Korrektheit und deren Durchsetzung im Umfeld langdauernder Abläufe

Von der Fakultät für Informatik der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von
Dipl.-Inform. Friedemann Schwenkreis
aus Stuttgart

Hauptberichter:	Prof. Dr.- Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr.- rer.nat. Frank Leymann
Betreuung	Prof. Dr.- Ing. Andreas Reuter

Tag der mündlichen Prüfung: 18. Juni 2001

Institut für Parallele und Verteilte Höchstleistungsrechner
der Universität Stuttgart

Vorwort und Danksagung

Vorwort

Diese Arbeit ist während meiner fünf-jährigen Tätigkeit am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart entstanden. Ein Kerngebiet mit dem sich diese Arbeit beschäftigt - dem *Workflow-Management* - befand sich dabei noch in der Entstehungsphase. Trotzdem war bereits in dieser Zeit die Relevanz der Workflowthematik allgemein anerkannt und vielfältige Aktivitäten sowohl im Forschungs- als auch im Produktbereich zu verzeichnen.

Obwohl sich mein Spezialgebiet nur mit einem kleinen Ausschnitt des Gebietes beschäftigt, hatte ich die Möglichkeit mir einen weiteren Überblick zu verschaffen, was teilweise auch in dieser Arbeit zum Ausdruck kommen soll. So hat sich z.B. durch die aktive Teilnahme an Treffen der *Workflow Management Coalition* (WfMC) und der Leitung des Projektes PoliFlow herausgestellt, daß es noch sehr viele Probleme zu lösen gilt bevor durch Workflow-Management-Systeme die angestrebten Verbesserungen tatsächlich zum Tragen kommen können.

Danksagung

Natürlich wäre diese Arbeit nicht ohne die Unterstützung einer Vielzahl von Personen zustande gekommen. Mein Dank gilt allen von ihnen, auch wenn ich an dieser Stelle nicht alle gesondert erwähnen kann.

Zunächst möchte ich meiner Lebensgefährtin besonderen Dank für ihre Geduld und ihr Verständnis aussprechen, da sie direkt unter dem Aufwand für diese Arbeit zu leiden hatte. Ebenso wichtig ist es mir, meinen Eltern zu danken, welche die Voraussetzung für diese Arbeit geschaffen haben.

Außerdem ist es mir ein besonderes Anliegen meinem “Doktorvater”, Herrn Professor Andreas Reuter zu danken. Durch seine Beziehungen zu Wissenschaftlern in aller Welt, war es mir möglich selbst Kontakte zu knüpfen, die diese Arbeit befruchteten.

Desweiteren möchte ich allen meinen Kollegen am IPVR der Universität für die angenehme Arbeitsumgebung und die vielen fachlichen Diskussionen danken. Insbesondere sei hier Herr Dipl.-Inform. Kutschera erwähnt, der als mein Zimmerkollege hiervon die Hauptlast zu tragen hatte.

Nicht zuletzt gilt mein Dank meinen Kollegen bei der IBM Deutschland Entwicklung GmbH. Ohne die angenehme Arbeitsatmosphäre, die maßgeblich durch

Herrn Dr. Arning und Herrn Dr. Bollinger geschaffen wurde, wäre diese Arbeit nicht zu einem Abschluß gekommen.

Darüber hinaus richte ich meinen Dank in herzlichster Weise auch an die Berichter dieser Dissertation, an Herrn Professor Mitschang und an Herrn Professor Leymann. Ohne deren intensive Mithilfe hätte diese Arbeit nicht zum angestrebten Abschluß kommen können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbolverzeichnis

a^t,a(t): Abort-Operation einer Transaktion t.

α^s: Abort-Operation einer Sub-transaktion s.

$\tilde{a}$: Step-Instanz eines Steps a.

b^t: Begin-Operation einer Transaktion t.

β^s: Begin-Operation einer Sub-Transaktion s.

C(o): ConTract-Bezeichner einer Operation o.

c^t, c(t): Commit-Operation der Transaktion t.

χ^s: Commit-operation einer Sub-Transaktion s.

ς_C: Kontext (Menge der Kontextvariablen) eines ConTract C.

E_C: Menge der Ereignisse einnes ConTract C.

E^i: Menge der internen Ereignisse.

E^e: Menge der externen Ereignisse.

ε(t,p): Establish-Operation eines Prädikates p innerhalb der Transaktion t.

f^C: End-Of-ConTract-Operation eines ConTract C.

γ(t,p): Check-Operation für ein Prädikat p innerhalb der Transaktion t.

H: Historie.

I(t): Interpretation einer Transaktion t.

I_C: Menge der Eingangsinvarianten eines ConTract C.

k^C: Kompensationsanforderung für einen ConTract C.

K_C: Menge der Kompensationsblöcke eines ConTract C.

O_C: Menge der Ausgangsinvarianten eines ConTrcat C.

P_C: Menge der Ablaufprädikate eines ConTract C.

Π(s): Menge von Parametern eines Steps s.

R(s): Menge der Resultate eines Steps s.

S_C: Menge der Steps eines ConTract C.

S^a: Menge der Anwendungsorientierten Steps.

S^v: Menge der verwaltungsorientierten Steps.

Σ: Menge von Operationen einer Historie.

T(o): Top-level Transaktionsbezeichner einer Operation o.

t(o): Transaktionsbezeichner einer Operation o.

T_C: Menge der Transitionen eines ConTract C.

τ: Transaktionaler Block.

1 Einleitung

1.1 Motivation

Die Automatisierung von Geschäftsprozessen, die einen Teilbereich der langlebigen Vorgänge darstellen, stellt in den letzten Jahren ein wichtiges Teilgebiet der Informationstechnik dar. Inzwischen werden dabei alle Aspekte, die bei der automatischen Abwicklung der Vorgänge zu berücksichtigen sind unter dem Begriff des *Workflow Management* zusammengefaßt.

Es sind vielfältige Anstrengungen sowohl in der Forschung als auch in der Produktentwicklung zu verzeichnen, die sich mit dem Workflow Management beschäftigen. Kennzeichnend für die Klasse von Abläufen, die als Workflow bezeichnet werden, ist die relativ lange Verweildauer im System (im Vergleich zu klassischen Datenbanktransaktionen) und die im allgemeinen verteilte Ausführung. Vergleichbar ist die Problemstellung mit einem Gebiet des Software-Engineering. Es beschäftigt sich mit den Unterschieden der Programmierung großer im Vergleich zu der Programmierung relativ kleiner Softwareprojekten: *programming in the large versus programming in the small*. Analog wird die Unterscheidung zwischen kurzen, einfachen Ausführungseinheiten und längeren, komplexen Abläufen häufig auf einen Begriff gebracht: *programming in the short versus programming in the long*.

In der Informatik-Forschung werden sowohl im Bereich des Software-Engineering als auch im Bereich der Transaktionsverarbeitung Anstrengungen unternommen, Lösungen für die Probleme des Workflow-Management zu entwickeln. Dabei waren die Ausgangspunkte völlig unterschiedlich und die Arbeiten fast unabhängig voneinander. Der Grund hierfür bestand darin, daß die Ansätze aus dem Bereich des Software-Engineering sich darauf konzentrierten Beschreibungsmechanismen für die Abläufe zu entwickeln, um dadurch die Ablaufstruktur großer Softwareprojekte beschreiben und automatisieren zu können. Im Gegensatz dazu, konzentrierten sich die Anstrengungen aus dem Bereich der Transaktionsverarbeitung darauf, die transaktionalen Garantien der Datenbanktransaktionen, auf langdauernde Abläufe zu übertragen.

Während sich die Forschungsarbeiten im Bereich der Transaktionsverarbeitung hauptsächlich darauf fokusieren, das transaktionale Fehlerverhalten auf Workflows zu übertragen, konzentriert sich diese Arbeit auf die Probleme, die durch die parallele Verarbeitung von Workflows entstehen. Bedenklicher als die wenigen Arbeiten auf diesem Gebiet ist die Diskussion, die über Notwendigkeit der Bearbeitung der Thematik geführt wird. Ausgelöst wurde die Diskussion durch die Un-

tersuchung realer (existierender) Vorgänge, an denen sich eindeutig zeigte, daß z.B. konkurrierende Zugriffe im Realfall gar nicht vorkommen und die Untersuchung der Thematik somit rein akademischer Natur ist. Außer acht gelassen wurde dabei jedoch die Tatsache, daß heutige Abläufe von der menschlichen Arbeitsweise geprägt sind, bzw. für diese entworfen wurden. Ein wichtiger Aspekt dabei ist, daß die gleichzeitige Bearbeitung eines Objekts von mehreren Menschen nur dann nicht chaotisch endet, wenn äußerst strenge Regularien eingeführt werden. Dabei erzwingen die Regularien meistens die serielle Ausführung der Arbeiten. Dies hat sich nicht nur in den tagtäglichen Vorgängen niedergeschlagen, sondern reicht sogar bis in die Gesetzgebung. So unterliegen z.B. Eintragungen in einem Grundbuch zeitlichen Restriktionen, die eine überlappende Änderung bezüglich des gleichen Objektes ausschließen.

Erstaunlicherweise werden diese Maßnahmen nicht als Synchronisationsmechanismen erkannt sondern als ablaufinherent eingestuft. Mit der Einführung einer elektronischen Unterstützung (z.B. durch elektronische Dokumente statt Papier) zeigt sich allerdings sehr schnell, daß dies nicht der Fall ist. Sieht man etwas in die Zukunft, ist leicht zu erkennen, daß die Beschränkungen nicht nur weitgehend unnötig sondern sogar nachteilig sind, da sie eine Parallelisierung der Vorgänge verhindern und somit einer (zeitlichen) Optimierung im Wege stehen. Dies stellt beispielsweise ein Problem dar, wenn ein sogenanntes *Business Process Reengineering* (BPR) durchgeführt werden soll, welches feststellen soll, wie ein Ablauf aufgebaut ist und gleichzeitig eine Optimierung desselben vornehmen soll. Isoliert man dagegen die synchronistationsbezogenen Teile von den Teilen des Ablaufs, die zur Zielerreichung notwendig sind und beschränkt die Synchronisation auf das Notwendigste (was von den eingesetzten Technologien und den Korrektheitskriterien abhängt), so ist man in der Lage das volle Optimierungspotential auszuschöpfen und die Vorgangsbearbeitung je nach Technologieentwicklung anzupassen, ohne die eigentlichen Abläufe ändern zu müssen.

1.2 Umfeld der Arbeit

Die Untersuchung der Probleme im Bereich der zuverlässigen Abwicklung langlebiger Vorgänge ist bereits seit Ende der 80er Jahre ein Themenschwerpunkt der Abteilung Anwendersoftware des Instituts für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart. In diesem Zusammenhang wurde das sogenannte *ConTract-Modell* entwickelt, welches die robuste Abwicklung von Abläufen unter transaktionalen Garantien zum Gegenstand hat. Ansatzweise enthält das Modell bereits Synchronistationsmechanismen, die es gestatten, eine ablaufübergreifende Kontrolle von konkurrierenden Zugriffen durchzuführen.

Die vorliegendende Arbeit konzentriert sich auf die Verfeinerung und Erweiterung

dieses Ansatzes, wobei die Konzepte jedoch auch auf andere vorgangsunterstützende Systeme übertragbar sind. Da während der Entstehung dieser Arbeit die Prototypentwicklung für ein ConTract-verarbeitendes System (*APRICOTS*) weiter voran getrieben wurde, konnte anhand von (gleichwohl akademischen) Beispielen gezeigt werden, welche Vorteile eine korrektheitserhaltende Synchronisation für eine Anwendung hat.

1.3 Einordnung der Arbeit

Die Grundlage für den hier verfolgten Ansatz bilden die aus dem Bereich der Transaktionsverarbeitung stammenden Ansätze und Verfahren zur sogenannten *Concurrency Control*. Da der deutsche Begriff der Synchronisation vielfach auch die zeitliche Abstimmung unabhängiger Abläufe bezeichnet, wird im folgenden der englische Begriff der Concurrency Control verwendet, wenn es gilt, den hier präsentierten Ansatz einzuordnen.

Der Schwerpunkt dieser Arbeit liegt auf der Entwicklung eines Korrektheitsmodells und der Entwicklung von Verfahren zur Sicherstellung der Korrektheit im Umfeld langlebiger Abläufe. Darüber hinaus wird eine Architektur vorgestellt werden, die es erlaubt, die vorgestellten Mechanismen in eine Systemplattform zu integrieren, die eine zuverlässige Abwicklung entsprechender Anwendungen unterstützt.

Obwohl Leistungsmaße wie Durchsatz und Antwortzeit bei jedem klassischen Verfahren zur Concurrency Control in Datenbanken zur Beurteilung herangezogen werden, wird in dieser Arbeit auf dieses Kriterium verzichtet. Der Grund hierfür ist das bereits erwähnte Problem, daß die gelichzeitige Bearbeitung von Objekten durch mehrere Abläufe bisher in den betrachteten Anwendungen nahzu fehlt, bzw. auf Grund fehlender Systemunterstützung nicht möglich ist und somit keine hinreichende Vergleichsbasis zur Verfügung steht. Selbst ein Vergleich mit dem rein seriellen Vorgehen würde nur einen unbefriedigenden Teil bewerten, da manche Arbeitsformen erst durch die hier vorgestellten Mechanismen möglich werden.

1.4 Überblick über die Arbeit

In Kapitel 2 wird ein Beispiel eingeführt, an dem im weiteren Verlauf dieser Arbeit die Kernprobleme des Workflow und die Anwendung der hier vorgestellten Mechanismen veranschaulicht werden. Zum einen soll dabei die systematische Erfassung von Abläufen durch Methoden des Workflow-Management gezeigt werden, und zum anderen wird deutlich gemacht, wie die Trennung von ablauf- und anwendungsbezogener Information erfolgen kann.

Kapitel 3 liefert einen Überblick über Ablaufmodelle im transaktionalen Umfeld. Es wird gezeigt, welche Zusicherungen bezüglich der Semantik von Abläufen durch die Verwendung dieser Modelle gegeben werden und für welche Anwendungen sie sich eignen.

Auf der Basis der eingeführten Modelle werden in Kapitel 4 formale Notationen für dynamische Abläufe eingeführt, da diese die Basis für die Definition von Korrektheitskriterien bilden. Insbesondere wird eine formale Notation zur Definition von Abläufen des ConTract-Modells detailliert vorgestellt werden.

Mit dem Thema "Korrektheit" stellt Kapitel 5 mit den Kern dieser Arbeit dar. Nach der Einführung klassischer Korrektheitskriterien und der Diskussion von Ansätzen, die über das klassische Transaktionsmodell hinaus gehen, wird ein Korrektheitskriterium für das ConTract-Modell entwickelt. Anschließend werden die Unterschiede zwischen den verschiedenen Ansätzen herausgearbeitet.

Kapitel 6 diskutiert Methoden, welche dazu entworfen wurden, um in Laufzeitsystemen die Verletzung von Korrektheitskriterien zu verhindern. Hierbei werden zunächst die Grundprobleme bei der Umsetzung von Korrektheitskriterien vorgestellt, bevor konkrete Ansätze im Bereich der sogenannten Synchronisierungsmethoden (engl. *Concurrency Control*) betrachtet werden. Wie in Kapitel 5 wird der Ansatz des ConTract-Modells eingehend besprochen und in einem abschließenden Abschnitt mit den anderen Ansätzen verglichen.

Kapitel 7 beschreibt die Umsetzung der in der Arbeit entwickelten Ansätze im Hinblick auf die Einbettung in das bestehende ConTract-Modell, als auch bezüglich der Realisierung in einer prototypischen Implementierung. Insbesondere wird deutlich werden, welche Erweiterungen an der Architektur der prototypischen Implementierung vorgenommen werden müssen und welche Auswirkungen durch die Einführung eines Korrektheitskriteriums und entsprechender Durchsetzungsmechanismen entstehen.

Abgeschlossen wird die Arbeit mit Kapitel 8, welches zunächst eine Zusammenfassung der Arbeit und eine kurze Diskussion der vorgestellten Mechanismen vornimmt. Dabei soll aus heutiger Sicht dargestellt werden, welche Probleme noch offen bzw. Gegenstand laufender Forschungsarbeiten sind.

2 Ein einleitendes Anwendungsbeispiel

2.1 Motivation

Trotz der vielfältigen Einsatzgebiete von Workflowsystemen finden sich in der (wissenschaftlichen) Literatur fast nur Trivialbeispiele, die kaum dazu geeignet erscheinen, die komplexen Anforderungen der Anwendungsbereiche zu repräsentieren. Aus diesem Grund soll in diesem Kapitel zunächst ein detailliertes Beispiel eingeführt werden, um so zum einen den Bedarf für die hier vorgestellten Verfahren zu motivieren und andererseits die Anwendbarkeit der Ansätze demonstrieren zu können.

Das für diese Arbeit gewählte Beispiel des universitären Urlaubsantrages ist ebenfalls nicht sonderlich komplex, wenn man es mit anderen Vorgängen wie zum Beispiel einem Kreditantrag vergleicht. Trotzdem erfüllt es aus Sicht dieser Arbeit alle notwendigen Kriterien, um zum einen die Probleme zu verdeutlichen, als auch die Anwendbarkeit der in dieser Arbeit entwickelten Verfahren zu demonstrieren.

Zunächst ist hier die genaue Kenntnis über den Vorgang selbst ein wichtiges Auswahlkriterium[1]. Erst durch die Kenntnis der Details eines Vorgangs ist die Erfassung desselben mit Methoden des Workflow Managements möglich. Darüber hinaus sind die Abwicklung von Urlaubsanträgen und Reisekostenabrechnungen durchaus übliche Anwendungen für die Einführung von Workflowsystemen.

An dem Beispiel wird außerdem deutlich, daß die parallele Abwicklung von Vorgängen ein Problem der Realwelt darstellt, da mehrere Mitarbeiter und Mitarbeiterinnen völlig unabhängig voneinander und gleichzeitig Urlaubsanträge stellen können. Zusätzlich ist es möglich, daß ein Mitarbeiter mehrere Anträge gleichzeitig stellt, was ebenso in den Bereich der Parallelverarbeitung fällt.

Da die Umsetzung der hier vorgestellten Verfahren in einer prototypischen Implementierung mit als Ziel verfolgt wurde, war sowohl die Implementierbarkeit des Beispielablaufs als auch seine konkrete Unterstützung durch das prototypische System ein weiteres wichtiges Auswahlkriterium.

Nicht zuletzt erfolgte die Auswahl des Beispiels auf Grund der relativ einfachen Darstellbarkeit und Verständlichkeit des Vorgangs. Erst hierdurch kann das Beispiel zur Erläuterung der Verfahren herangezogen werden.

1. Der Verfasser glaubt, diese Kenntnis zu besitzen.

2.2 Der universitäre Urlaubsantrag

2.2.1 Überblick

Oberflächlich betrachtet stellt ein Urlaubsantrag einen nahezu trivialen Vorgang dar, der mit ein paar Sätzen beschrieben werden kann:

1. Ein Mitarbeiter oder eine Mitarbeiterin füllt einen Urlaubsantrag aus.
2. Ein Stellvertreter unterschreibt (optional).
3. Der Abteilungsleiter genehmigt den Antrag per Unterschrift oder lehnt ihn ab.
4. Der Geschäftsführende Direktor zeichnet den Antrag ab.
5. Die Geschäftsleitung vermerkt die Anzahl der genommenen Urlaubstage in der Urlaubskartei.

Wird allerdings versucht, diesen Vorgang rechnergestützt abzuwickeln, stellt sich heraus, daß in dieser Beschreibung einige implizite Annahmen enthalten sind. Beispielsweise nimmt ein Abteilungsleiter an, daß ein Mitarbeiter nur dann Urlaub beantragt, wenn derjenige noch Urlaubstage zur Verfügung hat. Eine Verletzung dieser Annahme wird erst dann entdeckt, wenn die Geschäftsleitung versucht, die Urlaubskartei auf den neuesten Stand zu bringen.

Ein weiteres Problem der dargestellten Verfahrensweise stellt die Nichtbeachtung von Ausnahmefällen dar. Beispielsweise wird nicht beschrieben was passieren soll, wenn sich der Abteilungsleiter im Urlaub befindet. Außerdem endet die Beschreibung des Vorgangs bereits mit der Übernahme der Daten in die Urlaubskartei unter der Annahme, daß der Urlaub angetreten und vollständig durchgeführt wird. Tritt allerdings der Ausnahmefall auf, daß der Urlaub z.B. wegen Krankheit nicht angetreten oder nur teilweise durchgeführt werden kann, spiegeln die Daten des ursprünglichen Urlaubsantrages die tatsächlichen Gegebenheiten nicht wider. Somit sind weitere Aktionen notwendig, um die Daten entsprechend zu korrigieren. Diese Korrektur muß allerdings nicht Teil desselben Vorgangs sein.

2.2.2 Separation der Einzelaspekte

Wie bereits aus Abschnitt 2.2.1 deutlich wird, kann eine textuelle Beschreibung eines Vorgänge im Sinne eines Ablaufs sehr unübersichtlich werden. Dies rührt teilweise daher, daß sich Vorgänge durch eine textuelle Repräsentation nicht sehr kompakt darstellen lassen. Andererseits liegt ein Grund dafür in der zusammenfassenden Beschreibung aller Aspekte eines Vorganges.

Wie in [Jab95] dargestellt, läßt sich die Beschreibung eines Vorganges in verschiedene orthogonale Aspekte aufteilen. Neben der mehr anschaulichen Darstellung eines Vorgangs hat diese Aufteilung den weiteren Vorteil, daß man sich bei der Beschreibung auf einen bestimmten Aspekt konzentrieren kann und somit weniger die Gefahr eingeht Informationen nicht zu erfassen.

Da die Einführung des Beispiels nur für die Motivation und Erläuterung der in dieser Arbeit vorgestellten Verfahren dienen soll, werden nicht alle Aspekte detailliert dargestellt werden, die von Jablonski beschrieben wurden.

2.2.2.1 Funktionaler Aspekt

Unter den funktionalen Aspekten eines Vorgangs versteht man die Zerlegung des Gesamtvorganges in Teile, die für die gewünschte Funktionalität notwendig sind. In unserem Beispiel kann der Vorgang “Urlaubsantrag” in folgende funktionalen Teile zerlegt werden (es werden bereits alle Teile erfaßt, die für eine Automatisierung notwendig sind):

1. Dokumentenbereitstellung
2. Antragstellung
3. Vertretungszusage
4. Genehmigung
5. Überprüfung der Zulässigkeit
6. Änderung der Urlaubskartei
7. Rückmeldung durch Beantragenden
8. Änderungserfassung
9. Benachrichtigung

In dieser Zerlegung werden weder die Reihenfolge der Ausführung noch die Ausführungsinstanz festgelegt.

2.2.2.2 Verhaltensbezogener Aspekt

Die verhaltensbezogenen Aspekte berücksichtigen die Ablaufstruktur eines Vorganges (*Kontrollfluß*). Damit sind im allgemeinen Vorgänger/Nachfolger-Beziehungen sowie Verzweigungen gemeint. Üblicherweise wird der Kontrollfluß entweder in einer textuellen Notation mit Hilfe einer Programmiersprache oder in einer graphischen Form festgelegt. An dieser Stelle soll diese Festlegung mit Hilfe einer Programmiersprache erfolgen. Allerdings ergeben sich daraus auch Proble-

me. So wird z.B. durch die zeilenorientierte Schreibweise automatisch impliziert, daß die einzelnen Zeilen sequentiell abzuarbeiten sind. Ist es dagegen möglich, Teile parallel auszuführen, muß dies explizit angegeben werden (z.B. durch das Schlüsselwort **parallel**). Umgekehrt muß in einem parallel ausführbaren Teil die sequentielle Bearbeitung ausdrücklich festgelegt werden (z.B. durch ein Schlüsselwort **sequentiell**). Da an dieser Stelle nur das Verständnis des Beispiels im Vordergrund steht, wird hier keine Sprachdefinition eingeführt. Hierfür sei auf die Definitionen in [ZiCh91], [Wäch96] und [Jab95] verwiesen.

```
Beginn
Dokumentenbereitstellung
Antragstellung
Überprüfung der Zulässigkeit
Wenn nicht zulässig
    Benachrichtigung über Unzulässigkeit
sonst
    Vertretungszusage einholen
    Genehmigung einholen
    Wenn nicht genehmigt
        Benachrichtigung über Ablehnung
    sonst
        Genehmigung einholen
        Wenn nicht genehmigt
            Benachrichtigung über Ablehnung
        sonst
            Änderung der Urlaubskartei
            Benachrichtigung über Genehmigung
Ende
```

Abbildung 2-1: Ein Kontrollfluß-Beispiel

Wie aus dem Code-Stück in Abbildung 2-1 ersichtlich ist, kann ein Kontrollfluß mit bedingten Verzweigungen nicht ohne zusätzliches Wissen über einen anderen Bereich definiert werden, nämlich den informations- oder datenbezogenen Aspekt (s. Abschnitt 2.2.2.3). Da bedingte Verzweigungen im allgemeinen auf Prädikaten basieren, die wiederum über Variablen definiert werden, ist der Bezug auf Daten, die während der Ausführung erzeugt bzw. bekannt werden, notwendig.

2.2.2.3 Informationsbezogener Aspekt

Als informationsbezogenen Aspekt oder *Datenfluß* bezeichnet man diejenigen Teile einer Workflow-Definition, die festlegen, wie Daten in den Vorgang einfließen bzw. von einer Aktivität zur nächsten weitergeleitet werden. Grundsätzlich lassen sich dabei *workflow-relevante* und *workflow-fremde* Daten [WFM94] unterscheiden. Der Begriff workflow-relevant bezeichnet Daten, die einen direkten Einfluß

auf den Kontrollfluß haben. Workflow-fremde Daten haben im Gegensatz dazu keinen direkten Einfluß auf den Kontrollfluß.

Ein Beispiel für ein workflow-relevantes Datum, ist eine Änderungsanzeige welche in einem Verzweigungs-Prädikat auftritt. In diesem Fall haben Daten einer Aktivität direkten Einfluß auf den Kontrollfluß und sind somit relevant für den Ablauf. Daten, welche nur von einer Aktivität zu einer anderen weiter geleitet werden sind demgegenüber workflow-fremd (wie z.B. der Antrag in unserem Beispiel).

Wie sich im weiteren Verlauf dieser Arbeit noch herausstellen wird, ist es notwendig, workflow-fremde Daten noch weiter zu unterscheiden. Das Unterscheidungskriterium ist dabei, ob das Workflowsystem die vollständige Kontrolle über die Daten besitzt (*lokale Daten*) oder nicht (*globale Daten*). In unserem Beispiel sind die Daten in der Urlaubskartei nicht unter vollständiger Kontrolle des Ablaufs, während der Urlaubsantrag dieser Bedingung genügt.

Da die eigentliche Spezifikation des Datenflusses hier nicht weiter von Interesse ist, wird nur ein kurzes Beispiel gegeben, wie dieser festgelegt werden könnte. Die Notation lehnt sich an die in [Wäch96] gebrauchte an.

```
Aktivität:          Dokumentenbereitstellung(
Parameterart:       OUT
Parametertyp:       File
Parameter:          Neuer_Antrag:
Abstrakte Variable: Antragsdokument)

Aktivität:          Antragsstellung(
Parameterart:                 INOUT
Parametertyp:       File
Parameter:          Antrag:
Abstrakte Variable: Antragsdokument)
```

Abbildung 2-2: Schnittstellendefinition für den Datenfluß

Abbildung 2-2 zeigt, wie Daten, die von einer Aktivität bereit gestellt werden, einer anderen Aktivität als Eingabeparameter zugeordnet werden. Hierbei wird eine Stufe der Indirektion eingeführt, um von den Schnittstellenparametern der einzelnen Aktivitäten zu abstrahieren:

- Die Aktivität `Dokumentenbereitstellung` hat einen Ausgabeparameter vom Typ `File`. Dieser ist mit `Neuer_Antrag` bezeichnet und wird einer abstrakten Variablen `Antragsdokument` des Vorgangs zugeordnet.
- Die Aktivität `Antragstellung` hat einen Ein-/Ausgabe-Parameter der ebenfalls vom Typ `File` ist. Ihm wird ebenfalls die abstrakte Variable `An-`

`tragsdokument` zugeordnet, um so die Daten der ersten Aktivität zu dieser Aktivität übertragen zu können.

2.2.2.4 Der transaktionale Aspekt

Die transaktionale Aspekte eines Ablaufs lassen sich in drei Bereiche unterteilen:

1. Beschreibung der Isolationsbedürfnisse
2. Beschreibung des Verhaltens im Fehlerfall[1]
3. Beschreibung des Rücksetzverhaltens

Die Beschreibung der Isolationsbedürfnisse legt fest, wann und wie Datenobjekte, die von dem Ablauf geändert oder gelesen wurden, für andere Abläufe verfügbar sind. Im Falle einer Datenbanktransaktion ist beispielsweise ein geändertes Datenobjekt erst nach Abschluß einer Transaktion für andere Transaktionen zugänglich.

Das Verhalten im Fehlerfall wird im allgemeinen durch zwei Mechanismen beschrieben. Zum einen wird festgelegt, welche Zustände des Ablaufs (im Sinne des Fortschritts im Kontrollfluß) als konsistent angesehen werden. Zum anderen werden Erweiterungen des Kontrollflusses zur Behandlung von Fehlern vorgenommen. Somit muß im Fehlerfall zunächst auf den zuletzt erreichten konsistenten Zustand zurückgesetzt und danach die Fehlerbehandlung eingeleitet werden. Dies entspricht dem Verhalten von Datenbanksystemen nach einem System-Crash, wobei in einer Redo-Phase zunächst ein konsistenter Zustand hergestellt wird und anschließend in einer Undo-Phase nicht abgeschlossene Transaktionen zurückgesetzt werden.

Die Beschreibung des Rücksetzverhaltens legt fest, wie im Kontrollfluß auf einen früheren Zustand “zurückgesetzt” werden kann. Dabei ist der Begriff des Zurücksetzens sehr allgemein zu verstehen. Er soll nur bedeuten, daß aus Sicht des Kontrollflusses auf einen Punkt zurück gegangen wird, der vor dem aktuell erreichten liegt (im Sinne der Partialordnung der Einzelschritte des Kontrollflusses). Bei einem linearen Kontrollfluß bedeutet dies, daß auf einen vormalig erreichten Punkt bzw. Verarbeitungszustand zurückgesetzt wird. Liegt allerdings ein verzweigter Kontrollfluß vor, muß dies nicht unbedingt der Fall sein.

Üblicherweise werden zwei Mechanismen für das Zurücksetzen verwendet. Ein Mechanismus ist das atomare Zurücksetzen oder “Undo”. Das atomare Zurücksetzen eliminiert alle relevanten Effekte einer Ausführung und kommt bei Datenbanktransaktionen zum Einsatz.

1. Unter dem Begriff Fehlerfall ist hier das Fehlschlagen eines Schrittes im Kontrollfluß zu verstehen.

Die Verallgemeinerung des atomaren Zurücksetzens ist die sogenannte Kompensation. Bei der Kompensation wird festgelegt, welche Aktionen auszuführen sind, um auf einen früheren Zustand im Kontrollfluß zurückzusetzen. Dies stellt im allgemeinen die Definition eines (Teil-)Kontrollflusses dar, der sich nur dadurch vom "normalen" Kontrollfluß unterscheidet, daß er als Kompensation deklariert wird.

Betrachtet man nun die transaktionalen Aspekte bezüglich des eingeführten Beispiels, könnte man fordern, daß die einzelnen Schritte als Datenbanktransaktionen auszuführen sind. Ist allerdings eine Aktion bereits erfolgreich abgeschlossen, so soll eine Gegenaktion (Kompensation) ausgeführt werden, wenn auf einen früheren Zustand zurückgesetzt werden soll (beispielsweise bei einer Stornierung). Tabelle 2-1 gibt einen Überblick über entsprechende Gegenaktionen.

Aktivität	Gegenaktivität
Dokumente bereitstellen	leer
Antragstellung	leer
Überprüfung der Zulässigkeit	leer
Vertretungszusage einholen	Benachrichtigung über Stornierung
Genehmigung einholen	Benachrichtigung über Stornierung
Benachrichtigung (positiv)	Benachrichtigung über Stornierung
Änderung der Urlaubskartei	Änderung der Urlaubskartei

Tabelle 2-1: Aktivitäten und ihre Gegenaktivitäten

3 Transaktionale Ausführungsmodelle

Da im Rahmen dieser Arbeit Abläufe im transaktionalen Umfeld betrachtet werden sollen, ist es notwendig die grundsätzlichen Unterschiede der Ausführungsmodelle für langlebige Abläufe im Vergleich zu klassischen Datenbank-Transaktionen (DB-Transaktionen) herauszuarbeiten. Hierzu soll zunächst auf die Eigenschaften der sogenannten *ACID-Transaktionen* [HäRe83] eingegangen werden. Daran anschließend werden Ausführungsmodelle vorgestellt, die entworfen wurden, um Nachteile der ACID-Transaktionen im Falle langlebiger Abläufe zu vermeiden. Da das ConTract Modell [WäRe92] die Grundlage dieser Arbeit bildet, wird diesem ein eigener Abschnitt gewidmet werden.

Einige Ausführungsmodelle werden hier nicht vorgestellt, da sie bezüglich den vorgestellten Ansätzen keine neuen Aspekte beinhalten. Gute Übersichten und ausführlichere Darstellungen sind in [Elm92] und [Günt96] zu finden.

3.1 Klassische DB-Transaktionen

Mit der Einführung elektronischer Datenverarbeitung auf der Basis von Datenbanksystemen wurde der Begriff der DB-Transaktion geprägt. Dabei ist mit dem Begriff *Transaktion*[1] die Zusammenfassung von Operationen auf einem Datenbanksystem (und Nachrichten) zur Durchführung einer bestimmten Aktion gemeint. Das klassische Beispiel hierfür ist die Abwicklung einer (elektronischen) Überweisung, die einen Kontostand A erniedrigt und danach einen anderen Kontostand B um den entsprechenden Betrag erhöht. Wie einfach nachzuvollziehen ist, sind dabei vier Datenbankzugriffe zu einer Transaktion - der Überweisung - zusammengefaßt:

1. Lies Kontostand A
2. Erniedrige / Schreibe A
3. Lies B
4. Erhöhe / Schreibe B

Wie bereits an dem einfachen Beispiel abzulesen ist, gibt es nur zwei Basisoperationen, die unterschieden werden: das Lesen bzw. das Schreiben von Datenelementen. Eine Transaktion kann somit als eine einfache Sequenz von Lese- und Schrei-

1. Im Folgenden wird in diesem Abschnitt statt DB-Transaktion immer die verkürzte Schreibweise “Transaktion” bzw. “TA” benutzt.

boperationen angesehen werden [EGL76]. Diese Auffassung von Transaktionen wurde auch unter dem Begriff des *Lese-/Schreibmodells* (read/write model) bekannt [Papa86] [BHG87].

3.1.1 Grundprobleme von DB-Transaktionen

Ist mit der obigen Einführung von Transaktionen keine zusätzliche Semantik verbunden stellt eine Transaktion einzig und allein die syntaktische Gruppierung von Basisoperationen dar. Obwohl es Transaktionen ohne zusätzliche Semantik nie gegeben hat, läßt sich bereits an der rein syntaktischen Gruppierung von Operationen der Bedarf für weitere Forderungen motivieren:

1. Kommt es während der Ausführung einer Transaktion zu Systemausfällen, müssen geeignete Maßnahmen getroffen werden, um eine einmal begonnene Transaktion zu Ende führen zu können (erfolgreich oder nicht). Ein Zwischenzustand ist, wie am Beispiel der Überweisung ersichtlich, in den meisten Fällen ein inkonsistenter Zustand der Datenbank bezüglich aktiver Transaktionen [BHG87].
2. Die parallele Ausführung von Transaktionen führt zu Problemen, wenn auf gleiche Daten zugegriffen wird. So kann die Ausführung von für sich genommen korrekten Transaktionen im Parallelbetrieb zu inkonsistenten Datenbeständen führen [EGL76]. Formalisiert man die zu Grunde liegende Problematik auf der Basis des Lese-Schreibmodells, gelangt man zu drei grundsätzlichen Abhängigkeiten (“the three bad dependencies” [GrRe93] [BHG87]), die die Ursache von Inkonsistenzen bilden.

3.1.2 Die ACID Eigenschaften

Wie aus der Beschreibung in Abschnitt 3.1.1 hervorgeht sind einige Probleme bei der Programmierung von Transaktionen grundsätzlich gegeben und müssen prinzipiell bei jeder Anwendung berücksichtigt werden, die Transaktionen verwendet. Deshalb bietet es sich an, die notwendigen Maßnahmen zur Vermeidung eben dieser Probleme nicht immer wieder im jeweiligen Anwendungsprogramm zu realisieren, sondern diese in einem Laufzeitsystem zu verankern und durch simple Programmierprimitive für die Anwendung zugänglich zu machen. Dies hat die Vorteile, daß zum einen Anwendungsprogrammierer entlastet werden und zum anderen Spezialisten auf diese Problematik angesetzt werden können, um so eine effiziente Realisierung zu gewährleisten.

Dieser Ansatz wurde durch die Einführung der sogenannten ACID-Eigenschaften [HäRe83] für Transaktionen realisiert. Neben den oben genannten Problemen wurden dabei gleich zwei weitere Aspekte mit berücksichtigt.

3.1.2.1 Atomarität oder Ununterbrechbarkeit (A)

Um zu vermeiden, daß Transaktionen bei Systemausfällen oder sonstigen Unterbrechungen, wie z.B. einem Zusammenbruch einer Kommunikationsverbindung, teilweise Effekte im (Datenbank-) System hinterlassen, wurde die Ununterbrechbarkeit eingeführt. Ein transaktionsverarbeitendes System garantiert, daß eine einmal gestartete Transaktion entweder vollständig abgearbeitet wird, oder daß ihre Effekte rückgängig gemacht werden (*backward recovery*).

3.1.2.2 Konsistenzerhaltung (C)

Die Eigenschaft der Konsistenzerhaltung (engl. Consistency) läßt sich nicht direkt aus den Problemen von Abschnitt 3.1.1 ableiten und wird auch nicht ausschließlich von einem Laufzeitsystem realisiert. Vielmehr besagt diese Eigenschaft, daß eine Transaktion, wenn sie auf einem konsistenten Zustand (der Datenwelt) gestartet wird, wiederum einen konsistenten Zustand hinterläßt. Somit ergibt sich zum einen die Anforderung an einen Transaktionsprogrammierer, eine Transaktion so zu programmieren, daß sie dieser Anforderung genügen kann. Zum anderen hat das Laufzeitsystem zu überwachen, ob Transaktionen definierte Konsistenzbedingungen (engl. *constraints*) verletzen. Ist dies der Fall, müssen die Transanktionen, die eine Konsistenzbedingung verletzen zurückgesetzt werden.

3.1.2.3 Isolation (I)

Durch die parallele Ausführung von Transaktionen können Daten-Inkonsistenzen entstehen. Deshalb wird für DB-Transaktionen die Isolationseigenschaft gefordert. Das Laufzeitsystem garantiert hierbei, daß jede Transaktion in einer "virtuellen Ein-Benutzer-Umgebung" ausgeführt wird. Somit sind ungewünschte Seiteneffekte der Parallelverarbeitung von Transaktionen ausgeschlossen.

3.1.2.4 Dauerhaftigkeit (D)

Für erfolgreich abgeschlossene Transaktionen wird garantiert, daß deren Ergebnisse nicht verloren gehen. Dies bedeutet konkret, daß die Verantwortung für die Wiederherstellung des durch abgeschlossene Transaktionen erzeugten Zustandes nach einem Systemausfall, Gerätefehler o.ä. beim Transaktionssystem und nicht bei der Anwendung liegt.

3.1.3 Einsatzgebiete

ACID-Transaktionen haben sich auf dem Gebiet der Datenbanken als Programmierkonstrukt durchgesetzt. Durch eine einfache Klammerung von Datenbankoperationen mittels "Begin-Of-Transaction" (BOT) und "End-Of-Transaction" (EOT) erhält man weitreichende Zusicherungen, die durch das Laufzeitsystem realisiert

werden. In einigen Datenbanksystemen entfällt sogar die “öffnende” Klammer BOT, da implizit eine Transaktion begonnen wird, wenn der erste Datenbankzugriff erfolgt (*chained* / *unchained paradigm* [OSI92]).

Allerdings hat es sich bereits sehr früh gezeigt, daß sich ACID-Transaktionen nur für den Einsatz bei relativ kurzen Operations-Sequenzen eignen, die außerdem nur relativ wenige Datenelemente bearbeiten [Gra81a]. Der Grund hierfür ist die Isolationseigenschaft. Um die Isolation einer Transaktion zu garantieren, muß der Zugriff auf Datenelemente, die von der Transaktion verwendet werden, für andere Transaktionen weitgehend eingeschränkt werden. Dies resultiert in der Nichtverfügbarkeit von Datenelementen und somit in einer Blockierung anderer Transaktionen.

Es läßt sich einerseits keine allgemeine Schranke ermitteln, die bestimmen würde, wann sich eine ACID-Transaktion für den Einsatz eignet und wann nicht. Andererseits zeigen theoretische Untersuchungen, daß selbst mit den leistungsfähigsten Implementierungen der ACID-Eigenschaften, die Wahrscheinlichkeit für eine Verklemmung bzw. für den Abbruch einer Transaktion überproportional zur Verweilzeit (quadratisch) bzw. zur Anzahl der angefaßten Datenobjekte (proportional zur dritten Potenz) steigt [Gra81b] [Reut96].

Diese nachteiligen Effekte der ACID-Eigenschaften lassen sich ebenfalls beobachten, wenn relativ viele Transaktionen auf das gleiche Datenelement (ändernd) zugreifen und somit ein Datenelement zu einem sogenannten “Hot Spot” wird [Reut82] [PRS88]. Darüber hinaus ermöglichen DB-Produkte die Einschränkung der Isolationseigenschaft, so daß eine Erhöhung des Durchsatzes erreicht werden kann [GrRe93]. Allerdings setzt die Anwendung dieser Maßnahme eine genaue Kenntnis der Zugriffsoperationen voraus.

3.2 Transaktionen und Verkettung

Transaktionen, die durch Verkettung von Teiltransaktionen aufgebaut sind, stellen keine Ausführungsmodelle im eigentlichen Sinne dar. Vielmehr sind sie Programmiertechniken, um transaktionale Mechanismen im Sinne der ACID-Eigenschaften auch im langlebigen Fall ohne größere Nachteile nutzen zu können. Ziel der verketteten Transaktionen ist grundsätzlich eine Folge von ACID-Transaktionen auszuführen ohne der Atomaritätseigenschaft zu unterliegen - sprich, im Fehlerfalle nicht auf den Beginn der Transaktion zurücksetzen zu müssen sondern nur auf den Beginn einer Teiltransaktion. Somit wird der Verlust im Fehlerfalle minimiert.

3.2.1 Mini-Batch und Warteschlangen

Ein Mini-Batch [GrRe93] stellt im Prinzip eine Programmiertechnik dar, die berücksichtigt, daß ein langlebiger Ablauf in kleinere Einheiten (ACID-Transaktionen) zerlegt werden kann. Durch diese Zerlegung wird erreicht, daß für den gesamten Vorgang die Atomaritätsbedingung entfällt. Prinzipiell wird also ein größerer Auftrag (welcher früher immer als Batch-Job ausgeführt wurde) in kleinere Pakete (Mini-Batch) mit ACID-Eigenschaften zerlegt. Eine Anwendung übernimmt dann die Verantwortung für die vollständige Ausführung. D.h. es müssen Daten über den Verarbeitungszustand auf einen ausfallsicheren (stabilen) Speicher geschrieben werden, so daß diese nach einem Systemfehler wieder gelesen werden können und die Verarbeitung fortgesetzt werden kann.

Eng verwandt mit dem Prinzip des Mini-Batch sind die sogenannten stabilen oder wiederherstellbaren Warteschlangen (engl. *recoverable queues*). Das Prinzip ist dabei, daß Aufträge nicht direkt an einen Server gerichtet werden, sondern in eine stabile (transaktionale) Warteschlange gestellt werden. Darüber hinaus wird in jedem Auftrag hinterlegt was als nächstes zu tun ist, so daß dieser als letzte Aktion wieder einen Auftrag in einer Warteschlange hinterlegt.

Sowohl der Mini-Batch als auch die stabilen Warteschlangen geben frühzeitig die Ergebnisse von Teilausführungen preis, so daß neben der Atomarität auch die Isolation verloren geht.

3.2.2 Transaktionsketten

Transaktionsketten (engl. *chained transactions*)[1] verfolgen das Prinzip des sogenannten *persistenten savepoint*. ACID-Transaktionen werden dadurch zu Transaktionsketten, daß anstatt einem üblichen *COMMIT WORK* ein spezieller Befehl zum Transaktionsabschluß benutzt wird: *CHAIN WORK*. Durch diesen Abschluß wird die aktuelle Transaktion beendet und eine neue begonnen. Obwohl die Änderungen der ersten Transaktion stabil gemacht werden, sind die (Teil-)Ergebnisse derselben nur in der unmittelbar folgenden Transaktion sichtbar.

Somit wird die Isolationseigenschaft gewährleistet, während die Atomarität teilweise aufgegeben wird. Aufgegeben deshalb, weil ein Systemausfall nur ein Rücksetzen aktiver Transaktionen bewirkt. Änderungen von Transaktionen, die mit *CHAIN WORK* abgeschlossen wurden sind dagegen dauerhaft.

3.3 Geschachtelte Transaktionen

Grundsätzlich lassen sich geschachtelte Transaktionen in zwei Kategorien eintei-

1. Es besteht die Gefahr, chained transactions mit dem chained paradigm zu verwechseln (siehe 3.1.3).

len. Zum einen in sogenannte *geschlossen geschachtelte* und zum anderen in *offen geschachtelte* Transaktionen. Während geschlossen geschachtelte Transaktionen exakt definiert sind, handelt es sich bei den offen geschachtelten eher um ein generisches Konzept. Deshalb wird das Prinzip der offen geschachtelten Transaktionen nur sehr kurz erläutert werden. Wie sich darüber hinaus herausstellen wird, stellen die im weiteren Verlauf vorgestellten Ablaufmodelle Spezialfälle der offen geschachtelten Transaktionen dar.

3.3.1 Geschlossen geschachtelte Transaktionen

Geschlossen geschachtelte (engl. *closed nested*) Transaktionen [Tra83] [Moss85] wurden entworfen, um das Granulat bei einem Zurücksetzen zu verfeinern. Um das Zurücksetzen einer Transaktion auf einen Teil beschränken zu können, unterteilt man ein Transaktion in weitere sogenannte Sub-Transaktionen (dies kann rekursiv fortgeführt werden). Die äußere Transaktion wird dann als *Top-Level Transaktion* bezeichnet.

Während für eine Top-Level Transaktion immer noch die ACID-Eigenschaften gefordert werden, wird die Dauerhaftigkeit für Sub-Transaktionen (oder Kind-Transaktionen) aufgegeben. D.h. beim Ende einer Sub-Transaktion werden die Kontrollstrukturen an die Elterntransaktion vererbt und erst wenn die Top-Level Transaktion erfolgreich beendet wird, werden die Ergebnisse dauerhaft. Bezüglich des Rücksetzens ergibt sich der Vorteil, daß das Zurücksetzen einer Sub-Transaktion sich nicht auf die Eltern-Transaktion auswirken muß. Allerdings sind beim Zurücksetzen der Eltern-Transaktion alle zugehörigen Sub-Transaktionen betroffen.

