Eine Objektoperation ist ein Element der Menge $OP = \{R, W\} \times D$. Die Operationen werden dabei üblicherweise nicht als Tupel, sondern in den Formen $R(x)$ und $W(x)$ mit $x \in D$ notiert. $R(x)$ heißt „Leseoperation auf dem Objekt x“ und $W(x)$ analog „Schreiboperation auf dem Objekt x“. ∎

Wenn der Effekt zweier Objektoperationen von ihrer Reihenfolge abhängt, stehen sie in **Konflikt**. Dieses ist genau dann der Fall, wenn sie dasselbe Objekt betreffen und mindestens eine von ihnen eine Schreiboperation ist.

4.2.1 Modellierung geschachtelter dynamischer Aktionen

Zur Modellierung der geschachtelten dynamischen Aktionen wird hier zuerst die Definition der dynamischen Aktion aus [Mock 1994] wiedergegeben. Die Schachtelungsstruktur wird anschließend bei der Definition eines vollständigen Schedules eingeführt.

Um dynamische Aktionen modellieren zu können, müssen noch zwei wichtige Ereignisse eingeführt werden, die sogenannten *Terminierungsereignisse*. Dieses sind die Ereignisse *Abbruch* einer dynamischen Aktion und *Complete*, welches besagt, daß die dynamische Aktion keine weitere Objektoperationen mehr ausführen wird. Es erlaubt aber im Gegensatz zum sonst üblichen Ereignis *Commit*, daß die dynamische Aktion anschließend noch abgebrochen werden kann, zum Beispiel durch einen kaskadierenden Aktionsfehler. Dieser Unterschied ist notwendig, da bei kooperativen Anwendungen eine Commit-Entscheidung oft nicht von einer einzelnen dynamischen Aktion autonom gefällt werden kann. Durch eine Beendigung mit dem Complete-Ereignis erklärt eine dynamische Aktion ihre Commit-Bereitschaft. Das Commit-Ereignis wird später in Abschnitt 4.3.2 eingeführt.

Eine dynamische Aktion läßt sich jetzt wie folgt definieren:

Definition 3 Dynamische Aktion

Eine dynamische Aktion ist ein Paar $A_i = (OP^{(i)}, \prec^{(i)}), i \in \mathbb{N}$ mit

1. $OP^{(i)} = \{o_{i1}, o_{i2}, \ldots, o_{im_i}\}, m_i \in \mathbb{N}$ der Menge der Objektoperationen von A_i
2. $\prec^{(i)} \subseteq (OP^{(i)} \cup \{a_i, t_i\}) \times (OP^{(i)} \cup \{a_i, t_i\})$ der (Intra-Aktions-) Vorrangrelation von A_i mit den folgenden Eigenschaften:
   (a) $\prec^{(i)}$ ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch
   (b) Stehen zwei Objektoperationen von A_i, o_{ik} und o_{il} miteinander in Konflikt, so gilt: $o_{ik} \prec^{(i)} o_{il} \vee o_{il} \prec^{(i)} o_{ik}$
   (c) Es gilt:
   $$\begin{aligned} & (\forall\, o \in OP^{(i)} :\; o \prec^{(i)} a_i \;\wedge\; \neg(o \prec^{(i)} t_i \;\vee\; t_i \prec^{(i)} o)) \\ \vee\; & (\forall\, o \in OP^{(i)} :\; o \prec^{(i)} t_i \;\wedge\; \neg(o \prec^{(i)} a_i \;\vee\; a_i \prec^{(i)} o)) \\ \vee\; & (\forall\, o \in OP^{(i)} :\; o \prec^{(i)} t_i \;\wedge\; t_i \prec^{(i)} a_i) \end{aligned}$$

∎

Die Unterscheidung aller Objektoperationen der dynamischen Aktion in Punkt 1 durch die Indizes erlaubt es, mehrere gleiche Operationen auf demselben Objekt zu modellieren. Dieses ist für die nicht-serielle Ausführung der dynamischen Aktionen bei Kooperation notwendig. Punkt 2 der Definition spezifiziert über die (Intra-Aktions-) Vorrangrelation die Reihenfolge der Operationen und der beiden möglichen Terminierungsereignisse t_i für Complete und a_i für den Abbruch der dynamischen Aktion. Dabei muß zwischen Operationen, die sich nicht in Konflikt befinden, keine Reihenfolge spezifiziert werden. Umgekehrt wird aber in Punkt 2.b) verlangt, daß zwischen im Konflikt stehenden Operationen eine Reihenfolge spezifiziert ist. Punkt 2.c) fordert außerdem, daß zwischen jeder Objektoperation und den Terminierungsereignissen eine Reihenfolge festgelegt ist. Genauer wird gefordert, daß entweder

- alle Objektoperationen vor dem Abbruch-Ereignis liegen und das Complete-Ereignis nicht auftritt (Zeile 1), oder
- alle Objektoperationen vor dem Complete-Ereignis liegen und das Abbruch-Ereignis nicht auftritt (Zeile 2), oder
- alle Objektoperationen vor dem Complete-Ereignis liegen und anschließend nur noch das Abbruch-Ereignis auftritt (Zeile 3).

Damit ist das Verhalten einer einzelnen dynamischen Aktion beschrieben. Zur Beschreibung mehrerer nebenläufiger dynamischer Aktionen, die auch in einem Vater-Sohn-Verhältnis stehen können, wird im folgenden der *Schedule* eingeführt. Zuvor muß aber noch eine Möglichkeit geschaffen werden, Vater-Sohn-Verhältnisse zu modellieren, welches mit Hilfe des sogenannte *Starte-Sohn-Ereignisses* geschieht.

Definition 4 Starte-Sohn-Ereignis

Das Ereignis $s_i(j)$ mit $i \neq j$ besagt, daß die dynamische Aktion A_i die dynamische Aktion A_j als Sohn startet. ∎

Mit dieser Definition und der Menge $T^{(i)}$, die die Menge der tatsächlich auftretenden Terminierungsereignisse der dynamischen Aktion A_i beschreibt (für die genaue Definition siehe [Mock 1994] Def. 5), kann jetzt ein Schedule mit Starte-Sohn-Ereignissen definiert werden:

Definition 5 Vollständiger Schedule mit Starte-Sohn-Ereignissen

Sei $A = \{A_1, A_2, \ldots, A_n\}, n \in \mathbb{N}$ eine Menge von dynamischen Aktionen. Ein vollständiger Schedule mit Starte-Sohn-Ereignissen für A ist ein Paar $S = (OP^S, \prec^S)$ mit:

1. $ST^S = \{s_i(j) \mid i, j \leq n \wedge i \neq j\}$ der Menge der Starte-Sohn-Ereignisse von S, für die gilt $\forall\, i, j, k \leq n :\; s_j(i) \in ST^S \wedge s_k(i) \in ST^S \Longrightarrow j = k$.

   Außerdem wird für jedes $i \leq n$ definiert: $ST^{(i)} = \{s_i(j) \mid s_i(j) \in ST^S\}$
2. $OP^S = \bigcup_{i=1}^{n}(OP^{(i)} \cup T^{(i)}) \;\cup\; ST^S$ der Menge der Schritte von S.
3. $\prec^S \;\subseteq\; OP^S \times OP^S$ der (Schedule-) Vorrangrelation von S mit den folgenden Eigenschaften:
   (a) $\prec^S$ ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch
   (b) stehen zwei Objektoperationen von S, p und q miteinander in Konflikt, so gilt: $p \prec^S q \vee q \prec^S p$
   (c) $\prec^S \supseteq \bigcup_{i=1}^{n} \prec^{(i)}$
   d.h. die (Schedule-) Vorrangrelation ist konform mit allen Intra-Aktions-Vorrangrelationen der dynamischen Aktionen von S.
   (d) $s_j(i) \in OP^S \Longrightarrow (\forall\, o \in T^{(j)} :\; s_j(i) \prec^S o)$
   d.h. dynamische Aktionen dürfen nur vor ihrer Terminierung Söhne starten.
   (e) $s_j(i) \in OP^S \Longrightarrow (\forall\, o \in OP^{(i)} \cup ST^{(i)} \cup T^{(i)} :\; s_j(i) \prec^S o)$
   d.h. Sohnaktionen dürfen erst nach ihrem Start-Ereignis Operationen ausführen.

∎

Im ersten Punkt dieser Definition wird die Menge der Starte-Sohn-Ereignisse eines Schedules beschrieben. Die Zusatzbedingung stellt dabei sicher, das jede dynamische Aktion durch höchstens ein solches Ereignis gestartet wird. Die Mengen $ST^{(i)}$ umfassen jeweils die Starte-Sohn-Ereignisse, die von der dynamischen Aktion A_i initiiert werden.

Punkt 2 definiert die Menge der Schritte des vollständigen Schedules als die Vereinigung aller Schritte und Terminierungsereignisse der dynamischen Aktionen mit der Menge der Starte-Sohn-Ereignisse.

Schließlich wird in Punkt 3 die Schedule-Vorrangrelation definiert. Dabei wird insbesondere gefordert, daß für zwei in Konflikt stehende Objektoperationen eine Reihenfolge festgelegt sein muß, auch wenn die Operationen von unterschiedlichen dynamischen Aktionen ausgeführt werden (Punkt 3.b).

Die Punkte 3.d) und 3.e) spezifizieren die Starte-Sohn-Ereignisse genauer. Dabei darf ein Starte-Sohn-Ereignis nur vor der Terminierung des Vaters auftreten und alle Objektoperationen, Terminierungs- und insbesondere auch weitere Starte-Sohn-Ereignisse des Sohnes dürfen erst auftreten, nachdem der Sohn gestartet wurde. Diese beiden Regeln schränken implizit die Wahl der Menge ST^S aus Punkt 1 der Definition ein, wie im folgenden Abschnitt genauer erläutert wird. Bezüglich des Endes eines Sohnes enthält diese Definition keine weiteren Einschränkungen, die über die Definition der dynamischen Aktion hinausgehen. Es gibt insbesondere keine Forderung, daß ein Sohn vor seinem Vater beendet sein müßte, allerdings hat eine Vater-Sohn-Beziehung Einfluß auf die Durchsetzung von Abbruch- und Commit-Korrektheit, wie später beschrieben wird.

Aus dem so definierten vollständigen Schedule mit Starte-Sohn-Ereignissen läßt sich analog zur Definition 7 aus [Mock 1994] der **Schedule mit Starte-Sohn-Ereignissen** als Präfix definieren.

4.2.2 Baumstruktur der dynamischen Aktionen

In diesem Abschnitt soll die Struktur genauer untersucht werden, die die Starte-Sohn-Ereignisse bezüglich der dynamischen Aktionen eines Schedules erzeugen. Es wird nachgewiesen, daß diese Ereignisse die bekannte Baumstruktur der geschachtelten dynamischen Aktionen induzieren, und dadurch wird deutlich, welche Einschränkungen für die Menge ST^S der Starte-Sohn-Ereignisse aus Definition 5 gelten.

Voraussetzungen für die folgenden Definitionen und Sätze

Sei $S = (OP^S, \prec^S)$ ein Schedule zu $A = \{A_1, A_2, \ldots, A_n\}, n \in \mathbb{N}$.

A_i, A_j, A_k, A_l, A_m und A_t seien aus A.

Ein Starte-Sohn-Ereignis repräsentiert die Vater-Sohn-Beziehung zwischen zwei dynamischen Aktionen, die durch die „Ist-Sohn-von"-Relation formalisiert wird.

Definition 6 „Ist-Sohn-von"-Relation

$$A_i \longmapsto_S A_j \quad (A_j \text{ ist Sohn von } A_i) \quad :\Longleftrightarrow \quad \exists\ s_i(j) \in OP^S$$

Eine andere Sprechweise ist: A_i ist Vater von A_j. ∎

Für die „Ist-Sohn-von"-Relation gelten folgende Eigenschaften:

Satz 1

- Die „Ist-Sohn-von"-Relation ist irreflexiv, eine dynamische Aktion ist also niemals ihr eigener Sohn.
- Eine dynamische Aktion kann höchstens Sohn von *einer* anderen dynamischen Aktion sein, also höchstens einen Vater haben.

  Das heißt, es gilt: $A_i \longmapsto_S A_k \ \wedge\ A_j \longmapsto_S A_k \implies A_i = A_j$

∎

Beweis:

Beide Aussagen folgen direkt aus der Definition der „Ist-Sohn-von"-Relation und aus Definition 5:

Damit die „Ist-Sohn-von"-Relation reflexiv wäre, müßte $s_i(i) \in OP^S$ für ein i gelten. Dieses ist aber bei der Definition der Menge ST^S ausgeschlossen worden.

Die zweite Aussage würde bedeuten, daß gilt: $s_i(k) \in OP^S \ \wedge\ s_j(k) \in OP^S \ \wedge\ i \neq j$. Auch dieses wurde bei der Definition der Menge ST^S ausgeschlossen. ∎

Es ist andererseits aber möglich, daß eine dynamische Aktion A_i mehrere Söhne, z. B. A_j und A_k, hat. Es würde dann also gelten: $s_i(j) \in OP^S \ \wedge \ s_i(k) \in OP^S \ \wedge \ j \neq k$.

Aufbauend auf der „Ist-Sohn-von"-Relation wird jetzt die Baumstruktur der dynamischen Aktionen definiert. Dazu wird zuerst mit der „Ist-Nachfahre-von"-Relation ihre transitive Hülle eingeführt:

Definition 7 „Ist-Nachfahre-von"-Relation

$$\begin{aligned} &A_i \longmapsto_S^* A_j \quad (A_j \text{ ist Nachfahre von } A_i) \\ &:\Longleftrightarrow \exists\ A_{k_1}, \ldots, A_{k_m}, m \in \mathbb{N} : A_i = A_{k_1} \wedge A_j = A_{k_m} \ \wedge \ A_{k_1} \longmapsto_S A_{k_2} \longmapsto_S \cdots \longmapsto_S A_{k_m} \end{aligned}$$

Andere Sprechweise: A_i ist Vorfahre von A_j. ∎

Satz 2

Die „Ist-Nachfahre-von"-Relation ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch. ∎

Beweis:

Transitivität: Zu zeigen ist: $A_i \longmapsto_S^* A_j \ \wedge \ A_j \longmapsto_S^* A_k \ \Longrightarrow \ A_i \longmapsto_S^* A_k$

Dies folgt direkt aus der Definition der „Ist-Nachfahre-von"-Relation: Nach Definition gibt es eine Kette von „Ist-Sohn-Von"-Relationen von A_i nach A_j und eine solche Kette von A_j nach A_k. Da A_j Ende der ersten und Anfang der zweiten Kette ist, gibt es über A_j auch eine Kette von A_i nach A_k.

Irreflexivität: Zu zeigen ist, daß eine dynamische Aktion nie ihr eigener Nachfahre sein kann. Der Beweis erfolgt indirekt.

Angenommen, A_i wäre ihr eigener Nachfahre. Dann gäbe es eine Kette von „Ist-Sohn-von"-Relationen von A_i nach A_i:

$$\exists\ A_{k_1}, \ldots, A_{k_m}, m \in \mathbb{N} : A_i = A_{k_1} \wedge A_i = A_{k_m} \ \wedge \ A_{k_1} \longmapsto_S A_{k_2} \longmapsto_S \cdots \longmapsto_S A_{k_m}$$

Die Menge der Schritte OP^S müßte dann folgende Ereignisse enthalten:

$$s_i(k_2), s_{k_2}(k_3), \ldots, s_{k_{m-1}}(i)$$

Aufgrund von Definition 5.3.e) muß dann auch gelten:

$$s_i(k_2) \prec^S s_{k_2}(k_3) \prec^S \ldots \prec^S s_{k_{m-1}}(i)$$

Da auch $s_{k_{m-1}}(i) \prec^S s_i(k_2)$ gelten müßte, $\prec^S$ aber eine Halbordnung ist, folgt der Widerspruch.

Asymetrie: Zu zeigen ist: Es gibt keine dynamischen Aktionen A_i und A_j, wobei A_i Vorfahre von A_j und A_j Vorfahre von A_i ist.

Der Beweis erfolgt analog zu dem der Irreflexivität durch Rückführung auf die Tatsache, daß $\prec^S$ eine Halbordnung ist. Auf eine genaue Ausformulierung wird hier verzichtet.

∎

Die beiden letzten Sätze machen die induzierte Baumstruktur deutlich: Zum einen wurde gezeigt, daß jede dynamische Aktion höchstens einen Vater hat, zum anderen ist die „Ist-Nachfahre-von"-Relation zyklenfrei, da es sich um eine partielle Ordnung handelt. Anhand dieser Relation wird die Menge A der dynamischen Aktionen in möglicherweise mehrere Bäume strukturiert. Es können dabei auch einelementige „Bäume" auftreten, wenn dynamische Aktionen weder einen Vater noch Söhne haben. Um diese Aktionsbäume für spätere Aussagen und Beweise handhabbar zu machen, werden jetzt noch einige Begriffe eingeführt.

Zuerst wird die sogenannte Toplevel-Aktion definiert:

Definition 8 Toplevel-Aktion

$$A_t \text{ ist eine Toplevel-Aktion } :\Longleftrightarrow \quad \nexists\, A_i \in A : \; A_i \longmapsto_S A_t$$

∎

Der folgende Satz zeigt, daß jede dynamische Aktion, die nicht selbst Toplevel-Aktion ist, genau eine Toplevel-Aktion als Vorfahre hat.

Satz 3

Jede dynamische Aktion A_i ist entweder

- Toplevel-Aktion, oder
- es existiert genau eine Toplevel-Aktion A_t mit $A_t \longmapsto_S^* A_i$. ∎

Beweis:

Angenommen A_i ist keine Toplevel-Aktion. Dann muß die Existenz und Eindeutigkeit einer solchen Toplevel-Aktion A_t mit $A_t \longmapsto_S^* A_i$ nachgewiesen werden.

Existenz: Da A_i keine Toplevel-Aktion ist, existiert nach Definition 8 eine dynamische Aktion A_{k_1} mit $A_{k_1} \longmapsto_S A_i$. Entweder ist A_{k_1} eine Toplevel-Aktion, oder es existiert wieder ein Vater. Es existiert also eine Folge von Vater-Sohn-Beziehungen der Art: $\cdots \longmapsto_S A_{k_2} \longmapsto_S A_{k_1} \longmapsto_S A_i$

Diese Folge muß einen festen Anfang haben, da die „Ist-Nachfahre-von"-Relation irreflexiv und die Anzahl der dynamischen Aktionen in A endlich ist. Also existiert eine solche Toplevel-Aktion.

Eindeutigkeit: Angenommen, es gäbe zwei solche Toplevel-Aktionen. Dann müßte eine dynamische Aktion zwei Väter haben. Dies ist aber nach Definition 5.1 verboten. ∎

Der Satz zeigt, daß jeder dynamischen Aktion A_i eindeutig eine Toplevel-Aktion zugeordnet werden kann. Diese wird mit $\mathrm{TOP}_S(A_i)$ bezeichnet und ist folgendermaßen definiert:

Definition 9 Zugeordnete Toplevel-Aktion

$$\mathrm{TOP}_S(A_i) := \begin{cases} A_i & \text{wenn } A_i \text{ eine Toplevel-Aktion ist} \\ A_t \text{ wobei } A_t \text{ Toplevel-Aktion mit } A_t \longmapsto_S^* A_i & \text{sonst.} \end{cases}$$

∎

Im folgenden werden noch zwei Mengen definiert: Dieses sind zum einen die Menge aller dynamischen Aktionen, die zu einem Teilbaum gehören, der eine bestimmte dynamische Aktion A_i als Wurzel hat. Darauf aufbauend wird die Menge aller dynamischen Aktionen, die den ganzen Aktionsbaum bilden, zu dem die dynamische Aktion A_i gehört, als der Teilbaum definiert, dessen Wurzel die A_i zugeordnete Toplevel-Aktion ist.

Definition 10 Teilbaum

Die Menge $\text{TEILBAUM}_S(A_i)$ enthält den Teilbaum mit der Wurzel A_i, d.h. A_i und alle Nachfahren.

$$\text{TEILBAUM}_S(A_i) := \{A_i\} \cup \{A_j |\ A_i \longmapsto^*_S A_j\}$$

∎

Die Menge $\text{BAUM}_S(A_i)$ enthält alle dynamischen Aktionen des kompletten Aktionsbaums, zu dem auch A_i gehört.

Definition 11 Aktionsbaum

$$\text{BAUM}_S(A_i) := \text{TEILBAUM}_S(\text{TOP}_S(A_i))$$

∎

Es muß jetzt noch bewiesen werden, daß $\text{BAUM}_S(A_i)$ tatsächlich die geforderten Eigenschaften erfüllt, nämlich daß A_i im entsprechenden Aktionsbaum $\text{BAUM}_S(A_i)$ liegt, und daß die Menge $\text{BAUM}_S(A_i)$ einem kompletten Aktionsbaum entspricht. Dies bedeutet, daß keine dynamische Aktion des Aktionsbaumes einen Vater oder Sohn hat, der außerhalb dieses Aktionsbaums liegt, und daß alle dynamischen Aktionen Teil desselben Aktionsbaums sind. Die letzte Aussage gilt trivialerweise, da die Menge $\text{BAUM}_S(A_i)$ genau als die Menge aller dynamischen Aktionen definiert ist, die von einer dynamischen Aktion abstammen. Ansonsten ist noch zu zeigen:

Satz 4

- $A_i \in \text{BAUM}_S(A_i)$
- $\forall\ A_i : A_j \in \text{BAUM}_S(A_i) \implies \{A_k \mid A_k \longmapsto_S A_j\} \subseteq \text{BAUM}_S(A_i) \wedge \{A_l \mid A_j \longmapsto_S A_l\} \subseteq \text{BAUM}_S(A_i)$

∎

Beweis:

Beide Aussagen folgen unmittelbar aus der Definition von $\text{BAUM}_S(A_i)$.

Punkt 1: $\text{BAUM}_S(A_i)$ ist als der Teilbaum unter der A_i zugeordneten Toplevel-Aktion definiert. Fall die dynamische Aktion A_i selbst die Toplevel-Aktion ist, liegt A_i in $\text{BAUM}_S(A_i)$, da sie nach Definition 10 in $\text{TEILBAUM}_S(A_i)$ liegt. Ist A_i keine Toplevel-Aktion, ist $\text{TOP}_S(A_i)$ so definiert, daß A_i Nachfahre von $\text{TOP}_S(A_i)$ ist. Da in $\text{TEILBAUM}_S(\text{TOP}_S(A_i))$ alle Nachfahren von $\text{TOP}_S(A_i)$ liegen, gilt die Aussage.

