Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen

Diplomarbeit

Marcel Kornacker

22. August 1995

betreut von

Prof. Dr. Joachim W. Schmidt
Universität Hamburg

Prof. Dr. Friedrich Vogt
Technische Universität Hamburg-Harburg

Inhaltsverzeichnis

1. Einleitung und Motivation **1**
- 1.1 Langlebige Aktivitäten in offenen Umgebungen 2
- 1.2 Zielsetzung und Überblick 6

2. Tycoon als transaktionale Umgebung **8**
- 2.1 Bedeutung des Transaktionskonzeptes für persistente Programmiersprachen . 8
- 2.2 Möglichkeiten der transaktionale Strukturierung von Programmen 12
- 2.3 Ein Transaktionsmodell für Tycoon basierend auf persistenten Sicherungspunkten 16

3. Sicherungspunkte für Tycoon-interne Zustände **21**
- 3.1 Anforderungen durch das abstrakte Speicherprotokoll 21
- 3.2 Implementationstechniken für Vorwärtsverarbeitung, Zurücksetzen und Wiederanlauf 24
  - 3.2.1 Logstrukturen für persistente Programmiersprachen mit abstraktem Speicherprotokoll 24
  - 3.2.2 Loggingaktivität während der Vorwärtsverarbeitung 26
  - 3.2.3 Beschreibung der Transaktionsalgorithmen 28
  - 3.2.4 Interaktion von Logging und Garbage Collection 30
- 3.3 Bewertung des Logging-Verfahrens 31

4. Sicherungspunkte für Operationen auf flüchtigen externen Daten **33**
- 4.1 Übersicht und Ziele des Loggingansatzes für flüchtige externe Daten 34
- 4.2 Charakterisierung externer Operationen auf flüchtigen Daten 35
- 4.3 Eine Programmierschnittstelle für das Logging von flüchtigen externen Daten 39
- 4.4 Anwendungsbeispiel der Programmierschnittstelle 42
- 4.5 Implementationstechniken für transaktionale Verarbeitung 45
  - 4.5.1 Datenstrukturen für das Operationslogging 45
  - 4.5.2 Loggingaktivität während der Vorwärtsverarbeitung 48
  - 4.5.3 Algorithmen für das Zurücksetzen 50
  - 4.5.4 Bereinigungsverfahren für die Log-Struktur 53
  - 4.5.5 Algorithmen für den Wiederanlauf 58
- 4.6 Anwendungsgrenzen des Operations-Logging für flüchtige externe Daten . . . 59

5. Sicherungspunkte für transaktionale externe Dienste **62**
- 5.1 Integration externer transaktionaler Dienste 62
- 5.2 Übersicht des X/Open-Standards für verteilte Transaktionsverarbeitung . . . 67
- 5.3 Integration von Tycoon in das X/Open-Modell 71

5.3.1 Setzen eines Synchronisationspunktes 73
5.3.2 Zurücksetzen 75
5.3.3 Wiederanlauf 76

6. Zusammenfassung **78**
6.1 Stand der Implementation 79
6.2 Ausblick 80

A. Schnittstelle Volatile **81**

B. Schnittstelle Transaction **84**

Abbildungsverzeichnis

2.1 Beispiel für die externe Umsetzung von Sicherungspunkten 16
2.2 Die Schnittstelle des Moduls *transaction*. 17
2.3 Die Ressourcenklassifizierung 19

3.1 Die Tycoon-Architektur 22
3.2 Die erweiterte Tycoon-Architektur 23
3.3 Ausschnitt aus der Log-Struktur mit Neuliste 27
3.4 Logstruktur mit Neuliste nach Sicherungspunkt 29

4.1 Die Architektur des Operations-Logging 36
4.2 Beispiel zum Operationsmodell 38
4.3 Externe Speicherung der Zeigerwerte 47
4.4 Code-Beispiel mit Log-Ausschnitt 49
4.5 Beispiel zur hierarchischen Fortsetzung von Markierungen 49
4.6 Die Phasen des Zurücksetzens 51
4.7 Beispielszenario für das Zurücksetzen 51
4.8 Beispiel für Kriterium 3 der Log-Bereinigung 54
4.9 Beispiel für Kriterium 2 der Log-Bereinigung 54
4.10 Markierung bei Commit 55
4.11 Markierung bei Sicherungspunkten 57
4.12 Markierung von Subobjekten 58

5.1 Die Architektur des Operations-Logging für transaktionale externe Daten . . 64
5.2 Definitionsmodul TransServers 64
5.3 Zustände und Nachrichten beim 2PC 66
5.4 Komponenten und Schnittstellen einer Instanz 68
5.5 Komponenten und Schnittstellen einer verteilten Anwendung 69
5.6 Integration eines Tycoon-Systems in das X/Open-Model 72
5.7 Die Rollen von Tycoon im X/Open-Modell 73
5.8 Ablauf des 2PC im Erfolgsfall 74
5.9 Zustand nach dem Abbruch der globalen Transaktion beim Setzen eines Synchronisationspunktes 74
5.10 Ablauf des 2PC im Fehlerfall 75
5.11 Zustand nach dem Abbruch der globalen Transaktion beim Zurücksetzen . . . 76
5.12 Wiederanlauf mit dem Transaktions-Manager 77

1. Einleitung und Motivation

Im Zuge der Neustrukturierung betrieblicher Prozesse in Unternehmen haben Workflows ([Jablonski 94]) beziehungsweise Workflowmanagementsysteme (WFMS, siehe beispielsweise [Georgakopoulos et al. 93; McCarthy, Sarin 93; Jablonski 95]) in den letzten Jahren sowohl in der Betriebswirtschaft als auch in der Informatik zunehmend an Interesse gewonnen. Workflows werden zur Implementierung rechnergestützter Vorgangsbearbeitung eingesetzt, die durch die Unterstützung durch EDV automatisiert und effizienter gestaltet werden soll. Die dazu erforderliche Basistechnologie wird durch WFMS bereitgestellt, die die Modellierung von Workflows als Abbild von Geschäftsprozessen und deren Ausführung erlauben.

In den letzten Jahren ist eine Vielzahl von kommerziellen WFMS auf den Markt gekommen und auch im Forschungsumfeld nimmt das Interesse an der Architektur von WFMS zu. Diese Arbeit soll einen Beitrag zu den Implementationsgrundlagen von WFMS liefern, insbesondere zu den Aspekten der Zuverlässigkeit und Fehlertoleranz bei der Ausführung von Workflows. Bei Workflows handelt es sich um langlebige Aktivitäten, deren Ausführung durch Transaktionen unterstützt werden muß. Die Forderung nach Transaktionsunterstützung ergibt sich auch aus dem Ausführungsumfeld von Workflows: Dieses ist durch Heterogenität und Verteilung gekennzeichnet, was zu besonders komplexen Fehlersituationen führen kann. Das Kernthema dieser Arbeit ist die Erweiterung einer offenen persistenten Programmiersprache um ein Transaktionskonzept, das auf persistenten Sicherungspunkten basiert. Diese Kombination stellt einen neuen Architekturansatz für WFMS dar, bei dem Workflows durch persistente Prozesse implementiert werden. Die Offenheit der Programmiersprache drückt sich durch die Fähigkeit aus, innerhalb eines Programmes beliebige externe Systeme über deren standardmäßig vorhandene Schnittstellen anzusprechen; damit ist Offenheit ein Schlüsselkonzept bei der Wiederverwendung existierender Software-Infrastruktur. Ein wichtiger Aspekt bei der transaktionalen Erweiterung einer offenen Programmiersprache ist die orthogonale Behandlung aller in der Programmiersprache vorkommenden Arten von Ressourcen. Bei einer offenen Programmiersprache zählen dazu auch die von externen Ressourcenmanagern verwalteten Daten, die innerhalb eines Programms angesprochen werden können.

Diese Arbeit wurde im Rahmen der Tycoon-Programmiersystems ([Matthes, Schmidt 93; Matthes 93]) ausgeführt, das eine Programmiersprache mit den oben erwähnten Eigenschaften—Persistenz, Offenheit und Erweiterbarkeit— anbietet. Der Beitrag der Arbeit besteht in der Erweiterung des Tycoon-Systems um Transaktionen mit persistenten Sicherungspunkten. Im Vordergrund stehen dabei die Fragen, wie die Architektur eines solchen Programmiersystems erweitert werden muß, um Transaktionen mit persistenten Sicherungspunkten zu integrieren, und welche transaktionalen Algorithmen zur Anwendung kommen müssen, um die verschiedenen Arten der Daten in integrierter und orthogonaler Weise in das Transaktionskonzept einzubeziehen.

Nachfolgend soll zunächst dargestellt werden, warum eine persistente Programmiersprache mit erweitertetem Transaktionskonzept eine geeignete Implementationsgrundlage für ein WFMS ist und welche Aspekte bei der Integration eines solchen erweiterten Transaktionsmodells in eine persistente Programmiersprache zu berücksichtigen sind.

1.1 Langlebige Aktivitäten in offenen Umgebungen

Workflows werden zur Abbildung betrieblicher Vorgänge herangezogen, die mit Rechnerunterstützung ausgeführt werden sollen. Diese Vorgänge erstrecken sich in der Regel über einen längeren Zeitraum, weswegen ein solcher Prozeß bei Abbildung auf einen Rechner auch als langlebige Aktivität bezeichnet wird. Langlebige Aktivitäten sind schon seit längerer Zeit ein Forschungsthema innerhalb der Informatik, insbesondere in den Gebieten der Transaktionssysteme und Datenbanken. Im folgenden werden die Begriffe Workflow und langlebige Aktivität aufgrund ihrer Gemeinsamkeiten synonym verwenden.

Um zu erkennen, welche Anforderungen von Geschäftsprozessen an WFMS gestellt werden, ist zunächst das heutige Umfeld von Geschäftsprozessen zu untersuchen. In vielen Unternehmen werden heute bereits aufgrund von Rationalisierungsmaßnahmen zahlreiche Geschäftsprozesse weitgehend durch EDV-Funktionen vollständig wahrgenommen oder zumindest unterstützt. Aber nicht nur die Geschäftsprozeße, sondern auch die EDV-Strukturen in den Unternehmen werden rationalisiert. Man kann einen allgemeinen Trend zur Ablösung von zentralen, großrechnerbasierten Systemen durch vernetzte Systeme auf Client/Server-Basis feststellen, der einhergeht mit dem Einsatz von Standardsoftware wie Textverarbeitungen oder Tabellenkalkulationen. Selbst innerhalb der Softwaresysteme wird heute immer mehr Wert gelegt auf die Unterstützung von standardisierten Schnittstellen und Protokollen, um die Interoperabilität von Softwaresystemen und -systemkomponenten zu gewährleisten. Die kennzeichnenden Merkmale eines Geschäftsprozesses in einem solchen heutzutage typischen Umfeld sollen zur Veranschaulichung an einem Beispiel dargestellt werden:

Man stelle sich dazu eine Schadensmeldung in einem Versichungsunternehmen vor. Die Schadensmeldung wird üblicherweise durch einen Telefonanruf des Versichungsnehmers initiiert und durchläuft anschließend mehrere Stadien der Bearbeitung, wie zum Beispiel Aufnahme der Schadensmeldung, Prüfung durch einen Sachverständigen und/oder durch eine Rechtsabteilung, Veranlassung einer Auszahlung, gegebenenfalls Änderung der Prämie, etc. Dieser Prozeß wird über einen längeren Zeitraum von mehreren Personen bzw. Dienststellen bearbeitet. Folglich werden die Einzelschritte des Gesamtprozesses räumlich getrennt und mit EDV-Unterstützung (z.B. Textverarbeitung, Buchhaltungssoftware oder ein Recherchesystem für Juristen) bearbeitet. Die Daten, die jeder einzelne Bearbeitungsschritt im Gesamtablauf produziert, sind häufig Eingabeparameter des Folgeschrittes und werden oftmals für längere Zeit archiviert. Beispielsweise dient die Entscheidung des Schadensprüfers, ob die geltend gemachten Ansprüche begründet sind, als Eingabe für die Zahlungsstelle, deren Auszahlung wiederum für längere Zeit archiviert wird.

Aus diesem einfachen Beispiel lassen sich bereits die zentralen Forderungen erkennen, denen ein WFMS für die Umsetzung von realen betrieblichen Prozessen genügen muß:

Verteilung: Die verteilte Bearbeitung eines Geschäftsprozesses resultiert aus der geographischen Verteilung beziehungsweise der dezentralisierten Organisation von Unternehmen, die oft

nach Aufgabenblöcken strukturiert und mit Ressourcen ausgestattet sind. Der Grund dafür ist unter anderem, daß heutige betriebliche Vorgänge zu komplex sind, um von einer einzigen Person durchgeführt werden zu können. Durch die nach Spezialisierung der Mitarbeiter erfolgende Aufgabenverteilung innerhalb eines Unternehmens ist es unumgänglich, daß mehrere Bearbeiter einem Geschäftsprozeß zugeordnet sind. Der Verteilungsaspekt von Workflows kann weiter untergliedert werden in:

- Nutzung verteilter Ressourcen für einen Bearbeitungsschritt: Beispielsweise kann der Schadensprüfer gleichzeitig auf eine Bilddatenbank und die Information über den Versicherungsnehmer zugreifen, die beide auf unterschiedlichen Systemen gespeichert sein können. Die verteilte Speicherung von Ressourcen ist heute bereits aufgrund des Trends zu Client/Server-Systemen weit verbreitet oder wird zumindest angestrebt.
- Räumlich verteilte Bearbeitung eines Vorgangs: Hierbei handelt es sich um die Tatsache, daß die Einzelschritte eines Geschäftsprozesses von unterschiedlichen Personen durchgeführt werden, die oft räumlich getrennt untergebracht sind und jeweils über eigene Rechner verfügen, mit denen die Bearbeitung vorgenommen werden soll.
- Zeitlich verteilte Bearbeitung eines Vorgangs: Es ist selbstverständlich, daß ein Vorgang, unabhängig von seinem Umfang, immer eine zeitliche Ausdehnung besitzt. Aber ein gesamter Geschäftsvorgang hat in der Regel eine bedeutend größere Ausdehnung als z.B. ein Einzelschritt wie eine Buchung in diesem Geschäftsvorgang. Analog zum vorhergehenden Punkt der räumlichen Verteilung werden die Einzelschritte eines Geschäftsprozesses auch zu unterschiedlichen Zeitpunkten ausgeführt, die oftmals durch längere Phasen der Inaktivität getrennt sein können.

Diese drei Arten der Verteilung sind voneinander unabhängig und treten oft gemeinsam auf. Ein WFMS muß deshalb die technische Grundlage zur Realisierung dieser drei Arten der Verteilung in einem Workflow bereitstellen. Verteilte Ressourcen werden durch Client/Server-Programmierung nutzbar gemacht, die technisch z.B. mit einem *remote procedure call* realisiert werden kann. Zur Unterstützung der örtlichen Verteilung kann der Workflow auf den Rechner des aktuellen Bearbeiters migriert werden. Um die zeitliche Verteilung angemessen zu berücksichtigen, muß das WFMS verhindern, daß Ergebnisse der bereits erledigten Bearbeitungsschritte verlorengehen, beispielsweise indem diese persistent gespeichert werden.

Fehlerresistenz: Um die korrekte Ausführung von Workflows zu gewährleisten, muß verhindert werden, daß von dem Workflow produzierte Daten verloren gehen. Zu diesen Daten gehören die langfristig zu speichernden Daten der Verarbeitungsschritte sowie weitere von diesen produzierte Ausgabedaten, die als Eingabe für Folgeschritte dienen. Ebenfalls dazuzurechnen ist der Bearbeitungszustand, der zwar nur für die Zeitdauer des Workflows existiert, aber dennoch ein Datum darstellt, da der Workflow auf einem Rechner ausgeführt wird. Weil Fehlersituationen wegen der Verteilung der Workflows häufig auftreten können, dürfen sie nicht ignoriert werden. Eine höhere Fehleranfälligkeit ergibt sich zunächst aus der räumlich verteilten Bearbeitung, da zur korrekten Ausführung eines Teilvorgangs alle benutzten Subkomponenten fehlerfrei funktionieren müssen. Erschwerend kommt hinzu, daß auch aufgrund der zeitlichen Ausdehnung des Workflows die Wahrscheinlichkeit steigt, daß er in seinem Verlauf mit einem Fehler konfrontiert wird. Es ist offensichtlich, daß auf eine solche Fehlersituation nicht mit dem vollständigen Abbruch des Workflows reagiert werden kann.

Der Verteilungsaspekt von Workflows ist nicht nur für eine erhöhte Fehleranfälligkeit verantwortlich, sondern auch für eine erhöhte Fehlerkomplexität, da in verteilten Systemen die Fehlermöglichkeiten weit komplexer sind als in zentralen. Weil die Subkomponenten in einem verteilten System unabhängig voneinander ausfallen können, steigt die Anzahl der zu unterscheidenden Fehlersituationen exponentiell mit der Größe des verteilten Systems. Bei der Erstellung eines Workflows stellt sich deswegen das Problem, auf eine Fehlersituation adäquat zu reagieren und von ihr aus wieder in einen konsistenten Zustand zu gelangen.

Um den Anforderungen nach Fehlertoleranz und -beherrschbarkeit nachzukommen, hat sich in der Datenbankwelt schon früh das Transaktionskonzept ([Eswaran et al. 76]) etabliert, welches durch die vier Eigenschaften Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (auch kurz durch das Akronym *ACID* zusammengefaßt) charakterisiert wird. Auf die Ausführung von Workflows übertragen schützt die Garantie der Dauerhaftigkeit vor Datenverlust. Die Eigenschaft der Atomarität erlaubt es dem Workflow-Programmierer, konsistente Ausführungszustände zu markieren, auf die im Fehlerfall automatisch zurückgekehrt werden kann. Mit zusätzlich entwickelten Techniken der verteilten Transaktionsverarbeitung lassen sich auch die ACID-Eigenschaften bei Bearbeitung von verteilt vorliegenden Datenbeständen garantieren.

Bei OLTP-Systemen ([Gray, Reuter 93]) handelt es sich bereits um rechnergestützte Vorgangsbearbeitungssysteme, die ein transaktionales Verhalten aufweisen. Diese sind allerdings für Workflows ungeeignet, da sie nur für Transaktionen von sehr kurzer Dauer gedacht sind (siehe dazu auch [Gray et al. 81; Wächter, Reuter 91]). So ist es beispielsweise in einem OLTP-System angemessen, auf eine Fehlersituation mit dem vollständigen Abbruch eines Vorgangs zu reagieren.

Einbeziehung der vorhandenen Software-Infrastruktur: Ein Workflow hat häufig die Aufgabe, bereits bestehende Einzelvorgänge zu integrieren und zu verbinden. Am Beispiel der Schadensmeldung in einer Versicherung wird deutlich, daß für viele Einzelschritte von Geschäftsprozessen schon ein Softwaresystem eingerichtet wurde, mit dessen Hilfe die Bearbeitung vereinfacht wird. Eine rechnergestützte Implementierung eines Geschäftsprozesses, die alle diese Einzelschritte verbindet, koordiniert und kontrolliert, muß allein aus ökonomischen Gründen die bestehende Software-Infrastruktur einbeziehen können. Häufig handelt es sich bei den zentralen Applikationen eines Unternehmens um sogenannte Altsysteme (*legacy systems*), die nur unter hohem finanziellen und zeitlichen Aufwand sowie teilweise auch unter erheblichem Risiko für das Unternehmen reimplementiert werden könnten. Bei neuen Softwaresystemen geht der Trend zu kommerziell erhältlichen Standardkomponenten, wie z.B. Textverarbeitungssytemen, Tabellenkalkulationen und relationalen Datenbanksystemen. Dies resultiert nicht nur aus den kostengünstigeren Beschaffungsmöglichkeiten, die hinsichtlich dieser Software bestehen, sondern auch daraus, daß durch ihren Verbreitungsgrad ein hohes Maß an Vorwissen bei den Endbenutzern existiert und dadurch teure Schulungen eingespart werden können. Es ist also unerläßlich, daß ein WFMS die Möglichkeit zum Ansteuern externen Systeme bietet.

Aus diesen Anforderungen (Verteilung, Fehlerresistenz und Einbeziehung der vorhandenen Software-Infrastruktur), die für eine erfolgreiche EDV-gestützte Abbildung von Geschäftsprozessen erfüllt sein müssen, lassen sich Designziele für WFMS ableiten. Diese wurden schon

vereinzelt in den vorhergehenden Absätzen erwähnt, sollen hier aber noch einmal im Zusammenhang aufgeführt werden. Aus diesen Designzielen läßt sich auch erkennen, warum ein offenes persistentes Programmiersystem eine geeignete Basis eines WFMS darstellt und warum ein erweitertes Transaktionskonzept eine notwendige Ergänzung ist.

Eine grundlegende Fähigkeit eines WFMS ist die Realisierung einer Ablaufkoordination für die jeweiligen Einzelschritte des Workflows, wobei davon auszugehen ist, daß viele dieser Einzelschritte durch Rechnerunterstützung bereits automatisiert sind. Das System muß also die Möglichkeit bieten, die externen Systeme anzusprechen und die Einzelfunktionen, z.B. durch Schleifen und bedingte Verzweigung, zu kombinieren.

Dies weist auf eine Programmier- oder Skriptsprache hin, die auch Aufrufe an externe, getrennt vorliegende Bibliotheken und Betriebssystemschnittstellen zuläßt. Damit ist die Möglichkeit gegeben, beliebige externe Systeme, mit denen die zu kombinierenden Einzelschritte realisiert wurden, anzusprechen. Liegt ein solches externes System nicht in Form einer Bibliothek vor, kann es über die Kommunikationsmittel des Betriebssystems angesprochen werden. Da mit Workflows teilweise sehr komplexe Abläufe abgebildet werden sollen, muß die Programmiersprache orthogonal und modular aufgebaut sein, um den Programmierer bei seiner Tätigkeit zu unterstützen (siehe auch [Jablonski 95]). Damit die Einzelschritte selbst auch in der Programmiersprache geschrieben werden können, sollte auch die Möglichkeit der prozeduralen Abstraktion vorhanden sein.

Um die Robustheit der Workflows zu erhöhen, sollte das WFMS die migrierende Ausführung von Workflows unterstützen. Ziel ist es, den Workflow zu jedem Zeitpunkt auf dem Rechner des aktuellen Bearbeiters autonom auszuführen; es sollte nicht notwendig sein, den Workflow von einem zentralen Rechner, z.B. dem Ursprungsrechner, aus zu steuern. Durch die so gewonnene Autonomie wird die Wahrscheinlichkeit verringert, daß der Workflow wegen des Ausfalls eines Knotens im Netz nicht weiter ausgeführt werden kann. Die Migration führt darüberhinaus zu einer erhöhten Lokalität der Ausführung und damit zu einem reduzierten Kommunikationsbedarf ([Mathiske et al. 95]); mit dem verringerten Bedarf an Netzwerkressourcen erhöht sich zugleich die Performance. Ein weiterer Vorteil ist die erhöhte Skalierbarkeit des gesamten verteilten Systems, da kein zentraler Serverrechner mit der Workflowausführung belastet wird. Die migrierende Ausführung unterscheidet sich dabei wesentlich von einer verteilten Ausführung, wie sie z.B. mit Remote Procedure Calls (RPC) erreicht wird, da hierbei die Klientenapplikation stationär auf einem Rechner bleibt. Die Möglichkeit der Client/Server-Programmierung, beispielsweise mit RPCs, gewinnt unabhängig davon an Bedeutung, wenn Einzelschritte, die auf dem Client/Server-Prinzip beruhen, auch mit dem System programmiert werden sollen. In der Regel kann dies durch den Aufruf externer Bibliotheken erreicht werden.

