Universität Stuttgart
Software–Labor
Projekt 1.1:
Workflow–Management–Systeme
Breitwiesenstraße 20–22
D-70565 Stuttgart

Fakultätsbericht Nr. 1996/18
Software–Labor Bericht Nr. SL–4/96
CR–Klassifikation H.2.0, H.2.4

Der Einsatz von Workflow–Transaktionen in FlowMark *

Hubert Bildstein
Stefan Schreyjak
Stefan.Schreyjak@informatik.uni-stuttgart.de

20. Dezember 1996

*Diese Arbeit wird von der IBM Deutschland Entwicklung GmbH und dem Ministerium für Wissenschaft und Forschung, Baden Württemberg, unterstützt.

Zusammenfassung

In diesem Bericht wird skizziert, wie das Konzept der Workflow–Transaktionen im Workflowsystem FlowMark angewendet werden kann. Dazu wird zuerst ein einfaches funktionales Modell des Workflowsystems FlowMark Version 2.1 aufgestellt, das auf Veröffentlichungen über FlowMark basiert. Anschließend werden die wichtigsten Begriffe des Konzepts der Workflow–Transaktionen erläutert. Die zur Realisierung von Workflow–Transaktionen notwendige Technologien, CORBA und OTS, werden kurz beschrieben. Ausgehend von dem funktionalen Modell werden dann die notwendigen Änderungen erörtert, die in FlowMark–Systemkomponenten vorgenommen werden müssen. Verschiedene Lösungsansätze für mehrere identifizierte Probleme werden vorgestellt und abgewägt. Im Anschluß wird ein Beispielprogramm vorgestellt, mit dem SOM und OTS von uns evaluiert worden sind. Im Beispiel wird ein einfacher Kontoserver realisiert, der im Rahmen einer Überweisungstransaktion angesprochen wird. Ausführliche Codefragmente vervollständigen die Erläuterungen zum Beispielprogramm.

Inhaltsverzeichnis

1 Aufbau und Funktionsweise von FlowMark

In diesem Kapitel wird beschrieben, wie FlowMark aufgebaut ist und wie die Komponenten miteinander arbeiten. Das Wissen über den Aufbau und die Funktionsweise ist den bisherigen Veröffentlichungen (vor allem in [AGK 95]) entnommen worden. Mit diesem Kapitel soll die Basis für das Verständnis der weiteren Kapitel gelegt werden.

1.1 Die Systemarchitektur von FlowMark V2.1

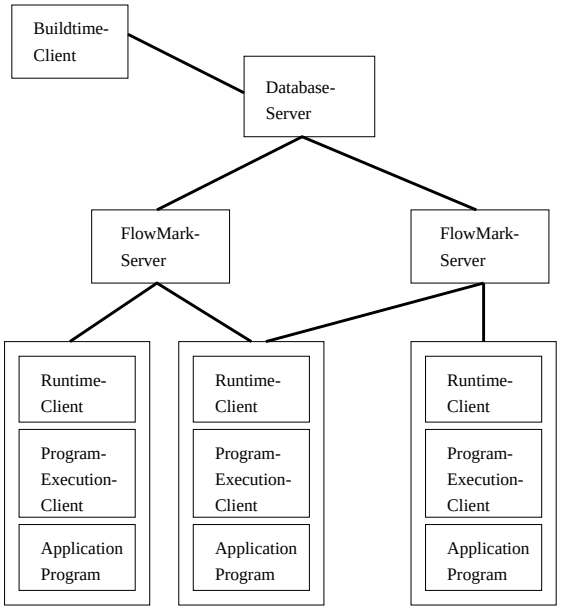

Abbildung 1: Die Systemarchitektur von FlowMark V2.1

Die Architektur von FlowMark in der Version 2.1 ist in der Abbildung 1 dargestellt. Die Kästen sind Komponenten von FlowMark. Die dicken Linien symbolisieren mögliche Netzwerkverbindungen zwischen Rechnerknoten. Das Workflowsystem besteht aus einer zentralen Datenbank, aus einem oder mehreren FlowMark–Servern und aus Runtime–Clients, die die Benutzerschnittstelle bilden. Ein Buildtime–Client dient zur Modellierung von Workflows. Ein Program–Execution–Client ruft die Anwendungsprogramme auf.

Die zentrale Komponente ist der Datenbankserver. Dort werden alle Prozeßdefinitionen und die Daten zur Steuerung der Prozesse gespeichert. Die FlowMark–Server greifen in der Rolle eines Datenbank–Clients auf die Workflow–Datenbank zu. Außer der Buildtimekomponente und den FlowMark–Servern dürfen keine anderen Programme direkt auf die Datenbank zugreifen. In der Version 2.1 wird das objektorientierte Datenbanksystem „Objectstore“ eingesetzt. In späteren Versionen soll die relationale Datenbank DB2 verwendet werden.

Die Buildtimekomponente ist ein Programm zur grafischen Modellierung von Geschäftsprozessen (Workflows) und Organisationsstrukturen. Die erstellten Workflows können durch Animation getestet werden.

Im FlowMark–Server ist die Workflow–Engine enthalten, die Workflows ausführt. Der Server bestimmt die nächsten ausführbaren Aktivitäten und verteilt sie anhand des Organisationsmodells und der Workflowdefinition auf die Arbeitslisten der Bearbeiter. Anfragen an die Datenbank werden über einen FlowMark–Server abgewickelt.

Ein Runtime–Client bildet die Benutzeroberfläche des Workflowsystems für den Bearbeiter. Er kann damit auf seine Arbeitsliste zugreifen, Aktivitäten und Prozesse starten und den Zustand eines laufenden Prozesses überwachen.

Der Program–Execution–Client wird vom FlowMark–Server zum Start eines Anwendungsprogramms im Rahmen einer Aktivität benutzt. Der Execution–Client informiert den FlowMark–Server über die Beendigung der Aktivität. Beliebige Anwendungsprogramme können innerhalb einer Aktivität ausgeführt werden.

1.2 Funktionsweise von FlowMark

Im folgenden wird exemplarisch beschrieben, wie eine Aktivität ausgeführt wird. Die Benutzer wählt eine Aktivität auf seiner Arbeitsliste zum Start aus. Der Runtime–Client schickt dem FlowMark–Server eine entsprechende Nachricht. Dieser führt die entsprechenden Operationen zum Start einer Aktivität aus und aktualisiert den Zustand auf der Datenbank. Dann wird der Program–Execution–Client angewiesen, das entsprechende Anwendungsprogramm zu starten. Der Runtime–Client wird über den Start des Programms informiert. Nach Beendigung des Anwendungsprogramms benachrichtigt der Program–Execution–Client den FlowMark–Server über das Ereignis. Der Server führt die Operationen zum Beenden einer Aktivität auf der Datenbank aus und informiert den Runtime–Client vom Ende der Aktivität. Der Server berechnet dann die nächsten auszuführenden Aktivitäten und verteilt sie auf die Arbeitslisten.

Das Funktionsprinzip der Workflow–Engine basiert darauf, auf Nachrichten aller von ihm abhängigen Clients zu warten, die entsprechenden Operationen auf der Datenbank auszuführen und eventuell weitere Bearbeitungsschritte mittels Nachrichten an die Clients anzustoßen.

1.3 Ein funktionales Modell von FlowMark

Wir können also vereinfachend annehmen, daß die Workflow–Engine sukzessive Nachrichten aus einer persistenten Warteschlange liest. Auf Nachrichten muß im allgemeinen mit einem Zustandsübergang des Workflows bzw. der Aktivitäten im Workflow reagiert werden. Um die Workflow–Engine vor inkonsistenten Daten zu schützen, die bei einem Absturz der Engine oder beim Auftreten eines anderen Fehlers erzeugt werden könnten, muß dieser Zustandsübergang durch eine Transaktion vor Fehlern geschützt werden. Das Warten auf Nachrichten und das Verarbeiten der Nachrichten im Rahmen von Transaktionen kann in einem einzigen Thread stattfinden.

2 Zugrundeliegende Technologie und Konzepte

In diesem Kapitel sollen zuerst die Grundbegriffe von Workflow–Transaktionen definiert werden. Danach werden die Konzepte und die Technologie von CORBA und OTS kurz beschrieben, um eine Verständigungsbasis zu schaffen.

2.1 Workflow–Transaktionen

Das Konzept der Workflow–Transaktionen wird ausführlich in [SB96b] beschrieben. Hier sollen nur nochmals die wesentlichen Begriffe und Konzepte erwähnt werden.

Eine *Sphäre* ist eine Menge von Aktivitäten in einem Workflow. Eine Sphäre muß keine Zusammenhangskomponente im Aktivitätennetz bilden. Eine Sphäre wird zur Modellierungszeit des Workflows spezifiziert. In der Abbildung 2 ist eine Sphäre in ein Aktivitätennetz eingezeichnet.

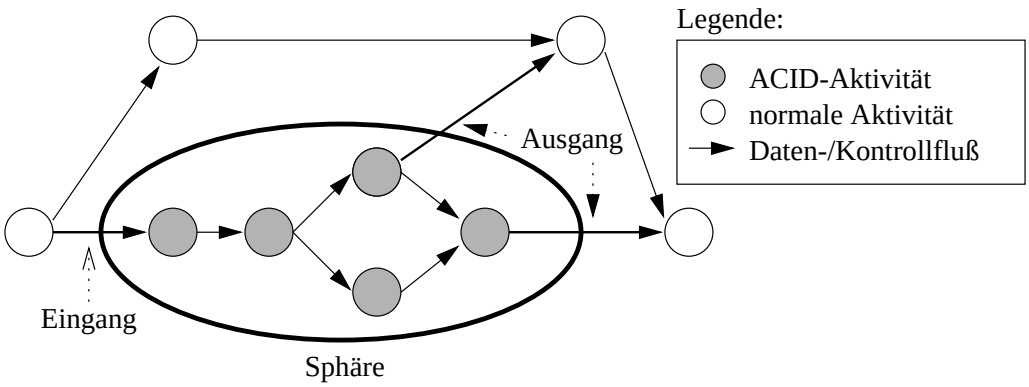

Abbildung 2: Eine Sphäre in einem Aktivitätennetz

Als *Eingang* einer Sphäre wird der Kontroll– bzw. Datenfluß bezeichnet, der von einer Aktivität außerhalb der Sphäre zu einer Aktivität innerhalb der Sphäre führt. Entsprechen wird als *Ausgang* der Kontroll– bzw. Datenfluß definiert, der von einer Aktivität innerhalb der Sphäre zu einer Aktivität außerhalb der Sphäre führt. Eine Sphäre kann mehrere Ein– und Ausgänge besitzen. In der Abbildung 2 hat die Sphäre einen Eingang und zwei Ausgänge.