Punkt 2: Alle Väter einer dynamischen Aktion aus $\text{BAUM}_S(A_i)$ liegen auch in $\text{BAUM}_S(A_i)$, da jede dynamische Aktion höchstens einen Vater hat und nach Definition jede dynamische Aktion in

$\mathrm{BAUM}_S(A_i)$ Nachfahre von $\mathrm{TOP}_S(A_i)$ ist. Also ist auch der Vater ein Nachfahre von $\mathrm{TOP}_S(A_i)$ und muß deshalb in $\mathrm{BAUM}_S(A_i)$ liegen. Mit dem gleichen Argument müssen auch alle Söhne einer dynamischen Aktion aus $\mathrm{BAUM}_S(A_i)$ selbst in $\mathrm{BAUM}_S(A_i)$ liegen, da auch sie Nachfahren von $\mathrm{TOP}_S(A_i)$ sind. ∎

Mit den bis hier gemachten Definitionen und Sätzen läßt sich die Schachtelung von dynamischen Aktionen in einem Schedule ausreichend beschreiben. Alle Aussagen beziehen sich dabei aber immer auf einen bestimmten Schedule S. Es wurden aber noch keine Aussagen darüber gemacht, wie sie sich bei Erweiterung des Schedules verhalten. Dieses Verhalten wird durch den folgenden Satz beschrieben.

Satz 5 Erweiterung von Schedules

Sei der Schedule S' ein beliebiger Erweiterung des Schedules S. Dann gilt:

1. $A_i \longmapsto_S A_j \Longrightarrow A_i \longmapsto_{S'} A_j$
2. $A_i \longmapsto^*_S A_j \Longrightarrow A_i \longmapsto^*_{S'} A_j$
3. A_i ist Toplevel-Aktion in S. Dann ist sie es auch in S'.
4. $\mathrm{TOP}_S(A_i) = A_t \Longrightarrow \mathrm{TOP}_{S'}(A_i) = A_t$
5. $\mathrm{TEILBAUM}_S(A_i) \subseteq \mathrm{TEILBAUM}_{S'}(A_i)$
6. $\mathrm{BAUM}_S(A_i) \subseteq \mathrm{BAUM}_{S'}(A_i)$

∎

Beweis:

1. Nach Definition der „Ist-Sohn-von"-Relation gilt: $A_i \longmapsto_S A_j \Longleftrightarrow s_i(j) \in OP^S$. Da für die Erweiterung eines Schedules gilt: $OP^S \subseteq OP^{S'}$, gilt auch: $s_i(j) \in OP^{S'}$. Daraus folgt: $A_i \longmapsto_{S'} A_j$.
2. Diese Aussage folgt direkt aus Punkt 1, da die „Ist-Nachfahre-von"-Relation die transitive Hülle der „Ist-Sohn-von"-Relation ist.
3. Wenn A_i in S' keine Toplevel-Aktion mehr wäre, müßte es in S' einen Vorfahren A_k von A_i geben, den es in S noch nicht gab. Es müßte also ein Ereignis $s_k(i)$ zum Schedule S hinzugekommen sein. Wenn dieses Ereignis im Schedule existiert, darf A_i aber erst nach diesem Ereignis Operationen ausführen. Da A_i aber schon in S existierte und deshalb dort schon mindestens eine Operation ausgeführt haben muß, folgt der Widerspruch.
4. Wenn A_i selbst eine Toplevel-Aktion in S ist, so ist sie es nach Punkt 3 auch in jeder Erweiterung S'. In diesem Fall ist A_i ihre eigene zugeordnete Toplevel-Aktion und diese ändert sich auch nicht in einer Erweiterung von S. Wenn A_i keine Toplevel-Aktion in S ist, ist $\mathrm{TOP}_S(A_i)$ eine Toplevel-Aktion, die Vorfahre von A_i ist. Diese Toplevel-Aktion bleibt nach Punkt 3 auch in S' eine Toplevel-Aktion und bleibt nach Punkt 2 auch Vorfahre von A_i. Sie ist also auch weiterhin die zugeordnete Toplevel-Aktion von A_i in S'.
5. Jeder Nachfahre von A_i in S ist nach Punkt 2 auch Nachfahre von A_i in S'. Deshalb gilt die geforderte Teilmengenbeziehung. Die Gleichheit der beiden Mengen kann nicht zugesichert werden, da durch weitere Starte-Sohn-Ereignisse noch weitere Nachfahren hinzukommen könnten.

6. Dieser Punkt folgt direkt aus den Punkten 4 und 5.

■

4.3 Fehlerbehandlung und Sicherung von geschachtelten dynamischen Aktionen

Mit den dargestellten Möglichkeiten zur Beschreibung von geschachtelten dynamischen Aktionen werden jetzt wichtige Eigenschaften bewiesen. In diesem Abschnitt wird gezeigt, daß dynamische Aktionen auch bei Berücksichtigung von Schachtelung wichtige Eigenschaften im Bereich der Fehlertoleranz erfüllen. Die im Zusammenhang dieser Arbeit zentralen Aspekte der Nebenläufigkeit werden anschließend im nächsten Abschnitt behandelt. Da Fehlertoleranz für Transaktionssysteme im allgemeinen und daher auch für dynamische Aktionen ein sehr wichtiger Aspekt ist, müssen diese Eigenschaften auch für das in dieser Arbeit eingeführte Integrationskonzept nachgewiesen werden. Das Integrationskonzept unterschiedet sich im Hinblick auf die Fehlertoleranz nicht von dem Konzept der geschachtelten dynamischen Aktionen. Da für dieses Konzept noch keine Einbindung in das in [Mock 1994] eingeführte, auf Schedules basierende formale Modell besteht, wird diese Einbindung jetzt durchgeführt.

Die beiden zentralen Eigenschaften im Bereich der Fehlertoleranz sind die sogenannte *Abbruch-* und die *Commit-Korrektheit*. In [Mock 1994] wurde gezeigt, daß flache dynamische Aktionen diese Eigenschaften erfüllen, da eine dynamische Aktion einen Abbruch oder Commit nicht autonom durchführt, sondern seitens der Transaktionsverwaltung auch informationsabhängige dynamische Aktionen berücksichtigt werden. Die Informationsabhängigkeiten wurden bei [Mock 1994] durch die „Liest-von“-Relation modelliert, die grundlegend für die Korrektheitsbeweise ist, da mit Hilfe dieser Relation die Menge der dynamischen Aktionen bestimmt werden kann, die von einem Abbruch oder Commit betroffen sind. Bei geschachtelten dynamischen Aktionen läßt sich diese Menge nicht mehr allein durch die „Liest-von“-Relation bestimmen. Zusätzlich zu dieser muß auch die Schachtelungsstruktur beachtet werden, wobei diese Struktur bei Abbruch und Commit jeweils eine unterschiedliche Auswirkung hat.

In diesem Abschnitt wird die „Liest-von“-Relation für den Beweis der Abbruch- und der Commit-Korrektheit jeweils durch eine neue Relation ersetzt, die zusätzlich das Verhalten von geschachtelten dynamischen Aktionen modellieren. Unter Verwendung dieser neuen Relationen bleiben alle Beweise, wie sie in [Mock 1994] für flache dynamische Aktionen geführt wurden, auch für geschachtelte dynamische Aktionen gültig. Daher wird in dieser Arbeit nur das geänderte Aktionsmodell dargelegt, aber auf die vollständige Ausformulierung der Korrektheitsbeweise verzichtet. Statt dessen wird auf die entsprechenden Passagen in [Mock 1994] verwiesen.

4.3.1 Abbruch-Korrektheit

Die Ausführung einer Menge von dynamischen Aktionen ist genau dann *abbruch-korrekt*, wenn jeder Fehler dazu führt, daß alle dynamischen Aktionen abgebrochen werden, deren Berechnungen direkt oder indirekt von diesem Fehler abhängen.

Im formalen Modell wird die Eigenschaft der Abbruch-Korrektheit über die Herbrand-Semantik des Schedules definiert: Ein Schedule ist dann abbruch-korrekt, wenn alle dynamischen Aktionen abgebrochen sind, die solche Operationen ausgeführt haben, deren Semantik am Ende des Schedules sich von der Semantik während der Ausführung der Operationen unterscheidet. Diese Eigenschaft wird formal in [Mock 1994], Definition 16 eingeführt.

In [Mock 1994] werden dann in Satz 22 und Korollar 26 zwei Kriterien für die Abbruch-Korrektheit erarbeitet. Die Überprüfung dieser Kriterien basiert auf der in [Mock 1994], Definition 20 eingeführten *„Liest-von“-Relation*. Diese Relation wurde dort auf Grundlage der Herbrand-Semantik eingeführt und besagt folgendes: Eine dynamische Aktion A_i liest direkt von A_j (Schreibweise $A_j \underset{LV_S}{\longrightarrow} A_i$), wenn A_i ein Objekt liest, das als letztes zuvor A_j geschrieben hat. Die „Liest-von“-Relation entspricht somit den in Abschnitt 2.6 eingeführten Informationsabhängigkeiten. Über die „Liest-von“-Relation lassen sich zum Beispiel mit der Abhängigkeitsmenge $AB_S(A_i)$ einer dynamischen Aktion A_i diejenigen dynamischen Aktionen bestimmen, deren Abbruch zur Einhaltung der Abbruch-Korrektheit auch den Abbruch von A_j nach sich ziehen muß.

Bei geschachtelten dynamischen Aktionen reicht die „Liest-von“-Relation als alleiniges Bestimmungsmittel der Abhängigkeitsmenge nicht mehr aus, vielmehr muß eine dynamische Aktion bei einem Abbruch des Vater selbst auch abbrechen, unabhängig davon, ob sie zu ihm auch in „Liest-von“-Relation steht. Daher wird hier die *Abbruch-Abhängigkeit* eingeführt, die neben der „Liest-von“-Relation auch Vater-Sohn-Beziehungen berücksichtigt.

Definition 12 Abbruch-Abhängigkeit

A_j ist in S direkt abbruch-abhängig von A_i (Schreibweise: $A_i \underset{AAB_S}{\longrightarrow} A_j$)

$$:\Longleftrightarrow \begin{cases} A_i \underset{LV_S}{\longrightarrow} A_j & \text{oder} \\ A_j \in \text{TEILBAUM}_S(A_i) & \end{cases}$$

Die Abbruch-Abhängigkeit ist die transitive Hülle der direkten Abbruch-Abhängigkeit:

A_j ist abbruch-abhängig von A_i $(A_i \underset{AAB_S^*}{\longrightarrow} A_j)$

$$:\Longleftrightarrow \exists\, A_{k_1}, \ldots, A_{k_m}, m \in \mathbb{N} : A_i = A_{k_1} \wedge A_j = A_{k_m} \;\wedge\; A_{k_1} \underset{AAB_S}{\longrightarrow} A_{k_2} \underset{AAB_S}{\longrightarrow} \cdots \underset{AAB_S}{\longrightarrow} A_{k_m}$$

∎

Mit dieser Abbruch-Abhängigkeit läßt sich jetzt die Abbruch-Abhängigkeitsmenge definieren. Diese Definition entspricht der Abhängigkeitsmenge in [Mock 1994], Definition 20, Satz 4.

Definition 13 Abbruch-Abhängigkeitsmenge

$$AAB_S(A_i) := \{A_j \in A \mid A_j \underset{AAB_S^*}{\longrightarrow} A_i\}$$

∎

Neben der Abhängigkeitsmenge, die diejenigen dynamischen Aktionen spezifiziert, von deren Abbruch eine dynamische Aktion betroffen ist, wird in [Mock 1994], Definition 24 noch die Nachfolgermenge $NF_S(A_i)$ der dynamischen Aktion A_i definiert, die diejenigen dynamischen Aktionen enthält, die von einem Abbruch von A_i betroffen sind. Auch diese Definition muß auf Basis der Abbruch-Abhängigkeit ergänzt werden:

Definition 14 Abbruch-Nachfolgermenge

$$ANF_S(A_i) := \{A_j \in A \mid A_i \underset{AAB_S^*}{\longrightarrow} A_j\}$$

∎

Diese drei neuen Definitionen integrieren bezüglich der Abbruch-Korrektheit die Schachtelung in das formale Modell. Sowohl die Kriterien der Abbruch-Korrektheit als auch alle anderen Sätze, Eigenschaften und Verfahren aus den Abschnitten 3.2 „Fehlerausbreitung und -bewertung“ und 3.3 „Fehlerbehebung“ in [Mock 1994] bleiben gültig, sofern die „Liest-von“-Relation durch die Abbruch-Abhängigkeit und die Mengen $AB_S(A_i)$ durch $AAB_S(A_i)$ bzw. $NF_S(A_i)$ durch $ANF_S(A_i)$ ersetzt werden. Auf die exakte Ausformulierung wird hier verzichtet.

Als zu Beginn dieses Kapitels in Definition 5 der vollständige Schedule mit Starte-Sohn-Ereignissen eingeführt wurde, wurden keine Aussagen über die Terminierung der Sohn-Aktionen gemacht. Das Kriterium der Abbruch-Korrektheit eines Schedules S zeigt aber, daß es durchaus eine Beziehung zwischen der Terminierung eines Vaters A_i und seines Sohnes A_j gibt: Wenn der Vater abgebrochen ist, also $a_i \in OP^S$ gilt, muß auch $a_j \in OP^S$ gelten. Diese Anforderung zeigt auch, daß falls der Vater vor dem Sohn abgebrochen würde, die Abbruch-Korrektheit kurzzeitig bis zum Abbruch des Sohnes verletzt wäre.

4.3.2 Modellierung geschachtelter dynamischer Aktionen mit Commit-Ereignissen

Bisher wurde noch nicht untersucht, unter welchen Bedingungen Operationen von dynamischen Aktionen nicht mehr durch Abbrüche zurückgesetzt werden können. Die Definition 3 der dynamischen Aktionen sieht die zwei Terminierungsereignisse Abbruch und Complete vor, wobei aber auch nach dem Complete-Ereignis noch ein Abbruch möglich ist. In diesem Abschnitt werden zusätzlich die *Commit-Ereignisse* eingeführt, welche besagen, daß dynamische Aktionen nicht mehr abgebrochen werden können. Im Fall von Kooperation kann die Entscheidung für eine unwiderrufliche Beendigung oft nicht von einer einzelnen dynamischen Aktion autonom getroffen werden, sondern mehrere dynamische Aktionen entscheiden gemeinsam darüber, daß sie gemeinsam einen sogenannten *Gruppencommit* durchführen. Um dieses Verhalten im formalen Modell widerzuspiegeln, muß sich ein Commit-Ereignis auf mehrere dynamische Aktionen beziehen können, im Gegensatz zu den beiden anderen bekannten Terminierungsereignissen, die immer genau eine dynamische Aktion betrafen. Jedes Commit-Ereignis bezieht sich also auf eine Menge von dynamischen Aktionen. Zusätzlich muß sichergestellt werden, daß eine dynamische Aktion nie von mehreren Commit-Ereignissen betroffen ist.

Zu diesem Zweck wurde in [Mock 1994] die hier wiedergegebene Definition einer Zerlegung gemacht.

Definition 15 Zerlegung

Sei M eine Menge. $I = \{I_1, \ldots, I_k\}, k \in \mathbb{N}$ heißt (endliche) Zerlegung von M, wenn gilt:

1. $I \subset POT(M)$
2. $\bigcup_{i \in \{1,\ldots,k\}} I_i = M$
3. $\forall\, i, j \in \{1, \ldots, k\}, i \neq j :\ I_i \cap I_j = \emptyset$

das heißt, I besteht aus Teilmengen von M (1), die M ganz überdecken (2) und paarweise disjunkt sind (3). ∎

Betrachtet man jetzt für M die Menge der Indizes der dynamischen Aktionen eines Schedules, läßt sich eine Zerlegung finden, die genau die Indizes derjenigen dynamischen Aktionen zusammenfaßt, für die ein gemeinsames Commit-Ereignis stattfindet. Dabei ist allerdings zu beachten, daß für abgebrochene dynamische Aktionen kein Commit-Ereignis im Schedule vorkommt und deshalb zur Spezifizierung der Commit-Ereignisse nur eine Teilmenge der Zerlegung benötigt wird.

Mit Hilfe der Zerlegung wird in [Mock 1994], Definition 42 für flache dynamische Aktionen der vollständige Schedule mit Commit-Ereignissen definiert. Zur Modellierung der Schachtelung muß dieser Schedule analog zu Definition 5 um die Starte-Sohn-Ereignisse erweitert werden. Dies führt zu folgender Definition:

Definition 16 Vollständiger Schedule mit Starte-Sohn- und Commit-Ereignissen

Sei $A = \{A_1, A_2, \ldots, A_n\},\ n \in \mathbb{N}$ eine Mengen von geschachtelten dynamischen Aktionen. Ein vollständiger Schedule mit Starte-Sohn- und Commit-Ereignissen für A ist ein Paar $S = (OP^S, \prec^S)$ mit:

0. Sei $I = \{I_1, \ldots, I_k\},\ k \in \mathbb{N}$ Teilmenge einer Zerlegung von $\{1, \ldots, n\}$ und sei $C^S = \{c_{I_1}, \ldots c_{I_k}\}$ die Menge der Commit-Ereignisse von S.
1. $ST^S = \{s_i(j) \mid i, j \leq n \wedge i \neq j\}$ der Menge der Starte-Sohn-Ereignisse von S, für die gilt $\forall\, i, j, k \leq n :\ s_j(i) \in ST^S \wedge s_k(i) \in ST^S \Longrightarrow j = k$.
   Außerdem wird für jedes $i \leq n$ definiert: $ST^{(i)} = \{s_i(j) \mid s_i(j) \in ST^S\}$
2. $OP^S = \bigcup_{i=1}^{n}(OP^{(i)} \cup T^{(i)}) \ \cup\ ST^S \ \cup\ C^S$ der Menge der Schritte von S.
3. $\prec^S\ \subseteq\ OP^S \times OP^S$ der (Schedule-) Vorrangrelation von S mit den folgenden Eigenschaften:
   (a) $\prec^S$ ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch
   (b) stehen zwei Objektoperationen von S, p und q miteinander in Konflikt, so gilt: $p \prec^{(i)} q \vee q \prec^{(i)} p$
   (c) $OP^S \supseteq \bigcup_{i=1}^{n} \prec^{(i)}$, d.h. die (Schedule-) Vorrangrelation ist konform mit allen Intra-Aktions-Vorrangrelationen der dynamischen Aktionen von S.
   (d) $\prec^S \supseteq \bigcup_{i=1}^{n} \prec^{(i)}$
   d.h. die (Schedule-) Vorrangrelation ist konform mit allen Intra-Aktions-Vorrangrelationen der dynamischen Aktionen von S.
   (e) $s_j(i) \in OP^S \Longrightarrow (\forall\ o \in T^{(j)} :\ s_j(i) \prec^S o)$
   d.h. dynamische Aktionen dürfen nur vor ihrer Terminierung Söhne starten.
   (f) $s_j(i) \in OP^S \Longrightarrow (\forall\ o \in OP^{(i)} \cup ST^{(i)} \cup T^{(i)} :\ s_j(i) \prec^S o)$
   d.h. Sohnaktionen dürfen erst nach ihrem Start-Ereignis Operationen ausführen.
4. Es gilt: $\forall\ i \in \{1, \ldots, n\} :\ \forall\ I' \in I :\ \ i \in I' \ \Longrightarrow\ (a_i \notin OP^S \ \wedge\ t_i \prec^S c_{I'})$

∎

Diese Definition unterscheidet sich von Definition 5 des vollständigen Schedules mit Starte-Sohn-Ereignissen in zwei Punkten: Punkt 0 führt die Commit-Ereignisse ein, die mit Hilfe einer Teilmenge einer Zerlegung der Menge der Indizes der dynamischen Aktionen des Schedules definiert werden. Diese Ereignisse gehören auch zur Menge der Schritte des Schedules. In Punkt 4 werden Bedingungen für das Auftreten von Commit-Ereignissen aufgestellt: Wenn solch ein Ereignis für eine dynamische Aktion A_i auftritt, darf A_i nicht abbrechen und vor dem Commit-Ereignis muß bereits das Complete-Ereignis t_i im vollständigen Schedule aufgetreten sein.

Ein **Schedule mit Starte-Sohn- und Commit-Ereignissen** wird entsprechend zu Definition 5 als Präfix des vollständigen Schedules mit Starte-Sohn- und Commit-Ereignissen definiert.

In der Definition des Schedules wurden keine Aussagen darüber gemacht, welche Commit-Ereignisse für ein Schedule zulässig sind, also welche Zerlegung gewählt werden darf. Zwei Kriterien haben darauf Einfluß: die im nächsten Abschnitt beschriebene Commit-Korrektheit und die Baumstruktur der dynamischen Aktionen.

Die Semantik der Baumstruktur von dynamischen Aktionen besagt, daß alle dynamischen Aktionen eines Aktionsbaums, die nicht abgebrochen wurden, gemeinsam in den Zustand *committed* übergehen. Im formalen Modell muß also sichergestellt werden, daß ein Commit-Ereignis sich immer auf alle nicht-abgebrochenen dynamischen Aktionen eines oder mehrerer Aktionsbäume bezieht. Zur Modellierung dieser Eigenschaft muß zuerst die Definition von $\mathrm{BAUM}_S(A_i)$ derart erweitert werden, daß nur die am Ende des Schedules S nicht-abgebrochenen dynamischen Aktionen berücksichtigt werden.

Definition 17 Nicht-abgebrochener Aktionsbaum

$$\mathrm{BAUM}_S^{NA}(A_i) := \{A_k |\ A_k \in \mathrm{BAUM}_S(A_i) \ \wedge\ a_k \notin OP^S\}$$

∎

Mit Hilfe dieser Menge läßt sich das Kriterium definieren, das die Menge der Commit-Ereignisse einhalten muß, damit sie zur Baumstruktur der dynamischen Aktionen von S konform ist.

Definition 18 Zur Baumstruktur der dynamischen Aktionen konforme Menge von Commit-Ereignissen

Sei $C^S = \{c_{I_1}, \dots c_{I_k}\}$ eine Menge der Commit-Ereignisse des Schedules S der dynamischen Aktionen $A = \{A_1, \dots, A_n\}$.