Als letzter Punkt muß schließlich den komplexen Fehlersituationen, die in verteilten Systemen auftreten können, durch ein geeignetes Transaktionsmodell Rechnung getragen werden. Dieses Transaktionskonzept muß es ermöglichen, konsistente Bearbeitungszustände zu markieren, damit der vollständige Zustand des Workflows zum Zeitpunkt der letzten Markierung wiederhergestellt werden und so einem Gesamtverlust aller Daten vorgebeugt werden kann. Der Workflow sollte auch auf die markierten Zustände zurückgesetzt werden können; auf diese Weise kann eine einfache Fehlerbehebung mit größtmöglicher Unterstützung des WFMS durchgeführt werden. Die Bedeutung der transaktionalen Unterstützung für die robuste und zuverlässige Ausführung von Workflows wird schon von einer Reihe anderer Arbeiten auf dem Gebiet der WFMS hervorgehoben (vergleich dazu [Alonso et al. 94; Sheth, Rusinkiewicz 93;

Hsu et al. 93; McCarthy, Sarin 93]).

Das Tycoon-Programmiersystem ([Matthes, Schmidt 93]) genügt bereits den meisten der genannten Anforderungen und bietet sich damit als Basis zur Implementierung eines WFMS an. Die in Tycoon verfügbare Programmiersprache TL ([Matthes, Schmidt 92]) ist eine polymorph typisierte, funktionale Programmiersprache, die nicht nur ausdrucksmächtig ist, sondern auch durch ein Modul- und Bibliothekskonzept ([Mathiske et al. 93]) die Programmierung im Großen unterstützt. Zudem lassen sich in Tycoon Funktionen in externen Bibliotheken aufrufen; es besteht sogar die Möglichkeit, in TL geschriebene Funktionen von externen Systemen aufrufen zu lassen (Callback-Möglichkeit). Durch die statische polymorphe Typisierung von TL lassen sich externe Dienste nahtlos in Tycoon integrieren, weshalb sich Tycoon als integrierte Umgebung zur Programmierung der Einzelschritte und zu ihrer anschließenden Kombinierung zu Workflows anbietet. Durch die Schnittstelle zu externen Bibliotheken stehen dem Programmierer auch die üblichen Methoden der verteilten Programmierung offen; darüberhinaus wird von Tycoon ein Konzept von migrierender Ausführung in Form von Threadmigration ([Mathiske et al. 95]) realisiert, das als Grundlage für ein orthogonales Migrationsverfahren dienen kann. Durch eine Threadmigration wird der Tycoon-interne Ausführungszustand eines Threads bzw. eines Programmes auf ein Zielsystem übertragen; im Anschluß an diese Migration kann der Thread autonom vom Ursprungssystem weiter ausgeführt werden.

Was zu Beginn der Arbeit von Tycoon noch nicht abgedeckt wurde, war ein erweitertes, allumfassendes Transaktionskonzept. Tycoon unterstützte zwar Transaktionen, allerdings waren diese aus zwei Gründen nicht für die Realisierung von Workflows geeignet: Einmal bietet das in Tycoon verwirklichte klassische, flache Transaktionsmodell keine adäquate Unterstützung für Workflows. Zum anderen wurde das Transaktionskonzept auch nicht auf den vollständigen, auch externe Daten umfassenden Ausführungszustand angewendet, so daß sich die Transaktionsbefehle nur auf die Tycoon-internen Daten auswirkten. Damit war die Fehlertoleranz, durch die der letzte konsistente Zustand *vollständig* wiederhergestellt werden muß, nicht gegeben. Ebenso machte das Transaktionskonzept in der zu Beginn der Arbeit vorliegenden Implementierung die in einem Programm auftretenden Fehlersituationen nicht weniger komplex, da ein automatisches Zurücksetzen auf einen als konsistent markierten Zustand nur für interne Daten, nicht aber für die eventuell weitaus wichtigeren externen Daten möglich war.

Tycoon muß also mit einem für Workflows geeigneten Transaktionskonzept ausgestattet werden, welches auch extern verwaltete Daten miteinbezieht. Aus diesen Defiziten läßt sich auch die technische Aufgabenstellung ableiten, die für die Erweiterung von Tycoon zu einem WFMS zu bearbeiten ist und die das Kernthema dieser Arbeit darstellt.

1.2 Zielsetzung und Überblick

Ziel dieser Arbeit ist es, Tycoon um ein orthogonales Transaktionskonzept zu erweitern, das auch die besonderen Anforderungen von Workflows berücksichtigt. Im Mittelpunkt steht dabei die Entwicklung von Verfahren zur integrierten transaktionale Behandlung aller innerhalb eines Tycoon-Programmes anfallenden Daten. Mit Hilfe dieser Verfahren sollen insbesondere auch beliebige externe Daten transaktional verwaltet werden können, so daß ähnlich wie die Persistenz auch die Transaktionalität der Ausführung zu einer orthogonalen Eigenschaft des Tycoon-Systems wird. Als Transaktionsmodell wird eine Ergänzung flacher Transaktionen um persistente Sicherungspunkte zugrunde gelegt. Bei der Auswahl des Transaktionsmodells

für Tycoon wurde die Möglichkeit des transaktions-internen Parallelismus (der z.B. durch geschachtelte Transaktionen [Moss 82] erreicht wird) ausgeklammert und stattdessen von rein sequentieller Bearbeitung ausgegangen. Ebenso wurde davon abgesehen, bei der Entwicklung der transaktionalen Verfahren Aspekte, die durch Mehrbenutzerzugriff entstehen, einzubeziehen. Diese beiden Einschränkungen wurden gemacht, um einen dem Rahmen einer Diplomarbeit angemessenen Aufgabenumfang zu erhalten.

Die Arbeit ist im weiteren Verlauf wie folgt unterteilt: Ein Begründung des gewählten Transaktionsmodells sowie eine Darstellung der damit erweiterten Tycoon-Architektur wird in Kapitel 2 vorgenommen. Als Grundlage für die Abwägung des Transaktionsmodells wird vorausgehend erörtert, was der Transaktionsbegriff im Kontext von Programmiersprachen bedeutet. Im Anschluß wird im Vergleich mit anderen Transaktionsmodellen gezeigt, warum persistente Sicherungspunkte eine geeignete Grundlage für Workflows sind und angesprochen, wie diese in Tycoon zu integrieren sind. Eine Klassifizierung der unterschiedlichen, transaktional zu verwaltenden Ressourcen in persistenten Programmiersystemen bildet den Abschluß und bereitet die Darstellung der transaktionalen Algorithmen in den darauffolgenden drei Kapiteln vor.

Kapitel 3 behandelt die Realisierung von persistenten Sicherungspunkten für Tycoon-interne Zustände und stellt dar, wie diese in die Store-Architektur integriert sind. Die Erweiterung der flachen Transaktionen in Tycoon wird durch eine Logging-Zwischenschicht realisiert, die auf die Store-Schnittstelle aufgesetzt wird. Es wird erläutert, wie die Transaktionsalgorithmen zum Setzen von Sicherungspunkten und zum Zurücksetzen in der Zwischenschicht realisiert sind. Um einen Eindruck von den durch Logging entstehenden Zusatzkosten in der Laufzeit zu erlangen, werden diese am Ende anhand einiger Beispiele abgeschätzt und durch Vergleichsmessungen illustriert.

Das vierte Kapitel beschäftigt sich mit den transaktionalen Verfahren für flüchtige externe Daten, ein Bereich, der von den existierenden persistenten Programmiersystemen (zum Beispiel [Dearle et al. 89; Copeland, Maier 84; Maier et al. 86; Bancilhon et al. 92]) bislang ignoriert wurde. Um einen Anhaltspunkt für die Bewertung der Verfahren zu erhalten, beginnt dieser Abschnitt mit der Festlegung von Effizienzkriterien für transaktionale Operationen auf flüchtigen Daten. Aus diesen Kriterien und einem abstrakten Modell externer Operationen werden dann transaktionale Verfahren abgeleitet, die auf dem Logging von Operationen basieren und eine Kooperation zwischen Anwendung und externen Schnittstellen voraussetzen. Diese Darstellung der Algorithmen wird durch eine Anleitung zum Einsatz der Logging-Funktionen am Beispiel eines Fenstersystems abgerundet.

Das fünfte Kapitel beschreibt, wie transaktionale externe Dienste, beispielsweise Datenbanken, in das Tycoon-Transaktionskonzept eingebunden werden müssen. Die Schwierigkeit besteht darin, daß externe transaktionale Dienste in der Regel nur flache Transaktionen unterstützen. Sie werden deshalb mit einer zweistufigen Architektur integriert, in der ein Sicherungspunkt in Tycoon auf eine abgeschlossene Transaktion im externen System abgebildet wird. Ein Zurücksetzen auf einen früheren Sicherungspunkt wird durch Ausführung von kompensierenden Operationen erreicht.

Kapitel 6 bildet mit einer Zusammenfassung der Arbeit den Abschluß. Dabei werden in einem Ausblick auch die Erweiterungsmöglichkeiten der entwickelten Verfahren sowie weitere mit der Aufgabenstellung zusammenhängende Forschungsaspekte angesprochen.

2. Tycoon als transaktionale Umgebung

Zur Unterstützung langlebiger Aktivitäten muß das Tycoon-System ([Matthes 93]) mit einem dafür angemessenem Transaktionsmodell ausgestattet werden. Langlebige Aktivitäten sollten aus Gründen der Ablaufkontrolle durch strukturierte Transaktionen implementiert werden, weswegen die in Tycoon vorzufindenden flachen Transaktionen ungeeignet sind. Als Kompromiß zwischen Ausdrucksmächtigkeit und Realisierbarkeit, besonders im Hinblick auf die Einbindbarkeit externer Dienste, ist eine Strukturierung flacher Transaktionen durch persistente Sicherungspunkte zu sehen. Eine langlebige (Trans-) Aktion wird dabei in sequentielle Teilabschnitte partitioniert, die durch die Sicherungspunkte begrenzt werden. Die Persistenz der Sicherungspunkte garantiert nach einem Ausfall, daß der letzte durch einen Sicherungspunkt markierte Zustand wiederhergestellt werden kann. Da transaktionale externe Dienste wie z.B. Datenbanksysteme nur flache Transaktionen unterstützen, müssen diese durch einen zweistufigen Ansatz in Tycoon integriert werden, der zum Beispiel an ConTracts [Wächter, Reuter 91] oder Sagas ([Garcia-Molina, Salem 87]) erinnert.

Als Grundlage für die Diskussion erörtert der nächste Abschnitt die Bedeutung des Transaktionskonzeptes im Kontext einer persistenten Programmiersprache. Die übliche Definition einer Transaktion läuft auf die ACID-Eigenschaften hinaus, die aber für Datenbanksysteme geprägt wurden. Inwieweit diese Eigenschaften noch für persistente Programmiersprachen anwendbar sind und welche Aussagen sie in diesem Zusammenhang machen, wird im folgenden Teilabschnitt geklärt. Die daran anschließende Begründung der Wahl des Transaktionsmodells geht insbesondere auf die Bedürfnisse langlebiger Aktivitäten ein. Nach einer kurzen Darstellung der in Tycoon realisierten Transaktionsschnittstelle für persistente Sicherungspunkte wird das Kapitel mit einer Klassifikation der in offenen persistenten Programmiersprachen vorkommenden Ressourcen abgeschlossen. Die Klassifikation reflektiert die für die transaktionalen Algorithmen relevanten Charakteristika der Ressourcen und begründet die Gliederung der Kapitel 3–5.

2.1 Bedeutung des Transaktionskonzeptes für persistente Programmiersprachen

Das Transaktionskonzept entstammt ursprünglich dem Datenbankbereich ([Eswaran et al. 76]), wo es als Konstrukt eingeführt wurde, um dem Programmierer ein einfache Semantik seiner Interaktionen mit der Datenbasis zu geben. Eine Transaktion zeichnet sich nach dieser Definition durch vier Eigenschaften aus, die man auch als ACID-Eigenschaften bezeichnet.

Diese Eigenschaften machen Aussagen über das zu beobachtende Verhalten von Transaktionen im Bezug auf Seiteneffekte. Die Eigenschaften sind:

Atomarität: Die Zustandsänderungen in der Datenbank, die durch die Transaktion hervorgerufen wurden, sind nach außen hin vollständig oder gar nicht sichtbar. Zur Wahrung dieser Eigenschaft existiert ein Mechanismus in Datenbanksystem, mit dem bei einem Systemfehler, der einen Absturz des DBMS und des ganzen Systems zur Folge hat, alle Zustandsänderungen seit dem letzten Transaktionsabschluß automatisch rückgängig gemacht werden. Die Möglichkeit zu einem solchen Verhalten wird dem Transaktionsprogrammierer auch als Mittel zur Fehlerbehandlung in Form der *Rollback*-Anweisung explizit zur Verfügung gestellt.

Konsistenz: Ausgehend von einem konsistenten Zustand der Datenbasis führt eine vollständige Transaktion zu einem neuen aber nach wie vor konsistenten Zustand. Hierbei muß erwähnt werden, daß der Konsistenzbegriff ein rein datenspezifischer ist und die Konsistenz der Änderungen durch eine Transaktion auch nur durch von der Semantik der Daten abhängige Prüfungen sichergestellt werden kann. Das Transaktionskonzept selbst beschäftigt sich nicht mit der Semantik der zu verwaltenden Daten. Es existieren auch sogenannte aktive Datenbanksysteme (sieh zum Beispiel [McCarthy, Dayal 89; Stonebraker et al. 87]), die die Definition konsistenzerhaltender Regeln erlauben. Diese können dann zum Transaktionsende überprüft werden und u.U. den Abbruch einer konsistenzverletzender Transaktions bewirken. Es soll hier festgehalten werden, daß die Konsistenzprüfung keine essentielle Funktionalität einer Implementierung von transaktionalen Eigenschaften ist und oft unabhängig von diesen implementiert wird.

Isolation: Eine einzelne Transaktion sieht die Datenbasis so, als ob sie der alleinige Benutzer wäre. Anders ausgedrückt: eine Transaktion nimmt selber das Vorhandensein anderer, parallel ablaufender Transaktionen nur bedingt war. Eine Transaktion sieht zwar die Effekte (erfolgreich) abgeschlossener Transaktionen, aber nicht solcher, die parallel ablaufen. Die Isolationseigenschaft erleichtert in großem Maße die Aufgabe des Transaktionsprogrammierers, da dieser sich nicht mit den komplexen und mannigfaltigen Interaktionsmöglichkeiten nebenläufiger Zustandsänderungen auseinandersetzen muß, sondern die Transaktionen so programmieren kann, als ob diese in einem Einbenutzersystem ausgeführt würden. Wären nebenläufige Transaktionen nicht voneinander isoliert, würden diese z.B. auch inkonsistente Zustände der Datenbasis sehen können und entsprechend darauf reagieren müssen.

Dauerhaftigkeit: Die Seiteneffekte einer erfolgreich abgeschlossenen Transaktion müssen dauerhaft gespeichert und auch gegen Fehler geschützt sein. In dieser Forderung drückt sich aus, daß es sich bei Datenbanksystemen um persistente Systeme handelt, die auch noch speziellen Fehlertoleranzanforderungen genügen müssen. Durch die Zusicherung der Dauerhaftigkeit wird der Programmierer von der Aufgabe entlastet, die Änderungen durch eigens zu spezifizierende Maßnahmen selber persistent zu machen; hierdurch wird wieder die Komplexität der Programmiertätigkeit reduziert. Durch Dauerhaftigkeit wird aber auch der Aspekt der Fehlertoleranz eingebracht, weil die gespeicherten Änderungen auch gegen Folgefehler geschützt sein müssen, die lange nach Beendigung der Transaktion auftreten könnten. Die Behandlung solcher Fehler liegt natürlich nicht im Einflußbereich der Transaktion, da diese

zu dem Zeitpunkt nicht mehr existiert; folglich kann diese Fehlertoleranz nur eine Eigenschaft des DBMS sein.

Zusammengefaßt kann man sagen, daß das Transaktionskonzept mehrere Aspekte in sich vereint:

- Die Nebenläufigkeit bei der Nutzung gemeinsamer Datenbestände wird durch Transaktionen maskiert, und ein Sonderverhalten, welches aus dem Vorhandensein von Nebenläufigkeit entsteht, wird ausgeschlossen. Die Nebenläufigkeit wird dabei nicht vollständig ausgeschlossen, sondern nur durch die Forderung der Transaktion nach Serialsierbarkeit eingeschränkt.

- Der Programmierer kann Transaktionen als Programmierkonstrukt mutzen, um die Komplexität seiner Anwendungen zu reduzieren. Beispielsweise erlaubt die *Rollback*-Anweisung ein kontrolliertes Zurücksetzen des Datenbankzustandes auf einen früheren konsistenten Zustand, was die Behandlung semantischer Fehler und Ausnahmesituationen sehr erleichtert. Die implizite Persistenz der Änderungen beim Transaktionsende nehmen dem Programmierer ebenfalls Arbeit ab.

Diese Aspekte verlieren teilweise an Bedeutung, wenn man von einem Mehrbenutzer-DBMS in den Kontext einer persistenten Programmiersprache übergeht. Der wichtigste Punkt ist, daß es in persistenten Programmiersprachen eine anderer Form von Nebenläufigkeiten gibt als in DBMS. In DBMS geht man von dem Vorhandensein mehrere konkurrierender Transaktionen aus: die Transaktionen bearbeiten nicht in kooperativer Weise dieselbe Aufgabenstellung, sondern jede Transaktion hat eine eigene Aufgabe und benötigt zu ihrer Bearbeitung Ressourcen, die sie sich mit anderern Transaktionen teilen muß. In persistenten Programmiersprachen hingegen existiert dieses Konzept der konkurrierenden, nebenläufigen Prozesse nicht: wenn auf demselben System zwei Prozesse dasselbe Programm ausführen, wird jeder der Prozesse mit seiner eigenen Kopie der Programmvariablen ausgestattet. Demhingegen kann ein Programm mehrere Threads beinhalten, die parallel auf dieselben Ressourcen zugreifen können; dies geschieht allerdings oft in kooperativer Form. Hierfür wäre aber das Isolationsprinzip nicht passend, weil zur Kooperation der Threads eine Durchbrechnung der Isolation teilweise unumgänglich ist. Für eine Unterstützung der Programmierung von Transaktionen mit internem Parallelismus wäre also eine kontrollierte Aufgabe der Isolation erforderlich, wie sie von erweiterten Transaktionsmodellen, die speziell Parallelität berücksichtigen, angeboten werden. Um das Themengebiet, welches in dieser Arbeit behandelt wird, überschaubar zu halten, habe ich bewußt diesen Bereich ausgeklammert und mich auf die Behandlung von persistenten Einbenutzerprogrammiersystemen beschränkt. Das Tycoon-System beinhaltet zwar Threads, die für die Programmierung paralleler Applikationen genutzt werden können, bei der Benutzung transaktionaler Eigenschaften muß die Nebenläufigkeit der Threads aber vom Programmierer kontrolliert werden.

Aus dem vorigen Absatz ergibt sich, daß wir in persistenten Programmiersprachen auf eine Behandlung der Nebenläufigkeit, wie sie in DBMS zu finden ist, absehen können. Eine persistente Programmiersprache soll sich also wie ein Einbenutzer-Datenbanksystem verhalten. Daraus ergibt sich, daß Transaktionen in persistenten Systemen vorwiegend den Aspekt der Komplexitätsreduzierung für den Programmierer haben und somit ein Programmierkonstrukt darstellt, welches innerhalb der Sprache verfügbar sein muß (in welcher Form wird damit nicht

gesagt). Diese „Aufgabenreduzierung“ von Transaktionen ist der Grund, warum diese in persistenten Programmiersprachen durch weniger als das ACID-Prinzip definiert werden. Die ACID-Eigenschaften haben in diesem Umfeld folgende Bedeutung:

Atomarität: Die Zustandsänderungen einer begonnenen Transaktion sollen weiterhin automatisch durch das System zurückgenommen werden, wenn die Transaktion durch einen Fehler unterbrochen wird. Ebenso wird gefordert, daß der Programmierer die Möglichkeit hat, selber das Zurücksetzen (z.B. als Reaktion auf einen Fehler) zu initiieren. Diese letzte Möglichkeit der vereinfachten Fehlerbehandlung wurde schon als ein für Workflows wichtiges Merkmal von Transaktionen angesprochen und wird bei der späteren Auswahl eines Transaktionsmodells eine wichtige Rolle spielen. Bei Verwendung eines offenen persistenten Systems, welches die Benutzung von extern verwalteten Ressourcen zuläßt, muß sichergestellt werden, daß alle von einem Programm angesprochenen Daten in diesem Rollback-Mechanismus eingeschlossen sind. Bei Datenbanksystemen als externen Ressourcenmanager kann man sich leicht vorstellen, daß sich die Rollbackfähigkeit nach außen tragen läßt. Dies soll aber auch für solche Ressourcen gelten, die noch nicht mit transaktionalen Eigenschaften ausgestattet sind. Dadurch ergibt sich die größtmögliche Vereinfachung für den Programmierer und Mächtigkeit des Rollback-Mechanismus, weil externe Daten nicht einer aufwendigen Sonderbehandlung bedürfen. Ob und wie dieser Aspekt des transaktionalen Verhaltens herstellbar ist wird in den Kapiteln 4 und 5 näher erläutert.

Konsistenz: Auch in persistenten Programmiersprachen ist die Konsistenzeigenschaft von der Semantik der Daten bestimmt. Das System kann aber Möglichkeiten der Validierung wie z.B. durch Zusicherungen (Ref Ingrid?) zur Verfügung stellen, die vom Applikationsprogrammierer genutzt werden. Einige der Möglichkeiten der Konsistenzprüfung könnten auch wie bei aktiven Datenbanksystemen mit dem Transaktionssubsystem im Laufzeitsystem der pers. PS zusammenarbeiten. Die Durchführung der Konsistenzprüfung kann aber größtenteils unabhängig von dem Transaktionssubsystem geschehen oder kann sehr leicht integriert werden. Die Konsistenzeigenschaft werden deshalb aus dem Kern der Transaktionseigenschaften ausgeklammert.

Isolation: Wie bereits erwähnt entfällt dieser Aspekt für persistente Programme an sich, da die Aufgabenstellung auf Einbenutzersysteme beschränkt wurde und auch programminterne Nebenläufigkeit ignoriert werden soll. Der Isolationsaspekt ist aber trotzdem von Bedeutung, weil andere, mittelbar über externe Aufrufe verwaltete Daten selbstverständlich von der nebenläufigen Ausführung anderer Transaktionen betroffen sein können. In diesem Fall können natürlich die nebenläufigen Änderungen nicht ignoriert werden, allerdings können sie auch nicht von dem persistenten Programmiersystem gesteuert werden, da nicht alle Zugriffe auf die externen Ressourcen über das persistente System erfolgen. Demzufolge muß die Isolationseigenschaft für externe Ressourcen von deren Managementsystem implementiert werden und kann nicht durch Maßnahmen in dem aufrufenden persistenten System erreicht werden.