In einer *Workflow–Transaktion* wird eine Menge von Aktivitäten (d. h. eine Sphäre im Workflowmodell) als eine ACID–Transaktion ausgeführt. Um dies im Rahmen eines Workflowsystems machen zu können, müssen bestimmte Anforderungen an die Aktivitäten gestellt werden.

Als *ACID–Aktivität* wird eine Aktivität bezeichnet, die an einer Workflow–Transaktionen teilnehmen kann. Sie muß die Daten, auf die sie zugreift, entweder selbst als Resource Manager verwalten oder einen externen Resource Manager zum Zugriff verwenden. Sie muß eine OTS–konforme Schnittstelle besitzen.

Mit einer Sphäre wird festgelegt, welche Aktivitäten in einem Geschäftsprozeß an einer Workflow–Transaktion teilnehmen. Folgende strukturelle Bedingungen müssen von einer Sphäre erfüllt werden [SB96b].

- Innerhalb einer Sphäre darf keine Subprozeßaktivität vorkommen. Blöcke sind dagegen erlaubt.
- Alle Aktivitäten innerhalb einer Sphäre müssen ACID–Aktivitäten mit einer OTS-konformen Schnittstelle sein. Für die Aktivitäten in einem Block innerhalb der Sphäre muß dieselbe Bedingung gelten.
- Es darf keinen Pfad von einem Ausgang einer Sphäre auf einen Eingang derselben Sphäre geben.
- Sphären dürfen sich nicht überlappen. Geschachtelte Sphären sind erlaubt.

2.2 OMA und CORBA

Die Object Management Group (OMG) ist ein Firmenkonsortium, das im April 1989 durch eine Reihe namhafter Hersteller gegründet worden ist. Die OMG hat sich zum Ziel gesetzt, grundlegende Standards im Bereich der objektorientierten Softwaretechnologie zu entwickeln.

Unter dem Begriff *Object Management Architecture* (OMA) [Obj90] wurde eine Referenzarchitektur zur Realisierung von verteilten objektorientierten Anwendungen vorgestellt. Die OMA basiert auf einem verallgemeinerten Objektmodell, mit dem ein möglichst breites Feld bereits bestehender und zukünftiger Objektmodelle beschrieben werden kann. In der Architektur wird eine gemeinsame Terminologie entwickelt, mit der die verschiedenen konkreten Objektmodelle beschrieben werden können. In Abbildung 3 sind die wesentlichen Komponenten der OMA Architektur abgebildet.

Der *Object Request Broker* (ORB) agiert als Kommunikationsmittler zwischen Objekten und bildet so die Basis für die Interoperabilität von Objekten in heterogenen Netzwerken. Er stellt die Infrastruktur bereit, damit Objekte plattformübergreifend und unabhängig vom verwendeten Objektmodell (bzw. der Programmiersprache) miteinander über Methodenaufrufe kommunizieren können.

Die *Object Services* stellen eine allgemeine Laufzeitumgebung bereit, die von einem breiten Spektrum von Objekten genutzt werden können, um ihre jeweiligen Aufgabe zu erfüllen. Dazu werden in den Object Services verschiedene Funktionen standardisiert. So sorgen Teile des Object Services für die physische Speicherung der Objekte. Andere Teile verwalten Klassendefinitionen und deren Verhältnisse untereinander. Es gibt Dienste, die Instanzen von Klassen erzeugen, aufrufen, kopieren, migrieren und löschen können. Objekte oder Klassen können durch Suchbedingungen gefunden werden. Möglichkeiten zur Sicherung der Integrität in einzelnen Objekten und der Konsistenz zwischen mehreren Objekten werden angeboten. Object Services müssen auf jeder Plattform verfügbar sein.

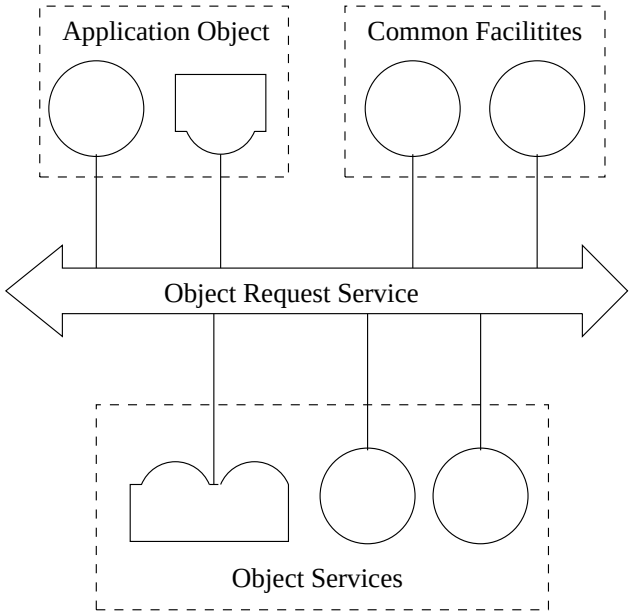

Abbildung 3: Die Komponenten der Objekt Management Architecture (OMA)

Die *Common Facilities* müssen dagegen nur optional verfügbar sein. Sie definieren Objektschnittstellen für typische Funktionen in speziellen Anwendungsbereichen. Beispiele für solche Common Facilities sind Schnittstellen zum Drucken von Dokumenten, zum Versenden von Email oder zum Zugriff auf Datenbanken. Anwendungen können diese Funktionen in einer standardisierten Weise nutzen.

Die *Application Objects* sind die Anwendungen in der klassischen Sichtweise. Sie nutzen die in den anderen Komponenten der OMA angebotenen Dienste und kombinieren diese für ihren speziellen Anwendungsfall.

Die *Common ORB Architecture and Specification* (CORBA) [Obj93] definiert ein Rahmenwerk für unterschiedliche ORB–Implementierungen, damit diese ihre Dienste mit derselben Schnittstelle anbieten können. Ein Client nutzt die Dienste des ORBs, um mit einer Objektinstanz ortstransparent zu kommunizieren (siehe Abb. 4). Die standardisierten Schnittstellen des ORBs und die standardisierte Beschreibung der Schnittstelle der Object Implementation durch eine Interface Definition Language (IDL) erlauben den Austausch der Object Implementation bzw. des Clients durch eine andere Portierung, d. h. ein unterschiedlich implementiertes Objekt mit gleicher Funktionalität.

Das *System Object Model* (SOM) zusammen mit der verteilten Variante *Distributed SOM* (DSOM) ist die Implementierung des CORBA Standards durch die Firma IBM. In der Version 3.0 sind zusammen mit dem ORB auch verschiedene Object Services realisiert. Insbesondere der *Object Transaction Service* (OTS) ist Gegenstand dieses Berichts.

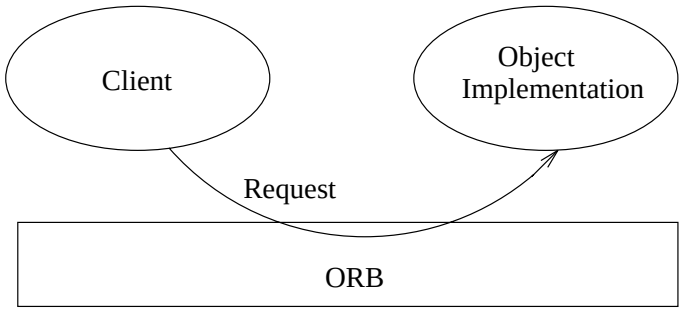

Abbildung 4: Ein Client benutzt den ORB zum Aufruf einer Methode in einer Object Implementation

2.3 Der Object Transaction Service der OMG

OTS ist die Spezifikation eines Object Services im Rahmen der OMA durch die OMG. OTS unterstützt die Abwicklung von Transaktionen in verteilten Client/Server Anwendungen. Ein Transaktions–Dienst definiert Schnittstellen zwischen Objekten. Er unterstützt nicht direkt die ACID–Eigenschaften einer Transaktion, sondern implementiert einen Protokollautomaten, der die an der Transaktion teilnehmenden Objekte so koordiniert, daß die Transaktion die ACID Eigenschaft garantieren kann. Die teilnehmenden Objekte können weitere Object Services der OMG nutzen, wie den Persistence Object Service (POS) oder den Concurrency Control Service, um die ACID–Eigenschaften zu erreichen. Die Aufgabe von OTS ist die Verwaltung des Transaktionskontexts und die Durchführung des Zwei–Phasen–Commit–Protokolls (2PC). OTS koordiniert dabei alle Teilnehmer einer Transaktion so, daß sie mit einem einheitlichen Ergebnis – erfolgreich oder nicht erfolgreich – die Transaktion abschließen. OTS stößt dazu ein *commit* oder ein *rollback* in den teilnehmenden Objekten an. Die Objekte müssen diese Funktionen selbst ausführen. Gegebenenfalls müssen diese Objekte auch ein selbständiges Recovery ausführen können.

Folgende Objekte spielen bei einer Transaktion, die OTS konform ausgeführt wird, eine Rolle. In Abbildung 5 ist das Zusammenspiel der OTS Objekte dargestellt.

- Transactional Client:
  Ein Transactional Client ist ein Anwendungsprogramm, das Aufrufe an ein Transactional Object macht, die durch eine Transaktion geschützt werden sollen. Der Initiator einer Transaktion ist zwangsläufig ein Transactional Client, es gibt aber auch Clients, die nicht Initiator einer Transaktion sind.

