Transaktionskonzepte und Recovery-Verfahren für Objektbanken:

Zuverlässige Abwicklung großer verteilter Anwendungen mit ConTracts - Architektur einer Prototypimplementierung

Andreas Reuter, Friedemann Schwenkreis, Helmut Wächter

email {reuter,schwenkreis,waechter}@informatik.uni-stuttgart.de

Universität Stuttgart, IPVR, Breitwiesenstr. 20-22, 7000 Stuttgart 80

1 Einleitung

Das in diesem Aufsatz beschriebene ConTract-Modell und seine prototypische Implementierung APRICOTS entstanden als Ergebnis eines Teilprojektes im Schwerpunktprogramm „Objektbanken für Experten". Das Teilprojekt hatte die Aufgabe, Transaktionsmechanismen für die funktionell erweiterten Datenbanksysteme zu entwickeln, die mit dem Arbeitstitel „Objektbanken" bezeichnet waren. Dabei ergab sich gleich am Anfang eine eigentlich nicht überraschende Schwierigkeit: Da die höheren Schichten der Objektbankkomponenten, also etwa die für Regelbehandlung, für die Verwaltung komplexer Objekte, für die Unterstützung aktiven Verhaltens usw., noch nicht entworfen waren, gab es auch noch keine klar definierten Anforderungen an die Transaktionsverwaltung. Zu warten, bis derartige Anforderungen erarbeitet waren, kam wegen des Zeitplanes und der engen Verzahnung der Teilvorhaben nicht in Betracht.

Es mußte also ein anderer Ansatz gefunden werden, Transaktionsmechanismen so zu entwerfen, daß diese leicht auf die erst später sich konkretisierenden Bedürfnisse der anderen Teilprojekte zugeschnitten werden konnten. Damit schieden alle in irgendeiner Art „geschlossenen" Modelle aus; es kam vielmehr darauf an, einen möglichst kleinen Satz von Basismechanismen zu entwickeln (unterhalb der klassischen ACID-Transaktionen), aus denen sich dann je nach Anforderung das gewünschte Betriebs- und Fehlerverhalten algorithmisch zusammensetzen ließ.

Aber auch das ist im Nachhinein leichter gesagt, als es seinerzeit getan war. Bezüglich der für ein flexibles Ausführungsmodell geeigneten Basismechanismen gab es nämlich keine Anhaltspunkte. Die Definition des klassischen Transaktionskonzeptes trägt nichts dazu bei. Es hat ja gerade den Vorteil, daß es deskriptiv, d.h. nur über seine beobachtbaren Eigenschaften definiert ist, und bezüglich der zur Implementierung verwendeten Basismechanismen keinerlei Vorgaben macht. Ansonsten gab es im Hinblick auf verschiedene Funktionsklassen mehr oder weniger lange Negativ-Listen: Atomizität ist für Entwurfstransaktionen nicht geeignet, Serialisierbarkeit ist für lang andauernde Berechnungen nicht akzeptabel, und so fort. Nun gilt zwar nach Spinoza, daß „omnis determinatio est negatio", doch das löst noch nicht das Problem, wie man eine derartige Liste abgelehnter Eigenschaften in einen positiven Entwurf eines flexiblen Ausführungsmodells wendet. Diesem

Aspekt ist das folgende Kapitel 2 gewidmet. Die weiteren Kapitel, insbesondere 3 -5, geben einen Überblick über das ConTract-Modell[1] und beschreiben dann die Architektur des Prototypsystems APRICOTS. Kapitel 6 diskutiert den ConTract-Ansatz im Vergleich zu anderen Entwürfen, und Kapitel 7 umreißt den derzeitigen Stand der Implementierung des ConTract-Systems und seiner Verwendung.

2 Klassifikation der Anforderungen in erweiterten DB-Systemen an die Ausführungsumgebung

Man kommt dem Problem der Identifikation von Basismechanismen zur Ablaufsteuerung in funktionell mächtigen Datenverwaltungssystemen dann bei, wenn man die ACID-Eigenschaften klassischer Transaktionen nicht als etwas Gegebenes auffaßt, das man brauchen kann oder nicht, sondern wenn man analysiert, für welche Komponenten einer Ablaufsteuerung die einzelnen Buchstaben des Akronymes Festlegungen treffen. Bei dieser Betrachtungsweise wird offenbar, daß herkömmliche Transaktionen zwar vielseitige, letztlich aber geschlossene Ausführungsmodelle sind, die durch Kombination bestimmter Optionen aus einem Satz von Basisfunktionen entstehen. Wenn diese Basisfunktionen herausgearbeitet sind und wenn klar ist, wie sie konkret instantiiert werden können, erhält man eine Erzeugungsvorschrift für eine Vielzahl von Ausführungsmodellen - mit ACID-Transaktionen als einem wichtigen Sonderfall.

Gehen wir nun die Buchstaben des ACID-Akronyms der Reihe nach durch und versuchen, die dahinter liegenden Aspekte einer Ablaufsteuerung zu identifizieren.

2.1 Atomarität (Atomicity)

Die Festlegung auf Ununterbrechbarkeit einer Transaktion ist der größte Vorteil des Konzeptes - und zugleich sein größter Nachteil, je nachdem, aus welcher Perspektive man urteilt. Der Vorteil liegt in der unübertrefflichen Einfachheit des resultierenden Ausführungsmodelles. Atomare Zustandsübergänge haben ein ideales Fehlerverhalten und sind notwendige Bausteine für die Konstruktion hochzuverlässiger komplexer Systeme. Der Nachteil der Ununterbrechbarkeit liegt in der impliziten „Nichts“-Option, und dieser Nachteil wiegt um so schwerer, je mehr Arbeit von der Transaktion eingeschlossen wird. So erwünscht es ist, daß im Fehlerfalle der inkonsistente Zustand automatisch verschwindet, so unerwünscht ist es, daß unter Umständen nützliche Arbeit innerhalb derselben Transaktion ebenfalls verschwindet. Aber es geht gar nicht primär darum, ob Atomizität gut oder schlecht ist (eine solche Frage wäre ohnehin falsch gestellt), sondern darum, welche Aspekte einer Ablaufsteuerung durch „atomicity“ auf eine bestimmte Variante festgelegt werden.

Hier sind zwei Effekte besonders hervorzuheben: Einmal die strikte Bindung aller (geschützten) Änderungen an die durch eine Transaktion definierte Kontrollsphäre, und zum anderen der Verzicht auf jegliche Kontrolle über Transaktionsgrenzen hinweg. Betrachten wir zunächst die Kapselung von Änderungen.

Atomizität bedeutet, daß Änderungen *dann und nur dann* entstehen, d.h. sichtbar nach außen werden, wenn die sie umschließende Transaktion erfolgreich abgeschlossen ist. So

[1] Für eine ausführliche Darstellung des ConTract-Modelles siehe auch [WäRe 90, WäRe 92].

natürlich dies auf den ersten Blick ist, hat es doch einige Konsequenzen, die für manche Anwendungen einschränkend sind. Es ist z.B. unmöglich, aus einer scheiternden Transaktion heraus eine Fehlermeldung sicher zu senden [GrRe92]. Entweder ist die Nachricht transaktionsgeschützt, dann bewirkt das Rollback der Transaktion, daß die Nachricht nicht abgeschickt wird. Oder aber die Nachricht ist nicht transaktionsgeschützt, dann kann ihre Ablieferung nicht garantiert werden. Ähnliche Probleme ergeben sich, wenn z.B. in der Datenbank mitgeführt werden soll, wie oft bestimmte Transaktionstypen gescheitert sind. Derartige Zähler müssen genau dann zuverlässig erhöht werden, wenn die jeweilige Transaktion Rollback durchführt.

Dies führt unmittelbar zum zweiten Aspekt: Die genannten Probleme ließen sich lösen, wenn es möglich wäre, Kontrolle über Transaktionsgrenzen hinweg auszuüben. Damit könnte dann ausgedrückt werden, daß eine Transaktion T_i dann und nur dann ausgeführt werden darf, wenn eine andere Transaktion T_j Commit bzw. Rollback aufgerufen hat. Derartige Beziehungen müßten natürlich genauso verbindlich und im Fehlerfall wiederherstellbar sein wie die konventioneller Transaktionen. Mit anderen Worten: Atomizität bedeutet den Verzicht auf jeglichen expliziten Kontrollfluß. Viele Anwendungen brauchen jedoch genau die Möglichkeit der Spezifikation von Kontrollflüssen über Transaktionsgrenzen hinweg.

2.2 Konsistenzerhaltung (Consistency)

Die simple Annahme hinter klassischen Transaktionen besagt, daß jede Transaktion einen global konsistenten Zustand der Datenbank sieht und nur dann Commit aufruft, wenn sie einen (neuen) konsistenten Zustand erreicht. Es ist unmittelbar klar, daß eine explizite Beschreibung der Konsistenzbedingungen auf dieser Ebene entfällt, da ja alle Bedingungen als erfüllt angenommen werden. Es ist ebenso klar, daß diese Annahme in den meisten Fällen äußerst unrealistisch ist, da ihre strikte Durchsetzung massive Einschränkungen des DB-Betriebes zur Folge hätte. Natürlich ist es in der Mehrzahl der Fälle auch gar nicht nötig, eine derart umfassende Annahme zu machen. Die Konsistenzbedingungen in Teilen der DB, die eine Transaktion nicht berührt, sind für diese Transaktion irrelevant; meistens sind sogar nur einige Konsistenzbedingungen auf den berührten Teilen für eine Transaktion von Bedeutung. Dem Rechnung zu tragen erfordert allerdings - im Unterschied zu herkömmlichen Transaktionen - daß jede Transaktion explizit macht, welche Konsistenzbedingungen erfüllt sein müssen, damit sie beginnen kann, und welche Konsistenzbedingungen am Ende erfüllt sein sollten. Daß es darüber hinaus einen durch das Schema bestimmten Satz von Konsistenzbedingungen gibt, die immer und für alle Transaktionen gelten müssen (z.B. Primärschlüsselbedingungen), ist klar.

2.3 Isolation

ACID-Transaktion laufen in einem virtuellen Einbenutzerbetrieb ab; sie sind isoliert gegen die Effekte anderer Transaktionen. Ob die Isolierung mit Hilfe von Sperrprotokollen erreicht wird, die Serialisierbarkeit garantieren, oder ob andere Methoden eingesetzt werden, ist zweitrangig. Ähnlich wie bei der Atomizität führt eine Analyse der Konzepte hier zu widersprüchlichen Ergebnissen. Einerseits vermeidet die Garantie der isolierten Ausführung jede Notwendigkeit der Behandlung von Parallelität, Asynchronität und Konflikten, andererseits impliziert ihre strikte Interpretation im Sinne der Wiederholbarkeit des Lesens starke Einschränkungen bezüglich des konkurrierenden Zugriffes auf gemeinsame Daten. Eine Lockerung der sehr weitreichenden Isolationsgarantien muß dahin gehen, daß die von einer Transaktion berührten Daten nicht völlig unzugänglich gemacht werden, sondern daß durch das Transaktionsprogramm explizit spezifiziert werden kann, welche Isolierungseffekte notwendig sind, um seinen konsistenten Ablauf zu garantieren. Bei der Ausarbeitung dieses Gedankens wird sich zeigen, daß dadurch zweierlei Freiheitsgrade gewonnen werden: Einmal kann die Semantik der einzelnen (Datenbank-) Operationen berücksichtigt werden, um festzulegen, was „Nebeneffekte“ sind; zum anderen können viele Isolations-Anforderungen auf temporäre Konsistenzbedingungen abgebildet werden, so wie sie in 2.2 erläutert wurden. Und in der Tat sind ja Isolationsgarantien strukturell nichts anderes als lokale, für eine Transaktion relevante Konsistenzbedingungen.