Bezüglich der Isolationseigenschaft sind zwei Aspekte erwähnenswert. Die Top-Level Transaktion ist isoliert bezüglich aller anderen Transaktionen, die keine “Nachkommen” von ihr sind. Direkte Sub-Transaktionen haben Zugriff auf die Datenobjekte der Elterntransaktion (Vererbung) und Elterntransaktionen erhalten den Zugriff auf alle Datenobjekte, die von erfolgreich abgeschlossenen Sub-Transaktionen angefaßt wurden. “Geschwister”, d.h. Sub-Transaktionen der gleichen Elterntransaktion, laufen isoliert von einander ab.

Bezüglich der Ablaufstruktur erweitern geschlossen geschachtelte Transaktionen die Möglichkeiten des klassischen Ansatzes. Neben der einfachen Sequenz ist es hier prinzipiell zulässig Sub-Transaktionen parallel ablaufen zu lassen.

3.3.2 Einsatzgebiete geschlossen geschachtelter TA

Geschlossen geschachtelte Transaktionen eignen sich gut für den Einsatz im Client/Server-Umfeld. Auf Grund der separaten Rücksetzbarkeit von Sub-Transaktionen ist es möglich, z.B. bei Ausfall eines Rechnerknotens oder einer Kommu-

nikationsverbindung, Teile einer Transaktion zurückzusetzen ohne daß die Transaktion als Ganzes davon betroffen ist. Besteht darüber hinaus die Möglichkeit die Sub-Transaktion auf einem alternativen Knoten (Replikat-Server) fortzusetzen, kann die Top-Level Transaktion trotz des Ausfalls erfolgreich zu Ende geführt werden.

Trotzdem sind geschlossen geschachtelte Transaktionen bisher kaum im kommerziellen Bereich anzutreffen. Dies liegt zum einen an dem nicht zu vernachlässigen Aufwand für die Implementierung der Isolations- bzw. Vererbungseigenschaften dieses Transaktionstyps. Zum anderen war das Verarbeitungsmodell bisher nicht in Standards zur Transaktionsverarbeitung berücksichtigt [XOP93] [OSI92].

Mit der Verabschiedung des *Object Transaction Services* (OTS) der *Object Management Group* (OMG) ist eine erste Bemühung zu verzeichnen, geschlossen geschachtelte Transaktionen zu standardisieren [OMG96]. Da die Spezifikation geschlossen geschachtelte Transaktionen nur als optionale Erweiterung vorsieht, bleibt jedoch abzuwarten, ob und wann entsprechende Implementierungen verfügbar sein werden.

3.3.3 Offen geschachtelte Transaktionen

Offen geschachtelte Transaktionen [Gra81a][Tra83] wurden zunächst nur als Konzept festgelegt. Ausführungsmodelle im eigentlichen Sinne entstanden erst später auf der Basis dieses Konzepts. Grundsätzlich unterscheiden sich die offen geschachtelten Transaktionen von den geschlossen geschachtelten dadurch, daß es Sub-Transaktionen möglich ist, die von ihnen vorgenommenen Änderungen vor dem Ende der Eltern-Transaktion freizugeben.

Somit entfällt die Isolations- und Atomaritätseigenschaft für Top-Level Transaktionen und dadurch die einfache Rücksetzbarkeit im Fehlerfall. Deshalb wurde bereits in [Gra81a] das Prinzip der Kompensation eingeführt: Anstatt eines Rücksetzens im Fehlerfall werden logische “Gegenaktionen” ausgeführt, wodurch in gewisser Weise eine *semantische Atomarität* gewährleistet wird.

Eine grundsätzliche Idee der offen geschachtelten Transaktionen ist die Trennung der verschiedenen Aspekte der ACID-Eigenschaften. Während z.B. die Atomarität der ACID-Transaktionen stark von der Isolationseigenschaft abhängt, ist die Kompensation der offen geschachtelten Transaktionen zunächst unabhängig von dieser Eigenschaft[1]. Diese Sichtweise der prinzipiellen Unabhängigkeit sogenannter Kontrollsphären (engl. *spheres of control*) geht auf [Dav78] zurück.

In den folgenden Abschnitten wird eingehend auf spezielle Ausprägungen der offen geschachtelten Transaktionen eingegangen. Deshalb wird an dieser Stelle auf

1. Inwiefern sich diese Aussage verallgemeinern läßt, wird im weiteren Verlauf diskutiert.

die genauere Betrachtung der Eigenschaften und der Einsatzgebiete des Konzeptes verzichtet.

3.4 Mehrschicht-Transaktionen

Mehrschicht-Transaktionen [Weik89],[WeSc92] (engl. *multi-level transactions*) stellen eine enge Verknüpfung des ACID-Prinzips und der offen geschachtelten Transaktionen dar. Einerseits werden durch die Verwendung offen geschachtelter Transaktionen die Atomarität und die Isolation aufgegeben, andererseits wird durch die Einführung einer strikten Aufrufhierarchie sichergestellt, daß auf der jeweiligen Hierarchiestufe entsprechende Garantien gegeben werden können.

Genauer gesagt: Eine Mehrschicht-Transaktion einer Ebene wird dadurch realisiert, daß sie Operationen der direkt darunter liegenden Ebene aufruft. Zugriffsbeschränkungen auf der tieferen Ebene werden nach dem Ende einer Operation auf dieser Ebene entfernt (entsprechend dem Verhalten offen geschachtelter Transaktionen). Allerdings müssen auf der Ebene der aufrufenden Transaktion Zugriffsbeschränkungen etabliert werden, um so Konflikte mit parallel laufenden Transaktionen (der selben Ebene) zu vermeiden.

Ununterbrechbarkeit gewährleisten Mehrschicht-Transaktionen gemäß dem Konzept der offen geschachtelten Transaktionen auf der Basis von Kompensationsaktionen, d.h. zu jeder Operation auf jeder Ebene muß eine entsprechende Kompensationsoperation zur Verfügung stehen.

Somit stellen Mehrschicht Transaktionen strenge Anforderungen an den Aufbau eines Systems, da es nicht zulässig ist, Operationen einer anderen Ebene als der direkt darunter liegenden aufzurufen. Die Vorteile dieser Verwendung offen geschachtelter Transaktionen ergeben sich insbesondere in Anwendungsbereichen, in denen das Zugriffsgranulat mit absteigender Hierarchiestufe zunimmt.

Ein Beispiel hierfür sind z.B. relationale Datenbanksysteme. Auf der Ebene des tupelorientierten Zugriffs besteht das Granulat der hier definierten Operationen aus Tupeln. Die darunter liegende Ebene bildet die Tupel auf Seiten eines stabilen Speichers ab. Geht man nun davon aus, daß mehrere Tupel in einer Seite gespeichert werden können, liegt der Vorteil der Mehrschicht-Transaktionen klar auf der Hand. Während Sperren auf der Tupelebene Inkonsistenzen bezüglich der Tupel verhindern, werden trotzdem parallele Zugriffe auf ein und diesselbe Seite zugelassen, da die Operationen auf der Seitenebene nach ihrer Ausführung keine Beschränkungen des Zugriffs fordern.

3.5 Sagas

Sagas [GaSa87] stellen eine spezielle Ausprägung offen geschachtelter Transak-

tionen dar. Das Modell der Sagas war eines der ersten, welches eine Trennung der Ablauflogik von den ausführenden Teilen vornahm. So wird eine Ausführung dadurch definiert, daß ein Kontrollfluß explizit zwischen sogenannten Steps definiert wird. Dabei war zunächst nur eine einfache sequentielle Verkettung zulässig, die später jedoch um weitere Konstrukte erweitert wurde (Schleifen, bedingte Verzweigung usw.).

Wie bei allen Vertretern der offen geschachtelten Transaktionen müssen Kompensationsaktionen definiert werden, die im Fehlerfall automatisch in inverser Ordnung zu den Originalsteps ausgeführt werden.

Wichtig zu erwähnen ist hierbei, daß durch die Einführung eines expliziten Kontrollflusses in Verbindung mit einer den Steps zugeordneten Kompensation die Notwendigkeit besteht, Daten bezüglich des Kontrollflusses persistent zu speichern. Beispielsweise muß die Information über erfolgreich ausgeführte Steps dauerhaft gespeichert werden, um im Fehlerfall die notwendigen Kompensationsaktionen ermitteln zu können.

Auf Fehlerfälle wird in Sagas durch Rücksetzen der gerade aktiven Transaktion(en) und der anschließenden Ausführung der Kompensationsaktionen der Vorgänger in umgekehrter zeitlicher Reihenfolge reagiert. Man kann somit von einer semantischen Ununterbrechbarkeit analog zu den offen geschachtelten Transaktionen reden.

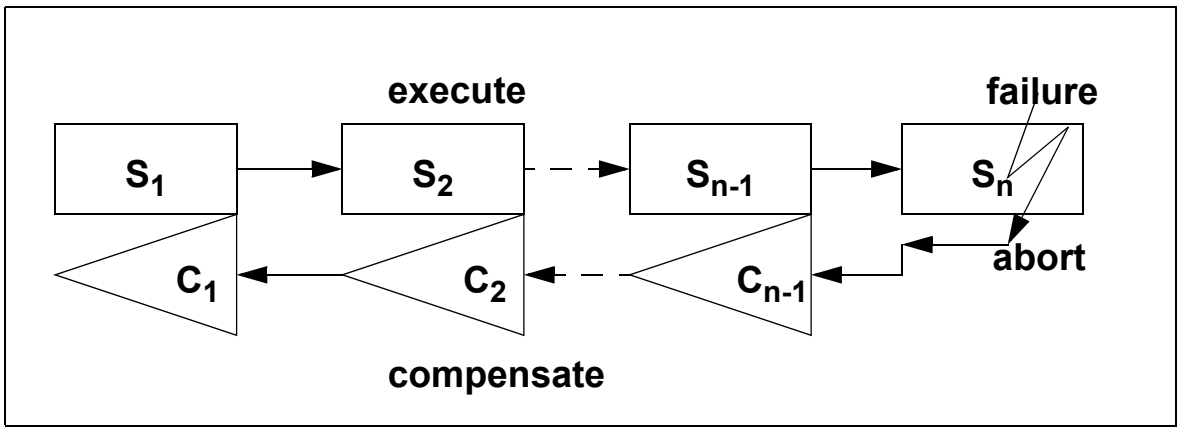

Abbildung 3-1: Prinzip der SAGAs

Steps stellen ACID-Transaktionen bzw. geschlossen geschachtelte Transaktionen dar, so daß auf dieser Ebene auch die Isolation gewährleistet ist. Auf der Ebene der Sagas selbst besteht jedoch kein Isolationsschutz. Deshalb sind Sagas nicht für Anwendungen geeignet in denen es zu Konflikten auf Grund von Parallelverarbeitung kommen kann (s. Abschnitt 3.1.1).

Weiterentwicklungen der Sagas [GGK90] [GGK91a] [GGK91b] veränderten die Semantik dieses Ausführungsmodells zum Teil radikal. So wurde mit der Einfüh-

rung des Schachtelungsprinzips eine Unterscheidung in notwendige (engl. *vital*), nicht-notwendige (engl. *non-vita*l) und unabhängige (engl. *independent*) Sagas vorgenommen, die die Fehlersemantik stark beeinflussen.

Es ergeben sich folgende Unterschiede zu der bekannten Eltern/Kind-Abhängigkeit im Falle der geschlossen geschachtelten Transaktion:

- Der Abbruch einer als notwendig deklarierten Sub-Saga bewirkt den Abbruch der entsprechenden Eltern-Saga und umgekehrt.
- Eine als unabhängig deklarierte Sub-Saga bleibt vom Abbruch der Eltern-Saga unbeeinflußt.

Ein weiterer Aspekt, der erst mit den Erweiterungen der Sagas eingeführt wurde, ist die Fortsetzbarkeit einer Saga nach einem Fehlerfall. Da dieser Aspekt dem Ansatz in ConTracts entspricht, sei an dieser Stelle auf Abschnitt 3.6.2.1 verwiesen.

Mit der Einführung der unabhängigen (Sub-) Sagas wurde die semantische Atomarität der Sagas aufgegeben. Somit sind Sagas in ihrer neueren Form nicht mehr vollständig in der Klasse der offen geschachtelten Transaktionen enthalten. Darüber hinaus sind mit den Erweiterungen erste Abhängigkeitsbeziehungen zwischen Kompensationsaktionen eingeführt worden, welche in neueren Arbeiten wieder aufgegriffen wurden [Leym95][RSS97].

3.6 ConTracts

ConTracts [Reut89][1] [WäRe92] sind eine Weiterentwicklung der Sagas und gehören ebenso zur Klasse der offen geschachtelten Transaktionen. Im Gegensatz zu Sagas gehen die Garantien des ConTract Modells sehr viel weiter und versuchen gemäß des Ansatzes von Davies [Dav78] weitgehend unabhängige Kontrollsphären anzubieten.

3.6.1 Das Skript

Wie bei Sagas muß in ConTracts ein expliziter Kontrollfluß zwischen den Einzelaktivitäten, den sogenannten Steps, definiert werden. Dabei sind nahezu beliebige Kontrollflußbeziehungen zulässig (Schleifen, bedingte Verzweigungen, resultatsabhängige Sprünge, Parallelverarbeitung usw.). Die für diese Arbeit relevanten Elemente des Skriptes werden hier kurz eingeführt. Für detailliertere Darstellungen sei auf [RSW92], [Schw93b] und [Wäch96] verwiesen.

1. Anmerkung des Autors: ConTracts wurden nach dem ursprünglichen Saga-Modell eingeführt; jedoch vor deren Erweiterungen.

3.6.1.1 Kontext

Da das ConTract-Modell die explizite Definition eines Datenflusses vorsieht, werden Daten, die von einem Step zu einem anderen weiter gereicht werden, in sogenannten Kontextvariablen abgelegt. Diese bilden zusammen den sogenannten Kontext, der neben diesen Variablen auch die ablaufrelevanten Variablen wie z.B. Schleifenzähler enthält.

Wichtige Eigenschaften des Kontext sind die Persistenz und die änderungslose Verwaltung der Variablen. Genauer, das Speichern einer geänderten Variablen überschreibt nicht den Originalwert sondern erzeugt eine neue Version.

3.6.1.2 Kompensation

Als Vertreter der offen geschachtelten Transaktionen benutzt auch das ConTract Modell das Prinzip der Kompensation, um ein logisches Zurücknehmen von Steps zu ermöglichen. Der ursprüngliche Ansatz, hierfür jedem Step einen Kompensationsstep zuzuordnen, wurde inzwischen erweitert, so daß es möglich ist einem Teilskript des Originalablaufs wiederum ein Teilskript als Kompensation zuzuordnen [RSS97].

3.6.1.3 Transaktionen

Das ConTract Modell basiert auf geschlossen geschachtelten Transaktionen als Ausführungsmodell für die Steps. Auf dieser Basis ist es möglich, Steps zu (geschlossen geschachtelten) Transaktionen zu Gruppieren, um so eine ACID Semantik für diese Gruppe zu definieren.

3.6.1.4 Invarianten

Da Änderungen auf Datenelementen am Ende einer der Transaktionen, die eine Gruppe von Steps umgeben, sichtbar werden, kann es zu Problemen durch parallel laufende ConTracts kommen (siehe Abschnitt 3.1.1). Anstatt, wie bei Sagas, Contracts als ungeeignet für diese Anwendungsfälle zu deklarieren, wurde das sogenannte Invariantenkonzept eingeführt [ReSw95]. Wie im Verlauf dieser Arbeit noch eingehend diskutiert wird, ist es mit den Invarianten möglich, Prädikate auf gemeinsam genutzten Datenelementen zu etablieren, um so Zugriffe anderer ConTracts einzuschränken.

3.6.2 Eigenschaften von ConTracts

In diesem Abschnitt werden die grundsätzlichen Eigenschaften von ConTracts erläutert. Diese bilden die Basis für die spätere Einführung eines Korrektheitskriteriums.

3.6.2.1 Fortsetzbarkeit

Im Gegensatz zur Atomarität der ACID-Transaktionen und dem Ansatz von Sagas garantieren ConTracts die Fortsetzbarkeit (engl. *forward recoverability*) eines einmal begonnenen Ablaufs. Bei einem Fehlerfall bezüglich der Ausführung wird somit zunächst der aktuelle Zustand des Ablaufs wieder hergestellt. Aktive Transaktionen werden dann zurück gesetzt (engl. *backward recovery*), und anschließend wird mit der Bearbeitung fortgefahren.

Das Fehlschlagen einer Stepausführung hat zunächst keine direkten Auswirkungen auf die Ausführung des ConTracts. Hier wird garantiert, daß eine begrenzte Anzahl von Wiederholungen versucht wird, bzw. ein alternativer Zweig des Skriptes ausgeführt werden kann.

Trotzdem kann auch bei diesem Mechanismus ein Zustand auftreten, in dem eine weitere automatische Maßnahme des Ausführungssystems keinen weiteren Erfolg verspricht. In diesem Fall wird die Ausführung angehalten (siehe auch Abschnitt 3.6.2.2) und die Benutzerin informiert. Diese kann dann entweder dem System mitteilen, daß eine weitere Fortsetzung sinnvoll ist, oder die Kompensation (siehe Abschnitt 3.6.2.4) der bisherigen Ausführung einleiten.

3.6.2.2 Dauerhaftigkeit

Die Eigenschaft der Dauerhaftigkeit erstreckt sich bei dem ConTract-Modell nicht nur auf Daten, sondern auf den Ablauf als Ganzes. D.h. sowohl der Zustand des Ablaufs (an welcher Stelle befindet sich der Kontrollfluß und welche Steps wurden bisher ausgeführt) als auch alle ablaufrelevanten Variablen (*Kontext*) sind persistent.

Somit ist jeder Verarbeitungszustand (nach einer abgeschlossenen ACID Transaktion) persistent. Diese Eigenschaft wird einerseits für die Fortsetzbarkeit nach einem Fehlerfall benutzt, kann jedoch auch für den “Normalablauf” sinnvoll eingesetzt werden, z.B. um eine Bearbeitung zeitweise unterbrechen zu können (engl. *suspend*), die Historie eines Ablaufs zu ermitteln oder einen ConTract migrieren zu können.

3.6.2.3 Durchlässigkeit

Wie bereits in Abschnitt 3.3.3 erwähnt, wird die Isolations-Eigenschaft im Falle offen geschachtelter Transaktionen aufgegeben. Um diesem Umstand auch begrifflich Rechnung zu tragen wird im weiteren nicht über die Isolation von ConTracts sondern von ihrer Durchlässigkeit (engl. *permeability*) gesprochen.

Grundsätzlich sind nach dem Abschluß einer ACID-Transaktion die Ergebnisse sichtbar für alle Aktivitäten (auch außerhalb des aktuellen ConTracts). D.h. es gibt

keinen Systemmechanismus der irgendwelche Schutzmaßnahmen, wie z.B. die Etablierung von Sperren, trifft. Sind allerdings Invarianten definiert schränken diese den Zugriff entsprechend ein.

Somit ist für einen ConTract garantiert, daß die von ihm etablierten Invarianten nicht verletzt werden. Damit ist eine anwendungsabhängige Regulierung der Freigabe von geänderten Daten möglich.

3.6.2.4 Kompensierbarkeit

Für einen ConTract ist garantiert, daß zu jedem beliebigen Zeitpunkt der Ausführung, die Kompensation eingeleitet werden kann. Für einen aktuellen Ablauf bedeutet dies, daß aktive Transaktionen abgebrochen und anschließend Kompensationsaktionen für erfolgreich abgeschlossene Step ausgeführt werden.

Ursprünglich war die Reihenfolge der Abarbeitung von Kompensationsaktionen nicht festgelegt, so daß sogar die Möglichkeit der gleichzeitigen Ausführung aller Kompensationen erwogen wurde. Wie sich im Verlauf dieser Arbeit noch herausstellen wird, kann diese Flexibilität im allgemeinen nicht unterstützt werden.

3.6.2.5 Konsistenz

Obwohl die Eigenschaft der Konsistenz bzw. der Konsistenzerhaltung allen hier betrachteten Ausführungsmodellen gemeinsam sind, soll sie an dieser Stelle noch einmal gesondert erwähnt werden.

Ein ConTract gewährleistet, daß wenn er auf einem konsistenten Zustand von Datenobjekten gestartet wird, diese wieder in einem konsistenten Zustand hinterläßt. Dabei kann in einen erfolgreichen und einen kompensierten Endzustand unterscheiden werden.

4 Formale Modelle konkurrierender Abläufe

Die formale Darstellung von Abläufen ist eine Voraussetzung für die Definition entsprechender (ebenso formaler) Korrektheitskriterien. Dieses Kapitel führt in entsprechende Formalismen ein und stellt insbesondere die für das ConTract-Modell gewählte Darstellung vor.

Ein wichtiger Aspekt hierbei ist, daß Formalismen zur Beschreibung einer Ausführung nicht unbedingt auch bei der Programmierung von Abläufen zum Einsatz kommen. Der Grund hierfür sind die unterschiedlichen Anforderungen an die Notationen. Während zur Programmierung Darstellungen herangezogen werden, die für einen menschlichen Benutzer möglichst einfach erlernbar sind, orientieren sich Notationen, die für Laufzeitsysteme gedacht sind, an der effizienten Ausführbarkeit und Problemunabhängigkeit.

Da es sich bei den formalen Sprachen zur Beschreibung von Abläufen im Allgemeinen um einfache aber sehr flexible Sprachen handelt, existiert üblicherweise keine isomorphe, sondern nur eine homomorphe Abbildung der auf der Programmierebene genutzten Notation auf die formale Darstellung. Es gehen also Informationen, die auf der Programmierebene zur Verfügung standen “verloren” und sind somit aus der Laufzeitnotation nicht mehr wieder zu gewinnen. Dies stellt einen eigenen Problembereich dar, wenn Änderungen von Abläufen zur Ausführungszeit unterstützt werden soll.

Im weiteren soll folgende Konvention bezüglich der verwendeten Termini gelten:

1. Die formale Notation, die aus einer von einem Programmierer erstellten Definition eines Ablaufs erzeugt wurde (oder werden kann), wird im weiteren als *Schablone* oder *Template* bezeichnet.
2. Eine spezielle Ausprägung eines Template, die zur Ausführung eingesetzt wird, wird als *Instanz* oder *Ausführungsinstanz* bezeichnet. Ein Template kann mehrfach instanziiert werden.

4.1 Das read/write Modell

Da ACID-Transaktionen nicht als eigentliches Ausführungsmodell, sondern nur zur deklarativen Zuordnung einer gewissen Ausführungssemantik eingeführt wurden, gibt es keine Laufzeitumgebung im Sinne einer virtuellen Maschine für Abläufe mit ACID-Eigenschaften. Trotzdem ist es aus abstrakter Sicht möglich eine Maschine zu definieren, die die Abarbeitungssemantik von ACID-Transak-

tionen hinreichend beschreibt. Diese Maschine basiert auf dem sogenannten Lese/Schreib-Modell (engl. *read-write model*) [Papa86][BHG87][GrRe93].

4.1.1 Operationen

Eine abstrakte Maschine zur Verarbeitung von ACID-Transaktionen kennt vier elementare Operationen[1].

Operation	Semantik
r: read(t, a)	Liefert den Wert eines Datenobjektes a an eine Transaktion t.
w: write(t,a)	(Über-) Schreibt den Wert eines Datenobjektes a im Auftrag einer Transaktion t.
c: commit(t)	Macht die Änderungen von t dauerhaft und beendet t.
a: abort(t)	Setzt die Änderungen von t zurück und beendet t.

Tabelle 4-1: Operation im read/write Modell

Wie einfach zu erkennen ist, verwaltet die abstrakte Maschine eine Menge von Datenobjekten, die mit der Operation “write” manipuliert werden können. Es mag erstaunen, daß es weder eine Operation “create” noch eine Operation “delete” gibt, die ein Datenobjekt erzeugen bzw. löschen. Diese ungewöhnliche Eigenschaft hat jedoch historische Gründe, da zu Beginn der Nutzung von Datenbanksystemen auf der Basis von Speicherseiten gearbeitet wurde. Nun läßt sich natürlich die Erzeugung von Datenobjekten (und der Löschung) auf einen Schreibzugriff auf eine Speicherseite abbilden, so daß keine zusätzlichen Operationen notwendig sind.

Eine weitere Operation, die den Beginn einer Transaktion anzeigt (fordert) wurde auf der Ebene der abstrakten Maschine ebenfalls nicht eingeführt. Auch dies läßt sich wiederum aus der Entstehungsgeschichte erklären. Ursprünglich wurde eine Transaktion über die Identifikation des ausführenden Prozesses[2] eindeutig gekennzeichnet, weshalb mit der ersten Operation des Prozesses implizit eine Transaktion begonnen wurde (siehe auch voriges Kapitel). Mit der Einführung verteilter Transaktionen und der gleichzeitigen Nutzung mehrerer Datenbankverbindungen in einem Prozeß stellt dieses Vorgehen ein Problem dar. Da in der weiteren Verwendung der eingeführten Notation diese Problematik nicht zum Tragen kommt, wird an dieser Stelle auf eine Erweiterung der Menge der Operationen verzichtet.

1. Im verteilten Fall kommt eine fünfte Operation “prepare” hinzu, die allerdings nicht auf Anwendungsebene zur Verfügung steht.
2. Hier ist ein Betriebssystemprozeß gemeint.

4.1.2 Ausführungen und ihre Semantik

Eine ACID-Transaktion kann nun mit Hilfe der Operationen der abstrakten Maschine dargestellt werden (es wird eine abkürzende Schreibweise für die Bezeichnung der Operationen benutzt). Da die eigentliche Definition der Transaktion nicht bekannt ist, wird diese Darstellung als formale Interpretation bezeichnet.

Definition 4-1 (ACID-TA): *Eine Interpretation I(t) einer ACID-Transaktion t ist ein Tupel (A, <), wobei A eine geordnete Menge von Operationen bezüglich der partiellen Ordnung "<" darstellt:*

$$A = \{o_i\}, o_i \in \{r, w, c, a\}$$

$$o_i \text{ vor } o_j \text{ in t} \Rightarrow o_i < o_j$$

$$(o_i, o_k \in A \wedge \overset{\exists}{o_i} = c \wedge o_k = a) \overset{\wedge}{} (o_j \in A, \overset{\forall}{o_j} \in \{c, a\}) \quad \neg \overset{\exists}{o_m} \in A \; o_j < o_m \quad \blacksquare$$

Eine formale Interpretation einer ACID-Transaktion besteht somit aus einer Menge von Lese- und Schreiboperationen, deren Abarbeitungsreihenfolge festgelegt ist. Ebenso gehören zu der Transaktion commit- bzw. abort-Operationen, denen jedoch keine weiteren Operationen folgen dürfen.

Im folgenden wird durch die Verwendung der Schreibweise $o_i < o_j$ ausgedrückt werden, daß o_j ein *unmittelbarer* Nachfolger von o_i ist. Mit $o_i <^+ o_j$ soll eine ausschließlich mittelbare Reihenfolgebeziehung bezeichnet werden. Die Bezeichnung $o_i <^* o_j$ subsummiert die unmittelbare und die mittelbare Reihenfolgebeziehung.

Die kurz angedeutete Semantik der Operationen soll hier nun etwas vertieft werden. Eventuell notwendige Seiteneffekte der Operationen zur Realisierung der Isolationseigenschaft werden im folgenden Kapitel behandelt.

1. Leseoperationen greifen auf ein Datenobjekt zu und liefern den jeweiligen Wert des Objektes an die Transaktion zurück.
2. Schreiboperationen erzeugen, löschen oder ändern ein Datenobjekt. Gleichzeitig wird dabei die Information hinterlegt, wie die Schreiboperation rückgängig gemacht werden kann. In den meisten Fällen geschieht dies automatisch durch die ausführende Maschine z.B. durch die Speicherung des sogenannten *before image*.
3. Die commit Operation hat ausschließlich die Aufgabe alle Änderungen einer Transaktion dauerhaft zu machen und die Transaktion abzuschließen.

4. Eine abort Operation führt alle gespeicherten Gegenaktionen (inverse Schreiboperationen) in der umgekehrten Reihenfolge der Ausführung der Originalschreibzugriffe aus, macht die Änderungen dauerhaft (soweit notwendig) und schließt eine Transaktion ab. Die Aktionen zur dauerhaften Speicherung der Änderungen und der Abschluß der Transaktion kann auch als eine commit Operation nach der Ausführung aller Gegenaktionen angesehen werden.

Durch die Atomaritätseigenschaft werden abort Operationen automatisch nach einem Fehlerfall ausgelöst. Dies bedingt die Verwaltung von persistenter Zustandsinformation, die im folgenden Abschnitt beschrieben wird.

Die Maschine als ganzes arbeitet nach dem folgenden Prinzip:

- Zunächst werden alle Operationen gesucht, die keinen Vorgänger bezüglich der Partialordnung besitzen. Diese werden zur Ausführung gebracht.
- Ist die Ausführung nicht erfolgreich, wird eine abort-Operation ausgeführt. Ist die Ausführung erfolgreich, werden die direkten Nachfolger gesucht und zur Ausführung gebracht.
- Das Ende ist erreicht, wenn alle Operationen beendet sind und keine Nachfolger gefunden werden können.

4.1.3 Persistente Zustände

Die abstrakte Maschine verwaltet zur Implementierung der ACID-Semantik Zustände der Transaktionen und der Datenobjekte. Zu diesem Zweck muß stabiler, persistenter Speicher zur Verfügung stehen, um auch nach einem Systemausfall die Atomarität und die Dauerhaftigkeit gewährleisten zu können.

Insgesamt nehmen ACID-Transaktionen in der abstrakten Maschine nur zwei Zustände ein:

1. Aktiv (nach der ersten Lese- oder Schreiboperation)
2. Abgeschlossen (nach einer commit oder abort Operation)

Auf Grund der ACID-Eigenschaft ist hiefür nur der Zustand “aktiv” stabil zu speichern, um ein eventuelles Rücksetzen nach einem Fehlerfall auslösen zu können.

Bezüglich der verwalteten Datenobjekte gestaltet sich die Zustandsverwaltung etwas komplexer, da deren Zustände von dem Zustand der manipulierenden Transaktion abhängig sind. Grundsätzlich können drei Zustände bei Datenobjekten unterschieden werden:

1. clean: alle bisherigen Zugriffe erfolgten von abgeschlossenen Transaktionen.
2. touched: es erfolgte ein lesender Zugriff einer aktiven Transaktion.
3. dirty: es erfolgte ein Schreibzugriff einer aktiven Transaktion.

Die Atomaritätseigenschaft impliziert nun, daß für alle Datenobjekte, die im Zustand “dirty” sind, Informationen darüber gespeichert werden müssen, wie diese Objekte wieder in einen “clean” Zustand überführt werden können. Dabei ist zu beachten, daß diese Information ebenso Dauerhaft sein muß, wie das geänderte Objekt selbst. Ist z.B. das geänderte Objekt nur im Hauptspeicher, genügt es die “Undo-Information” ebenfalls nur im Hauptspeicher abzulegen.

Auf Grund der Dauerhaftigkeit, muß ein Objekt, welches auf Grund einer commit-Operation vom Zustand dirty in den Zustand clean übergeht stabil gespeichert werden, bzw. es muß zumindest die Information stabil gespeichert werden, die ausreicht, um den Zustand auch nach einem Systemausfall wieder herzustellen.

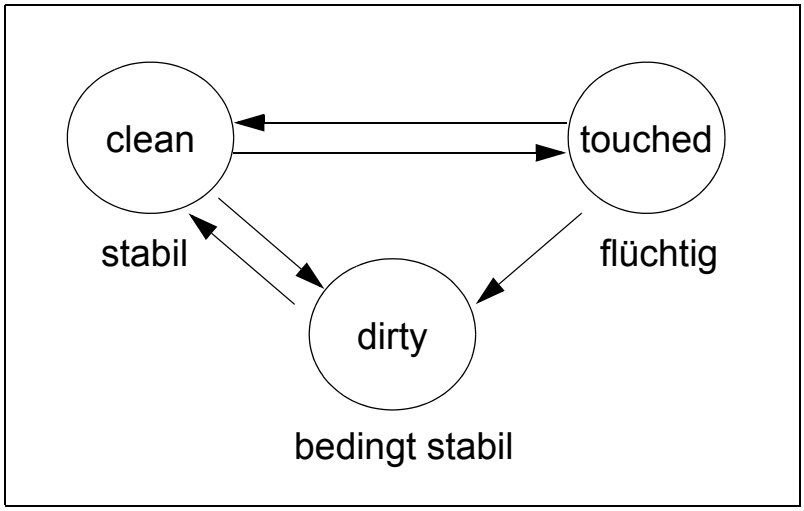

Abbildung 4-1: Zustandsdiagramm für Datenobjekte bei ACID-TA

4.1.4 Erweiterungen für geschachtelte Transaktionen

Das bisher beschriebene read/write Modell kann einfach erweitert werden, um auch die Semantik geschlossen geschachtelter Transaktionen zu beschreiben. Zunächst muß hierfür die Menge der Operationen erweitert werden:

Operation	Semantik
β: begin_SubTA(t)	Erzeugt eine Sub-Transaktion bezüglich der angegebenen (Eltern-)Transaktion t.

Tabelle 4-2: Erweiterte Operation im read/write Modell

Operation	Semantik
χ: commit_SubTA(t)	Beendet die Sub-Transaktion t
α: abort_SubTA(t)	Setzt die Änderungen der Sub-Transaktion t zurück und beendet die Sub-Transaktion

Tabelle 4-2: Erweiterte Operation im read/write Modell

Mit der Erweiterung der Menge der Operationen, ist natürlich auch eine entsprechende Erweiterung von Definition 4-1 notwendig.

Definition 4-2 (Geschlossen geschachtelte TA): *Eine Interpretation I(t) einer geschlossen geschachtelten Transaktion t ist ein Tupel (A, <), wobei A eine geordnete Menge von Operationen bezüglich der partiellen Ordnung "<" aus Definition 4-1 darstellt. Die Bedingung, daß weder einer abort- noch einer commit-Operation von t eine weitere Operation folgen darf gilt analog. Wird mit* β^s *bzw.* β^r *der Beginn und mit* ε^s *bzw.* ε^r *das Ende (*χ *bzw.* α*) zweier bestimmter Sub-Transaktionen s und r bezeichnet, gelten folgende Zusatzbedingungen:*

$$A = \{o_i\}, o_i \in \{r, w, c, a, \beta, \chi, \alpha\}$$

$$\forall_{(\beta^r \in A)} \neg \exists_{(\beta^s \in A)} \beta^r <^* \beta^s \wedge \neg(\varepsilon^r <^* \varepsilon^s)$$

$$\forall_{(\beta^r \in A)} \exists \varepsilon^r \in A \quad \blacksquare$$

Eine Subtransaktion, die durch die Operation β und eine der Operationen α oder χ begrenzt wird, bildet eine geschlossene Einheit. Deshalb ist es nicht zulässig, innerhalb einer Subtransaktion eine weitere zu beginnen, deren Ende nicht ebenfalls in der Subtransaktion enthalten ist. Ebenso ist es notwendig, daß für alle Beginn-Operationen von Sub-Transaktionen entsprechende Ende-Operationen vorhanden sind.

Mit dem Begriff *Top-Level-Transaktion* werden diejenigen Transaktionen bezeichnet, die keine Sub-Transaktionen von anderen Transaktionen sind. Somit werden Top-Level-Transaktionen auch nicht mit einer β-Operation begonnen. Die Operationen α bzw. χ erzeugen im Gegensatz zu den abort- und commit-Operationen von Top-Level-Transaktionen keine persistenten Zustände. Statt dessen werden beim Abschluß einer Sub-Transaktion alle die Transaktion betreffenden Verwaltungsdaten an die Eltern-Transaktion weiter gereicht. Handelt es sich bei der Eltern-Transaktion um eine Top-Level-Transaktion ist diese für die Persistenzeigenschaft verantwortlich.

Da sich somit durch die Einführung von geschlossen geschachtelten Transaktionen keine weiteren persistenten Zustände ergeben, erscheint die Erweiterung der abstrakten Maschine zunächst trivial. Allerdings ergeben sich nicht zu vernachlässigende Schwierigkeiten bei der Umsetzung der speziellen Isolationseigenschaften und der Übernahme der Verwaltungsdaten durch eine Elterntransaktion.

Bezeichnenderweise sind momentan kaum Datenbanksysteme zu finden, die geschlossen geschachtelte Transaktionen implementieren. Der Grund hierfür ist wieder historischer Art. Da bei klassischen Transaktionen keine "Weitergabe" irgendwelcher Informationen notwendig ist, sind die entsprechenden Algorithmen zur Realisierung auch hierauf optimiert. Betrachtet man speziell die Implementierung von Log-Systemen, können Log-Sätze durch einen simplen Bezeichner in einem Log-Satz einer Transaktion zugeordnet werden. Da eine Sub-Transaktion aber ihre Log-Sätze an ihre Eltern-Transaktion weitergibt, ist solch eine simple Zuordnung nicht mehr möglich.

Ein weiterer nicht-trivialer Aspekt ergibt sich aus der Atomaritätseigenschaft der Transaktionen. Diese besagt, daß eine geschlossen geschachtelte Transaktion jederzeit abgebrochen und somit ihre Änderungen rückgängig gemacht werden können. Somit ergibt sich eine sogenannte *abort-Abhängigkeit* [ChRa90][Günt96] zwischen Eltern- und Kind-Transaktionen, die besagt, daß im Falle eines Abbruchs der Eltern-Transaktion auch alle Kind-Transaktionen zurückgesetzt werden müssen.

4.2 Mehrschicht-Transaktionen

Wie bereits im vorigen Kapitel eingeführt, basieren Mehrschicht-Transaktionen auf einer strikten Aufteilung eines Systems in mehrere Abstraktionsebenen. Versucht man nun eine abstrakte Maschine zur Abwicklung von Mehrschicht-Transaktionen zu definieren, kann dies zunächst nur auf einer Ebene erfolgen. Dabei wird die darunter liegende Maschine mit benutzt und man erhält somit eine rekursive Definition. Im Datenbankbereich wird diese Rekursion durch die Abstraktionsebene begrenzt, die die Abbildung auf physische Speicherseiten vornimmt (mit Hilfe sogenannter *Mini-Transaktionen*) [GrRe93][Günt96]. Da diese jedoch im Prinzip keine neuen Aspekte im Vergleich zu dem bereits diskutierten read/write Modell aufwerfen, wird an dieser Stelle nur auf die entsprechende Literatur verwiesen. Im folgenden soll diese Ebene mit L_0 bezeichnet werden.

4.2.1 Operationen

Aus abstrakter Sicht kennt eine Maschine einer Stufe i die transaktionalen Operationen, sowie die Operationen, die auf dieser Stufe angesiedelt sind. Wichtig zu erwähnen ist, daß die anwendungsorientierten Operationen wiederum Operatio-

nen auf der nächst niedrigeren Stufe nutzen.

Operation	Semantik
e: execute(t, o_k, $\bar{o}_k$)	Führt die Operation o_k innerhalb der Transaktion t aus und gibt die zugehörige Kompensationsoperation $\bar{o}_k$ an.
c: commit(t)	Macht die Änderungen von t dauerhaft und beendet t.
a: abort(t)	Führt Kompensationsoperationen aus und beendet t.

Tabelle 4-3: Operationen von Multi-Level-Transaktionen

Grundsätzlich lassen sich in den Operationen von Mehr-Schicht-Transaktionen die manipulierten Datenobjekte nicht mehr identifizieren. Somit ist auch die automatische Generierung von inversen Aktionen im Allgemeinen nicht mehr möglich. Deshalb müssen die Kompensationsaktionen explizit angegeben werden. Da, wie bereits erwähnt, die isolationsbezogene Problematik im folgenden Kapitel besprochen wird, sind auch diesbezügliche Ergänzungsmöglichkeiten hier nicht berücksichtigt.

4.2.2 Ausführungen und ihre Semantik

Auf der Basis der Operationen kann nun eine Mehrschicht-Transaktion definiert werden. Diese Definition erfolgt wie bereits erwähnt rekursiv:

Definition 4-3 (Mehrschicht-TA): *O^i ($i > 0$) sei die Menge der Operationen der Stufe i ohne die transaktionalen Operationen a und c. Eine Interpretation $I(t^i)$ einer Mehrschicht-Transaktion der Stufe i ist ein Tupel (A,<), wobei A eine geordnete Menge von Operationen bezüglich der Ordnungsrelation "<" (siehe Definition 4-1) darstellt:*

$$A = \{x_j\},\ x_j \in \{e,a,c\},\ e = execute(t^i, o_k, \bar{o}_k),\ o_k, \bar{o}_k \in O^{i-1}$$

$$(\forall x_j \in A, x_j \in \{c,a\})\ \neg(\exists x_m \in A)\ x_j < x_m \quad \blacksquare$$

Interpretationen von Mehrschicht-Transaktionen bestehen aus einer Menge von Operationen, deren Ausführungsreihenfolge mit Hilfe einer partiellen Ordnung (siehe Abschnitt 4.1.2) festgelegt ist. Die eigentlich ausführenden Operationen (e) sind dabei eine Anforderung an die abstrakte Maschine, eine Transaktion auf der nächst tiefer liegenden Stufe ausführen zu lassen. Für die Operationen commit und abort gilt wie beim read/write Modell die Einschränkung, daß diesen Operationen keine weitere Operation folgen darf. Analog zum read/write-Modell erfolgt die Interpretation der Ordnungsrelation.

Bezüglich der Semantik der Operationen ergeben sich grundsätzliche Unterschiede zum read/write-Modell:

1. Eine Operation e stellt den Aufruf einer Transaktion der nächst tiefer liegenden Ebene dar. Da es sich bei Mehrschicht-Transaktionen um eine Ausprägung der offen geschachtelten Transaktionen handelt, werden die Änderungen bei einem erfolgreichen Abschluß bereits dauerhaft. Ebenso dauerhaft muß dann die Information über die Kompensationsaktion gespeichert werden.
2. Die commit-Operation der Mehrschicht-Transaktionen entspricht der commit-Operation des read/write-Modells mit der zusätzlichen Aufgabe, Informationen über eine eventuelle Kompensationstransaktion persistent zu speichern.
3. Eine abort-Operation kann gegenüber dem read/write Modell weit aus aufwendiger sein, da die Änderungen erfolgreicher Sub-Transaktionen bereits dauerhaft sind. Deshalb müssen Kompensations- oder Gegentransaktionen ausgeführt werden, die einen persistenten Zustand erzeugen, der äquivalent zu dem Ausgangszustand der Transaktion ist. Die Ausführung der Kompensationstransaktionen geschieht dabei in umgekehrter zeitlicher Reihenfolge zur Ausführung der Originaloperationen.

Da Mehrschicht-Transaktionen eine semantische Atomarität garantieren, werden nach einem Systemausfall automatisch abort-Operationen für aktive Transaktionen eingeleitet. Diese abort-Operationen müssen in einer Reihenfolge ausgeführt werden, die den Abstraktionsebenen entspricht. Somit erfolgt zunächst der Abbruch der aktiven Transaktionen auf Stufe 0, dann auf Stufe 1 usw.

4.2.3 Persistente Zustände

Eine Zuordnung zwischen persistenten Zuständen und aktuellem Verarbeitungszustand einer Mehrschicht-Transaktion ist einfach möglich. Da, wie bereits beschrieben wurde, beim Abschluß jeder Operation deren Ergebnisse dauerhaft gespeichert sind, muß diese Tatsache ebenso in der aufrufenden Transaktion persistent vermerkt sein (um die Atomarität gewährleisten zu können). Somit ist jeder Zwischenzustand, nach dem erfolgreichen Abschluß einer Operation bzw. Transaktion der nächst tieferen Ebene, persistent in dem Sinne, daß er einen Systemausfall überdauert.

4.3 Abläufe nach Korth et. al.

Einer der ersten Ansätze, erweiterte Transaktionsmodelle formal zu erfassen, um Aussagen über deren Korrektheit treffen zu können stammt von Korth et. al. [KoSp88] [KLS90]. Bemerkenswert an dem Ansatz ist dabei, daß zwei Erweiterungen im Vergleich zum read/write-Modell und den Mehrschicht-Transaktionen vorgenommen werden:

1. Konsistenzbedingungen werden explizit modelliert.
2. Die Semantik von Kompensationsaktionen wird formal erfaßt.

Grundsätzlich basiert das Modell auf geschachtelten Transaktionen allgemeiner Art und eignet sich auch für den Bereich der versionierten Datenhaltung. Allerdings soll an dieser Stelle nicht näher auf den Versionierungsaspekt eingegangen werden.

4.3.1 Operationen

Die Operationen der Ausführungsnotation nach Korth et. al. orientieren sich an den Operationen der offen geschachtelten Transkationen. Die Erweiterungen beziehen sich auf die Überprüfung von Konsistenzbedingungen:

Operation	Semantik
e: execute(t, x_k, $\overline{x_k}$)	Führt die Subtransaktion x_k innerhalb der Transaktion t aus und gibt die zugehörige Kompensationsoperation $\overline{x_k}$ an.
γ: check(t, i_k)	Überprüft das Prädikat i_k welches eine notwendige Bedingung zur Ausführung von x_k darstellt.
ε: establish(t, o_k)	Überprüft das Prädikat o_k und beauftragt die Ausführungsmaschine mit der Sicherstellung. Dabei beschreibt o_k den korrekten Endzustand von x_k.
c: commit(t)	Macht die Änderungen von t dauerhaft und beendet t.
a: abort(t)	Führt Kompensationsoperationen aus und beendet t.

Tabelle 4-4: Operationen nach dem Modell von Korth et al.

4.3.2 Ausführungen und ihre Semantik

Da bei dem Ansatz von Korth et. al. Konsistenzbedingungen explizit modelliert

werden, müssen die entsprechenden Operationen bei der Ablaufdefinition angegeben werden. Ebenso wird gefordert, daß Kompensationsaktionen zum Definitionszeitpunkt bekannt sein müssen.

Definition 4-4 (TA nach Korth): *Eine Interpretation I(t) einer Transaktion t nach Korth et. al. ist ein Tupel (A, <), wobei A eine geordnete Menge von Operationen bezüglich einer partiellen Ordnung "<" (siehe Definition 4-1) darstellt:*

$$A = \{x_j\},\ x_j \in \{e, a, c, \gamma, \varepsilon\}$$

$$(\forall x_i \in A, x_i = e)\left((\exists x_j \in A, x_j = \gamma)^{x_j < x_i} \wedge (\exists x_k \in A, x_k = \varepsilon)^{x_i < x_k}\right)$$

$$(\forall x_j \in A, x_j \in \{c, a\})\ \neg(\exists x_m \in A)^{x_j < x_m} \quad \blacksquare$$

Eine Transaktion nach Korth et. al. fordert die Definition sowohl von Eingangs- als auch Ausgangsprädikaten für Sub-Transaktionen. Grundsätzlich wird angenommen, daß es sich bei den Sub-Transaktionen um offen geschachtelte Transaktionen handelt, die die Dauerhaftigkeits-Eigenschaft besitzen.

Bezüglich der Standardoperationen e, a und c ergeben sich somit keine Neuerungen gegenüber dem vorherigen Abschnitt. Die Operationen γ und ε verändern dagegen die Semantik. Evaluiert ein Prädikat, welches mit einer γ Operation geprüft wird, zu "Falsch", bedeutet dies, daß die zugehörige Subtransaktion nicht ausgeführt und somit die aktuelle Transaktion nicht fortgeführt werden kann. Der entsprechende Fall tritt ein, wenn die Prüfung eines Prädikates auf Grund einer ε Operation fehlschlägt. Dieser Fall ist jedoch ein zusätzlicher Indikator für die Tatsache, daß ein Konflikt mit einer parallel laufenden Transaktion eingetreten ist (siehe auch Abschnitt 3.1.1).