- Transactional Object:
  Ein Transactional Object ist ein Objekt, das bei einem Aufruf im Rahmen einer Transaktion vom Transaktionskontext in irgendeiner Weise beeinflußt wird. Alle Teilnehmer an einer Transaktion sind Transactional Objects. Ein Objekt ist „transactional“, aber nicht „recoverable“, wenn das Objekt seinen Zustand

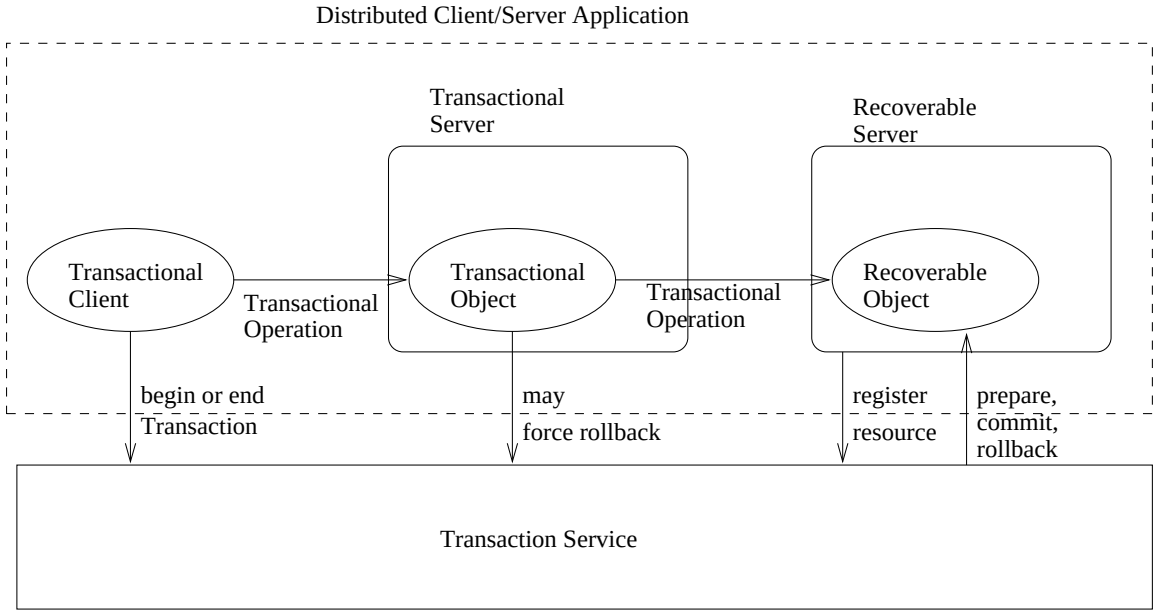

Abbildung 5: Das Zusammenspiel der in OTS definierten Komponenten

nicht selbst wie ein Resource Manager verwaltet, sondern z. B. über externe Recoverable Objects.

- Recoverable Object:
  Ein Recoverable Object ist per Definition ein Transactional Object. Es verwaltet seine internen Daten so, daß es jederzeit auf Anforderung des Transaktions–Dienstes entweder ein Commit oder ein Rollback für einen bestimmten Transaktionskontext durchführen kann. Ein Recoverable Object muß am Terminierungsprotokoll des Transaktions–Diensts teilnehmen (typischerweise ein 2PC–Protokoll).

- Transactional Server:
  Der transaktionale Server besitzt mindestens ein Transactional Object, hat aber keinen eigenen wiederherstellbaren Zustand. Er kann den Abbruch einer Transaktion erzwingen, nimmt aber nicht am 2PC teil.

- Recoverable Server:
  Ein Recoverable Server beinhaltet mindestens ein Recoverable Object. Er registriert seine Recoverable Objects mittels von Resource Objects beim Transaktions–Dienst als Teilnehmer an Transaktionen.

- Resource Object:
  Ein Resource Object ist das Bindeglied zwischen einem Recoverable Object und einer Transaktion. Es muß dafür sorgen, daß die Ressourcen des Recoverable

Objects entsprechend der Entscheidung des 2PC–Koordinators verwaltet werden. Ein Recoverable Object kann mehrere Resource Objects besitzen, falls es an mehreren Transaktionen gleichzeitig teilnimmt, ein Resource Object kann aber nur zu einer Transaktion gehören (siehe Abbildung 6).

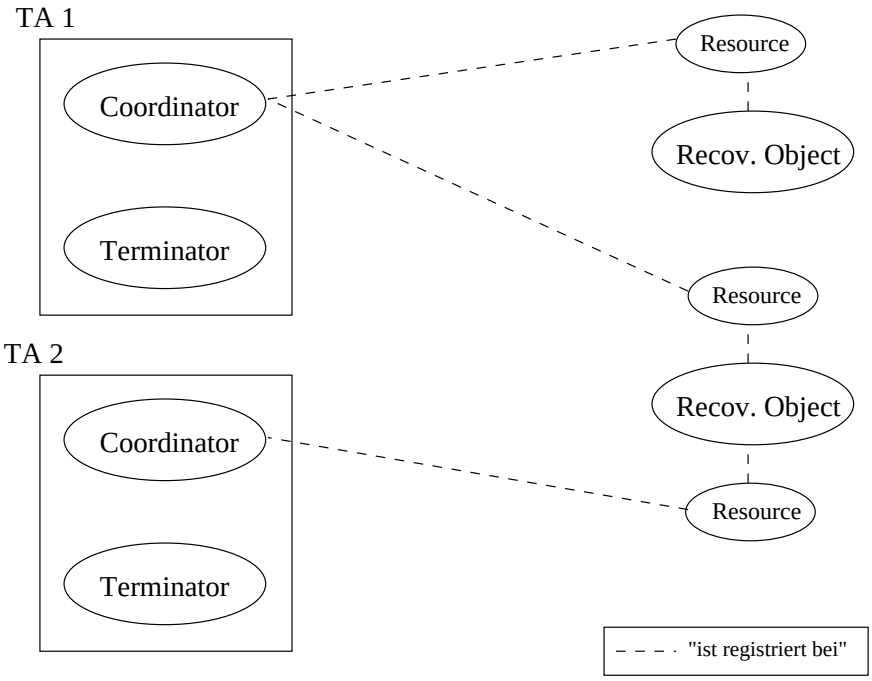

Abbildung 6: Zusammenhang Resource – Coordinator

Der Transaktions–Dienst besteht aus Objekten diverser Klassen mit verschiedenen Aufgaben. Die wichtigsten davon sind:

- Factory:
  Die *create*–Methode eines Objektes dieser Klasse erzeugt einen neuen Transaktionskontext (top–level) und liefert ein Control–Objekt zurück. Über die Factory wird eine Transaktion initiiert. Alternativ dazu kann das Current–Objekt benutzt werden (siehe unten).

- Control:
  Ein Control–Objekt repräsentiert den Transaktionskontext. Es bietet die Methoden *get_terminator* und *get_coordinator* an. Diese liefern Objekte der folgenden zwei Klassen.

- Terminator:
  Diese Klasse bietet dem Client die Methoden zum Beenden einer Transaktion, *commit* und *rollback*.

- Coordinator:
  Beim Coordinator–Objekt werden die Resource–Objekte einer Transaktion registriert (Methode *register_resource*). Alle Recoverable Objects, die ihre Resource–Objekte beim Coordinator–Objekt der Transaktion registriert haben, nehmen am Terminierungsvorgang der Transaktion teil. Weiterhin bietet diese Klasse eine Anzahl von Methoden zur Abfrage von Statusinformation an.
  Coordinator und Terminator steuern zusammen den Terminierungsvorgang (2PC).

Ein Client, der eine Transaktion beginnen will, erzeugt sich ein Factory–Objekt und führt die *create*–Methode darauf aus (Abb. 7 links). Der Verweis auf das *Control*–Objekt, den diese Methode zurückliefert, wird gespeichert. Um eine Transaktion zu beenden, ruft man *get_terminator* auf dem Control–Objekt auf. Man erhält eine Referenz auf das Terminator–Objekt der Transaktion und kann auf diesem *commit* oder *rollback* aufrufen.

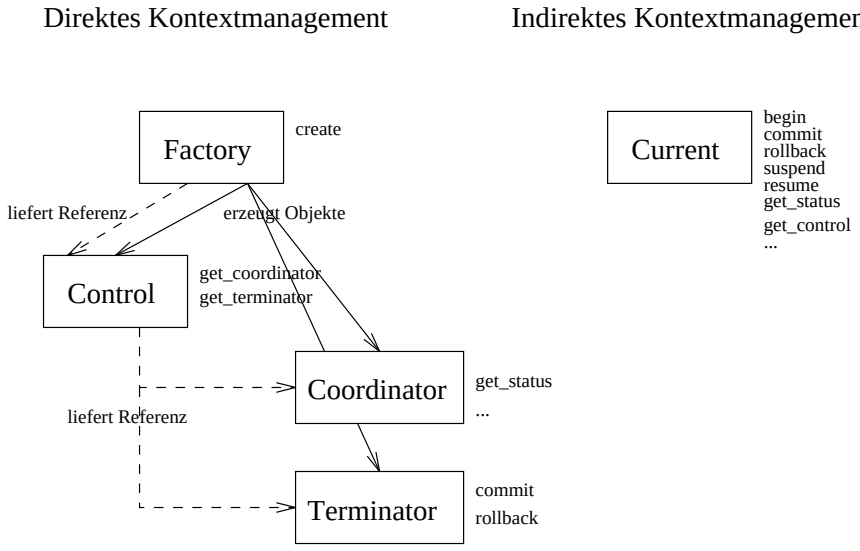

Abbildung 7: Steuerung einer Transaktion über Factory – Current

Alternativ dazu kann ein Client eine Transaktion über ein Objekt der Klasse *Current* beginnen. Ein Objekt dieser Klasse bietet zum Start einer Transaktion die *begin*–Methode, zum Beenden *commit* und *rollback*, das Abfragen von diversen Statusinformationen und weitere Methoden an (Abb. 7 rechts). Die Verwendung eines Current–Objektes anstatt der oben beschriebenen Objekte ist u. a. eine Vereinfachung für den Client–Programmierer, und beide Vorgehensweisen sind bis zu einem gewissen Grad äquivalent. Auch der Aufruf von *begin* auf dem Current–Objekt erzeugt natürlich die Control–, Coordinator– und Terminator–Objekte, diese müssen aber im Client–Code nicht explizit berücksichtigt werden. Die Verwendung der einen oder anderen Vorgehensweise impliziert aber auch eine bestimmte Methode der Kontext–Propagierung.

Wie wird einem Transactional Object der zugehörige Transaktionskontext mitgeteilt, wenn ein Client nach der Initiierung einer Transaktion dieses aufruft? Hier gibt es zwei Möglichkeiten: Die *implizite Propagierung* und *explizite Propagierung*.