2.4 Dauerhaftigkeit (Durability)

Auch die Dauerhaftigkeits-Garantie enthält zwei Komponenten. Einmal verpflichtet sich das System, Änderungen abgeschlossener Transaktionen wiederherstellbar zu halten, zum anderen befreit die Definition das System von der Verpflichtung, Änderungen abgeschlossener Transaktionen zurückzunehmen zu können. Eine genauere Analyse zeigt, daß die Verkopplung dieser beiden Aspekte nur dann Sinn macht, wenn man ein strikt atomares Ausführungsmodell hat. Verzicht auf „reine“ Ununterbrechbarkeit im Sinne einer Kontrollflußbeschreibung bedeutet automatisch auch, daß es Mechanismen zur Rücknahme abgeschlossener Transaktionen geben muß. Derartige Kompensationen können natürlich niemals automatisch aus der Originaltransaktion abgeleitet werden, da ihre Einzelschritte stark von anwendungsspezifischen Regeln abhängen [Davi78].

Zusammenfassend können wir festhalten, daß ein flexibleres Ausführungsmodell Möglichkeiten zur expliziten Beschreibung der folgenden Aspekte braucht: Kontrollfluß, Daten- und Aktionsabhängigkeiten, lokale Konsistenzbedingungen, Isolationsbedingungen, Konfliktbehandlungsmaßnahmen und Kompensationsschritte. Die damit spezifizierbaren Kontrollkonstrukte haben vom *ACID* der klassischen Transaktion weder das *A*, noch das *C*, noch das *I*, noch das *D*. Die Autoren sind daher der Auffassung, daß es nicht sonderlich glücklich ist, diese neuen Konstrukte trotzdem als (wie sehr auch immer erweiterte) Transaktionen zu bezeichnen. Im Sinne der begrifflichen Klarheit ist es vielmehr notwendig, eine andere Benennung zu finden.

3 Das ConTract-Modell

Bei der Entwicklung des ConTract-Modelles stand ein Gedanke im Vordergrund: Da Anwendungen auf Objektbanken, d.h. auf funktionell erweiterten Datenbanksystemen sicherlich komplexer sind als herkömmliche Anwendungen, darf die Unterstützung des Anwendungsprogrammierers im Hinblick auf Fehlerbehandlung und Systemverfügbarkeit nicht schlechter sein als in konventionellen Transaktionssystemen. Dies wiederum bedeutet, daß alle mächtigeren Ausführungsmodelle letztlich auf atomare Zustandsübergänge, d.h. auf ACID-Transaktionen zurückgeführt werden müssen.

Atomare Transaktionen sind die Grundbausteine von ConTracts. Ein ConTract umfaßt die Ausführung von beliebig vielen Transaktionen (sequentiell oder parallel), gemäß einem vordefinierten bzw. dynamisch sich verändernden Programm, welches den Kontrollfluß von einer Transaktion zur nächsten beschreibt. Dieses Programm wird im folgenden als

„Skript“ bezeichnet. Für jeden ConTract übernimmt das ausführende System folgende Garantien:

- Ein einmal gestarteter ConTract wird vom System nicht vergessen (auch nicht durch Knotenausfälle oder sonstige Komponentenverluste), bis er explizit beendet ist.
- Bei jedem Komponentenfehler stellt das System sicher, daß eventuell verfügbare Alternativ-Komponenten gefunden und für die laufenden ConTracts verwendet werden.
- Unter allen Umständen wird ein begonnener ConTract „nach vorn“, d.h. gemäß der Skript-Spezifikation fortgesetzt (kein Rollback).
- Nur auf explizite Anforderung eines Programmes im ConTract (oder des Systemverwalters) wird für einen ConTract Kompensation eingeleitet; die Beendigung der Kompensation ist garantiert.
- Abgesehen von Fehlern in der Anwendung (Endlos-Schleifen) wird jeder ConTract in endlicher Zeit sein normales Ende erreichen oder die Kompensation wird durchgeführt.

Zur Illustration der Konzepte betrachten wir das folgende Beispielskript:

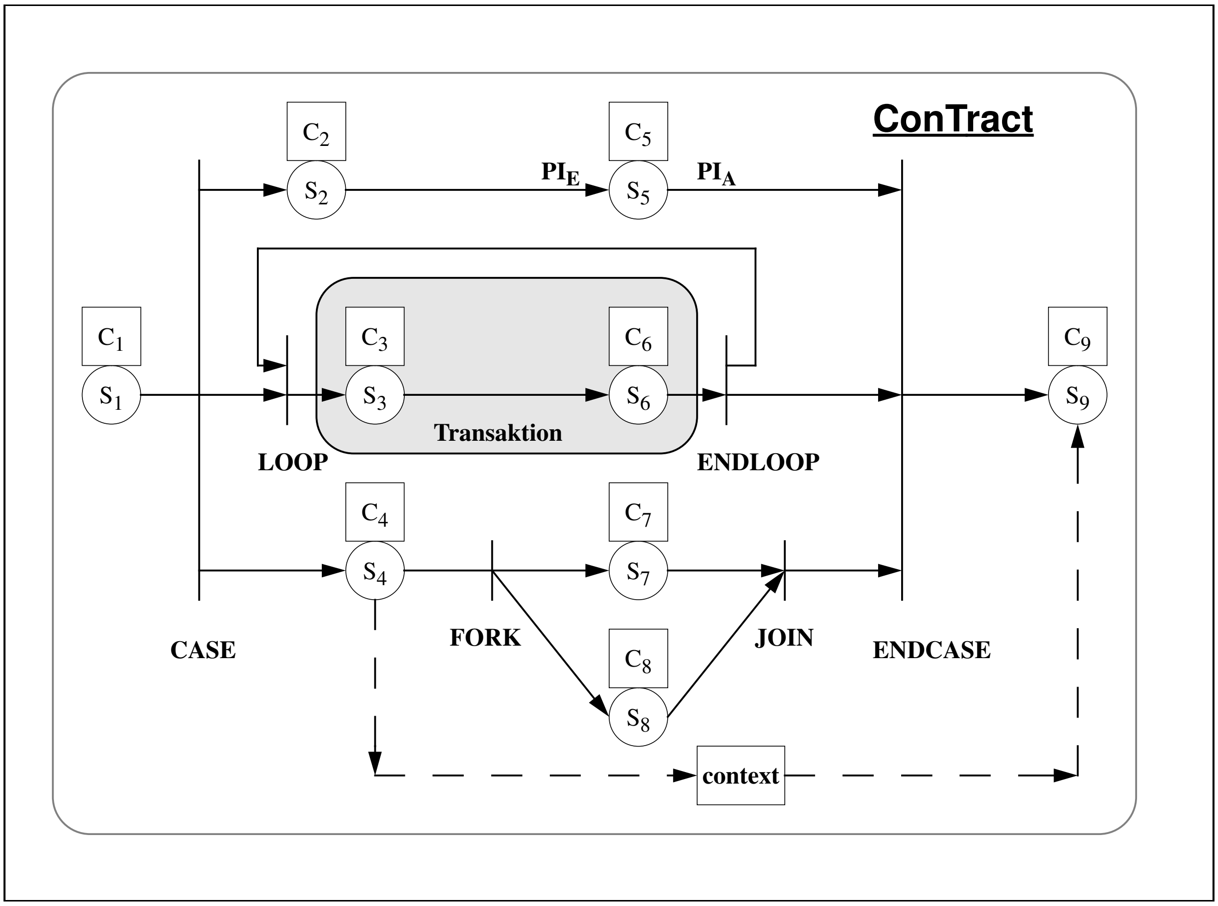

Abbildung 1: Exemplarisches ConTract-Skript

Es wird unmittelbar deutlich, daß auf der Skript-Ebene die üblichen Kontrollfluß-Konstrukte wie Sequenz, Verzweigung, Schleifen, fork, join u.ä. zur Verfügung stehen. Die Sprachkonstrukte ermöglichen die Beschreibung und Abwicklung von Abläufen in der Ausdrucksmächtigkeit von Petri-Netzen. Die mit S_i bezeichneten Knoten repräsentieren beliebige (sequentielle) Code-Stücke, die an der entsprechenden Stelle des Kontrollflusses aktiviert werden. Ohne weitere Angaben wird die Ausführung eines Schrittes (Steps) S_i in eine ACID-Transaktion eingebettet. Sind andere Gruppierungen gewünscht, so kann dies angegeben werden; z.B. bilden S_3 und S_6 zusammen eine Transaktion

Wird das fork-Konstrukt verwendet, so entstehen parallele Teilabläufe innerhalb eines ConTracts (S_7, S_8), die unabhängig voneinander vom System kontrolliert werden, bis sie über ein korrespondierendes join wieder zusammenlaufen. Jeder Step kann beliebige Zwischenergebnisse erzeugen, die solange stabil und wiederherstellbar gehalten werden, wie der ConTract existiert. Sie sind nur für andere Steps desselben ConTract sichtbar. Im obigen Beispiel schreibt S_4 etwa in eine Variable mit Namen *context*, und S_9 liest diese Variable. Ein später noch näher erläuterter Adressierungsmechanismus auf Versionsbasis stellt sicher, daß es trotz Parallelität innerhalb eines ConTract nicht zu Zugriffskonflikten auf Kontext-Variablen kommen kann.

Die Definition von lokalen Konsistenzbedingungen sowie die Spezifikation der Isolationsanforderungen der Steps ist am Beispiel von S_5 skizziert. S_5 ist „geschützt" durch ein Prädikat PI_E, das die folgende Wirkung hat: Wenn der Kontrollfluß soweit ist, daß S_5 ausgeführt werden könnte, wird getestet, ob PI_E erfüllt ist - hierbei handelt es sich üblicherweise um Bedingungen auf der globalen Datenbank. Ist dies der Fall, werden die entsprechenden Objekte durch normale Synchronisierungsmechanismen der S_5 umgebenden Transaktion stabilisiert. Ist PI_E *nicht* erfüllt, obwohl S_5 ansonsten aktiviert werden könnte, liegt ein Betriebsmittelkonflikt vor. Dieser muß entweder durch expliziten Eingriff aufgelöst werden, oder aber es muß auf ein „Konfliktbehandlungs-Skript" umgeschaltet werden (hier nicht gezeigt), das eine interne Bereinigung der Situation versucht. Mit diesem Mechanismus lassen sich nun auch Isolationsbedingungen über Transaktionsgrenzen hinweg dadurch kontrollieren, daß eine Eingangsbedingung PI_E auf ein sog. Invarianzprädikat PI_A eines vorangegangenen Programmschrittes Bezug nimmt. Die ACID-Isolationsgarantie beschränkt sich dagegen nur auf den Kontrollbereich einer einzigen Transaktion.