Da die persistenten Zustände denen des vorherigen Abschnitts gleichen, werden sie hier nicht weiter diskutiert.

4.4 Abläufe in ConTracts

Wie in der weiteren Darstellung deutlich wird, vereinigt das ConTract-Modell die Konzepte mehrerer anderer Ansätze. Ein Indikator für diese Tatsache ist die Menge an Grundoperationen, die zur Verfügung stehen.

Da in den bisher vorgestellten Notationen kein Programmiermodell definiert wurde, konnte auch keine Aussage über die möglichen Kontrollflußkonstrukte getroffen werden, so daß die Definition der Interpretationen direkt vorgenommen werden mußte. Im Falle der ConTracts gestaltet sich dies etwas komplexer, da mit

ConTracts ein Programmiermodell untrennbar verbunden ist und es somit notwendig ist die Ableitung einer Interpretation von einer Definition genauer zu betrachten.

Aus diesem Grund werden zunächst die Basiselemente und die Struktur eines ConTracts eingeführt, um auf dieser Basis die Operationen und die Semantik einer entsprechenden Maschine definieren zu können. Eine gesonderte Einführung der Grundelemente wird auch dadurch notwendig, daß ConTracts eine Trennung zwischen ausführenden Teilen (Steps) und Transaktionen vornehmen.

Grundsätzlich kann eine Schablone oder Template eines ConTracts mit einem beliebigen Hilfsmittel erstellt werden. In [WäRe92] und [Wäch96] wird hierfür beispielsweise eine Modula-ähnliche Sprache verwendet, während in [Schw95] eine graphische Notation eingeführt wird. Beiden Ansätzen gemeinsam ist jedoch die Tatsache, daß die Notationen in eine abstrakte Darstellung übersetzt wird, sobald eine Instanz eines ConTracts erzeugt wird. Deshalb beschäftigt sich dieser Abschnitt nur mit der formalen Notation von ConTract-Instanzen.

4.4.1 Grundelemente von ConTracts

Wie bereits in [Schw93b],[Schw94],[Seif96] und [RSS97] eingeführt wurde, wird die Beschreibung einer ConTract-Instanz auf der Basis eines Prädikat-Transitions-Netzes (*PTN*) [Brau87] vorgenommen. Wie der Name impliziert, sind PTNs Erweiterungen von Petri-Netzen. Diese Erweiterung erfolgt dahin gehend, daß Verbindungen zwischen *Stellen* und *Transitionen* mit *Prädikaten* annotiert sind. Somit ergibt sich die Semantik. daß ein *Token* nur dann von einer Stelle zu einer Transition weiter geleitet werden kann, wenn das entsprechende Prädikat erfüllt ist.

Die weiteren Unterabschnitte erläutern nun die Anwendung des PTN-Prinzips im Falle von ConTracts.

4.4.1.1 Steps

Steps stellen die aus Sicht der ConTract-Instanz atomaren Operationen dar. Sie werden mit den Stellen eines PTN assoziiert. Steps modifizieren die privaten Datenobjekte eines ConTracts (Kontext [ReSw95]) und liefern ein Resultat (z.B. erfolgreich, nicht erfolgreich, Fehler usw.)

Definition 4-5 (Kontext): *Der Kontext ς_C einer ConTract-Instanz C ist eine Menge von Kontextvariablen k, mit:*

$$k = (n,v,w, W)$$

Eine Kontextvariable ist ein Tupel (n,v,w,W), wobei n den Namen, v die Version, w den aktuellen Wert und W den Typ (oder Wertebereich) der Kontextvariablen repräsentiert.■

Auf der Ebene der ConTract-Instanz wird in anwendungsorientierte und verwaltungsorientierte Steps unterschieden. Steps der anwendungsorientierten Klasse haben dabei die Möglichkeit Datenobjekte außerhalb der ConTract-Instanz mittels sogenannter *Resource Manager* zu manipulieren. Verwaltungsorientierte Steps operieren im Gegensatz dazu nur auf dem Kontext, wie z.B. ein Step zur Inkrementierung eines Schleifenzählers und haben deshalb auch keine zugeordnete Kompensationssteps.

Eine wichtige Unterklasse der verwaltungsorientierten Steps sind die transaktionsbegrenzenden Steps: BOT, EOT, ABORT. Diese zeigen die Gruppierung von Steps zu Transaktionen an und haben spezifische Resultate. Beispielsweise hat ein Step vom Typ EOT zwei mögliche Resultate: erfolgreich und nicht erfolgreich, während ein Step vom Typ ABORT nur ein Resultat hat[1]: erfolgreich.

Definition 4-6 (Steps): *eine Stepmenge S_C einer ConTract-Instanz C ist die Vereinigungsmenge der anwendungsorientierten Steps S^a und der verwaltungsorientierten Steps S^v. Ein anwendungsorientierter Step s^a hat eine zugeordnete Menge von Resultaten $R(s^a)$, sowie eine Menge von Parametern $\Pi(s^a)$. Verwaltungsorientierte Steps s^v haben ebenfalls eine zugeordnete Menge Parametern und eine zugeordnete Menge von Resultaten die jedoch beschränkt ist:*

$$R(s^v) \subseteq \{success, nosuccess\}$$

$$S^v \supseteq \{BOT, EOT, ABORT, EOC, EVAL, ESTABLISH\}$$ ■

Verwaltungssteps stellen interne Verarbeitungssteps für eine ConTract-Instanz dar. Neben den transaktionalen Verwaltungssteps existiert für jede ConTract-Instanz noch mindestens ein spezieller Step der das Ende des ConTracts anzeigt *EOC*. Darüber hinaus gibt es weitere Verwaltungssteps, die die Evaluierung, *EVAL,* bzw. Etablierung von Invarianten (s. Abschnitt 4.4.1.4) übernehmen, *ESTABLISH*.

Wie später noch deutlich werden wird (s. Abschnitt 4.4.3), können Steps auf Grund von Schleifen in der Definition einer ConTract-Instanz mehrfach ausgeführt werden. Trotzdem ist es notwendig, die mehrfachen Ausführungen des gleichen Steps unterscheiden zu können:

Definition 4-7 (Step-Instanz): *Eine Step-Instanz $\tilde{a}$ eines Steps a einer ConTract-Instanz C ist eine eindeutig identifizierbare Version des Steps a und hat dieselben Effekte. Auf der Menge der Step-Instanzen $\{\tilde{a}_i\}$ eines Steps sei eine Totalordnung "<" definiert:*

$$\tilde{a}_i < \tilde{a}_j \Rightarrow \tilde{a}_i \text{ wurde vor } \tilde{a}_j \text{ erzeugt}$$ ■

1. Diese Vereinbarung entspricht dem *presumed abort* Protokoll [MoLi83] ohne heuristische Ausgänge

4.4.1.2 Ereignisse

Für eine ConTract-Instanz existieren interne und externe Ereignisse. Interne Ereignisse repräsentieren dabei das Resultat eines Steps während externe Ereignisse frei definiert werden können.

Definition 4-8 (Ereignis): *Eine Ereignismenge E_C einer ConTract-Instanz C ist die Vereinigungsmenge der internen Ereignisse E^i sowie der externen Ereignisse E^e. Ein internes Ereignis e^i ist ein Tupel (s, r), wobei s ein Step aus S_C und r aus der Menge der Resultate des Steps R(s) ist.*
Ein externes Ereignis e^e ist ein Tupel (◊, b), wobei ◊ einen abstrakten Step außerhalb der ConTract-Instanz repräsentiert und b ein Bezeichner des Ereignisses ist. Ein Ereignis e repräsentiert einen Wahrheitswert der angibt ob das Ereignis eingetreten ist oder nicht und ist somit ein Prädikat.

Für jede ConTract-Instanz muß mindestens das externe Ereignis “start” definiert sein. Dies wird im weiteren mit dem Tupel (◊, start) bezeichnet.

4.4.1.3 Ablaufprädikate

Zustände einer ConTract-Instanz können mit Hilfe von Prädikaten beschrieben werden. Dabei ist ein Prädikat eine Konjunktion von Prädikaten, von denen mindestens eines ein Ereignis repräsentiert. Ebenso zulässig sind Prädikate, die als logische Ausdrücke über Variablen des Kontext definiert sind. Beispielsweise wird das Verzweigungsprädikat einer If-Anweisung in der Definition eines ConTract-Templates mit Hilfe eines solchen Kontext-Wert-abhängigen Prädikates implementiert.

Definition 4-9 (Ablaufprädikat): *Ein Ablaufprädikat p der Menge von Ablaufprädikaten P_C einer ConTract-Instanz C ist eine Konjunktion von Prädikaten p_i, die eine Disjunktion von Prädikaten d_j darstellen.*

$$(p = p_1 \wedge p_2 \wedge \ldots \wedge p_n) \wedge (p_i = d_1 \vee d_2 \vee \ldots \vee d_m)$$

$$\underset{d_j}{\forall}\, d_j \in E_C \vee d_j = x_k \theta x_l,\ x_k, x_l \in \zeta_C \text{ oder konstant}, \theta \text{ Vergleichsoperator}$$

$$p_i \in p \Leftrightarrow \underset{p_k}{\exists} (p = p_1 \wedge p_2 \wedge \ldots \wedge p_k \wedge \ldots \wedge p_n) \wedge (p_i = p_k)$$

$$\underset{p_i \in p}{\exists} p_i = d_1 \wedge d_1 \in E_C \quad \blacksquare$$

Prädikate einer ConTract-Instanz stellen eine Konjunktion von Disjunktionen dar (konjunktive Normalform). Es gilt die Einschränkung, daß zu mindest ein Prädikat der konjunktiven Verknüpfung ein Ereignis ist. Somit können Prädikate nur nach dem Eintritt eines speziellen Ereignisses erfüllt sein.

Die Elementrelation $\in$ aus Definition 4-9, die für konjunktiv verknüpfte Prädikate eingeführt wurde, soll analog auch für Disjunktionen gelten.

4.4.1.4 Invarianten

Invarianten dienen in ConTracts zur Definition von Isolationsanforderungen und sind den anwendungsorientierten Steps zugeordnet. D.h., wenn für einen Ablauf die Notwendigkeit besteht, einen Zustand, der von einem Step "gesehen" wurde, für einen später auszuführenden Step wieder vorzufinden, kann dies mit Hilfe der Invarianten dem Ausführungssystem mitgeteilt werden.

Definition 4-10 (Ausgangsinvariante): *Eine Ausgangsinvariante o_k der Menge von Ausgangsinvarianten O_C einer ConTract-Instanz C ist eine Konjunktion von Invariantenprädikaten p_i:*

$$o_k = p_1 \wedge p_2 \wedge \ldots \wedge p_n, \quad n \geq 1$$ ■

Da die Details von Invarianten in Kapitel 6 noch näher besprochen werden, wird an dieser Stelle nicht näher auf die Prädikate p_i eingegangen.

Da sich, wie oben schon angedeutet, Invarianten auf einander beziehen, es ist erforderlich zunächst ein Hilfskonstrukt einzuführen.

Definition 4-11 (Prädikat-Referenz): *Eine Prädikat-Referenz $r(o_k,p_i)$ mit $p_i \in o_k$ ist ein Prädikat mit folgender Eigenschaft:*

$$r(o_k, p_i) \Leftrightarrow p_i$$ ■

Eine Prädikat-Referenz nimmt genau die Wahrheitswerte an, die das Prädikat der Ausgangsinvariante annimmt, welches referenziert wird. Mit diesem Hilfskonstrukt kann nun eine Eingangsinvariante definiert werden.

Definition 4-12 (Eingangsinvariante): *Eine Eingangsinvariante i_k der Menge von Eingangsinvarianten I_C einer ConTract-Instanz C ist eine Konjunktion von Prädikaten der folgenden Form:*

$$i_k = r_1 \wedge r_2 \wedge \ldots \wedge r_n, \; n \geq 1$$

$$\underset{r_j \in i_k}{\forall} r_j = r(o_l, p_m)$$ ■

Eingangsinvarianten sind somit eine Konjunktion von Prädikatreferenzen, die sich auf Prädikate beziehen, die Teil einer Ausgangsinvarianten sind. Einschränkungen bezüglich der Ausgangsinvarianten, die referenziert werden dürfen, ergeben sich aus den Beschränkungen die für Ausführungen gelten (siehe Abschnitt 4.4.4).

4.4.1.5 Transitionen

Eine Transition t ist eine Zuordnung eines Ablaufprädikates zu einem Step der Art, daß, wenn das Prädikat erfüllt ist, der entsprechende Step ausgeführt werden kann.

Definition 4-13 (Transition): *Eine Transition t der Menge von Transitionen T_C einer ConTract-Instanz C ist ein Tupel (p,s), mit:*

$$p \in P_C \wedge s \in S_C \quad \blacksquare$$

Transitionen der Notation für ConTract-Instanzen entsprechen somit nahezu Transitionen aus den PTN bzw. den Petri-Netzen. Unterschiede ergeben sich im Hinblick auf das *fork*-Konstrukt. Während bei den ursprünglichen PTN, das fork-Konstrukt durch eine Transition mit mehreren Ausgängen dargestellt werden kann, ist in der Notation für ConTract-Instanzen die Verwendung mehrerer Transitionen mit dem gleichen Prädikat notwendig.

4.4.1.6 Pfade

Der Begriff des Pfades wird als ein Hilfskonstrukt eingeführt, da Schleifenkonstrukte, wie später noch deutlich werden wird, mit Hilfe einer einfachen Partialordnung (siehe Definition 4-1) nur schwer formal zu beschreiben sind.

Definition 4-14 (Pfad): *Ein Pfad ist eine zweistellige Relation ⟨a,b⟩ über Steps.*

$$\langle a,b\rangle \wedge a,b \in S_C \Rightarrow \underset{t_i \in T_C}{\exists} t_i = (p,b) \ni \underset{p_j \in p}{\exists} (p_j = e \in E^i) \wedge e = (a,r)$$

$$\langle a,b\rangle \wedge \langle b,c\rangle \Rightarrow \langle a,c\rangle^+ \quad \langle a,b\rangle^+ \wedge \langle b,c\rangle \Rightarrow \langle a,c\rangle^+ \quad \langle a,b\rangle \vee \langle a,b\rangle^+ \Rightarrow \langle a,b\rangle^*$$

$$\langle b,c\rangle^* \in \langle a,d\rangle^* \Rightarrow \exists\langle a,b\rangle^* \wedge \exists\langle b,c\rangle^* \wedge \exists\langle c,d\rangle^* \text{ Elementfunktion} \quad \blacksquare$$

Mit einem Pfad wird somit die Tatsache beschrieben, daß der Abschluß eines Steps a und somit ein "Resultatsereignis" (e=(a,r)) dieses Steps eine notwendige Bedingung für die Ausführung eines Steps b ist.

Definition 4-15 (Anfangsstep): *Ein Step a ist ein Anfangsstep ⟨*,a⟩ einer ConTract-Instanz wenn gilt:*

$$\underset{t_i \in T_C}{\exists} t_i = (p,a) \wedge \underset{p_j \in p}{\exists} (p_j = e \in E^e) \wedge e = (\Diamond, start)$$

Analog wird ein Pfad $\langle *,b\rangle^*$ als Anfangsstück einer ConTract-Instanz bezeichnet, wenn gilt

$$\exists\langle *,a\rangle \wedge \exists\langle a,b\rangle^* \quad \blacksquare$$

Ein Anfangsstück einer ConTract-Instanz ist ein Pfad, für den gilt, daß die Ausführung des ersten Steps des Pfades direkt vom Startereignis der ConTract-Instanz ab-

hängt.

4.4.2 Strukturelle Beschränkungen

Die beliebige Kombination der bisher eingeführten Grundelemente kann wegen semantischer Mehrdeutigkeiten nicht zugelassen werden. Da beispielsweise die transaktionalen Operationen in ihrer Reihenfolge den Strukturbeschränkungen geschlossen geschachtelter Transaktionen unterliegen, sind entsprechende Einschränkungen notwendig.

4.4.2.1 Beschränkungen für Invarianten

Invarianten sind notwendigerweise anwendungsorientierten Steps zugeordnet. D.h. ein Step kann maximal eine Ausgangsinvariante und eine Eingangsinvariante besitzen. Da außerdem Eingangsinvarianten nur über Referenzen auf Ausgangsinvarianten aufgebaut werden können, darf die Evaluierung einer Eingangsinvariante erst erfolgen wenn alle referenzierten Ausgangsinvarianten etabliert wurden.

Bedingung 4-1: *Für alle Invarianten einer ConTract-Instanz C müssen folgende Bedingungen erfüllt sein*[1]*:*

$$\overset{\forall}{(i_k \in I_C)} \overset{\exists}{\langle i_k,s\rangle} \wedge s \in (S_C \cap S^a) \wedge \overset{\forall}{(o_k \in O_C)} \overset{\exists}{\langle s,o_k\rangle} \wedge s \in (S_C \cap S^a)$$

$$\overset{\forall}{(r_i \in i_k \in I_C, r_i = (o_l,p_j))} \; \overset{\exists}{(\langle o_l,i_k\rangle^*)} \langle i_k,o_l\rangle^* \notin \langle *,o_l\rangle^* \quad \blacksquare$$

Anmerkung: Der zweite Teil der Bedingung drückt aus, daß alle referenzierten Ausgangsinvarianten vor der referenzierenden Invarianten etabliert sein müssen. Das verwendete Kriterium scheint etwas komplex zu sein, ist jedoch notwendig, da die Pfadrelation keine Schleifen berücksichtigt.

4.4.2.2 Transaktionale Blöcke

Obwohl nicht-transaktionale Ausführungsteile in der ursprünglichen Fassung des ConTract-Modells angedacht waren und auch in neuesten Erweiterungen wieder aufgegriffen wurden [Seif96] [RSS97], können diese im Rahmen dieser Arbeit nicht berücksichtigt werden. Entsprechend restriktiv sind die hier vorgestellten Strukturbeschränkungen.

Definition 4-16 (Transaktionaler Block): *Ein transaktionaler Block* τ *einer*

1. Anstatt der Schreibweise EVAL(i) und ESTABLISH(o) wird hier nur kurz i bzw. o verwendet.

ConTract-Instanz C ist ein Tupel $(s^a, S_\tau, T_\tau, N_\tau)$ *mit folgenden Eigenschaften:*

$$S_\tau \subseteq S_C \backslash \{EOC\} \wedge T_\tau \subseteq T_C \wedge N_\tau \subseteq S_\tau \wedge s^a \in S_\tau \wedge s^a = BOT(\tau)$$

$$\forall_{(s_j \in S_\tau \backslash \{s^a\})} \langle s^a, s_j \rangle^* \wedge \forall_{(t_i \in T_C \backslash T_b, t_i = (p, s_k) \wedge s_k \in S_\tau)} s_k = s^a$$

$$N_\tau = \{s_c, s_a\} \wedge s_c = EOT(\tau) \wedge s_a = ABORT(\tau)$$

$$\forall_{(t_k \in T_\tau, t_k = (p, s_m) \wedge s_m \notin S_\tau)} \exists_{(p_i \in p)} p_i = (s_j, r) \wedge s_j \in N_\tau$$

$$\forall_{(t_i, t_k \in T_\tau, t_k = (p_k, s_k) \wedge t_i = (p_i, s_i) \wedge s_i, s_k \in N_\tau)} s_i \neq s_k \Rightarrow p_k \cap p_i = \varnothing$$

$$\forall_{(s_i \in S_\tau \backslash (N_\tau \cup \{s^a\}) \wedge s_i \in \{BOT, EOT, ABORT\})} \exists_{mit (\tau' = (t^a, S', T', N'))}$$

$$S' \subset S_\tau \wedge T' \subset T_\tau \wedge N' \subseteq S_\tau \backslash N_\tau \wedge (s_i = t^a t \vee s_i \in N')$$

s^a heißt Anfangsstep des Blockes und jedes Element aus N_τ heißt Endstep des Blockes.■

Ein transaktionaler Block ist somit eine Menge von Steps und Transitionen mit der Eigenschaft, daß es genau einen Step gibt (BOT) mit dem der Block beginnt, und von dem aus alle anderen Steps des Blockes erreicht werden können. Zusätzlich wird gefordert, daß es nur einen Step gibt, der den transaktionalen Block erfolgreich abschließen kann (EOT), und ebenso existiert nur eine Step der die Transaktion abbricht (ABORT). Die Einschränkung der Vorbedingung zur Ausführung der Steps in den zugeordneten Transitionen schließt aus, daß zwei Endsteps gleichzeitig ausgeführt werden können. Ein Verlassen des primitiven transaktionalen Blocks ist nur über Endsteps möglich (notwendige Bedingung für die entsprechenden Transitionen). Steps, die weder Anfangs- noch Endsteps sind, können transaktionalen Verwaltungssteps sein, wenn sie wiederum Anfangs- oder Endstep eines transaktionalen Blockes sind, der vollständig in dem anderen enthalten ist.

Somit repräsentiert ein transaktionaler Block eine Zusammenfassung von Steps zu einer geschlossen geschachtelten Transaktion.

4.4.2.3 Kompensationsblock

Wie bereits in Abschnitt 3.6.2 eingeführt wurde, muß zu jedem anwendungsorientierten Step ein sogenannter Kompensationsstep definiert sein. Da diese Kompensationssteps im Bedarfsfall ebenso unter dem Schutz einer ACID-Transaktion ab-

laufen, gibt es für ihre Repräsentation in einer ConTract-Instanz entsprechende Anforderungen.

Definition 4-17 (Kompensationsblock): *Ein Kompensationsblock k der Menge der Kompensationsblöcke K_C einer ConTract-Instanz C ist ein transaktionaler Block $\tau = (s^a, S_\tau, T_\tau, N_\tau)$ mit:*

$$\exists_{s \in S_\tau} s \in S^a \wedge \forall_{s_i \in S_\tau \setminus \{s\}} s_i \notin S^a$$

Die Vereinigung aller Steps einer ConTract-Instanz, die in Kompensationsblöcken enthalten sind, wird mit $\overline{S}_C$ bezeichnet:

$$\overline{S}_C = \bigcup_{(s_i)} \exists_{(k \in K_C, k = (s^a, S_k, T_k, N_k))} s_i \in S_k \quad \blacksquare$$

Somit enthält ein Kompensationsblock nur einen anwendungsorientierten Step, der den eigentlichen Kompensationsstep darstellt.

Die Zuordnung von Kompensationsblöcken zu Steps erfolgt nicht über die Definition eines Pfades sondern rein deklarativ.

Definition 4-18 (Kompensationszuordnung): *Eine zweistellige Relation comp(s, k) ist die Zuordnung eines Kompensationsblockes k zu einem Step s.*

4.4.2.4 Struktur einer ConTract-Instanz

Mit den eingeführten Definitionen und Bedingungen lassen sich nun die notwendigen strukturellen Einschränkungen einer ConTract-Instanz formulieren.

Definition 4-19 (ConTract-Instanz): *Eine ConTract-Instanz C ist ein 8-Tupel $(S_C, T_C, K_C, E_C, P_C, I_C, O_C, \varsigma_C)$, wobei S_C eine Menge von Steps, T_C eine Menge von Transitionen, K_C eine Menge von Kompensationsblöcken, E_C eine Menge von Ereignissen, P_C eine Menge von Ablaufprädikaten, I_C eine Menge von Eingangsinvarianten, O_C eine Menge von Ausgangsinvarianten und ς_C der Kontext ist.* ■

Um die Einschränkungen bezüglich der Struktur einer ConTract-Instanz zusammenzufassen wird ein sogenanntes *Wohlgeformtheitskriterium* (engl. *well-formedness*) eingeführt. Folgende Beschränkungen beschreiben dabei die notwendigen Kriterien für die Wohlgeformtheit:

Bedingung 4-2: *Alle anwendungsorientierten Steps sind in einem transaktionalen Block enthalten:*

$$\forall_{(s_i \in S_C \cap S^a)} \exists_{(\tau = (s^a, S_\tau, T_\tau, N_\tau))} s_i \in S_\tau \quad \blacksquare$$

Bedingung 4-3: *Für jeden anwendungsorientierten Step, der nicht in einem Kompensationsblock enthalten ist, gibt es einen zugeordneten Kompensationsblock:*

$$\underset{(s_i \in (S_C \backslash \bar{S}_C) \cap S^a)}{\forall} \underset{(k_j \in K_C)}{\exists} comp(s_i,k_j) \quad \blacksquare$$

Bedingung 4-4: *Es gibt mindestens eine Transition, die das Startereignis als Triggerbedingung enthält:*

$$\underset{(t \in T_C)}{\exists} t = (p,s) \wedge \underset{e \in p}{\exists} e = (\Diamond, start) \quad \blacksquare$$

Bedingung 4-5: *Alle Steps, die nicht in einem Kompensationsblock enthalten sind, sind von dem Startereignis erreichbar und es existiert ein Endstep der ConTract-Instanz, der von diesen Steps erreichbar ist:*

$$\underset{(s_i \in S_C \backslash \bar{S}_C)}{\forall} \langle *, s_i \rangle^* \wedge$$

$$\underset{(s_e \in S_C \backslash \bar{S}_C, s_e = EOC(C))}{\exists} \underset{(s_i \in S_C \backslash (\bar{S}_C \cup \{s_e\}))}{\forall} \langle s_i, s_e \rangle^* \quad \blacksquare$$

Definition 4-20 (Wohlgeformtheit): *Eine ConTract-Instanz ist wohlgeformt, wenn sie die Bedingungen 4-1 bis 4-5 erfüllt.*■

4.4.3 Interpretation einer ConTract-Instanz

Im Gegensatz zu den bisher eingeführten formalen Ablaufnotationen ist es durch die exakte Definition einer ConTract-Instanz möglich, die zugehörige Definition einer Interpretation ebenfalls sehr exakt zu fassen.

Hierzu soll zunächst der Vorgang der Interpretation durch ein Regelwerk erläutert werden:

Regel 4-1 (Ereignisregel): *Tritt ein Ereignis e aus der Menge der Ereignisse E_C einer ConTract-Instanz C ein, werden alle Transitionen aus T_C ermittelt, die das Ereignis in ihrem Prädikatteil enthalten. Für jede dieser Transitionen wird das Ereignis durch den Wahrheitswert "TRUE" ersetzt.*

Regel 4-2 (Transitionsregel): *Ist das Prädikat p einer Transition t=(p,a) aus der Menge der Transitionen T_C einer ConTract-Instanz C erfüllt, wird eine Step-Instanz ã für den Step a erzeugt und zur Ausführung gebracht. In allen Transitionen die in ihrem Prädikatteil ein Resultatsereignis des Steps enthalten e=(a,r), wird der Stepteil in dem Ereignis durch die Stepinstanz ersetzt e=(ã,r). Das Prädikat p wird durch seine ursprüngliche Definition ersetzt.*

Regel 4-3 (Stepausführung): *Soll eine Step-Instanz ausgeführt werden (Regel*

4-2) so wird nach dem Typ der Step-Instanz unterschieden.
Handelt es sich um einen anwendungsorientierten Step wird ein Auftrag an einen Step-Server veranlaßt (execute).
Ist es eine transaktionale verwaltungsorientierte Step-Instanz wird die entsprechende transaktionale Operation ausgelöst (BOT, EOT, ABORT).
Bei einer verwaltungsorientierten Step-Instanz bezüglich Invarianten wird eine entsprechende Operation zur Etablierung bzw. Evaluierung ausgelöst.

Regel 4-4 (Kompensationsregel): *Tritt das externe Ereignis e=(◊,compensate) ein, werden alle aktiven Transaktionen abgebrochen und die Kompensationsblöcke in umgekehrter Reihenfolge zu den Step-Instanzen ausgeführt, denen sie zugeordnet sind.*

Somit ergeben sich folgende Basisoperationen einer abstrakten Maschine zur Ausführung von ConTract-Instanzen.

Operation	Semantik
e: execute(C,t, $\tilde{a}_j$, k)	Führt die Step-Instanz $\tilde{a}_j$ einer ConTract-Instanz C innerhalb der Transaktion t aus und gibt den zugehörigen Kompensationsblock k an.
γ: check(C, t,i_j,$\tilde{a}_j$)	Überprüft die Invariante i_j, welche eine notwendige Bedingung zur Ausführung eines Step-Instanz $\tilde{a}_j$ darstellt.
ε: establish(C, t,o_j,$\tilde{a}_j$)	Überprüft die Ausgangsinvariante o_j und beauftragt die Ausführungsmaschine mit der Sicherstellung. Dabei beschreibt o_j einen Zustand nach der Ausführung von $\tilde{a}_j$.
b: BOT(C,t)	Beginnt eine Transaktion bezüglich einer Elterntransaktion. Wenn t einen Null-Wert annimmt handelt es sich um eine Top-Level-Transaktion
c: EOT(C,t)	Beendet t und macht die Änderungen von t dauerhaft, wenn es sich bei t um eine Top-Level-Transaktion handelt.
a: abort(C,t)	Macht die Änderungen von t rückgängig.
k: compensate(C)	Leitet die Kompensation einer ConTract-Instanz ein.

Tabelle 4-5: Operationen für das ConTract Modell

Operation	Semantik
f: EOC(C)	Beendet die Ausführung einer ConTract-Instanz C und löscht Invarianten.

Tabelle 4-5: Operationen für das ConTract Modell

Wie Tabelle 4-5 zu entnehmen ist, bestehen die Operationen einer abstrakten Maschine zur ConTract-Bearbeitung aus einer Kombination der Operationen nach Korth et. al. (siehe Abschnitt 4.4.3) und den Operationen zur Abwicklung geschlossen geschachtelter Transaktionen. Neu hinzugekommen sind lediglich eine explizite Operation zur Kompensation sowie eine Operation zur Beendigung der Bearbeitung einer ConTract-Instanz.

Erwähnenswert ist dabei die Besonderheit, daß die Operationen zur Etablierung bzw. Evaluierung von Invarianten unter der gleichen ACID-Transaktion ablaufen wie die Ausführung der Step-Instanz, der sie zugeordnet sind (siehe Bedingung 4-1 und 4-3).

4.4.4 Ausführungen und ihre Semantik

Eine formale Darstellung der Ausführung einer Contract-Instanz bzw. der Interpretation derselben, abstrahiert noch weiter von dem ursprünglich definierten Template. So ist aus dieser Interpretation nicht mehr nachvollziehbar, wie die Reihenfolge der Abarbeitung zustande kam.

Definition 4-21 (ConTract-Interpretation): *Eine Interpretation I(C) einer wohlgeformten ConTract-Instanz C ist ein Tupel (A,<), welches durch die Regeln 4-1 bis 4-4 erzeugt wurde. Die Menge A={e,γ,ε,b,c,a,k,f} stellt eine geordnete Menge von Operationen o bezüglich der Partialordnung "<" mit folgender Eigenschaft dar:*

$$\langle o_i, o_k \rangle \text{ in C} \Rightarrow \tilde{o}_i < \tilde{o}_k \quad \blacksquare$$

Existiert ein Pfad zwischen zwei Operationen o_i und o_k in einem ConTract-Template, so gilt die Partialordnung für alle Instanzen dieser Operationen.

Da die Interpretation zwingend nach den eingeführten Regeln aus einer Instanz hervor ging, sind in dieser Definition keine zusätzlichen Einschränkungen notwendig. Da auch die Semantik der Operationen bereits mit in den Regeln besprochen wurde, soll auch hier keine weitere Ergänzung gemacht werden.

Anders verhält sich dies bezüglich der Semantik des Ablaufes als Ganzes. Für ConTracts wird die Fortsetzbarkeit garantiert, was bedeutet, daß nach einem Systemausfall alle offenen Transaktionen zurückgesetzt werden und die Verarbeitung dann nach Vorne fortgesetzt wird, d.h. der Ablauf wird damit fortgesetzt, daß die

BOT Operationen der abgebrochenen Top-level-Transaktionen erneut ausgeführt werden.

Auch die Reaktion auf das Kompensationsereignis soll noch etwas eingehender besprochen werden. Wie bereits gesagt, hält die compensate Operation einen aktiven Ablauf an, bricht alle aktiven Transaktionen ab und setzt den Ablauf mit der Abarbeitung der Kompensationsblöcke fort. Dies kann als eine Erweiterung der Menge A sowie der Partialordnung aufgefaßt werden und somit als eine Modifikation des Ablaufs selbst. Diese spielt insbesondere deswegen eine Rolle, da im Fehlerfalle der Vorgang auch während der Kompensation wieder fortsetzbar und somit wieder herstellbar sein muß. Somit erzeugt die Ausführung der compensate-Operation einen persistenten Zustand, aus dem sich die Kompensationsoperationen und ihre Abarbeitungsreihenfolge ableiten lassen.

4.4.5 Ein Anwendungsbeispiel

Anhand des eingeführten Beispiels aus Kapitel 2 kann ein Eindruck davon gegeben werden, wie die Umsetzung von der Definition der Schablone bis hin zu der Interpretation erfolgt. Da die vollständige Präsentation der verschiedenen Stufen der Abstraktion an dieser Stelle zu umfangreich werden würde, beschränkt sich die Diskussion auf die Umgebung eines spezifischen Steps: “Antragstellung” (siehe Abbildung 2-1 auf Seite 8).

Der Step “Antragsstellung” soll nach dem dargestellten Kontrollfluß nach dem Step “Dokumentenbereitstellung” und vor dem Step “Überprüfung der Zulässigkeit” ausgeführt werden. Wie in Tabelle 2-1 auf Seite 11 zusätzlich erläutert wird ist die zugeordnete Kompensation der leere Step. Da keine weiteren Aussagen über Transaktionsgrenzen und Invarianten getroffen werden, wird implizit sowohl angenommen, daß der Step unter einer Top-Level-Transaktion ausgeführt wird, als auch daß die Invarianten dem Prädikat “TRUE” entsprechen und somit einen konstanten Wahrheitswert annehmen.

4.4.5.1 Darstellung als Instanz

Umgesetzt in die Notation für ConTract-Instanzen ergibt dies für die betrachtete Teilmenge der Steps:.

Step	Bedeutung
a_1: EOT(t_1)	Beendet vorhergehende Transaktion
a_2: BOT(t_2)	Beginnt neue Transaktion

Tabelle 4-6: Steps der ConTract-Instanz für das Beispiel

Step	Bedeutung
a_3: EVAL(i=TRUE)	Evaluiert Eingangsinvariante von “Antragsstellung”
a_4: “Antragsstellung”	Eigentlicher Step
a_5: ESTABLISH(o=TRUE)	Etabliert Ausgangsinvariante
a_6: EOT(t_2)	Beendet die Transaktion um “Antragsstellung”
a_7: ABORT(t_2)	Setzt Transaktion zurück
a_8: BOT(t_3)	Beginnt Transaktion für Kompensationsstep
a_9: NullStep	Kompensationsstep
a_{10}: EOT(t_3)	beendet Transaktion für Kompensationsstep
a_{11}: ABORT(t_3)	Setzt Transaktion für Kompensationsstep zurück

Tabelle 4-6: Steps der ConTract-Instanz für das Beispiel

Diesen Steps sind natürlich entsprechende Resultatsereignisse zugeordnet. Um die Darstellung nicht unnötig komplex zu gestalten, werden nicht alle Fälle aufgeführt. So haben beispielsweise die Steps für die Kompensation entsprechende Ereignisse wie die “normalen” Steps und allen Steps außer den ABORT-Steps ist normalerweise auch ein “nicht erfolgreich” Ereignis zugeordnet.

Ereignis	Bedeutung
e_1: (a_1, SUCCESS)	Vorhergehende Transaktion erfolgreich
e_2: (a_2, SUCCESS)	Beginn neuer Transaktion erfolgreich
e_3: (a_3, SUCCESS)	Evaluierung der Eingangsinvarianten erfolgreich
e_4: (a_4, SUCCESS)	Eigentlicher Step war erfolgreich
e_5: (a_4, NOSUCCESS)	Step “Antragsstellung” war nicht erfolgreich
e_6: (a_5, SUCCESS)	Etablierung der Ausgangsinvariante war erfolgreich

Tabelle 4-7: Ereignisse der ConTract-Instanz für das Beispiel

Ereignis	Bedeutung
e_7: (a_6, SUCCESS)	Transaktion um “Antragsstellung” erfolgreich abgeschlossen
e_8: (a_6, NOSUCCESS)	Abschluß der Transaktion nicht erfolgreich.
e_9: (a_7, SUCCESS)	Transaktion zurückgesetzt

Tabelle 4-7: Ereignisse der ConTract-Instanz für das Beispiel

Mit Hilfe der Ereignisse und der Steps können nun die Transitionen definiert werden. Auch hier wird wiederum auf die explizite Darstellung der Transitionen für den Kompensationsblock verzichtet, da die entsprechenden Transitionen analog festgelegt sind.

Transition	Bedeutung
t_1: (e_1,s_2)	Erfolgreicher Abschluß der vorigen Transaktion startet die Transaktion für “Antragsstellung”
t_2: (e_2, s_3)	Wenn Transaktion begonnen wurde, evaluiere Invariante
t_3: (e_3, s_4)	Wenn Evaluierung erfolgreich, starte “Antragsstellung”.
t_4: (e_4, s_5)	Wenn “Antragstellung” erfolgreich leite Etablierung der Ausgangsinvarianten ein.
t_5: (e_5, s_7)	Wenn “Antragstellung” nicht erfolgreich setze Transaktion zurück
t_6: (e_6, s_6)	Wenn Etablierung der Ausgangsinvariante erfolgreich, schließe Transaktion ab
t_7: (e_7, Nextstep)	Wenn Transaktion um “Antragsstellung” erfolgreich abgeschlossen, starte Nextstep
t_8: (e_8, s_2)	Transaktionsabschluß nicht erfolgreich, somit Transaktion zurückgesetzt. Starte Transaktion erneut.
t_9: (e_9, s_2)	Transaktion zurückgesetzt; Neustart

Tabelle 4-8: Transitionen der ConTract-Instanz für das Beispiel

4.4.5.2 Grafische Darstellung

Da die Zusammenhänge der mengenorientierten Darstellung nur schwer nachvollziehbar sind, wird in Abbildung 4-2 eine grafische Repräsentation des Ausschnitts der ConTract-Instanz gegeben.

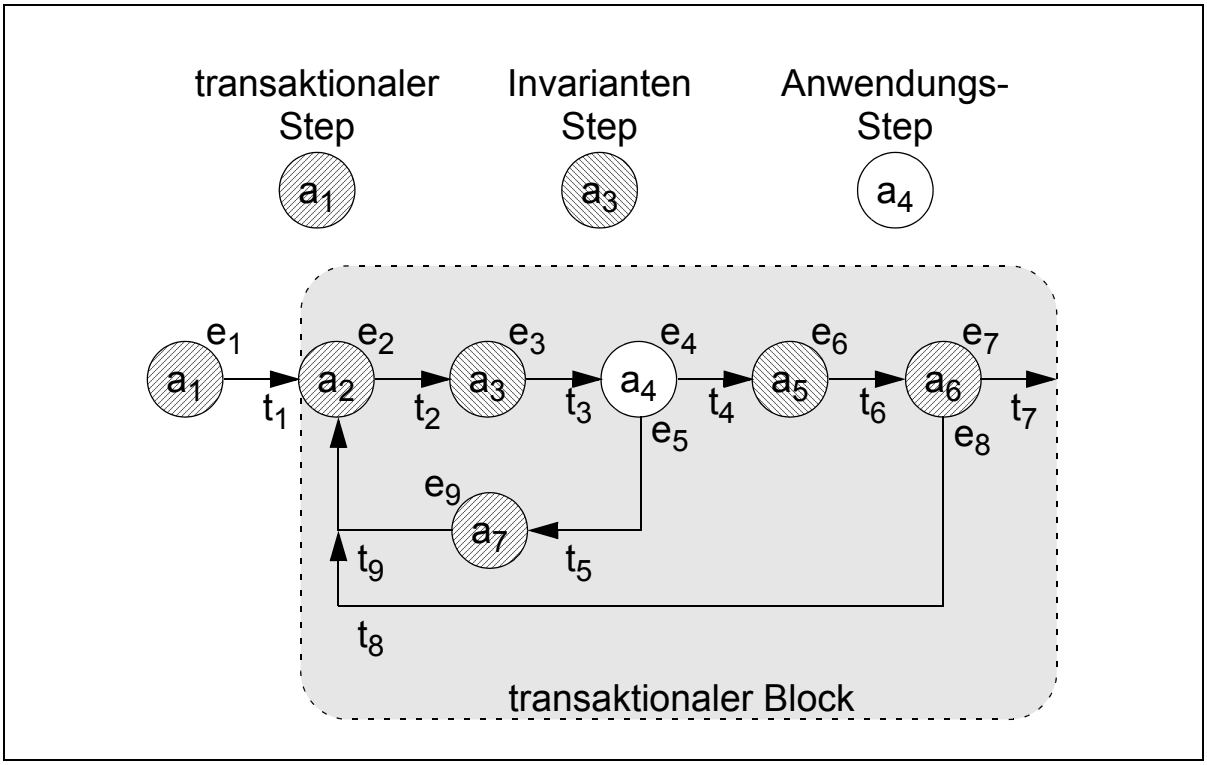

Abbildung 4-2: Grafische Darstellung des Beispielausschnitts

4.4.5.3 Interpretation

Das Problem der Interpretation einer ConTract-Instanz ist bereits an dem relativ simplen Ausschnitt zu erklären. Betrachtet man die Transitionen t_8 und t_9 so stellt man fest, daß durch diese Transitionen ein Zyklus in der Instanz entsteht, d.h. die Steps a_2 bis a_7 können mehrfach durchlaufen werden. Ebenso wird durch die Definition mehrerer Transitionen, die auf unterschiedlichen Ereignissen eines Steps definiert sind eine bedingte Verzweigung modelliert, so daß auf Grund der vorliegenden Spezifikation des Ausschnittes der Instanz nicht festgelegt ist, wie die Interpretation, die durch eine tatsächlichen Ausführung entsteht, aussehen wird. Trotzdem soll für eine fiktive Ausführung eine Darstellung einer Instanz gegeben werden.

Für die fiktive Ausführung soll angenommen werden, daß die Steps a_1 bis a_3 zunächst erfolgreich durchlaufen werden und dann die Ausführung von Step a_4 fehlschlägt. Anschließend soll Step a_7 erfolgreich durchlaufen werden und die nachfolgenden Steps a_2 bis a_6 ebenfalls erfolgreich abgeschlossen werden.

Folgende Tabelle gibt somit Aufschluß über die Menge A der ausgeführten Opera-

tionen:.

Step-Instanz	von Step	Operation
$\tilde{a}_{1,1}$	a_1	o_1=c=EOT(C,t_1)
$\tilde{a}_{2,1}$	a_2	o_2=b=BOT(C,t_2)
$\tilde{a}_{3,1}$	a_3	o_3=γ=check(C,t_2,i)
$\tilde{a}_{4,1}$	a_4	o_4=e=execute(C,t_2,$\tilde{a}_{4,1}$,k)
$\tilde{a}_{7,1}$	a_7	o_5=a=abort(C,t_2)
$\tilde{a}_{2,2}$	a_2	o_6=b=BOT(C,t_2)
$\tilde{a}_{3,2}$	a_3	o_7=γ=check(C,t_2,i)
$\tilde{a}_{4,2}$	a_4	o_8=e=execute(C,t_2,$\tilde{a}_{4,2}$,k)
$\tilde{a}_{5,1}$	a_5	o_9=ε=establish(C,t_2,o)
$\tilde{a}_{6,1}$	a_6	o_{10}=c=EOT(C,t_2)

Tabelle 4-9: Operationen einer Interpretation

Die geltende Partialordnung ist in diesem Fall intuitiv durch die Indizierung der Operationen klar. Bei dem betrachteten Ausschnitt ist dies auch trivial, da keine Steps parallel ausgeführt wurden und somit die Partialordnung einer Totalordnung entspricht.

4.5 Weitere Notationen

Neben den vorgestellten Notationen gibt es noch weitere, die jedoch entweder keinen weiteren Beitrag zu dem eigentlichen Thema dieser Arbeit leisten oder aber nur zur Beurteilung anderer Modelle entwickelt wurden. Trotzdem werden sie an dieser Stelle berücksichtigt, um einen tieferen Einblick in die Tätigkeiten auf diesem Gebiet zu geben.

4.5.1 ECA-Regeln

Äußerst flexibel ist der Ansatz von Dayal et. al. [DHL90][DHL91]. Durch die Einführung der sogenannten *ECA-Regeln* (*Event-Condition-Action*) erhält man potentiell die gleiche Ausdrucksmächtigkeit wie durch die hier verwendeten PTN zur Beschreibung einer ConTract-Instanz. Allerdings wird keine Aussage über

Einschränkungen gemacht und auch kein Kriterium für die Wohlgeformtheit definiert, so daß als Folge davon auch keine Korrektheitsaussagen möglich sind.

Eine weitere Eigenschaft der ECA-Regeln verschärft dieses Problem weiter. Durch die Einführung sogenannter *Kopplungsmodi* (engl. *coupling modes*), die es ermöglichen, die Auswertung einer Regel zu verzögern oder sogar von einer aktuell laufenden Transaktion abzuspalten, läßt sich keine Aussage mehr über die (transaktionale) Semantik eines Ablaufs treffen.

Trotz der Tatsache, daß es auf der Basis der ECA-Regeln nicht möglich ist, Aussagen die Korrektheit eines Ablaufs zu treffen, sind diese mit als Einzigste der hier vorgestellten Notationen (außer den ACID-Transaktionen) zumindest teilweise implementiert - sogenannte aktive Datenbanksysteme [Daya88] bieten meist eine Teilmenge der Möglichkeiten der ECA-Regeln an.

4.5.2 ACTA

Der sogenannte ACTA-Formalismus [ChRa90][ChRa92] wurde entwickelt, um den Vergleich von transaktionalen Ablaufmodellen zu ermöglichen. Da der Formalismus im Kontext der "*erweiterten Transaktionsmodelle*" (engl. *extended transaction models*) entstand, basiert der Ansatz auf der axiomatischen Erfassung der Modelle und der prädikatenlogischen Beschreibung der Abhängigkeiten der transaktionalen Operationen. Somit werden die Modelle nicht als Ganzes von dem Formalismus beschrieben, sondern nur ihre transaktionalen Eigenschaften (siehe Abschnitt 2.2.2.4).

Eine Besonderheit von ACTA ist die Möglichkeit, auch Datenkonflikte formal erfassen zu können, was durch das Invariantenkonzept der ConTracts abgedeckt wird. ACTA verfolgt in diesem Bereich aber einen sehr flexiblen Ansatz, wodurch die Modellierung verschiedenster Konfliktrelationen (siehe Kapitel 5) möglich wird.

Neuere Arbeiten versuchen, den Ansatz für die Synthese von Transaktionsmodellen zu erschließen [ChRa94], um so aus einer formalen Spezifikation der Anforderungen ein Transaktionsmodell generieren zu können.

4.5.3 Abhängigkeitsregeln nach Klein

Eng verwandt mit dem ACTA-Modell ist der Ansatz von Klein [Klei91]. Der Ansatz von Klein hat zum Ziel, die transaktionale Semantik verteilter Abläufe formal zu beschreiben. Die dazu verwendete Notation basiert ebenfalls auf der Definition von Abhängigkeiten transaktionaler Ereignisse, wie z.B. create, commit und abort. Unterschiede zu dem in ACTA gewählten Ansatz ergeben sich bei der Mächtigkeit der gewählten Notation. Während ACTA auf Prädikatenlogik basiert, verwendet Klein Ausdrücke der Aussagenlogik zur Beschreibung der Abhängigkeiten.