Welche Propagierungsmethode verwendet werden muß, bestimmt das Transactional Object. Es ist entweder auf die eine oder auf die andere Methode ausgelegt. *Explizite Propagierung* bedeutet, daß jede Methode des Transactional Objects, die unter Transaktionsschutz ablaufen soll, einen explizit aufgeführten Parameter in Form eines *Control*–Objekts besitzt. Der Client übergibt einen Verweis auf dieses Objekt explizit bei jedem Aufruf. Das Control–Objekt repräsentiert den jeweiligen Transaktionskontext.

Implizite Propagierung bedeutet, daß die Übermittlung des Transaktionskontextes durch das System übernommen wird.

Die Verwendung eines Current–Objektes zur Transaktionsinitiierung und Terminierung nennt man *indirektes Kontextmanagement*, da der Transaktionskontext nicht in Form des Control–Objektes direkt vom Client verwaltet wird. Entsprechend liegt bei Verwendung einer Factory *direktes Kontextmanagement* vor. Wird eine Transaktion mittels eines Current–Objekts gestartet, ist automatisch der Programm–Thread, der dies durchführt, mit dem Transaktionskontext assoziiert. Dies ist die Voraussetzung für implizite Propagierung. Die explizite Propagierung hat als Voraussetzung das Vorhandensein des Control–Objekts der Transaktion, in deren Kontext eine Operation durchgeführt werden soll. Dies ist bei direktem Kontextmanagement gegeben.

Typischerweise wird man die Kombinationen *indirektes Kontextmanagement – implizite Propagierung* und *direktes Management – explizite Propagierung* verwenden. Es sind aber auch Mischformen möglich bzw. nötig, nämlich dann, wenn zwei Transactional Objects, die unterschiedliche Propagierungsarten fordern, an einer Transaktion teilnehmen müssen. Die Kombination *indirekt – explizit* ist dadurch möglich, daß die Current–Klasse die *get_control*–Methode anbietet, die das zur Transaktion zugehörige Control–Objekt liefert. Dieses kann dann vom Client als Parameter an eine Methode eines Transactional Objects übergeben werden. Die Kombination *direkt – implizit* ist dadurch zu erreichen, daß der Client ein Current–Objekt erzeugt und mittels der *resume*–Methode, mit dem Control–Objekt als Parameter, seinen Thread an die Transaktion bindet. Ab dann sind Aufrufe mit impliziter Propagierung möglich. Es kann jeweils zwischen den Kontextmanagementarten gewechselt werden.

Am häufigsten ist die Kombination indirekt – implizit anzutreffen, da zum einen die Verwendung eines Current–Objekts weniger Programmieraufwand bedeutet, zum anderen meist implizite Propagierung seitens der Transactional Objects gefordert wird.

Es bleibt die Frage, ob die Bindung des Transaktionskontextes an den Programm–Thread bei Verwendung des Current–Objekts zu starr ist. Beim direkten Kontextmanagement kann durch geeignete Verwaltung der Transaktionskontexte eine transaktionale Operation in der jeweils gewünschten Transaktion ausgeführt werden. Beim indirekten Management muß dazu erst die Transaktion verlassen werden. Das temporäre Aufheben der Bindung zwischen Transaktion und Programmthread ist durch

den Aufruf der Methode *suspend* des Current–Objekts möglich. Mit *resume* kann dieser Vorgang umgekehrt werden und der Thread wieder an einen Transaktionskontext gebunden werden.

3 Veränderungen an FlowMark

In diesem Kapitel werden die Veränderungen beschrieben, die einzelne FlowMark–Komponenten erfahren müssen, damit das Konzept der Workflow–Transaktionen in FlowMark eingesetzt werden kann. Die vorgestellten Veränderungen basieren zum einen auf den Vermutungen über die Funktionsweise der FlowMark Workflow–Engine, die wir in Kapitel 1.2 beschrieben haben. Dort wurde ein funktionales Modell von FlowMark entwickelt, das wir am Ende dieses Kapitels so erweitern, daß mit der Workflow–Engine Workflow–Transaktionen ausgeführt werden können.

Zum anderen basieren die Vorschläge auf den Erfahrungen, die aus der Implementierung des prototypischen Workflowsystems „Surro“ gewonnen worden sind. Eine Beschreibung des Workflowsystems Surro findet sich in [SB96a].

Falls einige der angestellten Vermutungen falsch sind, können die beschriebenen Veränderungen unnötig oder falsch sein.

3.1 Veränderungen an der Workflow–Engine

Eine Workflow–Engine, die Workflow–Transaktionen beherrscht, unterscheidet sich von der Workflow–Engine aus dem funktionalen Modell darin, daß sie eine „zweite Art“ von Transaktionen unterstützt. Es gibt weiterhin Transaktionen zum Ändern eines Zustandes, die durch das Eintreffen bestimmter Nachrichten gestartet werden. Die zweite Art der Transaktionen sind die Workflow–Transaktionen, die aus Sicht der Engine als Verschmelzung aller Zustandsänderungs–Transaktionen innerhalb einer Sphäre angesehen werden können.

Eine Sphäre besteht aus einer Menge von Aktivitäten. Diese Aktivitäten durchlaufen bei der Abarbeitung eine Menge von Zustandsübergängen, die jeweils in der ursprünglichen Workflow–Engine durch einzelne Transaktionen geschützt werden. Diese Menge von Transaktionen wird durch die neue Workflow–Engine zusammen mit den Operationen der Aktivitäten in einer einzigen Transaktion ausgeführt.

Eine Workflow–Transaktion muß spätestens nach der Zustandsänderung begonnen werden, mit der nach einem Rollback der Workflow–Transaktion die Sphäre noch korrekt wiederholt werden kann. Typischerweise ist dies die Zustandsänderung der ersten erreichten Aktivität in der Sphäre, die die Aktivität in einen startbaren Zustand überführt. Bei einem Zurücksetzen auf diesen Zustand kann die Sphäre wiederholt werden. Die Workflow–Transaktion endet, wenn die letzte Aktivität einer Sphäre, die noch nicht im erfolgreichen Zustand ist, in einen erfolgreichen Zustand überführt wird, oder wenn ein Rollback ausgelöst wird. Alle dazwischenliegenden Zustandsänderungen werden in einer einzigen Transaktion, der Workflow–Transaktion,

ausgeführt.

Das funktionale Modell muß also dahingehend erweitert werden, daß bei jeder Nachricht aus der Warteschlange geprüft wird, ob die Nachricht Aktivitäten innerhalb einer Sphäre betrifft. Falls dies der Fall ist, wird keine neue Transaktion begonnen, sondern die bereits mit der Sphäre verbundene Transaktion wird verwendet, um den Zustandsübergang auszuführen.

Die Workflow–Engine kann mehrere Sphären und damit mehrere Workflow–Transaktionen gleichzeitig in Bearbeitung haben. Sie muß daher über einen längeren Zeitraum an unterschiedlichen Transaktionen teilnehmen können. Dazu muß sie einen Transaktionskontext vor dem eigentlichen Ende der Transaktion verlassen und in einen anderen eintreten und ebenso wieder zurückwechseln können.

Durch die Anwendung von Workflow–Transaktionen kommt es zu der Situation, daß mehrere Transaktionen parallel, aber nicht unbedingt gleichzeitig, auf den Verwaltungsdaten des Workflowsystems arbeiten. Aufgrund der Isolationseigenschaft der Transaktionen kann es vorkommen, daß eine Transaktion auf Daten zugreift, die in einer parallelen, noch nicht beendeten Transaktion, verändert worden sind. Die Daten sind daher gesperrt. Eine später zugreifende Transaktion muß warten. Wenn nun die Transaktionen durch einen einzigen Thread durchgeführt werden, ist hier eine Verklemmung entstanden. Mit der zweiten Transaktion wartet auch der einzige Engine–Thread, und damit kann die erste Transaktion nie beendet werden. Die Engine wartet ewig.

Aus diesem Grund muß sichergestellt sein, daß die Transaktionen nie auf Sperren auflaufen. Dies kann durch drei unterschiedliche Maßnahmen erreicht werden:

- Die Workflow–Verwaltungsdaten der Sphären müssen so gespeichert werden, daß kein Zugriff auf gemeinsame Relationen erforderlich ist. Die Transaktionen dürfen also nie auf gemeinsam gespeicherte Daten zugreifen. Die Erfüllbarkeit dieser Methode hängt vom Datenmodell des Workflowsystems ab. Da uns das von FlowMark benutzte Datenmodell unbekannt ist, können wir keine weiteren Aussagen darüber machen. Weiterhin muß eventuell im Datenmodell berücksichtigt werden, in welcher Art und Weise die zugrundeliegende Datenbank ihre Sperrverwaltung organisiert.

- Jede Transaktion kann ihren eigenen Engine–Thread besitzen. Nur dieser Thread kann Aktionen im Kontext dieser Transaktion durchführen. Wenn ein Thread auf eine Sperre einer Sphären–Transaktion aufläuft, muß er solange warten, bis diese Sphäre beendet wird. Da Sphären nur kurz andauern sollen, ist die Wartezeit kurz und damit vertretbar. Mit dem Wechsel des Transaktionskontexts muß ein Threadwechsel erfolgen. Die Gefahr einer zyklischen Wartesituation (Deadlock) zwischen den parallelen Transaktionen ist damit aber nicht beseitigt. OTS bietet zu diesem Zweck einen Zeitüberwachungsmechanismus an, mit dem eine Verklemmung erkannt werden kann.

- Beim Zugriff auf die Workflow–Verwaltungsdaten wird in den Transaktionen

die Isolationseigenschaft aufgegeben. Daher können keine Transaktionen auf Sperren auflaufen. Es muß allerdings durch die Anwendung (hier die Engine) sichergestellt werden, daß keine inkonsistente Daten entstehen.

Bei der Implementierung des Workflowsystems Surro hat sich gezeigt, daß die verwendete Implementierung von OTS aus DSOM 3.0 β **nicht** in der Lage ist, multithreaded auf die DB2 V2.1.1 zuzugreifen. Da es wahrscheinlich ist, daß dies ein grundsätzliches und nicht einfach zu umgehendes Problem ist, werden im folgenden verschiedene Realisierungsansätze erläutert.