Dauerhaftigkeit: Auch in persistenten Programmiersprachen soll gelten, daß nach erfolgreichem Abschluß einer Transaktion der damit erreichte Zustand dauerhaft gemacht wurde und

auch gegen spätere Ausfälle geschützt ist.[1] Bei Betrachtung eines offenen persistenen Systems fällt auf, daß die Dauerhaftigkeit auch auf extern angelegte und verwaltete Daten ausgedehnt werden muß, wenn der vollständige Ausführungszustand persistent gemacht werden soll. Dies ist eine nicht-triviale Aufgabe für das Transaktionssubsystem eines offenen persistenten Programmiersystems, weil flüchtige Daten zu den genutzten externen Ressourcen zählen.

Zusammenfassend kann also folgendes über die Rolle und Auswirkung von Transaktionen in persistenten Programmiersprachen gesagt werden:

- Transaktionen sollen ein Programmierkonstrukt in der Programmiersprache sein, welches die Markierung konsistenter Zustände erlaubt und dem Programmierer gestattet, bei Auftreten von Fehler oder auch sonst unter Umgehung des normalen Kontrollflusses auf diese markierten Zustände zurückzuspringen. Durch so einen Rücksprung wird der normale Kontrollfluß durchbrochen, weil der gesamte Ausführungszustand des Prozesses einschließlich dessen Position im Code zurückgesetzt wird.

- Transaktionen stellen eine Systemeigenschaft dar, die Persistenz beinhaltet und um den Atomaritätsbegriff erweitert. Praktisch heißt das, daß bei Neustart eines persistenten Prozesses automatisch der Zustand des letzten Synchronisationspunktes wiederhergestellt werden muß.

- Die Transaktionseigenschaften müssen sich auf alle von einem Programm bearbeiteten Ressourcen beziehen. Demzufolge müssen durch Transaktionen auch konsistente externe Zustände markiert werden können und beim Zurücksetzen muß der externe Zustand wiederhergestellt werden. Ebenso muß die Persistenz der externen Daten zum Zeitpunkt des Transaktionsendes vom persistenten Programmiersystem sichergestellt werden.

Diese Charakterisierung von Transaktionen in persistenten Programmiersprachen soll im folgenden als Definition und Aufgabenzettel bei der Implementierung dienen. Bisher wurde bewußt darauf verzichtet, auf ein bestimmtes Transaktionsmodell bezug zu nehmen; die bisherigen Aussagen, die über Transaktionen getroffen wurden, gelten unabhängig von einem bestimmten Transaktionsmodell. Wie die genaue Form eines Transaktionsmodelles für Workflows aussehen soll, ist Bestandteil des nächsten Abschnittes.

2.2 Möglichkeiten der transaktionale Strukturierung von Programmen

Bei der Auswahl eines Transaktionsmodelles müssen Faktoren wie die Funktionalität gegen die durch Workflows gegebenen Rahmenbedingungen (Verteilung und Berücksichtigung von existierender Infrastruktur) abgewogen werden. Dieses wird dazu führen, funktional mächtigere Modelle, die aber nicht in die bestehende Infrastruktur integriert werden können, verworfen

[1] Der Schutz gegen spätere Ausfälle kann in einfachster Weise durch das Anfertigen von Bandsicherungen der persistenten Daten des Systems gemacht werden. Hochgeschwindigkeits-Datenbanksysteme sehen hier zwar noch andere Mechanismen vor ([Mohan et al. 92]), diese werden aber nur erforderlich, weil diese Systeme sehr großen Durchsatz- und Verfügbarkeitsanforderungen genügen müssen. Da wir mit solchen Anforderungen in persistenten Programmiersystemen selten konfrontiert sind, wollen wir diese gesonderten Fehlertoleranzmechanismen vernachlässigen.

werden müssen. Stattdessen wird sich herausstellen, daß eine einfache Erweiterung von flachen Transaktionen um persistente Sicherungspunkte bereits eine ausreichende Mächtigkeit mit sich bringt und auf der anderen Seite noch implementierbar ist.

Bevor man ein optimales Transaktionsmodell für Workflows auswählt, muß man sich fragen, welche qualitativen Unterschiede zwischen den Modellen bestehen und welche Aspekte der Modelle die für Workflows benötigte Funktionalität beeinflussen. Allen Transaktionsmodellen ist in irgendeiner Form gemeinsam, daß sie Daten dauerhaft speichern und Zustandsübergänge atomar machen; wie sich dies in persistenten Programmiersprachen auswirken soll wurde bereits im vorangegangenen Abschnitt festgelegt. Worin sich Transaktionsmodelle aber unterscheiden sind die Möglichkeiten zur Strukturierung der Transaktionen, insbesondere spielen die Möglichkeiten zum Setzen von *Synchronisationspunkten* eine entscheidende Rolle. Unter Synchronisationspunkten sollen die innerhalb einer Transaktion gesetzten Markierungspunkte zu verstehen sein, auf die mit Hilfe einer *Rollback*-Anweisung zurückgesprungen werden kann. Ein Extrembeispiel sind die klassischen, flachen Transaktionen, die zu jedem Zeitpunkt die Existenz von nur einem Synchronisationspunkt, dem Transaktionsanfang, unterstützen. Ein anderes Beispiel sind geschachtelte Transaktionen, die mit jeder neuen Subtransaktion einen neuen Synchronisationspunkt festlegen. Da ein Bestandteil von transaktionalen Systemen die Fähigkeit zum Zurücksetzen ist, bestimmen die Möglichkeiten für das Setzen von Synchronisationspunkten die Flexibilität des Programmierers. Gibt es keine Einschränkungen bzgl. der Möglichkeiten für Synchronisationspunkte, kann u.U. besser auf Fehlersituationen reagiert werden, weil es einen Synchronisationspunkt gibt, der genau diese Fehlersituation zurücknimmt, aber nicht mehr. Transaktionsmodelle unterscheiden sich auch in den Möglichkeiten zum intra-transaktionalen Parallelismus, was jedoch wegen der Einschränkung der Aufgabenstellung auf nicht-parallele Transaktionen nicht weiter verfolgt wird.

Die einfachste Möglichkeit zur transaktionalen Unterstützung besteht in der Verwirklichung eines Workflows durch genau eine flache Transaktion. Bereits in der frühesten Literatur über Transaktionen ([Gray et al. 81]) ist zu lesen, warum eine solche Strukturierung für Workflows ungeeignet ist; der Vollständigkeit halber soll hier die in der Einleitung skizzierte Argumentation einmal ausgeführt werden. Bei flachen Transaktionen dient nur der Transaktionsanfang als Synchronisationpunkt, demzufolge wird die Transaktion nach beim Wiederanlauf nach einem Ausfall abgebrochen. Dadurch geht die gesamte innerhalb der Transaktion verrichtete Arbeit verloren, was für Workflows unakzeptabel ist. Ebenfalls stellt das Zurücksetzen kein nutzenbringendes Konstrukt mehr dar, weil ebenfalls nur auf den Transaktionsbeginn gesprungen werden kann. Bezieht man noch extern vorliegende und transaktional verwaltete Ressourcen mit ein, stößt man auch auf Schwierigkeiten, weil zur Wahrung der Isolationseigenschaften z.B. in Datenbanken Sperren bis zum Transaktionsende gehalten werden. Entsprechend der Transaktionsdauer müssen die Sperren dann sehr lange gehalten werden, was den Fortschritt parallel ablaufender Transaktionen stark behindern kann.

Diese Argumente lassen erkennen, daß es essentiell ist, im Verlauf eines Workflows eine beliebige Anzahl von Synchronisationspunkten setzen zu können. Einerseits läßt sich so der Arbeitsverlust bei einem Systemausfall gering halten, wenn man davon ausgeht, daß die Sicherungspunkte auch gleichzeitig als Aufsetzpunkte beim Wiederanlauf dienen. Andererseits ist eine Vielzahl von Sicherungspunkten für den effektiven Einsatz der *Rollback*-Anweisung zur Fehlerbehandlung notwendig. Die einfachste Möglichkeit, der Anforderung nach einer beliebigen Anzahl von Synchronisationspunkten in einem Workflow nachzukommen, besteht in der Partitionierung eines Workflows in flache Transaktionen. Ein Synchronisationspunkt kann

gesetzt werden, indem die aktuelle Transaktion abgeschlossen und nahtlos im Anschluß eine neue begonnen wird. Mangelhaft an diesem Modell von „verketteten“ Transaktionen ist die fehlende Flexibilität des Rücksetzmechanismus, der nur ein Rücksetzen auf den letzten etablierten Synchronisationspunkt erlaubt. Grund dafür ist die Garantie des flachen Transaktionsmodells, daß alle Zustandsänderungen einer abgeschlossenen Transaktion dauerhaft sind,[2] so daß ein Zurücksetzen über das letzte Transaktionsende untersagt werden muß. Warum der Programmierer an einer Rücksetzmöglichkeit auf jeden Synchronisationspunkt innerhalb des aktuellen Workflows interessiert sein wird, zeigt folgendes Beispiel der Buchung einer Reise, die aus mehreren Einzelaktivitäten besteht: Flugbuchung, Buchung des Hotels, Reservierung einer Opernkarte etc. Wurden Flug und Hotel bereits gebucht (als Einzelaktivitäten in bereits abgeschlossenen Transaktionen), und schlägt jetzt die nächste Einzelaktivität in der Kette, die Reservierung von Opernkarten, fehl, gibt es mehrere Möglichkeiten, darauf zu reagieren. Beispielsweise kann der letzte Reservierungsversuch einfach abgebrochen werden und anschließend wird fortgefahren, als ob nichts geschehen sei; dies kann adäquat durch verkettete flache Transaktionen unterstützt werden. Eine plausiblere Variante ist aber, daß die Reise verschoben wird (falls der Zweck der Reise der Opernbesuch war), wozu sowohl die Hotelbuchung als auch die Flugbuchung erstmal storniert und anschließend mit verändertem Datum wiederholt werden müssen. Es ist wünschenswert, dies automatisch vom System durch ein Zurücksetzen auf den entsprechenden, vom Programmierer gesetzten Synchronisationspunkt durchführen zu lassen.

Eine Lösung für diesen letzten Fall stellt die Partitionierung einer Transaktion durch Sicherungspunkte dar, wobei ein Workflow wieder genau einer Transaktion entspräche. Eine Transaktion kann eine beliebige Anzahl von Sicherungspunkte enthalten, die vom Programmierer gesetzt werden und zur Identifikation vom System eine individuelle Kennung erhalten. Anhand dieser Kennung kann das Programm ein Zurücksetzen auf einen Sicherungspunkt veranlassen. Mit dem Beenden einer Transaktion werden die bis dahin gesetzten Sicherungspunkte ungültig, so daß kein Zurücksetzen einer bereits abgeschlossenen Transaktion mehr möglich ist und auch die Dauerhaftigkeit gewahrt bleibt. Sicherungspunkte sind schon seit geraumer Zeit in Datenbanksystemen implementiert ([Astrahan et al. 79]), unter anderem weil die flexibere Möglichkeit zum Zurücksetzen auch die Implementierung von Datenbanksystemen vereinfacht: das Datenbanksystem setzt automatisch vor jedem neuen Befehl einen Sicherungspunkt, so daß bei einem Fehler nur der letzte Befehl zurückgenommen werden muß und nicht die gesamte Transaktion.

Sicherungspunkte unterscheiden sich von Synchronisationspunkten, weil mit Sicherungspunkten nicht der Systenzustand dauerhaft gemacht wird und sie demzufolge nicht beim Wiederanlauf angesteuert werden können. Dies hat zur Folge, daß bei einem Systemfehler die aktuelle Transaktion automatisch zurückgesetzt wird und alle bisher durch sie ausgeführten Änderungen verloren sind. Um diesen Mangel zu beheben, müssen die Sicherungspunkte persistent gemacht werden, so daß sie nunmehr vollwertige Synchronisationspunkte sind. Das System wird beim Wiederanlauf den chronologisch jüngsten Synchronisationspunkt ansteuern, wobei es sich jetzt auch um einen Sicherungspunkt handeln kann. Mit persistenten Sicherungspunkten bleibt also die Flexibilität von nicht-persistenten Sicherungspunkten erhalten und man hat zusätzlich die Ausfallsicherheit von verketteten Transaktionen hinzugewonnen.

Der Einsatz von persistenten Sichronisationspunkten sei noch einmal am vorigen Beispiel

[2] Diese Aussage gilt natürlich nur für den Fall, daß die Zustandsänderungen nicht explizit rückgängig gemacht werden.

der Reisebuchung anschaulich gemacht. Um jeden einzelnen Arbeitsabschnitt zu sichern und die Möglichkeit zu erhalten, dessen Änderungen zurückzusetzen, würde der Programmierer nach dem erfolgreichen Abschluß eines Schrittes einen Sicherungspunkt setzen. Um zum Beispiel zurückzukehren: es würden nach der Flug- und Hotelbuchung jeweils ein Sicherungspunkt gesetzt werden. Tritt nun ein Systemfehler während der Kartenreservierung auf, hat dies auf die Gültigkeit der Flug- und Hotelbuchung keinerlei Auswirkungen. Beim Wiederanlauf würde direkt nach dem letzten vorhandenen Sicherungspunkt, also nach der Hotelbuchung, fortgefahren werden und der Benutzer müßte die Kartenreservierung wiederholen. Schlägt dies fehl, weil die gewünschte Vorstellung schon ausgebucht ist, kann z.B. nur die Kartenreservierung abgebrochen werden oder aber auch die Hotelbuchung (oder die Hotel- und Flugbuchung) storniert werden.

Wie man an dem Beispiel sieht, erreicht man mit dieser einfachen Erweiterung von flachen Transaktionen bereits einen hohen Grad an Ausfallsicherheit und Flexibilität in bezug auf die Rücksetzmöglichkeiten. Was zusätzlich für dieses Transaktionsmodell spricht ist seine leichte Erlernbarkeit. Da es sich hierbei um eine Ergänzung von flachen Transaktionen handelt, die durch die starke Verbreitung kommerzieller Datenbanksysteme einen weiten Bekanntheitsgrad haben, kann der Programmierer sein vorhandenes Wissen weiterhin einsetzen und muß es nur um einige Details ergänzen. Selbst dies wird selten notwendig sein, da viele Datenbanksysteme Sicherungspunkte auf Anwenderebene zur Verfügung stellen, und diese sich in ihrer Handhabung nicht von persistenten Sicherungspunkten unterscheiden. Ein weiterer hervorzuhebender Punkt ist, daß persistente Sicherungspunkte in einer offenen persistenten Programmiersprache auch im Hinblick auf die Einbindung externer transaktionaler Ressourcen implementierbar sind.

Die Problematik bei der Einbindung externer transaktionaler Dienste ist, daß diese oft nur flache Transaktionen unterstützen; Sicherungspunkte mögen zwar vorhanden sein, aber diese sind nicht persistent und deshalb auch nicht weiter verwertbar. Unser Ziel ist es, eine integrierte persistente Entwicklungsumgebung zur Verfügung zu stellen, in der externe Dienste transparent und uneingeschränkt innerhalb des Transaktionsmodells eingesetzt werden können. Um also das Verhalten von persistenten Sicherungspunkten in transaktionalen externen Diensten zu simulieren, muß eine Anpassung vorgenommen werden. Die Anpassung sieht so aus, daß für jeden Sicherungspunkt die extern ausgeführte Transaktion abgeschlossen und eine neue begonnen wird, wodurch man extern wieder eine Transaktionskette für einen einzelnen Workflow erhält. Ein Beispiel für dieses Vorgehen ist in Abbildung 2.1 zu sehen. Wenn innerhalb des Programms auf einen Sicherungspunkt zurückgesprungen wird, müssen extern die abgeschlossenen Transaktionen durch geeignete Maßnahmen kompensiert werden.

Der vorangegangene Vergleich der transaktionalen Strukturierungsmöglichkeiten von Workflows hat gezeigt, daß flache Transaktionen mit persistenten Sicherungspunkten die funktionalen Anforderungen von Workflows—Möglichkeiten zum Sichern von Zwischenzuständen und flexible Reaktionsmöglichkeiten auf Fehler—bereits erfüllen. Der folgende Abschnitt zeigt, wie persistente Sicherungspunkte dem Programmierer in Tycoon zur Verfügung stehen und spricht Aspekte der Implementierung als Vorbereitung auf die nächsten Kapitel an.

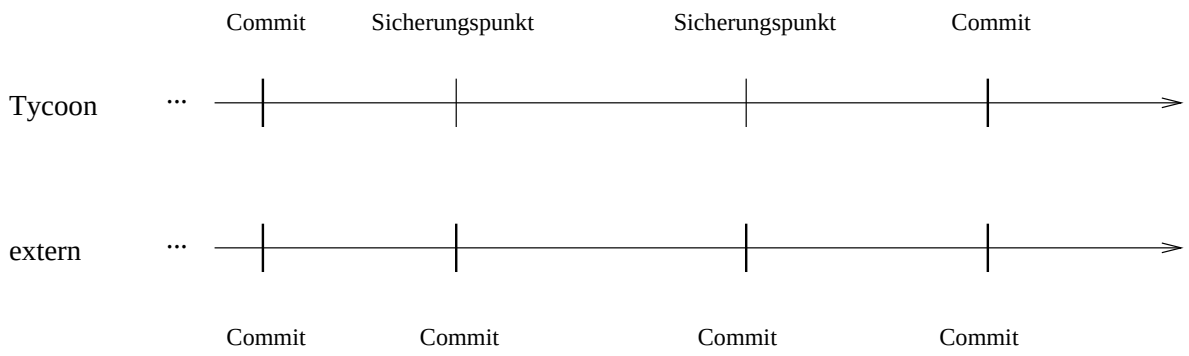

Abbildung 2.1: Beispiel für die externe Umsetzung von Sicherungspunkten

2.3 Ein Transaktionsmodell für Tycoon basierend auf persistenten Sicherungspunkten

Zentrale Philosophie von Tycoon ist die Erweiterbarkeit des Systems durch benutzergeschriebene Module. Die von diesen Modulen exportierten neuen Datentypen und zugehörige Operationen lassen sich über das polymorphe Typsystem nahtlos in die bestehende Umgebung integrieren, so daß die Grenze zwischen eingebauter und erweiterter Funktionalität verschwimmt. Da die Implementation eines Moduls auch Aufrufe an das Tycoon-Laufzeitsystem zuläßt, ist es naheliegend, die transaktionalen Befehle wie üblich über eine Modulschnittstelle dem Programmierer anzubieten. So etwas ähnliches ist bereits mit dem Modul *checkpoint*, einem Modul für flache Transaktionen, in Tycoon realisiert und wird als Ausgangsbasis für die Gestaltung eines neuen Moduls *transaction* für Transaktionen mit persistenten Sicherungspunkten dienen.

Das Modul *transaction* bildet die Schnittstelle zum erweiterten Transaktionsmechanismus in Tycoon; das Definitionsmodul ist in Abbildung 2.2 zu sehen. Genau wie vorher befindet sich auch unter dem erweiterten Transaktionskonzept ein Tycoon-Prozeß zu jedem Zeitpunkt in genau einer Transaktion, deren Verlauf durch die von *transaction* exportierten Funktionen gesteuert wird. Die Funktionen *commit()* und *rollback()* haben dieselbe Bedeutung wie die namensgleichen Funktionen des Moduls *checkpoint*: durch sie wird die aktuelle Transaktion abgeschlossen bzw. abgebrochen und unmittelbar eine neue Transaktion gestartet. Zum Setzen eines Sicherungspunktes dient die Funktion *savepoint()*, die dessen Systemkennung in Verbindung mit einem Statuswert zurückliefert. Der Statuswert teilt dem Programm mit, wie der Aufruf erreicht wurde: durch das ursprüngliche Setzen, durch ein Zurücksetzen oder durch einen Wiederanlauf. Derselbe Statuswert wird auch von der Funktion *commit()* zurückgegeben. Die Kennung eines Sicherungspunktes dient als Parameter der Funktion *rollbackTo()*, mit der das Zurücksetzen auf den entsprechenden Sicherungspunkt veranlaßt wird. Die Funktion *rollbackToLast()* führt ein Zurücksetzen auf den letzten Sicherungspunkt durch bzw. bricht die Transaktion ab, wenn noch kein Sicherungspunkt gesetzt wurde. Sie dient nur als Vereinfachung gegenüber *rollbackTo()*, weil sie keine Kennung des Sicherungspunktes als Parameter erfordert. Da sich ein Programm zu jedem Zeitpunkt in einer Transaktion befindet, ist keine Funktion notwendig, die explizit die Transaktionseigenschaften „einschaltet." Die Möglichkeit zum Ein- und Ausschalten transaktionalen Verhaltens mag auf den ersten Blick gewinnbringend aussehen, weil so der durch die transaktionalen Eigenschaften erforderliche

```
interface Transaction

export

  Via <:Ok

  Savepoint <: Ok

  error :Exception

  commitVia, rollbackVia, restartVia :Via

  savepoint() :Tuple s :Savepoint v :Via end

  commit() :Via

  rollbackTo(s :Savepoint) :Ok

  rollbackToLast() :Ok

  rollback() :Ok

end;
```

Abbildung 2.2: Die Schnittstelle des Moduls *transaction*.

Zusatzaufwand nach Bedarf umgangen werden kann. Diese Möglichkeit führt aber auch leicht zu Inkonsistenzen und schwer zu durchschauenden Programmierfehlern, so daß auf sie verzichtet wurde. Kapitel 3 wird auch zeigen, daß der mit Transaktionen verbundene Zusatzaufwand sich in erträglichen Grenzen hält.