Erwähnenswert ist jedoch, daß die Arbeiten von Klein in eine Architektur eines Laufzeitsystems umgesetzt wurden [Günt96], während das ACTA Modell bisher nur zu theoretischen Beurteilungen oder zu Spezifikationen eingesetzt wird.

5 Korrektheit

Die im vorigen Kapitel eingeführten Modelle dienen dazu Vorgänge formal zu erfassen. Darüber hinaus stellen sie aber auch eine Notation zur Verfügung, die eine Automatisierung ermöglicht. Im Folgenden soll die Definition eines Vorganges, die mittels einer der formalen Notationen erstellt wurde, als Ablauf bezeichnet werden. Die automatisierte Abwicklung eines solchen Ablaufs wird als Ausführung bezeichnet.

Die Besonderheit transaktionaler Ablaufmodelle besteht darin, daß diese zumindest teilweise die Semantik von Abläufen festlegen. Damit für eine Ausführung beurteilt werden kann, ob die semantischen Vorgaben erfüllt wurden, wird üblicherweise ein formales *Korrektheitskriterium* eingeführt. In diesem Kapitel werden verschiedene solcher Korrektheitskriterien vorgestellt und ein neues Kriterium für das ConTract-Modell entwickelt.

Wie sich im Verlauf dieses Kapitels noch zeigen wird, können zwei Arten von Kriterien unterschieden werden:

1. Kriterien, die es erlauben, die Korrektheit abgeschlossener Ausführungen zu beurteilen.
2. Kriterien, die es zu jedem Zeitpunkt einer Ausführung erlauben, die Korrektheit zu beurteilen.

Da das Ziel transaktionaler Ausführungsmodelle die Bereitstellung eines Laufzeitsystems zur Garantie der semantischen Vorgaben ist, liegt klar auf der Hand, daß Kriterien der ersten Kategorie schlecht für die reale Umsetzung geeignet sind. Allerdings lassen sich mit diesen Kriterien die Menge von zulässigen Abläufen relativ einfach definieren weshalb sie als Referenz herangezogen werden.

5.1 Grundlagen

Trotz der Unterschiede im Detail beruhen alle Korrektheitskriterien auf ähnlichen Grundbegriffen und formalen Konstruktionen. Dieser Abschnitt stellt diese gemeinsamen Grundbegriffe vor.

5.1.1 Historien

Wie eingangs dieses Kapitels erwähnt, beschäftigen sich Korrektheitskriterien mit der Beurteilung von Ausführungen. Zu diesem Zweck ist es zunächst notwendig,

eine Ausführung selbst formal beschreiben zu können. Ein Ansatz hierfür ist die Definition sogenannter *Historien* [BHG87], die ursprünglich zur Entwicklung von Korrektheitskriterien für ACID-Transaktionen entwickelt wurden. Verallgemeinert man aber das Konzept der Historien auf der Basis der im Kapitel 4 eingeführten abstrakten Maschinen, lassen sich damit sehr viele transaktionale Ablaufmodellen beschreiben.

Definition 5-1 (Historie): *Eine Historie H eines Systems zur Ausführung von Abläufen, die mit einer bestimmten abstrakten Maschine ausgeführt werden können, ist ein Partialordnung (Σ, $<_H$), wobei Σ die Menge der von der abstrakten Maschine ausgeführten Operationen und "$<_H$" eine binäre, nicht reflexive und transitive Relation ist. Die Partialordnung bestimmt die Reihenfolge, in der die Operationen ausgeführt wurden, sofern diese Reihenfolge bedeutsam ist.*■

Eine wichtige Grundannahme bei Historien ist, daß Operationen der abstrakten Maschine atomar sind und somit nur dann in der Historie erscheinen, wenn sie vollständig ausgeführt wurden. Da die parallele Bearbeitung von Operationen zugelassen wird, handelt es sich bei der Relation "$<_H$" nicht um eine Totalordnung.

Definition 5-2 (Präfix): *Ein Präfix H' einer Historie H ist eine Partialordnung (Σ', $<'$) mit:*

$$\Sigma' \subseteq \Sigma \wedge \left(\forall_{a,b \in \Sigma'} a <_H b \Leftrightarrow a <' b \right) \wedge \left(\forall_{a \in \Sigma'} \forall_{b \in \Sigma} b <_H a \Rightarrow b \in \Sigma' \right) \quad \blacksquare$$

Ein Präfix einer Historie ist eine Einschränkung der Menge der Operationen derart, daß für jede Operation in der reduzierten Menge auch alle Vorgängeroperationen bezüglich der Ordnungsrelation $<_H$ in der reduzierten Menge enthalten sind.

5.1.2 Kommutativität und Konflikte

Als Basis aller hier besprochenen Korrektheitskriterien dienen Historien, deren Korrektheit trivialerweise erfüllt ist. Eine konkret gegebene Historie gilt dann als korrekt, wenn sie durch eine endliche Anzahl von Umordnungen der Operationen in eine als korrekt definierte Historie überführt werden kann.

Hierzu muß zunächst geklärt werden, welche Umordnungen zulässig sind und welche nicht. Ein weit verbreitetes Kriterium ist dabei die Kommutativität [Papa86] von Operationen:

Definition 5-3 (Kommutativität): *Zwei Operationen a und b einer Historie H sind kommutativ, wenn gilt, daß das Ergebnis der Ausführungsfolge "ab" mit dem Ergebnis der Ausführungsfolge "ba" identisch ist.*■

Umgangssprachlich formuliert bedeutet die Kommutativität zweier Operationen, daß es keine Rolle spielt ob zuerst die eine und dann die andere Operation ausgeführt wird oder ob dies umgekehrt geschieht. Aus formaler Sicht liegt die eigent-

liche Problematik in der Definition der Identität. Obwohl intuitiv angenommen wird, daß hierbei die Identität der Werteausprägungen aller berührten Datenobjekte gemeint ist, muß dies nicht unbedingt der Fall sein. Beispielsweise können die zwei Ausführungsfolgen aus einer Anwendungssicht identisch in dem Sinne sein, daß nur eine bestimmte Bedingung erfüllt sein muß (z.B. Kontostand größer 1000). Somit wird der Begriff der Identität irreführend und es bietet sich an, ein alternatives Kriterium zu benutzen.

Eine weit verbreitete Alternative zur Verwendung der Kommutativität als Basis eines Korrektheitskriteriums ist die Einführung einer sogenannten *Konfliktrelation*. Statt zu definieren welche Operationen vertauscht werden können, wird festgelegt, welche Operationen nicht vertauscht werden dürfen.

Definition 5-4 (Konfliktordnung): *Stehen zwei Operationen a und b einer Historie H in einem Konflikt bezüglich einer Konfliktrelation conflict(a,b), so muß entweder* $a <_H b$ *oder* $b <_H a$ *gelten und a und b dürfen in H nicht vertauscht werden.*■

Kriterien, die auf Konfliktrelationen basieren, definieren eine Historie dann als korrekt, wenn sie durch endlich viele Vertauschungen von nicht in Konflikt stehenden Operationen in eine als korrekt definierte Historie umgeformt werden kann.

Da an dieser Stelle nur das Prinzip der Konfliktrelation deutlich gemacht werden soll, wird auf die Einführung der Relation selbst verzichtet. Wie im weiteren Verlauf dieses Kapitels aber deutlich werden wird, unterscheiden sich die verschiedenen Korrektheitsdefinitionen im wesentlichen bezüglich der verwendeten Konfliktrelation.

5.1.3 Isolation und Atomarität

Wie bereits in Kapitel 3 erläutert wurde, führen alle transaktionalen Ablaufmodelle mehr oder minder stark ausgeprägte semantische Einschränkungen bezüglich der Isoliertheit und der Atomarität von Abläufen ein. Da diese zwei Eigenschaften zunächst unabhängig voneinander sind, wurden die entsprechenden Korrektheitskriterien auch getrennt voneinander entwickelt [BHG87]. Wie sich im weiteren Verlauf dieses Kapitels herauskristallisieren wird, ist dies jedoch kein sinnvoller Ansatz, da die Eigenschaft der Atomarität einige Voraussetzungen bezüglich der Isolationseigenschaft fordert.

Insbesondere spielt dieser Zusammenhang dann eine Rolle, wenn eine Atomaritätseigenschaft im Zusammenhang mit der Wiederherstellung (engl. *Recovery*) eines konsistenten Zustandes nach einem Systemausfall betrachtet wird. Da im Falle von ConTracts der Aspekt der Zuverlässigkeit eine maßgebliche Aufgabe darstellt, wird deshalb auf eine getrennte Betrachtung von Korrektheitskriterien bezüglich der Permeabilität bzw. Atomarität verzichtet werden.

5.1.4 Anwendbarkeit

Korrektheitskriterien im transaktionalen Umfeld sind prinzipiell nur dann von praktischer Relevanz, wenn sie auch in einem Laufzeitsystem implementiert werden können. Wie eingangs dieses Kapitels bereits erwähnt wird dies nicht von allen Kriterien gewährleistet, da mit einigen Kriterien nur vollständig ausgeführte (abgeschlossene) Abläufe beurteilt werden können.

Formalisiert man diese Aussage auf der Basis des oben eingeführten Begriffs der Historie läßt sich die Anwendbarkeit eines Kriteriums schnell beurteilen.

Definition 5-5 (Präfix-abgeschlossen): *Ein Korrektheitskriterium, welches die Korrektheit einer Historie beurteilt heißt Präfix-abgeschlossen wenn gilt, daß aus der Korrektheit einer Historie auch die Korrektheit für jeden beliebigen Präfix der Historie folgt.*■

Ist nun eine Historie gemäß eines Präfix-abgeschlossenen Korrektheitskriteriums korrekt, müssen auch alle Präfixe korrekt sein. Daraus läßt sich ableiten, daß eine Ausführung eines Ablaufs nur dann korrekt sein kann, wenn bereits alle (Teil-)Historien, die während der Ausführung auftreten auch korrekt sind. Ist, im Gegensatz dazu, ein Korrektheitskriterium nicht Präfix-abgeschlossen, ist es während der Ausführung eines Ablaufs zulässig, daß nicht korrekte Teilhistorien auftreten.

Somit wird für praktisch anwendbare Korrektheitskriterien gefordert, daß sie Präfix-abgeschlossen sind. Umgesetzt werden die Präfix-abgeschlossenen Kriterien in einem sogenannten *Scheduler*. Diesem werden die Operationen der abstrakten Maschine übergeben, damit über die Zulässigkeit der Ausführung der Operationen entschieden werden kann. Dies bedeutet, daß der Scheduler überprüft, ob die Ausführung einer Operation eine nicht korrekte Historie erzeugen würde und somit die Operation zurückweist. Die Verfahren die dabei verwendet werden und die Informationen, die einem Scheduler zur Verfügung stehen müssen, um diese Entscheidung treffen zu können, werden im folgenden Kapitel vorgestellt werden.

5.2 Klassische Korrektheitskriterien

Klassische Korrektheitskriterien der hier betrachteten Art wurden zunächst für ACID-Transaktionen entwickelt; sie waren zur automatischen Kontrolle des gleichzeitigen Zugriffs auf gemeinsame Datenbestände notwendig. Im Gegensatz dazu war bei der zeitlich vorher liegenden Verwendung von Datenbanksystemen keine solchen Einschränkungen notwendig, da Zugriffe nacheinander in einem sogenannten *Batch-Betrieb* erfolgten.

Wie sich im weiteren Verlauf dieses Abschnitts noch herausstellen wird reichen diese Ansätze weiter, so daß gewisse Grundprinzipien der Korrektheitskriterien für ACID-Transaktionen auch in weit komplexeren Ablaufmodellen wiederzufin-

den sind. Das liegt daran, daß eine Eigenschaft von ACID-Transaktionen auch von allen anderen transaktionalen Ablaufmodellen übernommen wurde: die Konsistenzerhaltung. An dieser Stelle soll deshalb nochmals kurz wiederholt werden, was diese Eigenschaft bedeutet.

Die Eigenschaft der Konsistenzerhaltung besagt, daß wenn ein Ablauf, der diese Eigenschaft besitzt, auf einem konsistenten Datenbestand ausgeführt wird, der Datenbestand am Ende der Ausführung wiederum in einem konsistenten Zustand ist. Dies gilt natürlich nur unter der Annahme, daß die Ausführung nicht durch andere Ausführungen gestört wird.

5.2.1 Grundprobleme der ACID-Transaktionen

Wie bereits in Abschnitt 3.1.1 kurz erwähnt wurde, kommt es bei der parallelen Ausführung mehrerer ACID-Transaktionen zu Problemen bezüglich der Gewährleistung der ACID-Semantik. Da die Darstellung in Kapitel 3 nur zur Motivation diente, wurden diese Probleme nur informell beschrieben. Dieser Abschnitt dient nun dazu das Problem formal zu erfassen, um auf dieser Basis ein entsprechendes Korrektheitskriterium definieren zu können.

Betrachtet man die Historie $(\Sigma, <)$ einer abstrakten Maschine zur Abarbeitung von ACID-Transaktionen nach dem read/write-Modell (siehe Kapitel 4 Abschnitt 4.1), so enthält die Menge Σ nur die Operationen read, write, abort und commit verschiedener Transaktionen. Trotzdem lassen sich alle Konflikte, die zu einer Verletzung der ACID-Eigenschaften führen können, hinreichend mit dieser Menge von Operationen beschreiben [GrRe93].

5.2.1.1 Der read/write Konflikt

Findet man in einer Historie eine Lese-Operation read(t, a), die ein Objekt a innerhalb einer Transaktion t liest und eine nachfolgende Schreib-Operation write(s, a), die das selbe Objekt a schreibt, ohne daß die Transaktion t abgeschlossen wurde (mittels commit oder abort) liegt ein potentieller Konflikt der Transaktionen t und s vor. Das Lesen des Objektes a ist nicht unbedingt wiederholbar, d.h. es würde wahrscheinlich ein anderes Resultat liefern (engl. *unrepeatable read*). Da jedoch durch die Isolationseigenschaft dieses wiederholbare Lesen garantiert wird (solange die Transaktion den Wert nicht selbst ändert), wäre somit eine zugesicherte semantische Eigenschaft verletzt und die Ausführung nicht mehr korrekt.

5.2.1.2 Der write/read Konflikt

Der write/read Konflikt bildet das genaue Gegenstück zum read/write Fall. Eine Transaktion t führt also eine Schreiboperation auf ein Objekt a aus, während die Transaktion s das Objekt a liest. Auch diese Konstellation stellt wieder einen po-

tentiellen Konflikt dar, da die Transaktion s einen Wert liest der nicht unbedingt am Ende einer Transaktion für das Objekt a gilt (engl. *dirty read*). Dies kann dadurch verursacht werden, daß Transaktion t das Objekt a ein zweites mal schreibt und a somit einen anderen Wert erhält.

5.2.1.3 Der write/write Konflikt

Im Falle des write/write Konflikts greifen zwei Transaktionen schreibend auf das gleiche Objekt zu. Wie in den vorigen Fällen soll Transaktion t vor der Transaktion s eine Schreiboperation auf a ausgeführt haben. Wiederum liegt ein potentieller Konflikt vor, da eine folgende Leseoperation von t den Wert liefern würde den s geschrieben hat und somit eine Änderung von t verloren gegangen wäre (engl. *lost update*). Somit wäre die Isolationseigenschaft von t verletzt.

Ein anderer Aspekt bei dem write/write Konflikt ist das Verhalten durch die Atomaritätseigenschaft. Da es für ACID-Transaktionen garantiert sein muß, daß im Falle eines Zurücksetzens der Zustand vor ihrer Ausführung wieder hergestellt wird, wirkt sich der Konflikt im Falle eines Zurücksetzens von t oder t und s fatal aus. Wird beispielsweise t zurückgesetzt, werden die Änderungen von s ebenso unwirksam. Werden beide Transaktionen zurückgesetzt, hängt das Ergebnis von der Reihenfolge des Zurücksetzens ab. Wird t vor s zurückgesetzt, so stellt das Zurücksetzen von s den Zustand nach der Schreiboperation von t wieder her.

Wie an dem letzten Beispiel sichtbar wird, muß ein abstraktes transaktionsverabeitendes System mit ACID-Semantik vor jedem Schreibzugriff einen Lesezugriff ausführen, um den Zustand vor Ausführung der Transaktion wieder herstellen zu können. Deshalb wird in einigen Korrektheitsbetrachtungen implizit vorausgesetzt, daß vor jedem Schreibzugriff ein Lesezugriff erfolgt ist (siehe Abschnitt 5.2.3).

5.2.2 Klassische Serialisierbarkeit

Separiert man zunächst die Atomarität und die Dauerhaftigkeit von den Eigenschaften der Konsistenzerhaltung und Isoliertheit und betrachtet nur die zwei zuletzt genannten Eigenschaften, so ist intuitiv klar, daß alle seriellen Ausführungen von Transaktionen korrekt sein müssen. Dies folgt daraus, daß bei der seriellen Ausführung von Transaktionen (also dem klassischen Batch-Betrieb) keine gegenseitige Beeinflussung von aktiven Transaktionen vorkommen kann und somit auf Grund der Konsistenzerhaltung jeder einzelnen Transaktion die entstehende serielle Historie korrekt sein muß.

Genau diese Beobachtung bildet die Basis des Korrektheitskriteriums von ACID-Transaktionen. Ausgehend von der Menge der seriellen Ausführungen von Transaktionen kann für eine beliebige Ausführung abgeschlossener Transaktionen ent-

schieden werden, ob diese ebenfalls korrekt ist. Das Kriterium hierfür ist einfach die Äquivalenz der Reihenfolge der Operationen in der zu beurteilenden Ausführung mit der Reihenfolge einer (beliebigen) seriellen Ausführung. Dieses Kriterium soll nun formal dargestellt werden.

Definition 5-6 (Vollständigkeit): *Eine Historie $H = (\Sigma,<)$ heißt vollständig, wenn sie nur Operationen abgeschlossener Transaktionen enthält:*

$$\forall_{(read(t,o),write(t,o) \in \Sigma)} \exists commit(t) \in \Sigma \vee \exists abort(t) \in \Sigma \quad \blacksquare$$

Definition 5-7 (Serielle Historien): *Eine vollständige Historie H heißt seriell, wenn für alle vorkommenden Transaktionen sämtliche Operationen unmittelbar aufeinanderfolgen, ohne daß Operationen anderer Transaktionen dazwischen liegen.* ■

Damit nun eine beliebige Historie mit einer seriellen vergleichbar wird, ist es zunächst notwendig die Projektion der beliebigen Historie auf eine vollständige Historie vorzunehmen.

Definition 5-8 (Commit-Projektion): *Die Commit-Projektion C(H) einer Historie $H = (\Sigma,<)$ entsteht durch die Eliminierung aller Operationen von nicht abgeschlossenen Transaktionen (Transaktionen bei denen weder commit noch abort in Σ ist) aus Σ.* ■

Durch die Commit-Projektion erhält man somit eine vollständige Historie, die mit seriellen Historien vergleichbar ist.

Definition 5-9 (Historien-Äquivalenz): *Zwei Historien $H_1 = (\Sigma_1, <_1)$ und $H_2 = (\Sigma_2, <_2)$ sind äquivalent, wenn $\Sigma_1=\Sigma_2$ ist und alle Paare von Operationen die in einem Konflikt stehen (siehe Abschnitt 5.2.1) in der gleichen Ordnungsrelation bezüglich $<_1$ bzw. $<_2$ stehen.* ■

Die eigentliche Definition der *Serialisierbarkeit* ergibt sich nun relativ offensichtlich aus den Definitionen 5-7 bis 5-9. ■

Definition 5-10 (Serialisierbarkeit): *Eine Historie H ist serialisierbar, wenn ihre Commit-Projektion C(H) äquivalent zu einer seriellen Historie H_s ist.*

Die Menge aller serialisierbaren Historien wird mit SR bezeichnet. ■

Unglücklicherweise ist die Serialisierbarkeit nur für abgeschlossenen Historien definiert, so daß für nicht abgeschlossene Historien auch nicht entschieden werden kann, ob sie korrekt sind oder nicht. Dies hat zur Folge, daß auf der Basis dieses Kriteriums erst am Ende einer Transaktion entschieden werden kann, ob ihre Ausführung serialisierbar war oder nicht.

Um bereits während der Ausführung von Transaktionen entscheiden zu können ob die Historie serialisierbar sein kann oder nicht, wurde nach einem alternativen Kri-

terium gesucht. Dieses beruht auf der Beobachtung, daß die in Abschnitt 5.2.1 eingeführten Konflikte für sich allein genommen keine der semantischen Garantien verletzen. Erst wenn ein (bestimmter) zweiter Konflikt hinzukommt, wird auch tatsächlich eine der Garantien verletzt.

Die eigentliche Ursache hierfür liegt in der Auswirkung der eingeführten Konflikte auf das Verhältnis der Transaktionen als ganzes. Besteht ein Konflikt zwischen zwei Operationen unterschiedlicher Transaktionen, so impliziert die Reihenfolge der Ausführung der Operationen eine Ordnung zwischen den Transaktionen.

Definition 5-11 (Transaktions-Konflikt Reihenfolge): *Eine Transaktion t_2 ist in einer Konflikt-Reihenfolgebeziehung mit einer Transaktion t_1, wenn es mindestens zwei in Konflikt stehende Operationen o_1 (von t_1) und o_2 (von t_2) gibt und o_1 vor o_2 ausgeführt wurde. Diese Beziehung zweier Transaktionen t_1 und t_2 wird im weiteren mit der Notation $t_1 \rightarrow t_2$ bezeichnet werden.* ■

Die Konflikt-Reihenfolgebeziehung zweier Transaktionen ist somit eine nichttransitive und asymmetrische Relation, die auf der Reihenfolge der Ausführung von in Konflikt stehenden Operationen basiert.

Betrachtet man nun eine Historie, so läßt sich nach obiger Definition aus den Konflikten der Operationen in der Historie eine Reihenfolgebeziehung der zugehörigen Transaktionen konstruieren.

Definition 5-12 (Serialisierungsgraph): *Ein Serialisierungsgraph SG(H) einer Historie H ist ein gerichteter Graph, dessen Knoten die Transaktionen der in H enthaltenen Operationen sind. Zwei Knoten in SG(H) sind mit einer gerichteten Kante verbunden, wenn die zugehörigen Transaktionen in einer Konflikt-Reihenfolge-Beziehung stehen.* ■

Intuitiv ist einsichtig, daß eine Historie dann korrekt sein muß, wenn der Serialisierungsgraph, der aus ihrer Commit-Projektion entsteht keine Zyklen enthält und somit keine Transaktion mit sich selbst in einem (indirektem) Konflikt steht. Diese Aussage kann noch dahin gehend erweitert werden, daß eine Historie nur dann serialisierbar sein kann, wenn der Serialisierungsgraph azyklisch ist.

Auf den Beweis der obigen Aussage soll an dieser Stelle verzichtet werden, weshalb für eine detailliertere Darstellung auf [BHG87] verwiesen wird.

Der Vorteil des Kriteriums nach Definition 5-12 gegenüber dem klassischen Kriterium in Definition 5-10 liegt darin, daß ein Serialisierungsgraph auch für nicht vollständige Historien generiert werden kann. Darüber hinaus wird während der Fortführung der Historie eine einmal vorhandene Kante in dem zugeordneten Serialisierungsgraphen nicht wieder entfernt und somit ist das Kriterium Präfix-abgeschlossen. D.h. sobald eine Operation ausgeführt werden soll, die einen Zyklus in dem Serialisierungsgraph verursachen würde, müssen geeignete Maßnahmen

ergriffen werden, um diesen Zyklus zu vermeiden.

5.2.3 Recoverability und Spezialisierungen

Die in Abschnitt 5.2.2 eingeführten Kriterien beschränken sich (weitgehend) auf die Sicherstellung der Isolationseigenschaft von Transaktionen. Durch die Atomaritätseigenschaft der ACID-Transaktionen werden aber noch weitere Anforderungen notwendig. Da die Atomarität sich im Falle des benutzerinitiierten Abbruchs einer Transaktion genauso auswirkt wie im Falle eines Systemausfalls (und anschließenden Neustarts), werden die notwendigen Mechanismen und Kriterien meist unter dem Stichwort *Wiederherstellbarkeit* diskutiert.

Die notwendigen Erweiterungen des Serilisierbarkeitskriteriums resultieren aus der Anforderung, daß für jede erfolgreich ausgeführte Schreiboperation einer ACID-Transaktion T_1 zu garantieren ist, daß die entsprechende inverse Schreiboperation ebenfalls ausgeführt werden kann (solange die Transaktion nicht abgeschlossen wurde). Wird jedoch durch eine Operation einer anderen Transaktion T_2, die der Schreiboperation der ursprünglichen Transaktion nachfolgt eine Abhängigkeit impliziert, führt die Ausführung der inversen Schreiboperation notwendigerweise zu einem Zyklus im Serialisierungsgraphen und wäre damit nicht zulässig.

Definition 5-13 (Lesebeziehung): *Eine Transaktion T_i liest von einer Transaktion T_j, wenn eine Operation o_j von T_j mit einer Operation o_i von T_i in einem Schreib/Lese-Konflikt steht.* ■

Die Probleme, die durch diesen formalen Konflikt auftreten, lassen sich weiter differenzieren. Liest die Transaktion T_2 nur Daten von T_1, kann nur dann ein Problem auftreten, wenn Transaktion T_1 abbricht und T_2 erfolgreich beendet wird, da dann die von T_2 gelesenen Werte und somit die erzeugten Resultate eventuell nicht korrekt sind.

Definition 5-14 (Recoverability): *Eine Historie H heißt recoverable wenn gilt, daß im Falle, daß eine Transaktion T_i von einer Transaktion T_j liest und die commit-Operation c_i von T_i in H enthalten ist, die commit-Operation von T_j ebenfalls in H enthalten ist und $c_j < c_i$.*
Die Menge aller Historien, die recoverable sind, wird mit RC bezeichnet. ■

Durch das Recoverability-Kriterium wird somit gefordert, daß eine Transaktion erst dann eine commit-Operation durchführen darf, wenn alle Transaktionen von denen sie gelesen hat, bereits ihre commit-Operation durchgeführt haben.

Kehrt man dieses Kriterium um, bedeutet dies, daß der Abbruch einer Transaktion T_1 automatisch den Abbruch aller Transaktionen fordert, welche von T_1 gelesen haben. Diesen Effekt bezeichnet man mit kaskadierendem Zurücksetzen (engl. *cascading aborts*).

Rücksetzkaskaden, die durch eine Verletzung des Kriteriums in Definition 5-14 verursacht werden, haben zunächst nur den Nachteil, daß die Aktionen einer Transaktion (in diesem Falle die Operation abort) den Abbruch mehrerer anderer Transaktionen nach sich ziehen kann. Allerdings verletzt dies auch die Isolationseigenschaft im weiteren Sinne.

Definition 5-15 (Vermeidung des kaskadierenden Zurücksetzens): *Eine Historie vermeidet kaskadierendes Zurücksetzen wenn gilt, daß wenn eine Transaktion T_i von T_j liest, die commit-Operation von T_j in H enthalten sein muß und vor dem Lesen ausgeführt wurde.*

Die Menge der Historien, die dieses Kriterium erfüllen, wird mit ACA bezeichnet. ■

Wenn also für eine Historie gilt, daß sie kaskadierendes Zurücksetzen vermeidet, können Transaktionen nur von bereits abgeschlossenen Transaktionen lesen (im Sinne von Definition 5-13).

Es erscheint zunächst verwunderlich, daß die Definitionen der Mengen RC und ACA nur auf der Basis von Leseoperationen definiert wurden. Wie allerdings bereits erwähnt wurde, ist dies auf die Entstehungsgeschichte der theoretischen Arbeiten auf dem Gebiet der Transaktionsverarbeitung zurückzuführen. Bei diesen frühen Arbeiten wurde davon ausgegangen, daß vor jeder Schreiboperation auf jeden Fall eine Leseoperation auf das selbe Datenelement ausgeführt wurde.

Geht man nicht von dieser Annahme aus, reichen die eingeführten Kriterien noch nicht aus, um die Probleme bei der Garantie der Atomaritätseigenschaft zu vermeiden. Beispielsweise ist es möglich, daß eine Transaktion T_1 ein Datenelement schreibt und danach eine andere Transaktion T_2 ebenfalls eine Schreiboperation auf dieses Element durchführt. Folgende Fälle können hierbei unterschieden werden

1. Werden beide Transaktionen entweder erfolgreich beendet oder in der umgekehrten Reihenfolge ihrer Ausführung zurückgesetzt entsteht kein Problem.
2. Führt T_1 eine commit-Operation aus, der eine abort-Operation von T_2 folgt, entsteht kein Problem.
3. Führt T_2 eine commit-Operation aus, der eine abort-Operation von T_1 folgt, wird durch T_1 der Zustand vor der Ausführung von T_1 wieder hergestellt. Somit gehen die Änderungen von T_2 verloren (trotz commit!).
4. Bricht zunächst T_1 ab und danach T_2, dann stellt der Abbruch von T_2 den Zustand nach der Schreiboperation von T_1 her.

Wiederum ist einfach zu erkennen, daß ein Abbruch von T_1 den Abbruch von T_2 verlangt (als Voraussetzung für die Durchführung des Zurücksetzens). Somit erhält man wieder den Effekt des kaskadierenden Zurücksetzens. In diesem Falle allerdings in einer verschärften Form, da im Fall 3 Transaktion T_2 bereits abgeschlossen und somit eigentlich aus dem System entfernt wurde.

Um auch dieses Problem zu vermeiden, wird ein weiteres Kriterium eingeführt:

Definition 5-16 (Striktheit): *Eine Historie heißt strikt, wenn keine Operation einer Transaktion auf ein Datenelement zugreift, welches von einer Operation einer anderen, nicht abgeschlossenen Transaktion verändert wurde.*

Die Menge aller strikten Historien wird mit ST bezeichnet. ■

Bei strikten Historien ist es somit erforderlich, daß Transaktionen, die schreibend auf ein Datenelement zugegriffen haben, zunächst mittels Commit oder Abort abgeschlossen werden müssen, bevor andere Transaktionen auf die geänderten Daten zugreifen können.

Der Hauptunterschied der Kriterien zur Sicherstellung der Isolation und der Kriterien zur Sicherstellung der Wiederherstellbarkeit liegt in der Menge der betrachteten Operationen und der Art des ersten Zugriffs. Während das Serialisierbarkeitskriterium alle Arten von Konflikten auf der Ebene der Operationen betrachtet, setzen die Kriterien für ST, ACA und RC eine Schreib-Operation als ersten Zugriff voraus. Andererseits erlaubt das Serialisierungskriterium Konflikte, sofern sie nur durch zwei Operationen verursacht werden, während dies im Falle der Striktheit bereits ausgeschlossen wird, wenn nicht abgeschlossene Transaktionen betrachtet werden.

Es läßt sich zeigen, daß es eine echte Teilmengenbeziehung zwischen ST, ACA und RC gibt ($ST \subset ACA \subset RC$) [BHG87]. Auf Grund der unterschiedlichen Voraussetzungen läßt sich jedoch keine solche Teilmengenbeziehung bezüglich der serialisierbaren Historien finden.

Abbildung 5-1 zeigt eine grafische Darstellung der qualitativen Beziehungen der Mengen der Historien, welche die eingeführten Kriterien erfüllen. Im weiteren sollen Historien, die sowohl eines der Kriterien ST, ACA oder RC als auch das Serialisierbarkeitskriterium erfüllen mit ST-SR, ACA-SR, bzw. RC-SR bezeichnet werden.

5.2.4 Kombinierte Ansätze

Da keine der Mengen ST, ACA und RC in SR enthalten sind und auch SR nicht in ST, ACA oder RC enthalten ist, müssen zur Sicherstellung der Serialisierbarkeit und der Wiederherstellbarkeit immer zwei Kriterien überprüft werden. Betrachtet man entsprechende Umsetzungen in Laufzeitsysteme, stellt dies ebenfalls ein Pro-

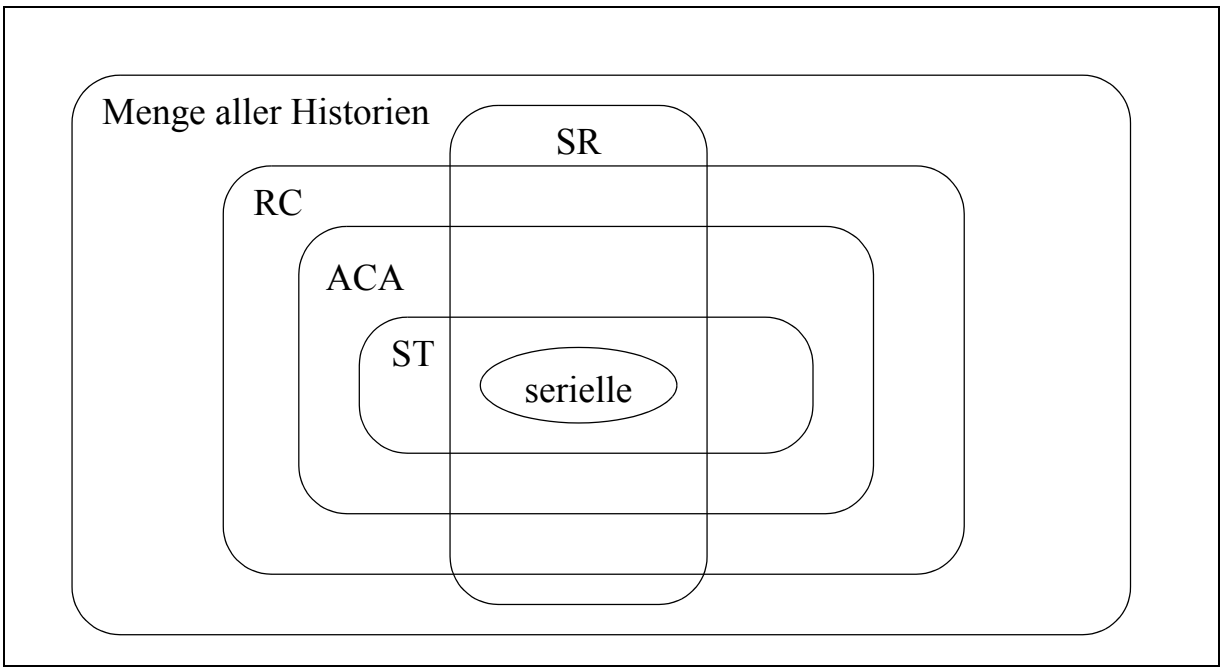

Abbildung 5-1: Teilmengenbeziehung der Kriterien

blem dar, da entweder für beide Kriterien getrennte Verfahren implementiert werden müssen oder weitere Einschränkung der möglichen Historien vorgenommen werden muß, bis eine Teilmenge gefunden wird, die mittels einem Kriterium geprüft werden kann.

Aus diesem Grund beschäftigten sich zwei Gruppen von Wissenschaftlern mit der Entwicklung eines kombinierten Kriteriums, welches sowohl die Atomarität als auch die Isolation berücksichtigt [AAE93][SWY93]. Erstaunlicherweise kamen beide Gruppen nahezu gleichzeitig zu einem fast identischen Resultat. Deshalb führten die zwei Gruppen ihre Ergebnisse in einem gemeinsamen Papier zusammen [AVA94a].

Die grundlegende Idee des Ansatzes ist es, im Gegensatz zu den Serialisierbarkeitsansätzen nicht die Commit-Projektion einer Historie zu betrachten. Statt dessen wird eine vorliegende Historie so weit ergänzt, daß nur abgeschlossene Transaktionen enthalten sind. Die Ergänzung einer Historie wird dabei für alle aktiven Transaktionen derart vorgenommen, daß zunächst alle aktiven Transaktionen als abgebrochen angesehen werden. Zu diesem Zweck wird die Menge der Operationen um eine sogenannte Gruppen-Abbruch-Operation erweitert.

Definition 5-17 (Group Abort): *Eine Gruppen-Abbruch-Operation $a(T_1, T_2, \ldots, T_k)$ zeigt an, daß für jede Transaktion T_j aus T_1 bis T_k die Gegenoperationen zu den Operationen auszuführen sind, die eine Transaktion T_j bisher ausgeführt hat.* ■

Somit wird eine Historie zunächst um die Operationen erweitert, die notwendig sind, um alle aktiven Transaktionen zurückzusetzen. Allerdings ist damit noch

nicht festgelegt, in welcher Reihenfolge diese Operationen auszuführen sind.

Um explizite Aussagen über diese Reihenfolge machen zu können, ist eine weitere Annahme notwendig. Anstatt eine abort-Operation wie bisher als eine atomare Einheit anzusehen, wird sie durch die notwendigen inversen (Schreib-)Operationen und eine abschließende Commit-Operation ersetzt. Da Leseoperationen keine Operationen bei einem Transaktionsabbruch bedingen, werden diese bei der Expandierung der abort-Operationen nicht betrachtet. Berücksichtigt man dann die Reihenfolge der Operationen in Abhängigkeit von den Konflikten, die während der "Vorwärtsverarbeitung" aufgetreten sind, kommt man zu folgender erweiterter Historie:

Definition 5-18 (Erweiterte Historie): *Sei $H=(\Sigma,<)$ eine Historie. Ihre erweiterte Historie H^e ist ein Tupel $(\Sigma^e, <^e)$, das durch folgende Regeln aus H entsteht:*

Σ^e entsteht aus Σ durch:

1. *$(o \in \Sigma \wedge o \neq a) \Rightarrow o \in \Sigma^e$*
2. *Eine Gruppen-Abbruch-Operation am Ende von H enthält alle aktiven Transaktionen.*
3. *Für alle Schreiboperationen w abgebrochener Transaktionen in Σ müssen die inversen Schreiboperationen w^{-1} in Σ^e enthalten sein:*
   $w_t \in \Sigma \wedge a_t \in \Sigma \Rightarrow w_t^{-1} \in \Sigma^e$
4. *Alle abort-Operationen in Σ werden durch commit-Operation in Σ^e ersetzt.*

$<^e$ entsteht durch folgende Regeln:

1. *Für alle Paare von Operationen o_i und o_j: $o_i < o_j \Rightarrow o_i <^e o_j$*
2. *Sei W die Menge aller Operationen, die in Transaktionen enthalten sind, deren Abort-Operation in einer Gruppen-Abbruch-Operation in Σ ist.*
   Gilt für je zwei Operationen w_i und w_k aus W $w_i < w_k$, dann muß $w_k^{-1} <^e w_i^{-1}$ für die inversen Operationen gelten.
3. *Alle inversen Operationen der Transaktionen, die in H keine commit-Operation ausgeführt haben, folgen bezüglich $<^e$ den Originaloperationen und sind bezüglich $<^e$ vor der commit-Operation der Transaktion für die sie die inverse Operation darstellen.*
4. *Für alle Operationen o vor einer Gruppen-Abbruch-Operation bezüglich $<$, die in Konflikt mit einer inversen Operation w^{-1} der Gruppen-Abbruch-Operation stehen, gilt $o <^e w^{-1}$. Umgekehrt gilt $w^{-1} <^e o$ für alle Operationen o, die bezüglich $<$ einer Gruppen-Abbruch-Operation folgen und mit einer Operation w^{-1} der Gruppen-Abbruch-Operation in Konflikt stehen.*

5. *Gilt für zwei Gruppen-Abbruch-Operationen eine Reihenfolge in H, dann muß die Reihenfolge bezüglich $<^e$ für alle in Konflikt stehenden inversen Operationen, die aus diesen Gruppen-Abbruch-Operationen entstehen, der Reihenfolge der Gruppen-Abbruch-Operationen bezüglich $<$ entsprechen.* ■

Auf der Basis dieser erweiterten Historie läßt sich nun das kombinierte Kriterium zur Überprüfung einer Historie im Hinblick auf Atomarität und Isolation formulieren:

Definition 5-19 (RED): *Eine Historie H ist reduzierbar, wenn ihre erweiterte Historie H^e durch Anwendung der folgenden Regeln in eine serielle Historie transformiert werden kann:*

1. *Kommutativitätsregel:*
   Wenn zwei Operation o_i und o_j nicht in einem Konflikt stehen und kein o_m existiert mit $o_i <^e o_m <^e o_j$, kann die Ordnung $o_i <^e o_j$ ersetzt werden durch $o_j < o_i$.
2. *Undo Regel:*
   Sind eine Operation o und ihre inverse Operation o^{-1} in H^e und gilt $o <^e o^{-1}$ ohne daß eine Operation o_m existiert mit $o <^e o_m <^e o^{-1}$, dann können o und o^{-1} aus der Historie entfernt werden.
3. *Regel der leeren Aktion:*
   Leseoperationen von abgebrochenen oder aktiven Transaktionen in H dürfen aus H^e entfernt werden.

Die Menge aller reduzierbaren Historien wird mit RED bezeichnet. ■

Wie aus dem RED-Kriterium abgelesen werden kann, basiert das eigentliche Kriterium wiederum auf der Überführbarkeit einer Historie auf eine serielle Historie. Dabei wird nicht die Commit-Projektion beurteilt, sondern die erweiterte Historie.

Ein Nachteil der Definition ist, daß sie nicht Präfix-abgeschlossen ist. Eine einfache Erweiterung des Kriteriums berücksichtigt diesen Nachteil.

Definition 5-20 (PRED): *Eine Historie heißt Präfix-reduzierbar, wenn jeder Präfix der Historie reduzierbar ist.*

Die Menge alle Präfix-reduzierbaren Historien wird mit PRED bezeichnet. ■

Abgesehen von der Tatsache, daß diese simple Erweiterung der Definition 5-19 keine Verbesserung im Hinblick auf eine entsprechende Implementierung liefert, ermöglicht sie doch den Vergleich mit den bisher eingeführten Kriterien. So läßt sich beispielsweise zeigen, daß SR-ST $\subset$ PRED $\subset$ SR-RC [SWY93].

Analog zum Fall der Serialisierbarkeit wurde auch im Falle der Präfix-Reduzierbarkeit nach einem Kriterium gesucht, welches die Umsetzung in ein Laufzeitsystem unterstützt. Im Falle von PRED bedeutet dies, ein Kriterium zu finden, wel-

ches während der Ausführung von Transaktionen nicht verletzt wird. Ausgangspunkt für dieses Kriterium war die Untersuchung bestehender Implementierungen von Transaktionssystemen, die die ACID-Eigenschaften garantieren. Grundlage dieser Implementierungen ist ein sogenanntes *Log* [GrRe93], welches Informationen über die ausgeführten Operationen und die entsprechenden inversen Operationen aufbewahrt. Es ist ein rein sequentiell beschreibbarer, stabiler Speicher, bei dem nur am Ende Daten eingefügt werden können.

Untersucht man nun existierende Implementierungen der Wiederherstellbarkeit auf der Basis eines Logs, so stellt man Bedingungen fest, die von einem Ablaufsystem für ACID-Transaktionen eingehalten werden müssen, um die Atomarität gewährleisten zu können. Genau diese Bedingungen werden in einem weiteren Kriterium zusammengefaßt, um eine entsprechende Anwendbarkeit für eine Implementierung des PRED-Kriteriums zu ermöglichen.

Definition 5-21 (SOT): *Eine Historie H heißt serialisierbar mit geordneter Terminierung, wenn sie wiederherstellbar (RC) und serialisierbar (SR) ist, und wenn für jedes Paar von in Konflikt stehenden Operationen w_i und w_j mit $w_i < w_j$ gilt:*

1. *T_j führt eine commit-Operation nur nach der commit-Operation von T_i aus.*
2. *T_i führt eine abort-Operation nur nach der abort-Operation von T_j aus, oder es existiert eine Gruppen-Abbruch-Operation in H die sowohl T_i als auch T_j enthält.*

Die Menge aller Historien die dieser Bedingung genügen wird mit SOT bezeichnet. ■

Zu betonen ist, daß das Kriterium auf der Commit-Projektion der Historie bezüglich der Serialisierbarkeit basiert. Die Zusatzbedingungen beziehen sich aber auf Operationen nicht terminierter Transaktionen, so daß beliebige Historien beurteilt werden können. Da das SOT Kriterium darüber hinaus nur relativ leichte Einschränkungen der bereits bekannten Kriterien vornimmt, zu denen auch Implementierungen existieren, eignet es sich sehr gut für den praktischen Einsatz.

Ein besonders wichtiger Aspekt ist außerdem, daß bewiesen werden kann, daß die Menge SOT äquivalent zur Menge PRED ist [AVA94b] und die maximale Menge von Historien darstellt, die sowohl im Hinblick auf die Isolationseigenschaft als auch im Hinblick auf die Wiederherstellbarkeit korrekt sind (SR-RC).

Allerdings haben die SOT-Historien noch die ungünstige Eigenschaft, daß kaskadierende Abbrüche auftreten können bzw. zulässig sind. Durch einen erweiterten Serialisierungsgraphen [AVA94b] kann diese ungünstige Eigenschaft darauf beschränkt werden, daß eine abort-Operation einer Transaktion sich nur auf bereits abgebrochene Transaktionen fortpflanzt, was keinen realen Einfluß mehr hat. An

dieser Stelle soll aber auf diese Spezialisierung nicht näher eingegangen werden.

5.2.5 Kriterien für geschlossen geschachtelte Transaktionen

Wie bereits in Abschnitt 3.1.1 eingeführt, verhalten sich geschlossen geschachtelte Transaktionen nach außen hin wie die klassischen ACID-Transaktionen. Das heißt, daß aus dieser Perspektive die bereits vorgestellten Kriterien analog zur Anwendung kommen. Betrachtet man allerdings die interne Struktur und die vorgegebenen Eigenschaften, ergeben sich weitere Aspekte.

Es wird ein kurzer Rückblick auf die Semantik der geschlossen geschachtelten Transaktionen vorgenommen, um das Verständnis zu erleichtern:

1. Eine Transaktion ohne Elterntransaktion heißt Top-Level-Transaktion und hat die ACID-Eigenschaften bezüglich aller Transaktionen, die nicht Nachkommen von ihr sind.
2. Eine Kind- (oder Sub-)Transaktion hat die ACI-Eigenschaften bezüglich aller Geschwister (Kinder der selben Elterntransaktion).
3. Eine Sub-Transaktion hat Zugriff auf alle Datenelemente, auf welche die Elterntransaktion Zugriff hat und vererbt an ihrem Ende alle Datenelemente an die Elterntransaktion.

Solange innerhalb geschlossen geschachtelter Transaktionen keine Parallelität zugelassen wird, entstehen prinzipiell keine neuen Probleme im Vergleich zu den nicht geschachtelten (flachen) Transaktionen. Erlaubt man jedoch die parallele Ausführung von Geschwistertransaktionen (ohne gleichzeitige Weiterführung der Elterntransaktion) resultiert dies in einer Konkurrenz der Geschwistertransaktionen um Datenelemente, da für sie die Isolationseigenschaft gegeben ist.

Prinzipiell ist es nun möglich, die bekannten Korrektheitskriterien einzusetzen. Allerdings resultiert dies je nach Implementierung (siehe Kapitel 6) in Konflikten zwischen Subtransaktionen einer Top-Level-Transaktion (Intra-Transaktionskonflikt) und kann bis zum Abbruch der Top-Level-Transaktion führen, wenn nicht jede beteiligte Komponente des Laufzeitsystems auf die Verarbeitung geschachtelter Transaktionen ausgelegt ist.