Multithreaded Engine

Eine multithreaded Engine ist der konzeptionell beste Ansatz. Die Ablaufstruktur ist folgendermaßen: Mit Eintritt in eine Sphäre muß eine neue Workflow–Transaktion und ein neuer Workflow–Thread erzeugt werden. Für jede Nachricht aus der zentralen Warteschlange muß überprüft werden, ob sie zu einer Sphäre gehört, und wenn ja, zu welcher. Wenn die Nachricht eine Aktivität betrifft, die zu keiner Sphäre gehört, dann führt der normale Thread der Workflow–Engine die nötigen Operationen aus. Andernfalls muß die Sphäre bestimmt werden zu der die Aktivität gehört. Der Thread dieser Sphäre muß dann angewiesen werden, die Nachricht aus der Warteschlange zu entnehmen und mit den notwendigen Operationen zu reagieren.

Der Umbau einer Workflow–Engine, die bisher mit nur einem Thread gearbeitet hat, in eine, die mehrere Threads benutzt, ist voraussichtlich mit erheblichem Aufwand verbunden.

Singlethreaded Engine

Hier besteht die Hauptaufgabe darin, das Datenmodell und die Zugriffe auf die Daten so zu realisieren, daß es zu den oben beschriebenen Sperrkonflikten nicht kommen kann. Das Workflowsystem Surro ist dazu in der Lage, was aber kein Nachweis dafür ist, daß dies auch in FlowMark mit vertretbarem Aufwand möglich ist.

Ein einzelner Thread hat die Aufgabe, alle Nachrichten aus der Warteschlange auszulesen und den jeweiligen Transaktionskontext zu aktivieren, falls sich die Nachricht auf eine Sphäre bezieht. Wenn nicht, wird eine neue Transaktion für die anstehenden Operationen erzeugt. In [SB96a] wird dieser Ansatz ausführlicher erläutert.

Singlethreaded Engine mit Zerlegen der Sphären–Transaktion

Wenn der vorangehend beschriebene Ansatz nicht machbar oder zu riskant sein sollte, besteht noch die Möglichkeit, das Konzept der Workflow–Transaktionen abzuschwächen, um die Sperrkonflikte beim gemeinsamen Zugriff auf die Workflowdatenbank zu vermeiden. Dies könnte so aussehen, daß die Zugriffe auf den Workflowzustand in der Datenbank nicht mehr in der Transaktion abläuft, in der auch die Operationen der ACID–Aktivitäten eingebunden sind. Der Zugriff erfolgt dann genauso wie bei der Bearbeitung von Nachrichten von Aktivitäten außerhalb einer Sphäre in eigenen kurzen Transaktionen. Die ACID–Aktivitäten einer Sphäre werden nach wie vor unter dem Schutz eines einheitlichen Transaktionskontextes durchgeführt. Wenn

man dafür sorgt, daß die Ausführung der ACID–Aktivitäten den Engine–Thread nicht blockieren kann, können die beschriebenen Verklemmungen nicht auftreten.

Jetzt muß aber dafür gesorgt werden, daß im Fehlerfall der Workflowzustand — hergestellt durch korrekt beendete, einzelne Transaktionen — wieder auf einen Stand gebracht wird, der konsistent mit dem Zustand der jetzt zurückgesetzten ACID–Aktivitäten ist. Diese Veränderung des Workflowzustandes muß jetzt durch eine kompensierende Transaktion auf der Datenbank durchgeführt werden. Diese Kompensationstransaktion darf nicht fehlschlagen, ansonsten wäre der Gesamtzustand inkonsistent. Man muß also die zwei Transaktionen, die Sphärentransaktion und die Kompensationstransaktion, so koppeln, daß bei einem Rollback der Sphärentransaktion die Kompensationstransaktion angestoßen wird und dafür gesorgt wird, daß diese Transaktion irgendwann erfolgreich beendet wird. Dies ist aber nicht mit letzter Sicherheit zu garantieren.

Falls die Workflow–Transaktion fehlschlägt, wird sie zurückgesetzt. Zusammen mit dem Rücksetzen der Auswirkungen der Aktivitäten wird der Zustand des Workflows automatisch so verändert, als wäre die Sphäre nie ausgeführt worden. Dies geschieht auch bei der Variante mit zerlegten Transaktionen, nur nicht in einer einzigen Transaktion. Die Bearbeitung des Sphäre kann erneut beginnen. Hierbei ist es notwendig, daß man die Anzahl der Wiederholungen einer Sphäre begrenzt und nach dem Erreichen der Maximalzahl den gesamten Workflow mit einer Fehlermeldung abbricht oder eine alternative Vorgehensweise wählt. Falls nämlich ein nicht–transienter Fehler den Abbruch bewirkt, wäre man in einer Endlosschleife gefangen.

Die Arbeitsweise der Workflow–Engine bei der Bearbeitung von Aktivitäten innerhalb von Sphären unterscheidet sich in einem wichtigen Punkt von den Bearbeitung der Aktivitäten außerhalb der Sphäre: Während der Ausführung von Operationen innerhalb einer Sphäre muß die Workflow–Engine darauf achten, daß sie die Isolationseigenschaft der Sphäre nicht verletzt. Alle Daten– und Kontrollflüsse, die die Sphäre verlassen, dürfen nicht vor erfolgreichem Ende der Sphäre aktiv werden. Sie müssen bis zum Ende in ihrem ursprünglichen Zustand eingefroren werden. Erst nach dem erfolgreichen Ende der Workflow–Transaktion werden die Flüsse wieder aktiv.

Die Isolation der Anwendungsdaten muß durch die Aktivitäten, genauer durch ihre Eigenschaft als Resource Manager, gesichert werden. Das Workflowsystem hat keine direkten Eingriffs– oder Überwachungsmöglichkeiten. Im dritten Realisierungsansatz werden die Änderungen des Workflowzustandes in eigenen Transaktionen ausgeführt und sind somit auch schon vor dem Ende der Sphäre nach außen hin sichtbar. Andere Systemkomponenten könnten daher inkonsistente Daten lesen. Momentan kommt als einzige betroffene Komponente der Workflow–Monitor in Frage, der den Workflow–Zustand visualisiert und dem Benutzer präsentiert. Er führt aber keine weiteren Bearbeitungsschritte mit diesen Daten durch und ist somit unkritisch. Die Datenbank muß dem Monitor aber unbeschränkten Zugriff auf die Daten gewähren, sonst könnte der Fortschritt innerhalb einer Sphäre nicht dargestellt werden.

Dead-Path–Elimination

Die Dead–Path–Elimination hat die Aufgabe, Aktivitäten im Workflow, die aufgrund des momentanen Workflowzustandes nicht mehr erreicht werden können, als „terminiert“ zu markieren und deren ausgehende Kontrollkonnektoren zu „false“ zu evaluieren. Bei der Verwendung von Sphären muß die Dead–Path–Elimination folgende Regeln beachten:

- Eine Dead–Path–Elimination, die innerhalb einer Sphäre initiiert wird, darf die Sphäre nicht verlassen, bevor nicht die zugehörige Transaktion erfolgreich beendet ist (Isolation). Dies ist notwendig, da nach einem Rollback und Wiederanlauf sich der Ablauf des Workflows innerhalb der Sphäre ändern könnte, und damit die Dead–Path–Elimination eventuell nicht mehr gestartet werden muß.

- Eine Dead–Path–Elimination, die eine Sphäre von außen betritt, muß die Sphäre ungehindert betreten und durchlaufen können. Die Sphäre kann dabei auch vollständig der Elimination zum Opfer fallen. Wird eine Sphäre von der Dead–Path–Elimination betreten, aber nicht die gesamte Sphäre invalidiert, kann die Sphäre ohne Berücksichtigung der Isolation durchlaufen werden. Dies ist möglich, da auch durch ein Rollback und Neustart einer Sphäre die von der partiellen Elimination betroffenen Objekte keinen anderen Zustand erreichen können. Auch muß durch das Betreten einer Sphäre die Sphärentransaktion durch die Dead–Path–Elimination nicht aktiviert werden.

3.2 Veränderungen am Runtime–Client

Am Runtime–Client ist nur wenig zum Einbau von Workflow–Transaktionen vorzunehmen. Das Starten von ACID–Aktivitäten übernimmt der Program–Execution–Client. Es ist sinnvoll, daß der Benutzer auf seiner Arbeitsliste die Aktivitäten, die in Workflow–Transaktionen eingebettet sind, von normalen Aktivitäten unterscheiden kann. ACID–Aktivitäten verhalten sich nämlich anders: Auch nach einer erfolgreichen Bearbeitung einer solchen Aktivität kommt es bei einem Rollback der Workflow–Transaktion vor, daß die Aktivität nochmals auf der Arbeitsliste erscheint und erneut bearbeitet werden muß. Häufig dürften Sphären–Aktivitäten automatisch, also ohne auf der Arbeitsliste eines Benutzers zu erscheinen, ausgeführt werden.

Im Runtime–Client muß eine neue Funktion im Menü angeboten werden, mit der man Workflow–Transaktionen manuell abbrechen kann. Diese Funktion kann z. B. im Monitorfenster realisiert werden. Dort muß ebenso wie in der Buildtime–Komponente eine Sphäre mitsamt ihrem Zustand angezeigt werden können. In diesem Fenster muß dann mit der Maus eine aktive Sphäre ausgewählt und abgebrochen werden können. Der Abbruch von Sphären sollte allerdings über die Vergabe von Rechten geschützt werden können. Beim Abbruch einer Sphäre müssen die betroffenen Aktivitäten automatisch von den Arbeitslisten entfernt werden können.

3.3 Veränderungen an der Buildtime–Komponente

Die Buildtime–Komponente muß so verändert werden, daß mit ihr die oben beschriebene Art von Sphären modelliert werden kann. Mit der bereits bestehenden Modellierungs–Komponente der FlowMark Version 2.1 kann ein Arbeitsorganisator Geschäftsprozesse in der Form eines Aktivitätennetzes modellieren. Er erzeugt dabei Aktivitäten, füllt die Attribute der Aktivitäten mit Werten und legt die Position des Abbilds einer Aktivität auf einer Zeichenfläche fest. Die Aktivitäten werden dann manuell durch Daten– und Kontrollflüsse verbunden. Beim Verbinden wird überprüft, ob die neue Verbindung einen Zyklus bildet und somit nicht erlaubt wäre. Die Position einer Aktivität kann durch Drag–and–Drop–Technik nachträglich verändert werden. Die Pfeile der Daten– und Kontrollflüsse werden dabei automatisch angepaßt. In Abbildung 8 ist ein fertig modellierter Workflow zu sehen.