Für den Programmierer stellen sich also persistente Sicherungspunkte als simple Erweiterung des bestehenden Transaktionsmechanismus in Tycoon dar, bei der nur zwei[3] Funktionen zusätzlich hinzukommen. Anders als bisher sollen aber Transaktionen nicht nur für Tycoon-eigene Daten verantwortlich sein, sondern insbesondere auch die extern angesteuerten Daten miteinbeziehen. Bei Workflows stellen im allgemeinen die externen Daten die Nutzdaten dar, die zudem aus einer Vielzahl externen Systeme stammen. Als Beispiel wurden in den vorangegangenen Abschnitten schon mehrfach Datenbanksysteme angeführt, denen oft die zentrale Rolle bei der Datenspeicherung zukommt. Aber auch die Elemente einer graphischen Benutzeroberfläche, beschriebene Dateien, angesprochene Drucker und aufgebaute Netzwerkverbindungen sind Beispiele für häufig eingesetzte externe Ressourcen. Diese Liste kann noch beliebig fortgesetzt werden; die Menge der für Workflows relevanten und deshalb in den Transaktionsmechanismus zu integrierenden externen Ressourcen ist also unbeschränkt. Erschwerend kommt hinzu, daß diese Ressourcen teilweise sehr unterschiedliche Charakteristiken aufwei-

[3] Würde man auch die Funktion *rollbackToLast()* einrechnen, wären es drei Funktionen.

sen. Beispielsweise sind Ressourcen wie die Elemente graphischer Benutzeroberflächen flüchtig und müssen demzufolge auch anders behandelt werden als Datenbanken. Da es unmöglich ist, für jeden einzelnen Datentyp eigene transaktionale Algorithmen zu implementieren—dieses ist nicht nur unwirtschaftlich, sondern stellt die Erweiterbarkeit des Systems in Frage—muß nach Unterscheidungsmerkmalen gesucht werden, anhand derer sich die Datentypen kategorisieren lassen. Das Ziel ist, daß die Datentypen einer Kategorie mit denselben Verfahren in den Transaktionsmechanismus integriert werden können. Folgende drei Merkmale lassen sich dabei als entscheidend erkennen:

Herkunft Bei den Daten kann es sich sowohl um Tycoon-interne Werte als auch um extern erzeugte handeln. Dieser Aspekt ist sehr wichtig für die transaktionalen Verfahren, weil über externe Daten weit weniger Information vorliegt als über interne, und somit auch die Möglichkeiten eingeschränkt sind. Da externe Dienste nur über ihre Schnittstellen bekannt sind, läßt sich der Zustand externer Ressourcen im Regelfall nur durch die Sequenz der darauf ausgeführten Operationen darstellen. Eine transaktionale Behandlung externer Daten wird also zwangsweise auf der Sequenz der ausgeführten Operationen aufbauen müssen. Demgegenüber können Tycoon-Werte u.U. optimiert behandelt werden, weil von diesen auch die interne Darstellung einsehbar ist.

Transaktionalität Die Verwaltung der Ressourcen kann schon von sich aus die transaktionalen Eigenschaften zusichern, wie es z.B. für die Tycoon-internen wie auch einige externe Daten der Fall ist. Bei externen Daten ist zu beachten, daß Persistenz nicht mit transaktionalem Verhalten gleichgesetzt werden kann, wie das Beispiel von Betriebssystemdateien belegt.[4] Werden mehrere transaktional verwaltete Ressourcen in einen Workflow einbezogen, muß das Setzen von Synchronisationspunkten zwischen den entsprechenden Ressourcenmanagern mit Techniken der verteilten Transaktionsverarbeitung synchronisiert werden.[5]

Persistenz Weiter oben wurden die Elemente graphischer Oberflächen und Netzwerkverbindungen bereits als Beispiel für nicht-persistente Ressourcen aufgeführt; Beispiele für persistente „Daten“ sind Dateien oder auch Drucker.[6] Da Persistenz ein Bestandteil der transaktionalen Eigenschaften ist, muß bei flüchtigen Daten die Persistenz simuliert werden.

Die Nutzungsweise der Daten, also ob diese applikations-privat sind oder geteilt werden, wird als Unterscheidungsmerkmal ignoriert. Wie bereits in Abschnitt 2.1 bei der Diskussion transaktionaler Eigenschaft festgestellt wurde, spielt die Isolationseigenschaft im Kontext persistenter Programmiersprachen keine Rolle.

Die Klassifizierung von transaktional zu behandelnden Daten ist in Abbildung 2.3 als Entscheidungsbaum dargestellt, der in nur vier Klassen entsprechend bestimmter Merkmalsausprägungen mündet. Bei orthogonaler Anwendung obiger Merkmale entstünden acht Klassen; einige dieser Klassen beruhen aber auf widersinnigen Merkmalskombinationen. Für Tycoon-interne Daten ist es beispielsweise unsinnig anhand der Art der Verwaltung und Persistenz weitere Unterscheidungen zu machen, weil in Tycoon orthogonale Persistenz mit flachen

[4] Eine Ausnahme stellt Tandems *Guardian*-Betriebssystem dar, das mit einem transaktionalen Dateisystem ausgestattet ist.

[5] Die Begründung dieser Aussage wird später in Kapitel 5 vorgenommen.

[6] Ein Ausdruck hat einen persistenten Seiteneffekt, nämlich das bedruckte Blatt.

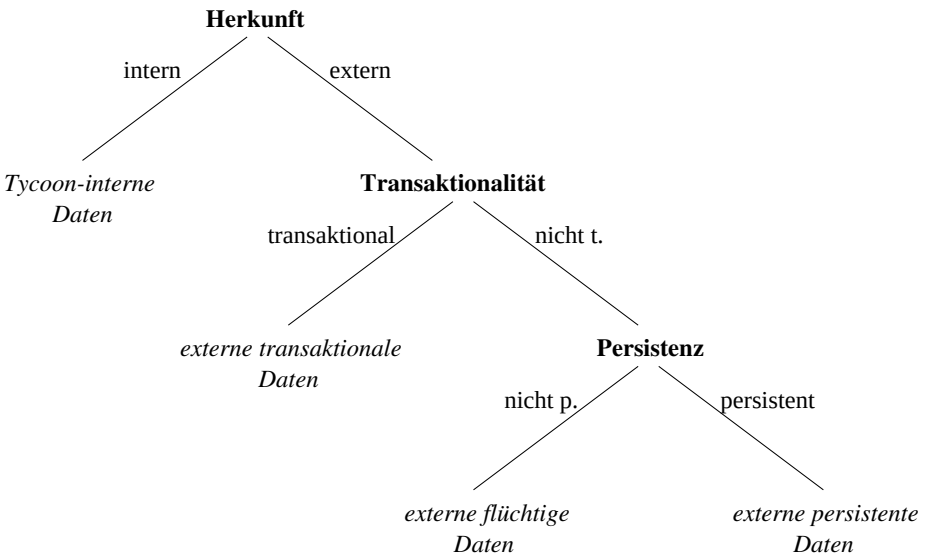

Abbildung 2.3: Die Ressourcenklassifizierung

Transaktionen vorgegeben ist. Ebenso ist die Persistenz eine zentrale Eigenschaft transaktionaler Daten. Damit kristallisieren sich vier Klassen von Daten heraus, die zur Implementierung transaktionalen Verhaltens jeweils spezialisierte Verfahren benötigen. Als Übersicht für die folgenden drei Kapitel sei hier zu jeder Klasse das Realisationsverfahren angesprochen.

Tycoon-interne Daten: In der Architektur von Tycoon ist bereits durch eine Store-Komponente eine orthogonale, also typunabhängige und allumfassende, Persistenz integriert. Ebenso wurde bei der Store-Architektur die Unterstützung flacher Transaktionen berücksichtigt. Um persistente Sicherungspunkte in Tycoon zur Verfügung zu stellen, muß also die bestehende Architektur in möglichst „schonender“ Weise erweitert werden, d.h. ohne die Integration des Stores zu beeinträchtigen. Das unmittelbar anschließende Kapitel beschreibt eine auf dem Store aufgesetzte Zwischenschicht, die eine funktionale Erweiterung des Stores um persistente Sicherungspunkte realisiert.

Flüchtige externe Daten: Bei dieser Klasse von Daten fehlt selbst die elementare Persistenzeigenschaft, so daß diese vom System unter Rückgriff auf die Sequenz der ausgeführten Operationen simuliert werden muß. Zu diesem Zweck werden die Operationen und die dazugehörigen kompensierenden Operationen chronologisch in einer Log-Struktur verzeichnet. Wie allgemein bei externen Daten wird auch hier der Zustand durch Ausführung kompensierender Operationen zurückgesetzt. Die Wiederherstellung des aktuellen Zustands wird durch Wiederholung der aufgezeichneten Operationen erreicht. Kapitel 4 beschreibt die Realisierung des Operations-Loggings für flüchtige Daten.

Transaktionale externe Daten: Da bei transaktional verwalteten externen Ressourcen die benötigten Persistenz- und Atomaritätseigenschaften schon vorhanden sind, geht es nun darum, diese mit Tycoon in sicherer Weise zu verbinden. Das Ziel ist, die erweiterte Transaktionssemantik von Tycoon auf die externen Dienste abzubilden und dabei insbesondere den

Zustand von Tycoon und den externen Ressourcen synchron zu halten. Die Integration erfolgt durch Abbildung von Sicherungspunkten auf extern abgeschlossene Transaktionen; da hierbei in mindestens zwei verschiedenen Ressourcen[7] Zustände gesichert werden sollen, muß ein zweiphasiges Protokoll (*2-phased Commit*, 2PC) zur Koordination eingesetzt werden. Die zum Zurücksetzen erforderlichen kompensierenden Operationen werden in einer Log-Struktur verzeichnet. Kapitel 5 geht auf die Architektur einer verteilten Transaktionsumgebung mit Tycoon ein und schildert den Protokollablauf.

Nicht-transaktionale persistente externe Daten: Da bei dieser Art von Daten die Dauerhaftigkeit schon vorhanden ist, muß für die Integration in Tycoon nur die Rücksetzbarkeit implementiert werden. Genau wie bei transaktionalen Daten werden auch hier zum Zurücksetzen kompensierende Operationen ausgeführt, die in einer Log-Struktur verzeichnet werden müssen. Die Implementationsdetails der Log-Struktur und des Verfahrens zum Zurücksetzen unterscheiden sich nicht von denen für transaktionale Daten, weshalb den nicht-transaktionalen persistenten Daten kein eigenes Kapitel gewidmet ist. Es sei aber noch angemerkt, daß im Unterschied zu transaktionalen Daten sich nicht garantieren läßt, daß der externe Zustand synchron mit Tycoon bei den Sicherungspunkten festgeschrieben wird. Aus diesem Grund ist die Einbeziehung von nicht-transaktionalen externen Daten mit dem Risiko möglicherweise inkonsistenter Zustände verbunden, die durch manuelle Eingriffe des Benutzers behoben werden müssen.

[7] Beim Zustand von Tycoon handelt es sich auch um eine Ressource im Sinne des Transaktionsmodells.

3. Sicherungspunkte für Tycoon-interne Zustände

Die erste Teilaufgabe der Implementierung persistenter Sicherungspunkte in Tycoon besteht in der Erweiterung des Tycoon-eigenen flachen Transaktionsmodells. Dabei ist zu beachten, daß in Tycoon der für die Persistenz zustände Objektspeicher über ein abstraktes Speicherprotokoll (*Tycoon Store Protocol*, im folgenden als *TSP* [Matthes et al. 92] bezeichnet). angebunden wird, welches nur flache Transaktionen unterstützt. Eine Randbedingung bei der Erweiterung ist deshalb die Beibehaltung der existierenden Speicherarchitektur und damit des Speicherprotokolls Um dieser Anforderung gerecht zu werden, sind die Sicherungspunkte durch eine Schicht oberhalb des TSP implementiert; diese Schicht bietet ein um persistente Sicherungspunkte erweitertes, ansonsten aber mit dem TSP identisches, Speicherprotokoll an. Der Realisierungsansatz beruht auf dem chronologischen Verzeichnen (*Logging*) von geänderten Objekten und der Abbildung von Sicherungspunkten auf abgeschlossenen Transaktionen auf der Ebene des Objektspeichers. Zum Zurücksetzen werden dann, analog zu den transaktionalen externen Daten, kompensierende Operationen auf dem Objektspeicher ausgeführt.

Dieses Kapitel beginnt mit einer Beschreibung des abstrakten Speicherprotokolls und seiner architekturellen Ergänzung durch die Logging-Zwischenschicht. Nach der Darstellung der Log-Struktur und der Log-Operationen wird auf die Algorithmen für die transaktionalen Operationen eingegangen. Zum Abschluß wird anhand von Vorüberlegungen und empirischer Messungen gezeigt, daß die Implementierung von Sicherungspunkten nur einen marginalen Performanzverlust bewirkt.

3.1 Anforderungen durch das abstrakte Speicherprotokoll

Die Speicherarchitektur von Tycoon unterscheidet sich grundlegend von der konventioneller Programmiersprachen, da sie keinen eigenen Objektspeicher vorsieht, sondern eine flexible Anbindung beliebiger Objektspeicher durch das abstrakte Speicherprotokoll TSP [Matthes et al. 92] erlaubt. Das TSP ist in seiner Funktionalität an einem generischen Objektspeicher angelehnt und dient dem Tycoon-Laufzeitsystem als alleinige Schnittstelle zum persistenten Speicher. Das Tycoon-Laufzeitsystem verwaltet alle anfallenden Daten über das TSP, wobei auch adminstrative Datenstrukturen von Daten, Programmen und Threads eingeschlossen sind. Zur Anpassung eines bestimmten Objektspeichers an Tycoon muß ein Adaptermodul geschrieben werden, welches die TSP-Funktionen mit dem Objektspeicher implementiert. Der Vorteil eines solchen Konzeptes ist die leichte Portierbarkeit der Schnittstelle auf beliebige Systeme und die Einsetzbarkeit selbst der neuesten kommerziellen Objektspeichertechnologie

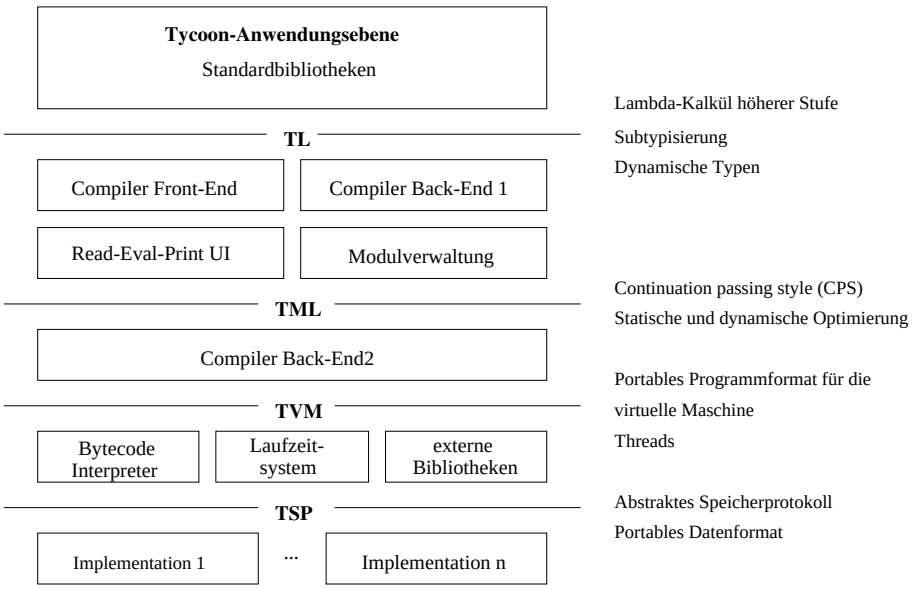

Abbildung 3.1: Die Tycoon-Architektur

(beispielsweise existiert eine Anbindung von ObjectStore [Lamb et al. 92]) ohne eigenen Implementierungsaufwand. Die Verwendung eines abstrakten Speicherprotokolls entspricht deshalb der Tycoon-Philosophie, existierende Technologien in das System einzubinden anstatt sie zu reimplementieren. Abbildung 3.1 (aus [Matthes 93]) gibt einen Überblick der Architektur. Die Abkürzungen TL, TML und TVM bezeichnen weitere Schichten in der Tycoon-Architektur und sind in [Matthes 93] erläutert.

Das TSP definiert ein bestimmtes Objektmodell sowie eine Reihe von Objektverwaltungsfunktionen und sieht einen *Garbage Collection*-Mechanismus vor, im folgenden auch als Garbage Collection bezeichnet. Ein Objekt stellt sich dem TSP-Anwender als ein Array von beliebigen Werten dar, denen sonst keine Struktur auferlegt wird und das anhand einer eindeutigen Objektreferenz (*OID*) identifiziert werden kann. Zur Unterstützung der Garbage Collection müssen allerdings Objektreferenzen gesondert markiert sein. Neben den eigentlichen Nutzdaten hat jedes Objekt auch einen Header, in dem verschiedene Statusinformationen über das Objekt gespeichert werden (beispielsweise ob es veränderbar ist).

Objekte werden mit den in mehreren Varianten zur Verfügung stehenden Erzeugungsfunktionen angelegt und sind immer persistent. Durch die Garbage Collection entfällt die Notwendigkeit für den TSP-Benutzer, Objekte explizit zu löschen. Die Garbage Collection wird periodisch aufgerufen und entfernt alle nicht mehr referenzierten Objekt selbsttätig. Die Lebendigkeit eines Objektes wird anhand transitiver Erreichbarkeit von eines designierten Wurzelobjekt festgestellt. Um einen Zugriff auf das Objekt zu ermöglichen, sind in TSP Funktionen zum Lesen und Schreiben des Objektinhaltes und einiger Objektattribute vorgesehen. Über die reine Objektpersistenz hinaus muß ein TSP-konformer Objektspeicher auch flache Transaktionen unterstützen. Diese werden in TSP durch die Funktionen *tsp_commit()* und *tsp_rollback()* zur Verfügung gestellt. Darüberhinaus gibt es noch administrative Funktionen, die jedoch für die vorgenommene Erweiterung keine Bedeutung haben.

Eine Grundvoraussetzung bei der Implementierung persistenter Sicherungspunkte war die Beibehaltung dieser Speicherarchitektur, deren zentrales Element TSP ist. Deswegen war

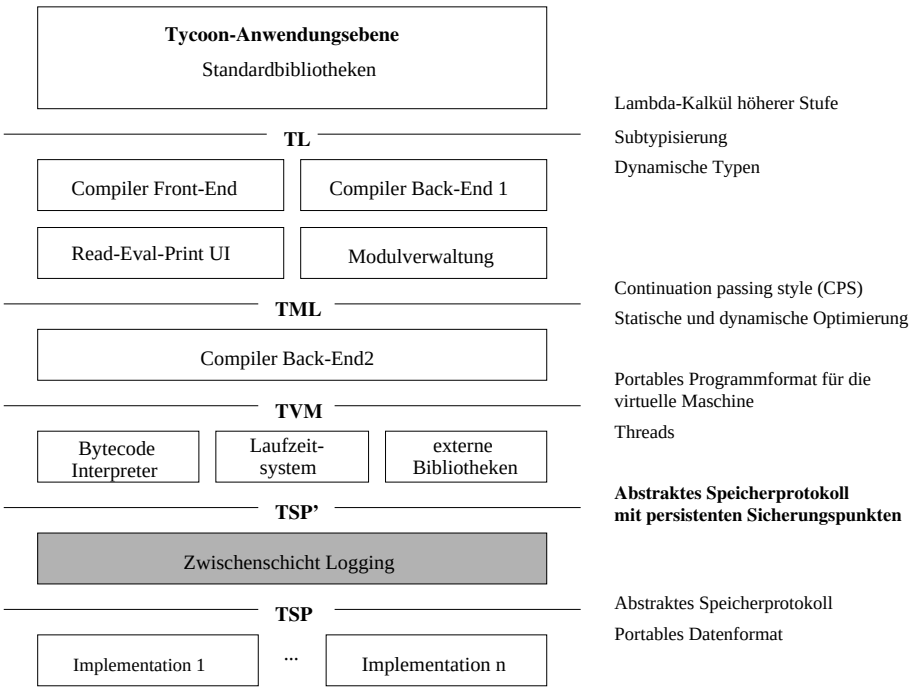

Abbildung 3.2: Die erweiterte Tycoon-Architektur

es nicht möglich, die Erweiterung des flachen Transaktionsmodells direkt in TSP zu verankern. Statt einer Implementierung von Sicherungspunkten in der Store-Anpassung werden diese deshalb von einer Zwischenschicht oberhalb des TSP realisiert. Diese Zwischenschicht implementiert ein um die fehlenden transaktionalen Operationen wie *tsp_savepoint()* und *tsp_rollbackTo()* erweitertes TSP. Die erweiterte Architektur ist in Abbildung 3.2 dargestellt. Die Zwischenschicht selbst greift selber ausschließlich auf das zugrundeliegende TSP zu und ist damit ebenso portabel und flexibel wie das Tycoon-Laufzeitsystem; insbesondere stehen damit für jeden TSP-konformen Objektspeicher automatisch persistente Sicherungspunkte zur Verfügung.

Um nun persistente Sicherungspunkte auf einem Objektspeicher zu simulieren, der nur flache Transaktionen unterstützt, wird, genauso wie für transaktionale externe Daten auch, bei Setzen eines Sicherungspunktes die laufende Transaktion im Speicher durch eine Commit-Operation abgeschlossen. Dadurch ergibt sich auch für Tycoon-interne Daten eine zweistufige Transaktions„schichtung“. Im folgenden soll deshalb zwischen Tycoon-Transaktionen, welche durch die erweiterte Zwischenschicht angeboten werden und Sicherungspunkte enthalten können, und Store-Transaktionen, welche zur Implementierung der Tycoon-Transaktionen dienen und nicht mehr direkt sichtbar sind, unterschieden werden. Das Zurücksetzen einer Tycoon-Transaktion auf einen beliebigen Sicherungspunkt oder den Transaktionsanfang wird durch Ausführung kompensierender Store-Operationen durch die Zwischenschicht erreicht. Die zur Kompensation erforderlichen Operationen werden anhand der chronologisch in einer Log-Datenstruktur verzeichneten TSP-Operationen ermittelt. Die Eintragungen in der Log-Struktur werden als „Nebeneffekte“ der TSP-Operationen von der Logging-Zwischenschicht vorgenommen und sind somit transparent für das Laufzeitsystem und alle höheren Ebenen des

Tycoon-Systems. Dadurch wird erreicht, daß alle persistenten Daten des Tycoon-Systems[1] in der Log-Struktur berücksichtigt werden, ohne daß dafür die Architektur des Tycoon-Systems in grundlegender Weise geändert werden müßte.

Um den Store-Zustand auf einen vorigen Sicherungspunkt zurückzusetzen, müssen im Log genügend Informationen zur Kompensation aller seit dem Sicherungspunkt erfolgten TSP-Operationen vorhanden sein. [Härder, Reuter 83] sprechen in diesem Zusammenhang auch von einem Undo-Log. Demhingegen ist es nicht erforderlich, beim Wiederanlauf den Store-Zustand durch Wiederholung von TSP-Operationen auf den Stand des letzten gesetzten Sicherungspunktes zu bringen, da der Tycoon-Sicherungspunkt auf ein Store-Commit abgebildet und so der Zustand automatisch festgeschrieben wurde. Der Wiederanlauf wird also im Store wahrgenommen, weswegen in der Logging-Zwischenschicht keine Redo-Information im Log verzeichnet werden muß; die Zwischenschicht hält also einen reinen Undo-Log.

3.2 Implementationstechniken für Vorwärtsverarbeitung, Zurücksetzen und Wiederanlauf

In diesem Abschnitt werden die Algorithmen beschrieben, die als Grundlage für die Implementierung der transaktionalen Operationen (*commit()*, *savepoint()*, *rollbackTo()*) dienen. Als Vorbereitung wird die Log-Struktur und die Logging-Aktivität während des normalen Programmablaufs (der *Vorwärtsverarbeitung*) dargestellt, im Anschluß daran wird gezeigt, daß die Existenz eines Logs die Wirksamkeit der Garbage Collection nicht beeinträchtigt.