C^S ist zur Baumstruktur der dynamischen Aktionen in S konform, wenn für jedes Ereignis $c_{I_j} \in C^S$ gilt:

$$\forall\ k \in I_j,\ m \in \{1, \dots, n\} :\ A_m \in \mathrm{BAUM}_S^{NA}(A_k) \Longrightarrow m \in I_j$$

∎

4.3.3 Commit-Korrektheit

Die Eigenschaft der *Commit-Korrektheit* besagt, daß keine Fehler auftreten können, die die Effekte einer dynamischen Aktion im Zustand *Committed* zurücksetzen würden, das heißt, daß eine dynamische Aktion im Zustand *Committed* nicht mehr abgebrochen werden kann.

Diese Eigenschaft wird in der Definition 16 des Schedules durch Satz 4 formal ausgedrückt, indem ein Abbruch-Ereignis für eine dynamische Aktion im Schedule verboten wird, sofern sich für diese ein Commit-Ereignis im Schedule befindet. Allerdings wird dort diese Eigenschaft nur gefordert, es werden aber keine Aussagen darüber gemacht, unter welchen Bedingungen diese Eigenschaft überhaupt einzuhalten ist. Eine handhabbare Definition der Commit-Korrektheit wird in [Mock 1994], Definition 45 gegeben, die darauf beruht, Commit-Ereignisse als *zulässig* zu definieren, wenn alle Schedules, die Erweiterungen des Schedules bis zum Commit-Ereignis sind, abbruch-korrekt sind. Die Commit-Korrektheit fordert die Zulässigkeit aller Commit-Ereignisse.

In [Mock 1994] wird anschließend in Satz 47 ein Kriterium für die Commit-Korrektheit von flachen dynamischen Aktionen entwickelt. Dieses Kriterium basiert wie die Kriterien zur Abbruch-Korrektheit auf der „Liest-von"-Relation, welche auch bei der Betrachtung der Commit-Korrektheit nicht mehr ausreicht, da auch hier Schachtelung berücksichtigt werden muß. Im Gegensatz zur Abbruch-Korrektheit, bei der der Abbruch einer dynamischen Aktion auch den Abbruch des bei ihr beginnenden Teilbaums zur Folge hat, muß bei der Commit-Korrektheit der komplette Aktionsbaum betrachtet werden. Nach Definition 18 darf ein Commit-Ereignis nur dann eine dynamische Aktion beinhalten, wenn es auch alle anderen dynamischen Aktionen dieses Teilbaums beinhaltet. Aus diesem Grund wird jetzt auf Basis der „Liest-von"-Relation die Commit-Abhängigkeit definiert:

Definition 19 Commit-Abhängigkeit

A_j ist direkt commit-abhängig von A_i $\left(A_i \underset{CAB_S}{\longrightarrow} A_j\right)$

$$:\Longleftrightarrow \begin{cases} A_i \underset{LV_S}{\longrightarrow} A_j & \text{oder} \\ A_j \in \mathrm{BAUM}_S^{NA}(A_i) \end{cases}$$

Die Commit-Abhängigkeit ist die transitive Hülle der direkten Commit-Abhängigkeit:

A_j ist commit-abhängig von A_i $\left(A_i \underset{CAB_S^*}{\longrightarrow} A_j\right)$

$$:\Longleftrightarrow \exists\ A_{k_1}, \ldots, A_{k_m}, m \in \mathbb{N} : A_i = A_{k_1} \wedge A_j = A_{k_m} \ \wedge\ A_{k_1} \underset{CAB_S}{\longrightarrow} A_{k_2} \underset{CAB_S}{\longrightarrow} \cdots \underset{CAB_S}{\longrightarrow} A_{k_m}$$

∎

Auch die Abhängigkeitsmenge und Nachfolgermenge müssen erweitert werden, so daß sie die Commit-Abhängigkeit berücksichtigen. Dies führt zu folgenden Definitionen:

Definition 20 Commit-Abhängigkeitsmenge

$$CAB_S(A_i) := \{A_j \in A |\ A_j \underset{CAB_S^*}{\longrightarrow} A_i\}$$

∎

Definition 21 Commit-Nachfolgermenge

$$CNF_S(A_i) := \{A_j \in A |\ A_i \underset{CAB_S^*}{\longrightarrow} A_j\}$$

∎

Analog zu den Betrachtungen bezüglich der Abbruch-Korrektheit ersetzen diese drei Definitionen die Definitionen der „Liest-von"-Relation und der Abbhängigkeits- bzw. Nachfolgermenge in [Mock 1994]. Mit diesen neuen Definitionen sind alle Nachweise und Eigenschaften aus den Abschnitten 3.4 „Sicherung von dynamischen Aktionen" und 3.5 „Konsolidierungsgruppen" weiter gültig und werden deshalb hier nicht ausformuliert.

4.4 Geschachtelte dynamische Aktionen mit Objektkontrollen

In diesem Abschnitt wird die Definition der geschachtelten dynamischen Aktionen um den Mechanismus der *Objektkontrolle* erweitert. Ferner werden auch die schon in [Mock 1994] eingeführten

Validierungsereignisse aufgenommen. Damit läßt sich das Verhalten der dynamischen Aktionen im Integrationskonzept umfassend modellieren. Anschließend wird anhand dieser Modellierung nachgewiesen, daß das Integrationskonzept das Konsistenzkriterium der *dynamischen kooperativen Serialisierbarkeit* erfüllt.

4.4.1 Modellierung

Zuerst werden zwei neue Operationen eingeführt, die jeweils den Beginn und das Ende einer Objektkontrolle markieren, die eine dynamische Aktion auf ein bestimmtes Objekt hält. Diese neuen Operationen werden anschließend in eine erweiterte Definition der dynamischen Aktion aufgenommen und in (Vorrang-) Relation zu den anderen möglichen Operationen gesetzt, die eine dynamische Aktion durchführen kann. Auf dieser Basis wird ein vollständiger Schedule definiert, der das Verhalten mehrerer nebenläufiger dynamischer Aktionen unter Berücksichtigung der Objektkontrolle beschreibt.

Definition 22 (Objekt-) Operation

Sei $OP := (\{R, W\} \times D) \ \cup \ (\{B, E\} \times D \times \mathbb{N})$ die Menge der Operationen auf der Menge der Objekte D.

Dabei heißen die Operationen R, W, B, E entsprechend „Read“, „Write“, „Beginn-einer-Objektkontrolle“ und „Ende-einer-Objektkontrolle“ und werden auch als $R(x), W(x), B_n(x), E_n(x)$ mit $x \in D, n \in \mathbb{N}$ notiert.

Ein Element aus OP heißt Objektoperation. ∎

Diese Erweiterung der Menge der Operation zieht nach sich, daß auch die Definition des Konflikts erweitert werden muß. Beginn und Ende einer Objektkontrolle stehen wie die Schreib-Operation mit anderen Operationen in Konflikt. Also ist ein Konflikt jetzt wie folgt definiert:

Definition 23 Konflikt

Zwei Operationen stehen in Konflikt, wenn sie dasselbe Objekt betreffen und mindestens eine Operation keine „Read“-Operation ist. ∎

Diese Operationen werden jetzt in die Definition der dynamischen Aktion aufgenommen, welches zu folgender Definition führt:

Definition 24 Dynamische Aktion mit Objektkontrollen

Eine dynamische Aktion mit Objektkontrollen ist ein Paar $A_i = (OP^{(i)}, \prec^{(i)}), i \in \mathbb{N}$ mit

1. $OP^{(i)} = \{o_{i1}, o_{i2}, \ldots, o_{im_i}\}, m_i \in \mathbb{N}$ der Menge der Objektoperationen von A_i
2. $B_{ij} \in OP^{(i)} \implies E_{ij} \in OP^{(i)} \ \wedge \ E_{ij} \in OP^{(i)} \implies B_{ij} \in OP^{(i)}$

   d. h. Beginn- und Ende-einer-Objektkontrolle treten für ein Objekt immer paarweise auf.
3. $\prec^{(i)} \ \subseteq \ (OP^{(i)} \cup \{a_i, t_i\}) \ \times \ (OP^{(i)} \cup \{a_i, t_i\})$ der (Intra-Aktions-) Vorrangrelation von A_i mit den folgenden Eigenschaften:

   (a) $\prec^{(i)}$ ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch

(b) Stehen zwei Objektoperationen von A_i, o_{ik} und o_{il} miteinander in Konflikt, so gilt: $o_{ik} \prec^{(i)} o_{il} \vee o_{il} \prec^{(i)} o_{ik}$

(c) Es gilt:
$$\begin{aligned} & (\forall\, o \in OP^{(i)} :\ o \prec^{(i)} a_i \ \wedge\ \neg(o \prec^{(i)} t_i \ \vee\ t_i \prec^{(i)} o)) \\ \vee\ & (\forall\, o \in OP^{(i)} :\ o \prec^{(i)} t_i \ \wedge\ \neg(o \prec^{(i)} a_i \ \vee\ a_i \prec^{(i)} o)) \\ \vee\ & (\forall\, o \in OP^{(i)} :\ o \prec^{(i)} t_i \ \wedge\ t_i \prec^{(i)} a_i) \end{aligned}$$

(d) Es gilt:
$$\begin{aligned} \forall\, B_{ij}(x), E_{ij}(x) \in OP^{(i)} :\ & B_{ij}(x) \prec^{(i)} E_{ij}(x) \\ \wedge\ & \neg(\exists\, B_{ik}(x) \in OP^{(i)} :\ B_{ij}(x) \prec^{(i)} B_{ik}(x) \prec^{(i)} E_{ij}(x)) \\ \wedge\ & \neg(\exists\, E_{ik}(x) \in OP^{(i)} :\ B_{ij}(x) \prec^{(i)} E_{ik}(x) \prec^{(i)} E_{ij}(x)) \end{aligned}$$

∎

Im Vergleich zur ursprünglichen Definition 24 sind in dieser Definition neben der erweiterten Operationenmenge $OP^{(i)}$ die Punkte 2 und 3.d) hinzugekommen: Punkt 2 fordert, daß zusammengehörende Beginn- und Ende-einer-Objektkontrolle-Operationen immer gemeinsam in der Menge $OP^{(i)}$ auftreten müssen. Punkt 3.d) legt die Reihenfolge der Objektkontrollen-Ereignisse fest: Der Beginn muß immer vor dem Ende der zugehörigen Objektkontrolle liegen und zwischen ihnen dürfen keine weiteren Objektkontrollen-Operationen für dasselbe Objekt liegen, wodurch eine Schachtelung dieser Operationen ausgeschlossen wird.

Mit den so definierten dynamischen Aktionen wird jetzt ein vollständiger Schedule definiert, indem festgelegt wird, wie Objektkontrollen die Objektzugriffe beeinflussen. Außerdem werden analog zur Definition 77 aus [Mock 1994] die Validierungsereignisse eingeführt. Diese Ereignisse werden später dazu benötigt, die dynamische kooperative Serialisierbarkeit durchzusetzen und gelten ähnlich den Commit-Ereignissen in Definition 16 immer für eine Menge dynamischer Aktionen.

Definition 25 Vollständiger Schedule mit Starte-Sohn- und Validierungsereignissen und Objektkontrollen

Sei $A = \{A_1, A_2, \ldots, A_n\}, n \in \mathbb{N}$ eine Menge von geschachtelten dynamischen Aktionen mit Objektkontrollen. Ein vollständiger Schedule mit Starte-Sohn- und Validierungsereignissen und Objektkontrollen für A ist ein Paar $S = (OP^S, \prec^S)$ mit:

0. Sei $I = \{I_1, \ldots, I_k\}$, $k \in \mathbb{N}$ Teilmenge einer Zerlegung von $\{1, \ldots, n\}$ und sei $V^S = \{v_{I_1}, \ldots v_{I_k}\}$ die Menge der Validierungsereignisse von S.

1. $ST^S = \{s_i(j) \mid i, j \leq n \wedge i \neq j\}$ der Menge der Starte-Sohn-Ereignisse von S, für die gilt $\forall\, i, j, k \leq n :\ s_j(i) \in ST^S \wedge s_k(i) \in ST^S \Longrightarrow j = k$.

   Außerdem wird für jedes $i \leq n$ definiert: $ST^{(i)} = \{s_i(j) \mid s_i(j) \in ST^S\}$

2. $OP^S = \bigcup_{i=1}^{n}(OP^{(i)} \cup T^{(i)})\ \cup\ ST^S\ \cup\ V^S$ der Menge der Schritte von S.

3. $\prec^S\ \subseteq\ OP^S \times OP^S$ der (Schedule-) Vorrangrelation von S mit den folgenden Eigenschaften:

   (a) $\prec^S$ ist eine Halbordnung, also irreflexiv, transitiv und asymmetrisch

   (b) stehen zwei Objektoperationen von S, p und q miteinander in Konflikt, so gilt: $p \prec^{(i)} q \vee q \prec^{(i)} p$

   (c) $\prec^S\ \supseteq \bigcup_{i=1}^{n} \prec^{(i)}$
   d.h. die (Schedule-) Vorrangrelation ist konform mit allen Intra-Aktions-Vorrangrelationen der dynamischen Aktionen von S.

   (d) $s_j(i) \in OP^S \Longrightarrow (\forall\, o \in T^{(j)} :\ s_j(i) \prec^S o)$
   d.h. dynamische Aktionen dürfen nur vor ihrer Terminierung Söhne starten.

(e) $s_j(i) \in OP^S \Longrightarrow (\forall\, o \in OP^{(i)} \cup ST^{(i)} \cup T^{(i)} :\ s_j(i) \prec^S o)$
d.h. Sohnaktionen dürfen erst nach ihrem Start-Ereignis Operationen ausführen.

(f) Es gilt: $\forall\, o_{ij}(x), B_{kl}(x), E_{kl}(x) \in OP^S :\ o_{ij}(x) \prec^S B_{kl}(x) \ \vee\ E_{kl}(x) \prec^S o_{ij}(x) \ \vee\ A_i \in \text{TEILBAUM}_S(A_k)$

4. Für die Validierungsereignisse gilt:

(a) $\forall\, i \in \{1, \ldots, n\} :\ a_i \notin OP^S \implies \bigcup_{l \in \{1,\ldots,k\}} I_l$

(b) $\forall\, i \in \{1, \ldots, n\} :\ \forall\, I' \in I :\ \ i \in I' \implies \forall\, o_i \in OP^{(i)} :\ o_i \prec^S v_{I'} \ \vee\ v_{I'} \prec^S o_i$

∎

In diesem vollständigen Schedule werden in Punkt 0 der Definition die Validierungsereignisse eingeführt. Jedes dieser Ereignisse betrifft dabei eine Menge von dynamischen Aktionen, allerdings darf es für jede dynamische Aktion höchstens ein Validierungsereignis geben. In Punkt 4 werden dann weitere Anforderungen an die Validierungereignisse gestellt: Punkt 4.a) besagt, daß es für jede nicht-abgebrochene dynamische Aktion ein Validierungsereignis geben muß und Punkt 4.b) fordert, daß dieses Ereignis mit allen anderen Operationen der betroffenen dynamischen Aktion in Vorrangrelation stehen muß.

Den Einfluß der Objektkontrolle auf das Verhalten der dynamischen Aktionen wird in Punkt 3.f) formalisiert: Für alle Operationen und Beginn-/Ende-einer-Objektkontrolle-Paare auf demselben Objekt x wird dazu gefordert, daß die Objektoperation vor dem Beginn oder nach dem Ende der Objektkontrolle stattfindet, oder daß die Objektoperation von einer dynamischen Aktion aus dem Teilbaum ausgeführt wird, der bei der dynamischen Aktion beginnt, deren Beginn- und Ende-der-Objektkontrolle-Operationen gerade betrachtet werden. Hierbei ist zum einen zu beachten, daß nach Definition 24 immer Paare der Beginn- und Ende-einer-Objektkontrolle-Operationen existieren müssen. Zum anderen steuert eine Objektkontrolle auch den möglichen Beginn einer anderen Objektkontrolle auf demselben Objekt, da die Objektkontrollenoperationen auch zur Menge OP^S gehören. Solange eine Objektkontrolle gehalten wird, also in der Zeit zwischen der Beginn- und der Ende-Operation, kann deshalb nur die haltende dynamische Aktion selbst oder einer ihrer Nachfahren auf dem betroffenen Objekt Operationen durchführen. Weitere Objektkontrollen können von Nachfahren begonnen werden. Ein Beginn einer weiteren Objektkontrolle durch die haltende dynamische Aktion wird bereits durch Definition 24 verboten.

4.4.2 Dynamische kooperative Serialisierbarkeit

Das Konsistenzkriterium der *dynamischen kooperativen Serialisierbarkeit* wurde in [Mock 1994], Kapitel 3.6 formal eingeführt und in Definition 67 definiert.

Das Kriterium gilt dann als erfüllt, wenn die dynamischen Aktionen jeder sogenannten *Kooperationsklasse* seitens der Applikationssemantik miteinander kooperieren dürfen, also *Kooperationsmengen* ([Mock 1994], Definition 61) bilden. Die in [Mock 1994], Definition 66 eingeführten Kooperationsklassen fassen alle committeten dynamischen Aktionen zusammen, die untereinander in zyklischer Konflikt-Abhängigkeit stehen. Nach [Mock 1994], Lemma 65 sind diese Kooperationsklassen auch Äquivalenzklassen.

In [Mock 1994], Abschnitt 3.7 wurde dann gezeigt, daß Schedules flacher dynamischer Aktionen mit Validierungsereignissen das Kriterium der dynamischen kooperativen Serialisierbarkeit erfüllen, sofern sie *validierungskorrekt* ([Mock 1994], Definition 82) und commit-korrekt sind und nur validierte dynamische Aktionen committed werden ([Mock 1994], Satz 80).

Hier soll nun gezeigt werden, daß das Konsistenzkriterium der dynamischen kooperativen Serialisierbarkeit auch für die hier definierten Schedules geschachtelter dynamischer Aktionen mit Starte-Sohn- und Validierungsereignissen und Objektkontrollen gilt. Zu diesem Zweck wird bewiesen, daß aus diesen Schedules die in [Mock 1994] definierten „vollständigen Schedules mit Validierungsereignissen“ entstehen, wenn mit geeigneter Projektion die Objektkontrolle-Operationen und die Starte-Sohn-Ereignisse entfernt werden. Für solche Schedules wurde bereits in [Mock 1994] die Erfüllung des Konsistenzkriteriums gezeigt.

Satz 6

Sei $S = (OP^S, \prec^S)$ ein vollständiger Schedule mit Starte-Sohn- und Validierungsereignissen und Objektkontrollen für die Menge von geschachtelten dynamischen Aktionen mit Objektkontrollen $A = \{A_1, A_2, \ldots, A_n\}, n \in \mathbb{N}$.

Die Projektion $PR_{o \notin \{B_k(x), E_k(x), s_i(j),\ \ k \in \mathbb{N}, i,j \leq n\}}(S) = (OP^{S'}, \prec^{S'})$ ist ein vollständiger Schedule mit Validierungsereignissen nach [Mock 1994], Definition 77 für die Menge von dynamischen Aktionen $A' = \{A'_1, A'_2, \ldots, A'_n\}, n \in \mathbb{N}$, nach [Mock 1994], Definition 4. ∎

Beweis:

Es wird gezeigt, daß die Definitionen 25 und 24 nach der Projektion den genannten Definitionen aus [Mock 1994] entsprechen.

In Definition 25 des Schedules führt der Wegfall der Objektkontrolle-Operationen und Starte-Sohn-Ereignisse zu folgenden Änderungen:

- Die Definition der Menge der Starte-Sohn-Ereignisse ST^S in Punkt 1 wird nicht mehr benötigt.
- Die Definition der Menge der Schritte OP^S in Punkt 2 kann um die Teilmenge ST^S verkürzt werden, da diese leer ist.
- Die Punkte 3.d) und 3.e), die Aussagen über die Vorrangrelation bezüglich der Starte-Sohn-Ereignisse sind, sind immer erfüllt, da es keine Starte-Sohn-Ereignisse mehr in der Menge der Schritte gibt.
- Ebenso ist Punkt 3.f) immer erfüllt, da die Menge der Schritte auch keine Objektkontrollen-Operationen enthält.

Die so resultierende Definition entspricht genau der Definition 77 aus [Mock 1994] für einen vollständigen Schedule mit Validierungsereignissen. Auf die Definition der dynamischen Aktion hat die Projektion folgende Auswirkungen:

- Punkt 2 von Definition 24 ist immer erfüllt, da Beginn- und Ende-einer-Objektkontrollen nicht mehr als Operation auftreten.
- Aus diesem Grund ist auch Punkt 3.d) immer erfüllt.

Wie beim Schedule führt die Projektion auch bei der dynamischen Aktion zu der in [Mock 1994] gemachten Definition. ∎

Damit wurde bewiesen, daß die angegebene Projektion den in [Mock 1994] definierten Schedule mit Validierungsereignissen erzeugt, für den dort die Erfüllung des Konsistenzkriteriums der dynamischen kooperativen Serialisierbarkeit gezeigt wurde. Dadurch ist die dynamische kooperative

Serialisierbarkeit für Schedules geschachtelter dynamischer Aktionen mit Starte-Sohn- und Validierungsereignissen und Objektkontrollen dann nachgewiesen, wenn das für das Konsistenzkriterium relevante Verhalten der dynamischen Aktionen durch die Projektion nicht beeinflußt wird:

Die dynamische kooperative Serialisierbarkeit basiert darauf, Aussagen über erlaubte Konflikte zwischen dynamischen Aktionen zu machen. Diese Konflikte beruhen auf Schreib- und Leseoperationen unterschiedlicher dynamischer Aktionen auf demselben Objekt. Diese beiden Objektoperationen sind von der Projektion nicht betroffen, so daß Ursprungs- und projizierter Schedule dieselben Schreib- und Leseoperationen haben und auch dieselben Konflikte vorhanden sind.

Die zusätzlichen Operationen zur Objektkontrolle erlauben keinen tatsächlichen Zugriff auf die Objekte, sondern beeinflussen nur mögliche andere Objektoperationen. Daher sollen sie auch nicht zur Überprüfung der Serialisierbarkeit herangezogen werden. Dieses wird durch die Projektion erreicht, da sie die Objektkontrollen-Operationen entfernt und diese so keine Konflikte mehr verursachen können. Da die Schachtelung keine Einfluß auf die Nebenläufigkeit sondern nur auf die Fehlerbehandlung hat, brauchen auch die Starte-Sohn-Ereignisse beim Nachweis der dynamischen kooperativen Serialisierbarkeit nicht betrachtet werden.

Damit ist gezeigt, daß der Nachweis der dynamischen kooperativen Serialisierbarkeit auch für das Integrationskonzept gültig ist. Integrationskonzept und das Konzept der dynamischen Aktionen für kooperative Anwendungen sind also bezüglich ihres Konsistenzkriteriums verträglich.