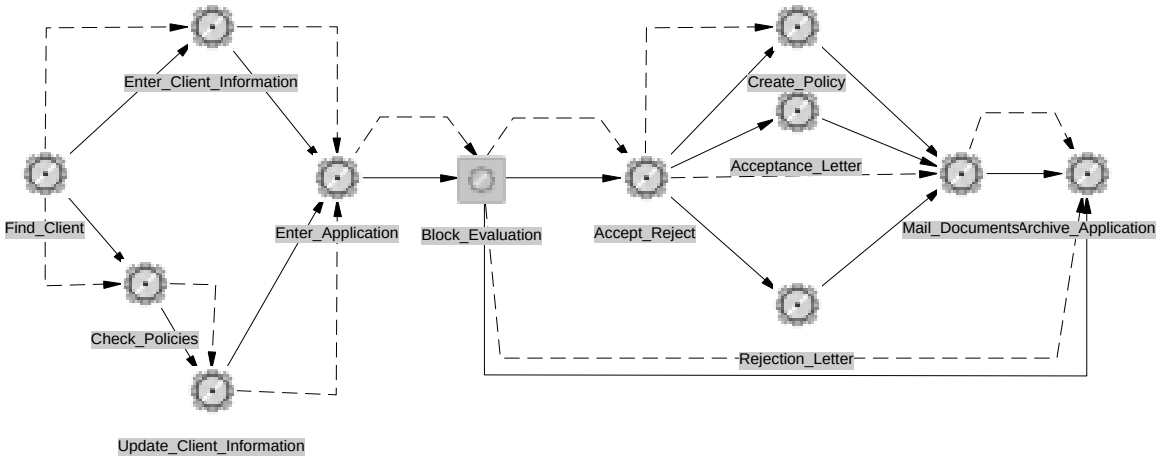

Abbildung 8: Ein in FlowMark modellierter Geschäftsprozeß

Folgende Funktionen müssen in der Modellierungskomponente zusätzlich implementiert werden:

- Nimm Aktivität in Sphäre auf
  Eine bestehende Aktivität muß in eine Sphäre aufgenommen werden können. Das kann mittels Drag–and–Drop geschehen.

- Entferne Aktivität aus der Sphäre
  Eine Aktivität in der Sphäre muß aus der Sphäre genommen werden können.

- Überprüfe strukturelle Bedingungen
  Eine Funktion zur Überprüfung, ob die geforderten strukturellen Bedingungen der Sphäre eingehalten werden, muß beim Einfügen und Entfernen von Aktivitäten aus der Sphäre ausgeführt werden. Diese Funktion sollte auch über ein Menüpunkt durch den Arbeitsorganisator abrufbar sein.

Das Erzeugen einer neuen Aktivität in einer Sphäre kann durch eine Kombination einer bestehenden Funktion ERZEUGE AKTIVITÄT und der neuen Funktion NIMM AKTIVITÄT IN SPHÄRE AUF realisiert werden. Das Löschen einer Aktivität in einer Sphäre ist mit dem normalen Löschen einer Aktivität identisch.

Eine Sphäre gruppiert eine Menge von Aktivitäten. Auf der Zeichenfläche muß ersichtlich sein, welche Aktivität zu welcher Sphäre gehören. Dieses Problem wird dadurch erschwert, daß die Menge von Aktivitäten keine Zusammenhangskomponente in dem Aktivitätennetz bilden muß.

Eine Methode, die Zugehörigkeit anzeigen, ist das Einfärben aller Aktivitäten einer Sphäre in einer einheitlichen Farbe. Der Vorteil dieser Methode liegt in der einfachen Realisierung. Der Nachteil in der relativ beschränkten Anzahl der zur Verfügung stehenden Farben. Auch kann eine einzelne, abseits liegende Aktivität leicht übersehen werden. Ein– und Ausgänge von Sphären sind so nur schwer zu erkennen.

Eine zweite Methode ist das Umhüllen aller Aktivitäten einer Sphäre. Aufgrund der Bedingung, daß auch zusammenhanglose Aktivitäten Teil einer Sphäre sein können, kann die Hülle der Sphäre als grafisches Objekt nicht immer einfach aufgebaut sein. Eine Ellipse oder ein Rechteck ist z. B. für diesen Zweck nicht geeignet. Es muß daraus gefolgert werden, daß die Hülle frei verformbar sein muß. Wenn die Hülle per Freihandzeichnung in das Aktivitätennetz eingezeichnet wird, ergibt sich das Problem, daß beim Verschieben einer Aktivität die Aktivität aus der Sphäre entfernt werden kann. Es muß daher ein Mechanismus gefunden werden, der, analog zum Mitziehen der Daten– und Kontrollflüsse, auch ein Mitziehen der Sphäre mit der Aktivität erlaubt. Hieraus ergibt sich aber wieder ein neues Problem. Eine vorher nicht in der Aktivität liegende Aktivität darf durch das Mitziehen der Sphäre nicht in der neu erstellten Hülle zu liegen kommen. Die grafische Darstellung der Sphäre erfordert daher Heuristiken, wie ein Aktivitätennetz mit Sphäre übersichtlich dargestellt werden kann. Im Monitor des Workflowsystems Surro ist eine solche einfache Heuristik implementiert.

3.4 Veränderungen an der Kommunikationsstruktur

Als Kommunikationsprotokolle werden in der jetzigen FlowMark Version TCP/IP, NetBIOS oder APPC eingesetzt. In Zukunft soll das IBM–Produkt MQSeries zur Kommunikation zwischen FlowMark–Komponenten eingesetzt werden. MQSeries garantiert die sichere Auslieferung von Nachrichten über Rechnergrenzen hinweg in heterogenen Systemumgebungen. Die Kommunikationswege werden dazu als persistente Warteschlangen modelliert. Nachrichten können damit asynchron an Prozesse verschickt werden, d. h. die Empfängerprozesse müssen beim Absenden nicht unbedingt existieren. Die Nachrichten werden solange persistent in der Kommunikationswarteschlange gespeichert, bis der Empfängerprozeß aktiv wird und die Nachricht abholt.

Durch den Einsatz von OTS muß eine neue Kommunikationsart in das FlowMark–System eingeführt werden. Mit Hilfe eines Object Request Brokers müssen Ob-

jektaufrufe systemweit durchgeführt werden können. Schon der Beginn einer Workflow–Transaktion erfordert mehrere Methodenaufrufe in OTS–Objekten durch die Workflow–Engine.

Während der Bearbeitung einer Workflow–Transaktion müssen beim Start einer Aktivität Methodenaufrufe in den Anwendungsobjekten, den ACID–Aktivitäten, durchgeführt werden können. Es gibt drei Möglichkeiten, wer diese Aufrufe absetzen kann:

1. Die Engine kann den Methodenaufruf selbst absetzen. Da die Engine meistens auf einem anderen Rechner als das Anwendungsprogramm läuft, muß hier DSOM eingesetzt werden. Der Program–Execution–Client wird bei dieser Kommunikationsart nicht eingesetzt. Der Methodenaufruf kann entweder die Arbeit im Anwendungsobjekt anstoßen, oder in der Methode wird die ganze Arbeit geleistet. Im Fall des Anstoßens muß das Objekt einen weiteren Aufruf an die Engine absetzen, wenn es seine Arbeit beendet hat. Wenn in der Methode die ganze Arbeit erledigt wird (synchroner Aufruf), bleibt die Engine während der Bearbeitungszeit blockiert. Sie muß daher einen Thread erzeugen, in dem der Methodenaufruf abgesetzt wird. Diese Aufgabe kann auch ein Program–Execution–Client übernehmen, der einen Methodenaufruf im Auftrag der Engine übernimmt.

2. Auch der bisher eingesetzte Program–Execution–Client kann den Methodenaufruf im Anwendungsobjekt vornehmen. Die bisherige Kommunikation zwischen Engine und Execution–Client kann damit wie bisher genutzt werden. Als einzige Ergänzung muß dem Auftrag an den Execution–Client, einen Methodenaufruf zu tätigen, der Transaktionskontext mitgegeben werden. Der Thread des Program–Execution–Client assoziiert sich bei impliziter Propagierung mit der Transaktion und führt den Aufruf durch. Bei expliziter Propagierung wird der Kontext als Parameter durchgereicht. Auch hier muß berücksichtigt werden, daß der Execution–Client mehrere Threads verwendet, damit mehrere Anwendungsobjekte auf einem Rechner parallel angesprochen werden können.

3. Die dritte Möglichkeit besteht darin, ein Hüllenprogramm (Wrapper–Programm) zu nutzen, das den Methodenaufruf durchführt. Die Kommunikation zwischen Workflow–Engine und Program–Execution–Client bleibt bis auf den zusätzlichen Parameter, den Transaktionskontext, unverändert. Der Program–Execution–Client bleibt auch weitgehend unverändert. Er muß nur den mitgelieferten Transaktionskontext in einen Stringparameter konvertieren und in der Kommandozeile dem Wrapperprogramm übergeben. Der Thread des Wrapper–Programms bindet sich an die Workflow–Transaktionen und macht dann den Methodenaufruf. Der Wrapper kann während des Methodenaufrufs blockiert bleiben.

4 Beispielhafte Anwendung von OTS: eine transaktionale Konto–Klasse

Um die Funktionsweise von OTS besser zu verstehen und um mit den OTS– und SOM–Aufrufen vertraut zu werden, wurde von uns ein Beispielprogramm erstellt, das zur Transaktionverwaltung OTS benutzt. Als Anwendung wurde das klassische Beispiel einer Überweisung eines Geldbetrags von einem Konto auf ein anderes gewählt. Die Implementierung benutzt SOM 3.0 von IBM, in der bereits mehrere Object Services (insbesondere OTS) implementiert sind. SOM 3.0 ist bisher nur auf OS/2 verfügbar und noch im Beta–Zustand.