Die jedem Schritt S_i zugeordneten Kästen C_i repräsentieren den Code für die Kompensation dessen, was der Originalschritt getan hat. Sollte es erforderlich sein, im Verlauf einer ConTract-Ausführung externalisierte Änderungen zurückzunehmen, garantiert das System die Ausführung der Kompensationsschritte in der umgekehrten zeitlichen Ordnung der Originalschritte mit den dafür jeweils vorgehaltenen Aufrufparametern. Da weder im Skript noch im Step-Code eine feste Zuordnung der Schritte zu Rechnerknoten erfolgen muß, kann es sein, daß Schritt C_i auf einem anderen Knoten ausgeführt wird als Schritt S_i - sofern beide alle erforderlichen Betriebsmittel haben.

Zusammenfassend seien nochmals die Basismechanismen aufgezählt, aus denen das flexible Ausführungsmodell ConTract besteht: Transaktionen als Basis-Ausführungsschritte; allgemeine Kontrollfluß-Konstrukte zur Verknüpfung der einzelnen Schritte; stabile, fehlertolerante Abwicklung des Kontrollflusses; stabile, wiederherstellbare Kontextvariablen für Zwischenergebnisse; Behandlung lokaler Konsistenzbedingungen und längerfristiger Isolationsbedingungen durch Eingangs- und Ausgangsinvarianten; explizite Behandlung von Betriebsmittel-Konflikten; garantierte Ausführung der inversen Kompensations-Sequenz; dynamische Abbildung der Teilschritte auf verfügbare Rechnerknoten.

Es ist ganz klar, daß dieses Konstrukt weit von der Einfachheit der ACID-Transaktion entfernt ist. Diese hat den Vorteil der deskriptiven Definition und den Nachteil der Abgeschlossenheit. Eine entsprechende deskriptive Definition der mächtigen Kontrollkonstrukte für neuartige Anwendungen scheint (zumindest derzeit) nicht erreichbar zu sein. Deshalb geht das ConTract-Modell ganz bewußt auf eine tiefere, prozedurale Stufe, um die erforderliche Flexibilität durch „Programmierung" auf Skript-Ebene zu erreichen.

4 Anforderungen an eine Architektur zur Realisierung des ConTract-Modells

Die Architektur des Prototypsystems zur ConTract-Verarbeitung ist entscheidend von zwei übergeordneten Entwurfszielen geprägt:

1. Wie zu Beginn von Kapitel 3 bereits angedeutet, soll ein ConTract-System die oben beschriebenen Verarbeitungsmechanismen als *anwendungsneutralen Systemdienst* für eine *breite Klasse von Anwendungen* realisieren. Das beinhaltet zwei Anforderungen: Erstens sollen alle Aspekte der Fehlerbehandlung, Kontrolle der Nebenläufigkeit, Asynchronität, Betriebsmittelverwaltung usw. insoweit zur Ebene der Anwendung hin verborgen bleiben, daß alle diesbezüglich zur Laufzeit anfallenden Aufgaben vom System implementiert werden. Zweitens muß die ConTract-Laufzeitumgebung außer Datenbank-Programmen auch Anwendungen unterstützen, die daneben auf andere, u. U. völlig verschiedenartige Ressourcen (Objekte) zugreifen, wie beispielsweise auf Drucker und Mail-Server, aber auch auf physische Geräte wie Transportfahrzeuge oder Fertigungsmaschinen in einer Fabrikhalle.

2. Die Implementierung eines ConTract-Systems soll auf Standard-Hardware– und –Software-Komponenten aufbauen. Vorhandene Datenbank-Techniken sollen dabei mit Lösungsansätzen aus den Bereichen Betriebssysteme und Programmiersprachen so kombiniert werden, daß existierende Systeme und Schnittstellen nur in einem unbedingt notwendigen Umfang erweitert werden müssen.

Aus dem Architekturschema eines erweiterten Objektbanksystems in Abb. 2 wird ersichtlich, daß die ConTract-Verwaltung zwischen Anwendungsebene und den Objektverwaltern angesiedelt ist. In der folgenden Anforderungsanalyse konzentrieren wir uns auf die wichtigsten Aufgaben bei der ConTract-Abwicklung sowie die Schnittstellen und Anforderungen an die Objektverwalter und ausgewählte Basisdienste. Auf die Schnittstelle zur Anwendungsebene näher einzugehen, ist im Rahmen dieses Berichtes leider nicht möglich (vergl. dazu [Wahl91, Zell92, Bilg92, Eitl92]).

4.1 Robuste Kontrollflußabwicklung

Robuste Kontrollflußverwaltung gemäß dem ConTract-Modell beinhaltet eine dreifache Anforderung an das Laufzeitsystem zur ConTract-Verwaltung. Erstens müssen das Ereignis *„Ende einer Step-Ausführung"* sowie die damit eingetretenen Aktivierungen nachfolgender Programmschritte als *stabile Ereignisse* verwaltet werden, d.h. insbesondere, auch nach einem Systemausfall noch gültig sein. Zweitens hat der ConTract-Manager dafür Sorge zu tragen, daß Steps, die infolge eines Systemfehlers scheitern, wiederholt werden.

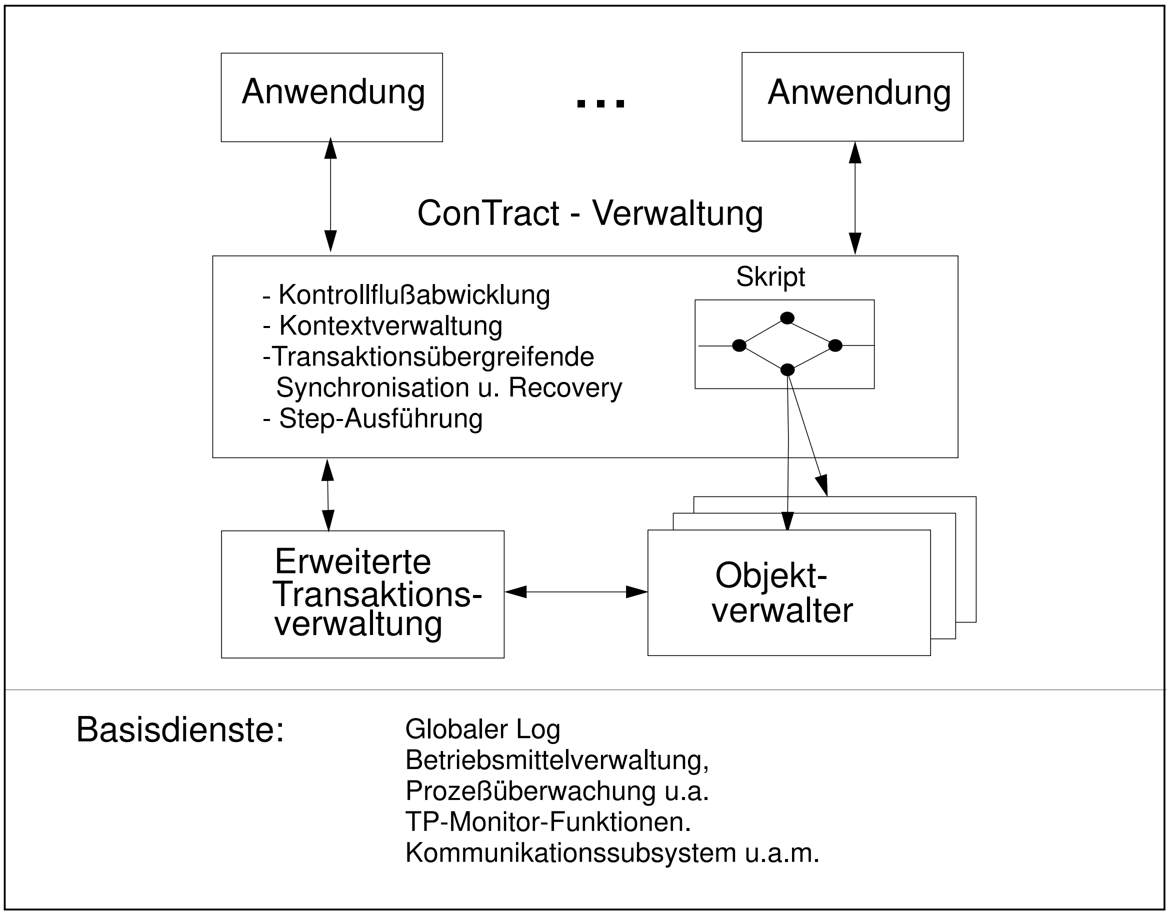

Abbildung 2: Architektur eines Objektbanksystems mit ConTract-Verwaltung

Gegebenenfalls muß er dazu auf eine funktional äquivalente Verarbeitungsinstanz (Prozeß, Knoten) ausweichen. Keinesfalls darf ein betroffener Programmschritt aber mehrfach ausgeführt werden. Und schließlich muß es drittens möglich sein, beim Totalausfall eines Knotens eine unterbrochene ConTract-Verarbeitung auf ein Ersatzsystem zu verlagern und dort korrekt weiterzuführen. Dazu werden Recovery-Verfahren benötigt, die den Verarbeitungszustand eines ConTracts stabil verwalten bzw. vollständig rekonstruieren können [Masm92].

4.2 Kontextverwaltung und -adressierung

Um eine unterbrochene ConTract-Verarbeitung wiederaufnehmen und korrekt nach vorne fortsetzen zu können, müssen außer dem Inhalt der globalen Datenbank und dem Ausführungszustand einer Skript-Abwicklung auch noch alle *lokalen Zustandsdaten* einer Anwendung stabil verwaltet werden. Zum lokalen Zustand einer Anwendung gehören u.a. globale Programmvariable, Parameter entfernter Objektverwalteraufrufe, etablierte Kommunikationsverbindungen (sessions), Fensterinhalte, Dialogzustände, offene Dateien und Cursor, die von mehr als einem Step bearbeitet werden usw. Diese Auflistung zeigt, daß Kontext über mehr als einen Prozeß verstreut sein kann und deshalb die herkömmlichen Verfahren zur Erzeugung von Sicherungspunkten auf Prozeß- und Datenbankebene nicht ausreichen, um die vorwärts orientierte ConTract-Recovery zu gewährleisten.