Kapitel 5

Algorithmen und Datenstrukturen

5.1 Einleitung

In diesem Kapitel werden Algorithmen und Datenstrukturen beschrieben, mit denen ein Modul zur Nebenläufigkeitskontrolle das in Kapitel 3 eingeführte kooperative Sperrverfahren realisieren kann. Ein solches Modul läßt sich in die zwei weitgehend unabhängigen Komponenten der Sperren- und der Abhängigkeitsverwaltung aufteilen, die die Anforderung und Freigabe der Sperren steuern, beziehungsweise die Protokolle zur Bildung von Kooperationsgruppen und zur Validierung durchführen. Die Nebenläufigkeitskontrolle wird durch eine Reihe von Operationen realisiert, die das Modul zur Verfügung stellt und vom Applikationsprogramm aufgerufen werden müssen. Anhand einer Schnittstellen-Beschreibung werden diese Operationen eingeführt und anschließend werden die Algorithmen beschrieben, die diese Operationen realisieren. Dabei liegt der Schwerpunkt auf den Operationen der Sperrenverwaltung, da die Abhängigkeitsverwaltung durch ein generisches Graphmodul (siehe [Theisohn 1996]) realisiert werden kann.

Da die Nebenläufigkeitskontrolle mit beliebigen Objekttypen zusammenarbeiten soll, werden die Objekte unabhängig von ihrer Speicherung im Objektspeicher durch einen eindeutigen Identifikator repräsentiert. Dieser Identifikator kann, solange die Eindeutigkeit gewährleistet ist, eine beliebige Form haben und wird der Nebenläufigkeitskontrolle durch die Applikation zur Verfügung gestellt. Die Applikation kann hierfür zum Beispiel den Namen nehmen, den das Objekt auch in seiner Datenbank hat, ist aber in keiner Weise darauf festgelegt.

Auch für die dynamischen Aktionen und Kooperationsgruppen werden systemweit eindeutige Identifikatoren benötigt, wobei auch ihr Aufbau für die Algorithmen unerheblich ist, solange eine Möglichkeit besteht, eventuell existierende Verwandtschaftsbeziehungen von zwei dynamischen Aktionen zu bestimmen. In Kapitel 6, Abschnitt 6.1 ist eine mögliche Darstellung dieser Identifikatoren angegeben, wie sie von der Aktionsverwaltung (siehe zum Beispiel [Baer 1992]) für dynamische Aktionen verwendet wird.

Die Unabhängigkeit der Nebenläufigkeitskontrolle vom Objektspeicher führt dazu, daß die eigentlichen Objektzugriffe nicht durch die Nebenläufigkeitskontrolle überwacht werden können. Deshalb wird zwingend vorausgesetzt, daß die dynamischen Aktionen bezüglich ihres Sperrenverhaltens wohlgeformt sind: Eine dynamische Aktion darf eine Schreib- bzw. eine Leseoperation auf einem Objekt nur dann ausführen, wenn sie für dieses Objekt sowohl die entsprechende kooperative Sperre als auch die entsprechende Benutzungssperre hält.

5.2 Aufbau der Nebenläufigkeitskontrolle

Die für die Nebenläufigkeitskontrolle benötigten Operationen gliedern sich in zwei Gruppen: Operationen, die für die Anforderung und Freigabe von Sperren benötigt werden, und solche, die die Bildung von Kooperationsgruppen und die Validierung von dynamischen Aktionen durchführen. Der Unterschied zwischen diesen beiden Gruppen besteht darin, daß die Operationen zur Sperrenverwaltung nur Informationen benötigen, die direkt mit dem betroffenen Objekt zusammenhängen, während die anderen Operationen unabhängig von den Objekten sind, sondern ihre Entscheidungen aufgrund der vorher entstanden Konfliktabhängigkeiten fällen. Diese Unterscheidung in den Anforderungen spiegelt sich in der Struktur des Moduls zur Nebenläufigkeitskontrolle wieder: Es besteht aus den zwei Komponenten zur *Sperren-* und zur *Abhängigkeitsverwaltung*. Auch wenn diese beiden Komponenten weitgehend unabhängig sind, gibt es doch einige Funktionen, mit denen beide Komponenten Informationen austauschen. Zum Beispiel erkennt die Sperrenverwaltung neu entstehende Konflikt-Abhängigkeiten und muß diese an die Abhängigkeitsverwaltung weitermelden. Umgekehrt muß die Abhängigkeitsverwaltung beispielsweise auch die Sperrenverwaltung über eine erfolgte Validierung informieren. Eine genaue Beschreibung der Schnittstelle folgt später.

Neben den beiden beschriebenen Gruppen von Operationen existieren noch zwei weitere Operationen, die die Nebenläufigkeitskontrolle über die Beendigung (Complete) und den Abbruch einer oder mehrerer dynamischer Aktionen informieren. Diese Information ist für beide Komponenten gleichermaßen von Bedeutung. Bei der Diskussion einer verteilten Implementierung in Kapitel 6, Abschnitt 6.3 wird deutlich gemacht werden, daß es sinnvoll ist, daß diese beiden Operationen zur Abhängigkeitsverwaltung gehören, welche die Information dann weiter an die Sperrenverwaltung leitet. Die Nebenläufigkeitskontrolle wird lediglich über Beendigung und Abbruch von dynamischen Aktionen informiert und hat keine Entscheidungsmöglichkeit über Annahme oder Ablehnung dieser Ereignisse. Hierfür ist in der Aktionsverwaltung ein gesondertes Modul zur Fehlerbehandlung zuständig, dessen Einbindung auch in Abschnitt 6.3 beschrieben ist.

Die Struktur des Moduls zur Nebenläufigkeitskontrolle wird in Abbildung 5.1 zusammengefaßt. Die Beschreibung der Schnittstellen erfolgt anschließend im folgenden Abschnitt 5.3. Die Operationen der Nebenläufigkeitskontrolle werden dabei entweder direkt aus dem Applikationsprogramm aufgerufen, wenn zum Beispiel seitens der Applikation nur die Nebenläufigkeitskontrolle aber keine Fehlerbehandlung benötigt wird, oder das Modul zur Nebenläufigkeitskontrolle ist in eine umfassende Aktionsverwaltung eingebunden. In diesem Fall erfolgen die Aufrufe nicht direkt vom Applikationsprogramm sondern durch die vorgeschaltete Aktionsverwaltung. Beide Szenarien sind im Kapitel 6 beschrieben.

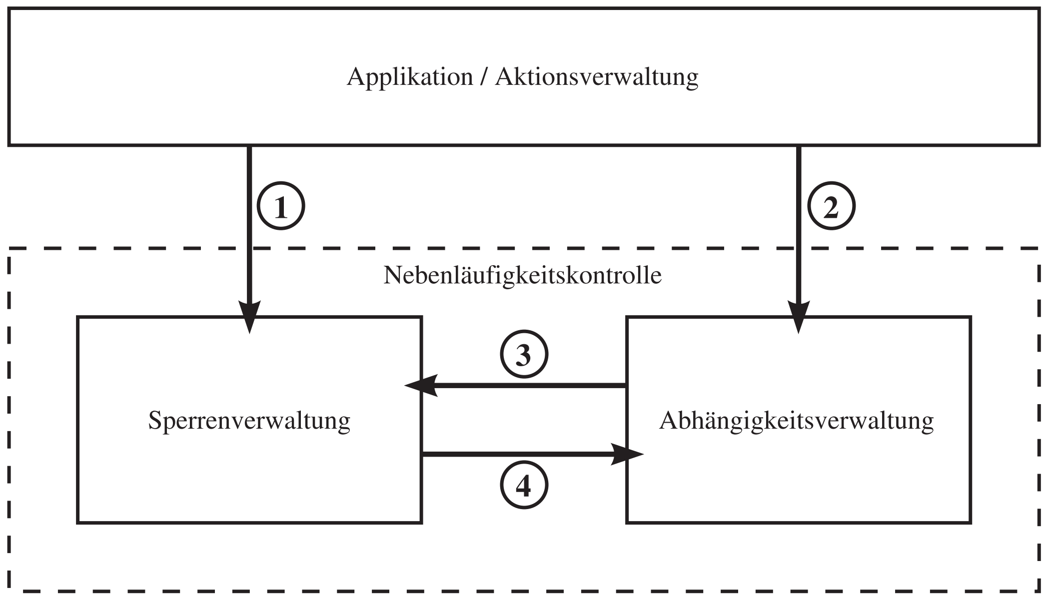
5.3 Funktionalität und Schnittstelle der Nebenläufigkeitskontrolle

Das Modul zur Nebenläufigkeitskontrolle stellt der Applikation seine Funktionalität durch die in Abbildung 5.1 aufgeführte Schnittstelle zur Verfügung. Die einzelnen Operationsaufrufe werden jetzt genauer beschrieben, wobei die Schnittstellenbeschreibung an die Programmiersprache „C“ bzw. „C++“ angelehnt ist.

5.3.1 Kooperative Sperren

Kooperative Sperren werden durch die Operationen `setlock` angefordert und `releaselock` freigegeben, deren Schnittstelle in Abbildung 5.2 dargestellt ist.

Durch den ersten Parameter `ReqAID` („Requesting Action-ID“) identifiziert sich die anfordernde



Operationen an den Schnittstellen zwischen den Komponenten:

(1)	(2)	(3)	(4)
`setlock`	`joinkog`	`CC_joinkog`	`add_CC_dependency`
`releaselock`	`validate`	`CC_validate`	
`setgrouplock`	`complete`	`CC_complete`	
`releasegrouplock`	`abort`	`CC_abort`	
`uselock`			
`freelock`			

Abbildung 5.1: Schematischer Aufbau der Nebenläufigkeitskontrolle

```
int setlock (
    AID_TYPE*   ReqAID,        // anfordernde dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des angeforderten Objektes
    RW_MODE     ReqMode,       // READ | WRITE
    int         nReqPhase,     // 1 | 2 | 3
    KOG_TYPE*   ReqKOG         // Kooperationsgruppe der anfordernden dyn. Aktion
)

int releaselock (
    AID_TYPE*   ReqAID,        // freigebende dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des freizugebenden Objektes
    RW_MODE     ReqMode        // READ | WRITE
)
```

Abbildung 5.2: Schnittstelle für kooperative Sperren beziehungsweise freigebende dynamische Aktion. Da genaue Codierung des Identifikators für die Algorithmen unerheblich ist, wird der Datentyp hier generisch mit `AID_TYPE` bezeichnet. Das betroffene Objekt wird durch den Parameter `ReqResID` („Requested Resource-ID“) spezifiziert. Der Parameter `ReqMode` kann entweder den Wert `READ` oder den Wert `WRITE` enthalten und gibt an, in welchem Modus die Sperre erworben werden soll. Dieser Parameter muß auch bei der Freigabe der Sperre angegeben werden, da es erlaubt sein soll, daß eine dynamische Aktion gleichzeitig eine Schreib- und eine Lesesperre für ein Objekt besitzt, und deshalb spezifiziert werden muß, welche dieser beiden Sperren freigegeben werden soll. So ist es möglich, daß eine dynamische Aktion eine Lesesperre auf eine Schreibsperre „erweitert“, ohne zwischenzeitlich die Lesesperre abgeben zu müssen, wodurch eine kurze Zeitspanne entstehen würde, in der die dynamische Aktion gar keine Sperre auf dieses Objekt besitzt.

Bei der Anforderung einer kooperativen Sperre müssen noch zwei weitere Parameter angegeben werden: Der Parameter `nReqPhase` gibt dabei an, in welcher Phase sich die anfordernde dynamische Aktion gerade befindet. Dabei bedeutet 1 „vor der Bildung der Kooperationsgruppe“, 2 „vor der Validierung“ und 3 „nach der Validierung“. Der Parameter `ReqKOG` hat nur in den Phasen 2 und 3 eine Bedeutung.

Die Operation `setlock` testet mit den angegebenen Parametern, ob nach den in Abschnitt 3.4.3 definierten Regeln zur Sperrenvergabe die kooperative Sperre gewährt werden kann. Ist dies der Fall, wird die dynamische Aktion als Halter der angeforderten kooperativen Sperre vermerkt und die Operation gibt den Wert `OK` zurück. Entsprechend wird bei der Operation `releaselock` nach erfolgreichem Test der Regeln die kooperative Sperre als freigegeben markiert. Falls die Regeln eine Sperrenvergabe oder -freigabe verbieten, oder falls eine kooperative Sperre freigegeben werden soll, die die dynamische Aktion gar nicht besitzt, wird der Wert `FAILED` zurückgegeben.

5.3.2 Kooperative Sperren für Kooperationsgruppen

Mit zwei weiteren Operationen können dynamische Aktionen stellvertretend für ihre Kooperationsgruppe kooperative Sperren anfordern. Die in Abbildung 5.3 angegebene Schnittstelle hat dabei einen ähnlichen Aufbau, wie die Schnittstelle für „normale“ kooperative Sperren, allerdings hat sie eine etwas andere Bedeutung.

Diese beiden Operationen können nur ausgeführt werden, wenn sich die aufrufende dynamische Aktion in der zweiten oder dritten Phase befindet, sie also zu einer Kooperationsgruppe gehört.

Bei einem Aufruf der Operation `setgrouplock` werden die Regeln zur Sperrenvergabe nicht bezüglich der angegebenen dynamischen Aktion überprüft, sondern bezüglich der Kooperationsgruppen-Aktion, die die Rolle des Vaters aller Toplevel-Aktionen der Kooperationsgruppe besitzt. Der Aufruf der Operation `setgrouplock` entspricht damit der Operation `setlock (ReqKOG, ReqResID, ReqMode, nReqPhase, ReqKOG)`, wobei angenommen wird, daß der Identifikator der Kooperationsgruppe in das Schema der Identifikatoren von dynamischen Aktionen paßt. Durch diese Rückführung auf `setlock` wird für die Anforderung einer kooperativen Sperren durch eine Kooperationsgruppe kein zusätzlicher Algorithmus benötigt.

Die anfordernde dynamische Aktion wird zusätzlich übergeben, da mehrere dynamische Aktionen gleichzeitig stellvertretend für die Kooperationsgruppe kooperative Sperren anfordern dürfen. Da jede stellvertretend angeforderte kooperative Sperre wieder von derselben dynamischen Aktion freigegeben werden muß, werden die anfordernden dynamischen Aktionen für jedes gesperrte Objekt einer Kooperationsgruppen-Aktion in einer separaten Liste vermerkt. Die Operation `setlock` wird nicht mehr aufgerufen, wenn in der Liste schon dynamische Aktionen enthalten sind, da sie dann bereits von einer anderen dynamische Aktion für dieselbe Kooperationsgruppe aufgerufen wurde.

```
int setgrouplock (
    AID_TYPE*   ReqAID,        // stellvertretend anfordernde dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des angeforderten Objektes
    RW_MODE     ReqMode,       // READ | WRITE
    int         nReqPhase,     // 2 | 3
    KOG_TYPE*   ReqKOG         // anfordernde Kooperationsgruppe
)

int releasegrouplock (
    AID_TYPE*   ReqAID,        // stellvertretend freigebende dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des freizugebenden Objektes
    RW_MODE     ReqMode,       // READ | WRITE
    KOG_TYPE*   ReqKOG         // freigebende Kooperationsgruppe
)
```

Abbildung 5.3: Schnittstelle für kooperative Sperren für Kooperationsgruppen

Die Operation `releasegrouplock` streicht die dynamische Aktion wieder aus dieser Liste und führt die Operation `releaselock (ReqKOG, ReqResID, ReqMode)` aus, wenn dies die letzte dynamische Aktion der Liste war.

Beide Operationen geben analog zu `setlock` und `releaselock` einen der Werte `OK` oder `FAILED` zurück.

5.3.3 Benutzungssperren

Zur Anforderung und Freigabe von Benutzungssperren stellt die Sperrenverwaltung die in Abbildung 5.4 dargestellten Operationen `uselock` und `freelock` zur Verfügung.

```
int uselock (
    AID_TYPE*   ReqAID,        // anfordernde dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des angeforderten Objektes
    RW_MODE     ReqMode        // READ | WRITE
)

int freelock (
    AID_TYPE*   ReqAID,        // freigebende dynamische Aktion
    RESID_TYPE* ReqResID,      // Name des freizugebenden Objektes
    RW_MODE     ReqMode        // READ | WRITE
)
```

Abbildung 5.4: Schnittstelle für Benutzungssperren

Die Parameter und möglichen Rückgabewerte dieser beiden Operationen haben dieselben Bedeutung wie bei den Operationen `setlock` und `releaselock`. Der Operation `uselock` müssen im

Gegensatz zu `setlock` die Phase der dynamischen Aktion und die Kooperationsgruppe nicht übergeben werden, da die Vergaberegeln von diesen Informationen unabhängig sind.

Benutzungssperren können nicht für Kooperationsgruppen angefordert werden, da diese nicht als dynamische Aktionen existieren und deshalb nicht auf Objekte zugreifen können. Daher existieren keine Operationen `usegrouplock` oder `freegrouplock`.

5.3.4 Bildung von Kooperationsgruppen und Validierung

Die Bildung von Kooperationsgruppen und die Validierung betreffen die Komponenten der Abhängigkeitsverwaltung. Die Schnittstelle dieser beiden Operationen ist in Abbildung 5.5 angegeben.

```
int joinkog (
    AID_TYPE** AIDs,     // Liste beitretender dynamischer Aktionen
    KOG_TYPE*  KOGID     // Kooperationsgruppe
)

int validate (
    KOG_TYPE*  KOGID     // zu validierende Kooperationsgruppe
)
```

Abbildung 5.5: Schnittstelle für Bildung von Kooperationsgruppen und Validierung

Mit der Operation `joinkog` wechseln eine oder mehrere dynamische Aktionen von der ersten in die zweite Phase und treten dabei der angegebenen Kooperationsgruppe bei. Existiert diese Kooperationsgruppe noch nicht, wird sie durch diese Operation automatisch erstellt. Die Operation ist nur dann erfolgreich und gibt den Wert `OK` zurück, wenn die neu zu erzeugende Kooperationsgruppe die in Abschnitt 2.6.3 geforderten Eigenschaften erfüllt. Bei einem Beitritt zu einer existierenden Kooperationsgruppe dürfen diese Eigenschaften durch die neuen dynamischen Aktionen nicht verletzt werden. Schlägt diese Überprüfung fehl, wird der Wert `FAILED` zurückgegeben.

Die Operation `validate` validiert alle zu der angegebenen Kooperationsgruppe gehörenden dynamischen Aktionen, die dadurch in die dritte Phase wechseln. Anhand des Abhängigkeitsgraphen wird auch hier vorher überprüft, ob die für eine Validierung notwendigen Eigenschaften erfüllt sind. Nur dann erfolgt die Validierung und es wird der Wert `OK` zurückgegeben. Ansonsten bricht die Operation mit dem Wert `FAILED` ab.

Beide Operationen informieren die Sperrenverwaltung bei Erfolg durch die Funktionen `CC_joinkog` und `CC_validate` über die betroffenen dynamischen Aktionen. Die Sperrenverwaltung vermerkt den Phasenwechsel in ihren internen Listen.

5.3.5 Beendigung und Abbruch von dynamischen Aktionen

Die in Abbildung 5.6 dargestellten Operationen `complete` und `abort` informieren die Nebenläufigkeitskontrolle über die Beendigung oder den Abbruch von dynamischen Aktionen.

Durch diese beiden Operationen wird zunächst die Komponente der Abhängigkeitsverwaltung über den Abbruch der in der Liste angegebenen dynamischen Aktionen informiert, die diese Information

```
void complete (
    AID_TYPE** AIDs     // Liste der beendeten dynamischen Aktionen
)

void abort (
    AID_TYPE** AIDs     // Liste der abgebrochenen dynamischen Aktionen
)
```

Abbildung 5.6: Schnittstelle für Beendigung und Abbruch von dynamischen Aktionen

über die Funktionen `CC_complete` und `CC_abort` weiter an die Sperrenverwaltung leitet. Die beiden Komponenten können die betroffenen dynamischen Aktionen bei Bedarf aus ihren internen Datenstrukturen löschen. Sie haben aber keine Möglichkeit, der Operation zu widersprechen, weshalb die Operationen den Typ `void` haben.

5.4 Realisierung der Sperrenverwaltung

Nachdem im vorherigen Abschnitt die Schnittstelle der Nebenläufigkeitskontrolle beschrieben wurde, werden im folgenden die Algorithmen vorgestellt, die die entsprechenden Operationen realisieren. Dieser Abschnitt behandelt dabei die Funktionalität der Sperrenverwaltung, während im nächsten Abschnitt kurz auf die Funktionalität der Abhängigkeitsverwaltung eingegangen wird.

Bevor im Anschluß die Algorithmen zur Anforderung und Freigabe der Sperren beschrieben werden, erfolgt zunächst eine Einführung in die Datenstrukturen, mit denen die kooperativen und die Benutzungssperren verwaltet werden. Diese Datenstrukturen sollen es ermöglichen, effizient zu entscheiden, ob die Regeln zur Sperrenvergabe aus Abschnitt 3.4.3 eine Vergabe oder Freigabe erlauben.

5.4.1 Datenstrukturen zur Verwaltung von kooperativen Sperren

Damit über die Vergabe von kooperativen Sperren entschieden werden kann, muß die Sperrenverwaltung Informationen darüber besitzen, welche dynamischen Aktionen für das betreffende Objekt zur Zeit der Anforderung kooperative Sperren besitzen. Zusätzlich muß sie aber auch über dynamische Aktionen informiert sein, die in der Vergangenheit kooperativen Sperren für das Objekt besaßen, da die Vergabeentscheidung abhängig vom letzten Schreiber oder letzten Halter einer kooperativen Sperren sein kann.

Daher liegt es nahe, für jedes Objekt eine Liste zu führen, in der alle kooperativen Sperren vermerkt sind, die für dieses Objekt angefordert wurden. Diese Liste ist nach der Reihenfolge der Vergabe der kooperativen Sperren geordnet. Regel 1 zur Sperrenvergabe erzwingt dabei, daß kooperative Schreibsperren nur seriell vergeben werden dürfen, sofern die anfordernden dynamischen Aktionen nicht in einem Vorfahre-Nachfahre-Verhältnis stehen. Außerdem dürfen mehrere kooperative Lesesperren zu einem Zeitpunkt von unterschiedlichen dynamischen Aktionen gehalten werden. Die Sperrenliste ist also keine totale Ordnung, sondern innerhalb der Liste gibt es Mengen von untereinander ungeordneten dynamischen Aktionen: Aufeinanderfolgende kooperative Lesesperren und aufeinanderfolgende kooperative Sperrenanforderungen von dynamischen Aktionen, die in Vorfahre-

Nachfahre-Beziehung stehen, sind jeweils untereinander nicht geordnet. Die Sperrenliste hat also einen Aufbau, wie er beispielhaft in Abbildung 5.7 dargestellt ist. Dabei ist zu beachten, daß $A_{m,1}$ ein Nachfahre von A_m sein muß. Zwei Mengen von dynamischen Aktionen sind bei der Überprüfung der Sperrenregeln wichtig: Die Menge der *letzten Sperren* (auch Menge der *letzten Halter* genannt) und die Menge der *letzten Schreiber*, wobei beide Mengen einelementig und auch identisch sein können.