In den folgenden Abschnitten wird zuerst das IDL–Interface der Kontoklasse erläutert. Dann folgt eine Beschreibung der Implementierung des Clients und des Servers mit eingestreuten Codefragmenten.

4.1 Interface-Definition

Die implementierte Klasse „account“ für Konto-Objekt bietet folgende Schnittstellen an (in IDL-Notation, gekürzt):

```
// IDL interface definition for an account object

#include <somtran.idl>

interface accountResource;

interface account: CosTransactions::TransactionalObject
{
   void accountInit(in long accNumber);
   // init account object with its number

   accountResource register_in_TA();
   // register myself with a coordinator

   float read();
   // get current value of account

   void  write(in float amount);
   // set new value of account

   void deposit(in float amount);
   // add amount to account

   void withdraw(in float amount);
   // subtract amount from account
```

```
    CosTransactions::Vote prepare();
    // 2PC-methods: prepare for commit

    void rollback();
    // 2PC-methods: rollback Transaction

    void commit();
    // 2PC-methods: commit Transaction

    attribute float value;       // amount of money of this account
    attribute long  accNumber;   // account number

    attribute string accfile;    // name of file which contains persistent value
    attribute string acctemp;    // name of file which contains prepare value

    attribute accountResource accRes; // stores accountResource  object
    attribute CosTransactions::Current current; // stores Current object
    ...
};
```

In den Attributen Kontostand (*value*) und Kontonummer (*accNumber*) werden die Daten eines Konto–Objekts gespeichert. Die anderen Attribute dienen Verwaltungszwecken. An Grundfunktionalität bietet ein Konto–Objekt die Methoden *read* und *write* zum Auslesen bzw. Setzen des Kontostandes, und *deposit* und *withdraw* zum Erhöhen bzw. Vermindern des Kontostandes an.

Zusätzlich müssen Methoden angeboten werden, die zur korrekten Teilnahme an einer Transaktion notwendig sind. Dies sind *prepare, commit* und *rollback*. Durch die Ableitung der Konto–Klasse von „CosTransactions::TransactionalObject“ benutzt die Klasse das Prinzip der impliziten Propagierung des Transaktionskontexts, d. h. daß bei jedem Methodenaufruf automatisch der Transaktionskontext mitgeliefert wird. Dies ist eine in SOM vorkommende Spezialisierung der OTS–Definition. Dort ist ein Transactional Object allgemein ein Objekt, dessen Verhalten vom Vorhandensein eines Transaktionskontextes beeinflußt wird. In SOM bedeutet die Ableitung von der Klasse „TransactionalObject“ zusätzlich die Verwendung der impliziten Propagierung.

Wenn der Kontext explizit propagiert werden soll, muß in jeder transaktionalen Methode ein zusätzlicher Parameter für den Kontext aufgenommen werden. Der Kontext wird dabei durch das *Control*–Objekt repräsentiert.

4.2 Die Client–Implementierung

Das Beispiel basiert auf einer Client–Server–Struktur. Der Client ruft in einer Transaktion Methoden der Konto–Objekte auf und nutzt so die Funktionalität, die von

den Konto–Objekten, den Servern, angeboten wird. Der Client ist ein *Transactional Client* und die Konto–Objekte sind gleichzeitig „Transactional Objects“ und „Recoverable Objects“[1]. Der Client führt folgende Operationen durch:

1. Start einer Transaktion
2. Aufrufen der transaktionalen Methoden in den Transactional Objects. Also z. B. das Umbuchen von einem Konto auf ein anderes.
3. Terminieren der Transaktion über Commit oder Rollback.

Der Ablauf im Client wird in den folgenden Punkten detailliert erläutert (Vergleiche dazu auch Abb. 9):

- Die Konto–Objekte und die Client–Prozesse müssen als DSOM–Server implementiert sein. Ein DSOM–Serverprozeß hat die Fähigkeit, Methoden lokaler Objekte nach außen anzubieten, die dann durch die Vermittlung eines ORB von anderen, entfernten Objekten aufgerufen werden können. Da das Coordinator–Objekt im Client erzeugt wird und dieses von den entfernten Konto–Objekten angesprochen werden muß, benötigt auch der Client einen eigenen DSOM–Server–Thread, der Aufrufe von außen bearbeitet. Zusätzlich müssen der Client und die Konto–Klasse im DSOM–Implementation–Repository als DSOM–Server registriert werden.

  Die Konto–Objekte sind nicht als eigenständigen Prozesse, sondern als DLL (dynamic link libraries) realisiert. Ein Standard–Serverprozeß (*somossvr.exe*) lädt diese Libraries und stellt Threads für die Ausführung von Methoden zur Verfügung.

  Bei der Übersetzung des Quellcodes der Konto–Klasse wird für den Client eine Stub–Library erzeugt, die dieser einbindet, um auf die Methoden im Konto–Objekt zugreifen zu können. Ein DSOM–Daemon (*somdd*) sorgt für die Kommunikationsverbindung zwischen den entfernten DSOM–Objekten.

- Nach der DSOM–Initialisierungsphase erzeugt der Client ein oder mehrere Konto–Objekte. Falls die Objekte schon existieren, muß dieser Schritt durch eine Suche über den Naming Service von DSOM ersetzt werden. Der Aufruf der *accountInit*-Methoden legt u. a. die Kontonummer fest, die als Parameter übergeben wird (s.u.). Zur Vereinfachung ist in den folgenden Codefragementen die Fehlerbehandlung weggelassen worden.

```
account *const acc1 = (account *)somdCreate(ev,"account",TRUE);
account *const acc2 = (account *)somdCreate(ev,"account",TRUE);

acc1->accountInit(ev,371);
acc2->accountInit(ev,535);
```

[1] siehe Seite 10

- Der Client startet eine Transaktion, indem er ein *Current*–Objekt erzeugt und die *begin*-Methode aufruft (indirektes Kontextmanagement). Der Client–Thread wird durch den begin–Aufruf mit der Transaktion assoziiert:

```
// create CURRENT-object
CosTransactions_Current *const current = new CosTransactions_Current;

// Begin a top-level transaction
current->begin(ev);
```

- Jetzt kann der Client Methoden der Konto–Objekte im Kontext dieser Transaktion aufrufen:

```
acc1->withdraw(ev,350);
acc2->deposit(ev,350);
```

  Die Methodenaufrufe enthalten den Transaktionskontext nicht als Parameter. Der Kontext wird implizit an das Konto–Objekt übergeben, da die Konto–Klasse von *TransactionalObject* abgeleitet ist und der Client–Thread über das Current–Objekt mit einer Transaktion assoziiert ist.

- Zur Beendigung einer Transaktion ruft der Client eine der beiden folgenden Methoden auf:

```
current->commit(ev,FALSE);
```

  oder

```
current->rollback(ev);
```

  Durch den Commit–Aufruf wird das 2–Phasen–Commit–Protokoll (2PC) angestoßen, in dem OTS als Protokollführer auftritt.

4.3 Die Implementierung der Konto–Klasse

In der Konto–Objekt Implementierung wird der Kontostand persistent in einer Datei gesichert. In einer weiteren Ausbaustufe könnte man die Objekte mittels des *Persistent Object Service* (POS) von SOM persistent halten.
Der Ablauf in der Konto–Klasse sieht folgendermaßen aus (Vergleiche dazu auch Abb. 9):

- Definition einer Resource–Klasse
  Damit ein Konto–Objekt in das 2PC–Protokoll eingebunden wird, muß es beim Koordinator einer Transaktion ein *Resource*–Objekt registrieren. Ein Resource–Objekt bietet dem Transaktionsmanager die zur Durchführung des 2PC notwendigen Schnittstellen an. Dies sind *prepare(), commit()* und *rollback()*. Für die Klasse „account“, die die Konto-Objekte realisiert, wird eine Klasse „accountResource“ definiert. Diese ist von der Klasse „Resource“ abgeleitet und erbt die erwähnten drei Methoden. Die IDL-Definition sieht folgendermaßen aus (gekürzt):

```
// define the resource class for accounts

#include <somtran.idl>      // Transaction Service

interface account;          // prototype

interface accountResource: CosTransactions::Resource
{
   attribute account myAccount; // stores relationship to account object
   attribute CosTransactions::Coordinator coord;

   void register_resource(in CosTransactions::Coordinator coord,
                          in account myAccount);
   ...
};
```

  Als einzige Methode wird *register_resource* deklariert. Mittels dieser Methode registriert sich ein Konto–Objekt beim eigenen Resource–Objekt. Die Registrierung wird im Attribut „myAccount“ gespeichert. Dies ist notwendig, damit das Resource–Objekt einen Bezug auf sein zugehöriges Konto–Objekt hat, um bei der Durchführung der Terminierung darauf zugreifen zu können.

- Initialisierung
  Nach dem Erzeugen eines Konto–Objektes wird dieses durch die Methode *accountInit* initialisiert. Dabei wird die Kontonummer gesetzt und der Kontostand ermittelt. Der Kontostand eines Konto–Objekts wird persistent in einer Datei gehalten. Der Name der Kontostandsdatei steht im Attribut „accfile“. Falls diese Datei schon existiert, wird der momentane Kontostand eingelesen.

- Ablauf eines Methodenaufrufs
  Beim ersten Aufruf einer Methode eines Konto–Objektes, die einen Transaktionskontext benötigt, wird ein „accountResource“–Objekt erzeugt. Diesem wird über die „register_resource“–Methode eine Referenz auf das Konto–Objekt übergeben. Das Konto–Objekt registriert dann das accountResource–Objekt beim Koordinator der Transaktion.