Darüber hinaus kann es bei der parallelen Verarbeitung von Geschwistertransaktionen zu Verklemmungen bezüglich ererbter Datenelemente kommen. Im Gegensatz zum Verklemmungsfall bei flachen Transaktionen (Inter-Transaktionskonflikt), kann eine solche Verklemmung nicht mit Hilfe von einfachen Abhängigkeitsgraphen erkannt werden, da die Sperren der Datenobjekte von derselben Transaktion angefordert wurden. Bei der Verwendung von zeitschranken-

basierten Mechanismen stellt zwar die Erkennung von Verklemmungen kein Problem dar, doch kann auch hier nicht erkannt werden, welche Subtransaktion abgebrochen werden muß, um die Verklemmung zu beheben. Es wird also immer die Top-Level-Transaktion abgebrochen, was einen erheblichen Vorteil von geschachtelten Transaktionen, die feinere Rücksetzgranularität, völlig außer Acht läßt.

Vor diesem Hintergrund betrachtet, ergeben sich bezüglich der Korrektheitskriterien keine neuen Aspekte. Allerdings ist es nicht möglich die bekannten Ansätze zur Sicherstellung der Korrektheit direkt auf den Fall der geschachtelten Transaktionen zu übertragen.

5.3 Korrektheit bei Mehrschichttransaktionen

Da die Mehrschichttransaktionen aus der Klasse der offen geschachtelten Transaktionen stammen, liegt die Vermutung nahe, daß sich die Korrektheitskriterien von ACID-Transaktionen und dieser Art der Transaktionen grundlegend unterscheiden. Wie sich im Verlauf dieses Abschnitts herausstellen wird, ist dies nicht der Fall.

5.3.1 Historien von Mehrschichttransaktionen

Die in Abschnitt 4.2 eingeführte abstrakte Maschine ist auf eine Ebene (oder Schicht) der Mehrschichttransaktionen beschränkt. Historien, die sich nur auf eine solche Maschine beziehen, können somit analog zu den bisher betrachteten Historien definiert werden. Betrachtet man jedoch eine Mehrschichttransaktion als Ganzes, so muß zur Beurteilung ihrer Korrektheit auf die Historien aller Ebenen zurückgegriffen werden.

Definition 5-22 (Mehrschicht-Historie): *Eine Historie H einer Mehrschichttransaktion ist die Menge aller Historien H_i der Schichten der Mehrschichttransaktion.*

$$H = \{(\Sigma_i, <_i)\} \quad \blacksquare$$

Eine strikte Hierarchie der Operationen der Mehrschichttransaktionen hilft bei dem Vergleich des Korrektheitskriteriums mit dem Korrektheitsbegriff des read/write-Modells. Auf unterster Ebene werden alle Operationen durch eine Menge von Lese- oder Schreiboperationen implementiert. Beurteilt man also die Historie der untersten Ebene muß dies auf der Basis der selben Kriterien wie beim Lese-/Schreib-Modell geschehen.

5.3.2 Konfliktbegriff der Mehrschichttransaktionen

Die Einführung mehrerer Abstraktionsebenen oberhalb des Lese-/Schreibmodells erfordert eine erweiterte Sicht der bereits eingeführten Begriffe der Kommutativität bzw. des Konflikts von Operationen. Da grundsätzlich davon ausgegangen wird, daß unterschiedliche Schichten bezüglich ihres Konfliktbegriffs unabhängig sind, wird wiederum eine schichtabhängige Definition von Konflikten eingeführt.

Definition 5-23 (Konflikt): *Für jede Schicht $i \in \{0,1,2,...\}$ existiert eine binäre Konfliktrelation CON_i, welche über der Menge der Operationen O_i einer Schicht definiert ist.*

$$CON_i(a,b) \Rightarrow (a < b \wedge \neg(b < a)) \vee (b < a \wedge \neg(a < b)) \quad \blacksquare$$

Die Konfliktrelation einer Schicht gibt somit an, ob zwei Operationen parallel ausgeführt werden dürfen, oder ob sie in irgendeiner Ordnung nacheinander auszuführen sind. Ein Problem wird bereits an dieser Definition deutlich: Für jedes Paar von Operationen einer Stufe muß die Konfliktrelation definiert sein. Erweitert man beispielsweise eine Stufe um eine neue Operation muß die Konfliktrelation ebenfalls erweitert werden.

5.3.3 Mehrschicht-Serialisierbarkeit

Die Einführung von schichtspezifischen Konfliktrelationen hat zunächst keine Auswirkungen auf andere Schichten. Dies ändert sich allerdings, wenn man die Implikationen eines Konflikts auf einer Ebene i auf die nächst höhere Ebene i+1 betrachtet. Analog zu den Auswirkungen der Konflikte im read/write-Modell, wo durch einen Konflikt und die Ordnung von Operationen eine Ordnung der Transaktionen impliziert wird, kann man die Auswirkungen des Konflikts von Operationen im Mehrschicht-Transaktionsmodell definieren.

Definition 5-24 (Ordnungsrelation): *Für jede Schicht i des Mehrschicht-Transaktionsmodells existiert eine binäre Ordnungsrelation $<_i^{\sim}$ bezüglich einer Ausführungshistorie. Für Stufe 0 gilt, daß die Ordnungsrelation $<_0^{\sim}$ gleich der Ordnungsrelation der Ausführungshistorie aller in Konflikt bezüglich CON_0 stehender Operationen ist. Für jede andere Stufe $i > 0$ gilt, daß zwei Operationen der Stufe i dann in Relation $<_i^{\sim}$ stehen, wenn sie jeweils eine Operation auf der nächst niedrigeren Stufe i-1 beinhalten, die bezüglich CON_{i-1} in Konflikt sind und in Relation $<^{\sim}_{i-1}$ stehen.* $\blacksquare$

Vergleicht man die Ordnungsrelation $<_1^{\sim}$ mit der Konfliktrelation von Transaktionen bei der Konstruktion eines Serialisierbarkeitsgraphen (siehe Abschnitt 5.2.2), erkennt man, daß es sich um identische Relationen handelt. Dies läßt sich verallgemeinern, so daß für jede Stufe ein Serialisierbarkeitsgraph konstruiert werden kann.

Definition 5-25 (Mehrschicht-Serialisierbarkeit): *Eine Historie H einer Mehrschichttransaktion heißt mehrschicht-serialisierbar, wenn für jede Schicht i gilt, daß alle in Konflikt stehenden Operationen der Schicht in einer Ordnung bezüglich der Historie H_i stehen und der Graph, der aus der Ordnungsrelation $<_i^{\sim}$ konstruiert werden kann, azyklisch ist.*

Die Menge aller mehrschicht-serialisierbaren Historien wird mit ML-SR bezeichnet. ■

Somit wird das Kriterium zur Sicherstellung der Isolationseigenschaft auf die Überprüfung der Azyklizität der Serialisierungsgraphen abgebildet. Folglich kann das klassische read/write-Modell als ein Spezialfall der Mehrschichttransaktionen angesehen werden, bei dem es nur zwei Schichten gibt.

5.3.4 Recovery bei Mehrschichttransaktionen

Da Mehrschichttransaktionen Vertreter der Klasse der offen geschachtelten Transaktionen sind, werden üblicherweise die Resultate am Ende einer Teil-Transaktion auch für Transaktionen sichtbar, die keine Vorfahren mit der beendeten Transaktion gemein haben. Deshalb ist der Ansatz, einen Zustand dadurch wieder herzustellen, daß der Wert eines geänderten Objektes, der vor der Ausführung der Transaktion vorgefunden wurde, einfach wieder etabliert wird, nicht anwendbar.

Grundsätzlich erfordert die Verwendung von Mehrschichttransaktionen die Definition von Gegen- oder Kompensationstransaktionen, die die inversen Operationen zu den durch die Transaktion auszuführenden Operationen ausführen. Die Idee bei der Implementierung eines Zurücksetzens ist dann, die Gegentransaktionen in umgekehrter Reihenfolge zu den ursprünglichen Teil-Transaktionen auszuführen. Dies entspricht der Ausführung der inversen Schreiboperationen bei dem Lese-/Schreibmodell. Allerdings nehmen die inversen Schreiboperationen keine Rücksicht auf eventuelle Änderungen durch andere Transaktionen, was im Falle der Mehrschicht-Transaktionen erfolgen muß.

Obwohl für das Modell der Mehrschichttransaktionen die Idee eines Kriteriums zur Sicherstellung der Wiederherstellbarkeit formuliert [WeSc92] und auch konkrete Ansätze zur Implementierung veröffentlicht wurden [Lom92], ist kein formales Kriterium im direkten Zusammenhang mit den Mehrschichttransaktionen definiert worden. Die Idee stimmt jedoch mit dem Ansatz des PRED-Kriteriums überein, wodurch sich dieses Kriterium relativ leicht auf die Mehrschichttransaktionen übertragen läßt. Somit kann das PRED-Kriterium als formales Korrektheitskriterium für Mehrschichttransaktionen herangezogen werden.

5.4 Korrektheit nach Korth et. al.

Abläufe nach Korth et. al. zeichnen sich dadurch aus, daß bei ihnen explizite Indikatoren verwendet werden, welche anzeigen, ob eine Operation (eine Subtransaktion) ausgeführt werden kann bzw. ob die Ausführung einer Operation erfolgreich war. Die Indikatoren basieren darauf, daß es für jede Operation explizit definierte Konsistenzbedingungen gibt, die anwendungsabhängig definieren, welcher Zustand (der Datenobjekte) für die Ausführung einer Operation notwendig ist bzw. welcher Zustand nach der Ausführung einer Operation als konsistent akzeptiert wird.

5.4.1 Historien nach Korth et. al.

Eine Historie eines Ablaufs nach Korth et. al. unterscheidet sich nun von den bisher betrachteten Historien dadurch, daß die Überprüfung der Konsistenz explizit durch die in Abschnitt 4.3.1 eingeführten Operationen γ und ε in einer Historie enthalten sind. Darüber hinaus ist durch die strukturellen Einschränkungen der Abläufe sichergestellt, daß einer Operation einer Transaktion immer eine γ-Operation vorausgeht und eine ε-Operation nachfolgt.

Um den Abschluß eines Ablaufs (oder Transaktion) in einer Historie erkennen zu können, wird formal die letzte Operation einer Transaktion vor ihrem commit als x_f bezeichnet. Der Operation x_f folgen dann noch die Konsistenzprüfungs-Operation ε sowie die commit-Operation der Transaktion.

Diese Unterschiede gegenüber der klassischen Definition von Historien sind nur geringfügig, so daß an dieser Stelle keine ausführliche Erläuterung notwendig ist.

5.4.2 Prädikatabhängige Konflikte

Der Konfliktbegriff, der dem Modell von Korth. et. al. zu Grunde liegt, scheint intuitiv durch die explizite Einführung von Konsistenzkriterien klar auf der Hand zu liegen. Man kann vermuten, daß eine ausführende Operation a dann in einem Konflikt mit einer anderen ausführenden Operation b steht, wenn entweder a ein Objekt ändert, welches in dem zugeordneten Eingangsprädikat von b auftaucht, oder umgekehrt b ein Objekt ändert welches in dem Eingangsprädikat von a referenziert wird.

Prinzipiell trifft diese Vermutung auch zu. Allerdings sind im Falle des Modells von Korth et. al. die Prädikate von Ablauf zu Ablauf verschieden, so daß es durchaus möglich ist, daß eine Operation a mit einer Operation b bei einem Ablauf in Konflikt steht und bei einem anderen Ablauf kein Konflikt von a und b vorliegt. Deshalb wird ein Konflikt im Modell von Korth et. al. anders als in den bisher eingeführten Modellen aufgefaßt. Der Grundgedanke bei dem Modell ist, daß es in

einem System eine Menge von Prädikaten gibt, wobei sowohl von Operationen definierte Prädikate berücksichtigt werden als auch die Bedingungen, die auf den Datenelementen definiert sein können (beispielsweise Wertebereichsbedingungen).

Um die Prädikate leichter handhabbar zu machen, wird davon ausgegangen, daß diese in konjunktiver Normalform vorliegen.

Definition 5-26 (Prädikat): *Ein Prädikat P nach Korth et. al. ist die konjunktive Verknüpfung von Prädikaten p_i, welche disjunktive Verknüpfungen von Atomen sind.*
Ein Prädikat p_i wird im weiteren als Term einer Konjunktion bezeichnet.
Ein Datenelement d heißt Element eines Terms einer Konjunktion, wenn es in einem Atom des Terms verwendet wird. ■

Definition 5-27 (Konflikt): *Eine Operation a ist in Konflikt mit einer Operation b bezüglich eines Terms einer Konjunktion, wenn a und b auf das gleiche Element eines Terms zugreifen und mindestens eine Operation dieses Element ändert.* ■

Definition 5-27 beschreibt somit eine Konfliktrelation, welche auf einem Term einer Konjunktion basiert. Grundsätzlich läßt sich damit entscheiden, ob zwei Operationen in einem Konflikt bezüglich der Bedingungen auf einem Datenelement stehen. Unter der Annahme, daß für jedes Datenelement ein solches Prädikat existiert und zunächst keine ablaufbezogenen Prädikate definiert sind, erhält man eine Konfliktrelation, welche äquivalent ist zu der Konfliktrelation der ACID-Transaktionen aus Abschnitt 5.2.1.

5.4.3 Prädikatbezogene-Serialisierbarkeit

Die Basis des Korrektheitsbegriffes nach Korth et. al. ist zunächst eine einfache Folgerung aus den eingeführten Konsistenzprädikaten.

Definition 5-28 (Korrektheit): *Eine vollständige Ausführung einer Transaktion ist dann korrekt, wenn alle Eingangsprädikate der Operationen erfüllt sind und das Ausgangsprädikat der Operation erfüllt ist.* ■

Es verwundert zunächst, daß nur das Ausgangsprädikat der letzten Operation erfüllt sein muß. Geht man allerdings davon aus, daß die Ausgangsprädikate der vorhergehenden Operationen nur dazu dienen, einen Zustand sicherzustellen, der das Eingangsprädikat der folgenden Operationen erfüllt, wird klar, daß das Kriterium hinreichend ist.

Wie in einigen der bisher vorgestellten Kriterien ergibt sich auch bei dem Kriterium in Definition 5-28 das Problem, daß nur für Historien mit abgeschlossenen Transaktionen entschieden werden kann, ob sie korrekt sind oder nicht. Da dieses für die praktischen Anwendung ungeeignet ist, wurden mehrere abgewandelte Kri-

terien entwickelt, die eine Umsetzung in einen Scheduler erlauben. In dieser Arbeit wird allerdings nur auf eines dieser Kriterien eingegangen werden: die prädikatbezogene Serialisierbarkeit.

Die Grundidee für die Entwicklung der prädikatbezogenen Serialisierbarkeit ist es, statt der abstrakten Aussage über die Korrektheit eines Ablaufs als Ganzes die Korrektheit mittels des eingeführten Konfliktbegriffs festzulegen. Da der Konfliktbegriff es zuläßt, daß zwei Operationen in einem Ablauf in Konflikt stehen können und in einem anderen nicht, ist die einfache Übertragung des Konflikts zweier Operationen auf den Konflikt von ganzen Transaktionen, wie im Falle der ACID-Transaktionen, kein geeignetes Mittel, um zu einem globalen Korrektheitsbegriff zu gelangen. Um trotzdem den Konflikt von ganzen Transaktionen auf den Konflikt von Operationen zurückführen zu können, wird gefordert, daß die Überprüfung eines Eingangsprädikates einer Operation gleichzeitig garantieren soll, daß das Prädikat bis zum Ende der Transaktion erfüllt ist, wenn es zum Zeitpunkt der Überprüfung erfüllt war.

Zur Definition des eigentlichen Kriteriums wird zunächst eine Hilfsdefinition benötigt.

Definition 5-29 (Reduzierte Historie): *Sei d eine Menge von Datenelementen. Eine bezüglich der Menge d reduzierte Historie $H_d=(\Sigma_d,<_d)$ entsteht aus der Historie $H=(\Sigma,<)$, indem alle Operationen aus Σ entfernt werden, die nicht auf ein Datenelement aus d zugreifen. Für je zwei Operationen a und b aus Σ_d soll gelten, daß, wenn $a < b$ gilt, muß auch $a <_d b$ gelten.* ■

Mit Hilfe der Definition 5-29 kann ein Kriterium definiert werden, welches nur Historien betrachtet, die Zugriffe auf bestimmte Mengen von Datenelementen enthalten. Der Äquivalenzbegriff, der in der folgenden Definition zur Anwendung kommt, entspricht dem aus Definition 5-9.

Definition 5-30 (Prädikat-Serialisierbarkeit): *Gegeben sei ein Prädikat P welches eine konjunktive Verknüpfung von Termen p_i ist. Die Menge von Datenelementen, welche in einem Term p_i verwendet werden sei mit d_i bezeichnet. Eine Historie H heißt serialisierbar bezüglich eines Prädikates P, wenn für jede bezüglich der Mengen d_i reduzierte Historie gilt, daß sie äquivalent zu einer seriellen Historie ist.*

Die Menge aller prädikat-serialisierbaren Historien wird mit P-SR bezeichnet.■

Korth et. al. verwenden somit nur die Tatsache, daß ein Datenelement in einem Prädikat verwendet wird und nicht die Erfüllung bzw. Nicht-Erfüllung des Prädikates. Dieser Ansatz ist ähnlich zu den Prädikatsperren, die in [EGL76] eingeführt wurden.

5.4.4 Recovery-Aspekte

Obwohl in [KLS90] detailliert auf die Anforderung an die Semantik von Aktionen zur Kompensation eingegangen wird, gibt es keine Spezialisierung des Korrektheitskriteriums im Hinblick auf diese. Dies läßt nur den Schluß zu, daß die Prädikate der "normalen" Operationen, die Anforderungen an eine eventuelle Kompensation mit abdecken müssen, um so zumindest die semantische Atomarität der Transaktionen gewährleisten zu können. Allerdings wird hierzu keine Aussage gemacht.

Eine wichtige Eigenschaft der Kompensationsaktionen ist jedoch erwähnenswert: Kompensationsaktionen sind atomar und haben somit die ACID-Eigenschaften. Wie jedoch gewährleistet wird, daß diese Aktionen auch ausgeführt werden können, wird nicht diskutiert.

Insgesamt wird das Verhalten der Transaktionen im Fehlerfall recht dürftig modelliert. Dies zeigt sich beispielsweise im Falle eines Systemausfalls. Zunächst machen Korth et. al. keine konkrete Aussage über die Semantik nach einem Systemausfall. Geht man davon aus, daß für eine nicht vollständige Ausführung die Kompensation eingeleitet wird, stellen sich einige Fragen nach den Details. Ein konkretes Problem tritt dann auf, wenn direkt nach der Ausführung einer wirklich ausführenden Operation ein Systemausfall auftritt. Da die zugehörige ε-Operation nicht ausgeführt werden konnte, ist es nicht entscheidbar, ob die Operation tatsächlich erfolgreich war oder nicht. Selbst wenn nach dem Systemausfall versucht wird das Prädikat zu evaluieren ist nicht gewährleistet, daß dies das gleiche Resultat wie vor dem Systemausfall liefert.

5.5 Korrektheit in ConTracts

ConTracts unterscheiden sich von den bisher betrachteten Modellen prinzipiell nur in dem Punkt, daß in ihnen Transaktionsgrenzen explizit definiert werden können. Zum einen werden dadurch die bereits bei dem Modell von Korth et. al. erwähnten Probleme vermieden, und zum anderen wird die flexible Eingrenzung des Wirkungsbereiches von fehlgeschlagenen Operationen möglich.

Mit der Einführung der zusätzlichen Flexibilität ist es natürlich notwendig, auch den Korrektheitsbegriff neu zu überdenken bzw. festzulegen. Da im Gegensatz zu den bisher vorgestellten Modellen die Fortführbarkeit eines ConTracts eine der grundlegenden Eigenschaften darstellt, ist es darüber hinaus unumgänglich, den Recovery-Aspekt in die Definition eines Korrektheitskriteriums mit einzubeziehen. Wie sich herausstellen wird, basiert der in diesem Abschnitt vorgestellte Ansatz auf einer Kombination des PRED-Kriteriums (siehe Definition 5-20) und der prädikatbezogenen Serialisierbarkeit (siehe Definition 5-30).

5.5.1 Semantische Ununterbrechbarkeit von ConTracts

Der Begriff der Ununterbrechbarkeit oder Atomarität der ACID-Transaktionen fordert, daß Transaktionen entweder vollständig ausgeführt werden oder keinen sichtbaren Effekt haben. Formal kann dies folgendermaßen definiert werden:

Definition 5-31 (Ununterbrechbarkeit): *Gegeben sei eine Menge von Datenobjekten sowie jeweils eine Menge von Änderungs- und Beobachtungsoperationen. Eine Transaktion stellt eine Folge von Änderungs- bzw. Beobachtungsoperationen dar. Eine Transaktion heißt ununterbrechbar, wenn sie entweder erfolgreich ausgeführt wurde oder wenn nach ihrem Zurücksetzen alle Beobachtungsoperationen anderer Transaktionen die gleichen Resultate wie vor dem Start der fehlgeschlagenen Transaktion liefern.*■

Diese Definition der Ununterbrechbarkeit ist für kleine Ausführungseinheiten geeignet und unterstützt Anwendungsentwickler mit einer wohldefinierten Fehlersemantik. Wie sich jedoch gezeigt hat, gibt es allerdings viele Anwendungen, für die diese Definition der Ununterbrechbarkeit keinen Sinn macht. Speziell bei langlaufenden Anwendungen zeigt es sich, daß einmal gestartete Abläufe Auswirkungen haben, die nicht zurückgesetzt werden können oder sogar gewollt im System verbleiben sollen.

Deshalb wird die sogenannte *semantische Ununterbrechbarkeit* von ConTracts zweistufig definiert. Zum einen bieten ConTracts die Möglichkeit flexibel Transaktionsgrenzen festzulegen. Für die dadurch definierten Transaktionen wird die Ununterbrechbarkeit im klassischen Sinne zugesichert. Betrachtet man ConTracts als Ganzes, so wird bei einem Abbruch (der sogenannten Kompensation) garantiert, daß für alle Steps, die innerhalb erfolgreich abgeschlossener Transaktionen ausgeführt wurden sogenannte Kompensationssteps ausgeführt werden. Diese Kompensationssteps stellen nun allerdings keinen Zustand her, der bezüglich den Beobachtungsoperationen äquivalent zum Ausgangszustand des ConTracts ist. Vielmehr wird nur gefordert, daß nach der Ausführung eines Kompensationssteps der zugehörige Originalstep wieder ausgeführt werden kann [RSS97].

Für die Kompensationssteps selbst wird die Ununterbrechbarkeit gemäß Definition 5-31 gefordert. Somit laufen Kompensationssteps wie alle anderen Steps unter dem Schutz von ACID-Transaktionen ab. Ein wichtiger Unterschied zu nichtkompensierenden Steps ist allerdings, daß Kompensationssteps unbedingt erfolgreich ausgeführt werden müssen, wenn die Kompensation eines ConTracts ausgelöst wurde. Sollte also während einer Kompensation die Ausführung eines Kompensationssteps fehlschlagen, ist zunächst zu versuchen die zugehörige Transaktion zurückzusetzen und die Ausführung erneut zu starten. Schlägt der (möglicherweise mehrfach) wiederholte Versuch der Ausführung fehl, befindet sich das System in einem inkonsistenten Zustand, der einen manuellen Eingriff er-

fordert.

Da ConTracts die Fortsetzbarkeit garantieren ergibt sich ein weiterer Unterschied zu den ACID-Transaktionen. Anstatt zur Recovery nach einem Systemausfall alle aktiven Transaktionen zurückzusetzen und dann die Kompensation einzuleiten, werden alle zurückgesetzten Transaktionen erneut gestartet und die Ausführung des ConTracts fortgesetzt.

5.5.2 Historien in ConTracts

Prinzipiell läßt sich die bereits bekannte Definition von Historien auch auf das ConTract-Modell übertragen. Für die weitere Diskussion des Korrektheitskriteriums von ConTracts ist es jedoch sinnvoll, die Besonderheiten explizit herauszuarbeiten.

Die Operationen der abstrakten Maschine zur Abarbeitung von ConTracts kennt insgesamt 8 Operationen, die bereits in Abschnitt 4.4.3 eingeführt wurden:

1. Die execute-Operation: e
2. Die check-Operation: γ
3. Die establish-Operation: ε
4. Die BOT-Operation: b
5. Die EOT-Operation: c
6. Die Abort-Operation: a
7. Die compensate-Operation: k
8. Die EOC-Operation: f

Somit ergibt sich die Menge der zulässigen Operationen O zu {e,γ,ε,b,c,a,k,f}.

Definition 5-32 (Historie eines ConTract-Systems): *Eine Historie H eines ConTract-verarbeitenden Systems ist eine Partialordnung* $(\Sigma, <)$ *mit* $\Sigma = \{o_i\}$ *und* $o_i \in O$. *Eine Historie* $H_C=(\Sigma_C, <_C)$ *heißt Projektion einer Historie H auf eine ConTract-Instanz C, wenn* Σ_C *eine Teilmenge von* Σ *ist, nur Operationen von C enthält und für jedes Paar von Operationen a und b aus* Σ_C *gilt, daß aus* $a < b$ *in H,* $a <_C b$ *in* H_C *folgt.*

Eine ConTract-Instanz ist in einer Historie enthalten, wenn die Projektion der Historie auf die ConTract-Instanz nicht leer ist.■

Die allgemeine Form der Historie ist natürlich unabhängig von dem Verarbeitungszustand einer ConTract-Instanz. Da dieser andererseits anhand der Historie durch-

aus bestimmt werden kann, bietet es sich an, auch eingeschränkte Formen formal zu definieren.

Definition 5-33 (Vollständigkeit): *Eine Historie H heißt vollständig, wenn für alle in ihr enthaltenen ConTract-Instanzen C_i gilt, daß $f_i = EOC(C_i)$ in H enthalten ist.* ■

Vollständige Historien sind somit Historien, die nur abgeschlossene ConTract-Instanzen beinhalten. Eine weitere Spezialisierung der allgemeinen Historien, sind solche, die zwar noch nicht abgeschlossene ConTract-Instanzen enthalten, bei denen jedoch keine Operationen enthalten sind, die zu aktiven Transaktionen gehören. Um diese Historien ebenfalls formal erfassen zu können, sind zunächst zwei Hilfskonstrukte notwendig.

Definition 5-34 (Transaktionsbezeichner): *Für jede Operation o der Menge der Anwendungsoperationen $A=\{e,\gamma,\varepsilon\}$, die in einer Historie H enthalten sind, existiert eine Abbildung t(o), welche einer Anwendungsoperation o einen Transaktionsbezeichner zuordnet. t(o) identifiziert die Transaktion, unter deren Schutz o in H ausgeführt wurde. Mit T(o) wird die Abbildung bezeichnet, die o bezüglich H den Transaktionsbezeichner der Top-Level-Transaktion zuordnet unter der o ausgeführt wurde.* ■

Die Abbildung t dient dazu, in einer Historie die Transaktion zu ermitteln, unter der eine Anwendungsoperation abgelaufen ist.

Definition 5-35 (ConTract-Bezeichner): *Für jede Operation o, die in einer Historie enthalten ist, existiert eine Abbildung C(o), welche o einen ConTract-Instanz-Bezeichner zuordnet. C(o) identifiziert die ConTract-Instanz, für die o ausgeführt wurde.* ■

Die Abbildung C(o) ermittelt in einer Historie die ConTract-Instanz, für die eine Operation o ausgeführt wurde.

Definition 5-36 (Transaktionskonsistenz): *Eine Historie H eines ConTract-verarbeitenden Systems heißt transaktionskonsistent wenn für alle Operationen, welche Anwendungsoperationen sind, gilt:*

$$\forall_{(o \in (\Sigma \cap A))} a(t(o)) \in \Sigma \vee c(t(o)) \in \Sigma \quad ■$$

Eine transaktionskonsistente Historie enthält nur Operationen, deren zugehörige Transaktion entweder erfolgreich abgeschlossen oder zurückgesetzt wurde. Die Forderung nach der Fortführbarkeit von ConTracts nach einem Systemausfall bedingt nun, daß das Resultat der Recovery nach einem Systemstart ein System ist, dessen Historie transaktionskonsistent ist.

Definition 5-37 (Recovery-Erweiterung): *Eine Recovery-Erweiterung H_R einer Historie H ist eine transaktionskonsistente Historie, welche aus H durch*

Anwendung der folgenden Regeln entsteht:

1. *Alle Operationen von H sind auch in H_R enthalten.*
2. *Gilt $o_i < o_j$ in H, dann gilt auch $o_i < o_j$ in H_R.*
3. *Für alle Transaktionen t für die weder a(t) noch c(t) in H enthalten ist, wird a(t) der Historie H_R hinzugefügt. Es gilt $o_j < a(t)$ in H_R wenn entweder $t(o_j) = t(a(t))$ oder wenn es eine Operation o_i in H gibt mit $o_j < o_i$, $t(o_j) \neq t(o_i)$ und $t(o_i) = t(a(t))$.*
4. *Für alle abort-Operationen die in H_R aber nicht in H enthalten sind gilt: $a(t_i) < a(t_j)$, $t_i \neq t_j$ in H_R, wenn es zwei Operationen o_j , o_i gibt mit $t(o_j) = t_j$ und $t(o_i) = t_i$ und weiterhin gilt daß $o_j < o_i$ in H.* ■

Die Recovery-Erweiterung einer Historie erweitert eine gegebene Historie um jene Operationen, welche für die Beschreibung des Abbruches der aktiven Transaktionen notwendig sind. Dies entspricht dem Ansatz, der bei der Definition des PRED-Kriteriums zum Einsatz kam. Da der Detaillierungsgrad der Historien eines ConTract-verarbeitenden Systems relativ grob-granular ist und darüber hinaus die Ausführung der Anwendungsoperationen auf Ebene eines Contracts atomar erscheint, soll im weiteren angenommen werden, daß die Recovery-Erweiterung Historien nur um die notwendigen Abbruchoperationen der abstrakten Maschine zur Bearbeitung von ConTracts erweitert.

Da, entsprechend dem Ansatz der Präfix-Reduzierbarkeit, ein Korrektheitskriterium für ein ConTract-verarbeitendes System sowohl die Durchlässigkeit als auch die semantische Atomarität berücksichtigen soll, ist es notwendig, die Korrektheit von Historien im Hinblick auf eine eventuelle Kompensation beurteilen zu können. Im Gegensatz zu den bisher betrachteten Modellen ist es dazu allerdings notwendig, den Mechanismus zur Kompensation nochmals genauer zu betrachten.

Wie bereits in Abschnitt 4.4.2 eingeführt, werden Steps nicht einfach Kompensationssteps sondern sogenannte Kompensationsblöcke zugeordnet. In der Historie eines ConTract-verarbeitenden Systems werden deshalb Kompensationsaktionen nicht als einfache execute-Operationen widergespiegelt. Statt dessen ist die Kompensation einer einzelnen execute-Operation eine Folge von Operationen, die zu einer Transaktion zusammen gefaßt werden.

Definition 5-38 (Kompensationsfolge): *Eine Kompensationsfolge I(k) ist eine Interpretation eines Kompensationsblockes k nach Definition 4-21. Es existiert eine zweistellige Relation comp(e, I(k)), mit folgender Eigenschaft:*

comp(e,I(k)) gilt, wenn die execute-Operation e aus einer Step-Instanz $\tilde{s}$ hervorgeht, welche wiederum aus einem Step s einer ConTract-Instanz erzeugt wurde und comp(s, k) gilt.

Gilt comp(e, I(k)), wird für I(k) die Schreibweise $I(e^{-1})$ verwendet. ■

Die Definition der Kompensationsfolge dient zur verkürzten Schreibweise bei der Definition von Historien. Sie abstrahiert von der Tatsache, daß bei ConTracts im allgemeinen mehrere Operationen zur Kompensation einer Operation notwendig sind.

Definition 5-39 (Kompensations-Erweiterung): *Eine Kompensations-Erweiterung $\hat{H} = (\hat{\Sigma}, \hat{\rightarrow})$ einer Recovery-Erweiterung H_R einer Historie H entsteht aus H_R durch folgende Regeln:*

$\hat{\Sigma}$ *ist eine Menge von Operationen, die aus Σ_R in der folgenden Weise entsteht:*

1. *Für alle Operationen o gilt:* $o \in \Sigma_R \Rightarrow o \in \hat{\Sigma}$
2. *Für alle e-Operationen in Σ_R gilt:*
   $(e <_R k(C(e)) \vee k(C(e)) \notin \Sigma) \wedge c(T(e)) \in \Sigma \wedge f(C(e)) \notin \Sigma_R \Rightarrow I(e^{-1}) \in \hat{\Sigma}$

Die Ordnungsrelation $\hat{\rightarrow}$ ist folgendermaßen festgelegt:

1. *Für jeweils zwei Operationen o_i und o_j gilt:* $o_i <_R o_j \Rightarrow o_i \hat{\rightarrow} o_j$
2. *Alle nicht-kompensierenden Operationen einer ConTract-Instanz erscheinen vor den Interpretationen ihrer Kompensationsblöcke in der Historie.*
3. *Für alle Paare von Kompensationsfolgen $I(e_1^{-1})$ und $I(e_2^{-1})$ gilt:*
   $e_1 <_R e_2 \Rightarrow I(e_2^{-1}) \hat{\rightarrow} I(e_1^{-1})$
   wobei die Relation $\hat{\rightarrow}$ dann für zwei Kompensationsfolgen gelten soll, wenn sie für die letzte Operation der ersten Folge und die erste Operation der zweiten Folge gilt. ■

Die Kompensations-Erweiterung stellt die Analogie zu den erweiterten Historien (siehe Definition 5-18) dar. Die Idee dabei ist, Historien um die Kompensationsfolgen zu erweitern, welche zur Kompensation nicht abgeschlossener ConTract-Instanzen notwendig sind. Notwendig sind dabei Kompensationsfolgen für execute-Operationen erfolgreich abgeschlossener Top-Level-Transaktionen. Die Reihenfolge, in der die Kompensationsfolgen auszuführen sind, entspricht dabei der umgekehrten Reihenfolge der Originaloperationen (sofern eine solche vorhanden ist). Implizit wird davon ausgegangen, daß innerhalb der Kompensationsfolgen die Partialordnung zu einer Totalordnung verschärft wird.

5.5.3 Konfliktbegriff von ConTracts

Ähnlich dem Modell von Korth et al. stellen die Invarianten Prädikate dar, die Bedingungen für die Ausführbarkeit von execute-Operationen festlegen. Allerdings sind die sogenannten Eingangsinvarianten nicht wie die Eingangsprädikate bei Korth beliebig definierbar. Statt dessen beziehen sie sich auf (Teil-)Prädikate, welche bereits von früheren Operationen als Ausgangsinvarianten etabliert wur-

den. Trotz dieses Unterschiedes ließe sich nun der Konfliktbegriff von ConTracts analog definieren. Bei genauerer Betrachtung aus dem Blickwinkel lang laufender Ausführungen erkennt man jedoch, daß diese Auffassung des Konfliktbegriffs weit restriktiver als eigentlich notwendig ist.

Die Ursache der unnötig weitgehenden Beschränkung liegt in der von der Auswertung des Prädikates unabhängigen Definition eines Konfliktes. D.h. es kann ein Konflikt zweier Operationen bezüglich eines Prädikates vorliegen, obwohl das Prädikat selbst nicht verletzt wird. Darüber hinaus wird bei Korth gefordert, daß ein Eingangsprädikat bis zum Ende der Ausführung nicht verletzt wird, obwohl aus Sicht der Operationen kein Bedarf für diese Anforderung besteht.

Das Ziel bei der Entwicklung des Konfliktbegriffs für ConTracts ist es, die Nachteile des Ansatzes von Korth et. al. zu vermeiden und somit einen höheren Grad an Parallelverarbeitung zuzulassen. Da die Eingangsinvarianten nur Referenzen auf bereits etablierte Ausgangsinvarianten sind, bilden die Invarianten eine Art Klammer, welche einen Isolationsbedarf repräsentiert. Somit ist ein Konflikt bezüglich einer Invariante (bzw. eines Teils einer Invariante) nur innerhalb dieser Klammer von Relevanz[1].

Definition 5-40 (Geschlossene Invariantenklammer): *Zwei Operationen* ε *und* γ *einer ConTract-Instanz bilden eine geschlossene Invariantenklammer* $[\varepsilon,\gamma]$*, wenn gilt, daß sowohl* ε *als auch* γ *in einer Historie enthalten sind und die Eingangsinvariante der* γ*-Operation eine Prädikat-Referenz (Definition 4-11) auf die Ausgangsinvariante enthält, die von der* ε*-Operation etabliert wird.*

Eine Operation o liegt innerhalb einer geschlossenen Invariantenklammer $[\varepsilon,\gamma]$ *bezüglich einer Historie* ($o \in [\varepsilon,\gamma]$)*, wenn o nach der* ε*-Operation und vor der* γ*-Operation ausgeführt wurde.* ■

Eine geschlossene Invariantenklammer ist sozusagen ein Gültigkeitsbereich eines Teils einer Ausgangsinvariante. Innerhalb dieses Bereiches muß der Teil der Ausgangsinvariante gelten, der von der Eingangsinvarianten referenziert wird.

Problematisch bei der Betrachtung der geschlossenen Invariantenklammer ist allerdings die Tatsache, daß es nicht immer entscheidbar ist, ob jemals eine geschlossene Invariantenklammer vorliegen wird, wenn eine ε-Operation ausgeführt wurde. Dies ist zum einen darauf zurückzuführen, daß es innerhalb einer ConTract-Instanz bedingte Verzweigungen gibt. Zum anderen ist gewährleistet, daß zu jeder Zeit eine Kompensation eingeleitet werden kann, wodurch ebenfalls die Menge der Pfade verändert wird, die ausgeführt werden können.

Als Konsequenz aus dieser Beobachtung ist es notwendig, die eingeführte Notation der Invariantenklammer zu verfeinern, um so ein Kriterium zu erhalten, welches

1. Diese Aussage trifft nur für Abläufe zu, deren Struktur sich nicht zur Laufzeit ändert.

sich bei beliebigen Historien beurteilen läßt.

Definition 5-41 (Offene Invariantenklammern): *Eine ε-Operation, die in einer Historie H enthalten ist, heißt rechts offene Invariantenklammer: [ε,-). Eine Operation o ist Teil einer rechts offenen Invariantenklammer o ∈ [ε,-), wenn o in H enthalten ist und nach der ε-Operation ausgeführt wurde.*

Eine γ-Operation, die in einer Historie H enthalten ist heißt links offene Invariantenklammer: (-,γ]. Eine Operation ist Teil einer links offenen Invariantenklammer o ∈ (-,γ], wenn o in H enthalten ist und vor der γ-Operation ausgeführt wurde. ■

Offene Invariantenklammern teilen eine Historie H grundsätzlich in Operationen, die vor bzw. nach einer Invariantenoperation ausgeführt wurden. Die Konstruktion der geschlossenen Invariantenklammer aus offenen Invariantenklammern ist trivial, so daß hier nicht näher darauf eingegangen wird.

Auf der Basis der Invariantenklammern kann nun die Konfliktrelation definiert werden:

Definition 5-42 (Invariantenbasierte Konflikte): *Gegeben sei eine Historie H. Eine execute-Operation e_1 einer ConTract-Instanz C_1 und eine execute-Operation e_2 einer ConTract-Instanz C_2, C1 ≠ C2, sind in Konflikt bezüglich einer Ausgangsinvarianten auf Grund eines Prädikates p_k: $conf_\varepsilon(e_1,e_2,p_k)$, wenn gilt:*

1. *p_k ist Teil der Ausgangsinvarianten, die in einer ε-Operation für e_1 etabliert wird: $\varepsilon = establish(C_1, t, o, e_1)$.*
2. *$e_2 \in [\varepsilon,-)$*
3. *p_k ist nach der Ausführung von e_2 nicht erfüllt.*

$conf_\varepsilon(e_1,e_2,p_k) \Rightarrow \varepsilon < e_2$ in H.

Eine execute-Operation e_1 einer ConTract-Instanz C_1 und eine execute-Operation e_2 einer ConTract-Instanz C_2, C1 ≠ C2, sind in Konflikt bezüglich einer Eingangsinvarianten auf Grund einer Prädikat-Referenz r_k: $conf_\gamma(e_1,e_2,r_k)$, wenn gilt:

1. *r_k ist Teil der Eingangsinvarianten, die in einer γ-Operation für e_1 geprüft wird: $\gamma = check(C_1, t, i, e_1)$.*
2. *$e_2 \in (-,\gamma]$*
3. *r_k ist nach der Ausführung von e_2 nicht erfüllt.*

$conf_\gamma(e_1,e_2,r_k) \Rightarrow \gamma < e_2$ in H. ■

Konflikte nach Definition 5-42 basieren im Gegensatz zum allgemeinen Konfliktbegriff (siehe Definition 5-4) auf einer bestimmten Ordnung der Operationen. Al-

lerdings ist bei den eingeführten Definitionen noch nicht berücksichtigt, ob überhaupt eine überlappende Ausführung von ConTract-Instanzen vorliegt oder nicht.

5.5.4 Invariantenorientierte Serialisierbarkeit

Um die Korrektheit von Historien beurteilen zu können, ist die Auswirkung der Konflikte auf die umgebenden ConTract-Instanzen zu untersuchen. Ein Aspekt dabei ist, daß eine ConTract-Instanz nur dann von einem Konflikt auf der Ebene der Operationen beeinflußt werden kann, solange sie aktiv ist. Diese Tatsache hat zwei grundsätzliche Implikationen:

1. Operationen vor dem Startereignis einer ConTract-Instanz können nicht in einem Eingangsinvarianten-Konflikt mit einer Operation der ConTract-Instanz stehen.
2. Operationen nach der Ende-Operation einer ConTract-Instanz können nicht in einem Ausgangsinvarianten-Konflikt mit einer Operation der ConTract-Instanz stehen.

Berücksichtigt man diese Beobachtungen und propagiert die Ordnungsrelationen der in Konflikt stehenden Operationen auf die Ebene der ConTract-Instanzen, gelangt man zu folgendem Ordnungsbegriff:

Definition 5-43 (Invariantenbasierte Ordnung): *Zwei ConTract-Instanzen C_A und C_B stehen in einer Ordnungsrelation $<_p$ bezüglich eines Prädikates p: $C_A <_p C_B$, wenn es eine Historie $H = (\Sigma,<)$ gibt, in der gilt:*

1. Es gibt zwei e-Operationen e_A und e_B der ConTract-Instanzen in H.

2. e_B wurde vor $EOC(C_A)$ ausgeführt und $conf_\varepsilon(e_A,e_B,p)$
oder
$Start(C_B)$ wurde vor e_A ausgeführt und $conf_\gamma(e_B,e_A,r)$, wobei $r=(o,p)$

Die Ordnungsrelation $<_p$ ist transitiv. Die transitive Hülle wird mit $<_p^*$ bezeichnet. ■

Die Ordnung, die ein Konflikt zweier Operationen unterschiedlicher ConTract-Instanzen impliziert, entspricht der Ordnung der in Konflikt stehenden Operationen. Somit ist wiederum die Basis geschaffen, ein Kriterium zu formulieren, welches der klassischen Serialisierbarkeit ähnelt.

Definition 5-44 (Invariantenbasierte Serialisierbarkeit): *Eine Historie H ist korrekt wenn:*

1. Alle in ihr enthaltenen Projektionen von ConTract-Instanzen eine Interpretation einer wohlgeformten Contract-Instanz sind.

2. Für alle ConTract-Instanzen C_i in der Kompensations-Erweiterung $\hat{H}$ gilt, daß es kein Prädikat p gibt, so daß:

$$C_i <^*_p C_i$$

Die Menge aller invariantenbasierten-serialisierbaren Historien wird mit I-SR bezeichnet. ■

Auf Grund des eingeführten Konfliktbegriffs ist einfach zu zeigen, daß das Kriterium in Definition 5-44 Präfix-abgeschlossen ist und sich somit auch für den Entwurf eines Schedulers eignet. Da das Kriterium - abgesehen von der Sicherstellung der strukturellen Korrektheit - auf der Kompensations-Erweiterung einer Historie basiert, können die Invarianten grundsätzlich in zwei Kategorien unterschieden werden:

1. Invarianten, die bezüglich einer Kompensationsaktion eine Invariantenklammer bilden.
2. Invarianten, die bezüglich einer "normalen" Operation eine Invariantenklammer bilden.

Diese Kategorien werden unterschieden, da eine Verletzung einer Invariante der ersten Kategorie durch das Korrektheitskriterium ausgeschlossen ist, während dies im zweiten Fall zugelassen wird. Diese Eigenschaft kann bei der Entwicklung von Verfahren zur Sicherstellung der Korrektheit benutzt werden, was im folgenden Kapitel noch deutlich werden wird.

5.5.5 Kaskadierende Kompensation

Ein Problem der Atomarität, das bei dem Lese-/Schreib-Modell bereits besprochen wurde, das kaskadierende Zurücksetzen, hat ein Pendant im Falle der semantischen Ununterbrechbarkeit: die kaskadierende Kompensation. Unter kaskadierender Kompensation versteht man die Notwendigkeit, bei der Kompensation einer ConTract-Instanz, die Kompensation einer anderen ConTract-Instanz auslösen zu müssen.

Wie beim Lese-/Schreibmodell kann diese Fortsetzung dann notwendig werden, wenn eine ConTract-Instanz A Daten gelesen hat, die von einer zweiten ConTract-Instanz B verändert wurden und die Instanz B die Kompensation einleitet. Grundsätzlich stellt sich dann die Frage, ob die Daten, welche Instanz A gelesen hatte, trotz der Kompensation von B gültig sind. Die Antwort auf diese Frage ist anwendungsabhängig und kann allgemein nicht beantwortet werden.

Im Gegensatz zum Ansatz der klassischen Transaktionen wird deshalb in ConTracts davon ausgegangen, daß keine kaskadierende Kompensation notwendig

ist. Vielmehr obliegt es dem Anwendungsprogrammierer, das Konzept der Invarianten so zu nutzen, daß ein inkorrektes Verhalten einer ConTract-Instanz auf Grund der Kompensation einer anderen Instanz von vornherein ausgeschlossen ist.

Man kann diese sehr weit reichende Annahme durch Beobachtungen von langlebigen Abläufen aus realen Anwendungen rechtfertigen. Ein wichtiger Unterschied zur transaktionalen Welt ist die Tatsache, daß Resultate, die einmal sichtbar waren, nicht dadurch "ungültig" werden, daß ein Vorgang storniert wird. Zumindest für eine bestimmte Zeit war ein Resultat gültig, so daß dieser Umstand unter Berücksichtigung des Zeitaspekts auch nicht mehr "ungeschehen" gemacht werden kann.

Ein Beispiel aus dem Bankwesen soll diese Beobachtung verdeutlichen. Wird eine Reisebuchung durchgeführt, wird normalerweise eine Anzahlung geleistet. Unter der Annahme, daß diese Zahlung von einem Konto abgebucht wird, wird die Aktion dieses Reisebuchungsablaufes für einen anderen Ablauf zur Zinsberechnung sichtbar. Wird die Reisebuchung storniert, bleiben die von der Zinsberechnung ausgeführten Aktionen bestehen (wenn auch zum Leidwesen des oder der Buchenden). Eine kaskadierende Kompensation kommt in diesem Beispiel nicht in Frage, da die Tatsache, daß für eine bestimmte Zeit weniger Geld auf dem Konto war, nicht ungeschehen gemacht werden kann. Selbst wenn der entstehende Zinsverlust von dritter Seite ausgeglichen wird, ist zwar der Effekt quasi kompensiert worden, jedoch ohne Beeinflussung des Ablaufes zur Zinsberechnung.