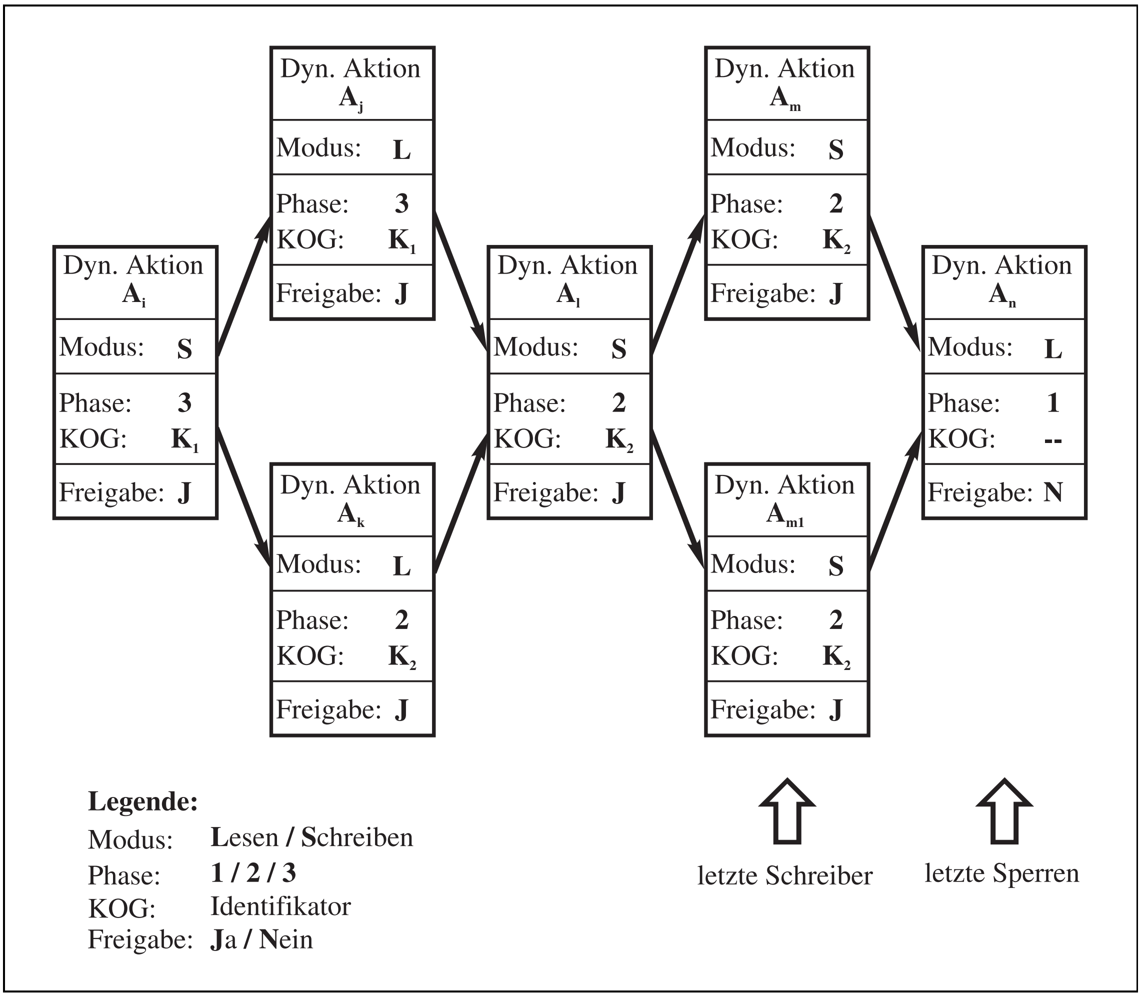


Abbildung 5.7: Sperrenliste eines Objekts

Die einzelnen Einträge der Sperrenliste besitzen jeweils folgende Informationen: die Identifikatoren der anfordernden dynamischen Aktion und (falls bereits vorhanden) ihrer Kooperationsgruppe, die Phase, in der sich die dynamische Aktion befindet, den Modus, in dem die kooperative Sperre angefordert wurde und eine Markierung, ob die Sperre bereits wieder freigegeben wurde. Abbildung 5.8 zeigt einen solchen Sperreneintrag im Pseudo-Code, wie er in den Algorithmen verwendet wird.

Die Regeln zur Sperrenvergabe setzen durch, daß nur Einträge aus der Menge der letzten Sperren noch nicht freigegeben sind. Eine einmal gesetzte Freigabe-Markierung kann nicht wieder entfernt werden. Wenn eine dynamische Aktion nach eine Freigabe erneut eine Sperre auf dieses Objekt erwirbt, wird statt dessen ein neuer Eintrag in der Sperrenliste angelegt. Sind alle Einträge aus der Menge der letzten Sperren als „freigegeben" markiert, dann ist das Objekt nicht gesperrt, ansonsten ist es im Schreibmodus gesperrt, wenn die letzten Sperreneinträge Schreibsperren, und im Lesemodus gesperrt, wenn sie Lesesperren sind.

Außerdem lassen sich Aussagen über die Phase der Sperreneinträge machen: Nach Regel 2 zur

```
typedef struct LOCK_ENTRY
{
    ACTION_TYPE AID;          // dargestellte dynamische Aktion
    RW_MODE     Mode;         // READ | WRITE
    int         nPhase;       // 1 | 2 | 3
    BOOL        bReleased;    // TRUE | FALSE
    KOG_TYPE    KOG;          // Kooperationsgruppe (in Phase 2 + 3)
    ...                       // Verkettung der Sperreneinträge
} LOCK_ENTRY_TYPE;
```

Abbildung 5.8: Datenstruktur eines Sperreneintrags

Sperrenvergabe und den Protokollen zur Bildung von Kooperationsgruppen und zur Validierung müssen alle Sperreneinträge vor einem Eintrag einer dynamischen Aktion in der Phase 2 oder 3 entweder von einer dynamischen Aktion derselben Kooperationsgruppen sein oder die dynamischen Aktionen diese Einträge müssen bereits selbst in der Phase 3, also validiert sein.

Obwohl zur Überprüfung der Sperrenregeln aus der Sperrenliste nur die Mengen der letzten Sperren und letzten Schreiber benötigt werden, reicht es nicht aus, nur diese beiden Mengen zu speichern. Falls beispielsweise die dynamischen Aktionen aus der aktuellen Menge der letzten Schreiber abgebrochen und damit aus der Sperrenliste gelöscht werden, müssen ältere Einträge der Sperrenliste zur Verfügung stehen, um erneut die Menge der letzten Schreiber zu bilden. Um diese Menge bei Abbrüchen immer bilden zu können, müßte die Sperrenliste alle Einträge bis einschließlich zur jüngsten schreibenden dynamischen Aktion enthalten, die bereits im Zustand *committed* ist. Bei vielen längeren dynamischen Aktionen könnten diese Listen dadurch sehr lang werden. Um dieses zu vermeiden, besteht ein praktikabler Ansatz darin, die Sperreneinträge bis vor den letzten validierten Schreiber aus der Liste zu entfernen. Sollte dieser Eintrag aufgrund eines Abbruchs gelöscht und die gespeicherte Sperrenliste dadurch leer werden, ist bei der nächsten Sperrenanforderung folgendes zu beachten:

- Ist die anfordernde dynamische Aktion in der ersten Phase, kann ihr die Sperre unabhängig von den letzten Schreibern und letzten Sperren vergeben werden, so daß die Liste gar nicht betrachtet werden muß.

- In der zweiten Phase darf eine Sperre nur dann vergeben werden, wenn abhängig vom Modus die letzten Schreiber oder die letzten Halter in derselben Kooperationsgruppe liegen oder bereits validiert sind. Alle aus der Liste bereits entfernten Einträge müssen aber bereits validiert gewesen sein, da es die abgebrochene dynamische Aktion auch war und dieses wie eben gezeigt impliziert, daß sie nur validierte Vorgänger hat. Bei einer leeren Sperrenliste kann einer dynamischen Aktion in der zweiten Phase die angeforderte Sperre also immer gewährt werden.

- Wenn sich die anfordernde dynamische Aktion bereits in der dritten Phase befindet, darf ihr bei einer leeren Sperrenliste die angeforderte Sperre nicht mehr gewährt werden. Da das Konsistenzkriterium in jedem Fall garantiert werden muß, und dieses fordert, daß bei einer Sperrenanforderung in der dritte Phase der letzte Halter in derselben Kooperationsgruppe liegen muß, dieses aber nicht mehr verifiziert werden kann, wird sicherheitshalber die Sperre nicht gewährt. Falls der letzte bereits entfernte Halter zur selben Kooperationsgruppe wie die anfordernde dynamische Aktion gehört, wird eine Sperre verweigert, die eigentlich hätte gewährt werden können. Da die Sperrenliste allerdings nicht bei jedem Zugriff überarbeitet wird, sondern nur bei einer Validierung, ist der Fall einer fälscherweise nicht gewährten Sperre aber sehr unwahrscheinlich.

Die Algorithmen basieren darauf, daß alle Sperreneinträge aktuell sind und nicht den Zustand der dynamischen Aktion beim Anfordern der Sperre beibehalten. Dies bedeutet, daß die Sperreneinträge bei der Bildung einer Kooperationsgruppe oder bei der Validierung aktualisiert werden müssen. Um dieses effizient durchführen zu können, sind alle Sperreneinträge noch auf eine weitere Art verkettet. Sie bilden für jede dynamische Aktion eine Liste aller von ihr erzeugten Sperreneinträge. So muß zur Aktualisierung aller Sperreneinträge einer dynamischen Aktion nur diese Liste durchlaufen werden. Auch bei Abbruch oder Complete einer dynamischen Aktion lassen sich so alle noch gehaltenen kooperativen Sperren finden.

5.4.2 Datenstrukturen zur Verwaltung von Benutzungssperren

Die Datenstrukturen zur Verwaltung von Benutzungssperren sind deutlich einfacher, als die zur Verwaltung von kooperativen Sperren, da hier zur Überprüfung der Regeln zur Sperrenvergabe nur die aktuell gehaltenen Sperren betrachtet werden müssen. Außerdem sind Informationen über Phase und Kooperationsgruppe für Benutzungssperren nicht relevant.

Um die Entscheidung über die Vergabe oder Freigabe von Benutzungssperren treffen zu können, wird für jedes Objekt eine Menge von in Abbildung 5.9 dargestellten Sperreneinträgen geführt. Zur effizienten Bearbeitung von Complete und Abbruch einer dynamischen Aktion existiert auch auf diesen Sperreneinträgen eine Verkettung, die jeder dynamischen Aktion alle von ihr erzeugten Sperreneinträge zuordnet.

```
typedef struct USELOCK_ENTRY
{
    ACTION_TYPE AID;          // dargestellte dynamische Aktion
    RW_MODE     Mode;         // READ | WRITE
    ...                       // Verkettung der Sperreneinträge
} USELOCK_ENTRY_TYPE;
```

Abbildung 5.9: Datenstruktur eines Benutzungssperreneintrags

5.4.3 Algorithmus zur Anforderung von kooperativen Sperren

Kooperative Sperren werden mit der Operation `setlock` angefordert. In diesem Abschnitt wird vorgestellt, wie diese Operation algorithmisch realisiert wird.

Zur algorithmischen Formulierung der Operation wird ein Pseudo-Code verwendet, der an die Programmiersprache „C“ bzw. „C++“ angelehnt ist. Zusätzlich zu den dort vorhandenen Sprachkonstrukten ist insbesondere die Verwendung von Quantoren und Mengenoperatoren möglich. Außerdem werden in Algorithmen einige Funktionen benutzt, deren Wirkung aufgrund ihres Namens selbsterklärend sein sollte. So sind beispielsweise auf Sperrenlisten die Funktionen `LastEntries` und `LastWriters` definiert, die jeweils die Menge der letzten Sperren bzw. der letzten Schreiber zurückgeben. Im Unterschied zur „Ist-Nachfahre-von“-Relation aus Kapitel 4 ist die Funktion `IsAncestor` auch reflexiv.

Abbildung 5.10 zeigt in Pseudo-Code den Algorithmus der Operation `setlock`.

Die Operation `setlock` gliedert sich in zwei Abschnitte: Im ersten Abschnitt wird anhand der Sperrenliste überprüft, ob die angeforderte kooperative Sperre gewährt werden kann. Ist dieses der Fall, wird im zweiten Abschnitt ein neuer Eintrag in der Sperrenliste angelegt und mögliche Konfliktabhängigkeiten werden der Abhängigkeitsverwaltung mitgeteilt.

Für die Vergabe von kooperativen Sperren sind die Regeln 1 und 2 aus Abschnitt 3.4.3 relevant und werden im ersten Abschnitt des Algorithmus überprüft.

Unter dem Punkt „*Teste Schreib-/Lesekompatibilität ...*“ wird Regel 1 überprüft, in der kontrolliert wird, ob es einen nicht freigegebenen Eintrag in der Sperrenliste gibt, der mit der angeforderten Sperre nicht verträglich ist. Wie bereits dargelegt, können nicht freigegebene Einträge nur in der Menge der letzten Sperren auftreten, so daß es ausreichend ist, statt der gesamten Sperrenliste nur die Menge `LastEntries` zu betrachten.

Die Regel 2 zur Sperrenvergabe wird durch den Punkt „*Teste Kooperationsmodus ...*“ überprüft. Es werden dabei abhängig von der Phase der anfordernden dynamischen Aktion und dem gewünschten Sperrenmodus die in Regel 2 gestellten Anforderungen an die Mengen der letzten Schreiber und letzten Sperren überprüft.

Wenn die Sperre gewährt und ein neuer Eintrag in die Sperrenliste aufgenommen wird, müssen noch neu entstandene Konfliktabhängigkeiten der Abhängigkeitsverwaltung mitgeteilt werden. Da die Sperrenverwaltung nicht entscheiden kann, ob eine Abhängigkeit bereits vorhanden oder neu hinzugekommen ist, werden alle direkten Konfliktabhängigkeiten zur dynamischen Aktion, deren Eintrag zur Sperrenliste hinzukommt, der Abhängigkeitsverwaltung mitgeteilt. Die neue dynamische Aktion hängt bei einer Schreibsperre von allen letzten Sperreneinträgen und bei einer Lesesperre von allen letzten Schreibern ab.

5.4.4 Algorithmus zur Freigabe von kooperativen Sperren

Kooperative Sperren werden über die in Abbildung 5.11 gezeigte Operation `releaselock` freigegeben.

Diese Operation ist wie schon die Operation `setlock` in zwei Abschnitte aufgeteilt: Nach erfolgreicher Überprüfung der Sperrenregel 3 aus Abschnitt 3.4.3 im ersten Abschnitt wird im zweiten Abschnitt der entsprechende Sperreneintrag freigegeben.

Zur Freigabe einer kooperativen Sperre müssen lediglich zwei Kriterien überprüft werden: In der Sperrenliste muß ein Eintrag für freizugebene Sperre existieren und die freigebende dynamische Aktion darf keine entsprechende Benutzungssperre für das Objekt halten.

5.4.5 Algorithmus zur Anforderung von Benutzungssperren

In Abbildung 5.12 ist die Operation `uselock` dargestellt, mit der Benutzungssperren angefordert werden.

Um die für die Vergabe von Benutzungssperren maßgebliche Regel 4 aus Abschnitt 3.4.3 zu überprüfen, muß zuerst sichergestellt werden, daß die anfordernde dynamische Aktion bereits eine passende kooperative Sperre für dieses Objekt besitzt. Wenn eine Lese-Benutzungssperre angefordert wird, ist jede kooperative Sperre passend; eine angeforderte Schreib-Benutzungssperre setzt eine kooperative Schreibsperre voraus. Anschließend muß überprüft werden, ob die angeforderte Benutzungssperre mit einer anderen bereits gehaltenen Benutzungssperre in Konflikt steht. Dies ist

```
int setlock (AID_TYPE* ReqAID, RESID_TYPE* ReqResID, RW_MODE ReqMode,
             int nReqPhase, KOG_TYPE* ReqKOG)
{
   LOCK_ENTRY_TYPE *LockList = GetLockList (ReqResID);

   // Teste Schreib-/Lesekompatibilität ...
   if (∃ entry ∈ LastEntries(LockList): (entry.bReleased == FALSE &&
                                         !IsAncestor (entry.AID, ReqAID) &&
                                         (entry.Mode == WRITE ||
                                          ReqMode == WRITE)))
      return (FAILED);

   // Teste Kooperationsmodus ...
   switch (nReqPhase) {
   1: break;    // keine weitere Überprüfung notwendig
   2: if (ReqMode == READ &&
          !(∃ entry ∈ LastWriters(LockList): (entry.KOG == ReqKOG ||
                                               entry.nPhase == 3)))
         return (FAILED);
      if (ReqMode == WRITE &&
          !(∀ entry ∈ LastEntries(LockList): (entry.KOG == ReqKOG ||
                                               entry.nPhase == 3)))
         return (FAILED);
      break;
   3: if (ReqMode == READ &&
          !(∃ entry ∈ LastWriters(LockList): entry.KOG == ReqKOG ||
            ∃ entry ∈ LastEntries(LockList): entry.KOG == ReqKOG))
         return (FAILED);
      if (ReqMode == WRITE &&
          !(∀ entry ∈ LastEntries(LockList): entry.KOG == ReqKOG))
         return (FAILED);
      break;
   }
   // Alle Regeln erfüllt ⟹ Sperre kann vergeben werden!

   // Melde Konfliktabhängigkeiten an die Abhängigkeitsverwaltung ...
   if (ReqMode == READ)
      ∀ entry ∈ LastWriters(LockList): add_CC_dependency (entry.AID, ReqAID);
   if (ReqMode == WRITE)
      ∀ entry ∈ LastEntries(LockList): add_CC_dependency (entry.AID, ReqAID);

   // Erzeuge neuen Sperreneintrag und hänge diesen an die Sperrenliste an ...
   LOCK_ENTRY_TYPE *pNewEntry = new (LOCK_ENTRY_TYPE);
   ...    // Speichere Daten von angefordertem Sperreneintrag in *pNewEntry.
   append (pNewEntry, LockList);

   return (OK);
}
```

Abbildung 5.10: Algorithmus der Operation `setlock`

```
int releaselock (AID_TYPE* ReqAID, RESID_TYPE* ReqResID, RW_MODE ReqMode)
{
   LOCK_ENTRY_TYPE    *LockList    = GetLockList (ReqResID);
   USELOCK_ENTRY_TYPE *UselockList = GetUselockList (ReqResID);

   // Wird freizugebende kooperative Sperre zur Zeit gehalten?
   if (!(∃ entry ∈ LastEntries(LockList): (entry.bReleased == FALSE &&
                                            entry.AID       == ReqAID &&
                                            entry.Mode      == ReqMode)))
      return (FAILED);
   else
      EntryToRemove = entry;

   // Wird zur Zeit auch eine Benutzungssperre gehalten?
   if (∃ entry ∈ UselockList: (entry.AID   == ReqAID &&
                               entry.Mode  == ReqMode))
      return (FAILED);

   // Alle Regeln erfüllt ⟹ Sperre kann freigegeben werden!

   // Entferne Sperreneintrag aus Sperrenliste ...
   remove (EntryToRemove, LockList);

   return (OK);
}
```

Abbildung 5.11: Algorithmus der Operation `releaselock`

```
int uselock (AID_TYPE* ReqAID, RESID_TYPE* ReqResID, RW_MODE ReqMode)
{
   LOCK_ENTRY_TYPE    *LockList    = GetLockList (ReqResID);
   USELOCK_ENTRY_TYPE *UselockList = GetUselockList (ReqResID);

   // Wird eine passende kooperative Sperre zur Zeit gehalten?
   if (!(∃ entry ∈ LastEntries(LockList): (entry.bReleased == FALSE &&
                                            entry.AID       == ReqAID &&
                                            (ReqMode == WRITE ⟹
                                             entry.Mode == WRITE))))
      return (FAILED);

   // Teste Schreib-/Lesekompatibilität ...
   if (∃ entry ∈ UselockList: (entry.AID != ReqAID &&
                               (entry.Mode == WRITE || ReqMode == WRITE)))
      return (FAILED);

   // Alle Regeln erfüllt ⟹ Sperre kann vergeben werden!

   // Erzeuge neuen Sperreneintrag und hänge diesen an die Sperrenliste an ...
   USELOCK_ENTRY_TYPE *pNewEntry = new (USELOCK_ENTRY_TYPE);
   ...   // Speichere Daten von angefordertem Sperreneintrag in *pNewEntry.
   append (pNewEntry, UselockList);

   return (OK);
}
```

Abbildung 5.12: Algorithmus der Operation `uselock` dann der Fall, wenn eine Schreib-Benutzungssperre angefordert oder bereits gehalten wird, wobei allerdings zugelassen werden soll, daß eine dynamische Aktion gleichzeitig sowohl eine Schreib- als auch eine Lese-Benutzungssperre hält.

Da Benutzungssperren nicht dazu dienen, die Serialisierbarkeit durchzusetzen, müssen bei `uselock` keine Abhängigkeiten erkannt und an die Abhängigkeitsverwaltung gemeldet werden.