Die Realisierung des Kontext-Datenmodells stellt folgende Anforderungen an die Verwaltung der Kontextvariablen eines ConTracts. Für die korrekte Adressierung der Kontextvariablen ist ein Mechanismus erforderlich, der die im Step-Programm verwendeten Variablennamen auf die im Skript eingeführten logischen Bezeichner umsetzt und die zur Laufzeit referenzierten Kontextvariablen automatisch auf das korrekte Element der Kontextdatenbank abbildet. Diese Adreßbindung muß gewährleisten, daß auch bei mehrfacher, eventuell sogar paralleler Aktivierung eines Steps (z.B. in Schleifen) die dann auch mehrfach erzeugten (Ausgabe-)Kontextelemente eindeutig referenzierbar sind. Außerdem dürfen alte Werte der Kontextvariablen nicht überschrieben werden, da sonst eine eventuell erforderliche Kompensation nicht mehr durchführbar ist. Ebenso ist das die Voraussetzung für zeitabhängige Zugriffe auf die Ausführungshistorie eines ConTract, etwa zur Beantwortung der Frage nach dem Ausgabekontext von Step S_4 des Beispielskriptes in Kapitel 3 (Abb. 1 auf Seite 5).

Zur eindeutigen Referenzierung der Eingabeparameter, die ein Programmschritt (S_9 in Abb. 1) aus dem Kontext benötigt, müssen die Kontextvariablen bei ihrer Entstehung mit mindestens den folgenden Attributen gekennzeichnet werden:

- Identifikator des laufenden ConTracts
- Name des erzeugenden Steps
- Step-Version (die wievielte Aktivierung dieses Steps, z.B. in einer Schleife)
- ein Zähler zur Unterscheidung paralleler Aktivierungen desselben Steps in *PAR-FOREACH*–Anweisungen
- Name und Datentyp des Kontextelementes
- Zeitstempel

4.3 Transaktionsübergreifende Synchronisation

In herkömmlichen DBMS ist das Halten und Verwenden von Sperren an genau eine aktive Transaktion gebunden. Sind in einem ConTract-Skript jedoch mehrere Steps zu einer Transaktion zusammengefaßt, ist es u.U. erforderlich, die erworbenen Sperren von einer Transaktion des Basissystems an die nächste weiterzureichen. Zur Synchronisation langlebiger Aktivitäten braucht man weiterhin einen Mechanismus, der es einem suspendierten ConTract auch *ohne* eine aktive Transaktion erlaubt, Sperren zu halten oder auf andere Weise Synchronisationsbedingungen zu kontrollieren - und das auch über einen Systemausfall hinweg.

Die Synchronisation mit Invarianten erfordert insbesondere zweierlei. Erstens benötigt man einen Formalismus, um die Prädikate PI_E und PI_A zu formulieren und auszuwerten. Im Kontext relationaler DBMS bietet sich dazu die Sprache SQL an. Um zweitens die gesetzten Invarianten gemäß der im Skript angegebenen Strategie zu verwalten, müssen die Zugriffe auf die Objekte eines Isolationsprädikates PI_A entsprechend kontrolliert werden. Das kann beispielsweise über geeignete Sperrmodi oder über Prädikate realisiert werden, die einem Objekt zugeordnet sind und die ein Zugreifer beachten muß.

Minimalanforderung des Invariantenmechanismus an die Objektverwalter sind Schnittstellen und Protokolle, die das Vorliegen sowie die Art eines Zugriffskonfliktes der Anwendung mitteilen, damit diese überhaupt Aktionen zur Konfliktbehandlung unternehmen kann. Auf keinen Fall darf ein Aufrufer implizit (d.h. ohne seine Zustimmung) blockiert werden, da sonst eine explizite Behandlung und Auflösung von Konflikten prinzipiell unmöglich wäre.

Auch bei optimistischer Verwaltung einer gesetzten Invariante darf eine elementare Korrektheitsbedingung nicht verletzt werden: Bezieht sich eine Eingangsinvariante PI_E auf ein zuvor definiertes Prädikat PI_A, dann muß sichergestellt sein, daß die darin referenzierten Objekte zwischenzeitlich nicht gelöscht (oder schlimmer noch: mit völlig anderer Semantik neu erzeugt) werden. Das ist beispielsweise über das Prinzip der Objektidentität oder durch sog. Existenzsperren möglich, die nichts anderes außer dem Löschen eines Objektes verhindern.

4.4 Erweiterte Transaktionsverwaltung

Eine separate Komponente zur Transaktionsverwaltung ist notwendig, um ConTracts koordinieren zu können, die mehrere Objektverwalter unter einer Transaktion aufrufen. Im Extremfall kann ein Programmschritt auf ein Ensemble von Objektverwaltern verschiedener Hersteller und unterschiedlichen Typs zugreifen. Man denke beispielsweise an eine Lagerverwaltung eines Fertigungsunternehmens, wo der Kontrollbereich um die Transaktion "Entnahme, Verbuchung und Auslieferung eines Artikels" sowohl Zugriffe auf mehrere Datenbanken (Bestandsführung, Kostenverrechnung, Fertigungsplanung u.a.) als auch physische Aktionen verschiedener Geräte und Maschinen (Lager- und Transportfahrzeuge usw.) einschließt. Die Verwaltung derart "heterogener" und umfassender Kontrollbereiche stellt eine zentrale Herausforderung an Transaktionssysteme dar, die auch im Fertigungsbereich einsetzbar sein sollen [ReSc91].

Um solche verteilten und heterogenen Berechnungen zu koordinieren, muß die Transaktionsverwaltung zur Anwendungsebene und zu den Objektverwaltern eine standardisierte Schnittstelle anbieten, wie sie etwa in dem XA-Modell in [XOpen92] spezifiziert ist. Umgekehrt eröffnet eine externe Transaktionsverwaltung mit einem standardisierten Protokoll jedem Objektverwalter, an einer transaktionsorientierten Verarbeitung teilnehmen zu können, ohne selbst eine Komponente zur Transaktionsverwaltung implementieren zu müssen.

Darüber hinaus benötigt man zur Realisierung des ConTract-Modells flexiblere und zuverlässigere Commit-Protokolle als die in herkömmlichen DBMS üblichen. Unter anderem muß es möglich sein, den Koordinator eines Zweiphasen-Freigabe-Protokolls erst am Ende einer langdauernden Aktivität zu bestimmen [KlRe88], sowie die Protokollabwicklung von einem unsicheren auf einen anderen, verläßlicheren Knoten zu verlagern [RoPa90]. Für die konsistente und effiziente ConTract-Abwicklung braucht man allgemeine geschachtelte Transaktionen, um sowohl die Verwaltungsaktivitäten und (verteilten) Step-Bearbeitungsaufträge als auch die Zugriffe auf die Objektverwalter zu kontrollieren. Um gescheiterte Step-Transaktionen korrekt zu behandeln, muß es außerdem möglich sein, zwischen einem Transaktionsabbruch aufgrund eines Systemfehlers (Verklemmung, Systemabsturz, usw.) und anwendungsbedingtem Zurücksetzen zu unterscheiden: Während im ersten Fall ein Neustart des Steps angebracht ist, muß im zweiten Fall die Zahl der Wiederholungsversuche begrenzt werden, um nicht in eine Endlosschleife zu geraten.

4.5 Einbindung der Objektverwalter in die ConTract-Verarbeitung

Wie eben angedeutet, können alle Objektverwalter an ein ConTract-System angeschlossen werden, falls sie über eine Schnittstelle verfügen, die der XA-Spezifikation aus [XOpen92] genügt. Das bedeutet, daß sie alle Objektzugriffe unter der Kontrolle einer globalen (d.h. extern koordinierten) Transaktion ausführen. Um an einem extern koordinierten Commit-Protokoll teilnehmen zu können, müssen sie insbesondere Funktionen für das *prepare*, *commit* und *abort* einer Transaktion anbieten und für jede Operation auf den von ihnen verwalteten Objekten die entsprechenden undo- bzw. redo-Informationen protokollieren [HaeRe83, GrRe92].
Soll die Implementierung eines Objektverwalters auf einem herkömmlichen DBMS und dessen lokaler Transaktionsverwaltung aufsetzen, muß diese entsprechend um eine *prepare-to-commit*-Funktion erweitert werden. Diese Öffnung des Zweiphasen-Freigabeprotokolls ist eine Grundvoraussetzung für die kontrollierte Abwicklung verteilter Anwendungen auf mehreren Objektverwaltern und damit letztendlich eine *conditio sine qua non* für deren Interoperabilität schlechthin.

Alternativ zu der Verwaltung eigener, lokaler Transaktionen sollten insbesondere anwendungsdefinierte Objektverwalter den globalen Log (4.6) zur Protokollierung von undo- und redo-Informationen zu jeder durchgeführten Operation verwenden. Dadurch vereinfacht sich die globale Transaktionsverwaltung und Recovery erheblich (vgl. z.B. das *Camelot*-System bzw. das TP-System *Encina* [EMS91, EpDi92]). Falls nämlich jeder Objektverwalter sein eigenes Log führen würde, müßten deren lokale Protokolloperationen (z.B. das Schreiben eines Commit-Eintrages) über ein Zweiphasen-Commmit-Protokoll koordiniert werden. Zudem wären die Protokolleinträge zu einer Transaktion über mehrere Log-Komponenten verteilt. Dieser unverhältnismäßig hohe Aufwand unterstreicht die Forderung nach einem einzigen Log pro Knoten.

4.6 Globaler Log

Die zuverlässige Abwicklung verteilter Anwendungen ist elementar auf eine globale, verteilte Protokollkomponente angewiesen, in die der ConTract Manager und alle Objektverwalter schreiben, die transaktionsgeschützte Objekte implementieren. Die ConTract-Verwaltung verwendet diesen Log u.a. dazu, um die für einen fehlerrobusten Ablauf notwendigen stabilen Ereignisse zu realisieren. Beim Wiederanlauf nach einem Systemausfall muß der ConTract Manager aus dem Log seinen eigenen Zustand sowie den aller bearbeiteten ConTracts rekonstruieren. Weil unterbrochene Anwendungen auch nach dem endgültigen Ausfall eines Knotens (und damit auch dessen lokalen Protokolldateien) korrekt nach vorne fortgesetzt werden sollen, ist eine Replikation der Protokolldaten auf mehrere Knoten im Netzwerk erforderlich [DaST87]. Bei einer Migration muß sich der Log komplett mit der ConTract-Verarbeitung auf einen anderen Knoten verlagern lassen oder zumindest über Knotengrenzen hinweg referenziert werden können.

Schreiben mehrere Komponenten in denselben Log-Puffer, müssen sie sich offensichtlich beim Übertragen des Pufferinhaltes auf den Sekundärspeicher und bei anderen Verwaltungsoperationen gegenseitig abstimmen. Auf die damit zusammenhängenden Probleme kann hier aber nicht weiter eingegangen werden.

4.7 Anforderungen an das Basisbetriebssystem

Die ConTract-Verwaltung und andere wichtige Systemkomponenten müssen zur Tolerierung der angedeuteten Systemfehler als stabile Prozesse realisiert werden. Dazu können sog. Prozeßpaare oder Schattenprozesse [GrRe92] zum Einsatz kommen. Zur Sicherung weniger wichtiger Prozesse genügt ein einfacherer Mechanismus zur Prozeß(fern)überwachung, der abgestürzte Prozesse entdeckt und automatisch neu startet bzw. ein individuell angebbares Recovery-Verfahren einleitet. Geeignete Protokolle dafür sind z.B. in [Walt82, Kim84, Kübl 90] beschrieben.