Andererseits ist unverkennbar, daß sich parallele Abläufe beeinflussen. Dieser Effekt ist teilweise sogar gewünscht, wodurch der Bedarf nach Ablauf-übergreifendem Informationsaustausch entsteht. Diese Art des Zusammenspiels von Abläufen ist jedoch nicht Gegenstand dieser Arbeit und wird deshalb nicht weiter diskutiert. Trotzdem kann man sich vorstellen, daß der hier präsentierte Begriff der Korrektheit bei weitem nicht ausreicht, um für ein solch komplexes Szenario zu genügen.

5.6 Diskussion

Die in diesem Kapitel vorgestellten Korrektheitsbegriffe stellen nur einen Ausschnitt aus der Vielfalt entwickelter Kriterien dar. Gerade im Zusammenhang mit dem klassischen Serialisierbarkeitsbegriff im Umfeld des Lese/Schreib-Modells wurden einige Verfeinerungen entwickelt, um eine theoretische Grundlage für die im nachfolgenden Kapitel diskutierten Verfahren einzuführen. Grundsätzlich werden durch diese Verfahren jedoch kaum neue Aspekte aufgeworfen, so daß auf eine tiefergehende Einführung dieser Verfahren verzichtet werden kann.

Bei genauerer Betrachtung der vorgestellten Verfahren lassen sich zwei orthogonale Klassifikationskriterien für Korrektheitsbegriffe identifizieren:

1. Art der Konfliktbestimmung
2. Art der Isolationsbedarfsfestlegung.

Die Art der Konfliktbestimmung entscheidet darüber, ob das Kriterium auf einem Konfliktbegriff basiert, welcher nur durch die Betrachtung der Signatur der Operationen festgelegt ist. Ist dies der Fall, so spricht man von einem *syntaktischen* Verfahren. Muß dagegen die Bedeutung der Operationen selbst betrachtet werden, um über einen Konflikt entscheiden zu können, nennt man das Verfahren *semantisch*.

Die Art der Isolationsbedarfsfestlegung dient dazu, aus Sicht einer Anwendung zu beurteilen, wie der Isolationsbedarf geäußert werden kann. Muß in der Anwendung selbst hierzu keinerlei Maßnahme getroffen werden spricht man von einem *impliziten* Verfahren, anderenfalls von einem *expliziten* Verfahren. Diese Unterscheidung sagt natürlich nichts darüber aus, ob auf der Stufe des Ausführungssystems explizit festgelegt wurde, ob zwei Operationen in einem Konflikt stehen (z.B. durch sogenannte Kompatibilitätstabellen), oder ob diese Konflikterkennung implizit im Programmcode versteckt ist.

Tabelle 5-1 gibt einen Überblick über die Klassifikation der vier wichtigsten Kriterien, die in diesem Kapitel vorgestellt wurden. Anhand der Tabelle wird deutlich, daß die vier vorgestellten Modelle alle Kombinationen der zwei Klassifikationskriterien repräsentieren und somit hinreichend repräsentativ sind.

Auffällig ist, daß der Korrektheitsbegriff nach Korth et. al. die explizite Festlegung des Isolationsbedarfs unterstützt und gleichzeitig eine syntaktische Konfliktbestimmung verfolgt. Dies resultiert aus der Tatsache, daß das Korrektheitskriterium nach Korth zwar Prädikate verwendet, die auf Anwendungsebene definiert werden, andererseits aber ein Konflikt nicht darauf beruht, daß ein Prädikat verletzt wird. Ein Konflikt beruht nur auf der Tatsache, daß ein Objekt “syntaktisch” in einem Prädikat verwendet wird und eine ändernde Operation auf dieses Objekt zugreift. Die Art des Zugriffs beeinflußt die Entscheidung über einen Konflikt nicht.

Kriterium	Modell	Konfliktbestimmung	Isolationsbedarf
PRED	read/write-Modell	syntaktisch	implizit
ML-SR	Multilevel	semantisch	implizit
P-SR	Korth et. al.	syntaktisch	explizit
I-SR	ConTracts	semantisch	explizit

Tabelle 5-1: Klassifikation von Korrektheitskriterien

Ein wichtiges Vergleichskriterium für Korrektheitsbegriffe ist die Betrachtung der möglichen korrekten Historien. Grundsätzlich gilt dabei, daß bei einer größeren Menge an möglichen Transaktionen auch der mögliche Parallelitätsgrad zunimmt und somit ein höherer Durchsatz (im Hinblick auf Transaktionen pro Zeiteinheit) erzielt werden kann. Wie bereits in Abschnitt 5.2 im Fall des Lese/Schreib-Modells eingeführt wurde, wird ein entsprechender Vergleich durch eine Teilmengenbeziehung der Mengen möglicher Historien ausgedrückt.

Eine Vorgehensweise zu einer Aussage über Teilmengenbeziehungen zu kommen ist die Untersuchung eines Verfahrens auf die Möglichkeit der Simulation eines anderen Verfahrens. Einfach kann dies am Beispiel des Modells nach Korth und dem ConTract-Modell vorgenommen werden. Schränkt man bei den ConTracts die Prädikate so ein, daß eine Verletzung des Prädikats bei einem beliebigen ändernden Zugriff eines enthaltenen Datenelements auftritt und definiert die Eingangsinvariante des letzten Steps als Konjunktion aller vorigen Eingangsinvarianten, so erhält man das gleiche Verhalten wie bei dem Ansatz von Korth. Auch das PRED-Kriterium ist mit dem Ansatz von ConTracts entsprechend zu simulieren. Eine Aussage über die Teilmengenbeziehung mit den Mehrschichttransaktionen ist allerdings nicht möglich, da bei diesen zum einen eine strenge Hierarchie gefordert wird und zum anderen nicht festgelegt ist, welches Verfahren auf einer bestimmten Ebene zur Anwendung kommt.

Darüber hinaus kann für alle Verfahren, die eine Kompatibilitätstabelle für eine beliebige (erweiterbare) Menge von Operationen verwenden, keine Aussage über eine Teilmengenbeziehung gemacht werden, da der Konfliktbegriff zwar definiert aber veränderlich bezüglich der Zeit bei konstantem Zustand der Daten ist.

Der Zeitaspekt wird momentan von keinem Korrektheitsbegriff berücksichtigt, obwohl dieser im Umfeld langdauernder Abläufe eine nicht zu vernachlässigende Rolle spielt. Ein Beispiel soll dies verdeutlichen. Ein Ablauf zur Abwicklung eines Hauskaufs möge einen Step enthalten, der dazu dient festzustellen, ob für die spätere Zahlung des Kaufpreises genügend Geld zur Verfügung steht. Aus Sicht der Anwendung muß ab dem Zeitpunkt der Überprüfung nur sichergestellt werden, daß zum Zeitpunkt der Zahlung ein entsprechender Betrag zur Verfügung steht. Somit könnte unter dieser Voraussetzung jeglicher Zugriff auf das Budget erlaubt werden.

Bei genauerer Betrachtung erkennt man, daß das Beispielsszenario zwei Aspekte enthält, die heutige Korrektheitskriterien nicht berücksichtigen (können). Der eine Aspekt ist die bereits erwähnte temporale Beschränkung von Isolationskriterien. Der zweite Aspekte, der auch den eigentlich kritischen Bereich darstellt, ist die Berücksichtigung einer Garantie über eine zukünftige Aktion bzw. einen zukünftigen Zustand der Daten. Obwohl es in der Realität Usus ist, Garantien über zukünftige

Ereignisse abzugeben, kann dies bei Korrektheitsmodellen kaum berücksichtigt werden, da solche Garantien nur begrenzt eingehalten werden können und bei einer Verletzung ein nicht korrekter bzw. inkonsistenter Zustand entstehen kann.

Trotzdem ist es speziell im Umfeld langlebiger Abläufe notwendig, weitgehend die Flexibilität der Realität nachbilden und somit auch inkonsistente Zustände berücksichtigen zu können. Eine zukünftige Aufgabe der Entwicklung von Korrektheitskriterien wird es somit sein, Inkonsistenzen nicht grundsätzlich zu verhindern, sondern kontrolliert mit ihnen umzugehen.

6 Kontrolle von Abläufen

Die Einführung von Korrektheitskriterien im transaktionalen Umfeld ist die Voraussetzung, um Mechanismen zur Vermeidung von Anomalien bei der parallelen Ausführung von Abläufen in ein Laufzeitsystem zu integrieren und somit Anwendungsprogrammierer und -programmiererinnen von diesem Aufwand zu entlasten. Neben dem Vorteil, daß insgesamt der Code-Umfang der Anwendungen dadurch geringer wird, können die Mechanismen des Laufzeitsystems von Spezialisten entwickelt und optimiert werden. Somit wird das Fehlerrisiko minimiert und gleichzeitig ein optimales Leistungsverhalten garantiert. Dieses Kapitel beschäftigt sich mit den unterschiedlichen Ansätzen zur Umsetzung der im vorigen Kapitel eingeführten Korrektheitskriterien in eben solche Laufzeitsysteme, so daß sich der Begriff der "Kontrolle von Abläufen" nur auf die Sicherstellung der transaktionalen Korrektheit bezieht (engl. *concurrency control*) und nicht auf andere Bereiche wie z.B. den Datenschutz.

Wie sich herausstellen wird, gibt es eine Vielzahl von Möglichkeiten die vorgestellten Kriterien in ein Laufzeitsystem umzusetzen. Neben der eigentlichen Aufgabe, der Sicherstellung der Korrektheit selbst, werden dabei noch weitere Aspekte eine Rolle spielen, wobei der Leistungsaspekt im Hinblick auf das Durchsatzverhalten des Gesamtsystems, das überwiegend wichtigste Beurteilungskriterium zur Bewertung der Verfahren sein wird.

6.1 Grundprobleme

Wie im Falle der Korrektheitskriterien können auch bei deren Umsetzung grundsätzliche Probleme identifiziert werden, die es zu lösen gilt. Die drei nachfolgenden Unterabschnitte sollen einen Einblick in diese Probleme geben und eine Motivation für die Vielzahl an Ansätzen liefern.

6.1.1 Statische versus dynamische Ansätze

Wie bereits erwähnt, ist das Ziel der Umsetzung von Korrektheitskriterien die Entwicklung eines Laufzeitsystems. Die Hauptkomponente eines solchen Systems ist dabei die Komponente, die über die Ausführbarkeit bzw. die Zulassung einer Transaktion zur Ausführung entscheidet: der sogenannte *Scheduler*. Aus Sicht dieses Schedulers wird dessen Aufgabe um so einfacher, je mehr Information über die Operationen und die benötigten Datenobjekte der Transaktionen vor deren Ablauf zur Verfügung stehen.

Ein Ansatz zur Sicherstellung von korrekten und (Durchsatz-)optimalen Historien

basiert auf dieser Beobachtung. Bei diesem Ansatz wird gefordert, daß eine Transaktion bereits vor ihrem eigentlichen Ablauf alle benötigten Datenelemente "reserviert" (engl. *pre-claiming*). Wird außerdem noch die Art des Zugriffs spezifiziert (wie z.B. ändernder Zugriff oder nicht-ändernder Zugriff), ist es einem Scheduler prinzipiell möglich, die verschachtelte Ausführung von Transaktionen einfach festzulegen und (nahezu) jede Konfliktsituation vorherzusehen. Da sich die Information, die zur Erkennung von Konflikten benötigt wird, zur Laufzeit einer Transaktion nicht ändert, wird diese Art von Ansätzen zur Sicherstellung[1] der Korrektheit als *statisch* (im engl. *conservative* [BHG87]) bezeichnet.

Voraussetzung für den Einsatz von statischen CC-Verfahren ist jedoch, daß die Menge der berührten Objekte einer Transaktion vorab bekannt ist und daß Transaktionen möglichst keine bedingten Verzweigungen enthalten. Letzteres ist zwar vom Standpunkt der Korrektheit nicht unbedingt notwendig, da jedoch ganze Zweige einer Transaktion möglicherweise nicht durchlaufen werden, sind alle Maßnahmen zur Konfliktvermeidung für Objekte, die nur in diesen Zweigen berührt werden unnötig und resultieren in einer Verschlechterung des Leistungsverhaltens.

Durch diese notwendigen Voraussetzungen eignen sich die statischen Verfahren nur für eine sehr beschränkte Klasse von Anwendungen, weshalb in anwendungsunabhängigen Produkten wie z.B. Datenbanksystemen eine andere Klasse von Verfahren zur Anwendung kommt: die *dynamischen* Verfahren.

Dynamische CC-Verfahren erfordern keine Reservierung der in einer Transaktion benötigten Datenelemente vor der Ausführung einer Transaktion. Statt dessen wird bei der Ausführung einer Operation ermittelt, welche Datenelemente berührt werden und welche Konflikte mit bereits ausgeführten Operationen anderer Transaktionen hierdurch entstehen. Anhand dieser Information entscheidet der Scheduler ob die Operation ausgeführt werden darf, bzw. welche Maßnahmen getroffen werden müssen, um die Korrektheit des Systems zu gewährleisten.

Obwohl durch den dynamischen Ansatz jede einzelne Operation einer Transaktion verzögert wird, rechtfertigt sich der Einsatz einer solchen Methodik durch den hohen Flexibilitätsgrad und der Unabhängigkeit von der Menge der von Transaktionen berührten Datenelemente.

6.1.2 Durchsatz und Verklemmung

Im Bereich der ACID-Transaktionen ist neben der Korrektheit selbst der Durchsatz bzw. eine Durchsatzsteigerung bezüglich der seriellen Ausführung das Hauptbeurteilungskriterium für ein CC-Verfahren. Der Grund hierfür ist die Entstehungsgeschichte der in diesem Kapitel vorgestellten Verfahren. Da Transaktio-

1. Im weiteren wird die abkürzende Bezeichnung CC-Verfahren verwendet.

nen zunächst in der Batch-Verarbeitung eingesetzt wurden, gab es keine Probleme durch den parallelen Zugriff - Transaktionen wurden rein sequentiell ausgeführt. Erst durch die Einführung des OLTP-Betriebes stellte sich das durch CC-Verfahren adressierte Problem. Die einfachste Lösung wäre, die sequentielle Verarbeitung zu erzwingen und somit nur eine aktive Transaktion im System zuzulassen. Führt man stattdessen Verfahren zur Durchsetzung der Korrektheit bei gleichzeitiger Parallelverarbeitung ein, muß sichergestellt sein, daß der Durchsatz des Systems zumindest höher ist als im rein sequentiellen Fall.

Obwohl der Durchsatz als Beurteilungskriterium im Falle der ACID-Transaktionen durchaus seine Berechtigung hat (auf Grund der Isolationseigenschaft), kann dies nicht ohne weiteres auf andere “erweiterte” Modelle übertragen werden. Der Grund hierfür ist die Tatsache, daß das ACID-Modell keine kooperativen Tätigkeiten zuläßt. Das sind Verarbeitungsformen, bei denen der parallele bzw. verschachtelte Zugriff auf diesselben(!) Datenelemente gewollt und notwendig ist. Deshalb ist auch die rein sequentielle Verarbeitung der Transaktionen von vornherein ausgeschlossen und das Beurteilungskriterium nicht anwendbar. Trotzdem ist es auch für Verfahren, die Kooperation unterstützen wünschenswert, daß neben ihrer Anwendbarkeit im ACID-Fall auch das Durchsatzkriterium entsprechend angewandt werden kann.

Bei dynamischen CC-Verfahren entsteht ein weiteres Problem, welches Auswirkungen auf das Gesamtsystem haben kann: die Verklemmung (engl. *deadlock*). Da die meisten Verfahren darauf basieren, daß bei einem Konflikt zweier Transaktionen eine davon blockiert wird (die TA, deren Operation nicht zur Ausführung zugelassen wird), kann es vorkommen, daß eine zirkuläre Wartesituation auf Grund unterschiedlichen Ressourcenbedarfs vorliegt.

Ein Beispiel soll dies verdeutlichen:

1. Transaktion 1 hat eine Ressource A bereits bearbeitet und dafür exklusiv reserviert;
   Dasselbe gilt für Transaktion 2 bezüglich einer anderen Ressource B.
2. Möchte Transaktion 1 nun Ressource B bearbeiten, verweigert der Scheduler die Ausführung der Operation, da Transaktion 2 noch nicht abgeschlossen ist.
3. Möchte Transaktion 2 Ressource A bearbeiten wird Transaktion 2 ebenfalls blockiert, da Transaktion 1 noch nicht abgeschlossen ist.

Nach Schritt 3 sind somit Transaktion 1 als auch Transaktion 2 blockiert, und es gibt keine Möglichkeit diese Blockierung aufzulösen, ohne daß eine der Transaktionen abgebrochen wird (die Atomaritätseigenschaft wird vorausgesetzt).

Es gibt unterschiedliche Ansätze, Verklemmungen zu erkennen und dann durch Zurücksetzen einer der beteiligten Transaktionen aufzulösen. Die zwei Hauptansätze basieren dabei auf der Verwendung von Abhängigkeitsgraphen (ähnlich dem Serialisierbarkeitsgraphen) bzw. auf einem Zeitschrankenmechanismus (engl. *timeout*). Letzterer Ansatz hat den Nachteil, sehr anwendungsabhängig zu sein, da die maximale Dauer einer Transaktion vorab festgelegt werden muß. Andererseits ist er sehr viel einfacher zu implementieren und weniger zeitaufwendig, weshalb er in den gängigen Datenbanksystemen eingesetzt wird.

Verklemmungen bei ACID-Transaktionen sind somit vergleichsweise harmlos, wenn man den Performancegesichtspunkt unbeachtet läßt. Anders gestaltet sich dies jedoch bei geschachtelten Transaktionen. Bei diesen ist es durchaus möglich, daß zwei Sub-Transaktionen der gleichen Top-Level-Transaktion in eine Verklemmung geraten, bzw. sogar eine Sub-Transaktion mit einer Eltern-Transaktion. Einerseits kann auch dieser Konflikt durch Zurücksetzen der niedrigsten Sub-Transaktion aufgelöst werden, doch erfordert dies zunächst Information über die Transaktionsbeziehungen. Darüber hinaus ist der Effekt des Sub-Transaktions-Abbruchs auf den weiteren Verlauf der Eltern-Transaktion dem System unbekannt, wodurch der Abbruch der Top-Level-Transaktion ausgelöst werden kann.

Dieser Effekt bei geschachtelten Transaktionen ist bisher noch wenig untersucht. Allerdings stellt sich auch die Frage, ob eine Verklemmung innerhalb einer geschachtelten Transaktion nicht eigentlich ein Indikator für einen Programmierfehler darstellt.

Im weiteren Verlauf dieser Arbeit wird sowohl der Durchsatz-Aspekt als auch das Thema der Verklemmung im Kontext des ConTract-bezogenen Ansatzes nicht weiter diskutiert werden. Der Durchsatz-Aspekt wird nicht weiter betrachtet, da es sich im Falle des ConTract-Modells durchaus um einen kooperationsunterstützenden Ansatz handelt. Die genauere Untersuchung der Verklemmungsproblematik im Falle von ConTracts ist durchaus relevant, würde aber den Rahmen dieser Arbeit sprengen und stellt ein eigenständiges Forschungsthema dar.

6.1.3 Wartbarkeit

Unter dem Begriff der Wartbarkeit versteht man ein Maß für die Möglichkeit, ein Verfahren an neue Gegebenheiten anzupassen bzw. für den Aufwand, der hierzu notwendig ist. Im Falle von Transaktionsmodellen tritt dies auf, wenn zu der Menge der möglichen Operationen eines Modells neue Operationen hinzukommen. Bei ACID-Transaktionen spielt dies keine Rolle, da die Menge der Operationen per Definition auf die Lese- und Schreiboperation beschränkt ist und somit keine neuen Operationen hinzukommen können. Bei allen Modellen, die mehrstufig aufgebaut sind oder anwendungsdefinierte Operationen zulassen (wie z.B. bei den

Multi-Level-TAs oder den ConTracts) können im Gegensatz dazu jederzeit neue Operationen in das System eingebracht werden. Dabei stellt sich natürlich die Frage, inwieweit dies Auswirkungen auf das Laufzeitsystem zur Sicherstellung der Korrektheit hat.

Wie bereits bei den Mehrschichttransaktionen erwähnt, ist ein relativ hoher Aufwand notwendig, wenn die zum Einsatz kommenden Verfahren auf einem explizit zu definierendem Konfliktschema basieren, wenn also explizit (z.B. im Scheduler) festgelegt werden muß, ob eine Operation A mit einer Operation B in einem Konflikt steht, wenn beide Operationen auf das gleiche Datenelement zugreifen.

Ein solcher Aufwand ist natürlich in einem System mit vielen Operationen nicht vertretbar, weshalb auch in der Entwicklung eines CC-Verfahrens für ConTracts entsprechende Ansätze nicht berücksichtigt wurden. Statt dessen wird darauf geachtet werden, daß bei der Einbringung neuer Operationen keine Änderung des Laufzeitsystems zu erfolgen hat.

6.2 Klassische Ansätze

Unter den klassischen Ansätzen versteht man diejenigen Verfahren, die zur Abwicklung von ACID-Transaktionen im OLTP-Bereich entwickelt wurden. Die relativ lange Entwicklungsgeschichte dieser Verfahren hat dazu geführt, daß es eine Vielzahl von veröffentlichten Methoden gibt, die sich größtenteils nur in wenigen speziellen Punkten unterscheiden. Deshalb sollen in diesem Abschnitt nur die grundsätzlichen Prinzipien vorgestellt und entsprechende Hinweise auf die Literatur gegeben werden.

6.2.1 Pessimistische Verfahren

Die sogenannten *pessimistischen* Verfahren beruhen auf der Idee, Konflikte noch vor der Ausführung einer Operation zu erkennen und daraufhin geeignete Maßnahmen zu ergreifen.

6.2.1.1 Zwei-Phasen Sperrverfahren

Populärster Vertreter der pessimistischen Verfahren ist das sogenannte *Zwei-Phasen-Sperrverfahren* (engl. *two phase locking* oder *2PL*) [BHG87]. Das Prinzip des 2PL beruht darauf, daß jede Lese- und Schreiboperation einer ACID-Transaktion vor ihrer Ausführung eine Sperre für das Objekt anfordert, auf das während der Operation zugegriffen werden soll. Der Begriff der Zweiphasigkeit rührt von dem Prinzip her, daß eine Sperre erst dann wieder frei gegeben wird, wenn keine weitere Sperre mehr benötigt wird. Somit erhält man eine “Wachstumsphase” (engl. *growing phase*), während der die Anzahl an gehaltenen Sperren zunimmt und eine

"Freigabephase" (engl. *shrinking phase*) während der die Anzahl wieder abnimmt.

Es läßt sich zeigen, daß ein Scheduler auf der Basis des 2PL ausschließlich serialisierbare Historien erzeugt [BHG87]. Allerdings bleibt die Problematik der Wiederherstellbarkeit und die des kaskadierenden Zurücksetzens unberücksichtigt. Ein Beispiel soll dies verdeutlichen:

1. Eine Transaktion 1 liest ein Objekt A und schreibt ein Objekt B.
2. Anschließend gibt Transaktion 1 die Sperre für Objekt B frei.
3. Eine Transaktion 2 liest Objekt B und ändert wertabhängig von B Objekt C
4. Transaktion 1 bricht ab.[1]
5. Transaktion 2 wird erfolgreich beendet.

Nach diesem Ablauf ist der Wert von Objekt C möglicherweise falsch, da dieser auf der Basis des geänderten Wertes von B berechnet wurde. Da Transaktion 1 jedoch zurückgesetzt wurde, ist der Wert von B inzwischen auf dem alten Stand: man hat die klassische "dirty read" Situation (wie bereits in Abschnitt 5.2.3 angesprochen könnte dies durch kaskadierendes Zurücksetzen vermieden werden).

Eigentlicher Grund für dieses Problem ist die Freigabe einer Sperre auf einem geänderten Objekt ohne zu wissen, ob die Transaktion erfolgreich abschließt oder nicht. Deshalb wurde eine leichte Abwandlung des 2PL eingeführt: das *strikte 2PL*. Bei diesem Verfahren wird außer den Anforderungen des 2PL noch zusätzlich verlangt, daß Sperren ändernder Operation erst beim commit bzw. beim abort der Transaktion freigegeben werden. Für das strikte 2PL läßt sich zeigen, daß Scheduler, die das strikte 2PL einsetzen nur strikte Historien (siehe Abschnitt 5.2.3) erzeugen [BHG87].

Da bei Sperrverfahren üblicherweise eine Transaktion blockiert wird, wenn eine Sperre für ein Objekt angefordert wird auf dem bereits eine in Konflikt stehende Sperre besteht, wird das 2PL und auch das strikte 2PL üblicherweise als "verklemmungsanfällig" bezeichnet. Somit sind zusätzliche Mechanismen notwendig (siehe Abschnitt 6.1.2), um Verklemmungen erkennen und auflösen zu können.

6.2.1.2 Graphbasierte Verfahren

Mann kann für die Zwecke der CC auch die Information in Abhängigkeitsgraphen verwenden (siehe beispielsweise [Günt96]). Wie bereits beim Serialisierbarkeitsgraphen (Abschnitt 5.2.2) eingeführt, führt der Konflikt zweier Operationen zu ei-

1. Bei [BHG87] wird davon ausgegangen, daß bei commit oder abort keine Sperren angefordert werden.

ner Abhängigkeitsrelation der Transaktionen, die diese Operationen ausführen. Diese Relation läßt sich auch als gerichteter Graph auffassen, der azyklisch sein muß, um die Serialisierbarkeit zu gewährleisten. Die Grundversion der Konfliktgraphen eignet sich allerdings nicht dazu, auch die Recoverability zu gewährleisten. Hierzu sind einige Erweiterungen notwendig, die jedoch relativ einfach sind und deshalb hier nicht weiter erläutert werden.

Das grundsätzliche Kriterium zur Beurteilung, ob ein Graph eine korrekte Historie widerspiegelt oder nicht, bleibt auch bei den erweiterten Graphen die Azyklizität. Wenn also eine Operation einer Transaktion einen Zyklus in dem Graphen erzeugen würde, ist sie nicht zulässig, und es müssen entsprechende Maßnahmen ergriffen werden. Darüber hinaus ist selbst bei einer blockierenden Strategie einfach festzustellen, ob eine Verklemmung eintreten würde oder nicht. Da durch eine verklemmungsverursachende Operation auf jeden Fall auch ein Zyklus in dem Graphen entstehen würde, ist bei der Blockierung einer solchen Operation zu untersuchen, ob alle an dem Zyklus beteiligten Transaktionen ebenfalls blockiert sind. Ist dies der Fall, so liegt eine Verklemmung vor, und eine der beteiligten Transaktionen muß zurückgesetzt werden.

6.2.2 Optimistische Verfahren

Optimistische Verfahren unterscheiden sich von den pessimistischen Verfahren dadurch, daß nicht grundsätzlich versucht wird, Operationen zu verhindern, die das Korrektheitskriterium verletzen würden. Statt dessen wird hier der Ansatz verfolgt, Transaktionen bis zur prepare-Phase auszuführen und erst dann zu überprüfen, ob ein Konflikt während der Ausführung vorlag, der das Korrektheitskriterium verletzen würde. Ist dies der Fall, muß eine Transaktion zurückgesetzt werden oder das commit einer Transaktion solange verzögert werden, bis alle anderen in Konflikt stehenden Transaktionen ebenfalls ihr commit ausführen wollen.

Obwohl diese Verfahren offensichtlich verklemmungsfrei sind, hat der Realeinsatz gezeigt, daß sie im Hinblick auf den Durchsatz den pessimistischen Verfahren unterlegen sind. Deshalb wird nicht weiter auf die Details dieser Verfahren eingegangen werden. Eine kurze Diskussion der Ursache für dieses Verhalten ist jedoch angebracht.

Optimistische Verfahren führen grundsätzlich alle Operationen von Transaktionen so aus, als ob sie auf privaten Kopien der Daten ausgeführt werden würden. Bei der Überprüfungsphase muß nun (für alle Operationen) festgestellt werden, ob eventuell ein Konflikt aufgetreten ist, der die Korrektheit verletzt. Dies ist ein nicht zu vernachlässigender Aufwand, der z.B. durch ein graphbasiertes Verfahren realisiert wird. Im Falle eines Konfliktes müssen nun die Gegenoperationen für alle ändernden Operationen ausgeführt werden, was einen größeren Aufwand als für die

eigentliche Transaktion erfordert, da zunächst die Gegenaktionen ermittelt und dann angewendet werden müssen. Abgesehen von der Tatsache, daß die Gegenaktionen zumeist vom Log gelesen werden müssen und damit einen Zugriff auf eine Festplatte bedeuten können (je nach dem ob dieser Teil des Log bereits stabil geschrieben wurde), bedeutet dies auch, daß während dieser Zeit der Zugriff auf das Log von anderen Transaktionen nicht unerheblich beeinflußt wird.

Eine weitere nachteilige Folge ergibt sich aus der gleichzeitigen Ausführung der Gegenoperationen und dem Fortschreiten anderer Transaktionen. Da die im Vorwärtsablauf befindlichen Transaktionen nicht blockiert werden, wenn ein Konflikt auftritt, kann es während der Ausführung von Gegenoperationen einer Transaktion wieder zu Konflikten mit Operationen des "Normalablaufs" kommen, was zu einem Zurücksetzen führt. Gerade im Falle von sogenannten *Hot-Spots* (Datenobjekte auf die sehr häufig zugegriffen wird) stellt dies einen gewichtigen Nachteil der optimistischen Verfahren dar, da im Grenzfall keine der Transaktionen erfolgreich zu Ende geführt werden kann.

Weiterhin ist zu erwähnen, daß optimistische Verfahren anfällig für sogenannte *live-locks* sind. Im Gegensatz zu den *dead-locks* (den Verklemmungen) wird mit live-locks ein Zustand beschrieben, bei dem Transaktionen zwar aktiv sind und versuchen Operationen ausführen, jedoch keine der Transaktionen zu Ende kommt. Die Voraussetzung für das Eintreten einer live-lock Situation entspricht der des dead-lock Szenarios. Da jedoch bei optimistischen Verfahren keine Transaktion blockiert wird, entsteht ein Zyklus aus Objektzugriffen und dem Zurücksetzen der Transaktionen. Live-locks sind allerdings noch weit unangenehmer als dead-locks, da sie zum einen schwer zu entdecken sind und zum anderen sich auch noch negativ auf den Systemdurchsatz auswirken.

6.3 Semantikbasierte Ansätze

Die klassischen Ansätze wurden, wie bereits erwähnt, für die ACID-Transaktionen bzw. für das Lese-/Schreibmodell entwickelt. Sie basieren daher ausschließlich auf der Interpretation der Signatur der Operationen und nicht auf ihrer Bedeutung. Dementsprechend werden diese Ansätze auch als syntaxbasiert bezeichnet, da es möglich ist, durch eine rein syntaktische Analyse einer Historie zu entscheiden ob sie korrekt ist oder nicht.

Aus der Sicht der Anwendung werden Transaktionen aber nicht mittels Lese- und Schreiboperationen entworfen, sondern mit mächtigeren Operationen. Ein Beispiel hierfür sind die Inkrement- und die Dekrementoperation, die im Bankwesen die Zubuchung bzw. die Abbuchung auf Konten implementieren. Nun greifen zwar beide Operationen ändernd auf Datenobjekte zu und könnten somit wie Schreiboperationen aufgefaßt werden, andererseits haben sie zusätzliche Eigen-

schaften die bei der rein syntaktischen Betrachtungsweise unberücksichtigt bleiben.

Die wichtigste Eigenschaft ist die Art des Zugriffs. Im Gegensatz zu den Lese-/Schreiboperationen ist der absolute Wert eines Datenobjektes (mit Einschränkungen) relativ uninteressant für die Inkrement-/Dekrementoperation. Es werden nur Werte addiert bzw. subtrahiert. Da die Addition einfach mittels einer Subtraktion rückgängig gemacht werden kann und dies auch umgekehrt zutrifft, ohne daß der gerade aktuelle Wert des Datenobjektes berücksichtigt werden muß, können die für die Wiederherstellbarkeit gemachten Forderungen gelockert werden, was den möglichen Parallelitätsgrad erhöht. Diese semantischen Eigenschaften der Operationen werden in den sogenannten semantikbasierten Ansätzen genutzt. Auf Grund des Wissens über die Operationen kann der mögliche Parallelitätsgrad erhöht und somit insgesamt ein höherer Durchsatz des Systems erzielt werden.

Das Hauptproblem bei den semantikbasierten Ansätzen liegt in der Schwierigkeit, die Semantik der Operationen in einer Weise zu beschreiben, daß sie von einem Laufzeitsystem genutzt werden kann. Die folgenden Abschnitte stellen die prinzipiellen Methoden vor und führen einen neuen Ansatz ein, der gerade für langlaufende Transaktionen, wie im Falle der ConTracts, Vorteile gegenüber den bisherigen Verfahren zeigt.

Außer im Falle des Verfahrens, welches für die Anwendung in ConTracts entwikkelt wurde, wird auf die Darstellung der Details der anderen Verfahren verzichtet, da diese den Rahmen dieser Arbeit sprengen würde. Statt dessen wird auf die entsprechende Literatur verwiesen.

6.3.1 Frühzeitige Sperrfreigabe

Den klassischen Sperrverfahren am nächsten kommen Verfahren, bei denen Transaktionen explizit Sperren freigeben können, die nicht mehr benötigt werden. Diese Verfahren werden auch als uneigennützige Sperrverfahren (engl. *altruistic locking*) bezeichnet [SGS94].

Das Grundprinzip bei den uneigennützigen Sperren beruht darauf, daß eine Transaktion “weiß”, wann auf ein Datenobjekt zukünftig nicht mehr zugegriffen wird und somit der Zugriff anderer Transaktionen zugelassen werden kann. Aus Sicht der Transaktion kann die Sperre auf dem Datenobjekt dann freigegeben werden, womit sich die Transaktion gegenüber anderen Transaktionen “uneigennützig” verhält.

Aus Sicht der anderen Transaktionen können sich allerdings Probleme ergeben, wenn durch die Transaktion, welche die Sperre freigegeben hat, ein ändernder Zugriff erfolgte und der Abbruch der Transaktion erforderlich ist (s. Abschnitt 6.2.1.1). Da das Korrektheitskriterium der uneigennützigen Verfahren dem im

klassischen Fall entspricht, müßte für diesen Fall gewährleistet werden, daß sich der Abbruch der Transaktion auf alle abhängigen Transaktionen (Transaktionen die das Datenobjekt gelesen haben) fortpflanzt, was beispielsweise durch die Verwaltung eines Abhängigkeitsgraphen erreicht werden kann. Dieser Abhängigkeitsgraph kann dazu benutzt werden, das Commit von Transaktionen, die auf vorzeitig freigegebene Objekte zugegriffen haben zu verzögern. Damit wird gewährleistet, daß keine Transaktion ihre Änderungen persistent machen kann, solange eine Transaktion von der sie gelesen hat noch nicht beendet wurde.

Das altruistic locking ist somit ein Verfahren, welches dem Laufzeitsystem Informationen über die Semantik von Transaktionen dadurch zugänglich macht, daß Sperren explizit freigegeben werden. Einerseits wird dadurch der Parallelitätsgrad erhöht, andererseits entsteht im Falle von ändernden Zugriffen ein Zusatzaufwand für die Verwaltung von Abhängigkeiten. Außerdem muß die explizite Sperrfreigabe in den Transaktionen selbst vorgenommen werden, was im allgemeinen Fall durch den Transaktionsprogrammierer zum Programmierzeitpunkt vorgesehen werden muß. Dadurch entsteht eine Fehlerquelle, die durch den klassischen Ansatz vermieden wird.

6.3.2 Wertunabhängige, prädikatbasierte Ansätze

Der Begriff der wertunabhängigen, prädikatbasierten Ansätze bezeichnet Ansätze, die den Isolationsbedarf von Transaktionen mit Hilfe eines Prädikates beschreiben. Dabei wird allerdings nicht vorausgesetzt, daß die Werte der Datenobjekte während des Ablaufs von Transaktionen Beschränkungen unterliegen. Entsprechende Verfahren übergeben dem Laufzeitsystem ein Prädikat, welches Informationen darüber enthält, welche Datenobjekte von der Transaktion benötigt werden, d.h. welche von dem Laufzeitsystem vor dem Zugriff anderer Transaktionen zu schützen sind.

Zwei Arten der Informationsübergabe können bei diesen Verfahren unterschieden werden:

1. syntaxbasiert
2. prädikatwert-abhängig

Bei der syntaxbasierten Vorgehensweise, wie beispielsweise bei Korth [KLS90], wird ein Datenobjekt dann geschützt, wenn es syntaktisch in einem Prädikat enthalten ist. Somit ist die Evaluierung des Prädikates nicht notwendig.

Im Gegensatz dazu ermittelt der prädikatwert-abhängige Ansatz die Menge der zu schützenden Datenobjekte dadurch, daß er alle Objekte sucht, für die das Prädikat zutrifft. Dieser Ansatz ist unter dem Begriff der Prädikatsperren (engl. predicate

locking) bekannt geworden [EGL76].

Der Ansatz der Prädikatsperren wurde als Erweiterung für das klassische ACID-Modell entworfen und eignet er sich dadurch auch für langlebige Transaktionen wie im Falle von Korth et al. Der Unterschied der beiden Verfahren liegt prinzipiell nur in der Art und Weise wie die Menge der benötigten Objekte spezifiziert wird.

Das zugrunde liegende Korrektheitskriterium, die Serialisierbarkeit, ist beiden Ansätzen gemein. Allerdings ermöglicht der Ansatz der Prädikatsperren Objekte zu schützen, die zum Programmierzeitpunkt nicht vorhanden sind, da die Prädikate zur Laufzeit ausgewertet werden. Somit kann dadurch auch das sogenante Phantomproblem (s. [GrRe93]) einfach vermieden werden.

Beide Verfahren übergeben nur einen kleinen Teil der semantischen Information dem Laufzeitsystem. Nämlich nur die Menge der benötigten Objekte. Wird allerdings die zusätzliche Forderung erhoben, daß für jede Operation diese Menge anzugeben ist, kann ermittelt werden, welche Objekte nicht mehr benötigt werden. Mit dieser zusätzlichen Forderung eignen sich wertunabhängige, prädikatbasierte Ansätze auch für uneigennützige Sperrverfahren.

6.3.3 Field Calls

Mit dem englischen Begriff *field calls* wird ein Verfahren bezeichnet, das als Vorläufer der nachfolgend beschriebenen Escrow-Sperren und des Prüfe/Revalidiere-Verfahrens angesehen werden kann. Obwohl “field calls” eines der wenigen semantischen Verfahren ist, welches in einem Produkt implementiert wurde [GaKi85], hat sich kein deutscher Begriff für diese Methode durchgesetzt.

Motivation für dieses Verfahren waren die sogenannten Hot-Spots, also Datenelemente die fast von jeder Transaktion berührt und verändert werden. Da somit fast jede Transaktion ändernd auf ein solches Datenobjekt zugreift, führen Sperrverfahren dazu, daß alle Transaktionen durch dieses Datenobjekt gezwungen werden hintereinander abzulaufen. Neben der Tatsache, daß hierdurch eine fast serielle Historie erzeugt wird, entsteht durch den notwendigen Verwaltungsaufwand für die wartenden Transaktionen ein solcher Aufwand, daß der Durchsatz des Gesamtsystems sogar schlechter wird als im seriellen Fall.

Der Lösungsansatz der field calls versucht nun, die Sperrdauer auf den Hot-Spots zu minimieren. Der Mechanismus hierzu beruht darauf, daß Operationen auf Datenelemente zweigeteilt werden. Eine Operation besteht dann aus folgenden Teilen:

1. einem Prädikat,
2. einer Transformation

Anstatt nun während der Laufzeit einer Transaktion die Operation vollständig auszuführen, wird nur das Prädikat geprüft. Hierfür wird nur eine kurze Lesesperre benötigt, die nach der Evaluierung freigegeben wird. Ist das Prädikat erfüllt, wird die Operation sozusagen für die spätere Ausführung hinterlegt (beispielsweise im Log). Ist das Prädikat nicht erfüllt, muß die Transaktion zurückgesetzt werden.

Zum Commit-Zeitpunkt der Transaktion werden alle hinterlegten Operationen tatsächlich ausgeführt. Dies bedeutet, daß die Prädikate erneut ausgewertet werden. Ist die Evaluierung des Prädikates wiederum erfolgreich, werden exklusive Sperren auf den zu ändernden Objekten erworben und die Transformation (die eigentliche Operation) ausgeführt. Im Falle, daß die Evaluierung fehlschlägt, muß die Transaktion zurückgesetzt werden.

Das Ziel des Verfahrens, die Sperrzeit auf einem Hot-Spot zu minimieren, wird mit Hilfe eines Prädikates und sehr verkürzter Dauer von exklusiven Sperren erreicht. Gerade bei den bereits erwähnten Operationen aus dem Bankwesen (Zubuchung bzw. Abbuchung) zeigt dieses Verfahren große Vorteile gegenüber den einfachen Sperrmethoden. Da die Vorbedingung einer Zubuchung nur die Existenz des Kontos ist, ist die Wahrscheinlichkeit für das Fehlschlagen der Prädikatevaluierung sehr gering. Auch bei der Abbuchung muß nur sichergestellt werden, daß das Konto nicht überzogen wird, was für die meisten Transaktionen ebenfalls zutreffen dürfte. Wie einfach einsichtlich ist, sind die Prädikate recht simpel und dadurch mit wenig Aufwand zu evaluieren, so daß der Gewinn durch verkürzte Dauer von exklusiven Sperren überwiegt. Für den allgemeinen Fall ist jedoch zu klären, ob die zweimalige Evaluierung des Prädikates nicht mehr Aufwand verursacht als die eigentliche Operation.

Das Verfahren der “field calls” basiert somit darauf, dem Laufzeitsystem semantische Informationen in dem Sinne zukommen zu lassen, daß dem System mitgeteilt wird, welcher Zustand der Datenelemente für eine auszuführende Operation akzeptabel ist. Ein Zustand ist dabei durch die Existenz der Datenelemente und ihrer aktuellen Wertebelegung gekennzeichnet.

6.3.4 Escrow Sperren

Das Verfahren der “field calls” hat den Nachteil, daß das Prädikat einer Operation zum Commit-Zeitpunkt einer Transaktion nochmals geprüft werden muß. Neben der Tatsache, daß dies einen doppelten Aufwand bedeutet, hat dies auch den Effekt, daß langlaufende Transaktionen “benachteiligt” sind. Benachteiligt deswegen, da die Wahrscheinlichkeit der Verletzung des Prädikates mit der Anzahl der Zugriffe auf die Datenelemente des Prädikates und somit mit zunehmender Dauer der Transaktion wächst.

In [Reut82] wurde die Idee veröffentlicht, nach der ersten Evaluierung des Prädikates durch das Laufzeitsystem sicherzustellen, daß das Prädikat bis zum Commit-Zeitpunkt nicht mehr verletzt werden kann. In [ONei86] wurde diese Idee aufgegriffen und mit dem Begriff Escrow-Sperren (engl. *escrow locking*) bezeichnet.

Das Prinzip des Verfahrens beruht nun darauf, die Evaluierung bzw. Sicherstellung des Zutreffens von Prädikaten, auf Operationen auf die Datenobjekte abzubilden. Das heißt, daß ein Datenobjekt mit mehreren Bereichen für die Wertebelegung versehen wird:

1. einem Frei-Bereich,
2. einem oder mehreren Reservierungs-Bereichen (engl. *escrows*).

Die Evaluierung eines Prädikates wird nun darauf abgebildet, daß entsprechend dem Prädikat ein Teil des Wertes von dem Freibereich in einen Reservierungsbereich übertragen wird. Durch die Vorabprüfung des Prädikates wird dabei sichergestellt, daß die Wertebeschränkungen des Datenobjektes auf dem Freibereich nicht verletzt werden. Allerdings ist für jede Zugriffsart ein Reservierungsbereich notwendig.

Ein einfaches Beispiel soll die Vorgehensweise verdeutlichen. Wird von einem Konto ein Betrag abgebucht, lautet die Vorbedingung, daß ein bestimmter Betrag auf dem Konto vorhanden sein muß. Dieses Prädikat wird auf dem Freibereich geprüft und anschließend der Abbuchungsbetrag von dem Freibereich in den Reservierungsbereich für Abbuchungen übertragen. Der Betrag der Abbuchung wird dabei von dem Betrag im Freibereich abgezogen und zu dem Wert im Reservierungsbereich addiert. Entsprechend geschieht dies bei Zubuchungen, mit dem Unterschied, daß der Zubuchungsbetrag zu dem Wert im Zubuchungs-Reservierungsbereich addiert wird, ohne daß der Freibereich verändert wird. Erst zum Commit-Zeitpunkt wird der jeweilige Betrag aus dem Reservierungsbereich entfernt und im Falle der Zubuchung zu dem Wert des Freibereiches addiert (ein Transaktionsabbruch erfolgt analog).

Das Verfahren zeigt somit nur dann Vorteile, wenn das Zugriffsgranulat kleiner ist als das Objekt selbst. Genauer gesagt dürfen die Zugriffe nur einen Teil des Wertes des Objektes benötigen (wie im Falle der Inkrement-/Dekrementoperationen) und nicht auf den Absolutwert zugreifen. Dies ist im Falle von numerischen Datentypen auch relativ einfach implementierbar. Bei allgemeinen Datentypen stößt man jedoch schnell auf komplexe Probleme.

Die Information über die Semantik von Operationen wird bei den Escrow-Mechanismen mittels der Information über das Granulat des Zugriffes an das Laufzeitsystem übergeben. Mit dieser Information wird auch garantiert, daß die auszuführen-

den Operationen relativ zu dem aktuellen Wert ausgeführt werden können und auch beispielsweise im Recoveryfall die Gegenaktionen relativ zum aktuellen Wert ausgeführt werden.

6.3.5 Prüfe und Revalidiere

Eine weitere Abwandlung der field calls ist das Verfahren des Prüfens und Revalidierens (engl. *check/revalidate*) [PRS88]. Das Verfahren adressiert einen Nachteil der field calls, der darin besteht, daß der Wert eines Datenobjektes zum Programmierzeitpunkt einer Transaktion unbekannt ist und manche Transaktionen nur die Einschränkung besitzen, daß ein einmal von einer Operation "gesehener" Wert eines Datenobjektes bei einer späteren Operation wieder vorgefunden werden muß. Welcher Wert dabei vorliegt, spielt keine Rolle.

Die Idee des Verfahrens ist, daß bei jeder Operation dem Laufzeitsystem ein sogenannter Prüfausdruck (engl. *check expression*) übergeben werden kann, der einen eindeutigen Namen besitzt. Das Laufzeitsystem ermittelt für alle in dem Prüfausdruck enthaltenen Datenobjekte die Wertebelegung und speichert diese. Eine spätere Operation kann nun mittels des Prüfausdrucknamens den eigentlichen Prüfausdruck referenzieren. Dies veranlaßt das Laufzeitsystem, die Datenobjekte erneut zu prüfen und die Wertebelegung mit der Wertebelegung bei der ersten Überprüfung zu vergleichen (revalidate). Stimmen alle Wertebelegungen überein, ist dies das Kriterium, um die Operation auszuführen. Andernfalls muß die Transaktion zurückgesetzt werden.

Somit ist die Übergabe des Prüfausdruckes an das Laufzeitsystem eine semantische Information bezüglich des Isolationsbedarfs der Transaktion. Außerdem dient der Prüfausdruck als Indikator für die Art des Zugriffs der Transaktion.