3.2.1 Logstrukturen für persistente Programmiersprachen mit abstraktem Speicherprotokoll

Wie bereits erwähnt, werden im Log die kompensierenden Operationen verzeichnet, die zum Zurücksetzen der Tycoon-Transaktion notwendig sind. Ein naheliegender Ansatz, der auch häufig in relationalen Datenbanksystemen (siehe zum Beispiel [Gray et al. 81; Härder, Reuter 83; Mohan et al. 92; Gray, Reuter 93]) gewählt wird, könnte wie folgt aussehen: Vor jeder Änderung wird der noch gültige Wert im Log verzeichnet, beim Zurücksetzen wird dann das Log umgekehrt chronologisch abgearbeitet und alle gesicherten alten Werte werden wiederhergestellt. Auf diese Weise wird sichergestellt, daß jede Zustandsänderung garantiert zurückgenommen wird. Dieses Verfahren ist für relationale Datenbanksysteme durchaus angemessen, sollte jedoch für persistente Programmiersprachen vermieden werden, weil in diesem Kontext einzelne Objekte oft mehrfach im Laufe einer Transaktions geändert werden (vergleiche dazu [Kent et al. 85]). Beispielsweise kann es vorkommen, daß eine Schleifenvariable mehrere hundert Änderungen in einem sehr kurzen Zeitraum erfährt. Das sofortige Logging von Undo-Information mit dem Auftreten einer Änderung ist aus mehreren Gründen zu vermeiden:

- Es entstehen viele überflüssige Einträge, weil in einer Sequenz von Undo-Einträgen für ein bestimmtes Objekt nur die jeweils ersten nach Synchronisationspunkten erforderlich sind.

[1] Zu den persistenten Daten zählen nicht nur die in der Anwendungsebene erzeugten, sondern auch viele Datenstrukturen des Laufzeitsystems, wie zum Beispiel der Programm-Stack oder die Liste der Threads.

- Das Anlegen eines Log-Eintrages stellt einen hohen konstanten Zeitaufwand für eine Änderungsoperation dar.
- Beim Zurücksetzen werden entweder eine Menge überflüssiger Operationen ausgeführt oder es wird eine Log-Analysephase erforderlich, die die relevanten Logeinträge ermittelt. In beiden Fällen ist jedoch mit zusätzlichem Aufwand zu rechnen.

Das Ziel muß es also sein, nur bei der ersten Änderung eines Objekt zwischen je zwei Synchronisationspunkten die notwendige Undo-Information zu verzeichnen und dies bei Folgeänderungen zu vermeiden. [White, DeWitt 95] beschreiben eine Logging-Variante für einen Objektspeicher, die auf Seitenbasis kumulierte Änderungen (also Redo/Undo-Information) loggt und damit genau diesen Effekt erzielt. Dieser Ansatz muß für die Anwendung in der Logging-Schicht abgeändert werden, weil hier keine Speicherseiten sichtbar sind. Da oberhalb des TSP nur Objekte sichtbar sind, wird die Zwischenschicht Undo-Information auf der Basis von ganzen Objekten kumulieren. Das implementierte Verfahren funktioniert wie folgt:

- Vor der ersten Änderung eines Objektes nach einem Synchronisationspunkt wird eine vollständige Kopie des Objektes im Log verzeichnet und das Objekt als kopiert markiert.
- Folgeoperationen, ganz gleich welchen Teil des Objektes sie betreffen, erkennen, daß das Objekt als geloggt markiert ist und müssen selbst kein Logging mehr durchführen. Der mit jeder Änderungsoperation verbundene konstante Zusatzaufwand ist jetzt also auf das Prüfen der Markierung reduziert worden und damit unerheblich (wie auch die später vorgestellten Messungen beweisen).
- Beim Zurücksetzen wird der vollständige Objektzustand, welcher durch die Kopie verfügbar ist, wiederhergestellt.

Der Nachteil dieses Verfahrens ist, daß selbst bei nur partieller Änderung eines Objektes immer eine vollständige Kopie angelegt wird, so daß auch Objektdaten im Log verzeichnet sind, die nicht geändert wurden. Als Folge wird das Log-Volumen erhöht und das Zurücksetzen aufwendiger. Anstelle von Objektkopien könnten deswegen auch *Objektdeltas* im Log verzeichnet werden. Diese enthalten ausschließlich den Originalzustand der *geänderten* Objektdaten und müßten ermittelt werden, indem vor der ersten Objektänderung eine vollständige Kopie angelegt wird, die dann vor dem nächsten Synchronisationspunkt mit dem aktuellen Wert des Objektes verglichen wird. Das Verzeichnen von Objektdeltas setzt aber laufzeitintensive Analysealgorithmen voraus, weshalb davon abgesehen wurde und stattdessen vollständige Objektkopien im Log verzeichnet werden. Ein Nachteil ist, daß auch große Objekte wie zum Beispiel Audiodaten vollständig kopiert werden, selbst wenn darin nur ein kleiner Bereich geändert wurde. Allerdings werden Objekte dieser Art im allgemeinen als Ganzes gelesen und geschrieben, so daß dieses Problem nicht auftritt.

Bei der Implementierung der Log-Struktur stellt sich noch die Frage, ob diese intern oder extern gespeichert werden soll. Bei einer internen Speicherung ist das Log eine Datenstruktur im Objektspeicher, z.B. eine verkettete Liste, die auch durch TSP-Operationen erzeugt und geändert wird. Die Objektkopien, die als Einträge im Log verkettet sind, werden bei dieser Alternative ebenfalls im Store als reguläre Objekte abgespeichert. Bei einer externen Speicherung wird das Log in einer eigenen Daten außerhalb des Objektspeichers aufbewahrt. Die geänderten Objekte müssen dann in diese Datei kopiert werden, was tech-

nisch aber auch keine Schwierigkeiten bereitet, weil hierbei keine „tiefe“ Kopie aller transitiv erreichbaren Objekte erforderlich ist. Bei der externen Speicherung besteht das Potential zur höheren Performanz als bei einer Speicherung im Objektspeicher, weil extern das Speicherformat, z.B. als rein sequentiell beschriebene Datei auf einer dedizierten Platte, optimiert werden kann. Hierbei ergibt sich aber die Schwierigkeit, daß die Durchführung eines Store-Commit mit dem externen Log über ein zweiphasiges Commitprotokoll ([Gray 78; Bernstein et al. 87]) synchronisiert werden müßte, da sonst die Konsistenz des externen Logs nicht garantiert werden kann. Hinzu kommt, daß die in den Objektkopien enthaltenen Referenzen auf andere Objekte bei jeder Garbage Collection aktualisiert werden müssen, da sich Objektreferenzen dadurch ändern können. Bei Verzeichnen der Objektkopie im Objektspeicher wird die Aktualisierung der OIDs selbsttätig bei der Garbage Collection vorgenommen. Die erhöhte Komplexität, die eine externe Speicherung der Logstruktur mit sich bringt, stellt den erzielbaren Performanzgewinn in Frage und führte zu der Entscheidung, die Logstruktur intern zu speichern.

Die Logstruktur wird in der Zwischenschicht als Liste von Logeinträgen, bei denen es sich im wesentlichem um Objektkopien handelt, mit Hilfe des darunterliegenden Objektspeichers realisiert. Die Objektkopien sind folglich für den Objektspeicher persistente Objekte, die von denen der Tycoon-Laufzeitumgebung nicht zu unterscheiden sind. Wird bei einem Sicherungspunkt ein Store-Commit durchgeführt, dient dies auch gleichzeitig zur Sicherung der Logstruktur. Wird ein Objekt wiederholt über den Verlauf mehrerer Sicherungspunkte geändert, muß für jeden der Sicherungspunkte genau eine Kopie im Log vorhanden sein; somit läuft dieses Loggingverfahren auf eine auf die Sicherungspunkte ausgerichtete Objektversionierung heraus.

Nach dieser Übersicht sollen an dieser Stelle die Vorteile zusammengefaßt werden, die sich aus der Abbildung eines Tycoon-Sicherungspunktes auf ein Store-Commit ergeben:

- Durch die Sicherung des Store-Zustandes durch ein Commit ist es nicht erforderlich, im Log auch Redo-Information zu verzeichnen, was das Logvolumen reduziert und auch den Laufzeitaufwand verringert.

- Aus diesem Grund wird auch der Wiederanlauf erheblich vereinfacht, weil der Objektspeicher automatisch auf den Stand des letzten Sicherungspunktes gebracht wird.

- Da das Log nur für das Zurücksetzen benötigt wird, kann der Log-Inhalt bei einem Tycoon-Commit gelöscht werden. In relationalen Datenbanksystemen ist dafür ein aufwendiger Checkpointmechanismus in Verbindung mit einer Logarchivierung erforderlich (siehe auch wieder [Gray et al. 81; Mohan et al. 92; Bernstein et al. 87]).

3.2.2 Loggingaktivität während der Vorwärtsverarbeitung

Die zum Zurücksetzen benötigte Undo-Information muß während der Vorwärtsverarbeitung in Form von Logeinträgen gesammelt werden. Bei diesen handelt es sich weitgehend um Objektkopien, teilweise kommt man aber auch mit weniger Information aus. Im folgenden wird für jede einzelne Operation bzw. Operationsgruppe mit Seiteneffekten beschrieben, welche Logging-Operationen für diese in der Zwischenschicht ausgeführt werden. Abbildung 3.3 zeigt anhand eines Beispiels die verschiedenen Log-Einträge.

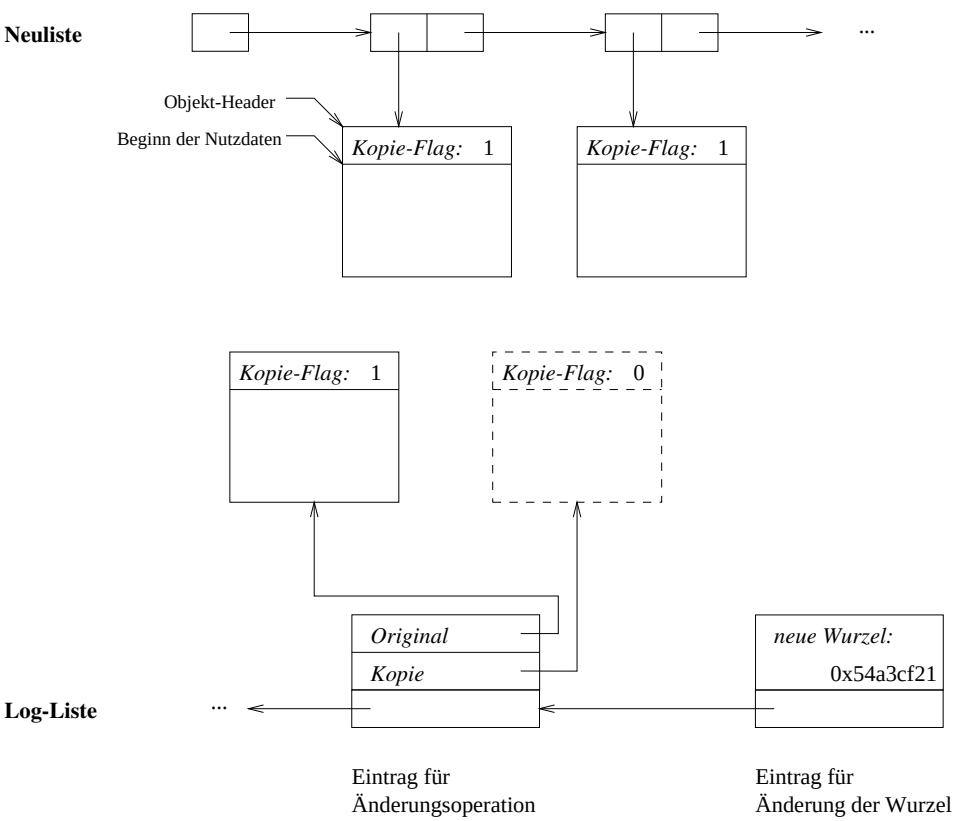

Abbildung 3.3: Ausschnitt aus der Log-Struktur mit Neuliste

Änderungsoperationen Dazu gehören alle Funktionen zum Setzen eines Slots oder eines Header-Datums von einem Objekt sowie die Funktion *tsp_openReadWrite()*, die an sich noch keine Änderung durchführt, aber vorbereitet. Vor Ausführung der Änderung wird eine vollständige Objektkopie mitsamt dem Header angelegt und ein Log-Eintrag mit zwei Teilfeldern konstruiert. Das erste Teilfeld enthält eine Referenz auf die Objektkopie, das zweite eine Referenz auf das Originalobjekt. Im Anschluß daran wird das Originalobjekt durch ein Feld in seinem Header als kopiert markiert, so daß keine weiteren Kopien von darauffolgenden Änderungsoperationen angelegt werden.

Änderung der Wurzel des Objektspeichers Da es sich bei der Wurzel des Objektspeichers nicht um ein Objekt mit einem Header handelt, muß die Änderung dieses „Objektes" gesondert behandelt werden. Zu diesem Zweck wird ein eigener Log-Eintrag für diese Änderungsoperation festgelegt, der den alten Wurzelwert enthält. Da die Wurzel nicht als kopiert markiert werden kann, wird diese Tatsache in einer booleschen Variablen in der Zwischenschicht gespeichert, die bei darauffolgenden Wurzeländerungen abgeprüft wird.

Objekterzeugung Um die Objekterzeugung zu kompensieren, müßte das Objekt wieder gelöscht werden. Weil aber das Löschen von Objekten automatisch durch die Garbage Collection durchgeführt wird, ist für die Objekterzeugung keinerlei Undo-Information nötig. Es muß nur gewährleistet sein, daß beim Zurücksetzen alle Referenzen auf das neuangelegte Objekt

entfernt werden, was durch die Kompensation der Objektänderungen der referenzierenden Objekte erreicht wird. Um neuangelegte Objekte davor zu bewahren, überflüssigerweise bei einer darauffolgenden Änderungsoperation kopiert zu werden, werden die neuen Objekte bereits initial als „kopiert“ markiert. Zusätzlich werden sie für die im nächsten Abschnitt beschriebenen transaktionalen Algorithmen in eine eigene Liste, die Neuliste, eingetragen. Diese Liste muß ebenso wie die Log-Liste im Store gespeichert sein, damit die enthaltenen Objektreferenzen nicht durch eine Garbage Collection ungültig werden.

Änderung der Objektgröße Das TSP unterstützt auch die Änderung der Slot-Anzahl eines Objektes, wobei es dem Adaptionsmodul des jeweiligen Objektspeichers überlassen ist, wie dies implementiert ist. Bei einem *tsp_resize()* kann deshalb das Objekt direkt verändert werden, es ist aber auch möglich, daß eine Kopie des Objektes mit geänderter Slot-Anzahl angelegt wird. Selbst innerhalb eines Stores muß dies nicht einheitlich geregelt sein; so ist es zulässig, daß beim Vergrößern eines Objektes ein neues angelegt wird und beim Verkleinern das Objekt geändert wird. Aus diesem Grund ist es nicht möglich, eine Objektverkleinerung durch eine Vergrößerung auf die alte Größe zu kompensieren (auch nicht bei Nachlieferung der fehlenden Slotwerte): wird die Verkleinerung direkt am Objekt vorgenommen, bei der kompensierenden Vergrößerung aber ein neues Objekt angelegt, müßten auch die referenzierenden Objekte aktualisiert werden, was unmöglich ist. In der Logging-Zwischenschicht wird deshalb die „Größenänderung“ selbst vorgenommen, indem eine Objektkopie mit der geforderten Slotanzahl nach dem vorigen Schema (Markierung als kopiert und Eintrag in Neuliste) angelegt wird. Das Originalobjekt mit der alten Slot-Anzahl muß nicht explizit im Log verzeichnet werden, weil dieses durch die bestehenden Referenzen noch erreichbar ist. Falls ein in der Größe zu änderndes Objekt bereits als kopiert markiert ist, kann auf das explizite Anlegen einer Kopie verzichtet und die Store-Funktion *tsp_resize()* aufgerufen werden, weil diese Größenänderung nicht kompensiert werden muß.

3.2.3 Beschreibung der Transaktionsalgorithmen

Die transaktionalen Operationen *tsp_commit()*, *tsp_savepoint()*, *tsp_rollback()* und *tsp_rollbackTo()* werden unter Zuhilfenahme des Logs und der Store-Transaktionen implementiert.

Beim Setzen eines Synchronisationspunktes ist es notwendig, die Markierungen der kopierten Objekte wieder zurückzunehmen, damit bei der nächsten darauf folgenden Änderungsoperation erneut eine Kopie angelegt wird. Wie bereits erwähnt, handelt es sich bei den markierten Objekten einmal um geänderte Objekte, für die Logeinträge erzeugt wurde, und um neu angelegte Objekte. Bei den ersteren kann die Kopie-Markierung einfach bei einem Log-Durchlauf zurückgesetzt werden, die neuen Objekte sind über eine gesonderte Liste erreichbar. Für das Setzen eines Sicherungspunktes werden folgende Schritte ausgeführt:

1. Durchlauf des Logs bis zum letzten Sicherungspunkt oder alternativ bis zum Anfang des Logs, wobei die Markierung der kopierten Originalobjekte zurückgesetzt wird;

2. Durchlauf der Neuliste mit Zurücksetzen der Markierung der neu angelegten Objekte; danach kann der Inhalt der Neuliste „gelöscht“ werden (Referenz wird entfernt);

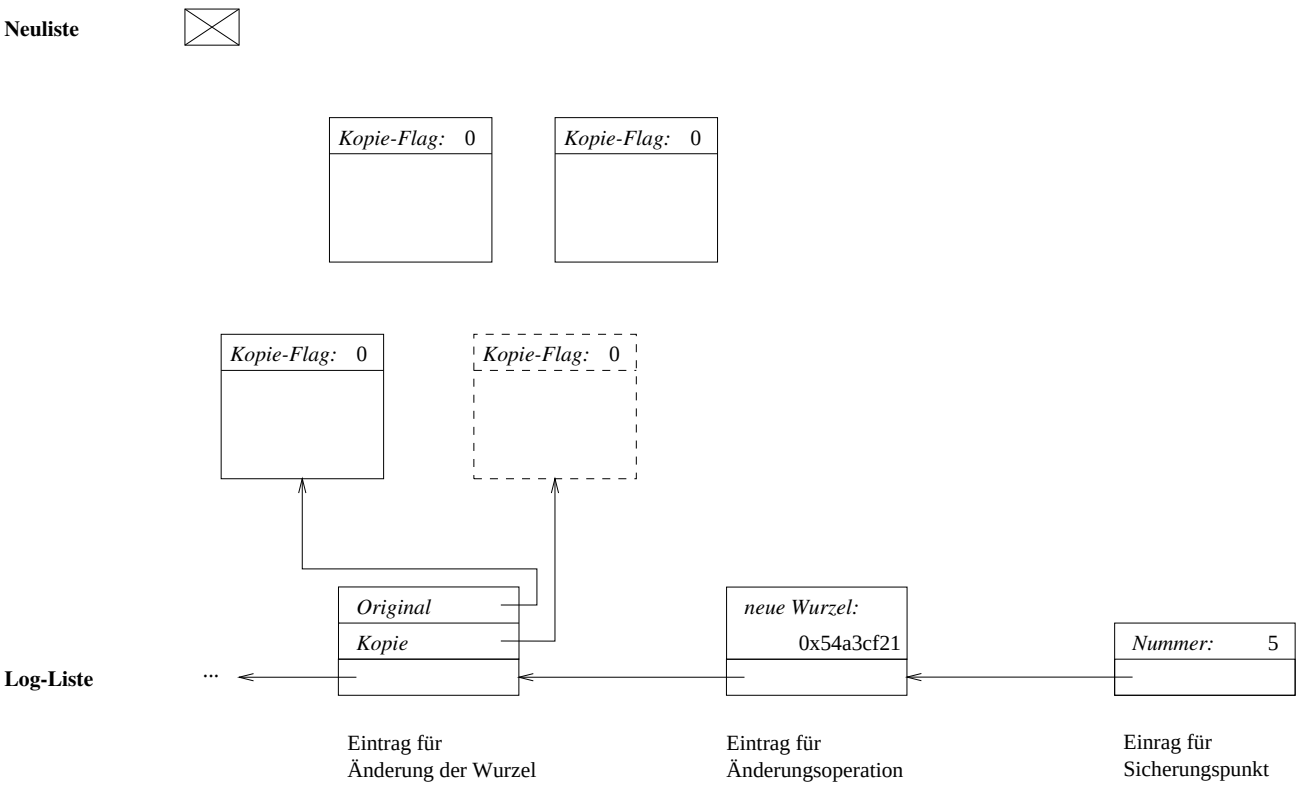

Abbildung 3.4: Logstruktur mit Neuliste nach Sicherungspunkt

3. Schreiben eines Log-Eintrages zur Markierung des Sicherungspunktes; dieser Log-Eintrag enthält die Nummer des Sicherungspunktes und dient als Zielmarke der Funktion *tsp_rollbackTo* sowie beim Zurücksetzen der Objektmarkierungen in Schritt 1 dieser Operation;

4. Ausführung eines Store-Commit, um den Zustand des Tycoon-Systems und des Logs festzuschreiben.

Abbildung 3.4 zeigt den Log-Ausschnitt aus Abbildung 3.3 nach Setzen eines Sicherungspunktes. Zur Durchführung eines Tycoon-Commit werden nahezu dieselben Schritte ausgeführt. Der einzige Unterschied ist, daß statt einen Log-Eintrag für das Commit zu schreiben das Log abgeschnitten wird.

Um den Zustand des Objektspeichers bis zu einem bestimmten Sicherungspunkt zurückzusetzen, werden die Logeinträge in umgekehrt chronologischer Reihenfolge abgearbeitet, bis man beim geforderten Sicherungspunkt angelangt ist. Letzteres läßt sich über die im Log vorhandenen Einträge für Sicherungspunkte feststellen. Bei Antreffen eines Änderungseintrages wird die Kopie wieder auf das Originalobjekt vollständig zurückkopiert; bei Antreffen eines Eintrags für eine Änderung der Wurzel wird der im Eintrag enthaltene Wert auf die Wurzel zurückkopiert. Bei Erreichen des Log-Eintrages für den anvisierten Sicherungspunkt wird das Log hinter diesem Eintrag abgeschnitten, so daß auch auf Seiten des Logs der zum Zielsicherungspunkt gültige Zustand wieder vollständig hergestellt ist. Zum Abschluß muß ein Store-Commit durchgeführt werden, um den restaurierten Zustand festzuschreiben. Handelt

es sich beim den angepeilten Sicherungspunkt um den letzten gesetzen, ist die Ausführung einer kompensierenden Store-Transaktion nicht notwendig und die Durchführung eines Store-Rollbacks genügt.

3.2.4 Interaktion von Logging und Garbage Collection

Da die Log-Struktur im Store gespeichert wird, ist sie einerseits der Garbage Collection unterworfen, andererseits beeinflußt sie die Garbage Collection aber auch, weil in den Log-Einträgen teilweise Referenzen auf Objekte vorhanden sind, die sonst nicht weiter referenziert werden. Beim Zusammenwirken von Logging und Garbage Collection müssen die folgenden beiden Phänomene unbedingt vermieden werden.