Außerdem benötigt ein ConTract-System eine Komponente, die üblicherweise als TP-Monitor bezeichnet wird, u.a. für die Zuteilung von Betriebsmitteln (Prozessen) an die Step-Bearbeitungsaufträge, zur Lastbalancierung, Zugriffskontrolle, Verwaltung der Prozeß- und Systemkonfiguration usw.

5 Architektur eines Prototypen

5.1 Systemarchitektur

Das mehrstufige Programmiermodell motiviert einen schichtenartigen Aufbau eines ConTract-Laufzeitsystems, dessen zentrale Komponenten in Abb. 3 dargestellt sind:

1. Der sog. **ConTract Manager** fungiert als Koordinator einer ConTract-Ausführung. Er realisiert die Schnittstellen zur Anwendungsebene (Ablaufsteuerung) und zu den Komponenten auf tieferen Schichten des Laufzeitsystems. Seine Hauptaufgabe liegt in der fehlertoleranten Kontrollflußabwicklung und Abbildung von Steps auf Transaktionen. Außerdem nimmt er Step- bzw. transaktionsübergreifende Kontrollaufgaben wahr (Verwaltung der Konsistenz- und Isolationsbedingungen, Durchführung der Konfliktauflösung, Kompensation usw.).

2. Die eigentliche Ausführung der Programmschritte ist vom ConTract Manager abgekoppelt und in eine spezielle Ablaufumgebung für ConTract-Steps, die sog. *Step Computation Server* ausgelagert, die in anderen Systemen auch als Transaktions- oder Anwendungs-Server bezeichnet werden [BET91]. Sie implementieren ein oder mehrere Step-Programme, die vom ConTract Manager an der jeweiligen Stelle einer Skript-Abwicklung aufgerufen werden.

3. Die Step-Programme greifen zur Realisierung einer Anwendungsfunktion i.d.R. auf gemeinsam genutzte Objekte (typischerweise eine Datenbank) zu. **Objektverwalter** (auch **Resource Manager** genannt) kapseln die interne Struktur eines Objektes oder einer Objektklasse und lassen Zugriffe darauf nur über vordefinierte Operationen zu, die über eine RPC-ähnliche Schnittstelle aufgerufen werden. Dabei ist es vollkommen unerheblich, ob ein Aufruf aus einem ConTract oder einer sonstigen Anwendung erfolgt. Ein Objektverwalter muß nur die in 4.5 skizzierten Anforderungen erfüllen und braucht von der ConTract-Semantik nichts zu wissen. Mit der Unterstützung geeigneter Werkzeuge kann er wie ein normaler Server programmiert werden [Zell92].

4. Der **ConTract Processing Monitor** führt allgemeine Verwaltungsaufgaben durch (Prozeß- und Benutzerverwaltung usw.), die hier nicht weiter von Interesse sind.

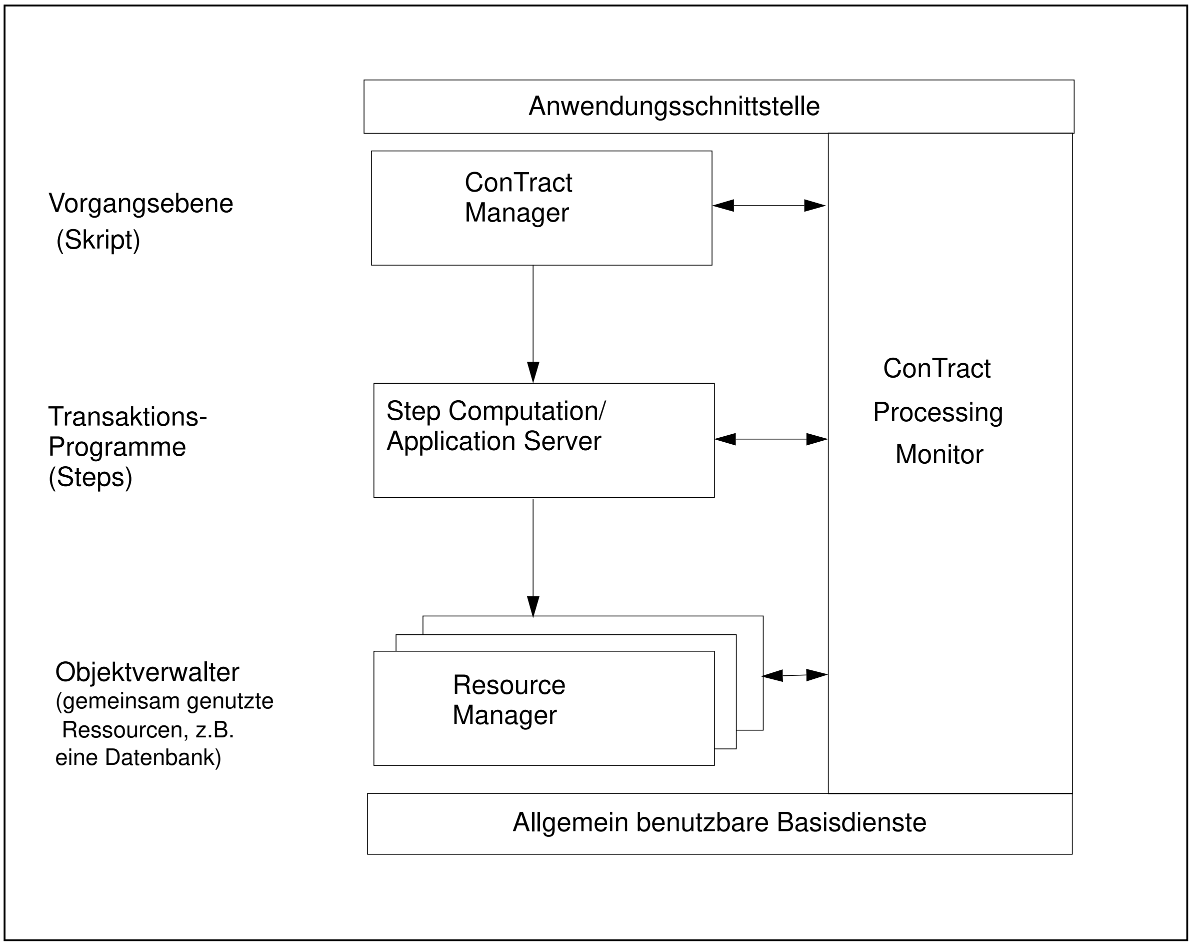

Abbildung 3: Schichtenarchitektur eines ConTract-Systems

5.2 Betriebsmodell

Die skizzierten Komponenten arbeiten bei einer ConTract-Abwicklung folgendermaßen zusammen: Eine Anwendung startet mit dem Kommando `activate <contract-script-name>` den spezifizierten Vorgangstyp und erhält eine systemweit eindeutige Kennung (`contract_id`) zurück. Unter Angabe dieser Kennung kann der weitere Ablauf jederzeit angehalten (`suspend`), fortgesetzt (`resume`), auf einen anderen Knoten verlagert (`migrate`) oder abgebrochen (`cancel`) werden.

Die textuelle Beschreibung eines ConTract-Skriptes (Kap. 3) wird in ein Prädikat-Transitionsnetz vorübersetzt, das für jeden Step Start- und Terminierungsbedingungen enthält. Der ConTract Manager wickelt den in diesem Netz beschriebenen Kontrollfluß ereignisgesteuert ab, indem er für jeden Programmschritt folgenden Zyklus durchläuft:

Ist das Startereignis eines Steps ausgelöst, wird seine Ausführung gestartet, sobald das Prädikat PI_E für seine Aktivierung erfüllt ist und die von ihm benötigten Betriebsmittel zur Verfügung stehen. Dazu erteilt der ConTract Manager einem geeigneten Step Computation Server den Auftrag zur asynchronen Bearbeitung des Steps. Eine asynchrone Ausführung ist offensichtlich notwendig, um eine echt parallele Step-Verarbeitung zu erzielen.
Bis das Ergebnis eines Step-Bearbeitungsauftrages vorliegt, kann der ConTract Mana-

ger andere Aufgaben wahrnehmen, wie z.B. weitere Step-Instanzen aktivieren oder auf die oben erwähnten Befehle zur Ablaufsteuerung seitens der Anwendung reagieren. Trifft die Fertigmeldung eines Programmschrittes ein, wird das dem ConTract Manager durch ein entsprechendes Ereignis angezeigt. Er wertet daraufhin den Rückgabestatus und die Terminierungsbedingungen des Steps aus. Eine erfüllte Terminierungsbedingung löst ein oder mehrere Ereignisse aus, die wiederum die nachfolgenden Steps aktivieren. Damit beginnt der Zyklus von neuem. Ist ein ConTract-Skript vollständig abgearbeitet, wird die Anwendung davon benachrichtigt und der interne Verarbeitungszustand einschließlich der angefallenen Kontextdaten abgebaut.
Der wesentliche Vorteil einer ereignisorientierten Skript-Abwicklung liegt darin, daß der ConTract Manager an keiner Stelle synchron (d.h. blockierend) auf das Ende einer Step-Ausführung warten muß. Deshalb kann er zu jedem beliebigen Zeitpunkt sowohl in der Rolle als Server (zur Ablaufsteuerung für die Anwendung) als auch als Client (Empfang der Fertigmeldung eines Step-Bearbeitungsauftrages) agieren. Diese Möglichkeit ist eine wichtige Implementierungsanforderung bei der Abwicklung paralleler Abläufe in geschichteten Client-Server-Architekturen.

5.3 Prozeßmodell

Client-Server-Anwendungen basieren üblicherweise auf folgendem Prozeßmodell: Client und Server (hier auch Objektverwalter) laufen in unterschiedlichen Adreßräumen und auch die Server sind jeweils in einen eigenen Prozeß geladen. Abhängig von den Gegebenheiten der Systemarchitektur (Einprozessorsystem, Mehrprozessoranlage, vernetzte Arbeitsplatzrechner usw.) und dem Kommunikationsmodell der Anwendung können sich aus dieser starren Zuordnung zu separaten Prozessen erhebliche Leistungseinbußen ergeben. Unter Umständen ist es sinnvoller, mehrere Objektverwalter zusammen in einen Prozeß zu laden, etwa zur Lastbalancierung oder um relativ teure Interprozeßkommunikation zu vermeiden. Das ist beispielsweise dann der Fall, wenn die Objektverwalter relativ klein sind oder zur Implementierung ihrer Dienstaufrufe häufig mit anderen Objektverwaltern kommunizieren müssen.

Das Prozeßmodell eines ConTract-Systems berücksichtigt diese Forderung nach Flexibilität dadurch, daß die beteiligten Server-Komponenten (ConTract Manager, Server zur Step-Ausführung, Objektverwalter) in beliebiger Weise auf Prozesse abgebildet und im Netzwerk konfiguriert werden können: Über eine entsprechende Schnittstelle der Systemverwaltung wird spezifiziert, welche Server zusammen in einem Prozeß ablaufen und wieviele Instanzen davon auf einem Knoten vorhanden sein sollen. So ist es beispielsweise möglich, alle Komponenten innerhalb eines einzigen Prozesses ablaufen zu lassen (etwa auf einem leistungsfähigen Einprozessorsystem). Oder aber jede Komponente (insbes. Step Computation Server und Objektverwalter) wird jeweils einem eigenen Prozeß zugeordnet, um z.B. auf einer Mehrprozessoranlage die im Skript spezifizierte Parallelität optimal zu nutzen.