Als Beispiel ist die *write*–Methode dargestellt. Der Aufruf *register_in_TA* überprüft, ob der Transaktionskontext bereits bekannt ist. Falls nicht, und falls der Aufruf aus einer Transaktion heraus erfolgt, wird ein neues Objekt der Klasse „accountResource“ erstellt. Dieses Objekt wird beim Koordinator registriert. Anschließend wird das Konto–Objekt bei seinem Resource–Objekt registriert. Nach der Registrierungsroutine wird der neue Kontostand im *value*–Attribute gespeichert. Falls die Transaktion bestätigt wird, wird in der commit–Methode dieser Wert in die Kontostandsdatei geschrieben.

```
/*
 * set new value of account
 */

SOM_Scope void  SOMLINK write(account *somSelf,  Environment *ev,
                              float amount)
{
    accountData *somThis = accountGetData(somSelf);
    accountMethodDebug("account","write");

    accountResource *resource = somSelf->register_in_TA(ev);
    somThis->value = amount; // update account in memory
}

/*
 * register myself with a coordinator
 */

SOM_Scope accountResource*  SOMLINK register_in_TA(account *somSelf,
                                                   Environment *ev)
{
    accountData *somThis = accountGetData(somSelf);
    accountMethodDebug("account","register_in_TA");

    CosTransactions_Control             *ctrl = NULL;
    CosTransactions_Coordinator         *coord = NULL;
    CosTransactions_RecoveryCoordinator *rcoord = NULL;

    // only one resource object -> only one TA
    // do I already have a resource object?
    if (somThis->accRes != NULL) {
       return (somThis->accRes);
    }

    // get control object from current object
    ctrl = somThis->current->get_control(ev);
```

```
    error(ev,"get_control failed");

    // get_control returns a NULL-pointer if there is no transaction
    // context associated with the call.
    if (ctrl==NULL) {
       somPrintf("register_in_TA: Call to account object without TA\n");
       return NULL; // no transaction
    } /* endif */

    coord = ctrl->get_coordinator(ev);
    error(ev,"get_coordinator failed");

    // create new resource object
    somThis->accRes = (accountResource *)
                      somdCreate(ev,"accountResource",TRUE);
    error(ev,"creation of accountResource object failed");
    if (!somIsObj(somThis->accRes)) {
       somPrintf("creation of accountResource object failed!\n");
       return NULL;
    } /* endif */

    // register your resource object at the coordinator
    // returns recovery coordinator
    rcoord = coord->register_resource(ev,somThis->accRes);
    error(ev,"register_resource of coordinator failed");

    // register coordinator and account object (self) at the resource
    // (user defined method of accountResource)
    somThis->accRes->register_resource(ev,coord,somSelf);
    error(ev,"register_resource of accountResource failed");

    return somThis->accRes;
}
```

Falls es möglich ist, daß aus mehreren unterschiedlichen Transaktionskontexten heraus auf dasselbe Konto–Objekt zugegriffen werden kann, muß zusätzlich eine Verwaltung von Transaktionskontexten eingerichtet werden. Diese registriert, an welchen Transaktionen das Objekt teilnimmt, und kann so feststellen, ob ein Methodenaufruf aus einer neuen oder einer bereits bekannten Transaktion heraus erfolgt und dann entsprechend reagieren. In der hier beschriebenen Testimplementierung ist dies vorerst nicht realisiert, da immer nur ein Client zu einem Zeitpunkt existiert.

4.4 Ablauf des 2PC

Im folgenden wird der Ablauf der Terminierung einer Transaktion näher beschrieben. Leitet der Client die Terminierung über einen Commit–Aufruf ein, so wird das 2PC angestoßen. Bei einem Rollback ist dies nicht notwendig. Der Koordinator ruft in allen bei ihm registrierten Resource–Objekt die *prepare()*–Methode auf. Im accountResource–Objekt wird der prepare–Aufruf an das Konto–Objekt durchgereicht.

```
// accountResource-Klasse

SOM_Scope CosTransactions_Vote  SOMLINK prepare(accountResource *somSelf,
                                                Environment *ev)
{
    accountResourceData *somThis = accountResourceGetData(somSelf);
    accountResourceMethodDebug("accountResource","prepare");

    account *acc = somThis->myAccount;

    vote = acc->prepare(ev);
    return vote;
}
```

Aufgabe der prepare–Methode im Konto–Objekt ist es, entweder den Abbruch der Transaktion zu initiieren, falls ein entsprechender Fehlerzustand festgestellt wird (d. h. das Objekt votiert mit „VoteRollback“), oder den momentanen Zustand so zu sichern, daß auch nach einem Fehler beide Entscheidungen – Commit oder Rollback – möglich sind (Objekt votiert mit „VoteCommit“).

```
//account-Klasse

SOM_Scope CosTransactions_Vote  SOMLINK prepare(account *somSelf,
                                                Environment *ev)
{
    accountData *somThis = accountGetData(somSelf);
    accountMethodDebug("account","prepare");

    // save the current value (afterimage) in a temp file;
    // beforeimage is in original file
    ofstream tempfile(somThis->acctemp);
    if (!tempfile) {
       somPrintf("Prepare: output file %s open error\n",somThis->acctemp);
       return CosTransactions_VoteRollback;
    } /* endif */
    tempfile << somThis->value;
    tempfile.close();
```

```
    return CosTransactions_VoteCommit;
}
```

Hier wird der momentane Kontostand („afterimage“) in einer temporären Datei gesichert. Der Wert des Kontos bei Beginn der Transaktion („beforeimage“) ist der Inhalt der Kontostandsdatei, die während einer Transaktion nicht geändert wird. Falls die Sicherung des afterimage nicht möglich ist, votiert das Objekt mit „Rollback“.

Wenn alle beteiligten Objekte ihr Votum abgegeben haben, ruft der Koordinator — je nach Ausgang der Votierphase — die *commit()*– oder *rollback()*–Methoden der Resource–Objekte auf. Diese werden, wie beim prepare(), direkt an das Konto–Objekt durchgereicht. Bei einer Commit–Entscheidung ersetzt das Konto–Objekt seinen alten Kontostand mit dem neuen. Beim Rollback wird der neue Stand verworfen (die temporäre Datei wird gelöscht) und der alte Kontostand bleibt unverändert bestehen.

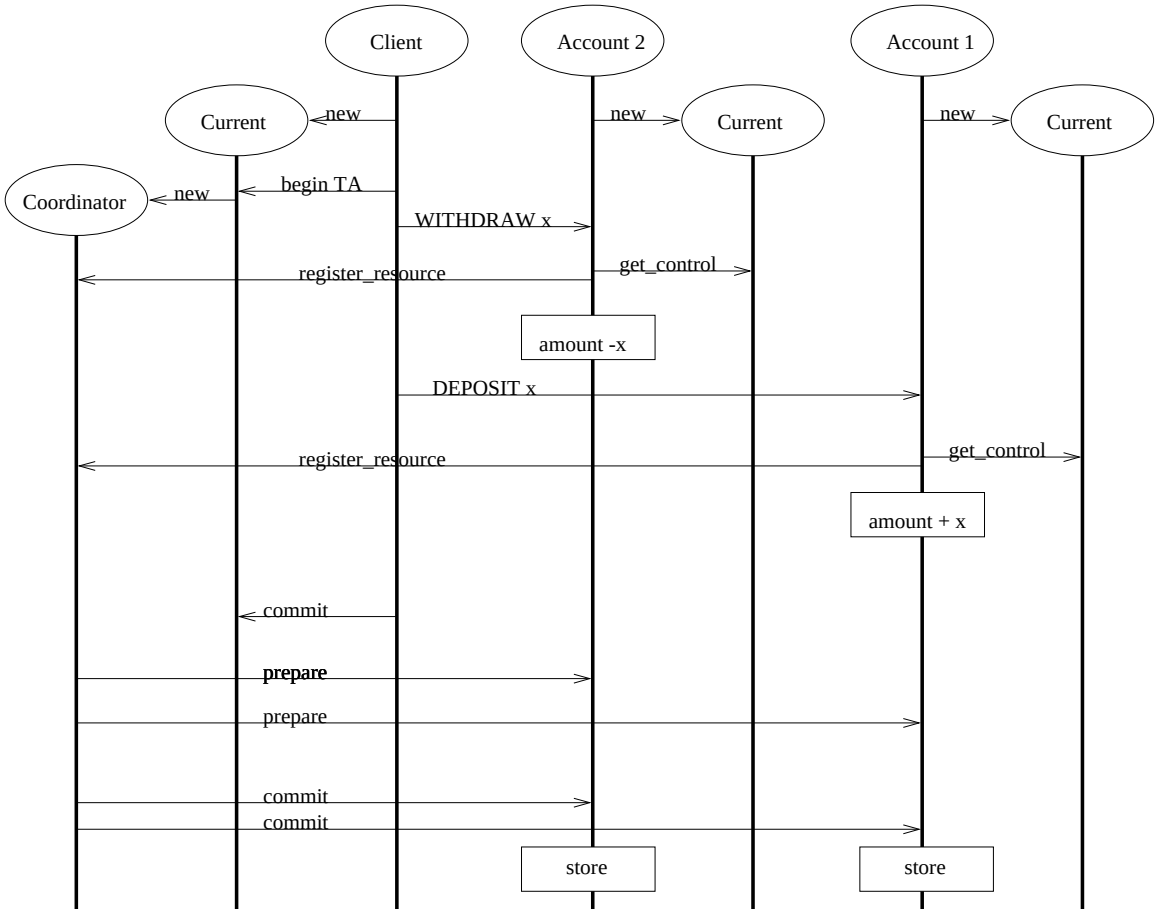

Abbildung 9: Gesamtablauf einer Transaktion

In Abb. 9 ist noch einmal der Ablauf einer Transaktion zusammengefaßt. Aus Übersichtlichkeitsgründen fehlen in der Darstellung die accountResource–Objekte und einige Methodenaufrufe. Die account–Objekte werden als schon vorhanden vorausgesetzt.

Literatur

[AGK95] Alonso, G. ; Günthör, R. ; Kamath, M. ; Agrawal, D. ; El Abbadi, A. ; Mohan, C.: Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. **In:** *Proc. of 3rd Int. Conference on Cooperative Information Systems*. Wien, 1995

[Obj90] Object Management Group (OMG): *Object Management Architecture Guide*. November 1990

[Obj93] Object Management Group (OMG): *The Common Object Request Broker Architecture and Specification. Revision: 1.2*. 1993

[SB96a] Schreyjak, Stefan ; Bildstein, Hubert: *Beschreibung des prototypisch implementieren Workflowsystems Surro* Universität Stuttgart, Software–Labor. 1996. – Fakultätsbericht Nr. 1996/19, Software–Labor Bericht SL–5/96

[SB96b] Schreyjak, Stefan ; Bildstein, Hubert: *Fehlerbehandlung in Workflow–Management–Systemen* Universität Stuttgart, Software–Labor. 1996. – Fakultätsbericht Nr. 1996/17, Software–Labor Bericht SL–3/96