- Teile der Logstruktur oder der dazugehörigen Daten wie z.B. Objektkopien werden durch die Garbage Collection entfernt und bewirken ein Scheitern des Zurücksetzens. Dieser Effekt kann unmöglich eintreten, weil eine Referenz auf den Anfang der Log-Liste persistent und von der Speicherwurzel aus transitiv erreichbar gespeichert wird, somit die gesamte Logliste transitiv erreichbar ist und bei einer Garbage Collection nicht entfernt wird. Gleiches gilt für die von den Log-Einträgen referenzierten Objektkopien sowie für die Liste der neu angelegten Objekte, deren Anfang auch transitiv vom Wurzelobjekt erreichbar ist.
- Die Garbage Collection wird durch das Logging behindert, weil in der Log-Struktur Referenzen auf Objekte enthalten sind, die auch beim Zurücksetzen nicht mehr gebraucht werden. Hierdurch wird Speicherplatz verschwendet und letztlich die Performanz des Systems beeinträchtigt.

Der letzte Punkt läßt sich durch eine Analyse der Log-Struktur und des Logging-Verhaltens widerlegen.

Am häufigsten sind im Log Änderungseinträge zu finden, die für ein eventuelles Zurücksetzen benötigt werden und demzufolge wie schon im ersten Punkt erwähnt nicht entfernt werden dürfen. Durch die Markierung der im Log verzeichneten Objekte wird verhindert, daß pro Sicherungspunkt und geändertem Objekt mehr als eine Kopie angelegt wird. Wird ein Sicherungspunkt durch ein Tycoon-Commit unerreichbar, wird durch Abschneiden des Logs sichergestellt, daß auch alle mit diesem Sicherungspunkt zusammenhängenden Log-Einträge und Objektkopien nicht mehr transitiv erreichbar sind. Es kann vorkommen, daß Objektkopien Referenzen auf ansonsten unreferenzierte (und auch nicht im Log verzeichnete) Objekte enthalten; diese dürfen aber auch nicht durch eine Garbage Collection entfernt werden, weil sie für ein eventuelles Zurücksetzen erhalten werden müssen.

Die einzige Behinderung der Garbage Collection entsteht bei temporären Objekten wie z.B. lokalen Variablen, die beim Eintritt in eine Funktion angelegt werden und eigentlich gleich bei Verlassen der Funktion entfernt werden könnten. Da eine Referenz auf diese Objekte in der Neuliste verzeichnet wird, und diese Liste erst beim nächsten Sicherungspunkt aufgelöst wird, besteht immer mindestens eine Referenz auf die temporären Objekte, so daß eine frühzeitige Garbage Collection der überflüssigen temporären Objekte verhindert wird. Dieses Problem könnte durch die Einführung sogenannter schwacher Referenzen (*weak references*, dazu Ref??) verhindert werden. Diese werden von der Garbage Collection nicht als vollwertige Referenzen behandelt und ein ausschließlich schwach referenziertes Objekt wird trotzdem entfernt,

allerdings werden die bestehenden schwachen Referenzen auf ungültige Werte gesetzt. Würde das TSP einen solchen Mechanismus vorsehen, ließen sich die neuen Objekte durch schwache Referenzen in der Neuliste verzeichnen und die Garbage Collection für die neuen Objekt könnten ungehindert stattfinden.

Zusammenfassend kann gesagt werden, daß ein sinnvolles Zusammenwirken von Logging und Gargabe Collection mit vertretbarem Aufwand erreicht wurde. Der derzeitige Implementierungsstand ist in dieser Hinsicht bereits fast optimal und kann nur noch mit der Einführung schwacher Referenzen verbessert werden.

3.3 Bewertung des Logging-Verfahrens

Durch die Logging-Zwischenschicht entsteht für viele TSP-Funktionen ein Zusatzaufwand, der, wenn er zu hoch wäre, die Einsetzbarkeit des gewählten Implementierungsansatzes in Frage stellen würde. Empirische Vergleichsmessungen einiger ausgewählter Tycoon-Programme bestätigen allerdings, daß der durch die Logstruktur erforderliche Zusatzaufwand weniger als 30% Performanzunterschied mit sich bringt. Bevor die Vergleichsmessungen präsentiert werden, soll noch einmal die Belastung der Operationen durch das Logging zusammengefaßt werden. Zum Abschluß wird das in diesem Kapitel entwickelte Verfahren einem anderen für Tycoon entwickelten Logging-Ansatz ([Niederée 92]) gegenübergestellt.

Loggingaktivität ist nur bei Operationen mit einem Seiteneffekten (z.B. Objektänderung, aber auch Erzeugung) oder bei transaktionalen Operationen, die auf das Log zurückgreifen müssen, erforderlich. Im einzelnen lassen sich folgende Zusatzaufwände feststellen:

- Bei wiederholter Änderung eines Objektes wird zuerst eine Kopie angelegt und im Log verzeichnet, danach werden nur noch die Kopie-Markierungen geprüft. Der mit dem Prüfen der Markierung verbundene Aufwand ist sehr klein und kann daher vernachlässigt werden. Der mit dem Kopieren eines Objektes und Anlegen eines Log-Eintrages verbundene hohe Aufwand muß auf die gesamte Anzahl der Änderungsoperationen umgelegt werden. Mit steigender Anzahl von Änderungsoperationen für ein bestimmtes Objekt nimmt folglich der Zusatzaufwand pro Einzeloperation ab.

- Objekterzeugungen haben demgegenüber höhere konstante Zusatzkosten, weil damit auch immer die Erzeugung eines Eintrages für die Neuliste verbunden ist. Objekterzeugungen sind jedoch im Vergleich zu Objektänderung seltener, weswegen man erwarten kann, daß die hohen Zusatzkosten der Objekterzeugung während der Laufzeit weniger ins Gewicht fallen.

- Die Änderung der Wurzel des Objektspeichers wird immer von der Erzeugung eines Log-Eintrages begleitet und hat so ebenfalls hohe konstante Zusatzkosten. Diese Operation ist aber so selten, daß die damit verbundenen Kosten unbedeutend sind.

- Der Aufwand zum Setzen eines Synchronisationspunktes durch *transaction.commit()* oder *transaction.savepoint()* erhöht sich gegenüber der Durchführung einer Store-Commit-Operation ebenfalls, weil zusätzlich noch Log und Neuliste inspiziert werden. Allerdings wird immer nur ein Ausschnitt des Logs inspiziert und die Neuliste anschließend entfernt, so daß der auf die TSP-Operationen umgelegte Zusatzaufwand konstant ist und nicht mit der Anzahl der Synchronisationspunkte ansteigt.

Die Auswirkungen der Logging-Aktivität auf die Laufzeit wurde in einem Tycoon-System durch Kompilation mehrere Module gemessen. Der Grund für die Auswahl von Kompilationsvorgängen als Basis der Messungen war, daß die Kompilation sehr rechenintensiv ist und viele Schreib- und Lesezugriffe macht, aber auch viele temporäre Objekte erzeugt, so daß alle TSP-Operationen in einem ausgewogenen Verhältnis in Anspruch genommen werden. Die Messungen wurden wiederholt mit und ohne Logging für zwei Module der Standardbibliothek, *print* und *int*, durchgeführt; die gemittelten Meßergebnisse sind in Tabelle 3.1 abgebildet. Man kann an den Zahlen erkennen, daß der Performanzunterschied 30% nicht übersteigt. Eine Messung des durch Logging entstehenden zusätzlichen Platzbedarf wurde vorerst unterlassen, weil mit dem Fehlen von schwachen Referenzen im TSP noch keine Garbage Collection von temporären Objekten möglich ist.

	ohne Logging	mit Logging	Performanzverlust
Kompilation Modul *print*	23 s	31 s	25,9 %
Kompilation Modul *int*	35 s	48 s	27,1 %

Tabelle 3.1: Resultate der Performanzmessungen beim Einsatz vom Logging

In [Niederée 92] wird ein Verfahren zur Implementation von Transaktionen auf Tycoon-internen Daten vorgestellt, das auch auf einem im Tycoon-Store liegenden Undo-Log basiert. Die Arbeit spricht auch eine Erweiterung zur Unterstützung von Sicherungspunkten an und schlägt ebenfalls vor, für diese spezielle Einträge im Log zu verzeichnen. Im Unterschied zu dem in diesem Kapitel entwickelten Verfahren wird aber das Logging in der Anwendungsebene durchgeführt, weshalb im Log als Kompensationsinformation Operationen verzeichnet werden.[2] Ein solches Verfahren hat zwei prinzipielle Nachteile gegenüber einem Logging-Verfahren auf TSP-Ebene. Zum einen muß die Anwendung selbst das Logging durchführen, was den Implementationsaufwand des Anwendungsprogrammierers deutlich erhöht. Zum anderen ist es nicht möglich, den vollständigen Anwendungszustand in das Transaktionskonzept einzubeziehen, weil beispielsweise interne Datenstrukturen wie die Thread-Liste des Laufzeitsystems nicht in der Anwendungsebene zur Verfügung stehen. Folglich können deren Modifikation im Verlauf einer Transaktion auch nicht durch Ausführung von im Log verzeichneten Operationen in der Anwendungsebene kompensiert werden.

[2] Es handelt sich also um ein Operations-Logging-Verfahren, das jedoch nicht für flüchtige oder transaktionale externe Daten geeignet ist und sich auch vollständig von den in den nächsten beiden Kapiteln vorgestellten Verfahren unterscheidet.

4. Sicherungspunkte für Operationen auf flüchtigen externen Daten

Ein Großteil der Ausführungsumgebung eines Workflows ist durch nicht-persistente externe Daten, wie z.B. Elemente einer graphischen Benutzeroberfläche oder Netzwerkverbindungen, geprägt. Um für diese Art von Daten ein transaktionales Verhalten zu simulieren, muß deren Wiederherstellbarkeit und Zurücksetzbarkeit gewährleistet sein. Der gewählte Lösungsansatz basiert auf dem chronologischen Aufzeichnen der ausgeführten Operationen in einer Log-Datenstruktur. Die Wiederherstellbarkeit des externen flüchtigen Zustands wird durch einen neuartigen Wiederanlaufmechanismus erreicht, der zur Fehlererholung die bis zum letzten Sicherungspunkt in der Log-Struktur akkumulierten Operationen wiederholt. Die Rücksetzbarkeit der einzelnen externen Operationen wird durch Verzeichnen der kompensierenden Operationen in der Log-Struktur erreicht. Der Logging-Mechanismus kann auch zur Unterstützung der Migration persistenter Threads dienen, um auf dem Zielknoten deren flüchtigen externen Zustand wiederherzustellen. Wie in Tycoon üblich, wird das Operations-Logging nicht durch eine syntaktische Erweiterung der Schnittstellenspezifikation zur Verfügung gestellt, sondern als Erweiterung der Programmierumgebung durch ein Bibliotheksmodul, das Logging-Funktionen speziell für flüchtige externe Daten anbietet.

Dieses Kapitel gliedert sich wie folgt. Nach einer Übersicht der Ziele des Operations-Logging und einer Einordnung der Logging-Bibliothek in die Tycoon-Gesamtarchitektur wird eine Charakterisierung externer flüchtiger Operationen vorgenommen. Dies dient der Entwicklung eines abstrakten Modells externer flüchtiger Operationen und von Effizienzkriterien für die darauf ausgeführten transaktionalen Operationen. Das Modell und die Kriterien werden für die Entwicklung einer Programmierschnittstelle für das Operations-Logging herangezogen, die im darauf folgenden Abschnitt vorgestellt und durch ein kurzes Anwendungsbeispiel veranschaulicht wird. Die den Rest des Kapitels beschreibt die Implementierung und beginnt mit einer Vorstellung der für das Logging benötigten Datenstrukturen. Darauf aufbauend werden die Algorithmen für das Zurücksetzen und die Wiederherstellung erläutert; anschließend wird gezeigt, wie die Log-Struktur durch Herauslöschen sich gegenseitig neutralisierender Operationen komprimiert wird. Den Abschluß des Kapitels bildet die Darstellung der Restriktionen und Anwendungsgrenzen des Operations-Logging als Technik zur transaktionalen Verarbeitung externer flüchtiger Daten.

4.1 Übersicht und Ziele des Loggingansatzes für flüchtige externe Daten

Da die Zustandsänderungen externer Daten in Tycoon allgemein nur über die darauf ausgeführten Operationen sichtbar sind, ist es naheliegend, die Operationsausführung persistent aufzuzeichnen, um ein transaktionales Verhalten zu simulieren. Zur Wiederherstellung des flüchtigen externen Zustandes müßten diese Operationen chronologisch wiederholt werden, was auf die Speicherung der Operationen in einer Log-Struktur hinweist. Das Zurücksetzen des externen Zustandes ist ebenfalls nur über die Ausführung externer Operationen möglich; diese müßten so beschaffen sein, daß sie die vorherig ausgeführten Operationen kompensieren. Aus diesen Überlegungen folgt, daß die Persistenz und Rücksetzbarkeit externer flüchtiger Daten auf Basis einer Log-Struktur für externe Operationen implementiert werden muß. In dieser Log-Struktur werden die ausgeführten Operationen und deren kompensierende Operationen chronologisch aufgezeichnet. Um die kompensierenden Operationen parallel zu den eigentlichen Operationen bei ihrer Ausführung im Log zu verzeichnen, müssen die kompensierenden Operationen in irgendeiner Form vom Programmierer spezifiziert werden. Anders als im letzten Kapitel ist die Operationssemantik nicht mehr dem System bekannt, folglich können auch die kompensierenden Operationen nicht automatisch durch das System im Log verzeichnet werden. Es wäre möglich, die Tycoon-Programmiersprache TL so zu erweitern, daß beispielsweise in einem Definitionsmodul zu jeder Funktion die kompensierenden Operationen angegeben werden könnten. Eine Spracherweiterung widerspricht dem Grundansatz von Tycoon, Systemerweiterungen flexibel durch TL-Bibliothken einzubinden. Aus diesen Gründen wurde entschieden, die Operations-Logging-Struktur auf TL-Ebene in Form eines Bibliotheksmoduls anzubieten, das Funktionen zur chronologischen Aufzeichnung externer Operationen und deren kompensierende Operationen exportiert. Bei Aufruf der externen Operationen werden diese Funktionen benutzt, um die Log-Struktur mit den relevanten Informationen zu füllen.

Aber auch der Anwendungsprogrammierer stellt Anforderungen, die bei der Gestaltung einer Programmierschnittstelle für das Operations-Logging berücksichtigt werden müssen. Zum einen sollen die transaktionalen Eigenschaften flüchtiger Daten nicht durch einen erhöhten Programmieraufwand erkauft werden sondern möglichst transparent auf der Ebene der Anwendungsprogrammierung sein. Beispielsweise wäre es nicht akzeptabel, wenn eine bestehende, nicht-„persistente“ Anwendung vollständig umstrukturiert werden müßte, nur um die verwendeten flüchtigen Daten transaktional zu machen. Diese Forderung schließt aus, daß die Anwendung selber die ausgeführten Operationen im Log verzeichnen oder die kompensierenden Operationen kennzeichnen muß. Zum anderen darf die zusätzliche Funktionalität, die die Anwendung z.B. in Form der Wiederherstellbarkeit des vollständigen Zustands gewinnt, nicht ineffizient erbracht werden. So muß z.B. bei der Wiederholung der aufgezeichneten Operationen zum Zweck der Wiederherstellung unbedingt vermieden werden, daß unnötige Operationen ausgeführt werden. Beispielsweise darf ein Fenster bei der Wiederherstellung nicht 50 mal geöffnet und geschlossen werden, auch wenn dies der Historie des vorangegangenen Programmlaufs entspricht.

Aus diesen Programmieranforderungen läßt sich die Vorgehensweise für die Durchführung des Operations-Logging ableiten. Die eigentliche Aufzeichnung einer Operation und ihrer kompensierenden Operation im Log wird von dem die externen Operationen exportierenden Bibliotheksmodul vorgenommen. Innerhalb dieses Bibliotheksmoduls ist auch die Semantik der Operationen bekannt und damit auch, wie die einzelnen exportierten Funktionen kompensiert

werden müssen, so daß die Logging-Aufrufe selbständig von der *Bibliothek* durchgeführt werden können. Für das *Anwendungsprogramm* sind diese Logging-Vorgänge transparent, weil nach wie vor nur die externen Operationen aufgerufen werden. Die erwähnte Effizienzanforderung zeigt aber auch, daß eine naive Vorgehensweise beim Logging und den transaktionalen Operationen (Beispiel: zum Wiederanlauf wird jede jemals ausgeführte Operation wiederholt) nicht zum Ziel führt. Es muß also unterschieden werden, welche Operationen in bezug auf die Log-Struktur „wichtig" sind und welche nicht. Beim obigen Beispiel erkennt man, daß zumindestens die ersten 499 Öffnungs- und Schließvorgänge des Fensters in diesem Sinne unwichtig sind[1]; ebenfalls ersichtlich ist, daß die Unterscheidung von dem speziellen Anwendungsfall gemacht werden muß, und nicht von der Bibliothek, die die Operationen zur Verfügung stellt. Mit anderen Worten, die Applikation muß in einer noch genauer zu bestimmenden Form in die Logging-Vorgänge eingreifen können, um unwichtige Operationen zu kennzeichnen.

Eine graphische Darstellung der diskutierten Designentscheidungen sowie des Zusammenwirkens der beteiligten Module ist in Abbildung 4.1 zu sehen. Das Modul *volatile* implementiert das globale Operations-Log und bietet verschiedene Gruppen von Funktionen darauf an. Die für die Anpassung der externen Operationen zuständigen Module führen die primären Logging-Funktionen durch: Sie veranlassen die Aufzeichnung von Operationen und ihren kompensierenden Operationen. Die eigentliche Anwendung kann zur Optimierung des Logging-Verhaltens in die von den Bibliotheken initiierten Logging-Vorgängen eingreifen. Die Wiederherstellung und das Zurücksetzen des externen Zustandes wird auch vom Modul *volatile* übernommen, allerdings werden diese Operationen durch das Modul *transaction* angestoßen. Dadurch ist gewährleistet, daß interner und externer Zustand immer synchron zurückgesetzt bzw. wiederhergestellt werden.

Werden Operationen ohne weitere Vorkehrungen bei ihrem Aufruf im Log verzeichnet, ergibt sich das Problem, daß das Logvolumen permanent ansteigt. Dies hätte den Nachteil, daß besonders bei langlebigen Aktivitäten der Platzverbrauch durch das Log prohibitiv hoch sein kann. Zudem hat, wie man später sehen wird, der Wiederanlauf eine zur Log-Länge proportionale Laufzeit, weshalb ein hohes Log-Volumen außerdem zu Performanzproblemen führen kann. Glücklicherweise müssen nicht alle Log-Einträge permanent aufbewahrt werden. Als veranschaulichendes Beispiel stelle man sich dazu ein Ausgabefenster einer Applikation vor, welches schon vor dem letzten Commit gelöscht wurde. Da das Fenster in der Applikation nicht mehr existiert, braucht es auch beim Wiederanlauf nicht nocheinmal erzeugt zu werden. Ebenso kann es auch nicht im Rahmen einer Rücksetzoperation neu erzeugt werden, weswegen die zu diesem Fenster gehörenden Log-Einträge aus dem Log entfernt werden könnten. Um den Umfang des Logs zu beschränken, muß das Logging-Modul *volatile* also mit einem Mechanismus ausgestattet sein, der eine periodische Bereinigung des Logs um die überflüssigen Einträge erlaubt. Aus dem Beispiel ist bereits klar, daß dazu weitere semantische Informationen über die im Log verzeichneten Operationen erforderlich sind.

4.2 Charakterisierung externer Operationen auf flüchtigen Daten

Im vorangegangenen Abschnitt wurde bereits gezeigt, daß die im Log verzeichneten Operationen semantisch unterschieden werden müssen. Zum einen kann man einen Effizienzgewinn

[1] In Abschnitt 4.2 wird noch konkretisiert, wann Operationen als „wichtig" und „unwichtig" anzusehen sind.

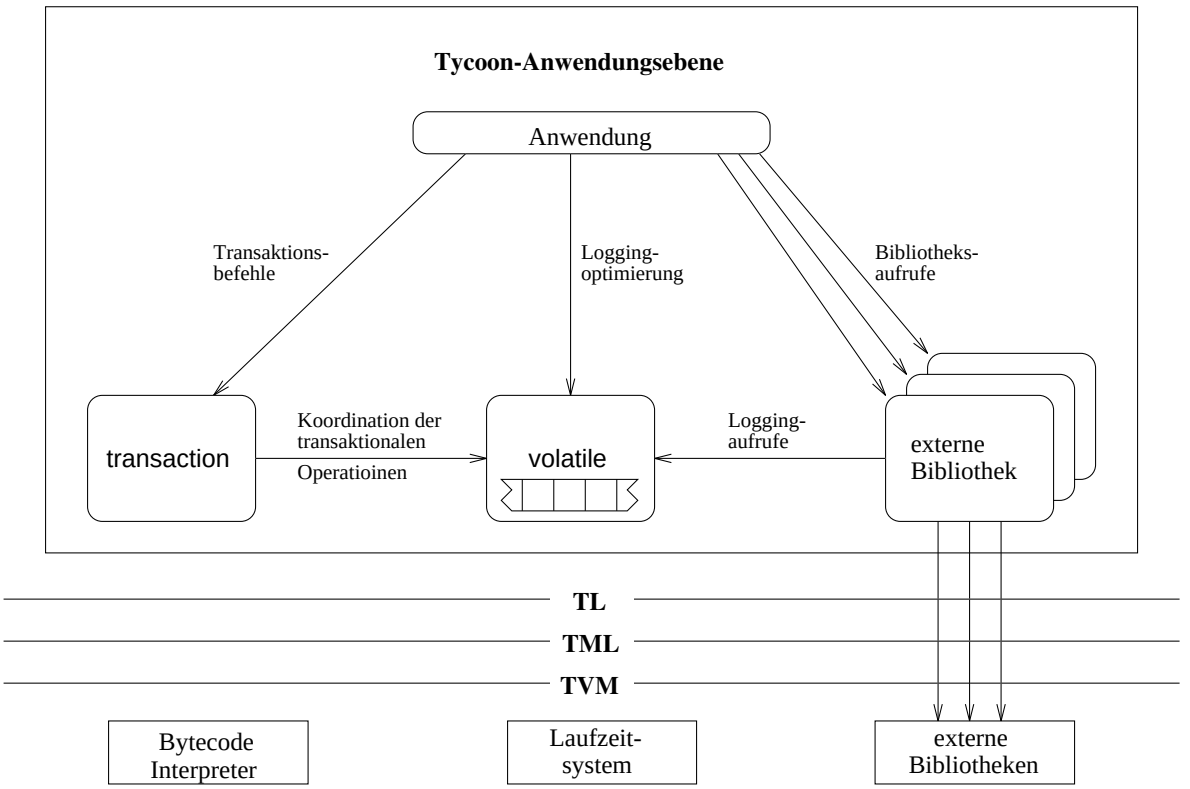

Abbildung 4.1: Die Architektur des Operations-Logging

erzielen, wenn man unwichtige Operationen bei den transaktionalen Operationen unberücksichtigt läßt; zum anderen ist das Erkennen überflüssiger Einträge im Log eine Voraussetzung für dessen periodische Bereinigung und Komprimierung. In diesem Abschnitt soll aus der Charakterisierung externer Operationen für diese ein abstrakes Modell erarbeitet werden, welches die zur Effizienz und Log-Bereinigung relevanten semantischen Unterscheidungen ausdrücken kann. Unter Zuhilfenahme dieses Modells werden Effizienzkriterien für die transaktionalen Operationen formuliert, die Richtlinien für die Gestaltung der Logging-Schnittstelle und der Algorithmen zur transaktionalen Verarbeitung aufstellen.