5.4 Basiskommunikation

Aus dieser Flexibilität resultieren mehrere Anforderungen an den verwendeten Kommunikationsmechanismus. Da ein Anwendungsprogrammierer a priori nicht wissen kann (und eigentlich auch gar nicht zu wissen braucht), auf welchem Knoten und in welchem Prozeß ein aufgerufener Dienst (bzw. Server) lokalisiert ist, müssen alle Aufrufe über eine einheitliche Nachrichtenschnittstelle abgewickelt werden, mit der in gleicher Weise innerhalb *desselben Adreßraumes* (Prozeß), wie auch zwischen *verschiedenen Prozessen* und ebenso zwischen *entfernten, u.U. heterogenen Rechnern* kommuniziert werden kann. Das heißt, daß alle Dienste der Server-Komponenten ortstransparent adressiert werden. Für die ConTract-Steps ist das bereits durch das Programmiermodell gewährleistet, da sie unabhängig von der aktuellen Aufgabenverteilung über die Knoten definiert werden (Kap. 3). Da die Bindung eines Dienstaufrufes an die Adresse einer zuständigen Server-Komponente nun nicht mehr vom Anwendungsprogramm vorgenommen wird, ist ein Mechanismus erforderlich, der diese Aufgabe zur Laufzeit durchführt. Dabei sollte nicht nur eine Adreßumsetzung über rein statische Tabellen erfolgen, sondern es sollten dazu auch dynamische Kriterien herangezogen werden, wie z.B. die Auslastung und Verfügbarkeit möglicher Server-Komponenten sowie die zu erwartende Antwortzeit oder die Service-Kosten.

Für die Programmierung hat ein solcher Kommunikationsmechanismus eine Reihe wichtiger Vorteile: Durch die ortstransparente Adressierung vereinfacht sich die Erstellung von ConTract-Skripten und -Steps enorm, da für den Programmierer die Unterscheidung von lokalen und entfernten Aufrufen und insbesondere die Verwendung verschiedener Sprachmittel (lokaler bzw. entfernter Prozeduraufruf) entfällt. Gleichzeitig ist der resultierende Programmtext vollkommen unabhängig von Änderungen in der (Server-) Prozeßkonfiguration. Sogar Prozeßausfälle bleiben für die Anwendung bei hinreichender Server-Replikation transparent.

5.5 Implementierungsaspekte

Die Entwicklung des prototypischen ConTract-Systems *APRICOTS* findet in einer Hardware- und Software-Umgebung statt, die aus einer Reihe vernetzter Sun-Arbeitsplatzrechner besteht, auf denen eine Variante des Betriebssystems *Unix* und die relationalen Datenbanksysteme *TransBase* und *Ingres* laufen. Die verfügbaren Systeme erfüllen nur wenige der Anforderungen aus Kapitel 4. Dieser Abschnitt beleuchtet einige der Schwierigkeiten und die unternommenen Anstrengungen, um das beschriebene ConTract-Betriebsmodell dennoch soweit wie möglich zu realisieren.

5.5.1 Betriebssystemdienste und Kommunikationsmechanismen

Die Unix-Version *SunOS* enthält weder eine persistente Programmiersprache noch stabile Prozesse. Deshalb mußten der Laufzeitkern des ConTract-Systems (insbesondere der ConTract Manager) einschließlich der benötigten Mechanismen zur Ereignisverwaltung, Prozeßüberwachung usw. als normale Unix-Anwendungen implementiert werden.
Der RPC-Mechanismus in SunOS erwies sich als ungeeignet zur Abwicklung des Client-Server-orientierten Betriebsmodells aus 5.2. Beispielsweise ist es nicht möglich, daß der ConTract-Manager zu einem beliebigen Zeitpunkt sowohl Client gegenüber den beauftragten Anwendungs-Servern (SCS) als auch Server der ConTract-Anwendung ist. Das liegt hauptsächlich daran, daß die Client-Server-Rollenverteilung innerhalb eines SunOS-Prozesses nicht frei wählbar, sondern in der Programmstruktur des Clients (bzw. Servers) statisch festgelegt ist. Um das skizzierte asynchrone und ereignisorientierte Verarbeitungsmodell dennoch mit dem SunOS-RPC zu realisieren, hätte der Quell-Code des Betriebssystems geändert werden müssen. Zudem sind sog. billige Prozesse (threads) in SunOS nicht gleichzeitig mit dem RPC-Mechanismus benutzbar, womit beispielsweise die asynchrone

Skript-Abwicklung im ConTract Manager zu organisieren gewesen wäre. Deshalb wurde auch ein flexibler Basiskommunikationsmechanismus für die Aufrufe an die Step-Server bzw. Objektverwalter (auf der Basis von TCP/IP und *shared memory*) selbst entwickelt, der den herkömmlichen RPC [BiNe84] u.a. um die Konzepte „Transaktionen" und „Ereignisse" (asynchrones Versenden und Empfangen von Aufträgen) erweitert [Laub92].

5.5.2 Datenbank-Mechanismen

Auf der Datenbankseite sind die Gegebenheiten hinsichtlich der Realisierungsanforderungen nicht viel günstiger:
Zu Beginn der Implementierungsarbeiten war kein kommerzielles DBMS mit einer *prepare-to-commit*-Funktion erhältlich. In Kooperation mit einem Hersteller konnte jedoch ein System speziell für das ConTract-Projekt dahingehend erweitert werden. Allerdings floß diese Erweiterung nicht in eine Produktversion des DBMS ein – mit allen daraus resultierenden Konsequenzen für den Anwender. Inzwischen bieten mehrere Hersteller verteilter Datenbanksysteme *prepare*-Schnittstellen an (z.B. Ingres, Sybase), die bezüglich der Offenheit des Commit-Protokolls allerdings noch viele Wünsche offen lassen. Schwierigkeiten bestehen u.a. in der Bindung einer Transaktion an genau einen Prozeß und in dem fehlenden Zugang zum DBMS-Log, der z.B. für die Integration der transaktionsorientierten Protokollierung (undo/redo-Informationen) der Objektverwalter und der Ereignisprotokollierung des ConTract-Managers erforderlich ist. Deshalb mußte für die Realisierung der robusten Kontrollflußabwicklung ein separater Protokollierungs- und Wiederanlaufmechanismus implementiert werden [Masm92].

Besonders schwer fällt die restriktive Sperrverwaltung heutiger DBMS ins Gewicht, da ein ConTract-System hinsichtlich der Invariantenverwaltung stark auf die Kooperation mit dem darunterliegenden DBMS angewiesen ist, vorausgesetzt, man will nicht eine Datenverwaltungskomponente von Grund auf neu implementieren, sondern die Objektverwalter auf existierende Systemen aufsetzen.
Lange, stabile Sperren oder gar eine Weitergabe erworbener Sperren an eine Folgetransaktion (die u.U. in einem anderen Prozeß abläuft) machen nach Herstelleraussagen anscheinend größere Eingriffe in ein DBMS notwendig. Im Prototypsystem werden deshalb Existenzsperren (4.3) über den Umweg einer trickreichen Verwendung verteilter und replizierter Lesesperren simuliert. Da sich Sperren nicht weitergeben lassen, bleibt stattdessen nur die „mißbräuchliche" Verwendung derselben DBMS-Transaktion für mehrere Step-Transaktionen hintereinander, was natürlich höchstens in einer Prototypanwendung akzeptabel ist, aber keine befriedigende, allgemeine Lösung darstellt.

Daß schließlich geschachtelte Transaktionen noch in keinem DBMS-Produkt[2] implementiert sind, hängt offensichtlich mit der eben diskutierten Sperrproblematik zusammen. Daher mußte im APRICOTS-Prototypen die Strukturierung einer ConTract-Abwicklung in mehrere geschachtelte und verkettete Subtransaktionen ebenfalls durch selbst entwickelte Mechanismen angenähert werden.
Insgesamt wurde bei allen Eigenentwicklungen auf Verträglichkeit mit existierenden Standards (bzw. Vorstufen dazu) geachtet [DCE91, XOpen92, OSI-TP].

[2] Einzige Ausnahme ist das seit kurzem erhältliche Encina-Toolkit, einem neuen TP-System der Transarc Corporation.

5.5.3 Fazit

Trotz der aufgezählten Restriktionen konnte ein funktionsfähiges System gebaut werden, das die Kernmechanismen des ConTract-Modells prototypisch implementiert. Dieses Prototypsystem *APRICOTS* wird zur Zeit anhand einer Anwendung aus dem Fertigungsbereich evaluiert und weiterentwickelt.

6 Vergleich mit anderen Ansätzen

Der folgende Abschnitt beschäftigt sich mit dem Vergleich zwischen dem ConTract-Modell und anderen Ansätzen, wie SAGAs, Flex-Transaktionen , „triggers and transaction“ und Mehrschichttransaktionen, die sich ebenfalls mit der Abwicklung langlebiger Transaktionen befassen.

Wie sich zeigen wird, beinhaltet das ConTract-Modell viele Aspekte anderer Modelle. Das Ziel von ConTracts ist es, diese Eigenschaften zu nutzen und zu einem System zusammenzufassen, um dadurch eine Ablaufumgebung zu schaffen, die dem Anwendungsprogrammierer eine einfache Verwendung ermöglicht.

Das Modell beinhaltet insbesondere Maßnahmen zur robusten Verwaltung der Abläufe von ConTracts. Dies schließt die Definition eines Fehlermodells ein, um eine automatische Behandlung durch das System zu ermöglichen (z.B. wiederholte Ausführung im Fehlerfall, Migration). Ähnliche Ansätze lassen sich bei den SAGAs [GaSa87] finden, deren einzige Fehlerbehandlung jedoch im Zurücksetzen der SAGAs liegt (zumindest bis zu einem explizit definierten „savepoint“).

Zusammenfassend läßt sich sagen, daß das ConTract-Modell nicht nur eine Erweiterung des Transaktionskonzeptes auf Datenbanken definiert, sondern eine Ausführungsumgebung für verschiedene Anwendungsgebiete (wie z.B. der Einsatz im CIM) darstellt, um dabei das Konzept der Transaktionen anwendbar zu machen. Dies stellt einen großen Unterschied zu bisherigen Modellen dar, da diese sich weitgehend auf den Einsatz im Datenbankbereich beschränken.

6.1 Verteilte TP-Systeme, Föderierte und Verteilte DBMS

Die Architektur eines ConTract Systems hat viele Gemeinsamkeiten mit der verteilter Transaktionssysteme [GrRe92], wobei eine Komponente zur Kontrollflußverwaltung und transaktionsübergreifenden Ablaufsteuerung wie der ConTract Manager in heutigen TP-Monitoren nicht enthalten ist. Insbesondere sind nirgends so weitreichende Mechanismen zur Vorwärts-Recovery, Kontextverwaltung und transaktionsübergreifenden Synchronisation zu finden.