6.4 Der Ansatz in ConTracts

Die Sicherstellung der Korrektheit in einem ConTract-verarbeitenden System unterscheidet sich in wichtigen Bereichen von den bisher vorgestellten Ansätzen. Allein die Tatsache, daß das ConTract-Modell noch weiter entwickelt wird, um Anforderungen moderner Workflowsystemen gerecht zu werden, erfordert einen äußerst flexiblen Ansatz[1]. Ein weiteres Unterscheidungsmerkmal ist die Menge an Anwendungen, die von ConTracts adressiert wird. Anstatt klassische Transaktionsanwendungen unterstützen zu wollen, oder die Problematik von Hot-Spots zu adressieren, wurde das ConTract-Modell speziell für langlaufende Abläufe entwickelt. Abläufe also, für die von vornherein ein klassischer Sperrmechanismus ungeeignet ist [Gra81a].

1. Momentan werden die dynamische Änderung zur Laufzeit und Kooperationsmechanismen entwickelt.

6.4.1 Typen von Invariantenprädikaten

Wie bereits in Kapitel 5 eingeführt wurde, unterscheidet sich das Korrektheitskriterium von ConTracts von der klassischen Serialisierbarkeit. Die Grundlage des Kriteriums sind die Invarianten, die jedoch nur konzeptionell festgelegt sind. Für eine entsprechende Umsetzung in ein Laufzeitsystem ist es deshalb dringend notwendig, die Invarianten zu konkretisieren und eine entsprechende Verwaltungsstrategie festzulegen.

Obwohl das Korrektheitskriterium für sich genommen nur eine Unterscheidung in Eingangs- und Ausgangsinvarianten vornimmt, wird bei genauerer Betrachtung des Kriteriums klar, daß eine weitere Differenzierung angebracht ist. Diese Erkenntnis beruht auf der Beobachtung, daß Eingangsinvarianten nur Referenzen auf Prädikate von Ausgangsinvarianten sind und das Korrektheitskriterium auf der Kompensations-Erweiterung der Historien definiert wird.

Betrachtet man die Notation der Invariantenklammern (siehe Abschnitt 5.5.3) im Zusammenhang mit einer Kompensations-Erweiterung einer Historie, ist einfach zu erkennen, daß alle Eingangsinvarianten, die zu einer Kompensationsfolge gehören, eine geschlossene Invariantenklammer mit den Ausgangsinvarianten bilden, die referenzierte Prädikate enthalten. Somit würde jede Operation einer ConTract-Instanz, die ein Prädikat verletzt, welches von der Eingangsinvarianten einer Kompensationsfolge einer anderen ConTract-Instanz referenziert wird, eine Verletzung des Korrektheitskriteriums verursachen, wenn sie vor der Kompensationsoperation ausgeführt werden sollte. Entsprechend muß sichergestellt werden, daß ein Prädikat, welches von einer Ausgangsinvariante etabliert wurde und von der Eingangsinvarianten einer Kompensationsfolge referenziert wird, nie verletzt wird bis die Ausführung der ConTract-Instanz beendet wurde oder die Kompensationsfolge zur Ausführung kam.

Im Gegensatz dazu können Prädikate, die nicht von einer Eingangsinvarianten einer Kompensationsfolge referenziert werden, durchaus von Operationen anderer ConTract-Instanzen verletzt werden, ohne das Korrektheitskriterium zu verletzen. Erst wenn eine der referenzierenden Eingangsinvarianten überprüft wird (die Invariantenklammer geschlossen werden soll), müssen geeignete Maßnahmen ergriffen werden.

Bei der Umsetzung des Korrektheitskriteriums in ein Laufzeitsystem können diese unterschiedlichen Eigenschaften dazu benutzt werden, um jeweils optimierte Verwaltungsstrategien zu entwerfen. Hierzu werden zunächst zwei Kategorien von Prädikaten unterschieden:

1. obligatorische Prädikate,

2. nicht-obligatorische Prädikate.

Ob ein Prädikat ein obligatorisches oder ein nicht-obligatorisches Prädikat ist, kann automatisch aus der Definition einer ConTract-Instanz ermittelt und dem Laufzeitsystem mitgeteilt werden. Das Laufzeitsystem ist dann dafür verantwortlich, entsprechende Strategien zur Sicherstellung der Korrektheit anzuwenden. Im Falle der obligatorischen Prädikate kann dies beispielsweise durch einen escrow-ähnlichen Mechanismus erfolgen, während im nicht-obligatorischen Fall eine Strategie zur Anwendung kommen kann, die dem check/revalidate-Mechanismus entspricht.

Der ursprüngliche Ansatz, mit der Definition von Invarianten eine Verwaltungsstrategie (engl. *policy*) durch den Programmierer festlegen zu lassen (s. [WäRe92]), hatte eine ähnliche Zielrichtung ohne jedoch einen Korrektheitsbegriff festzulegen. Darüber hinaus wurde die Verwaltungsstrategie nur für komplette Invarianten durch die Programmierer vorgegeben, wodurch die Gefahr besteht, daß durch Programmierfehler unzureichende Isolationseigenschaften vom Laufzeitsystem gefordert bzw. zu restriktive Zugriffsbeschränkungen etabliert werden.

6.4.2 Umsetzung auf Objektebene

In der bisherigen Einführung der Invarianten wurde keine nähere Beschreibung der Invariantenprädikate vorgenommen. Für die Sicherstellung des Korrektheitskriteriums ist es aber unumgänglich, näher auf diese Prädikate einzugehen. Ein Ziel soll dabei sein, die Grundlage für ein Verfahren zu schaffen, welches die Sicherstellung der Korrektheit ermöglicht, ohne eine zentrale Verwaltungsinstanz zu benötigen.

6.4.2.1 Zustand eines Objektes

Wie bei den Transaktionsabhängigkeiten im Falle des ACTA-Modells [ChRa90] wird bei den Invariantenprädikaten davon ausgegangen, daß es sich um prädikatenlogische Ausdrücke handelt. Im Falle der Invariantenprädikate werden diese Ausdrücke über dem Zustand von persistenten Datenobjekten definiert. Allerdings, ist der Begriff des Zustandes eines Objektes im Falle des hier vorliegenden Ansatzes nicht nur auf die Erfassung der aktuellen Wertausprägung beschränkt, sondern besteht aus folgenden Teilen:

1. dem (eindeutigen) Namen des Objektes,
2. dem aktuellen Wert des Objektes,
3. den Zugriffsbeschränkungen des Objektes.

Diese Auffassung des Zustandes eines Objektes unterscheidet sich von der üblichen Auffassung dadurch, daß die Zugriffsbeschränkungen mit Teil des Objektzustandes sind. Zugriffsbeschränkungen sollen dabei aus einer Menge von Paaren bestehen, wobei ein Paar ein Prädikat sowie eine Identifikation des Einbringers des Prädikates sein soll. Somit können die Zugriffsbeschränkungen, wie die anderen Teile des Objektzustandes, unter Transaktionsschutz modifiziert werden und unterliegen der Persistenzeigenschaft.

Somit wird deutlich warum für die Invariantenoperationen establish und check gefordert wird, daß sie unter Transaktionsschutz ausgeführt werden. Die establish-Operation einer Invarianten resultiert zunächst in einer Prüfung der in ihr enthaltenen Prädikate (lesen der Datenobjekte). Daran anschließend erfolgt die Einbringung neuer Zugriffsbeschränkungen (schreibender Zugriff), welche durch die Prädikate definiert sind. Da die Zugriffsbeschränkungen die Persistenzeigenschaft haben, sind Zugriffe auf sie ebenso transaktional zu schützen wie Zugriffe auf Datenobjekte.

Grundsätzlich ist diese Sichtweise von Zugriffsbeschränkungen nicht neu, da beispielsweise in Datenbanksystemen Wertebeschränkungen (die auch als Zugriffsbeschränkungen aufgefaßt werden können) in ähnlicher Weise gehandhabt werden. Der neue Aspekt liegt darin, daß Zugriffsbeschränkungen auch Sperren umfassen, die in bisherigen Ansätzen getrennt von den zugeordneten Objekten betrachtet wurden und auch nicht mittels Transaktionen manipulierbar waren.

6.4.2.2 Isolation auf Objektebene

Mit der eingeführten Auffassung eines Objektzustandes konkretisieren sich auch die im Ansatz von ConTracts möglichen Invariantenprädikate:

1. Prädikate, die über dem Namen eines Objekts definiert sind:
   Zugriffsbeschränkungen unabhängig von der Semantik des Objektes.
2. Prädikate, die über den Werten von Objekten definiert sind:
   Wertebereichsbeschränkungen des Objektes.

Darüber hinaus sind noch weitere Prädikate denkbar, die über den Zugriffsbeschränkungen definiert werden können. Diese spielen jedoch im Zusammenhang mit dem in dieser Arbeit vorgestellten Korrektheitsbegriff keine Rolle.

Prädikate der ersten Kategorie sind äquivalent zu den klassischen Sperren, die ebenso als Prädikate aufgefaßt werden können. So kann beispielsweise eine Schreibsperre als ein einstelliges Prädikat über dem Namen des betroffenen Objektes formuliert werden, welches zutrifft wenn der Wert des Objektes mit dem spezifizierten Namen nach der Etablierung des Prädikates nicht geändert wurde (der ei-

gentliche Wert ist dabei irrelevant).

Im Gegensatz dazu ist der Wert eines Objektes bei Prädikaten der zweiten Kategorie durchaus von Bedeutung. In diese Kategorie fallen beispielsweise Prädikate, der Art: Wert(Objekt) > 0. Ebenso unterscheiden sich die Prädikate der zweiten Kategorie von denen der ersten Kategorie dadurch, daß die funktionelle Kombination von Objektwerten möglich ist (z.B. Wert(Objekt1) + Wert(Objekt2) > 0) und diese kein Äquivalent zu einer boolschen Verknüpfung von Prädikaten ohne funktionelle Kombination hat.

6.4.2.3 Replizierte virtuelle Objekte

Durch die funktionale Verknüpfung von Objektwerten innerhalb von Prädikaten entsteht das Problem, daß die Auswertung eines Prädikates nicht mehr ausschließlich lokal bei einem Objekt vorgenommen werden kann. Es ist somit notwendig, Informationen über den Zustand anderer Objekte einzuholen, um die Auswertung vornehmen zu können. Da im Falle von ConTracts davon ausgegangen wird, daß Objekte beliebig in einem Netz verteilt sein können, kann dies ohne zusätzliche Maßnahmen ein sehr zeitraubender Vorgang werden.

Dieses Problem wird durch das Konzept der *virtuellen Objekte* gelöst [ScRe96]. Dabei wird aus einer funktionalen Verknüpfungen von Objektwerten ein virtuelles Objekt generiert, dessen Wertfunktion der funktionalen Verknüpfung entspricht. Das virtuelle Objekt existiert, solange eine Zugriffsbeschränkung mit der entsprechenden funktionalen Verknüpfung existiert und wird bei jedem Objekt generiert (repliziert), dessen Wert in der Zugriffsbeschränkung referenziert wird. Die eigentliche Zugriffsbeschränkung wird ebenfalls bei dem virtuellen Objekt hinterlegt.

Durch die virtuellen Objekte wird es möglich, die Auswertung der Prädikate von der Überprüfung der aktuellen Wertebelegung zu entkoppeln, so daß für die Aktualisierung im Falle einer Werteänderung eines der Originalobjekte, flexible Strategien zum Einsatz kommen können. Da die virtuellen Objekte in diesem Sinne einen Cache der Originalobjektwerte darstellen, gibt es hierzu wohl bekannte und optimierte Protokolle [Sten90].

6.4.3 Verwaltung der Invarianten

Die Verwaltung von Invarianten stellt die zentrale Aufgabe innerhalb eines Mechanismus zur Sicherstellung der Korrektheit dar. Durch eben diese Invariantenverwaltung als Teil des Schedulers muß garantiert werden, daß keine Verletzung des Korrektheitskriteriums zugelassen wird und entsprechende Maßnahmen im Konfliktfall ergriffen werden.

Da die Invarianten eine Konjunktion von Prädikaten darstellt, läßt sich das Ver-

waltungsproblem einfach auf die Verwaltung der Prädikate selbst abbilden. Durch die konjunktive Verknüpfung ist sichergestellt, daß wenn alle Prädikate einer Invarianten nicht verletzt werden, auch die Invariante nicht verletzt wird, bzw. wenn ein Prädikat verletzt wird, auch die Invariante verletzt ist.

6.4.3.1 Etablierung von Ausgangsinvarianten

Das Problem der Verwaltung von Ausgangsinvarianten gliedert sich in zwei Teile. Einerseits ist zu betrachten, wie und wann ein Prädikat etabliert werden kann. Andererseits sind die Seiteneffekte einer solchen Etablierung zu berücksichtigen. Unter der Etablierung eines Prädikates ist dabei folgender Vorgang zu verstehen:

- Überprüfung der Verträglichkeit mit bereits bestehenden Prädikaten.
- Überprüfung, ob das Prädikat zutrifft.
- Sicherstellung des künftigen Zutreffens.

Die Überprüfung der Verträglichkeit mit bereits bestehenden Prädikaten ist im allgemeinen ein NP-hartes Problem und sollte deshalb möglichst vermieden werden. Im Ansatz für ConTracts wird dies durch eine Grundannahme über die Arbeitsweise der Steps ermöglicht: Jeder Step hinterläßt die von ihm modifizierten Datenelemente in einem Zustand, der die bereits etablierten Invarianten nicht verletzt und der eigenen Ausgangsinvariante genügt.

Bezüglich der Berücksichtigung von bereits etablierten Invarianten entspricht dieses Verhalten dem Verhalten klassischer Transaktionen (Konsistenzeigenschaft). Die Forderung, Objekte nur so zu verändern, daß die eigene Ausgangsinvariante zutrifft, kann auf den gleichen Mechanismus abgebildet werden, obwohl die Ausgangsinvariante erst nach der Ausführung des Steps etabliert wird. Da die Etablierung der Ausgangsinvarianten unter dem Schutz der gleichen Transaktion wie die Ausführung des Steps geschieht (siehe Abschnitt 4.4.2), ist die Ausführung des Steps mit der nachfolgenden Etablierung der Ausgangsinvarianten ebenfalls atomar im transaktionalen Sinne.

Die allgemeine Verträglichkeitsprüfung der Ausgangsinvarianten mit bereits etablierten Prädikaten wird somit ersetzt durch die Überprüfung, ob der aktuelle Zustand der von einer Step-Instanz geänderten Datenobjekte zum Commit-Zeitpunkt der umgebenden Transaktion die etablierten Prädikate erfüllt. Die Etablierung der Ausgangsinvarianten kann sich dann darauf beschränken, die Sicherstellung der in ihr enthaltenen Prädikate zu veranlassen. Sollte sich bei der Überprüfung der Prädikate zum Commit-Zeitpunkt ergeben, daß eine Verletzung vorliegt, muß ein Zurücksetzen der entsprechenden Transaktion ausgelöst werden, was sich auch auf die innerhalb der Transaktion etablierten Prädikate auswirkt.

6.4.3.2 Behandlung von Eingangsinvarianten

Da Eingangsinvarianten nur Referenzen auf bereits etablierte Prädikate enthalten können, ist die Behandlung von Eingangsinvarianten weniger komplex als die der Ausgangsinvarianten. Ursprünglich waren Eingangsinvarianten in allen Fällen dazu vorgesehen, Konflikte vor der Ausführung eines Steps zu erkennen. Mit der Unterscheidung der zwei Arten von Invarianten (obligatorische und nicht-obligatorische) dienen Eingangsinvarianten von Kompensationsblöcken nun zur Erkennung von Verletzungen notwendiger Isolationsbedürfnisse, da das Korrektheitskriterium fordert, daß die referenzierten Prädikate von Ausgangsinvarianten nach ihrer Etablierung nicht verletzt werden dürfen (siehe auch Abschnitt 6.4.4). Somit besteht keine Notwendigkeit irgendwelche Aktionen bei Eingangsinvarianten von Kompensationsblöcken auszulösen[1].

Anders gestaltet sich dies bei der Behandlung von Eingangsinvarianten von anwendungsorientierten Steps. Hier muß eine Evaluierung aller referenzierten Prädikate erfolgen, da hier durchaus die Eingangsinvariante verletzt sein kann und dies ein Indikator für eine Verletzung der geforderten Isolationseigenschaften ist. Wird eine Verletzung der Eingangsinvarianten festgestellt, darf nicht weiter mit der Ausführung fortgefahren werden. Dies würde eine Verletzung des Korrektheitskriteriums darstellen. Somit muß die umgebende (Sub-)Transaktion abgebrochen werden. Anders als bei den klassischen transaktionsverarbeitenden Systemen ist es bei ConTracts jedoch vorgesehen, daß das Ausführungssystem nicht nur den Transaktionsabbruch mitgeteilt bekommt. Vielmehr wird dem Ausführungssystem zusätzlich mitgeteilt, daß die Evaluierung einer (nicht-obligatorischen) Eingangsinvariante fehlgeschlagen ist. Auf Grund dieser Information ist dann eine anwendungsorientierte Behebung des Konfliktes möglich (engl. *conflict resolution*) [WäRe92].

Obwohl Eingangsinvarianten auch zur Überprüfung genereller Ausführungsbedingungen von Steps herangezogen werden können, beschränkt sich diese Arbeit auf die Nutzung von Invarianten zur Definition von Isolationsbedürfnissen. Deshalb ist es auch nicht möglich, Eingangsinvarianten zu Beginn eines ConTracts festzulegen. Da Eingangsinvarianten über Prädikatreferenzen definiert werden und zu Beginn eines ConTracts noch keine Ausgangsinvarianten referenziert werden können, ist eine entsprechende Festlegung von Eingangsinvarianten nicht zulässig (siehe Bedingung 4-1).

6.4.4 Konfliktbehandlung

Neben der Verwaltung der Invarianten stellt die Erkennung und Behandlung von Konflikten durch die Modifikation von Objektzuständen einen Hauptaspekt eines

1. Als Konsistenzprüfung kann trotzdem eine Evaluierung erfolgen.

Mechanismus zur Sicherstellung der Korrektheit dar. Ein Konflikt ist dabei die Verletzung einer der Zugriffsbeschränkungen (Prädikate). Wie bereits im Abschnitt 6.4.3 besprochen, erfolgt die Erkennung zum Commit-Zeitpunkt der die Steps umgebenden Transaktionen, bzw. während der prepare-Phase, da im Falle von ConTracts ein Zwei-Phasen-Commit-Protokoll (kurz 2PC) [GrRe93] zur Anwendung kommt.

Innerhalb des 2PC wird allen beteiligten Objekten bzw. den Ressourcen-Verwaltern (siehe Kapitel 7), die diese Objekte verwalten, ein prepare-Aufruf übermittelt. Dies veranlaßt eine Überprüfung der Prädikate bezüglich der aktuellen Wertebelegung des Objektes, sowie die Überprüfung weiterer Konsistenzbedingungen, die auf den Datenobjekten definiert wurden. Alle beteiligten Ressourcen-Verwalter geben auf Grund der Überprüfung ein Votum ab, welches signalisiert, ob sie bereit zum Commit der Transaktion sind oder nicht. Votieren alle Ressource-Verwalter positiv, ergeht der eigentliche Commit-Aufruf an alle beteiligten RM. Votiert ein RM gegen das Commit, ergeht eine Abort-Nachricht an alle RM und die Transaktion wird abgebrochen.

Grundsätzlich sind mehrere Fälle bei der Erkennung eines Konfliktes zu behandeln. Zum einen muß die Verletzung von obligatorischen und nicht-obligatorischen Prädikaten unterschieden werden. Zum anderen ist zu berücksichtigen, daß ein Konflikt auch mit Prädikaten der eigenen ConTract-Instanz auftreten kann.

6.4.4.1 Konflikte mit obligatorischen Prädikaten

Die Einführung der sogenannten obligatorischen Prädikate berücksichtigt die Tatsache, daß das Korrektheitskriterium über der Kompensations-erweiterten Historie definiert ist. Das heißt, daß Prädikate die von einer Invarianten eines Kompensationsblockes referenziert werden nicht verletzt werden dürfen. Wird also ein obligatorisches Prädikat bezüglich der aktuellen Wertebelegung verletzt, muß die umgebende Transaktion zurückgesetzt werden, um eine Verletzung des Korrektheitskriteriums zu verhindern.

6.4.4.2 Konflikte mit nicht-obligatorischen Prädikaten

Wird ein Konflikt mit einem nicht-obligatorischen Prädikat entdeckt, stellt dies keine Verletzung des Korrektheitskriteriums dar. Die umgebende Transaktion kann somit erfolgreich abgeschlossen werden. Allerdings bedeutet dies, daß die Ausführung der ConTract-Instanz, deren Prädikat verletzt wurde, möglicherweise nicht mehr erfolgreich zu Ende geführt werden kann.

6.4.4.3 Intra-ConTract-Konflikte

Konflikte bezüglich von Prädikaten der ConTract-Instanz, welche die konfliktver-

ursachende Zustandsänderung durchgeführt hat, deuten auf einen Programmierfehler hin. Ist in diesem Fall ein obligatorisches Prädikat betroffen, bedeutet dies, daß die weitere Ausführung der ConTract-Instanz eine notwendige Voraussetzung für die eigene Kompensation verletzen würde. Somit ist keine weitere Fortführung, sondern nur die Kompensation möglich. Dies wird jedoch nicht automatisch durch das Laufzeitsystem veranlaßt.

Im Falle der Verletzung eines nicht-obligatorischen Prädikates ist die weitere Ausführung durchaus möglich, obwohl eine hohe Wahrscheinlichkeit für das Fehlschlagen einer späteren Eingangsinvarianten-Evaluierung gegeben ist. Auf Grund des Konfliktauflösungsmechanismus ist jedoch eine gesonderte Behandlung bei der Feststellung des Konfliktes nicht notwendig.

6.4.5 Gültigkeitsdauer von Invarianten

Wie in Abschnitt 6.4.2 dargestellt, werden Prädikate von Invarianten innerhalb von Transaktionen etabliert und haben somit die Persistenzeigenschaft. Somit ist es notwendig, in einem ConTract-Laufzeitsystem Transaktionen auszulösen, die diese Prädikate wieder entfernen sobald sie nicht mehr benötigt werden. Ansonsten würden immer mehr Zugriffsbeschränkungen in das System eingebracht, die schließlich die Ausführung von ConTract-Instanzen verhindern würden.

Wie aus dem Korrektheitskriterium für ConTracts ersichtlich (siehe Abschnitt 5.5.4), werden Invarianten einer ConTract-Instanz nur solange benötigt, wie die Instanz aktiv ist. Das heißt, daß alle Invarianten (und somit deren Prädikate) einer ConTract-Instanz spätestens dann gelöscht werden dürfen, wenn eine “End-Of-ConTract”-Operation von dieser ConTract-Instanz ausgeführt wird. Es stellt sich allerdings die Frage, ob dies der früheste Zeitpunkt ist, um Invarianten wieder zu entfernen.

Der eingeführte Begriff der Invariantenklammer deutet an, daß Invarianten nicht außerhalb einer solchen Klammer benötigt werden. Wäre also sichergestellt, daß zu einem Zeitpunkt keine weitere geschlossene Invariantenklammern zu der aktuellen kompensations-erweiterten Historie einer ConTract-Instanz hinzukommen kann, könnte die Invariante aus dem System entfernt werden.

Im Falle von statischen ConTract-Instanzen ist die Erkennung solcher Zeitpunkte möglich und die Entfernung von Invarianten automatisch durch das Laufzeitsystem realisierbar. “Statisch” heißt in diesem Zusammenhang, daß eine ConTract-Instanz aus einem Template erzeugt wurde und danach nicht mehr veränderbar ist (außer durch die Auslösung des Kompensationsereignisses). Ein Ansatz hierzu wäre, die Menge der Operationen von ConTracts um eine weitere Invariantenoperation delete() zu ergänzen. Diese delete-Operation ist allerdings nicht für Programmierer eines ConTract verfügbar. Stattdessen wird sie bei der Instanziierung

eines ConTracts automatisch durch das Laufzeitsystem an den Stellen eingefügt, bei denen erkannt wird, daß bezüglich einer Invariante keine weitere geschlossene Invariantenklammer auftreten kann.

Erlaubt man die Modifikation einer ConTract-Instanz zur Laufzeit, beispielsweise zur Realisierung einer flexiblen Ausnahmebehandlung, ist diese einfache Strategie nicht mehr möglich. Verhindert wird dies durch die Möglichkeit, neue Steps zu der ConTract-Instanz hinzuzufügen, die in ihren Eingangsinvarianten Prädikate von Ausgangsinvarianten der bestehenden ConTract-Instanz referenzieren und somit eine neue geschlossene Invariantenklammer bilden können. Somit ist jede Ausgangsinvariante potentieller Teil einer Invariantenklammer, selbst wenn dies nicht in einem ConTract-Template festgelegt wurde.

Berücksichtigt man also die Modifikation einer ConTract-Instanz zur Laufzeit, besteht keine Möglichkeit, Invarianten vor dem Ende der Abarbeitung der ConTract-Instanz freizugeben, ohne daß Erweiterungen der Definition eines ConTract-Templates vorgenommen werden. Denkbar wäre hierbei die Einführung von Klassifikatoren, die Invarianten explizit als frühzeitig entfernbar deklarieren (ähnlich dem altruistic locking [SGS94]) und gleichzeitig das Laufzeitsystem anweisen, eine Referenzierung durch später eingefügte Teile zu verhindern.

6.5 Vergleich der Mechanismen

Der für ConTracts gewählte Ansatz stellt eine Vereinigung einiger der vorgestellten Ansätze dar. So sind beispielsweise Sperrverfahren einfach auf die wertunabhängigen Prädikate des ConTract-Ansatzes abbildbar. Direkt übernommen wurden das Escrow-Verfahren bzw. der check/reavalidate-Machanismus im Falle der wertabhängigen Prädikate. Nicht direkt vergleichbar ist das Prinzip des altruistic lokking, da in ConTracts kein Abhängigkeitsgraph verwaltet wird. Berücksichtigt man jedoch die dynamische Erweiterung einer ConTract-Instanz, müßte ein entsprechender Mechanismus zum Einsatz kommen.

Somit stellt der hier vorgestellte Mechanismus einen äußerst flexiblen Ansatz dar, der versucht die Vorteile unterschiedlicher Ansätze in sich zu vereinigen. Darüber hinaus unterstützt das ConTract-Modell die anwendungsorientierte Auflösung von Konflikten. Das heißt, daß bei einem Konflikt bezüglich des Korrektheitskriteriums das Laufzeitsystem zwar die aktuelle Transaktion abbricht, die ConTract-Instanz jedoch über den Grund des Abbruchs informiert wird. Abhängig von der Definition im ConTract-Template können dann geeignete Maßnahmen zur Konfliktbehebung ergriffen werden. Somit kann es bei ConTracts auch nicht zu Verklemmungen durch Konflikte bezüglich Invarianten kommen, da keine Wartesituation durch das Laufzeitsystem erzwungen wird.

Mit der Flexibilität des Verfahrens entsteht allerdings gleichzeitig ein erhöhter Informationsbedarf des Laufzeitsystems. Wie bereits dargestellt, ist es notwendig, Informationen über die Semantik der Steps in Form der Invarianten zur Verfügung zu stellen. Dadurch wird der Programmierer bzw. die Programmiererin vor eine nicht einfache Aufgabe gestellt, was in gewisser Weise dem Grundsatz des Transaktionsprinzips widerspricht: Transaktionen sind deshalb so weit verbreitet, weil sie es ermöglichen, Anwendungen zu programmieren, ohne Rücksicht auf Mehrbenutzeranomalien bzw. Parallelverarbeitung nehmen zu müssen. Andererseits sieht man bei den klassischen Sperrverfahren, daß sie nur für eine sehr eingeschränkte Menge von Anwendungen geeignet sind und muß somit zwischen Programmierkomfort und Flexibilität bzw. Durchsatzpotential abwägen.

Ein weiterer Unterschied zwischen dem Ansatz in ConTracts und Ansätzen zur Abwicklung von ACID-Transaktionen steckt in dem Aufwand für die Sicherstellung der Korrektheit. Während beispielsweise Sperrverfahren relativ einfach und schnell zu implementieren sind, ist das Verfahren der ConTracts weit aus aufwendiger. Allerdings unterscheiden sich auch die Anwendungsgebiete sehr stark. Während die klassischen Ansätze für relativ kurze Transaktionen entwickelt wurden (Laufzeit < 1 Sekunde), adressieren ConTracts Anwendungen deren Laufzeit mehr als 1 Jahr betragen können. Bei dieser Laufzeit spielt der Aufwand zur Sicherstellung der Korrektheit eine untergeordnete Rolle, so daß die Vorteile, die durch die Flexibilität des Verfahrens und den hohen Grad an möglicher Parallelverarbeitung gegeben sind, überwiegen.

Gerade bei langlaufenden Anwendungen spielt darüber hinaus die Kooperationsfähigkeit eine große Rolle. Bei einer Anwendung, die sich über mehrere Monate erstreckt, wäre es undenkbar, daß ein Datenelement, welches einmal während des Ablaufs verändert wurde, bis zum Ende der Anwendung gesperrt bleibt. Statt dessen ist es notwendig, die Einschränkungen so gering wie möglich halten zu können, um so eine kooperative Bearbeitung (mehrer ConTract-Instanzen) zu ermöglichen. Durch die Einführung bzw. spezielle Verwaltung der nicht-obligatorischen Prädikate ist bei dem Verfahren von ConTracts außerdem die Voraussetzung für die vorübergehende Verletzung von Zugriffsbeschränkungen geschaffen. Dies erhöht den möglichen Kooperationsgrad noch weiter.

Ein weiteres Unterscheidungsmerkmal ist die Erweiterung des Objektzustandes um die Zugriffsbeschränkungen. Die Atomarität bei Änderungen von Zugriffsbeschränkungen ist gewährleistet, da sie mittels ACID-Transaktionen erfolgt. Aus dem selben Grund sind die Änderungen an Zugriffsbeschränkungen auch dauerhaft. Somit wird auch die Persistenz des Zustandes einer ConTract-Instanz durch das Verfahren unterstützt.

7 Integrationsaspekte

Mit der Definition eines Korrektheitsbegriffs und der Einführung eines Mechanismus zur Sicherstellung desselben ergeben sich Seiteneffekte auf andere Teile des ConTract-Modells. Besonders zu erwähnen ist dabei das Programmiermodell von ConTracts. Da in dem Programmiermodell die Vorgehensweise zur Entwicklung von ConTract-Templates konzeptionell festgelegt ist, muß eine Ergänzung vorgenommen werden, die auch die Definition von Invarianten in das Programmiermodell integriert.

Ein weiterer Integrationsaspekt ergibt sich aus der Tatsache, daß neben der rein konzeptionellen Entwicklung des ConTract-Modells auch eine Architektur entwikkelt wurde, die die Umsetzung der Ansätze in ein Laufzeitsystem ermöglicht. Gleichzeitig wurde eine prototypische Implementierung dieser Architektur vorgenommen, um die Umsetzbarkeit der Architektur zu verifizieren [RSW92][1]. Wie die vorgestellten Konzepte in die Architektur bzw. die prototypische Implementierung integriert werden können, soll in den folgenden Abschnitten diskutiert werden.

Wie bereits erwähnt, unterliegt das ConTract-Modell einer ständigen Weiterentwicklung. Parallel zu dieser Arbeit wurde bereits an Erweiterungen gearbeitet, die demnächst Teil des ConTract-Modells und der Architektur werden sollen. Wie sich die Ergebnisse dieser Arbeit mit den Neuerungen vertragen, ist ebenfalls ein Aspekt der bei einer Integrationsdiskussion berücksichtigt werden muß und wird in Abschnitt 7.3 angesprochen.

7.1 Auswirkungen auf das Programmiermodell

ConTracts unterscheiden sich von anderen erweiterten Transaktionsmodellen, die sich für langlebige Abläufe eignen, unter anderem dadurch, daß die Vorgehensweise zur Definition eines ConTracts in dem Modell selbst festgelegt ist: Durch das sogenannte Programmiermodell. Ein Aspekt dieses Programmiermodells ist die Trennung der Programmierung von Steps von der Programmierung der ConTract-Templates (zweistufiges Programmiermodell). Das heißt, daß ein Programmierer eines Steps nichts über den ConTract wissen muß, in dem der Step später verwendet werden soll. Umgekehrt weiß die Programmiererin eines ConTract-Templates nichts über die Implementierungsdetails eines Steps und kennt nur dessen Schnittstelle.

In den folgenden Unterabschnitten wird vorgestellt werden, wie die Definition von

1. APRICOTS: “A Prototype Implementation of a ConTract System”

Invarianten in das Programmiermodell integriert werden kann. Dabei wird davon ausgegangen, daß dieses erweiterte Programmiermodell in eine spezielle Entwicklungsumgebung für die Definition von ConTract-Templates umgesetzt wird, um so die optimale Unterstützung bei der Programmierung zu gewährleisten.

7.1.1 Grundprobleme

Aus der strikten Trennung der Programmierung von Steps und ConTract-Templates ergeben sich Probleme im Hinblick auf die Definition von Invarianten. Invarianten werden grundsätzlich in einem ConTract-Template spezifiziert. Allerdings legen sie fest, welche Voraussetzung erfüllt sein müssen, damit ein Step ausgeführt werden kann (Eingangs-Invariante), bzw. welcher Zustand von Objekten nach der Ausführung eines Steps von Relevanz für "später" auszuführende Steps ist (Ausgangs-Invariante).

Zunächst stellt sich dabei das Problem, daß eine ConTract-Programmiererin die Objekte zu identifizieren hat, deren Zustandsraum beschränkt werden muß, um die Ausführbarkeit des Steps zu garantieren. Als nächstes müssen die Zustandsraumbeschränkungen als Prädikate formuliert und in Invarianten zusammengefaßt werden. Bereits die Identifikation der Objekte ist mit dem bisherigen Programmiermodell nicht möglich, da die Programmiererin des ConTract-Template keine Informationen über die interne Arbeitsweise von Steps hat. Somit muß zunächst eine Erweiterung dahingehend erfolgen, daß weitere Informationen über Steps bei der Programmierung von ConTract-Templates verfügbar sind.

Darüber hinaus wurde in ConTracts ursprünglich davon ausgegangen, daß die Datenobjekte nicht direkt von Steps modifizierbar sind, sondern gemäß dem verteilten Transaktionsmodell der X/Open Organisation [XOP93] in sogenannten Ressourcen-Verwaltern (engl. *Resource Manager* oder kurz *RM*) zusammengefaßt und nur über diese ansprechbar sind. Mit der Verfügbarkeit neuerer Umgebungen für die verteilte Ausführung von Anwendungen, wie beispielsweise CORBA, DCOM und JavaBeans, ergeben sich teilweise Unterschiede zum X/Open Modell. Da die CORBA-Spezifikation einerseits die notwendigen Mechanismen zur verteilten Verarbeitung geschachtelter Transaktionen bereitstellt und andererseits auf Ansätze wie DCOM oder JavaBeans abbildbar ist, geht der hier beschriebene Ansatz davon aus, daß Objekte über den in CORBA spezifizierten Mechanismus adressiert werden. Für die globale Adressierbarkeit eines Datenobjektes bedeutet dies, daß zunächst der zugehörige Ressourcen-Verwalter (ein transaktionales CORBA-Objekt) adressiert werden muß, um bei diesem den Zugriff auf das entsprechende Datenobjekt veranlassen zu können.

Ein weiteres Problem ergibt sich aus der Tatsache, daß Datenobjekte in unterschiedlichen Arten von Ressourcen-Verwaltern zusammengefaßt sind. Hierdurch

unterscheidet sich die Adressierung von Datenobjekten innerhalb der RM. Beispielsweise werden Tupel oder Attribute einer relationalen Datenbank im allgemeinen assoziativ adressiert (z.B. mittels SQL über einen Primärschlüssel). Im Gegensatz dazu werden Dateiobjekte in einem Dateisystemen direkt mittels eines Pfades und eines Namens adressiert.

Zusammengefaßt ergeben sich also drei grundsätzliche Aufgaben bei der Einführung des invariantenbasierten Korrektheitsbegriffes:

1. Bereitstellung von Information über die Ausführungsvoraussetzungen von Steps.
2. Globale Adressierbarkeit von RM.
3. Flexible Adressierbarkeit von Objekten innerhalb von RM.

Während Punkt eins eine Aufgabe darstellt, die durch das Programmiermodell angegangen werden kann, hängen die Punkte zwei und drei stark von der konkreten Umsetzung der Konzepte in eine Definitionssprache für ConTract-Templates bzw. der gewählten Ausführungsumgebung ab. Da im Rahmen dieser Arbeit davon ausgegangen wird, daß Punkt zwei durch die CORBA-Spezifikation festgelegt ist und darüber hinaus nur die grundsätzlichen Konzepte und die Auswirkungen auf die Architektur besprochen werden sollen, wird deshalb im weiteren nur auf Punkt eins näher eingegangen.

7.1.2 Step-Programmierung

Der bisherige Ansatz von ConTracts sieht vor, für Steps eine Signatur zu definieren, die bei der Definition eines ConTract-Templates verwendet werden kann. Dabei wird unter der Signatur eines Steps folgende Information verstanden:

1. Name des Steps
2. Schnittstellendefinition des Steps.

Wie bei einem verteilten objekt-orientierten System nach CORBA [Sieg96] dient die Signatur eines Steps dazu, einem Laufzeitsystem genügend Information zur Verfügung zu stellen, um einen Aufruf einer Methode zu generieren. Im Unterschied zur Adressierung bei CORBA identifiziert der Name eines Steps dabei die gewünschte Semantik der Methode anstatt eine Methode in einem Objekt zu identifizieren[1]. Somit stellt der Name eines Steps eine logische Adresse einer Methode

1. Neuere Erweiterungen von CORBA beinhalten einen sogenannten “broker service” mit ähnlicher Semantik.

dar, die durch das Laufzeitsystem aufgelöst werden muß. Die Schnittstellendefinition stellt wie bei CORBA die korrekte Übergabe von Parametern sicher und hilft darüber hinaus bei der Identifikation der gewünschten Methode.

Da die Signatur eines Steps die einzige Information ist, die von einem Step-Programmierer über den Step zur Verfügung gestellt wird, ist es mit der Einführung der invariantenbasierten Korrektheit notwendig, diese Signatur zu erweitern. Genauer gesagt muß ein Step Programmierer in der Signatur mögliche Ein- bzw. Ausgangs-Invarianten definieren, die von einer Programmiererin von ConTract-Templates benutzt werden kann.

Hierfür werden sogenannte *Invarianten-Templates* oder *Invarianten-Schablonen* eingeführt. Dies sind Prädikate, die mittels Referenzen auf Datenobjekte und sogenannten *Platzhaltern* definiert werden. Die Referenzen auf Datenobjekte ermöglichen es, die notwendige Adressierungsinformation für den Zugriff auf Datenobjekte dem Laufzeitsystem zur Verfügung zu stellen. Die Platzhalter stellen wie bei einer Schnittstellendefinition typisierte Variablen dar, die auf der Ebene des ConTract-Templates entweder mit Kontextvariablen assoziiert oder durch konstante Werte ersetzt werden können.

An dem Anwendungsbeispiel aus Kapitel 2 kann das Konzept verdeutlicht werden. Der universitäre Urlaubsantrag enthält in seinem zweiten Schritt die Überprüfung der Zulässigkeit des Antrages, was normalerweise bedeutet, daß überprüft wird, ob der Antragstellende noch genügend Urlaubstage zur Verfügung hat. Ein mögliches Ausgangsinvarianten-Template dieses Steps könnte somit formuliert werden als:

$$\text{Ref:AnzahlFreieUrlaubstage(\%Antragsteller\%)} \gtrsim \text{\%beantragteAnzahl\%}.$$

Hierbei steht “Ref:AnzahlFreieUrlaubstage(Antragsteller)” für eine Referenz auf ein Datenobjekt, während “%Antragssteller%” als auch %beantragteAnzahl% Platzhalter sind, die erst zur Laufzeit durch Werte ersetzt werden (siehe 7.1.3). Der Operator $\gtrsim$ ist ebenfalls erläuterungsbedürftig. Bei dem referenzierten Datenobjekt handelt es sich um ein sogenanntes kumulatives Objekt. Das heißt, daß der absolute Wert des Objektes nicht von Interesse ist. Statt dessen soll nur gewährleistet werden, daß die untere Schranke des Objektes nicht verletzt wird, wenn die Anzahl beantragter Urlaubstage abgezogen wird. Anstatt also eine absolute Wertebeschränkung zu fordern spezifiziert der $\gtrsim$ Operator nur den notwendigen Anteil der benötigt wird. Beispielsweise könnte der Antragsteller vier Urlaubstage beantragen. Dies würde in der folgenden Invariante ausgedrückt:

$$\text{Ref:AnzahlFreieUrlaubstage(\%Antragsteller\%)} \gtrsim 4$$

Bei der Umsetzung in eine Wertbeschränkung auf Datenobjektebene wird nun der rechte Teil der Bedingung auf die bestehende Wertebeschränkung “AnzahlFreie-

Urlaubstage > 0" addiert und somit die konjunktive Verknüpfung der bestehenden Wertbeschränkung und der Invariante auf eine funktionale Verknüpfung mit der rechten Seite der bestehenden Wertebeschränkung abgebildet. Als Resultat entsteht eine neue Wertebeschränkung "AnzahlFreieUrlaubstage > 4".

Wenn mehrere solcher Prädikate bei dem Objekt etabliert werden, bedeutet dies, daß die konjunktive Verknüpfung dieser Prädikate einem einzigen Prädikat entspricht. Dieses kumulative Prädikat würde dann fordern, daß der Wert des Datenobjektes größer oder gleich der Summe der geforderten Werte in den Einzelprädikaten plus der unteren Schranke sein muß.

Im Schritt 5 des Anwendungsbeispiels findet die Änderung der Urlaubskartei statt. Dies bedeutet, daß die Anzahl der noch zur Verfügung stehenden Urlaubstage um die Anzahl der beantragten Tage reduziert wird. Dies setzt jedoch voraus, daß überhaupt genügend Urlaubstage zur Verfügung stehen. Ein Eingangsinvarianten-Template, welches dies sicherstellt, entspricht dem Ausgangsinvarianten-Template für Step 2.

7.1.3 ConTract-Template-Programmierung

Bei der Programmierung eines ConTract-Templates stellt sich nun die Aufgabe, die mit der Signatur der Steps bereitgestellten Invarianten-Templates in konkrete Ein- und Ausgangsinvarianten umzusetzen. Dies erfordert zunächst, daß die Platzhalter der Invarianten-Templates, mit Kontextvariablen verknüpft werden.

In dem angeführten Beispiel für das Ausgangsinvarianten-Template von Step 2 heißt dies, daß die Platzhalter %Antragsteller%" und %beantragteAnzahl% durch Kontextvariablen des gleichen Typs ersetzt werden. Dies entspricht dem Vorgang bei der Verknüpfung von Kontextvariablen mit Schnittstellenparametern von Steps. Darüber hinaus werden Referenzen von Datenobjekten ebenfalls dem Kontext hinzugefügt. Somit können Datenobjektreferenzen wie gewöhnliche Variablen des Kontext behandelt werden.

In einem weiteren Schritt muß jedes Prädikat der Eingangsinvarianten-Templates durch Referenzen auf Ausgangsinvarianten ersetzt werden. In dem Beispiel aus Abschnitt 7.1.2 ist dies einfach, da das Eingangsinvarianten-Template keine konjunktive Verknüpfung mehrerer Prädikate darstellt und außerdem dem Ausgangsinvarianten-Template eines Vorgängersteps entspricht. Im allgemeinen Fall stellt sich die Aufgabe, alle Prädikate der Eingansinvariante in Ausgangsinvariante bereits eingefügter Steps zu finden und zu diesen Referenzen zu generieren. Dabei können folgende Fälle auftreten:

1. Genau ein Prädikat einer Ausgangsinvarianten entspricht einem Prädikat der Eingangsinvarianten.

2. Zu einem Prädikat einer Eingangsinvariante existiert kein passendes Prädikat einer Ausgangsinvariante.
3. Es existieren mehrere Prädikat von Ausgangsinvarianten, die einem Prädikat einer Eingansginvariante entsprechen.

Während Fall 1 die Voraussetzung für die automatische Generierung von Prädikatreferenzen erfüllt, ist die automatische Generierung im Fall zwei und drei nicht möglich. Bei mehreren passenden Prädikaten ist durch den Programmierer festzulegen auf welches Prädikat referenziert werden soll. Im Falle, daß kein Prädikat einer Ausgangsinvarianten zu einem Prädikat einer Eingangsinvarianten paßt, kann nicht zugelassen werden, daß der Step eingefügt wird, da dies das Wohlgeformtheitskriterium aus Definition 4-20 verletzen würde. Der Programmierer muß daher aufgefordert werden, einen weiteren Step so einzufügen, daß dessen Ausgangsinvariante von der Eingangsinvariante des gerade abgewiesenen Steps referenziert werden kann und damit Fall 1 eintritt.

Sowohl die Überprüfung der Typkompatibilität bei der Zuweisung von Kontextvariablen zu Platzhaltern als auch die Prüfung der Übereinstimmung von Prädikaten kann durch eine Programmierumgebung weitgehend automatisiert werden. Somit können zum einen Programmierfehler vermieden und zum anderen eine große Zeitersparnis erzielt werden.

7.2 Architekturaspekte

Das ConTract-Modell wurde entworfen, um die Basis für ein Laufzeitsystem zur Abwicklung langlebiger Abläufe bereitzustellen. Deshalb begleitet die Entwicklung des Modells auch stets die Entwicklung einer Architektur einer entsprechenden Laufzeitumgebung. Dieser Abschnitt beschäftigt sich mit den Auswirkungen, die die Einführung des Korrektheitsbegriffes bzw. seine Sicherstellung mit sich bringen.

Unter einer Architektur eines Laufzeitsystems für ConTracts soll eine logische Aufteilung der notwendigen Funktionalitäten in sogenannte Komponenten verstanden werden. Dabei werden für eine Komponente sowohl die Funktionalität selbst als auch die notwendigen Schnittstellen festgelegt. Wie nun in einem realen Laufzeitsystem die Komponenten in (Betriebssystem-)Prozessen realisiert werden, oder wie die Kommunikation zwischen Komponenten erfolgt, ist nicht Teil der Architektur und wird im folgenden auch nicht diskutiert.

7.2.1 Bisherige Architektur

Bereits in [RSW92] wurde eine Architektur für ein ConTract-verarbeitendes Sy-

stem vorgestellt. Prinzipiell wurden in dieser Architektur fünf funktionale Komponenten unterschieden:

1. ConTract Manager (CM)
2. ConTract Processing Monitor (CPM)
3. Step Computation Server (SCS)
4. Resource Manager (RM)
5. Transaction Manager (TM)

Der ConTract Manager koordiniert die Ausführung einer ConTract-Instanz. Er hat somit eine Schnittstelle zu dem Anwender, der über diese Schnittstelle die Ausführung beeinflussen kann (start, stop, usw.). Außerdem verwaltet der CM die Konsistenz- bzw. Isolationsinformation und ist aus Sicht des Transaktionsverwalters Auftraggeber für transaktionale Dienste (begin transaction, end transaction, usw.).

Der ConTract Processing Monitor war dafür vorgesehen, die prozeßorientierte Verwaltung eines ConTract-Systems abzuwickeln. Das heißt, daß Komponenten mittels des CPM in das System eingebracht, entfernt und überwacht werden können. Mit der Verfügbarkeit von CORBA kann diese Aufgabe jedoch von Standardkomponenten übernommen werden, so daß ein Dienst wie der CPM inzwischen nicht mehr als funktionale Komponente eines ConTract-Systems angesehen wird, sondern eher als ein Bestandteil der Basisdienste.