Da beim Wiederanlauf oder Zurücksetzen Operationen wiederholt oder kompensiert werden, ist es aus Effizienzgründen wünschenswert, daß eine minimale, aber dennoch ausreichende Menge an Operationen ausgeführt wird. Für die einzelnen transaktionalen Operationen bedeutet das folgendes:

Wiederanlauf Es werden nur die Operationen wiederholt, die nötig sind, um den Zustand zum letzten Sicherungspunkt wiederherzustellen. Damit ist die Wiederholung des 500fachen Öffnens und Schließens eines Fensters ausgeschlossen. Ebenso ausgeschlossen ist, daß z.B. ein externes Objekt erzeugt wird, nur um zu einem späteren Zeitpunkt beim Wiederanlauf erneut zerstört zu werden.

Zurücksetzen Es werden keine Operationen kompensiert, die anschließend wiederholt werden

müßten.[2] Zusätzlich sollen auch keine unwichtigen Operationen kompensiert werden.

Aus dieser Darstellung kann man schon die wichtigen semantischen Unterscheidungsmerkmale für Operationen herausziehen, die einen zu folgendem Operationsmodell führen. Der externe Zustand eines Programmes besteht aus explizit erzeugten oder implizit vorhandenen externen Objekten. Ein extern vorhandener Datenwert ist dabei als Objekt anzusehen, wenn er eine eigene Identiät (z.B. über einen Zeiger) besitzt. Im vorigen Abschnitt wurde schon mit dem Beispiel des Fensters einer graphischen Oberfläche ein explizit erzeugtes Objekt aufgeführt. Ein implizit vorhandenes Objekt könnte z.B. der unbenannte Kontext eines Grafikpaketes sein, mit dem globale Zeichenparameter wie Strichstärke und Farbe gesetzt werden. Der externe Zustand wird verändert, indem

- neue Objekte erzeugt oder bestehende gelöscht werden
- Operationen mit Seiteneffekten auf die Objekte ausgeführt werden (dabei kann dies auch implizit vorhandene Objekte betreffen).

Beim Löschen eines Objektes werden Erzeugung sowie alle auf das Objekt angewendeten Änderungsoperationen kompensiert. Die Änderungsoperationen kann man weiter in zwei Kategorien unterteilen:

1. Die Operationen werden einmal oder selten ausgeführt bzw. der betroffene Zustand wird nur einmal oder selten geändert. Das ist typischerweise der Fall, wenn der Titel eines Fenster gesetzt oder ein permanenter Parameter eines Grafikpaketes eingestellt wird. Diese Operationen sollen im folgenden als niederfrequent bezeichnet werden.
2. Die Operationen werden häufig ausgeführt bzw. der betroffene Zustand wird häufig geändert. In diese Kategorie fällt das Beispiel des wiederholt geöffneten und geschlossenen Fensters. Diese Operationen sollen im folgenden als hochfrequent bezeichnet werden.

Dabei ist zu beachten, daß die Unterscheidung, ob eine Operation der einen oder anderen Kategorie angehört, nur anhand ihrer speziellen Verwendung in einer Applikation, aber nicht für die Operation pauschal entschieden werden kann. So ist es denkbar, daß ein Fenstertitel in einer Applikation auch sehr häufig gesetzt wird, zum Beispiel im Fall einer Textverarbeitung, die Dokumentnamen im Fenstertitel anzeigt.

Häufig findet man in externen Bibliotheken auch eine hierarchische Anordnung der Objekte vor: Ein Objekt ist Subobjekt eines anderen, wenn mit dem Löschen des Hauptobjektes auch das Subobjekt gelöscht wird. Eine graphische Benutzeroberfläche ist ein gutes Beispiel, weil hier Darstellungselemente wie Fenster oft mit Popup-Menüs ausgestattet werden können. Diese Menüs besitzen eine eigenständige Identität und es können auch Änderungsoperationen auf sie ausgeführt werden, weswegen man sie als Objekte im Sinne des Modells bezeichnen muß. Wird das Fenster zerstört, verschwinden auch die Menüs, weswegen man sie auch als Subobjekte des Fensters bezeichnen kann. Die Existenz von Subobjekten soll in das Operationsmodell aufgenommen werden, weil durch das implizite Löschen von Objekten einige Log-Einträge überflüssig werden können, was unter anderem für die Durchführung der Logbereinigung wichtig ist.

[2] Dies schließt einen einfachen Ansatz aus, bei dem zuerst alle Operationen kompensiert werden und anschließend nur die Operationen vor dem anvisierten Sicherungspunkt wiederholt werden.

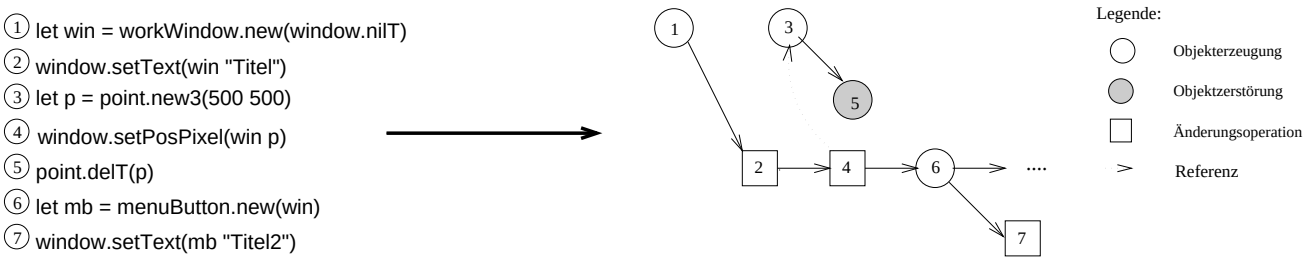

Abbildung 4.2: Beispiel zum Operationsmodell

Um die Ausführbarkeit einer Operation zu gewährleisten, muß sichergestellt sein, daß die Parameterobjekte der Operation existieren. Man muß deshalb als ein Operationsmerkmal auch dessen referenzierte Objekte aufnehmen.

Die vorgestellten Elemente des Operationsmodells sollen an dieser Stelle mit einem Beispiel illustriert werden. Abbildung 4.2 zeigt, wie der Code zur Erzeugung eines Fenster mit einem Popup-Menü mit der StarView-Bibliothek ([Stard93 93]) im Operationsmodell dargestellt würde. Die Verkettung der Operationen beschreibt deren Zusammengehörigkeit und keine zeitliche Anordnung (diese wird natürlich gemäß dem Code vorgenommen); beispielsweise ist Operation 2 eine Änderungsoperation des mit Operation 1 erzeugten Objektes.

Anhand dieses Operationsmodells lassen sich die zu Anfang dieses Abschnitts genannten Forderungen nach Effizienz präziser als Effizienzkriterien formulieren:

1. Zum Wiederanlauf werden nur alle zum Zeitpunkt des letzten Sicherungspunktes noch nicht kompensierten Operationen wiederholt. Damit dies korrekt durchgeführt werden kann, sollen auch die für Parameterobjekte verzeichneten Operationen wiederholt werden.

2. Beim Zurücksetzen werden nur Operationen wiederholt, die nach dem Ziel-Sicherungspunkt kompensiert wurden und es werden nur die seit dem Ziel-Sicherungspunkt ausgeführten Operationen kompensiert.

3. Hochfrequente Operationen werden allgemein nicht einzeln wiederholt oder kompensiert, sondern der durch sie erreichte Endzustand wird insgesamt wiederhergestellt.

Im Sinne der Log-Bereinigung sollen solche Log-Einträge als überflüssig betrachtet werden, die weder für die Ausführung des Wiederanlaufs noch für das Zurücksetzen auf einen beliebigen Sicherungspunkt oder den Anfang der Transaktion benötigt werden.

Die Kriterien präzisieren in den Worten des gewählten Operationsmodells den vorher anhand von Beispielen und Gegenbeispielen beschriebenen Effizienzbegriff. Das vorgestellte Operationsmodell ist also adäquat für das Logging von Operationen auf externen flüchtigen Daten, weil es die für ein effizientes Verfahren wichtigen semantischen Aspekte der Operationen erfaßt. Im nächsten Abschnitt wird eine Logging-Schnittstelle vorgestellt, die dieses Operationsmodell umsetzt.

4.3 Eine Programmierschnittstelle für das Logging von flüchtigen externen Daten

Die in diesem Abschnitt vorgestellte Schnittstelle erlaubt das Logging von Operationen auf externen flüchtigen Daten, wobei das Operationsmodell des vorigen Abschnittes zugrunde gelegt ist. Die im Log zu verzeichnenden Operation werden in Erzeugungs- und Änderungsoperationen unterschieden und es wird verlangt, daß jeder Operation auch eine kompensierende Operation durch den Benutzer der Schnittstelle zugeordnet wird. Für die Objekterzeugung ist dies die Löschoperation, die nicht nur die Erzeugung sondern auch alle Änderungen und Subobjekterzeugungen kompensiert und zu diesem Zweck vom Programmierer explizit aktiviert werden kann. Um die Aufzeichnung von hochfrequenten Operation im Log zu vermeiden, läßt sich das Logging abschalten und es können sogenannte *Restaurationsprozeduren* registriert werden. Letztere haben den Zweck, den durch die hochfrequenten Operationen erreichte Zustand zusammengefaßt wiederherzustellen. Die von der Bibliothek exportierten Funktionen lassen sich entsprechend der von ihnen Gebrauch machenden Module gruppieren: externe Bibliotheken verwenden die Funktionen zur Erzeugung der Log-Einträge für Operationen; Anwendungen optimieren das Logging-Verhalten durch zeitweiliges Deaktivieren des Logging-Vorgangs und Registration von Restaurationsprozeduren; das Modul *transaction* stößt das Setzen von Sicherungspunkten und Commit an und initiiert die Durchführung des Zurücksetzens und des Wiederanlaufs. Die folgende Darstellung der Schnittstellenfunktionen hält sich an diese Gruppierung. Die vollständige Schnittstelle ist zusammenhängend in Anhang A aufgeführt.

Externe flüchtige Objekte besitzen eine Identität, so daß sie als Parameter für Änderungsoperationen eingesetzt werden können. Das Modul *volatile* exportiert einen polymorphen Typoperator

Let *T(E <:**Ok**) =* ***Tuple*** *get():E* ***end***,

der ein externes flüchtiges Objekt vom Typ E einkapselt. Die *get*-Funktion liefert diesen Wert, der einem C-Zeigerwert entsprechen muß (in Tycoon als *word.T* repräsentiert). Letzteres wurde so gewählt, weil in der Programmiersprache C, die häufig die Implementierungssprache externer Bibliotheken ist, Objekte üblicherweise durch Zeiger identifiziert und referenziert werden. Werden die Objekte einer externen Bibliothek auf andere Weise identifiziert, beispielsweise durch Zeichenketten, muß die Referenz in einem Adaptionsmodul auf C-Ebene durch etwas Buchhaltung in einen Zeiger umgewandelt werden. Der exportierte Typ

Let *Any = T(**Ok**)*

ist der Supertyp aller *volatile.T(E)*-Typen und wird in der Signatur der nun folgenden Operationen verwendet.

Zur Erzeugung eines transaktionalen flüchtigen Objektes vom Typ E dient die Funktion

*create(E <:**Ok** create() :E destroy(:E) :**Ok***
*indepDestroy :**Bool** parent :Any references :**Array**(Any)) :T(E).*

Die Erzeugungs- und Löschfunktionen werden hier als zu verzeichnende Operationen *create* und *destroy* übergeben. Falls es sich bei dem erzeugten Objekt um ein Subobjekt handelt,

muß das Vaterobjekt als Parameter *parent* übergeben werden. Das Modul *volatile* exportiert ein ausgezeichnetes Objekt *nil :T(Nok)*, so daß auch die Erzeugung vaterloser Objekte durch *volatile.create* möglich ist. Das Operationsmodell sieht die Angabe von Parameterobjekten der Erzeugungs- und Löschfunktionen vor, die als Parameter *references* übergeben werden. Der Parameter *indepDestroy* gibt für ein Subobjekt an, ob dies explizit durch Aufruf der *destroy*-Parameterfunktion mit dem Hauptobjekt zerstört werden muß.[3] Die *create*-Schnittstellenfunktion verzeichnet die übergebenen Parameter in der Log-Struktur und erzeugt anschließend das Objekt durch Aufruf der *create*-Parameterfunktion. Der dadurch erhaltene (Zeiger-) Wert vom Typ *E* wird als ein „persistenter Zeiger" vom Typ *volatile.T(E)* zurückgeliefert.

Da eine direkte persistente Speicherung eines physikalischen Zeigerwertes unsinnig ist,[4] stellt der Datentyp *volatile.T(E)* eine Indirektion dar. Über die Attributfunktion

get() :E

läßt sich der aktuell gültige Zeigerwert des flüchtigen Objektes ermitteln.

Die Funktion

*log(V <:***Ok** *redo() :V undo() :***Ok** *parent :T references :***Array***(Any)) :V*

dient der Durchführung von Änderungsoperationen auf flüchtige Objekte. Die auszuführende Operation und deren kompensierende Operation werden durch die Parameter *redo* und *undo* angegeben. Der Parameter *parent* bezeichnet das Objekt, welches geändert werden soll. Die Schnittstelle exportiert auch ein ausgezeichnetes Null-Objekt (*nil :T(Nok)*), so daß auch Änderungsoperationen für nicht explizit erzeugte Objekte im Log verzeichnet werden können. Der Parameter *references* bezeichnet, wie auch bei *create*, die Parameterobjekte der *redo*- und *undo*-Funktionen. Bei Aufruf von *log* werden ebenfalls die übergebenen Parameter im Log verzeichnet, die *redo*-Funktion unmittelbar ausgeführt und deren Ergebnis zurückgeliefert.

Eine Objektlöschung wird nicht als Änderungsoperation ausgeführt, sondern von der Schnittstellenfunktion

*destroy(E <:***Ok** *vol :T(E)) :***Ok**,

die die bei Aufruf von *create* im Log verzeichnete *destroy*-Funktion aufruft.

Um zu vermeiden, daß hochfrequenten Operationen im Log verzeichnet werden, läßt sich die Logging-Aktivität mittels

*disableLogging() :***Ok**
*enableLogging() :***Ok**
*restoreLogging() :***Ok**

steuern. Mit *disableLogging* wird das Logging deaktiviert, so daß Aufrufe von *create* oder *log* nur noch die Parameterfunktionen ausführen, aber keine Log-Einträge mehr erzeugen. Mit *enableLogging* läßt sich das Logging wieder aktivieren; alternativ kann mit *restoreLogging* der vorige Aktivierungszustand wiederhergestellt werden. Als Abkürzung kann eine Funktion auch mit

[3] Die Verwendung dieses Parameters, der nicht durch das Operationsmodell motiviert ist, wird in Abschnitt 4.5.3 erläutert.

[4] Bei wiederholter Erzeugung eines flüchtigen Objektes während des Wiederanlaufs oder Zurücksetzens wird ein neuer Zeigerwert erzeugt; der alte ist dann ungültig.

*unlogged(do() :**Ok**) :**Ok***

ausgeführt werden, was der Sequenz

disableLogging()
do()
restoreLogging()

entspricht. Zur Ermittlung des Aktivierungszustands ist die Funktion mit der Funktion

*loggingEnabled() :**Bool***

vorgesehen.

Um den von den hochfrequenten Operationen erreichten Zustand bei einem Wiederanlauf oder Zurücksetzen wiederherstellen zu können, sind in der Schnittstelle sogenannte Restaurationsprozeduren vorgesehen. Diese werden mit

*registerRestore(restore() :**Ok** execNow :**Bool** parent :Any) :RestoreProc*

registriert (*RestoreProc <:**Ok*** wird auch exportiert). Von der Parameterfunktion *restore* wird erwartet, daß sie den Teil des Zustandes, der durch hochfrequenten Operationen geändert wurde, zusammengefaßt wiederherstellt. Dies ist erforderlich, weil die hochfrequenten Operationen beim Logging ausgeblendet wurden und deshalb nicht im Log verzeichnet sind. Die Liste aller registrierten Restaurationsprozeduren muß den nicht durch Logging erfaßten Teil des Zustandes vollständig wiederherstellen und wird immer zum Abschluß des Wiederanlaufs und Zurücksetzens einmalig ausgeführt. Eine Restaurationsprozedur kann auch über den Parameter *parent* mit einem flüchtigen, durch *create* erzeugten Objekt assoziiert werden. Dies ist sinnvoll, wenn die Restaurationsprozedur sich auf das Objekt bezieht und dessen Zustand wiederherstellt. Bei Zerstören des Objektes durch *destroy* wird dann die Restaurationsprozedur automatisch deregistriert. Der Parameter *execNow* schließlich gibt an, ob die zu registrierende Prozedur unmittelbar einmal ausgeführt werden soll.

Die restlichen Schnittstellenfunktionen betreffen die Durchführung von transaktionalen Operationen wie das Setzen eines Synchronisationspunktes, Wiederanlauf und Zurücksetzen und sind ausschließlich zur Verwendung durch das Modul *transaction* gedacht. Im einzelnen handelt es sich dabei um die Funktionen

*doSavepoint() :**Ok***
*doCommit() :**Ok***
*doRollbackTo(syncPoint :**Int**) :**Ok***
*doRestart(afterRollback :**Bool**) :**Ok***

Intern werden Sicherungspunkte als Zahlenwerte dargestellt, deswegen wird der Ziel-Sicherungspunkt bei der Funktion *doRollbackTo* auch durch eine Zahl (0 für das letzte Commit) angegeben.

4.4 Anwendungsbeispiel der Programmierschnittstelle

Die Verwendung der im vorigen Abschnitt vorgestellten Programmierschnittstelle sowie die Arbeitsteilung zwischen Applikation und Bibliotheken in bezug auf das Logging soll an einem kleinen Beispiel demonstriert werden. Die Beispielapplikation zeigt einen Meßwert, z.B. eine Spannung, als Balken in einem Fenster an. Dabei soll die Größe des Balkens proportional zur Größe des Meßwertes sein und die Anzeige muß bei Veränderung des Meßwertes aktualisiert werden. Als externe Bibliothek für die graphische Benutzeroberfläche wird die StarView-Bibliothek ([Stard93 93]) verwendet. Dieses kleine Beispiel verwendet bereits alle Elemente der Logging-Schnittstelle: bei dem Fenster handelt es sich um ein externes flüchtiges Objekt, das Zeichnen des Balkens sowie die Repositionierung und Größenänderung sind Änderungsoperationen des Fensters. Beim Wiederanlauf dürfen die Balkenbewegungen und die Veränderungen der Fensterposition und -größe nicht chronologisch nachvollzogen werden, weswegen die Applikation zeitweilig das Logging deaktivieren muß und für die Wiederherstellung des Balkens und der Fensterkoordinaten Restaurationsprozeduren registrieren muß.

Der erste Schritt ist die Erzeugung des Fensters:

```
let w = workWindowP.new(windowP.nilT)
```

Bei der Bibliothek *workWindowP* handelt es sich um die persistente Anpassung der StarView-Bibliothek *workWindow*.[5] Die Erzeugung eines Wertes vom Typ *workWindowP.T = volatile.T(workWindow.T)* wird in der Bibliothek durch einen Aufruf von *volatile.create* durchgeführt:

```
let new(parent :windowP.T) :T =
  volatile.create(
    fun() :workWindow.T  workWindow.new(parent.get())
    fun(w :workWindow.T) :Ok
      begin
        systemWindow.close(w)
        workWindow.delT(w)
      end
    false
    parent
    array end)
end
```

Die Funktionen *workWindow.new*, *systemWindow.close* und *workWindow.delT* werden unmittelbar von den StarView-Bibliotheken exportiert und benötigen daher die physikalischen Zeigerwerte als Parameter. Für *workWindow.new* muß dieser aus dem *volatile.T(workWindow)*-Parameter *parent* durch Aufruf der Funktion *get* extrahiert werden. Das Modul *window* implementiert *nilT* als *volatile.nil*, weswegen der obige Aufruf ein vaterloses Fenster erzeugt.

Die Applikation setzt danach den Fenstertitel mit

```
windowP.setText(w "Anzeige")
```

[5] Analog dazu ist *windowP* die persistente Anpassung von *window* usw.

Der *volatile.T(workWindow)*-Wert *w* wird von der Applikation wie ein C-Zeigerwert vom Typ *workWindow.T* für das entsprechende Fenster verwendet und als Parameter an *setText* übergeben. Der Logging-Aspekt ist deshalb bei der Benutzung einer an das Logging angepaßten Bibliothek für die Applikation nicht sichtbar und der Zusatzaufwand entsteht nur einmal bei der Anpassung der externen Bibliotheken. Bei *setText* handelt es sich um eine Änderungsoperation des Fensters, für die eine kompensierende Operation zum Setzen des alten Fenstertitels erforderlich ist und die wie folgt implementiert ist:

```
let setText(self :T str :String) :Ok =
  if volatile.loggingEnabled() then
    let oldSvString = ...
    volatile.log(
      fun() window.setText(self.get() str)
      fun() window.setText(self.get() svString.get(oldSvString))
      self
      array oldSvString end)
  else
    window.setText(self.get() str)
  end
```

Ist das Logging von der Applikation deaktiviert worden, muß die Bibliothek selber dafür sorgen, daß es auch bei Änderungsoperationen nicht durchgeführt wird.[6]

Bei der Ermittlung des alten Fenstertitels erhält man ein flüchtiges externes Objekt—ein Wert vom Typ *svString.T*—dessen Erzeugung ebenfalls im Log verzeichnet werden muß:

```
let oldSvString = volatile.create(
  fun() svString.getText(self.get())
  fun(w :svString.T) svString.delT(w)
  true
  self
  array end))
```

Dabei handelt es sich bei diesem String-Objekt nicht um ein Subobjekt des Fensters im eigentlichen Sinne (es wird nicht implizit mit dem Fenster gelöscht), aber das String-Objekt sollte mit dem Fenster gelöscht werden, da es dann nicht mehr benötigt wird.[7] Da das String-Objekt lokal zu *setText* ist und so nicht explizit von der Applikation entfernt werden kann, wird es als explizit zu löschendes (*indepDestroy == true*) Subobjekt des Fensters erzeugt.

Nach der Initialisierung wird mit der Anzeige des Meßwertes begonnen. Dazu wird in einer Schleife der Meßwert ausgelesen und ein Balken in analoger Größe in das Fenster gezeichnet:

```
loop
  let m = messwert.auslesen()
```

[6] Dies könnte auch vom Modul *volatile* erledigt werden, hätte allerdings einen Nachteil: für die Konstruktion der kompensierenden Operation muß der aktuelle Zustand ausgelesen werden (im Beispiel wird der alte Fenstertitel ermittelt), was zeitintensiv sein kann und bei deaktiviertem Logging nicht notwendig ist.