5.4.6 Algorithmus zur Freigabe von Benutzungssperren

Die in Abbildung 5.13 gezeigte Operation `freelock` zur Freigabe von Benutzungssperren realisiert die Sperrenregel 5 aus Abschnitt 3.4.3. Dort werden keine weitergehenden Bedingungen für die Freigabe einer gehaltenen Benutzungssperre gestellt, so daß lediglich überprüft werden muß, daß die dynamische Aktion tatsächlich die freizugebende Benutzungssperre besitzt.

```
int freelock (AID_TYPE* ReqAID, RESID_TYPE* ReqResID, RW_MODE ReqMode)
{
   USELOCK_ENTRY_TYPE *UselockList = GetUselockList (ReqResID);

   // Wird freizugebende Benutzungssperre zur Zeit gehalten?
   if (!(∃ entry ∈ UselockList: (entry.AID   == ReqAID &&
                                 entry.Mode  == ReqMode)))
      return (FAILED);
   else
      EntryToRemove = entry;

   // Alle Regeln erfüllt ⟹ Sperre kann freigegeben werden!

   // Entferne Sperreneintrag aus Sperrenliste ...
   remove (EntryToRemove, UselockList);

   return (OK);
}
```

Abbildung 5.13: Algorithmus der Operation `freelock`

5.4.7 Zusammenwirken mit der Abhängigkeitsverwaltung

Neben den bisher vorgestellten Operationen zur Sperrenvergabe enthält die Sperrenverwaltung auch die in Abbildung 5.14 aufgeführten Funktionen, mit denen die Abhängigkeitsverwaltung die Sperrenverwaltung über Zustandsänderungen von dynamischen Aktionen informiert. Alle Funktionen dienen dabei ausschließlich der Information der Sperrenverwaltung und sind daher vom Typ `void`. Sie übergeben jeweils im ersten Parameter eine Liste der betroffenen dynamischen Aktionen.

Die Funktion `CC_joinkog` setzt in allen Sperreneinträgen, die zu einer angegebenen dynamischen Aktionen gehören, den Phaseneintrag auf den Wert 2 und speichert den Identifikator der Kooperationsgruppe.

Durch die Funktion `CC_validate` werden die Phaseneinträge aller betroffenen Sperreneinträge auf den Wert 3 gesetzt. Außerdem werden die Sperrenlisten überarbeitet, um eventuell überflüssig gewordene Sperreneinträge zu entfernen.

```
void CC_joinkog (AID_TYPE** AIDs, KOG_TYPE* KOG)
void CC_validate (AID_TYPE** AIDs)
void CC_complete (AID_TYPE** AIDs)
void CC_abort (AID_TYPE** AIDs)
```

Abbildung 5.14: Schnittstelle der Sperrenverwaltung zur Abhängigkeitsverwaltung

Mit der Funktion `CC_complete` wird die Sperrenverwaltung über die Beendigung der angegebenen dynamischen Aktionen informiert. Diese gibt daraufhin alle eventuell noch gehaltenen kooperativen Sperren und Benutzungssperren frei.

Schließlich informiert die Funktion `CC_abort` die Sperrenverwaltung über den Abbruch von dynamischen Aktionen. Alle von ihnen gehaltenen Sperren werden freigegeben und alle zu diesen dynamischen Aktionen gehörende Sperreneinträge werden aus den Sperrenlisten entfernt.

5.5 Realisierung der Abhängigkeitsverwaltung

Die Abhängigkeitsverwaltung ist durch ein generisches Graphmodul implementiert, welches in [Theisohn 1996] entwickelt und beschrieben wurde. Dieses Graphmodul kann beliebige Graphen verwalten und frei definierbare Abfragen konsistent durchführen. Diesem Modul teilt die Sperrenverwaltung über die Funktion `add_CC_dependency`, deren Schnittstelle in Abbildung 5.15 gezeigt ist, entstandene Konfliktabhängigkeiten mit. Das Graphmodul erstellt dynamisch aus den so mitgeteilten Abhängigkeiten den Abhängigkeitsgraphen.

```
void add_CC_dependeny (AID_TYPE* FromAID, AID_TYPE* ToAID)
```

Abbildung 5.15: Schnittstelle der Abhängigkeitsverwaltung zur Sperrenverwaltung

Der Abhängigkeitsgraph wird bei der Durchführung der Operationen `joinkog` und `validate` ausgewertet, um die Zulässigkeit der Operationen zu überprüfen und um die Menge der dynamischen Aktionen zu ermitteln, die über die Funktionen `CC_joinkog` und `CC_validate` der Sperrenverwaltung mitgeteilt wird.

Die Kriterien zur Bildung von Kooperationsgruppen und die Validierungsalgorithmen sind bereits in [Mock 1994] umfassend beschrieben, so daß sie hier nicht erneut diskutiert werden müssen. Bei der Bildung von Kooperationsgruppen muß zusätzlich lediglich sichergestellt werden, daß Aktionsbäume immer komplett zu einer Kooperationsgruppe gehören. Dies wird dadurch realisiert, daß bei Beitritt einer dynamischen Aktion zu einer Kooperationsgruppe automatisch alle anderen dynamischen Aktionen desselben Aktionsbaums in diese Kooperationsgruppe aufgenommen werden. Wenn eine dynamische Aktion, die bereits zu einer Kooperationsgruppe gehört, eine Sohn startet, wird dieser Sohn automatisch derselben Kooperationsgruppe zugeordnet.

Die Validierung hat die Aufgabe, daß die dynamischen Aktionen zum einen ihr Einverständnis zu ihren Kooperationspartnern geben, die aufgrund der Konfliktabhängigkeiten entstanden sind, und zum anderen erklären, daß sie auf keine zusätzlichen Objekte mehr zugreifen, damit keine weiteren Konfliktabhängigkeiten entstehen. Da im hier beschriebenen kooperativen Sperrverfahren die Kooperationspartner bereits durch die Bildung der Kooperationsgruppen feststehen und die Sperrenregeln die Beibehaltung der Kooperationsgruppen-Eigenschaften garantieren, kann der Algorithmus zur Validierung in der Form vereinfacht werden, daß lediglich das Einverständnis aller dynamischen Aktionen einer Kooperationsgruppe über den *Zeitpunkt* der Validierung eingeholt werden muß. Dieses läßt sich über den in [Theisohn 1996] eingeführten Zwischenzustand *validierungsbereit* realisieren, in den alle dynamischen Aktionen einer Kooperationsgruppe gewechselt sein müssen, bevor sie gemeinsam validieren. Dieser Zustandswechsel kann implizit durch die Operation `validate` geschehen, indem dadurch der Wechsel in den Zustand *validierungsbereit* vollzogen und anschließend das Validierungsprotokoll gestartet wird. Das Validierungsprotokoll ist erst dann erfolgreich, wenn es vom letzten Mitglied einer Kooperationsgruppe gestartet wird. Der Zustand *validierungsbereit* wird nur in der Abhängigkeitsverwaltung gespeichert, hat aber keinen Einfluß auf die Sperrenverwaltung und muß ihr deshalb auch nicht mitgeteilt werden.

Die Operationen `abort` und `complete` führen dazu, daß die betroffenen dynamischen Aktionen aus dem Abhängigkeitsgraphen entfernt beziehungsweise entsprechend markiert werden, so daß der geänderte Zustand bei zukünftigen Auswertungen berücksichtigt werden kann. Außerdem werden die angegebenen dynamischen Aktionen über die Funktionen `CC_abort` und `CC_complete` an die Sperrenverwaltung weitergeleitet.

Kapitel 6

Implementierung

In diesem Kapitel werden zwei mögliche Implementierungen vorgestellt: eine Implementierung der Nebenläufigkeitskontrolle als eigenständiges Modul und die Einbindung in das verteilte Aktionssystem für dynamische Aktionen. Vorher werden noch die in beiden Szenarien benutzten Realisierungen der zugrundeliegenden Datentypen beschrieben.

6.1 Zugrundeliegende Datentypen

Die im vorherigen Kapitel beschriebenen Algorithmen sind unabhängig von der tatsächlichen Implementierung der Identifikatoren von Objekten, dynamischen Aktionen und Kooperationsgruppen. Allerdings wurden einige Bedingungen aufgestellt, die die Identifikatoren für eine ordnungsgemäße Funktionalität der Nebenläufigkeitskontrolle einhalten müssen.

Von den Identifikatoren der Objekte wird nur ihre Eindeutigkeit gefordert. Da keine weiteren Anforderungen existieren, sind die Identifikatoren als einfache Zeichenkette implementiert, für deren Eindeutigkeit die Applikation verantwortlich ist.

An die Identifikatoren der dynamischen Aktionen und der Kooperationsgruppen werden umfassendere Anforderungen gestellt: Ihre Eindeutigkeit muß systemweit auch bei verteilten Systemen gewährleistet werden. Dabei sollen die Identifikatoren vor jeder Stelle autonom erzeugt werden können, da eine zentrale Vergabe oder eine andere Synchronisation zu zeitaufwendig wäre. Außerdem muß die Eindeutigkeit auch bei zwischenzeitlichen Systemausfällen gewährleistet sein, das heißt vor und nach dem Ausfall dürfen nicht dieselben Identifikatoren generiert werden. Um diese Anforderungen zu erfüllen, werden für die dynamische Aktionen Identifikatoren auf Basis der in Abbildung 6.1 gezeigten *unique identifiers (UID)* eingeführt (siehe hierzu auch [Baer 1992] und [Bröhl 1995]):

unique identifier:	$<pc,\ gn,\ vc>$	mit	*pc:*	permanent counter
			gn:	generator name
			vc:	volatile counter

Abbildung 6.1: Identifikator (UID) für dynamische Aktionen

Die systemweite Eindeutigkeit dieser Identifikatoren wird dadurch erreicht, daß jeder Stelle ein eindeutiger *generator name* zugeordnet wird. Bei jeder Erzeugung eines Identifikators auf einer Stelle wird der Zähler *vc* inkrementiert, so daß für den fehlerfreien Betrieb die Eindeutigkeit der Identifikatoren gesichert ist. Da dieser Zähler bei einem Systemausfall seinen Wert verliert, existiert zusätzlich der Zähler *pc*. Dieser Zähler wird permanent gespeichert und bei jedem Systemstart inkrementiert. Lautet der letzte erzeugte Identifikator auf einer Stelle `<3,7,4>`, so wird bei fehlerfreiem Betrieb auf dieser Stelle als nächstes der Identifikator `<3,7,5>` erzeugt. Nach einem Ausfall der Stelle würde hingegen der Identifikator `<4,7,1>` erzeugt.

Eine weitere Anforderung an die Identifikatoren von geschachtelten dynamischen Aktionen besagt, daß mit ihrer Hilfe Vorfahre-Nachfahre-Beziehungen von dynamischen Aktionen erkannt werden müssen. Dies wird dadurch erreicht, daß ein Identifikator einer dynamischen Aktion aus einem Pfad von UIDs zusammengesetzt wird: Der Identifikator einer Toplevel-Aktion ist eine eindeutige UID. Beim Erzeugen eines Sohnes wird an den Identifikator-Pfad des Vaters ein neuer eindeutiger UID angehängt. Abbildung 6.2 zeigt Beispiele für Identifikatoren von dynamischen Aktionen.

Identifikator einer Toplevel-Aktion:	`/<5,1,1>1`
Identifikator eines Sohns:	`/<5,1,1>1/<5,1,2>1`

Abbildung 6.2: Beispiel für Identifikatoren von dynamischen Aktionen

Verwandtschaftsbeziehungen lassen sich anhand der Identifikatoren folgendermaßen erkennen: Der Identifikator einer dynamischen Aktion ist Präfix der Identifikatoren aller ihrer Nachfahren; zwei dynamische Aktionen gehören zum selben Aktionsbaum, wenn ihre Identifikatoren einen gemeinsamen Präfix haben.

Der Eintrag `1` nach jedem UID gibt den Status der zugehörigen dynamischen Aktion an. Für die Nebenläufigkeitskontrolle sind dabei zwei Werte von Interesse: Der Wert `1` bezeichnet dynamische Aktionen und mit dem Wert `2` werden Kooperationsgruppen gekennzeichnet.

Kooperationsgruppen werden bei der Vergabe von kooperativen Sperren als virtuelle Väter der Toplevel-Aktionen behandelt. Dieses wird realisiert, indem der Kooperationsgruppe wie einer dynamischen Aktion ein Identifikator zugeordnet wird. Den Identifikatoren der dynamischen Aktion dieser Kooperationsgruppe wird beim Wechsel in die zweite Phase der Identifikator der Kooperationsgruppe als Präfix vorangestellt wird, wie in Abbildung 6.3 gezeigt ist. Dadurch sind die Algorithmen zur Vergabe und Freigabe von kooperativen Sperren ohne spezielle Anpassung auch bei Sperren für Kooperationsgruppen anwendbar.

Identifikator einer Kooperationsgruppe:	`/<5,1,4>2`
Identifikator einer Toplevel-Aktion:	`/<5,1,4>2/<5,1,1>1`
Identifikator eines Sohns:	`/<5,1,4>2/<5,1,1>1/<5,1,2>1`

Abbildung 6.3: Beispiel für Identifikatoren von dynamischen Aktionen nach Beitritt zu einer Kooperationsgruppe

6.2 Eigenständiges Modul zur Nebenläufigkeitskontrolle

Das kooperative Sperrverfahren wurde als eigenständiges Modul zur Nebenläufigkeitskontrolle implementiert. Dieses Modul kann als Bibliothek in Applikationsprogramme eingebunden werden und stellt diesem die in Kapitel 5 beschriebenen Operationen zur Verfügung. Die Implementierung der Nebenläufigkeitskontrolle folgt dabei der in Abbildung 5.1 dargestellten Struktur, wobei die Abhängigkeitsverwaltung, wie bereits in Abschnitt 5.5 dargelegt, durch den in [Theisohn 1996] eingeführten generischen Abhängigkeitsgraphen realisiert wird. Die Implementierung erstreckt sich daher hauptsächlich auf die Komponente der Sperrenverwaltung.

6.2.1 Rahmenbedingungen der Implementation

Die Sperrenverwaltung wurde in der Programmiersprache „C“ nach dem ANSI-Standard für SUN Sparc Workstations sowohl unter SunOS als auch unter Solaris entwickelt. Da die Sperrenverwaltung aber nicht auf spezifischen Architektur- oder Betriebssystemeigenschaften basiert, ist eine Portierung auf andere Plattformen relativ einfach möglich, sofern ein ANSI-C-Compiler zur Verfügung steht. Die Sperrenverwaltung hat einen Umfang von ca. 11.000 Zeilen Programmcode.

Zur Implementierung der Sperrenverwaltung für das kooperative Sperrverfahren wurden Teile des Programmcodes und der Datenstrukturen aus der Sperrenverwaltung übernommen, die bereits als Komponente der Aktionsverwaltung für geschachtelte dynamische Aktionen implementiert war.

6.2.2 Funktionalität der Nebenläufigkeitskontrolle

Die implementierte Nebenläufigkeitskontrolle realisiert in vollem Umfang die in Kapitel 5 beschriebenen Operationen. Durch die Implementation als Programmbibliothek wird dem Applikationsprogramm die Möglichkeit gegeben, die Mechanismen zur Nebenläufigkeitskontrolle von dynamischen Aktionen für kooperative Anwendungen zu nutzen. Die dynamische kooperative Serialisierbarkeit wird dabei für die durch dieses Modul verwalteten dynamischen Aktionen durchgesetzt, das heißt eine Durchsetzung des Konsistenzkriteriums über mehrere verteilte Stellen hinweg ist nicht möglich. Eine Diskussion der Implementierung in einem verteilten Aktionssystem folgt in Abschnitt 6.3. Das eigenständige Modul zur Nebenläufigkeitskontrolle läßt sich beispielsweise dann einsetzen, wenn kooperatives Arbeit mehrerer Benutzer auf einem zentralen Server unterstützt werden soll. Eine solche Anwendung wurde in [Mock, Gergeleit 1997] zur Kooperationsunterstützung auf einem Server im World Wide Web vorgestellt.

Damit die Nebenläufigkeitskontrolle in der Praxis nutzbar ist, muß das Modul um eine weitere Funktionalität ergänzt werden, die eigentlich Aufgabe der Fehlerbehandlung ist: Im Falle eines Abbruchs muß eine Schadensbewertung stattfinden, damit alle abhängigen dynamischen Aktionen ebenfalls abgebrochen werden können. Zu diesem Zweck müssen neben den Konflikt- auch die Informationsabhängigkeiten erkannt und gespeichert werden. Zur Speicherung und Auswertung läßt sich der generische Abhängigkeitsgraph, wie in [Theisohn 1996] gezeigt, um einen weiteren Kantentyp erweitern. Entstehende Informationsabhängigkeiten können durch die Operation `setlock` erkannt werden und werden durch die in Abbildung 6.4 gezeigte Erweiterung des Algorithmus an die Abhängigkeitsverwaltung gemeldet. Dabei wird eine Informationsabhängigkeit von der letzten schreibenden zur anfordernden dynamischen Aktion gemeldet, falls in der Sperrenliste ein schreibender Eintrag vorhanden ist.

```
int setlock (AID_TYPE* ReqAID, RESID_TYPE* ReqResID, RW_MODE ReqMode,
             int nReqPhase, KOG_TYPE* ReqKOG)
{
   LOCK_ENTRY_TYPE *LockList = GetLockList (ReqResID);

   ...    // Teste, ob Sperre gewährt werden kann (siehe Abbildung 5.10)

   // Melde Konfliktabhängigkeiten an die Abhängigkeitsverwaltung ...
   if (ReqMode == READ)
      ∀ entry ∈ LastWriters(LockList): add_CC_dependency (entry.AID, ReqAID);
   if (ReqMode == WRITE)
      ∀ entry ∈ LastEntries(LockList): add_CC_dependency (entry.AID, ReqAID);

   // Melde Informationsabhängigkeit an die Abhängigkeitsverwaltung ...
   if (∃ entry ∈ LockList: entry.Mode == WRITE)
      add_information_dependency (LastWriter(LockList).AID, ReqAID);

   ...    // Hänge neuen Eintrag an die Sperrenliste an (siehe Abbildung 5.10)

}
```

Abbildung 6.4: Erweiterung des Algorithmus der Operation `setlock`

6.2.3 Interaktive Nutzerschnittstelle

Zur Demonstration des kooperativen Sperrverfahrens wurde eine interaktive Nutzerschnittstelle entwickelt, die es dem Anwender erlaubt, über Kommandozeilenbefehle alle Operationen der Nebenläufigkeitskontrolle durchzuführen. Neben der tabellarischen Darstellung aller dynamischen Aktionen und Objekte erlaubt die Nutzerschnittstelle außerdem, den Abhängigkeitsgraphen mit Hilfe des VCG-Werkzeugs (siehe [Sander 1994] und [Sander 1995]) graphisch darzustellen.

Die Ausgabe des `HELP`-Kommandos in Abbildung 6.5 zeigt alle erlaubten Kommandos der interaktiven Nutzerschnittstelle, die sich in drei Gruppen aufteilen lassen: Der erste in Abbildung 6.5 dargestellte Block enthält Kommandos, die keinen Einfluß auf das Verhalten der dynamischen Aktionen haben, sondern nur der Steuerung der Nutzerschnittstelle dienen. Zum zweiten Block gehören die Kommandos, die den Zustand der dynamischen Aktionen beeinflussen, während die dritten Block die Kommandos zur Sperrenanforderung und -freigabe zusammengefaßt sind. Die beiden Kommandos `BEGIN` und `SPAWN` im zweiten Block haben dabei keine direkte Auswirkung auf die Nebenläufigkeitskontrolle, sondern dienen der Erzeugung von Identifikatoren für dynamische Aktionen, die in darauf folgenden Kommandos der Nebenläufigkeitskontrolle als Argumente der Operationen übergeben werden. Für die Operationen `setgrouplock` und `releasegrouplock` existieren keine zusätzlichen Kommandos. Diese Operationen werden aufgerufen, indem bei den Kommandos „`S`“ und „`R`“ der Identifikator der Kooperationsgruppe als erster Parameter angegeben wird.

Abbildung 6.6 zeigt ein Beispiel für die Ausgabe der Nutzerschnittstelle. Links sind dabei die dynamischen Aktionen mit ihren Zuständen dargestellt und rechts sieht man die Objekte und die Sperren, die die einzelnen dynamischen Aktionen momentan auf diesen Objekten halten. Dabei stehen Großbuchstaben („`R`“ und „`W`“) für kooperative Sperren und Kleinbuchstaben („`r`“ und „`w`“) für Benutzungssperren.

```
Enter command: h

Valid commands are:

  X               - Exit program
  H               - Display this help screen
  !               - Refresh display
  @               - Draw/Update VCG dependency graph
  > [filename]    - Start/stop recording
  <  filename     - Playback recorded input

  B               - Begin dynamic action
  W father-aid    - spaWn child action
  C aid           - Complete dynamic action
  A aid ...       - Abort dynamic action
  J aid ... [kog] - Join or create a kog
  V kog           - Validate a kog

  S aid oid r/w   - Set lock
  R aid oid r/w   - Release lock
  U aid oid r/w   - Use lock
  F aid oid r/w   - Free lock

Enter command: _
```

Abbildung 6.5: Operationen der interaktiven Nutzerschnittstelle

```
+-----------------------------------------------------------------------------
| # |   KOG    Action-ID                      State Phase| Obj1 Obj2
+---+----------------------------------------------------+-------------------
|   |                                                    |
| 1 | <1,1,4>                                 -KOG-   2  |   W
|   |   ###                                              |
| 2 |   ###    /<1,1,1>1                      active  2  |   Ww
|   |   ###                                              |
| 3 |   ###    /<1,1,2>1                      compl.  2  |
|   |                                                    |
| 4 |          /<1,1,3>1                      active  1  |      rR
+-----------------------------------------------------------------------------

Enter command: f #2 Obj1 w_
```

Abbildung 6.6: Beispiel für Ausgabe der interaktiven Nutzerschnittstelle

In dem dargestellten Szenario gehören die dynamischen Aktionen `/<1,1,1>1` und `/<1,1,2>1` zur Kooperationsgruppe `<1,1,4>` und befinden sich in der zweiten Phase, während sich die dynmische Aktion `/<1,1,3>1` noch in der ersten Phase befindet und daher noch zu keiner Kooperationsgruppe gehört. Die dynamische Aktion `/<1,1,2>1` ist bereits im Zustand *completed* und darf daher keine Sperren mehr anfordern. `/<1,1,1>1` besitzt zur Zeit eine kooperative Schreibsperre und eine Schreib-Benutzungssperre auf Objekt `Obj1` und die zugehörige Kooperationsgruppe besitzt ebenfalls eine kooperative Schreibsperre auf dieses Objekt. Die dynamische Aktion `/<1,1,3>1` besitzt beide Lesesperren auf Objekt `Obj2`. Mit dem nächsten, bereits eingegebenen Befehl gibt die dynamische Aktion `/<1,1,1>1` die Schreib-Benutzungssperre auf `Obj1` wieder frei („`#2`" ist dabei eine Abkürzung für die dynamische Aktion aus der zweiten Zeile).