Das ConTract-Modell definiert nur Konzepte zur Abwicklung verteilter Anwendungen, trifft aber bewußt keine weiteren Annahmen über die Organisation der Datenhaltung in den Objektverwaltern. Dazu können sog. Föderierte [ShLa90] oder Verteilte Datenbanksystemen [CePe84] oder noch andere Verfahren zur Verwaltung verteilter Datenbestände zum Einsatz kommen, sofern sie die oben skizzierte, genormte Schnittstelle zur Transaktionsverwaltung anbieten. Unsere These ist allerdings, daß die hier beschriebenen Basismechanismen ihre Implementierung wesentlich vereinfachen.

6.2 Verwaltung von Transaktionen

Ähnlich wie in anderen Ansätzen wird im ConTract-Modell ein langlebiger Ablauf in kurze Ausführungseinheiten (Steps) zerlegt. Diese Ausführungseinheiten beinhalten jedoch keine Transaktionssemantik. Diese wird erst auf der Ebene des Skripts definiert, so daß eine flexible Abbildung von Steps auf Transaktionen möglich ist. Dies ist bei anderen Ansätzen nicht realisierbar.
Ein Beispiel aus der Fertigung soll die Flexibilität verdeutlichen:

step$_1$**:** erhitze_bauteil

step$_2$**:** biege_bauteil

Die vorgestellten Steps können einzeln als zwei Einzeltransaktionen ausgeführt werden, wenn z.B. für eine Einbrennlackierung ein Bauteil erhitzt werden muß oder ein Kupferstab gebogen wird. Soll jedoch ein Bauteil gebogen werden, welches nur im erhitzten Zustand bearbeitet werden kann, ist es notwendig, beide Steps zu *einer* Transaktion zusammenzufassen, da ein Wiederaufsetzen auf dem Zustand vor der Ausführung des Biegens nicht möglich wäre (bei Abbruch des Biegens). Ebenso wäre die Angabe des Zustands des Bauteiles nicht konsistent, da es als erhitzt vermerkt wäre, inzwischen aber abgekühlt ist.

Durch das ConTract-System wird es dem Skript-Programmierer ermöglicht, entsprechende Situationen einfach zu modellieren, da er die Transaktionsgrenzen um Steps frei definieren kann. Definiert er entsprechende Grenzen nicht, werden vom System automatisch Transaktionen für jeden Step generiert und die entsprechende Semantik garantiert (s.a. [XOpen92]).

6.3 Kontrollflußverwaltung

Der Kontrollfluß in einem ConTract wird explizit durch das Skript beschrieben. Im ConTract-Modell werden, ähnlich den Petri-Netzen der Flex-Transaktionen [LeEB90], Prädikat-Transitionsnetze zur Darstellung herangezogen. Diese Darstellung verdeutlicht auch, daß im Gegensatz zu den SAGAs [GaSa87], die Verwendung parallelisierender Konstrukte zugelassen ist. Der Ablauf des ConTracts ist jedoch abhängig vom jeweiligen Kontext, da Ablauf-Kontrollkonstrukte, die auf dem Kontext basieren, benutzt werden können (if, for, case ...). Das Modell der „triggers and transactions“ [DHL90] bietet einen ebenso mächtigen Mechanismus für einen variablen Ablauf, doch werden hier nur Regeln definiert, die eine bestimmte Aktion bei einem bestimmten Ereignis auslösen. Das heißt, die möglichen Ablaufzweige und die entsprechenden Kontrollkonstrukte werden nur implizit realisiert. Dies birgt die Gefahr in sich, daß der Programmierer sehr schnell die Übersicht über den Ablauf verliert und somit parallele Aktionen ausgelöst werden, die ungewollt konkurrieren oder Zyklen entstehen, die nur sehr schwer zu erkennen sind. Durch die Verwendung einer prozeduralen Sprache im ConTract-Modell wurde versucht, diese Gefahren zu umgehen. Die Realisierung des Modells erfolgt jedoch mit Hilfe einer Abbildung auf einen Mechanismus, der entsprechend den „triggers and transactions“ arbeitet.

6.4 Synchronisation

Die Invarianten im Skript stellen Prädikate dar (Eingangs- und Ausgangsinvarianten), die, ähnlich dem Ansatz von Korth/Speegle [KoSp88], zur Synchronisation nach pessimistischen (z.B. locks) oder optimistischen (z.B. check/revalidate [PRS88]) Prinzipien genutzt werden können. Im Gegensatz zu den „postconditions“ [KoSp88] werden die „Ausgangs-Invarianten“ nicht zur Erkennung der korrekten Abarbeitung eines Steps definiert, sondern zur Sicherung des konsistenten Ablaufs des ConTracts, indem sich die „Eingangs-Invarianten“ nachfolgender Steps auf diese beziehen und damit die Voraussetzung für die korrekte Bearbeitung durch den Step sicherstellen.

Andere Modelle lassen die Frage der Synchronisation über Transaktionsgrenzen hinaus offen („triggers and transaction“, Flex-Transaktionen) oder stellen heraus, daß ihre Modelle nur anwendbar sind, falls keine entsprechenden Maßnahmen getroffen werden müssen (SAGAs).

Mit der Definition von Isolationsbedingungen (Invarianten) durch die Anwendung verwendet das ConTract-Modell eine semantische Synchronisationsmethode. Werden diese nach dem „check/revalidate“-Prinzip verwaltet, so stellt dies einen Mechanismus dar, der eine weit größere Parallelität zuläßt als Verfahren, die auf dem Serialisierbarkeitskriterium basieren. Besonders hervorzuheben sind hierbei die Mehrschichttransaktionen [Weik91, WeSc92], da ConTracts in eingeschränkter Weise auf diese abgebildet werden können, wobei die Ebene der Skripte eine Schicht und die Ebene der Steps die darunter liegende Schicht darstellen.

6.5 Fehlerbehandlung

Im Gegensatz zu den meisten Ansätzen wird ein Fehlermodell definiert, um dadurch eine automatische Fehlerbehandlung durch das System zu ermöglichen. Obwohl im Fall der „triggers and transactions“ ein mächtiger Mechanismus zur Verfügung steht, wird dort kein Fehlermodell definiert, so daß wie bei anderen Modellen wohl nur der Abbruch der Applikation und die anschließende Kompensation [Gray81, GaSa87, KLS90] vorgesehen ist. Nur bei den SAGAs ist, durch die explizite Definition von „savepoints“, die Möglichkeit vorgesehen, die Kompensation zu beschränken und die Abarbeitung ab einem savepoint wieder aufzunehmen.

Das ConTract-Modell unterscheidet zwischen verschiedenen Fehlerklassen und ist in der Lage, mittels vom System überwachten Wiederholungs-Mechanismen oder Migration auf Fehler zu reagieren. Weiterhin wurde für die Behandlung von Invarianten-Fehlern eine Konflikt-Behandlung eingeführt, durch die der Skript-Programmierer die Möglichkeit hat, konflikt-behebende Maßnahmen („Conflict Resolution Steps“) anzugeben. Im Fall, daß eine Eingangsinvariante verletzt wird, startet das System diese Conflict Resolution Steps und versucht, nach erfolgreicher Ausführung, den eigentlichen Step erneut zu starten (Wiederholung nach Fehlerbehebung). Erst wenn die maximale Wiederholungszahl bzw. die Konflikt-Behandlung fehlschlägt, wird die Bearbeitung abgebrochen und die Kompensation der ausgeführten Steps eingeleitet.

6.6 Programmiermodell

Bisherige Ansätze beinhalten nur ein höchst einfaches Programmiermodell. Hierin wird vorausgesetzt, daß der Programmierer jeder Komponente transaktionsorientiert vorgeht. Dies bedeutet, daß alle notwendigen Aktionen (BOT, EOT, abort, commit ...) explizit zu programmieren sind. Der Vorteil des ConTract-Modells liegt darin, daß der Programmierer der Steps keine Transaktionssemantik zu berücksichtigen hat. Dies hat zur Folge, daß seine Ausführungseinheiten universell einsetzbar sind. Hieraus folgt zusätzlich die Freiheit des Skriptprogrammieres, die Steps beliebig mit einer Transaktionssemantik versehen zu können.

Da das Ausführungssystem alle Dienste ortstransparent zur Verfügung stellt (Step-Server, Ausführungsdienste usw.), müssen weder der Skriptprogrammierer noch der Stepprogrammierer besondere Maßnahmen vorsehen, um entsprechende Dienste adressieren zu können. Gerade in verteilten Umgebungen spielt dieser Aspekt eine große Rolle, da bisherige Mechanismen [BiNe84] nicht ortstransparent arbeiten, der Programmierer explizit Rechnerknoten zu adressieren hat und somit der Wechsel eines Dienstes auf einen anderen Knoten ohne Programmänderung (Adreßänderung) nicht möglich ist.

7 Zusammenfassung

Das ConTract-Modell mit den beschriebenen Verarbeitungsmechanismen (Kap.3) sowie dem Client-Server-orientierten Betriebsmodell (Kap.5) verallgemeinern das klassische Transaktionskonzept in drei wichtigen Bereichen:

(a) Durch die Erweiterung des Blickwinkels von einzelnen, kurzen DB-Zustandsübergängen auf *komplette Anwendungsvorgänge* konnten Verarbeitungsmechanismen definiert werden, die ein angemesseneres Ausführungsmodell für große verteilte Anwendungen auf gemeinsamen Ressourcen definieren als ACID-Transaktionen. Von den transaktionsübergreifenden Eigenschaften einer ConTract-Abwicklung sind insbesondere die robuste Kontrollflußverwaltung mit der automatischen Vorwärts-Recovery bei Systemfehlern zu erwähnen, außerdem die stabile Kontextverwaltung für zusammengehörige Verarbeitungsschritte, sowie das mehrstufige und dadurch äußerst flexible Programmiermodell für die ConTract-Erstellung.

(b) Das Invarianten-Konzept erweitert das in herkömmlichen DBMS fest eingebaute Korrektheitskriterium (Serialisierbarkeit) zu einem flexiblen, anwendungsgesteuerten Mechanismus für die transaktionsübergreifende Synchronisation, der situativ den Erfordernissen des jeweiligen Ablaufkontextes angepaßt werden kann. Langlebige Anwendungen bekommen damit ein mächtiges Instrument in die Hand, mit dem sie ein hohes Maß an Parallelität durch individuell zugeschnittene Isolations- und Konsistenzanforderungen realisieren können. Im Unterschied zu den meisten anderen Modellen werden Synchronisationskonflikte der Anwendung mitgeteilt, um ihr eine konstruktiven Auflösung derselben zu ermöglichen.