Die Step Computation Server kapseln die Funktionalität von Steps. Ursprünglich war für sie eine standardisierte Schnittstelle ähnlich einem RPC zur Kommunikation mit dem CM vorgesehen. Wiederum hat die Verfügbarkeit von CORBA diesen Schritt überflüssig gemacht, so daß nur noch gefordert wird, daß neben den Schnittstellen für Steps transaktionale Schnittstellen (commit, prepare, abort) verfügbar sind.

Eine weitere Schnittstelle ergibt sich aus der Tatsache, daß Steps zur Implementierung ihrer Funktionalität mit Resource Managern kommunizieren müssen. Wie bereits erwähnt, verwalten die RM persistente Datenobjekte. Sie entsprachen ursprünglich der DTP-Spezifikation der X/Open [XOP93]. Inzwischen wird auch für die RM gefordert, daß sie transaktionale Objekte im Sinne von CORBA sind, da die X/Open-Spezifikation keine geschachtelten Transaktionen berücksichtigt.

Grundsätzlich nicht ConTract-spezifisch, jedoch unbedingt notwendig für ein ConTract-verarbeitendes System ist ein Transaction Manager, der die Abwicklung transaktionaler Protokolle übernimmt. Der TM soll der OTS-Spezifikation (Object Transaction Service) von CORBA entsprechen.

Abbildung 7-1 gibt einen Überblick über die Komponenten und ihr Zusammenspiel:.

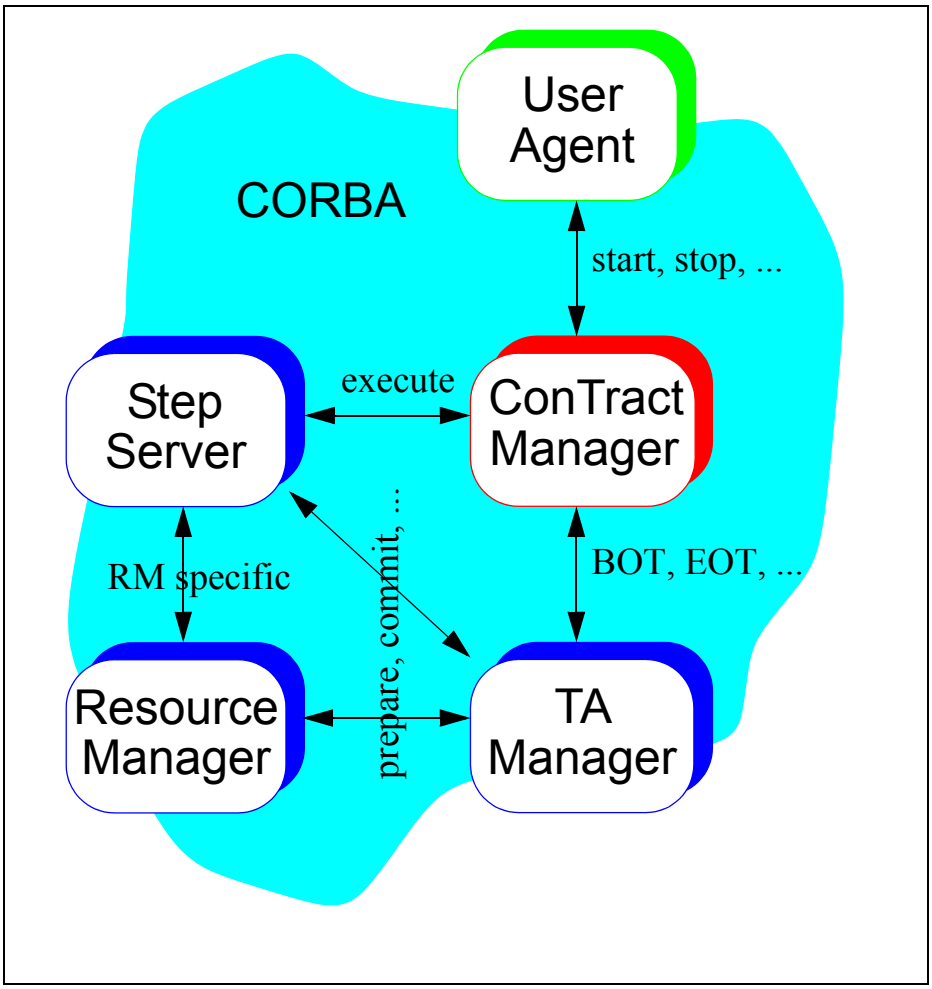

Abbildung 7-1: Architektur eines ConTract-verarbeitenden Systems

7.2.2 Autonomie

Der vorgestellte Mechanismus zur Sicherstellung der Korrektheit erfordert die Verwaltung von Invarianten sowie die Realisierung der Zugriffsbeschränkungen, die durch die Invarianten gefordert werden. Diese Funktionalität kann prinzipiell auf drei verschiedene Arten in einer Architektur berücksichtigt werden:

1. Durch Einführung einer zentralen Instanz zur Verwaltung der Invarianten, bzw. der in Invarianten enthaltenen Prädikate.
2. Durch Verteilung der Verwaltungsfunktionalität auf mehrere Komponenten, die im Verbund die notwendigen Aufgaben erledigen.
3. Durch einen hybriden Ansatz, der mittels einer zentralen Komponente arbeitet, jedoch einen Teil auf mehrere Komponenten verteilt.

Da die Datenobjekte in den RM verwaltet werden, stellt jeder Zugriff auf ein Datenobjekt auch einen Zugriff auf einen RM dar. Bezüglich der drei oben eingeführten Ansätze ist somit zu beurteilen, wie die Funktionalität der RM verändert oder erweitert werden muß, damit der entsprechende Ansatz realisiert werden kann.

Im allgemeinen kann davon ausgegangen werden, daß die RM nicht neu entwickelt werden, sondern daß existierende Komponenten (wie z.B. ein Datenbanksystem) in ein ConTract-System integriert werden. Dabei stellt sich das sogenannte *Autonomieproblem*, welches am Beispiel von Datenbanksystemen erläutert werden soll. Datenbanksysteme enthalten bereits Mechanismen zur Verwaltung und Sicherstellung von Zugriffsbeschränkungen. Diese sind üblicherweise nicht erweiterbar und nur über eine relativ restriktive Schnittstelle zugänglich[1]. Betrachtet man nun bereits existierende Anwendungen dieser Datenbanksysteme, wäre es völlig inakzeptabel, diese Anwendungen ändern zu müssen, weil das Datenbanksystem in einem ConTract-verarbeitenden System integriert wird. Somit sind das Datenbanksystem selbst und auch die darauf basierenden Anwendungen als autonom anzusehen, und diese Autonomie ist beim Entwurf einer Architektur zu berücksichtigen.

Alle Ansätze, die nur die Etablierung einer zentralen Instanz vorsehen, verlangen massive Eingriffe in den Zugriff von existierenden Anwendungen auf Datenobjekte. Da Zugriffsbeschränkungen, die durch Invarianten gefordert werden, auch für Zugriffe gelten, welche von Anwendungen ausgelöst werden, die nicht zu einem ConTract-verarbeitenden System gehören, müssen diese Zugriffe an die zentrale Instanz umgeleitet werden. Dort wird zunächst geprüft, ob der Zugriff zulässig ist und erst dann kann der Zugriff an den RM weitergeleitet werden. Dies ist bei vielen Anwendungen nicht ohne weiteres möglich, da die Zugriffe direkt im Anwendungscode selbst verankert sind.

Aufgrund der Autonomieprobleme ist somit die Einführung einer zentralen Instanz zur Verwaltung der Invarianten nicht sinnvoll bzw. nicht realisierbar.

7.2.3 Fehlertoleranz

Fehlertoleranz ist eine der wichtigsten Eigenschaften eines ConTract-verarbeitenden Systems. Zum einen wird garantiert, daß eine einmal gestartete ConTract-Instanz in endlicher Zeit abgearbeitet wird (sofern sie überhaupt terminieren kann). Zum anderen soll auch gewährleistet sein, daß Einkomponentenfehler eine weitere Bearbeitung von ConTract-Instanzen nicht ausschließt. Unter Einkomponentenfehler ist dabei der Ausfall einer der logischen Komponenten gemeint.

Natürlich verhindert der Ausfall einer Komponente, die nur einmal im System vorhanden ist, die weitere Bearbeitung der ConTract-Instanzen, die diese Komponente

1. Zum Schutz der Anwendungen ist dies auch unbedingt notwendig.

aktuell benutzen. Betrachtet man nun den hybriden Architekturansatz (siehe Abschnitt 7.2.2), so enthält dieser eine solche Komponente. Beim Ausfall dieser Komponente ist eine weitere Bearbeitung von ConTract-Instanzen nicht möglich, da keine Invarianten mehr in das System eingebracht, bzw. überprüft werden können.

Grundsätzlich kann dieses Problem durch zusätzliche Hard- und Softwareredundanz gelöst werden. Beispielsweise kann die zentrale Komponente auf einem hoch zuverlässigen System angesiedelt werden, so daß ein Ausfall äußerst unwahrscheinlich wird. Allerdings sind die Kosten für eine solche Lösung erheblich und erfordern eine spezielle Hardwarekonfiguration.

Aus diesem Grund sind zentrale Komponenten in einer Architektur für ein ConTract-verarbeitendes System zu vermeiden. Darüber hinaus ist darauf zu achten, daß beim Ausfall einer Komponente nur die ConTract-Instanzen betroffen sind, die diese Komponente gerade benötigen. Alle anderen Instanzen sollten von dem Ausfall unbeeinflußt bleiben. Somit ist auch die mehrfache Einbringung von Komponenten der gleichen Funktionalität sinnvoll. Arbeiten beispielsweise mehrere ConTract-Manager in einem System, sollte der Ausfall eines der CM die anderen CM nicht beeinflussen.

7.2.4 Verteilungsaspekte

Durch die diskutierten Nachteile der Ansätze mit zentralen Komponenten verbleibt als einzig sinnvolle Alternative ein Architekturansatz, der eine verteilte Verwaltung von Invarianten bzw. Prädikaten vorsieht. Allerdings ist zu entscheiden, in welchen Komponenten die Verwaltung angesiedelt wird.

Wie bereits in Abschnitt 7.2.2 diskutiert, ist es auf Grund der Autonomie der RM sinnvoll, die Verwaltung von Zugriffsbeschränkungen direkt bei den RM anzusiedeln. Die Zugriffsbeschränkungen resultieren dabei aus den Prädikaten, die in Invarianten konjunktiv verknüpft sind.

Darüber hinaus ist noch die Verwaltung der Invarianten als Ganzes zu berücksichtigen. So ist die Etablierung, die Entfernung sowie die Überprüfung von Invarianten als Funktionalität in einer Komponente zu realisieren. Dabei kommen drei Komponenten zur Auswahl:

1. ConTract-Manager,
2. Step-Server,
3. eine neu zu definierende Komponente.

Der Ansatz, die Invarianten im ConTract-Manager selbst zu verwalten, hat den

Vorteil, daß der stabile Speicher, der in einem ConTract-Manager vorhanden sein muß, auch zur persistenten Speicherung der Invarianten-Verwaltungsinformation genutzt werden kann. Da der Ausfall eines CM durchaus die Ausführung der gerade aktiven ConTract-Instanzen beeinflussen darf, stellt dieser Ansatz auch aus Sicht der Fehlertoleranz kein Problem dar. Allerdings würde bei diesem Ansatz dem ConTract-Manager die Aufgabe übertragen, die Prädikate bei den zuständigen RM zu etablieren. Somit müßten die CM die Referenzen in Invarianten interpretieren, um die Prädikate entsprechend weiterleiten zu können.

Die Alternative, die Invariantenverwaltung bei den Step-Servern anzusiedeln, hat zunächst den Vorteil, daß diese Zuordnung mit dem Programmiermodell übereinstimmt. Da Invarianten-Templates von Step-Programmierinnen für die Steps definiert werden, ist hier die notwendige Information für den Zugriff auf die RM vorhanden, die in Invarianten referenziert Objekte verwalten. Somit wäre es nicht notwendig, daß Referenzen auch von ConTract-Mangern interpretiert werden müßten und könnten als “opak”[1] angesehen werden. Darüber hinaus ist auch bei den Step-Servern persistenter Speicher für die Abwicklung des transaktionalen Protokolls vorhanden, der auch für die Speicherung der Invarianteninformation genutzt werden kann.

Eine neu zu definierende Komponente, die ausschließlich die Verwaltung von Invarianten zur Aufgabe hat, erscheint wenig sinnvoll. Zum einen wäre das Problem, daß Referenzen aufgelöst werden müssen, ebenso gegeben wie bei einer Ansiedlung der Funktionalität in den CM. Zum anderen ist der Aufwand zur Realisierung eines persistenten und transaktionalen Speichers nicht unerheblich, so daß sich dieser Aufwand für die relativ geringe Funktionalität kaum lohnt.

Die Abwägung der Vor- und Nachteile der vorgestellten Ansätze läßt nur den Schluß zu, die Invariantenverwaltung bei den Step-Servern anzusiedeln. Somit wird die bisher vorliegende Architektur nicht um weitere Komponenten, sondern nur um zusätzliche Funktionalität bzw. Schnittstellen erweitert. Die Funktionalität der Step-Server wird um die Invariantenverwaltung erweitert. Damit ergibt sich eine Erweiterung der Schnittstelle zwischen Step-Server und ConTract-Manager für den Aufruf folgender Funktionalitäten:

1. Etabliere Invariante.
2. Überprüfe Invariante.
3. Entferne Invariante.
4. End of ConTract (Entfernen aller Invarianten einer ConTract-Instanz).

1. Unter einem opaken Datentyp versteht man einen Typ dessen Struktur unbekannt ist.

Die RM implementieren die Funktionalität zur Verwaltung und Sicherstellung der Zugriffsbeschränkungen. Dies resultiert ebenfalls in einer Erweiterung der Schnittstelle zwischen Step-Server und Ressourcen-Verwalter um den Aufruf der folgenden Funktionalitäten:

1. Etabliere Prädikat.
2. Überprüfe Prädikat.
3. Entferne Prädikat.
4. Lege virtuelles Objekt an.
5. Entferne virtuelles Objekt.

Die neu eingeführten Schnittstellen für RM hängen eng mit den Schnittstellen der Step-Server zusammen. Wird die Etablierung einer Invarianten bei einem Step-Server angefordert (Step-Server Funktion zur Etablierung einer Invarianten), resultiert dies in einer Zerlegung der Invarianten in Einzelprädikate. Für diese Prädikate wird zunächst geprüft, ob sie funktionale Kombinationen enthalten. Ist dies der Fall, legt der Step-Server bei allen betroffenen RM die notwendigen virtuellen Objekte an, die die funktionale Kombination kapseln (RM Funktion zum Anlegen eines virtuellen Objektes). Existieren bei einem RM bereits diese virtuellen Objekte, muß nur ein Referenzzähler inkrementiert werden. Als letzter Schritt werden durch den Step-Server alle Prädikate bei den RM etabliert (RM Funktion zum Etablieren eiines Prädikates).

Wird eine Invariante entfernt (Step-Server Funktion zum Entfernen von Invarianten bzw. End of Contract), entfernt der Step-Server alle Prädikate bei den RM, die er für die Invariante etabliert hat (RM Funktion zum Entfernen eines Prädikates). Anschließend werden auch die virtuellen Objekte entfernt, bzw. der Referenzzähler dekrementiert (RM Funktion zum Entfernen eines virtuellen Objektes). Die Überprüfung einer Invariante (Step-Server Funktion zum Überprüfen einer Invariante) resultiert in der Überprüfung der Prädikate die für die Invariante bei den RM etabliert wurden (RM Funktion zum Überprüfen eines Prädikates). Eine Überprüfung einer Invariante ohne vorherige Etablierung kann nicht auftreten. Dies wird durch die Wohlgeformtheitsbedingung (Definition 4-20) und die Grundannahme aus Abschnitt 6.4.3.1 gewährleistet.

Die Erweiterungen der Architektur führen neue ConTract-spezifische Schnittstellen ein, die von Step-Servern bzw. RM implementiert werden müssen, um die korrekte Abwicklung einer ConTract-Instanz zugewährleisten. Darüber hinaus basiert der Ansatz darauf, daß sich die Funktionalität zur Verwaltung und Sicherstellung der Zugriffsbeschränkungen in RM so realisieren lassen, daß auch

bestehende Anwendungen die Korrektheit nicht korrumpieren können. Beispielsweise bedeutet dies für ein relationales DB-System, daß die Beschränkungen für Prädikate sich in sogenannten “Constraints” auf der DB-Ebene ausdrücken.

7.3 Erweiterbarkeit

Wie bereits erwähnt, befindet sich das ConTract-Modell in einer laufenden Entwicklung. Da das Modell inzwischen speziell auf den Workflowbereich ausgerichtet wurde, sind die Anforderungen aus diesem Gebiet die Motivation für ständige Erweiterungen. Momentan sind es drei Bereiche, welche Gegenstand laufender Arbeiten sind. Inwieweit der hier eingeführte Korrektheitsbegriff und das CC-Verfahren auf diese Neuerungen übertragbar und welche Anpassungen notwendig sind, soll in den folgen drei Unterabschnitten kurz diskutiert werden.

7.3.1 Flexible Kompensation

Der in dieser Arbeit verwendete Kompensationsbegriff beruht auf dem Prinzip, daß zu jedem Step ein vordefinierter Kompensationsstep (hierunter fallen auch leere Kompensationssteps) zur Verfügung steht. Verallgemeinert man diese Auffassung dahingehend, daß ein Step nicht nur durch einen Kompensationsstep kompensiert werden kann, sondern durch einen transaktionalen Block beliebiger Art, ist die bisherige Definition der kompensationserweiterten Historie nicht ausreichend.

Da im Falle einer solchen *erweiterten Kompensation* mehrere Anwendungssteps zur Ausführung kommen können, muß für alle diese Steps die Ausführbarkeit garantiert sein, um die Kompensierbarkeit einer ConTract-Instanz zu gewährleisten. Hierfür reicht die Forderung nach dem Zutreffen der Eingangsinvariante eines Steps nicht mehr aus. Es müssen alle Eingangsinvarianten der Steps in dem Kompensationsblock erfüllt sein, um diese Bedingung zu gewährleisten.

Die naive Erweiterung des Korrektheitskriteriums würde somit die kompensationserweiterte Historie so definieren, daß alle Steps in einem Kompensationsblock auch in der erweiterten Historie definiert sind. Dies würde jedoch dem statischen CC-Ansatz für diesen Kompensationsblock entsprechen. Wie bereits diskutiert, ist dieser Ansatz aber nur für eine sehr eingeschränkte Menge von Anwendungen geeignet, so daß die Untersuchung von Alternativen notwendig ist.

Eine Alternative stellt eine Zusatzforderung bezüglich der Struktur einer ConTract-Instanz dar. Das Korrektheitskriterium kann einfach dahingehend erweitert werden, daß gefordert wird, daß die Eingangsinvarianten der ersten Steps eines Kompensationsblockes erfüllt sein müssen und zusätzlich garantiert sein muß, daß die Steps eines Kompensationsblockes das Zutreffen der Eingangsinvarianten von Nachfolgesteps garantieren.

Ist beispielsweise für einen Step S ein Kompensationsblock als Sequenz zweier Steps A und B definiert, so müßte gefordert werden, daß die Ausgangsinvariante von S die Verletzung der Eingangsinvariante von A ausschließt. Ebenso gilt dies für die Ausgangsinvariante von A bezüglich der Eingangsinvariante von B. Mit diesen Bedingungen wäre gewährleistet, daß sowohl A als auch B (im Anschluß an A) ausgeführt werden können.

Die Definition eines Kompensationsblockes (Definition 4-17) könnte einfach dahingehend erweitert werden, ohne das darauf aufbauende Korrektheitskriterium ändern zu müssen. Grundsätzlich ist somit eine Anpassung des hier vorgestellten Korrektheitskriteriums im Hinblick auf einen erweiterten Kompensationsmechanismus relativ einfach möglich. Da die Arbeiten auf diesem Gebiet jedoch längst nicht abgeschlossen sind, kann eine tiefergehende Diskussion hier nicht erfolgen.

7.3.2 Dynamische Abläufe

Kurz angerissen wurde bereits das Gebiet der dynamischen Abläufe. Trotzdem soll an dieser Stelle noch einmal erläutert werden, was darunter zu verstehen ist. Der bisherige Ansatz in ConTracts sieht vor, daß ConTract-Instanzen aus einem ConTract-Template erzeugt werden und zur Laufzeit weder Teile (beispielsweise neue Steps oder Pfade) hinzu- noch wegkommen können. Im Workflowbereich hat es sich jedoch gezeigt, daß beispielsweise für Ausnahmebehandlungen die Erweiterung einer Ablaufdefinition zur Laufzeit unbedingt notwendig ist.

Grundsätzlich stellt die Erweiterbarkeit für das hier vorgestellte Korrektheitskriterium kein Problem dar, solange bereits ausgeführte Teile nicht entfernt werden und die Kompensationsblöcke nicht verändert werden können. Allerdings ergeben sich für das CC-Verfahren andere Voraussetzungen. Beim bisherigen Modell ist automatisch zu erkennen, wann kein Step mehr durchlaufen werden kann, der eine Referenz auf eine bestimmte Ausgangsinvariante in seiner Eingangsinvariante besitzt. Ist kein weiterer Step vorhanden, dessen Eingangsinvariante eine bestimmte Ausgangsinvariante referenziert, dürfte diese Ausgangsinvariante aus dem System entfernt werden. Ist es allerdings möglich, daß Steps zur Laufzeit eingefügt werden können, ist diese Entfernung nicht mehr unbedingt zulässig.

Enthält die Eingangsinvariante des neu eingefügten Steps eine Referenz auf eine Ausgangsinvariante, die bereits aus dem System entfernt wurde, ist diese Referenz ungültig. Drei Ansätze können verfolgt werden, um dieses Problem zu vermeiden:

1. Erneute Etablierung aller referenzierten Ausgangsinvarianten zum Zeitpunkt der Erweiterung.
2. Grundsätzliche Verhinderung der Entfernung von Invarianten.

3. Explizite Unterscheidung in löschbare Invarianten und solche, die auch von Erweiterungen referenziert werden dürfen.

Sieht man von den eingangs erwähnten Beschränkungen für die Dynamik von ConTract-Instanzen ab, haben die Änderungen zur Laufzeit keinen Einfluß auf das Korrektheitskriterium, da die kompensationserweiterte Historie nicht verändert wird. Allerdings gilt es zu untersuchen, welche der erwähnten Strategien für die Freigabe von Invarianten die meisten Vorteile hat.

Sollen auch Kompensationsblöcke Gegenstand von Änderungen zur Laufzeit sein, können sich schwerwiegende Probleme mit dem hier eingeführten Korrektheitskriterium ergeben, da der eingeführte Begriff der Invariantenklammern nicht mehr ausreicht die Ausführbarkeit der Kompensationssteps zu garantieren. In wieweit hierfür Zusätze oder Änderungen erforderlich sind, ist Gegenstand zukünftiger Forschungsarbeiten.

7.3.3 Nicht-transaktionale Steps

Nicht-transaktionale Steps stellen seit der ersten Einführung des ConTract-Modells ein Thema dar, welches allerdings nur in Teilbereichen angegangen wurde [Schm93]. Unter nicht-transaktionalen Steps sind solche Steps zu verstehen, die nicht die ACI(D)-Eigenschaften erfüllen. Gerade die Atomaritätseigenschaft beeinflußt das Korrektheitskriterium sehr stark, da Steps entweder als erfolgreich abgearbeitet oder als zurückgesetzt in der Historie erscheinen. Teilweise abgearbeitete Steps sind hierbei nur sehr schwer zu berücksichtigen.

In einigen Ansätzen wird versucht, das Zurücksetzen eines nicht-atomaren Steps durch eine Ergänzung des Ablaufs um weitere Steps zu implementieren. Die Erweiterung dient dann quasi dazu, die Überbleibsel eines nicht-transaktionalen Steps zu beseitigen und dadurch, ähnlich dem Kompensationsmechanismus, eine semantische Atomarität zu gewährleisten.

Welcher formale Zustand in welchen Fällen von einer solchen semantischen Atomarität hergestellt werden kann, ist bisher immer noch Gegenstand von Forschungsarbeiten. Darüber hinaus ist noch ungeklärt, ob die Zusatzaktivitäten selbst wieder Gegenstand einer eventuellen Kompensation sein müssen oder nicht.

Zusammengefaßt läßt sich zu den nicht-transaktionalen Steps sagen, daß es äußerst schwierig ist, diese in ein Korrektheitskriterium einzubeziehen. Bevor eine formale Beschreibung der Fehlersemantik solcher nicht-transaktionalen Steps gefunden ist, wird auch kaum ein entsprechendes Korrektheitskriterium festgelegt werden können. Denkbar wäre hierbei wiederum ein prädikativer Ansatz, welcher dazu dient den Zustand nach dem “Zurücksetzen” formal zu beschreiben.

Obwohl momentan noch keine konkret anwendbaren Lösungen zur Verfügung ste-

hen, stellen nicht-transaktionale Aktivitäten insbesondere im Workflowbereich einen äußerst wichtigen Bereich dar, der unbedingt untersucht werden muß.

8 Diskussion und Ausblick

8.1 Korrektheit und langlebige Abläufe

Der aus dem Datenbankbereich bekannt gewordenen Begriff der (ACID-)Transaktion stellt ein Programmierprimitiv dar, welches Anwendungsprogrammierer stark entlastet. Sowohl durch das definierte Verhalten im Fehlerfall, dem Schutz vor Anomalien im parallelen Mehrbenutzerbetrieb, als auch durch die einfache Handhabung haben sich Transaktionen im Bereich der Datenbanksysteme durchgesetzt. Allerdings sind die Eigenschaften der ACID-Transaktionen nicht auf alle Anwendungen übertragbar. Insbesondere bei langdauernden Abläufen, wie sie beispielsweise bei Workflow-Anwendungen die Regel sind, ist festzustellen, daß einige Eigenschaften von Nachteil sind.

Der Grundgedanke der Transaktionen, Aufgaben wie die Fehlerbehandlung und die Isolation von anderen Abläufen, von den Anwendungen in ein Laufzeitsystem zu verlagern, ist jedoch auch auf langlebige Ausführungen übertragbar. Das Ziel dabei ist, die Konsistenz der an den Anwendungen beteiligten Systeme zu garantieren, ohne daß in jeder Anwendung entsprechende Mechanismen programmiert werden müssen.

Vor diesem Hintergrund wurde das ConTract-Modell entworfen. Dabei wurde besonders darauf geachtet, daß für Anwendungen als Ganzes durch die Implementierung als ConTract automatisch ein definiertes Fehlerverhalten garantiert werden kann. Obwohl auch die Problematik der Parallelverarbeitung von ConTracts zumindest teilweise berücksichtigt wurde, geschah dies bisher auf rein informelle Art und Weise. Diese Lücke wurde in dieser Arbeit angegangen.

8.1.1 Korrektheit - warum?

Ein Hauptgrund für den Erfolg des klassischen Transaktionsmodells ist der definierte Korrektheits- bzw. Konsistenzbegriff. Durch die formale Festlegung der Eigenschaften einer Transaktion ist es zum einen möglich, Mechanismen zu entwikkeln, die die Implementierung dieser Eigenschaften in einem Laufzeitsystem ermöglichen, und zum anderen wird erst dadurch beurteilbar, ob ein Ablauf korrekt ist oder nicht.

Somit stellt die Entwicklung des Korrektheitsbegriffes für ConTracts einen notwendigen Schritt dar, um für Anwendungen als Ganzes bestimmte Eigenschaften zusichern zu können. Erst durch den Korrektheitsbegriff wird festgelegt, welcher Zustand der Daten zulässig ist, und welche Zustände nicht erreicht werden dürfen. Auf dieser Basis können dann Mechanismen entwickelt werden, welche die Errei-

chung unzulässiger Zustände verhindern. Da dies vollständig unabhängig von einer Anwendung ist, ergibt sich somit auch die Möglichkeit, diese Mechanismen in einem Laufzeitsystem zur Verfügung zu stellen.

Der in dieser Arbeit vorgestellte Ansatz zur Sicherstellung der Korrektheit in einem ConTract-verarbeitenden System stellt einen solchen anwendungsunabhängigen Mechanismus dar. Bei der Entwicklung wurde streng darauf geachtet, daß sich der Ansatz einfach in ein Laufzeitsystem integrieren läßt. Allerdings wurde auf die Vorstellung einer konkreten Implementierung, wie z.B. die Festlegung konkreter Sprachkonstrukte, verzichtet. Zum einen würde dies den Rahmen dieser Arbeit sprengen, und zum anderen sind die vorgestellten Mechanismen auch zur Integration in bereits bestehende Systeme geeignet, so daß die konkrete Umsetzung stark von dem jeweiligen Umfeld abhängt.

8.1.2 Aufwand versus Nutzen

Wie bereits erwähnt, ist ein maßgebliches Beurteilungskriterium im Bereich der CC-Verfahren der Durchsatz eines Systems. Dabei liegt klar auf der Hand, daß die Reduktion des Aufwandes für die Sicherstellung der Korrektheit eine Erhöhung des Durchsatzes nach sich zieht. Das hier vorgestellte Verfahren ist deutlich aufwendiger als der klassische transaktionale Ansatz und erscheint somit wenig geeignet. Bei diesem Vergleich würde man allerdings außer acht lassen, daß die Anwendungsgebiete der beiden Ansätze grundsätzlich verschieden sind.

Während klassische Transaktionen für kurze Aktionen entworfen wurden, stellen ConTracts ein Modell für langlebige Abläufe dar. Vergleicht man dabei die Laufzeiten typischer Anwendungen, erhält man eine Relation, die sich in einem Bereich von mehr als drei Zehnerpotenzen bewegt. Gerade bei Workflowanwendungen, die darauf ausgerichtet sind, langlaufende Anwendungen mit elektronischen Mitteln zu unterstützen, ist der eigentliche Aufwand für Berechnungen üblicherweise verschwindend klein. Ebenso verhält es sich bezüglich des Aufwandes für das hier vorgestellte Verfahren.

Somit ist festzuhalten, daß es bei langlebigen Anwendungen weniger darauf ankommt, möglichst wenig Rechenzeit neben der eigentlichen Berechnung zu verbrauchen. Vielmehr ist es wichtig, einen hohen Grad an Fehlertoleranz zu erreichen und auf keinen Fall inkonsistente Datenbestände zu erzeugen[1]. Obwohl die in dieser Arbeit vorgestellten Ansätze in einem prototypisches Laufzeitsystem implementiert wurden, ist auf die konkrete Messung des Durchsatzes verzichtet worden. Dies ist gerechtfertigt, da konkrete oder standardisierte Anwendungen als Vergleichsbasis fehlen.

1. Dies ist von besonderer Bedeutung bei den typischen Anwender von Workflow-Systemen wie Banken und Versicherungen.

8.1.3 Flexibilität versus einfache Verwendung

Im Gegensatz zum klassischen Transaktionsansatz versucht das ConTract-Modell, Anwendungen als Ganzes mit den bereits erwähnten Eigenschaften auszustatten. Hierbei wird von der Art der Anwendung abstrahiert, so daß eine generelle Beschränkung der Semantik nicht vorgenommen werden soll. Beispielsweise ist bei klassischen Transaktionen grundsätzlich vorausgesetzt, daß keine Kooperation zwischen Transaktionen bestehen kann. Ebenso wurde ursprünglich gefordert, daß Transaktionen im Fehlerfall zurückgesetzt werden. Im Falle allgemeiner Anwendungen ist dies jedoch nicht möglich bzw. nicht gewünscht.

Werden einerseits weniger Eigenschaften durch das Laufzeitsystem vorgegeben, ergibt sich andererseits die Notwendigkeit, die Information über die gewünschten Eigenschaften von der Programmiererin zu erfragen. Dies ist notwendig, um dem Laufzeitsystem genügend Information übergeben zu können, damit die Maßnamen zur Sicherstellung der Korrektheit des Gesamtsystems nicht zu restriktiv bzw. ungenügend sind. Somit wird durch das ConTract-Modell weit mehr vom Programmierer gefordert als beim klassischen Transaktions-Modell. Man erkauft sich sozusagen die Flexibilität mit einem erhöhten Programmieraufwand.

8.2 Offene Probleme

Obwohl das ConTract-Modell selbst wie auch die in dieser Arbeit vorgestellten anderen Ansätze darauf ausgerichtet sind, für eine Vielzahl von Anwendungen anwendbar zu sein, gibt es noch einige Eigenschaften von Anwendungen, die bisher nicht adressiert werden. Nachfolgend werden einige dieser Punkte kurz erwähnt werden. Allerdings steht die Untersuchung der Eigenschaften von Anwendungen, wie zum Beispiel im Workflowbereich, noch an ihrem Anfangspunkt, so daß kein Anspruch auf Vollständigkeit erhoben wird.

8.2.1 Modifikationen zur Laufzeit

Die Modifikation von Anwendungen zur Laufzeit wurde bereits bei der Vorstellung des Mechanismus zur Sicherstellung der Korrektheit diskutiert. Dabei beschränkte sich die Untersuchung auf die Betrachtung der Auswirkungen auf die vorgestellten Ansätze.

Erweitert man diese Sichtweise, stellt sich die Frage, welchem Korrektheitsbegriff eigentlich eine solche Modifikation unterliegt. Der eingeführte Begriff der Wohlgeformtheit kann hier nicht mehr angewandt werden, da dessen Voraussetzungen nicht mehr erfüllt sind. Somit ist es notwendig, einen Korrektheitsbegriff für die Modifikation selbst zu entwickeln, so daß wiederum ein Laufzeitsystem entwickelt werden kann, welches diese Korrektheit sicherstellt.

Es sind momentan Untersuchungen im Gange, die zunächst prüfen, welche Arten der Modifikationen überhaupt möglich sind. Erst dann wird der nächste Schritt erfolgen können, Korrektheitskriterien zu entwerfen, die auch Modifikationen berücksichtigen.

8.2.2 Unterstützung der Programmierung

Der Aufwand für die Programmierung im Hinblick auf das vorgestellte Invariantenkonzept ist erheblich. Außerdem stellen die Invarianten ein relativ hohes Risiko im Hinblick auf Programmierfehler dar[1]. Deshalb ist zu überlegen, ob diese Programmiertätigkeit nicht zusätzlich durch weitere Mechanismen unterstützt werden kann.

Ein Ansatz, den es zu untersuchen gilt, ist die automatische Generierung von Invarianten. Dabei kann diese Generierung entweder statisch zum Programmierzeitpunkt von Steps als auch dynamisch zu deren Laufzeit erfolgen. Die dynamische Variante ist sicherlich die flexiblere und umfassendere, erfordert allerdings auch mehr Aufwand. Die in dieser Arbeit verwendete Methodik der direkten Referenzierung von Ausgangsinvarianten bei der Definition von Eingangsinvarianten ist dann sicherlich zu überarbeiten.

8.2.3 Der Kompensationsbegriff

Der Begriff der Kompensation ist seit seiner Einführung äußerst unterschiedlich interpretiert worden. Auch in der Literatur findet man vielfältige Definitionen, die sicherlich nicht zu einer Klärung beitragen. Die im ConTract-Modell verwendete Auffassung orientiert sich weitgehend an der anwendungsorientierten Auslegung des Begriffes, wodurch sich eine immer komplexer werdende Semantik ergibt.

Bisher ist diese Semantik entweder nur informell beschrieben oder auf ein einfaches formales Modell “kondensiert” worden. Aktuelle Arbeiten beschäftigen sich mit der Formulierung eines umfassenderen formalen Modells der Kompensation. Dies hat sicherlich auch Auswirkungen auf den Korrektheitsbegriff. Inwieweit dies eine Änderung der vorgestellten Ansätze erfordert, läßt sich aber aus heutiger Sicht noch nicht abschätzen.

1. Dies ist allerdings weit aus geringer als das Risiko bei der Implementierung der Konsistenzsicherung in den Anwendungen.

9 Literatur

[AVA94a] *A Unified Approach to Concurrency Control and Transaction Recovery*
G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H. Schek, G. Weikum
erschienen in
Proc. of the 4th Intern. Conf. on Extending Database Technology (EDBT), 1994.

[AVA94b] *Unifying Concurrency Control and Recovery of Transactions*
G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H. Schek, G. Weikum
Information Systems, 1994.

[AAE93] *A Unified Implementation of Concurrency Control and Recovery*
G. Alonso, D. Agrawal, A. El Abbadi
Technischer Bericht des Dept. of Computer Science, Universitity of California at Santa Barbara, TRCS93-19, 1993.

[BHG87] *Concurrency Control and Recovery in Database Systems*
P.A. Bernstein, V. Hadzilacos, N. Goodman
Addison Wesley Pub., 1987.

[BHL92] *Objektbanken für Experten*
R. Bayer, T. Härder, P. Lockemann (Hrsg.)
Springer-Verlag, 1992.

[Brau87] *Petri nets: central models and their properties; advances in Petri nets*
W. Brauer
Springer Verlag, 1987

[ChRa90] *ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior*
P. K. Chrysanthis, K. Ramamritham
erschienen in
Proc. of the ACM SIGMOD International Conference on Management of Data, 1990 .

[ChRa92] *ACTA: The Saga continues*
P. K. Chrysanthis, K. Ramamritham
Kapitel 10 in
Database Transaction Model for Advanced Applications,
A.K. Elmagarnid,
Morgan Kaufmann Publishers, 1992.

[ChRa94] *Synthesis of Extended Transaction Models using ACTA*
P. K. Chrysanthis, K. Ramamritham
erschienen in
ACM Transactions on Data Base Systems (TODS), 1994.

[DHL90] *Organizing Long-Running Activities with Triggers and Transactions*
U. Dayal, M. Hsu, R. Ladin
erschienen in
Proc. ACM SIGMOD Intern. Conf. on Management of Data, 1990.

[DHL91] *A Transaction Model for Long-Running Activities*
U. Dayal, M. Hsu, R. Ladin
erschienen in
Proc. 17. Conference on Very Large Data Bases (VLDB), S. 113-122, 1991.

[Dav78] *Data processing spheres of control*
C. T. Davies Jr.
erschienen in:
IBM Systems Journal, Vol. 17 No. 2 S. 179-198, 1978.

[Daya88] *Active Database Management Systems*
U. Dayal
erschienen in
Proc. 3. Int'l Conference on Data and Knowledge Bases, S. 150-169, 1988.

[EGL76] *The Notions of Consistency and Predicate Locks in a Database System*
K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger
erschienen in
Communications of the ACM, Vol. 19 No.11, S. 624-633, 1976.

[Elm92] *Database Transaction Models for Advanced Applications*
A.K. Elmargarmid (Hrsg.)
Morgan Kaufmann Publishers, 1992.

[GGK90] *Coordinating Multi-Transaction Activities*
H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem
Technischer Bericht: Princeton University, Department of Computer Science
CS-TR-247-90, 1990.

[GGK91a] *Coordinating Activities Through Extended Sagas: A Summary*
H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem
erschienen in
Proc. 36. IEEE Computer Society Intern. Conf. (CompCon), S. 568-573, 1991.

[GGK91b] *Modelling Long-Running Activities as Nested Sagas*
H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem
erschienen in
IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 14, No. 1,
S. 14-18, 1991.

[GaKi85] *Varieties of Concurrency Control in IMS/VS FastPath*
D. Gawlick, D. Kinkade
erschienen in
IEEE Database Engineering, Vol. 8, No. 2, S. 3-10, 1985.

[GrRe93] *Transaction Processing: Concepts and Techniques*
J. Gray, A. Reuter
Morgan Kaufmann Publishers, 1993.

[GaSa87] *SAGAS*
Hector Garcia-Molina, Kenneth Salem
erschienen in
Proc. ACM SIGMOD Intern. Conf. on Management of Data, S. 249-259 1987

[Gra81a] T*he Transaction Concept: Virtues and Limitations*
J.N. Gray
erschienen in
Proc. 7th Int . Conf. on VLDB, S. 144-154 1981

[Gra81b] *A Straw man Analysis of Probability of Waiting and Deadlock*
J.N. Gray
IBM Research Report No. RJ 3066, 1981

[Günt96] *Ein Basisdienst für die zuverlässige Abwicklung langdauernder Aktivitäten*
R. Günthör
Dissertation an der Fakultät Informatik der Universität Stuttgart, 1996.

[HäRe83] *Principles of Transaction-Oriented Database Recovery*
T. Härder, A. Reuter
erschienen in
ACM Computing Surveys, Vol. 13 No.2, S. 155-166, 1983

[Jab95] *Workflow-Management-Systeme*
S. Jablonski
Thomson Publishing, 1995

[KLS90] *A Formal Approach to Recovery by Compensating Transactions*
H. F. Korth, E. Levy, A. Silberschatz
erschienen in
Proc. 16th Intern. Conf. on Very Large Databases, S. 95-106 1990

[Klei91] *Advance Rule Driven Transaction Management*
J. Klein
erschienen in
Proc. of IEEE Computer Society Int'l Conference (CompCon) Spring 1991, Digest of Papers, S. 562-567, 1991.

[KoSp88] *Formal Model of Correctness without Serializability*
H. F. Korth, G. D. Speegle
erschienen in
Proc. ACM SIGMOD Intern. Conf. on Management of Data, 1988.

[Kuma96] *Performance of Concurrency ConTrol Mechanisms in Centralized Database Systems*
V. Kumar (Hrsg.)
Prentice Hall, 1996

[Leym95] *Supporting Business Transactions Via Partial Backward Recovery in Workflow* Management Systems
F. Leymann
erschienen in
Tagungsband Datenbanksysteme in Büro, Technik und Wissenschaft, GI Fachtagung Dresden, 1995

[Lom92] *MLR: A Recovery Method for Multi-level Systems*
D. B. Lomet
erschienen in
Proc. ACM SIGMOD Intern. Conf. on Management of Data, S. 185-194, 1992.

[MoLi83] *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactio*ns
C. Mohan , Lindsay, B.
erschienen in
Proc. ACM/SIGOPS Symposium on Principles of Distributed Computing, 1983.

[Moss85] *Nested Transactions: An Approach to Reliable Distributed Computing*
J. E. B. Moss
MIT Press, 1985.

[OMG96] *CORBAservices*
Object Management Group
1996.

[OSI92] *OSI TP Model; OSI TP Service*
Open Systems Interconnection - Distributed Transaction Processing
ISO/IEC JTC 1/SC 21 N, 1992.

[ONei86] *The Escrow Transactional Method*
P. E. O'Neil
erschienen in
ACM Transactions on Database Systems, Vol. 11 No.4,S. 405-430 1986.

[PRS88] *High Contention in a Stock Trading Database: A Case Study*
P. Peinl, A. Reuter, H. Sammer
erschienen in
Proc. ACM SIGMOD Intern. Conf. on Management of Data S. 260-268, 1988.

[Papa86] *The Theory of Database Concurrency Control*
C. Papadimitriou
Computer Science Press, 1986.

[RSS97] *ConTracts Revisited*
A. Reuter, K. Schneider, F. Schwenkreis
erschienen in
Advanced Transaction Models and Architectures,
S. Jajodia (Hrsg.),
Kluwer Pub. 1997.

[RSW92] *Zuverlässige Abwicklung großer verteilter Anwendungen mit ConTracts - Architektur einer Prototypimplementierung*
A. Reuter, F. Schwenkreis, H. Wächter
erschienen in
Objektbanken für Experten
R. Bayer, T. Härder, P. Lockemann
Springer-Verlag 1992.

[ReSw95] *ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems*
A. Reuter, F. Schwenkreis
erschienen in
Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society), Vol. 18, No. 1, 1995

[Reut82] *Concurrency on High-Traffic Data Elements*
A. Reuter
erschienen in:
Proc. ACM Symposium on Principles of Database Systems (PODS), 1982.

[Reut89] *ConTracts: A Means for Extending Control Beyond Transaction Boundaries*
A. Reuter
erschienen in
Proc. 3. International Workshop on High Performance Transaction Systems (HPTS), 1989

[Reut96] *An Analytic Model of Transaction Interference*
A. Reuter
erschienen in
Performance of Concurrency Control Mechanisms in Centralized Database Systems
V. Kumar
Prentice Hall, 1996.

[SGS94] *Altruistic Locking*
K. Salem, H. Garcia-Molina, J. Shands
erschienen in
ACM Transactions on Database Systems Vol.19 No.1 1994

[SWY93] *Towards a Unified Theory of Concurrency Control and Recovery*
H.J. Schek, G. Weikum, H. Ye
erschienen in
Proc. ACM Sysmposium ob Principles of Database Systems (PODS), 1993.

[ScRe96] *Synchronizing Long-Lived Computations*
F. Schwenkreis, A. Reuter
erschienen in
Performance of Concurrency Control Mechanisms in Centralized Database Systems
V. Kumar
Prentice Hall, 1996.

[Schm93] *Transaktionen in der Fertigung*
U. Schmidt
erschienen in
Tagungsband GI-Fachtagung Datenbanken in Büro Technik und Wissenschaft, 1993.

[Schw93b] *APRICOTS - A Prototype Implementation of a ConTract System: Management of the ConTrol Flow and the Communication System*
F. Schwenkreis
erschienen in
Proc. of the 12th Symposium on Reliable Distributed Systems (SRDS), IEEE Computer Society Press, 1993.

[Schw94] *A Formal Approach to Synchronize Long-lived Computations*
F. Schwenkreis
erschienen in
Proc. of the 5th Australasian Conference on Information Systems, 1994.

[Schw95] *APRICOTS - a workflow programming environment*
F. Schwenkreis
erschienen in
Proc. 6th High Performance Transaction Workshop (HPTS), 1995.

[Seif96] *Zuverlässige Workflowbearbeitung auf der Basis von OTS*
J. Seifert
Fakultät Informatik der Universität Stuttgart, Diplomarbeit Nr. 1404, 1996.

[Sieg96] *CORBA Fundamentals and Programming*
J. Siegel
John Wiley & Sons, Inc., 1996

[Sten90] *A Survey of Cache Coherence Schemes for Multiprocessors*
P. Stenström
erschienen in
IEEE Computer, No. 23,Vol. 6, 1990.

[Tra83] *Trends in System Aspects of Database Management*
I. L. Traiger
erschienen in
Proc. of the 2nd Intern. Conf. on Databases, 1983.

[WFM94] *Glossary*
Workflow Management Coalition
Doc. No. TC00-0011, 1994.

[WäRe92] *The ConTract Model*
H. Wächter, A. Reuter
Kapitel 7 in
Database Transaction Models for Advanced Applications
A. K. Elmagarmid
Morgan Kaufmann Publishers, 1992.

[Wäch96] *Eine Architektur für die zuverlässige Abwicklung verteilter Anwendungen auf gemeinsamen Ressourcen*
Helmut Wächter
Dissertation an der Fakultät Informatik der Universität Stuttgart, 1996.

[WeSc92] *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*
G. Weikum, H.-J. Schek
Kapitel 13 in
Database Transaction Models for Advanced Applications
A. K. Elmagarmid
Morgan Kaufmann Publishers, 1992.

[Weik89] *Principles and Realization Strategies of Multilevel Transaction Management*
G. Weikum
erschienen in
ACM Transactions on Programming Languages and Systems, Vol. 11, No. 2, S. 249-283, 1989.

[Weik91] *Principles and Realization Strategies of Multilevel Transaction Management*
G. Weikum
erschienen in
ACM Transations on Database Systems, Vol. 16, No. 1, 1991.

[XOP93] *Distributed Transaction Processing: Reference Model - Version 2*
X/Open Guide
X/Open Company Limited, 1993.

[ZiCh91] *Supercompilers for Parallel and Vector Computers*
H. Zima, B. Chapman
ACM and Addison Wesley Pub., 1991.

Index