6.2.4 Anwendungsbeispiel

Zur Veranschaulichung der Benutzerschnittstelle und des Moduls zur Nebenläufigkeitskontrolle wird hier wieder das Beispiel der kooperativen Texterstellung aus der Einleitung aufgenommen. Der hier dargestellt Ablauf entspricht genau dem in Abschnitt 3.4.4 beschriebenen Szenario.

Abbildung 6.7 zeigt die Situation, in der Autor A_1, repräsentiert durch die dynamische Aktion `/<1,1,1>1`, die Einleitung bearbeitet und deshalb für das Objekt `Einl` eine kooperative Schreibsperre und eine Schreib-Benutzungssperre hält. Der zweite Autor wird durch die dynamische Aktion `/<1,1,2>1` repräsentiert und die Kooperationsgruppe hat den Identifikator `<1,1,3>`. Das Übersetzungsbüro wurde zu diesem Zeitpunkt noch nicht beauftragt, so daß die zugehörige dynamische Aktion der Nebenläufigkeitskontrolle noch nicht bekannt ist. Die Kooperationsgruppe hält auf alle drei Objekte eine kooperative Schreibsperre, um Zugriffe von fremden dynamischen Aktionen zu verhindern.

```
+-------------------------------------------------------------------------
| # |   KOG    Action-ID                         State Phase| Einl Mitt Schl
+---+---------------------------------------------------+-----------------
|   |                                                   |
| 1 | <1,1,3>                                   -KOG-   2 |   W    W    W
|   |   ###                                             |
| 2 |   ###    /<1,1,1>1                        active  2 |   Ww
|   |   ###                                             |
| 3 |   ###    /<1,1,2>1                        active  2 |
+-------------------------------------------------------------------------

Enter command: _
```

Abbildung 6.7: Nutzerschnittstelle für Beispielanwendung (1)

In der folgenden Abbildung 6.8 hat Autor A_2 das durch `/<1,1,2>1/<1,1,4>1` repräsentierte Übersetzungbüro beauftragt und dieses bearbeitet gerade den Mittelteil. Autor A_2 hält eine kooperative Schreibsperre zur Durchsetzung des Auftragsprinzips auf den Mittelteil und Autor A_1 verhindert durch die kooperative Lesesperre auf die Einleitung Änderungen an diesem Objekt.

Der Abhängigkeitsgraph hat zu diesem Zeitpunkt die in Abbildung 6.9 vom VCG-Werzeug dargestellte Struktur. Das VCG-Werkzeug zeigt dabei nicht den gesamten Identifikatorpfad an, sondern nur den jeweils letzten - eindeutigen - Identifikator. Die Numerierung der Kanten symbolisiert die Abhängigkeitstypen, wobei „2" für eine Vater-Sohn-Abhängigkeit und „4" für eine Kon-

```
+---------------------------------------------------------------------------
| # |   KOG    Action-ID                         State Phase| Einl Mitt Schl
+---+-----------------------------------------------------+-----------------
|   |                                                     |
| 1 | <1,1,3>                                   -KOG-   2 |   W    W    W
|   |   ###                                               |
| 2 |   ###    /<1,1,1>1                        active  2 |  R
|   |   ###                                               |
| 3 |   ###    /<1,1,2>1                        active  2 |        W
| 4 |   ###    /<1,1,2>1/<1,1,4>1               active  2 |        Ww
+---------------------------------------------------------------------------

Enter command: _
```

Abbildung 6.8: Nutzerschnittstelle für Beispielanwendung (2)

fliktabhängigkeit steht. Die Konfliktabhängigkeit von `/<1,1,1>1` nach `/<1,1,2>1` entstand beim Anfordern der kooperativen Lesesperre auf die Einleitung durch Autor A_2. Entsprechend entstand die weitergehende Konfliktabhängigkeit zu `/<1,1,2>1/<1,1,4>1` bei der Vergabe der kooperativen Schreibsperre auf den Mittelteil an den Übersetzer, da Autor A_2 zwischenzeitlich auf dieses Objekt eine kooperative Schreibsperre hielt.

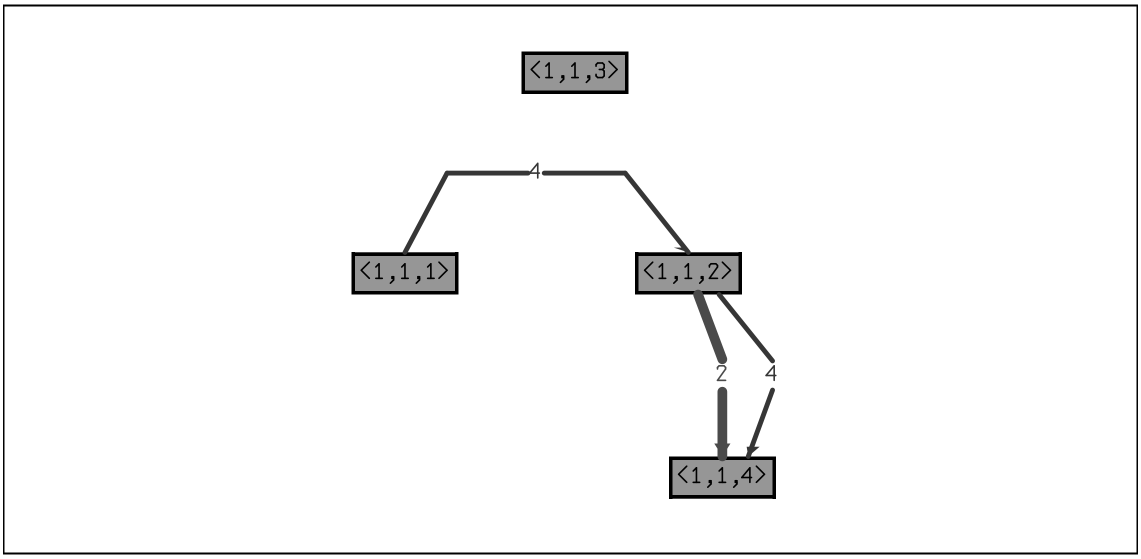


Abbildung 6.9: Abhängigkeitsgraph zu Beispielanwendung (2)

In dem letzten in Abbildung 6.10 gezeigten Beispiel hat Autor A_1 gerade die Arbeit am Schluß begonnen. Vorher haben der Übersetzer und Autor A_2 ihre Arbeit am Mittelteil beendet und Autor A_2 behält zum Schutz vor Änderungen eine kooperative Lesesperre auf den Mittelteil. Autor A_1 hält gerade beide Lesesperren auf den Mittelteil, um sich über dessen Inhalt zu informieren.

Der Abhängigkeitsgraph wurde entsprechend Abbildung 6.11 erweitert. Die entstandenen Zyklen sind dabei nur deshalb erlaubt, weil alle beteiligten dynamischen Aktionen zur selben Kooperationsgruppe gehören.

```
+-----------------------------------------------------------------------------
| # |   KOG    Action-ID                           State Phase| Einl Mitt Schl
+---+-------------------------------------------------+-------------------
|   |                                                  |
| 1 | <1,1,3>                                 -KOG-   2  |   W    W    W
|   |   ###                                            |
| 2 |   ###    /<1,1,1>1                      active  2  |  R   rR     Ww
|   |   ###                                            |
| 3 |   ###    /<1,1,2>1                      active  2  |      R
| 4 |   ###    /<1,1,2>1/<1,1,4>1             compl.  2  |
+-----------------------------------------------------------------------------

Enter command: _
```

Abbildung 6.10: Nutzerschnittstelle für Beispielanwendung (3)

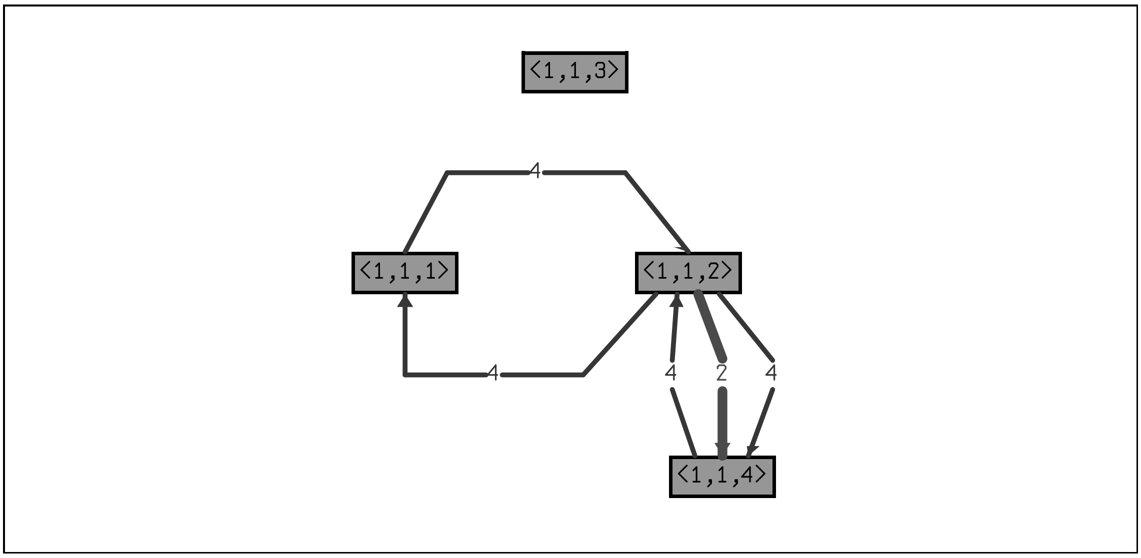


Abbildung 6.11: Abhängigkeitsgraph zu Beispielanwendung (3)

6.3 Einbettung in das verteilte Aktionssystem für dynamische Aktionen

Neben der im vorherigen Abschnitt beschriebenen Implementierung als eigenständiges Modul zur Nebenläufigkeitskontrolle soll das kooperative Sperrverfahren auch als möglicher Mechanismus zur Nebenläufigkeitskontrolle in das verteilte Aktionssystem für dynamische Aktionen integriert werden. Dieses Aktionssystem erlaubt die Verwaltung von dynamischen Aktionen, die sich im verteilten System über mehrere Stellen ausdehnen und auch unterschiedliche Arten von Ressourcen benutzen können.

Das Aktionssystem wurde bereits umfassend in [Kröger, Mock et al. 1990] oder auch [Mock 1994] beschrieben. Daher werden in diesem Abschnitt nach einem kurzen Überblick nur gezielt einige Aspekte der Nebenläufigkeitskontrolle betrachtet.

6.3.1 Architektur

Das Aktionssystem für dynamische Aktionen stellt in der Architektur eine Erweiterung des weit verbreiteten X/Open Modells dar (siehe [X/Open Company 1993]). Dementsprechend existieren auf jeder Stelle im verteilten System zur Aktionsunterstützung zwei unterschiedliche Komponenten: die *Aktions-* und die *Ressourcenmanager*. Grundsätzlich existieren auf jeder Stelle genau ein Aktionsmanager und möglicherweise mehrere Ressourcenmanager, wobei jedem Ressourcentyp ein eigener Manager zugeordnet ist. Relationale oder objektorientierte Datenbanken bilden dabei beispielsweise je einen Ressourcentyp, genauso wie Dateisysteme. Die Komponenten wirken, wie in Abbildung 6.12 dargestellt, zusammen und haben dabei folgende Aufgaben:

- Der Aktionsmanager verwaltet die Zustände der dynamischen Aktionen. Dies beinhaltet insbesondere Ende und Abbruch der dynamischen Aktionen einschließlich der Bestimmung von Abhängigkeitsmengen, die Ausbreitung der dynamischen Aktionen auf andere Stellen und die Durchführung verteilter Protokolle bei der Bildung von Kooperationsgruppen und bei der Validierung. Der Aktionsmanager besitzt keine Informationen über die von den dynamischen Aktionen benutzten Objekte.

- Die Ressourcenmanager steuern die Vergabe und Freigabe von Sperren und führt objektspezifische Operationen zur Fehlerbehandlung und Zustandssicherung durch. Außerdem erkennen sie entstehende Abhängigkeiten zwischen dynamischen Aktionen und melden diese an den Aktionsmanager. Die Ressourcenmanager besitzen keine umfassenden Informationen über die Zustände der dynamischen Aktionen, sondern nur solche, die sie für ihre Aufgaben benötigen. So kennt ein Ressourcenmanager zum Beispiel für die Sperrenvergabe die Phase einer dynamischen Aktion, die ein Objekt gesperrt hat, weiß aber nicht, auf welche anderen Stellen die dynamische Aktion sich ausgebreitet hat. Ein Ressourcenmanager besitzt nur Informationen über dynamische Aktionen, die auf ein von ihm verwaltetes Objekt zugreifen.

Aktions- und Ressourcenmanager kommunizieren miteinander. Der Ressourcenmanager teilt beispielsweise entstandene Abhängigkeiten mit und der Aktionsmanager informiert den Ressourcenmanager über Zustandswechsel von dynamischen Aktionen oder beauftragt ihn bei einem Abbruch, gesicherte Objektzustände zurückzuschreiben oder eventuell vorhandene Sperren freizugeben. Die Aktionsmanager auf unterschiedlichen Stellen haben die Möglichkeit, miteinander zu kommunizieren, da zum Beispiel von Abbrüchen oder der Validierung dynamische Aktionen auf unterschiedlichen Stellen betroffen sein können.

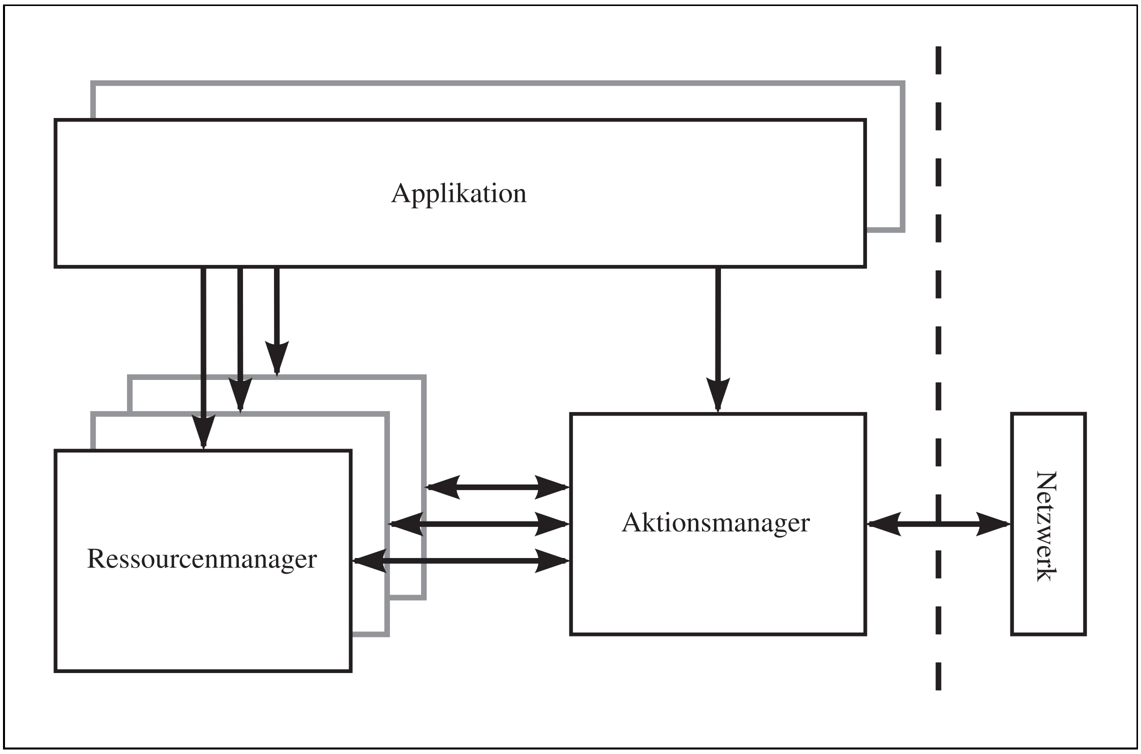


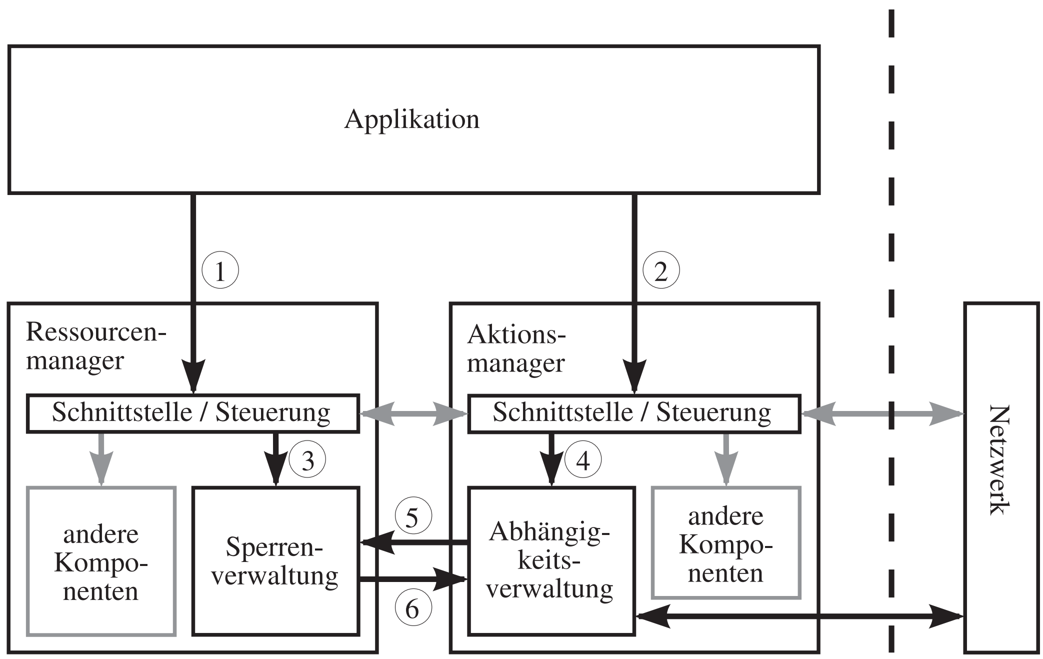
Abbildung 6.12: Komponenten des Aktionssystems auf einer Stelle

6.3.2 Nebenläufigkeitskontrolle im verteilten Aktionssystem

Die Beschreibung von Aktions- und Ressourcenmanager zeigt deutlich, wie die im vorherigen Abschnitt 6.2 eingeführten Komponenten der Nebenläufigkeitskontrolle in das Aktionssystem eingebunden werden müssen: Die Sperrenverwaltung ist Teil des Ressourcenmanagers, da sie die Zugriffe auf die einzelnen Objekte steuert. Da die Abhängigkeitsverwaltung für die verteilten Algorithmen wie die Validierung und die Bildung von Kooperationsgruppen zuständig ist, gehört sie zum Aktionsmanager. Die Einbindung der Nebenläufigkeitskontrolle in das verteilte Aktionssystem und die für die Nebenläufigkeitskontrolle relevanten Schnittstellen sind in Abbildung 6.13 dargestellt, wobei aus Gründen der Übersichtlichkeit auf die Darstellung mehrerer Ressourcenmanager und Applikationsprogramme auf einer Stelle verzichtet wurde.

In der dargestellten Architektur wird die Sperrenverwaltung über die gleichen Operationen aufgerufen, wie dieses auch schon im eigenständigen Modul zur Nebenläufigkeitskontrolle geschieht. Der einzige Unterschied besteht darin, daß die Operationen zur Sperrenanforderung und -freigabe nicht direkt vom Applikationsprogramm aufgerufen werden, sondern es existiert im Ressourcenmanager eine Zwischenschicht, die die Schnittstelle zum Anwendungsprogramm zur Verfügung stellt. Durch diese Zwischenschicht können die eigentlichen Aufrufe der Sperrenverwaltung vor dem Applikationsprogramm gekapselt werden. So findet die Kommunikation zwischen Ressourcenmanager und Applikation über Messages statt, während die Sperrenverwaltung wie beim eigenständigen Modul als Bibliothek in den Ressourcenmanager eingebunden werden kann. Außerdem lassen sich so neben der Sperrenverwaltung weitere Funktionen beispielsweise an das Anfordern einer Sperre koppeln. So ließe sich bei Gewährung einer Sperre automatisch durch die Fehlerbehandlung ein Sicherungspunkt für das betroffene Objekt anlegen.

Die für die eigenständige Nebenläufigkeitskontrolle implementierte Sperrenverwaltung läßt sich durch diese Architektur ohne Änderung auch im Ressourcenmanager einsetzen.



Operationen an den Schnittstellen zwischen den Komponenten:

(1)	**(2)**	**(3)**	**(4)**
`setlock`	`joinkog`	`RM_setlock`	`AM_joinkog`
`releaselock`	`validate`	`RM_releaselock`	`AM_validate`
`setgrouplock`	`complete`	`RM_setgrouplock`	`AM_complete`
`releasegrouplock`	`abort`	`RM_releasegrouplock`	`AM_abort`
`uselock`		`RM_uselock`	
`freelock`		`RM_freelock`	

(5)	**(6)**
`CC_joinkog`	`add_CC_dependency`
`CC_validate`	
`CC_complete`	
`CC_abort`	

Abbildung 6.13: Einbindung der Nebenläufigkeitskontrolle in das Aktionssystem

Auch die Abhängigkeitsverwaltung besitzt, abgesehen von der auch im Aktionsmanager vorhandenen Zwischenschicht, die gleichen Funktionen wie in der eigenständigen Nebenläufigkeitskontrolle. Hier kommt allerdings hinzu, daß Abhängigkeitsmengen bei der Bildung von Kooperationsgruppen und bei der Validierung über mehrere Stellen hinweg bestimmt werden müssen. Die einzelnen Abhängigkeitsverwaltungen kommunizieren hierzu miteinander über das Netzwerk. Der eingesetzte generische Abhängigkeitsgraph stellt die geforderte Funktionalität der verteilten Berechnung von Abhängigkeitsmengen bereits zur Verfügung, so daß zur genaueren Erörterung dieses Punktes auf [Theisohn 1996] verwiesen wird.

6.3.3 Blockierende und nicht-blockierende Aufrufe

In der bisher vorgestellten Implementation gibt die Sperrenverwaltung lediglich den Wert `FAILED` zurück, wenn eine Sperre zum Zeitpunkt der Anforderung nicht gewährt werden kann. Falls das Applikationsprogramm das Objekt zwingend benötigt, muß es nach einer geeigneten Pause erneut die Sperre anfordern. Dies muß so oft wiederholt werden, bis das Objekt nicht mehr gesperrt ist. Zum einen wird durch diese ständige Anfrage das System unnötig belastet und zum anderen besteht in einem verteilten System mit mehrere Applikationen die Gefahr, daß in der Wartezeit das Objekt freigegeben, aber auch schon wieder von einer dritten Applikation gesperrt wurde.