(c) Im konventionellen Ausführungsmodell klassischer DB-Transaktionen können die ACID-Eigenschaften nur für Anwendungen auf einer einzigen Ressourcenklasse, nämlich der *Datenbank*, genutzt werden. ConTracts verallgemeinern diesen engen Einsatzbereich dahingehend, daß ihre Eigenschaften für Anwendungen auf *beliebigen* gemeinsamen Ressourcen mit transaktionsgeschütztem Zugriff verfügbar sind. Für eine Anwendung heißt das konkret, daß sie über die Plattform eines ConTract-Systems auch auf sehr verschiedenartige Objektverwalter in einheitlicher Weise zugreifen kann. ConTracts bilden somit in Verbindung mit dem skizzierten Betriebsmodell transaktionsgeschützter Ressourcen-Zugriffe eine äußerst nützliche Grundlage für die Interoperabilität heterogener Objektverwalter.

Insgesamt definieren ConTracts eine Architektur für die zuverlässige Abwicklung einer breiten Klasse von verteilten Anwendungen [Wäch 91, Wäch 92]. Das Prototypsystem APRICOTS wird hauptsächlich anhand betrieblicher Abläufe aus der Fertigung evaluiert. Neben eigenen Untersuchungen [Bilg92] ist der Einsatz von APRICOTS zur Abwicklung geregelter arbeitsteiliger Büro- und Fertigungsabläufe [ReWe92, Reut92] sowie als Ausführungsmodell bzw. Laufzeitkern für Informationssysteme geplant, die in der objektorientierten Modellierungssprache T_{ROLL} spezifiziert wurden [JSHS91].

Zur Zeit laufen Vorbereitungen, um eine erste Version des ConTract-Prototypsystems innerhalb und außerhalb des DFG-Schwerpunktprogramms „Objektbanken für Experten" verfügbar zu machen. Die dabei gesammelten Erfahrungen werden in eine Weiterentwicklung von APRICOTS einfließen und sollen für zukünftige Arbeiten an der effizienten Auswertung und Verwaltung von Invarianten sowie für die Verfeinerung des Kompensationskonzeptes genutzt werden. Weitere Arbeitsschwerpunkte bilden in Zukunft vor allem Schnittstellen für die Anbindung herkömmlicher DBMS sowie anderer Objektverwalter. Daneben wird auch an Werkzeugen zur Unterstützung und Vereinfachung der ConTract-Programmierung gearbeitet. Besonderes wichtig für die Systemadministration und Ablaufsteuerung in einer realen Einsatzumgebung ist dabei eine geeignete graphische Darstellung der (Client-Server-) Systemkonfiguration sowie die Visualisierung eines ConTract-Ablaufes.

Literatur

[BiNe84] Birrell, A.D., Nelson, B.J.: *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, Vol. 2, No.1, Feb. 1984

[Bilg92] Bilger, A.: *Modellierung und Simulation betrieblicher Abläufe als ConTracts mit dem APRICOT-System.* Studienarbeit, Universität Stuttgart, Fakultät Informatik, 1992[3]

[BET91] Bernstein, P.A., Emberton, W.T., Trehan, V.: *DECdta- Digital's Distributed Transaction Processing Architecture.* Digital Technical Journal Vol.3(1), Winter 1991

[CePe84] Ceri, S., Pelagatti, G: *Distributed Databases. Principles and Systems.* MacGraw-Hill Book Company, 1984

[DCE91] *DCE Application Development Guide.* OSF DCE Version 1.0, Open Software Foundation, Cambridge, MA, 1991

[DHL90] Dayal, U., Hsu, M., Ladin, R.: *Organizing Long-Running Activities with Triggers and Transactions.* Proc. ACM SIGMOD 1990

[DaST87] Daniels, D.S., Spector, A., Thompson, D.: *Distributed Logging for Transaction Processing.* Proc. ACM SIGMOD 1987

[Davi78] Davies, C.T.: *Data Processing Spheres of Control.* IBM Systems Journal, Vol. 17(2), 1978, pp. 179-198

[EMS91] Eppinger, J.L., Mummert, L.B., Spector, A.Z. (Hrsg.): *Camelot and Avalon - A Distributed Transaction Facility.* Morgan Kaufmann Publishers, 1991.

[Eitl92] Eitler, M.: *Eine graphische Oberfläche für die interaktive Steuerung langlebiger verteilter Abläufe.* Diplomarbeit (in Vorbereitung), Universität Stuttgart, Fakultät Informatik, 1992

[3] Die referenzierten studentischen Arbeiten und weitere interne Berichte über das ConTract-Modell und das APRICOT-System sind über das Internet per *anonymous ftp* frei erhältlich:

Rechner: `ftp.informatik.uni-stuttgart.de`
Internet: `129.69.211.1`
Benutzer: `anonymous`
Passwort: beliebig, aber nicht leer
Pfad: `/pub/APRICOTS/...` (Postscript bzw. DVI-Format).

[EpDi92] Eppinger, J.L., Dietzen, S.: *Encina: Modular Transaction Processing.* Proc. Spring CompCon, San Francisco, California, USA, 1992

[GaSa87] Garcia–Molina, H., Salem, K.: *Sagas.* Proc. ACM SIGMOD 1987

[Gray81] Gray, J.: *The Transaction Concept: Virtues and Limitations.* Proc. VLDB, Cannes, Sept. 1981

[GrRe92] Gray, J., Reuter, A.: *Transaction Processing Systems: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, 1992

[HaeRe83] Härder, T., Reuter, A.: *Principles of Transaction-Oriented Database Recovery.* ACM Computing Surveys, 15(4), 1983

[JSHS91] Jungclaus, R., Saake, G., Sernadas, C.: *Object-Oriented Specification of Information Systems: The TROLL Language.* Technical Report 91-04, TU Braunschweig, 1991

[Kim84] Kim, W.: *Highly Available Systems for Database Applications.* ACM Computing Surveys 16(1), 1984

[KlRe88] Klein, J., Reuter, A.: *Migrating Transactions.* Proc. Workshop on the Future Trends of Distributed Computing Systems, Hong Kong, IEEE, Sept. 1988

[KLS90] Korth, H.F., Levy, E., Silberschatz, A.: *A Formal Approach to Recovery by Compensating Transactions.* Proc. VLDB 1990, pp. 95-106

[KoSp88] Korth, H.F., Speegle, G.D.: *Formal Model of Correctness Without Serializability.* Proc. ACM SIGMOD 1988

[Kübl90] Kübler, D.: *Ein Werkzeug zur Modellierung und Überwachung verteilter Systeme.* Diplomarbeit Nr. 681, Universität Stuttgart, Fakultät Informatik, August 1990

[Laub92] Laube, D.: *Ein transaktionsorientierter Kommunikationsmechanismus fuer das verteilte Transaktionssystem APRICOTS.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, 1992

[LeEB90] Leu, Y., Elmagarmid, A., Boudriga, N.: *Specification and Execution of Transactions for Advanced Database Applications.* Purdue University Technical Report, CSD-TR 1030, Oct. 1990

[Masm90] Masmanidis, I.: *Syntax und Compiler einer Sprache zur Beschreibung paralleler und verteilter Abläufe.* Studienarbeit, Universität Stuttgart, Fakultät Informatik, August 1990

[Masm92] Masmanidis, I.: *Robuste Kontrollflußverwaltung für langlebige und verteilte Datenbank-Anwendungen.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Mai 1992

[Moss81] Moss, E.J.B.: *Nested Transactions: An Approach to Reliable Computing.* Ph.D. Thesis, M.I.T., Report MIT-LCS-TR-260, 1981

[OSI-TP] *Open Systems Interconnection: Distributed Transaction Processing.* Draft Proposal 10026-1, -2, -3 (Model, Service Definition, Protocol Specification), International Standardization Organization, 1988.

[PRS88] Peinl, P., Reuter, A., Summer, H.: *High Contention in a Stock Trading Database - A Case Study.* Proc. ACM SIGMOD 1988

[Reut89] Reuter, A.: *ConTracts: A Means for Extending Control Beyond Transaction Boundaries* Proc. 2nd. Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1989

[Reut92] Reutzel, C.: *Transaktionsgestützte Abwicklung von Abläufen in geregelten arbeitsteiligen Anwendungen.* Diplomarbeit, Fachbereich Informatik, Universität Erlangen-Nürnberg, 1992

[ReSc91] Reuter, A., Schmidt, U.: *Transactions in Manufacturing Applications.* Proc. 4th. Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1991

[ReWe92] Reinwald, B., Wedekind, H.: *Integrierte Aktivitäten- und Datenverwaltung zur systemgestützten Kontroll- und Datenflußsteuerung.* Informatik Forschung und Entwicklung, Heft 1, 1992

[RoPa90] Rothermel, K., Pappe, S.: *Open Commit Protocols for the Tree of Processes Model.* Proc. 10th Int. Conf. on Distributed Computing Systems, pp. 236-244, 1990

[Schm92] Schmitd, U.: *A Framework for Automated Error Recovery in Flexible Manufacturing Systems.* Proc. 2nd Int. Conf. on Automation, Robotics and Computer Vision, Singapure, Sept. 1992

[ShLa90] Sheth, A., P., Larson, J., A.: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Computing Surveys, Vol. 22, No. 3, Sept. 1990

[Wäch91] Wächter, H.: *ConTracts - A Means for Improving Reliability in Distributed Computing.* Proc. IEEE Spring CompCon, San Francisco, Feb. 1991

[Wäch92] Wächter, H.: *Eine Architektur für verteilte Anwendungen mit hohen Zuverlässigkeitsanforderungen.* Dissertation (in Vorbereitung), Fakultät Informatik der Universität Stuttgart, 1992

[WäRe90] Wächter, H., Reuter, A.: *Grundkonzepte und Realisierungsstrategien des ConTract-Modells.* Informatik Forschung und Entwicklung, Heft 5, 1990

[WäRe92] Wächter, H., Reuter, A.: *The ConTract Model.* In: A.K. Elmagarmid (Hrsg.), *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, CA, 1992

[Wahl91] Wahl, P.: *Interpretative Abwicklung von Prädikat-Transitionsnetzen.* Studienarbeit, Universität Stuttgart, Fakultät Informatik, 1991

[Walt82] Walter. B.: *A Robust and Efficient Protocol for Checking the Availability of Remote Sites.* Comp. Networks No. 6, pp. 173-188, 1982

[WeSc92] Weikum, G., Schek, H.-J.: *Concepts and Applications of Multilevel Transactions and Open Nested Transactions.* In: A.K. Elmagarmid (Hrsg.), *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, CA, 1992

[Weik91] Weikum, G.: *Principles and Realization Strategies of Multilevel Transaction Management.* ACM TODS, Vol. 16(2), 1991

[XOpen92] X/Open: *Distributed Transaction Processing Reference Model.* X/Open Document No. XO/GUIDE/91/020

[Zell92] Zellmer, O.: *Ein Werkzeug zur Definition und Verwaltung transaktionsorientierter Client-Server-Anwendungen.* Studienarbeit, Universität Stuttgart, Fakultät Informatik 1992

Hinweis:

Die referenzierten studentischen Arbeiten u.a interne Berichte über das ConTract-Modell und das APRICOT-System sind über das Internet per *anonymous ftp* frei erhältlich:

Rechner: `ftp.informatik.uni-stuttgart.de`; Internet: `129.69.211.1`
Benutzer: `anonymous` (beliebiges, aber nicht leeres Passwort)
Pfad: `/pub/APRICOTS/...` (Postscript bzw. DVI-Format).