Um solche Effekte zu vermeiden, bietet der Ressourcenmanager die Möglichkeit, Sperren mit sogenannten *blockierenden* Aufrufen anzufordern. Dieser Aufruf unterscheidet sich vom bekannten, *nicht-blockierenden* Aufruf dadurch, daß er für eine von der Applikation festgelegte Zeit nicht mit dem Fehler `FAILED` beendet wird, sondern wartet, daß die Sperre freigegeben wird. Geschieht dieses in der angegebenen Zeit, wird die Sperre automatisch an die wartende dynamische Aktion vergeben.

Zu diesem Zweck muß die Sperrenverwaltung derart erweitert werden, daß sie bei der Freigabe einer Sperre überprüfen kann, ob eine weitere dynamische Aktion bereits auf diese Sperre wartet. Zur Realisierung dieser Funktionalität wird für jedes Objekt neben der eigentlichen Sperrenliste eine zweite Liste, die sogenannte *Waiterlist* geführt. In diese Liste werden diejenigen dynamischen Aktionen aufgenommen, die die Sperre blockierend angefordert haben und denen die Sperre zum Zeitpunkt der Anforderung nicht gewährt werden kann.

Enthält die Waiterlist bei der Freigabe einer Sperre einen oder mehrere Einträge, wird für die Einträge der Liste die Operation `setlock` beziehungsweise `uselock` durchgeführt. Dabei wird die Liste in der Reihenfolge bearbeitet, in der die Einträge in die Liste aufgenommen wurde. Ist eine Operation erfolgreich, wird der anfordernden dynamischen Aktion die Sperre gewährt, die Operation gibt den Wert `OK` zurück und der Eintrag wird aus der Waiterlist gelöscht. Es besteht aber auch die Möglichkeit, daß der anfordernden dynamischen Aktion aufgrund der Sperrenregeln die Sperre nicht gewährt werden darf. In diesem Fall kann entweder der anfordernden dynamischen Aktion der Wert `FAILED` zurückgegeben werden oder der Eintrag bleibt bis zum Ende der festgesetzten Zeit in der Waiterlist. Bei weiteren Freigaben von Sperren und bei Phasenwechseln und Abbrüchen von dynamischen Aktionen muß erneut überprüft werden, ob der wartenden dynamischen Aktion die Sperre nun gewährt werden kann. Erst bei Ablauf der angegebenen Zeit gilt die Anforderung als fehlgeschlagen und die Operation gibt den Wert `FAILED` zurück.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde das Konzept der *geschachtelten dynamischen Aktionen für kooperative Anwendungen* entwickelt. Dieses Konzept basiert auf den in [Nett 1991] eingeführten *dynamischen Aktionen* und integriert die beiden Konzepte der *geschachtelten dynamischen Aktionen* und der *dynamischen Aktionen für kooperative Anwendungen.*

Auftrags- und Kooperationsprinzip

Ausgangspunkt für die Entwicklung des vorgestellten Integrationskonzepts war die Beobachtung, daß zwischen den Aktivitäten der Benutzer eines verteilten Systems oft Beziehungen existieren. Diese Beziehungen resultieren in einem Informationsfluß zwischen den Transaktionen, die ein verbreitetes Mittel zur Beschreibung solcher Aktivitäten in verteilten Systemen sind.

Eine Betrachtung der in der Praxis auftretenden Arbeitsabläufe hat gezeigt, daß der Informationsfluß fast immer nach einem von zwei Prinzipien verläuft: dem Auftrags- oder dem Kooperationsprinzip. Diese beiden Prinzipien stellen bei der Steuerung von Transaktionen unterschiedliche Anforderungen an die Nebenläufigkeitskontrolle, wohingegen die Mechanismen zur Fehlerbehandlung nicht Bestandteil des jeweiligen Prinzips sind, sondern als korrekt arbeitend vorausgesetzt werden.

Beide Prinzipien werden einzeln bereits durch bekannte erweiterte Transaktionssysteme unterstützt. Mit den offen und den geschlossen geschachtelten Transaktionen wurden die zwei bekanntesten Konzepte zur Unterstützung des Kooperations- beziehungsweise des Auftragsprinzips vorgestellt. Am Beispiel des DOM Transaktionsmodells wurde gezeigt, daß die Entwicklung eines Integrationskonzepts durch Kombination beider geschachtelter Transaktionsmodelle nicht möglich ist. Der Grund hierfür liegt darin, daß diese Transaktionssysteme nicht zwischen Fehlerbehandlung und Nebenläufigkeitskontrolle trennen, sondern zur Durchführung der Fehlerbehandlung das spezielle nebenläufige Verhalten der Transaktionen ausnutzen. Die für die Integration notwendigen Änderungen im Bereich der Nebenläufigkeitskontrolle führen daher direkt zu Problemen bei der Fehlerbehandlung.

Das CONCORD Modell versucht daher, Auftrags- und Kooperationsprinzip nicht auf der Basis von offen oder geschlossen geschachtelten Transaktionen durchzusetzen, sondern wählt statt dessen den Weg einer expliziten Spezifizierung der Beziehungen zwischen den Aktivitäten der Benutzer in Kombination mit einem versionierten Objektsystem. Die Untersuchung dieses Ansatzes hat aber gezeigt, daß Auftrags- und Kooperationsprinzip auf diese Weise nur durchgesetzt werden können, indem die Anwender in großem Umfang die Aufgaben der Nebenläufigkeitskontrolle und Fehlerbehandlung zur Konsistenzerhaltung der Objekte übernehmen. Diese Aufgabe ist in großen verteilten Systemen mit vertretbarem Aufwand nicht zufriedenstellend zu lösen.

Integrationskonzept

Das Konzept der dynamischen Aktionen garantiert im Gegensatz zu den Transaktionskonzepten eine korrekte Fehlerbehandlung unabhängig von den eingesetzten Mechanismen der Nebenläufigkeitskontrolle. Daher lassen sich hier beide Prinzipien realisieren und auch integrieren, ohne die Fehlerbehandlung zu beeinflussen. Es existieren mit den *geschachtelten dynamischen Aktionen* und den *dynamischen Aktionen für kooperative Anwendungen* bereits zwei auf dynamischen Aktionen basierende Konzepte, die jeweils das Auftrags- beziehungsweise das Kooperationsprinzip realisieren. Durch Integration dieser beiden Konzepte wurde in dieser Arbeit ein Konzept entwickelt, welches das Auftrags- und das Kooperationsprinzip unterstützt.

Bei der Entwicklung des Integrationskonzepts wurde herausgearbeitet, daß die Mechanismen der Nebenläufigkeitskontrolle zwei verschiedene Aspekte betreffen: die Serialisierbarkeit von dynamischen Aktionen und die Sichtbarkeit von Objektzugriffen. An die Serialisierbarkeit stellen beide Prinzipien die gleiche Forderung: Sowohl Kooperationspartner als auch Auftraggeber und Auftragnehmer müssen untereinander nicht zwingend serialisierbar sein. Es wurde außerdem dargelegt, daß zur Vermeidung von unerwünschten Nebenläufigkeitsanomalien Serialisierbarkeit zwischen dynamischen Aktionen, die nicht direkt oder indirekt in Beziehung stehen, durchgesetzt werden sollte. Das im Rahmen der *dynamischen Aktionen für kooperative Anwendungen* eingeführte Konsistenzkriterium der *dynamischen kooperativen Serialisierbarkeit* setzt genau diese Anforderungen durch und wurde deshalb in das Integrationskonzept übernommen. Nach diesem Konsistenzkriterium werden dynamische Aktionen, die nicht serialisierbar sein müssen, zu *Kooperationsgruppen* zusammengefaßt. Zwischen dynamischen Aktionen aus unterschiedlichen Kooperationsgruppen wird Serialisierbarkeit garantiert.

Beide Prinzipien unterscheiden sich aber in ihren Anforderungen an die Sichtbarkeit von Objektänderungen, da Ergebnisse eines Auftragnehmers bei Beendigung der Aufgabe nur für den Auftraggeber sichtbar werden sollen, aber zur Kooperation Zwischenergebnisse sofort ausgetauscht werden müssen. Da es ausschließlich von der Anwendungssemantik abhängt, ob in einer bestimmten Situation das Auftrags- oder das Kooperationsprinzip angewandt wird, ist in dieser Arbeit der Mechanismus der *Objektkontrolle* eingeführt worden. Mit Hilfe der Objektkontrolle kann eine dynamische Aktion individuell für jedes Objekt bestimmen, wann Zugriffe allgemein und wann sie nur von der dynamischen Aktion selbst und möglichen Auftragnehmern erlaubt sind. Durch Anforderung der Objektkontrolle für die vom Auftragnehmer benutzten Objekte kann ein Auftraggeber so das Auftragsprinzip durchsetzen. Die Objektkontrolle läßt sich auch dazu verwenden, bestimmte Objekte für die Kooperation zweier dynamischer Aktionen zu reservieren, indem die Objektkontrolle für diese Objekte von einem gemeinsamen Vorfahren erworben wird. Insbesondere die in Abschnitt 3.3.3 vorgenommene Erweiterung der Objektkontrolle auf Kooperationsgruppen erlaubt die Reservierung von Objekten für alle dynamischen Aktionen, die nach dem Konsistenzkriterium miteinander kooperieren dürfen.

Das Integrationskonzept wurde mit der Zielsetzung einer größtmöglichen Verträglichkeit mit den beiden zugrundeliegenden Konzepten entwickelt. Durch die Modellierung von Schachtelung und die Einbettung in das formale Modell, welches in [Mock 1994] für die *dynamischen Aktionen für kooperative Anwendungen* eingeführt wurde, konnte die Verträglichkeit nachgewiesen werden. Insbesondere wurde gezeigt, daß das Integrationskonzept die Eigenschaften der Abbruch- und der Commit-Korrektheit garantiert und daß das Konsistenzkriterium der dynamischen kooperativen Serialisierbarkeit auch bei Schachtelung und Objektkontrollen erfüllt wird.

Sperrverfahren

Das Integrationskonzepts wird mit dem ebenfalls hier entwickelten *kooperativen Sperrverfahren für geschachtelte dynamische Aktionen* realisiert. Um auch auf der Ebene der Sperrverfahren die Verträglichkeit mit den zugrundeliegenden Konzepten zu gewährleisten, basiert das eingeführte kooperative Sperrverfahren auf dem *geschachtelten Sperrverfahren*, welches das Konzept der *geschachtelten dynamischen Aktionen* realisiert, und auf dem (flachen) *kooperativen Sperrverfahren* zur Realisierung der *dynamischen Aktionen für kooperative Anwendungen.*

Zur Durchsetzung des Konsistenzkriteriums der dynamischen kooperativen Serialisierbarkeit wurde aus dem flachen kooperativen Sperrverfahren das dreiphasige Verhalten einschließlich der Validierung übernommen. Die Schachtelung der dynamischen Aktionen machte ähnlich dem geschachtelten Sperrverfahren zwei Ebenen von Sperren notwendig: die *kooperativen Sperren* und die *Benutzungssperren.* Die kooperativen Sperren setzen dabei den Mechanismus der Objektkontrolle um und garantieren durch die in Abschnitt 3.4.3 beschriebenen Vergaberegeln die dynamische kooperative Serialisierbarkeit, während die Benutzungssperren der Synchronisation von Objektzugriffen innerhalb von Aktionsbäumen dienen.

Das kooperative Sperrverfahren wurde in einer Komponente zur Sperrenverwaltung implementiert. Zu diesem Zweck wurde eine Schnittstelle beschrieben, die die notwendigen Operationen zur Anforderung und Freigabe von Sperren zur Verfügung stellt. Ausführlich wurden auch die Algorithmen und Datenstrukturen beschrieben, mit denen die Sperren verwaltet und die Vergabe der Sperren überprüft werden können.

Die Implementierung wurde für zwei Szenarien vorgestellt: zum einen als eigenständiges Modul zur Nebenläufigkeitskontrolle, welches von Anwendungsprogrammen als Bibliothek benutzt werden kann, und zum anderen als mögliche Nebenläufigkeitskontrolle des verteilten Aktionssystems für dynamische Aktionen. In beiden Implementierungen benutzt die Sperrenverwaltung zur Überprüfung der Kooperationsgruppen-Eigenschaften und zur Validierung den in [Theisohn 1996] beschriebenen verteilten Abhängigkeitsgraphen. Da nur innerhalb dieses Abhängigkeitsgraphen verteilte Berechnungen durchgeführt werden müssen und die Sperrenverwaltung so konzipiert ist, daß sie ihre Entscheidung über eine Sperrenvergabe aufgrund von lokalen Informationen trifft, kann sowohl im eigenständigen Modul als auch im verteilten Aktionssystem die identische Komponente zur Sperrenverwaltung verwendet werden. Schließlich wurde noch zur Demonstration des kooperativen Sperrverfahrens eine interaktive Nutzerschnittstelle für die Nebenläufigkeitskontrolle entwickelt.

Ausblick

Der in dieser Arbeit eingeführte Mechanismus der Objektkontrolle erlaubt es, den Informationsfluß zwischen dynamischen Aktionen zu steuern, ohne die durch die dynamische kooperative Serialisierbarkeit bestimmte konsistente Ausführung der dynamischen Aktionen zu verletzen. Dadurch wird es möglich, durch weitergehende Mechanismen zur Steuerung der Sichtbarkeit von Objektänderungen die Beziehungen zwischen unterschiedlichen dynamischen Aktionen noch gezielter zu unterstützen. In der Realisierung des Auftragsprinzips ist zum Beispiel eine weitere Kapselung von Auftragnehmern denkbar: Durch geeignete Mechanismen könnte ein Auftraggeber diejenigen Objekte spezifizieren, die der Auftragnehmer benutzen darf, oder auch bestimmte Objekte, die beispielsweise für die Kooperation mit anderen dynamischen Aktionen verwendet werden, vor Zugriff durch den Auftragnehmer schützen. Eine andere Möglichkeit von erweiterten Sichtbarkeitsregeln bestünde darin, Objektänderungen nur gezielt für explizit angegebene dynamische Aktionen freizugeben. Dadurch könnte zur Kooperation zwischen zwei beliebigen dynamischen Aktionen einer Kooperationsgruppe systemseitig ein unterbrechungsfreier und vor anderen dynamischen Aktionen verborgener Informationsfluß garantiert werden.

Literaturverzeichnis

[Agrawal, Abbady 1990] D. Agrawal and A. E. Abbady: *Locks with Constrained Sharing.* 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 85-93, 1990

[Badrinath, Ramamritham 1992] B. R. Badrinath and K. R. Ramamritham: *Semantics-Based Concurrency Control: Beyond Commutativity.* ACM Transactions on Database Systems 17(1): 163-199, 1992

[Baer 1992] F. Baer: *Lokale Ablaufsteuerung einer verteilten Transaktionsverwaltung.* Diplomarbeit, Universität Bonn, 1992

[Beeri, Bernstein et al. 1989] C. Beeri, P. A. Bernstein and N. Goodman: *A Model for Concurrency in Nested Transaction Systems.* Journal of the ACM 36(2): 230-269, 1989

[Bernstein, Hadzilacos et al. 1987] P. A. Bernstein, V. Hadzilacos and N. Goodman: *Concurrency control and recovery in database systems.* Addison-Wesley, 1987

[Bröhl 1995] A. Bröhl: *Ein erweitertes Recoverymodell für Dynamische Aktionen.* Diplomarbeit, Universität Bonn, 1995

[Buchmann et al. 1992] A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, F. A. Manola: *A Transaction Model for Active Distributed Object Systems.* in Database Transaction Models for Advanced Applications. San Mateo, Morgan Kaufmann, 123-158, 1992

[Chrysanthis, Ramamritham 1992] P. K. Chrysanthis and K. R. Ramamritham: *ACTA: The SAGA continues.* in Database Transaction Models for Advanced Applications, Morgan Kaufmann, 349-398, 1992

[Davies 1978] C. T. Davies: *Data Processing Spheres of Control.* IBM Systems Journal 17(2): 179-198, 1978

[Eswaran 1976] K. P. Eswaran et al.: *On the Notions of Consistency and Predicate Locks.* CACM, Vol. 19, No. 11, 1976

[Garcia-Molina, Salem 1987] H. Garcia-Molina and K. Salem: *Sagas.* ACM SIGMOD Conference on Management of Data, 249-259, 1987

[Gray 1978] J. Gray: *Notes on database operating systems.* Lecture Notes in Computer Science 60, Springer, 1978

[Gray 1981] J. Gray: *The transaction concept: Virtues and Limitations.* 7th International Conference on Very Large Database Systems, 1981

[Gray, Reuter 1992] J. Gray and A. Reuter: *Transaction Processing Systems: Concepts and Techniques.* Morgan Kaufmann Publishers, 1992

[Hadzilacos 1988] V. Hadzilacos: *A Theory of Reliability in Database Systems.* Journal of the ACM 35(1): 121-145, 1988

[Härder, Rothermel 1993] T. Härder and K. Rothermel: *Concurrency Control Issues in Nested Transactions.* The VLDB Journal, 39-74, VLDB Endowment, 1993

[Korth, Levy et al. 1990] H. F. Korth, E. Levy and A. Silberschatz: *A Formal Approach to Recovery by Compensating Transactions.* 16th International Conference on Very Large Data Bases, 95-106, Brisbane, Australia, 1990

[Kröger, Mock et al. 1990] R. Kröger, M. Mock, R. Schumann and F. Lange: *An Extensible Architecture Supporting Reliable Distributed Applications.* 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, 156-164, 1990

[Mock 1994] M. Mock: *Aktionsunterstützung für verteilte, kooperative Anwendungen – Konzept und Realisierung.* Dissertation, Universität Bonn, 1994, veröffentlicht als GMD-Bericht Nr. 250, R. Oldenbourg Verlag, 1995

[Mock, Gergeleit 1997] M. Mock and M. Gergeleit: *Cooperative Concurrency Control on the Web.* eingereicht zur ISADS97

[Moss 1981] J. E. B. Moss: *Nested Transactions: An Approach to Reliable Distributed Programming.* PhD. Thesis, MIT/LCS/TR-260, 1981

[Nett 1991] E. Nett: *Supporting Fault Tolerant Distributed Computations.* Habilitationsschrift, Universität Bonn, 1991

[Nett, Grosspietsch et al. 1985] E. Nett, K.-E. Grosspietsch, J. Kaiser, R. Kröger et. al.: *PROFEMO - Design and Implementation of a Fault Tolerant Distributed System Architecture.* GMD-Studie 100, 1985

[Nett, Mock 1995] E. Nett and M. Mock: *How to commit concurrent, non-isolated computations.* Proc. of the 5th ICCC Workshop on Future Trends of Distributed Computing Systems, August 1995

[Nett, Weiler 1994] E. Nett and B. Weiler: *Nested Dynamic Actions - How to Solve the Fault Containment Problem in a Cooperative Action Model.* 13th Symposium on Reliable Distributed Computer Systems, Irvine, 1994

[Reuter 1989] A. Reuter: *ConTract: A means for extending control beyond transaction boundaries.* 2nd International Workshop on High Performance Transaction Systems, 1989

[Ritter et al. 1993] N. Ritter, B. Mitschang, T. Härder, M. Gesmann, H. Schönig: *Capturing Design Dynamics - The CONCORD Approach.* Forschungsbericht Nr. 24/93, Universität Kaiserslautern, 1993

[Ritter et al. 1994] N. Ritter, B. Mitschang, T. Härder, U. Nink: *Unterstützung der Ablaufsteuerung in Entwurfsumgebungen durch Versionierung und Konfigurierung.* Softwaretechnik in Automation und Kommunikation (STAK'94), Ilmenau, 1994

[Salem, Garcia-Molina et al. 1989] K. Salem, H. Garcia-Molina and R. Alonso: *Altruistic Locking: A Strategy for Coping with Long Lived Transactions.* in High Performance Transaction Systems. Springer Verlag, 175-199, 1989

[Salem, Garcia-Molina et al. 1992] K. Salem, H. Garcia-Molina and J. Shands: *Altruistic Locking.* ACM Transactions on Database Systems 17(1): 117 - 165, 1992

[Sander 1994] G. Sander: *Graph Layout through the VCG Tool.* Technical Report A03/94. Fachbereich Informatik, Universität des Saarlandes, 1994

[Sander 1995] G. Sander: *Graph Layout through the VCG Tool.* in Tamassia, Roberto, Tollis, Ionnis (Eds.), Graph Drawing, DIMACS International Workshop GD´94: Lecture Notes in Computer Science 894, pp. 194-205, Springer Verlag 1995

[Theisohn 1996] P. Theisohn: *Abhängigkeitsverwaltung in verteilten Systemen.* Diplomarbeit, Universität Bonn, 1996

[Traiger 1983] L. L. Traiger: *Trends in System Aspects of Database Management.* 2nd International Conference on Database Theory, Cambrige (UK), 1983

[Waechter, Reuter 1990] H. Waechter, A. Reuter: *Grundkonzepte und Realisierungsstrategien des ConTract-Modells.* Informatik Forschung und Entwicklung 5: 202-212, 1990

[Waechter, Reuter 1992] H. Waechter, A. Reuter: *The ConTract Model.* in Database Transaction Models for Advanced Applications. San Mateo, Morgan Kaufmann, 219-264, 1992

[Weihl 1988] W. Weihl: *Commutativity-Based Concurrency Control for Abstract Data-Types.* IEEE Transactions on Computers 37(12), 1988

[Weikum 1986] G. Weikum: *Transaktionsverwaltung in Datenbanksystemen mit Schichtenarchitektur.* Dissertation, Technische Hochschule Darmstadt, 1986

[Weikum, Scheck 1992] G. Weikum, H.-J. Scheck: *Concepts and Applications of Multilevel Transactions and Open Nested Transactions.* in Database Transaction Models for Advanced Applications. San Mateo, Morgan Kaufmann, 515-554, 1992

[X/Open Company 1993] X/Open Company: *X/Open Guide: Distributed Transaction Processing Reference Manual Version 2.* Berkshire, X/Open Company Limited, 1993

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, die vorliegende Arbeit

Ein kooperatives Sperrverfahren für geschachtelte dynamische Aktionen

selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht zu haben.

Bonn, den 20. März 1997 Armin Kruse