[7] Nach Löschen des Fensters sind implizit alle seine Änderungsoperationen kompensiert, so daß *setText* nicht mehr einzeln durch Ausführung der *undo*-Operation kompensiert werden muß.

```
  ...
  let r = rectangleP.new4(let left = xLeft
    let top = yTop
    let right = m
    let bottom = yBottom)
  outputDeviceP.drawRect(w r)
  rectangleP.delT(r)
  ...
end
```

Beim Wiederanlauf dürfen die innerhalb dieser Schleife ausgeführten Operationen nicht wiederholt werden, weil sich dadurch eine „flackernde“ Anzeige ergäbe und es außerdem unnütz Zeit kostet. In den Worten des Operationsmodells handelt es sich hierbei um Änderungsoperation, die nicht von Dauer sind und somit nicht als Initialisierungsoperationen bezeichnet werden können, was sie von *setText* unterscheidet. Die Operationen werden daher nicht im Log erfaßt:

```
volatile.unlogged(fun() :Ok
  loop
    ...
  end)
```

Damit durch den Wiederanlauf aber genau ein Balken gezeichnet wird, muß ein Restaurationsprozedur zu diesem Zweck registriert werden:

```
let restoreBar() :Ok =
  begin
    let r = rectangleP.new4(let left = xLeft
      let top = yTop
      let right = m
      let bottom = yBottom)
    outputDeviceP.drawRect(vWindow r)
    rectangleP.delT(r)
  end
let restore = volatile.registerRestore(restoreBar false w)
```

Diese Restaurationsprozedur bezieht sich auf die Tycoon-Variable *m*, die den Meßwert enthält und so immer den aktuellen Zustand des Balkens reflektiert. Dadurch wird bei Aufruf von *restoreBar* beim Wiederanlauf ein Balken in derselben Größe gezeichnet, wie sie zum Zeitpunkt des letzten Sicherungspunktes existierte. Bei Registrierung der Prozedur wird das Fenster *w* als Vaterobjekt angegeben, so daß die Prozedur bei Löschen des Fensters automatisch deregistriert wird.

An diesem Beispiel sieht man bereits, welche Aufgabe der Applikation in bezug auf das Logging zukommt: Das Logging muß bei hochfrequenten Operationen (der Balken wird mit jedem Schleifendurchlauf neu gezeichnet) ausgeblendet werden, anstelle dessen wird eine Restaurationsprozedur erforderlich, die den aktuellen Zustand in einem Schritt (im Beispiel wird nur ein Balken gezeichnet) wiederherstellen kann. Dabei dient diese Maßnahme einzig der Performanzverbesserung und kann entfallen, wenn der Performanzverlust als unbedeutend angesehen wird.

4.5 Implementationstechniken für transaktionale Verarbeitung

Dieser Abschnitt behandelt die Implementationstechniken, die für die Umsetzung des Operationsmodells und die transaktionale Verarbeitung der Operationen zum Einsatz kamen. Vor der detaillierten Beschreibung der Algorithmen zum Zurücksetzen, zum Wiederanlauf und zur Log-Bereinigung erfolgt eine Darstellung der Log-Struktur und der Logging-Aktivität während der Vorwärtsverarbeitung. Einige Details der Log-Struktur werden erst mit der Beschreibung der transaktionalen Operationen und der Log-Bereinigung verständlich werden. Den Abschluß bildet eine Diskussion der Grenzen des vorgestellten Logging-Verfahrens sowie ein Ausblick auf weiterführende Themen.

4.5.1 Datenstrukturen für das Operationslogging

Das Modul *volatile*, welches das Operations-Logging implementiert, führt als zentrale Datenstruktur ein Operations-Log, worin Operationen und die notwendigen Parameter und Zusatzinformationen zur Wiederholung und Kompensation sowie zur Verwaltung des Logs gespeichert werden. Dabei wird für jede verzeichnete Operation ein Eintrag im Log gehalten; dieser Eintrag gibt Auskunft über die zur Wiederholung oder Kompensation der Operation notwendigen Funktionen und Daten. Darüberhinaus enthält jeder Eintrag die für die Durchführung der Log-Bereinigung erforderlichen Verwaltungsdaten. Die eigentliche Log-Struktur ist eine lineare Liste, in der diese Einträge chronologisch nach den Zeitpunkten ihrer Erzeugung angeordnet sind; neu erzeugte Log-Einträge werden also immer an das Ende der Log-Liste angehängt. Neben dem Log existiert auch noch eine zweite globale Liste, die die Menge der aktuell registrierten Restaurationsprozeduren enthält.

Entsprechend der vom Operationsmodell unterschiedenen Operationstypen (Erzeugen, Löschen, Ändern) gibt es auch unterschiedliche Arten von Einträgen im Log, die durch einen gesonderten Eintragstyp für Sicherungspunkte ergänzt werden. Der Inhalt der Log-Einträge ist je nach ihrer Art unterschiedlich, es gibt jedoch einige Attribute, die allen gemeinsam sind und in Tabelle 4.1 abgebildet sind. Diese Attribute sind im einzelnen:

LSN steht für *log sequence number*, eine chronologisch ansteigende Sequenznummer, die die relative Position eines Log-Eintrages angibt. Da die Einträge im Log auch chronologisch vorgenommen werden, läßt sich über den Vergleich der LSNs zweier Log-Einträge auch deren zeitliche Präzedenz feststellen. Diese Eigenschaft wird für die transaktionalen Algorithmen wichtig sein.

Gültigkeitsmarke Diese gibt an, ob der Eintrag schon kompensiert wurde, wobei er als ungültig markiert wird. Diese Information ist nicht nur für die transaktionalen Algorithmen, sondern auch für die Log-Bereinigung wichtig.

Erzeugung: Ein Erzeugungseintrag wird von der *create*-Schnittstellenfunktion generiert und hat die in Tabelle 4.2 dargestellte Struktur. Die Attribute *create*, *destroy*, *parent* und *references* entsprechen den Parametern der *create*-Funktion. Bei den Attributen *subObjs*, *ops* und *restoreList* handelt es sich um Listen von Verweisen auf andere Log-Einträge oder Restaurationsprozeduren; diese werden bei Erzeugung neuer Subobjekte, Ausführung von Änderungsoperationen oder Registrierung von Restaurationsfunktionen ergänzt. Dadurch sind von

Attribut	Bedeutung
LSN	*log sequence number*; eine monoton steigende Sequenznummer, anhand derer die relative Position eines Eintrags im Log ermittelt werden kann
Gültigkeitsmarke	bei Kompensation einer Operation wird der zugehörige Eintrag als ungültig markiert

Tabelle 4.1: Allgemeine Attribute der Log-Einträge

Attribut	Bedeutung
create	Objekterzeugungsfunktion
destroy	Löschfunktion
externIndex	Referenz auf den Zeigerwert des externen Objektes
parent	Eintrag des Vaterobjektes
references	Liste der Parameterobjekte von create und destroy
subObjs	Liste von Subobjekten
ops	Liste von Änderungsoperationen
restoreList	Liste von Verweisen auf Restaurationsprozeduren
refCnt	Referenzzähler

Tabelle 4.2: Attribute des Erzeugungseintrags

jedem Erzeugungseintrag die damit zusammenhängenden anderen Log-Einträge erreichbar, was für die transaktionalen Algorithmen erforderlich ist.

Bevor die Bedeutung der verbleibenden Attribute, *externIndex* und *refCnt*, erläutert wird, soll auf die Implementierung des exportierten Datentyps *volatile.T* eingegangen werden. In Abschnitt 4.3 wurde schon gesagt, daß es sich dabei um einen quasi-persistenten Zeigerwert handelt. Eine Variable dieses Typs dient als Referenz eines externen flüchtigen Objektes, auch nach Wiederanlauf ihre Gültigkeit behält und „dasselbe“ externe Objekt referenziert.[8] Da der physikalische Zeigerwert nicht direkt persistent gemacht werden kann, ist eine Indirektion über *volatile.T* erforderlich. Ein Wert vom Typ *volatile.T* entspricht dem Erzeugungseintrag des entsprechenden Objektes. Dieser enthält aber nicht den physikalischen Zeigerwert, sondern eine weitere Referenz auf diesen. Der eigentliche Zeigerwert wird *außerhalb von Tycoon* in einem flüchtigen Array gespeichert, die im Erzeugungseintrag enthaltene Referenz ist ein Index in diesen Array (daher die Bezeichnung *externIndex*). Die sich daraus ergebende Konstruktion ist in Abbildung 4.3 zu sehen. Es wäre auch denkbar, den physikalischen Zeigerwert direkt im Erzeugungseintrag zu halten, dadurch ergäben sich aber Probleme beim Wiederanlauf und Zurücksetzen (auf diese wird in den folgenden Abschnitten näher eingegangen). Die wesentliche Funktion dieser doppelten Indirektion ist die Speicherung des Zeigerwertes außerhalb des persistenten Speichers und somit auch außerhalb der Transaktionen, die in diesem Speicher ausgeführt werden.

[8] Dabei handelt es sich selbstverständlich nicht um dasselbe Objekt, aber durch Wiederholung der im Log verzeichneten Operationen kann ein identisches Objekt neu erzeugt werden.

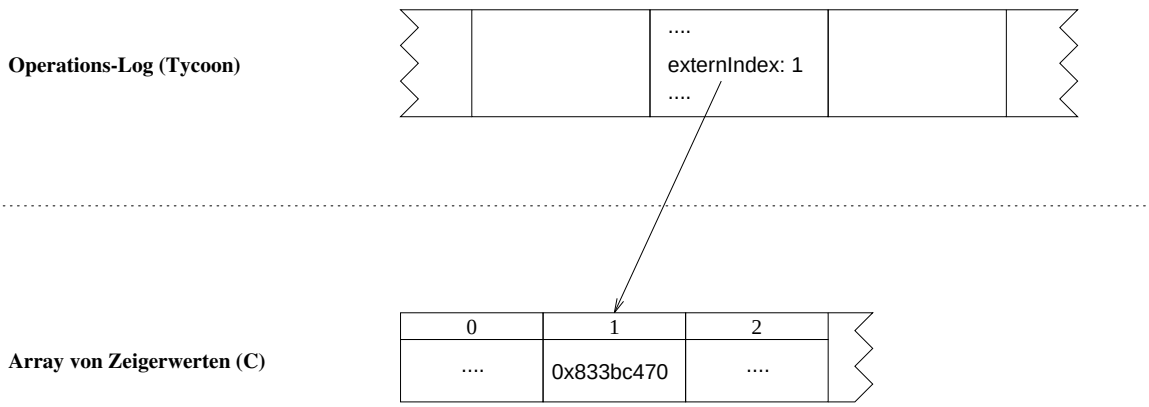

Abbildung 4.3: Externe Speicherung der Zeigerwerte

Das Attribut *refCnt* enthält einen Referenzzähler, der für die Log-Bereinigung erforderlich ist und angibt, wieviele Log-Einträge auf diesen Erzeugungseintrag verweisen. Der Referenzzähler eines Eintrags wird erhöht, wenn ein neuer Log-Eintrag angefügt wird, der das Objekt des ersteren referenziert. Analog wird der Referenzzähler verringert, wenn eine Referenz durch Löschen des referenzierenden Eintrages (nicht des zugehörigen Objektes, falls es sich um einen Erzeugungseintrag handelt) wegfällt.

Änderung: Der entsprechende Eintrag wird von der *log*-Schnittstellenfunktion erzeugt und hat die in Tabelle 4.3 dargestellte Struktur. Die Attribute entsprechen genau den Parametern der *log*-Funktion.

Attribut	Bedeutung
redo	eigentliche Operation
undo	kompensierende Operation
parent	Hauptobjekt
references	Liste der Log-Einträge der Parameterobjekte von redo und undo

Tabelle 4.3: Attribute des Änderungseintrages

Objektlöschung: Für die Objektlöschung wird beim Aufruf der *destroy*-Schnittstellenfunktion ein Eintrag ans Log angehängt, dessen Struktur in Tabelle 4.4 zu sehen ist. Das Löschen eines Objektes ist selbst keine neue Operation, sondern die Kompensation der Erzeugung des Objektes sowie implizit aller Änderungsoperationen und Subobjekte, weswegen im Log-Eintrag auch keine Operation vermerkt wird, sondern lediglich eine Referenz auf den Erzeugungseintrag. Die Objektlöschung wird zur Vereinfachung des Zurücksetzens im Log vermerkt.

Synchronisationspunkte: Beim Setzen eines Synchronisationspunktes wird dafür ein Eintrag an das Log angehängt (Tabelle 4.5), in dem die Nummer des Sicherungspunktes verzeichnet

Attribut	Bedeutung
createEntry	zugehöriger Erzeugungseintrag des gelöschten Objektes

Tabelle 4.4: Attribut des Löscheintrages

wird. Der letzte Commit-Punkt hat die Nummer 0, für die folgend gesetzten Sicherungspunkte wird der Zähler um jeweils 1 erhöht. Durch das Eintragen der Sicherungspunkte in das Log werden die relativen Zeitpunkte, zu denen diese gesetzt wurden, markiert. Über die LSN läßt sich von jeder im Log verzeichneten Operation herausfinden, ob diese vor oder nach einem bestimmten Sicherungspunkt ausgeführt wurde.

Attribut	Bedeutung
number	Nummer des Sicherungspunktes (0 für den letzten Commit-Punkt)

Tabelle 4.5: Attribut des Sicherungspunkteintrages

Der Log-Ausschnitt in Abbildung 4.4 illustriert die Log-Struktur und die verschiedenen Einträge anhand einer leichten Abwandlung des Code-Beispiels aus Abschnitt 4.4. Die Attribute *parent*, *subObjs*, *ops* und *createEntry* sind nicht explizit aufgeführt, sondern durch das Nummerierungsschema impliziert. Kompensierte Einträge, wie die des gelöschten Punktes p, sind grau unterlegt.

4.5.2 Loggingaktivität während der Vorwärtsverarbeitung

Im vorigen Abschnitt wurde schon vereinzelt angesprochen, wie die Log-Struktur während der Vorwärtsverarbeitung gefüllt wird; dieser Abschnitt soll die Logging-Aktivitäten zusammenhängend darstellen.

Bei Aufruf der Schnittstellenfunktionen *create* und *log* werden der Log-Struktur Erzeugungs- bzw. Änderungseinträge angehängt und die Informationen in den bestehenden Einträgen aktualisiert, soweit diese von den Neuzugängen betroffen sind. Zum einen müssen die Referenzzähler der Parameterobjekte[9] erhöht werden, zum anderen auch die Subobjekt- oder Operationsliste des Vaterobjektes ergänzt werden.

Bei Aufruf von *destroy()* wird die im Erzeugungseintrag des Objektes verzeichnete Löschfunktion ausgeführt, woraufhin a) die Erzeugungsoperation, b) alle Änderungsoperationen und c) alle Operationen im Zusammenhang mit Subobjekten kompensiert sind. Da die Einträge kompensierter Operationen im Log aus ungültig markiert werden, müssen auch alle Einträge von Änderungsoperationen und Subobjekten des gelöschten Objektes aufgesucht

[9] Diese Sprechweise ist nicht ganz korrekt, denn eigentlich werden Referenzzähler der Erzeugungseinträge der Parameterobjekte erhöht. Da aber die Parameterobjekte, Werte vom Typ *volatile.T*, auch gleichzeitig die Erzeugungseinträge sind, ist die Bedeutung unmißverständlich.

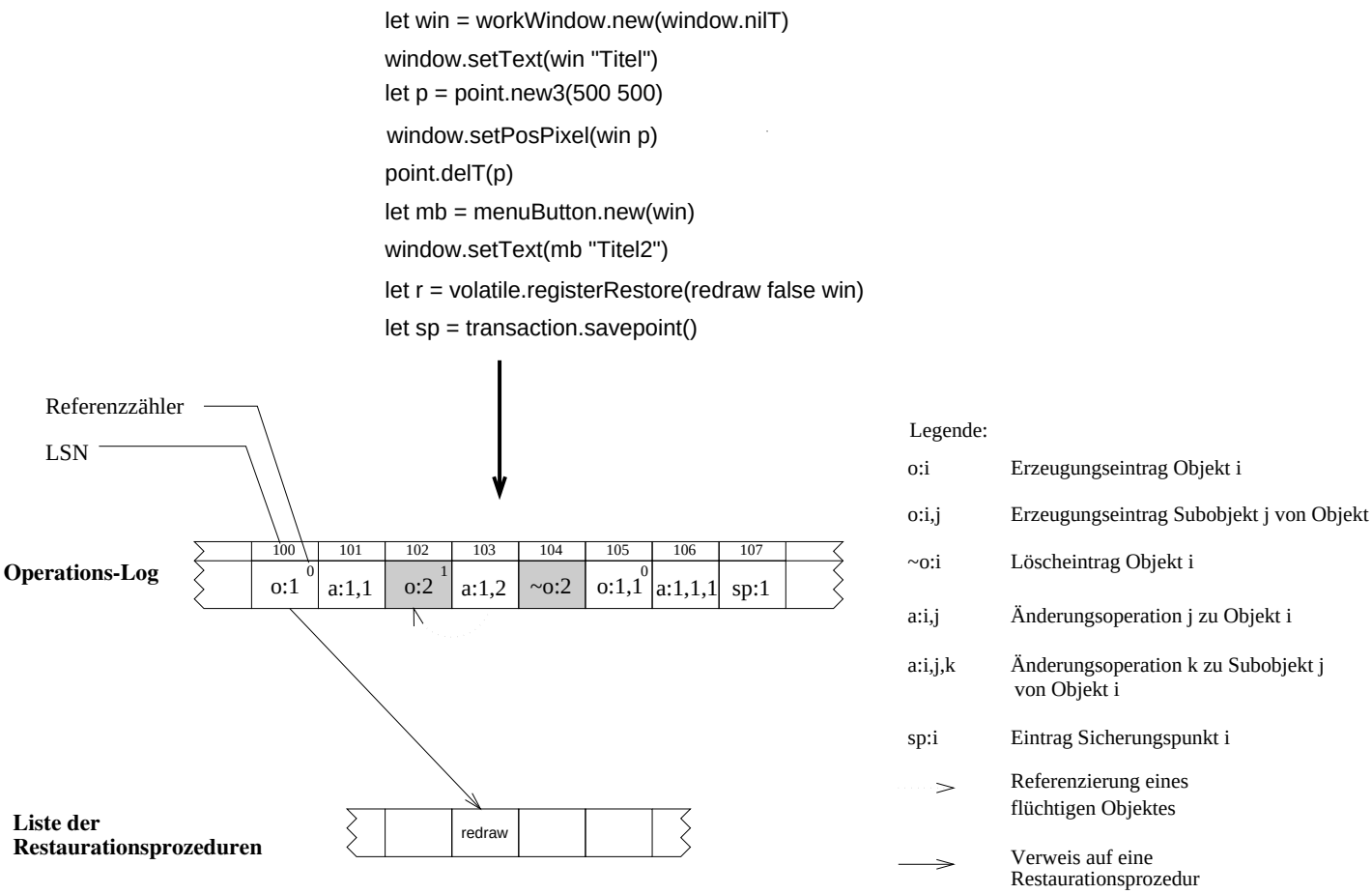

Abbildung 4.4: Code-Beispiel mit Log-Ausschnitt

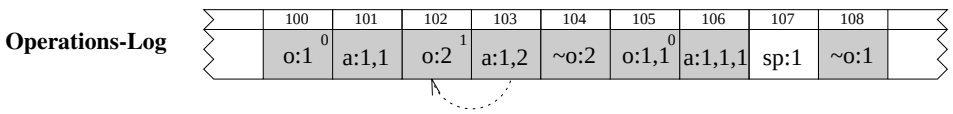

Abbildung 4.5: Beispiel zur hierarchischen Fortsetzung von Markierungen

werden. Unter Zuhilfenahme der Attribute *subObjs* und *ops* des Eintrages des gelöschten Objektes ist dies einfach durchführbar. Abbildung 4.5 zeigt den Zustand des Logs aus dem vorigen Beispiel, nachdem das Fenster *win* gelöscht wurde. Ist eines der Subobjekte unabhängig zu löschen, wird für dieses nicht nur der Eintrag als ungültig markiert, sondern es wird auch die darin verzeichnete Löschfunktion aufgerufen.

Um die gesetzten Synchronisationspunkte intern zu unterscheiden, werden diese aufsteigend nummeriert, beginnend mit der 0 für den letzten Commit-Punkt. Beim Setzen eines Synchronisationspunktes wird ein neuer Eintrag ans Log angehängt, in dem die interne Nummer des Synchronisationspunktes verzeichnet ist. Dadurch lassen sich auch diese Einträge unterscheiden und so als Zielmarke beim Zurücksetzen verwenden. Beim Beenden der Transaktion durch ein Commit wird der Zähler wieder auf 0 zurückgesetzt und vor dem Anhängen eines neuen Eintrags auch alle bestehenden Einträge für Synchronisationspunkt-Einträge aus dem Log

entfernt. Auf diese Weise geben die im Log vorhandenen Einträge immer über die Menge aller tatsächlich erreichbaren Synchronisationspunkte Auskunft.

4.5.3 Algorithmen für das Zurücksetzen

Das Zurücksetzen des externen flüchtigen Zustandes wird vom Modul *transaction* durch Aufruf der Funktion *volatile.rollbackTo* angestoßen, der die Nummer des anvisierten Synchronisationspunktes (im folgenden Zielpunkt genannt) als Parameter übergeben wird. Um den externen flüchtigen Zustand zum Zeitpunkt des Zielpunktes vollständig wiederherzustellen, müssen im Prinzip alle danach ausgeführten Operationen wieder rückgängig gemacht werden. Als Basis dessen dient die Log-Struktur, in der alle zu kompensierenden und zu wiederholenden Operationen verzeichnet sind.

Das Verfahren, mit dem die nach dem Zielpunkt ausgeführten Erzeugungs- und Änderungsoperationen rückgängig gemacht werden, ist offensichtlich. Die zugehörigen Einträge folgen dem Eintrag des Zielpunktes und haben die kompensierende Operation verzeichnet, die dann nur noch ausgeführt werden muß. Das fehlerhafte Kompensieren einer bereits kompensierten Operation, z.B. indem versucht wird, ein bereits gelöschtes Objekt nocheinmal zu löschen, läßt sich vermeiden, da die Einträge der kompensierten Operationen als ungültig markiert sind. Zusammengefäßt lautet das Verfahren so: bei einem sequentiellen Durchlauf des Logs vom Ende bis zum Eintrag des Zielpunktes werden alle gültigen Erzeugungs- und Änderungseinträge kompensiert.[10]

Wurden nach dem Zielpunkt Objekte gelöscht, muß etwas anders verfahren werden, denn die Objektlöschung ist nicht kompensierbar, sondern selbst eine Kompensationsoperation. Um das Löschen eines Objektes rückgängig zu machen, werden alle zu dem Objekt gehörenden Log-Einträge (d.h. auch die der Änderungsoperationen und Subobjekte), beginnend bei dem Erzeugungseintrag, chronologisch bis *hin* zum Eintrag des Zielpunktes wiederholt.[11] Die gelöschten Objekte lassen sich über die nach dem Eintrag des Zielpunktes verzeichneten Löscheinträge herausfinden, denn diese verweisen auf den Erzeugungseintrag des gelöschten Objektes.

Um alle gelöschten Objekte zu ermitteln ist eine Analyse des Log-Abschnittes erforderlich, der durch den Eintrag des Zielpunktes und das Log-Ende begrenzt wird. Diese Analyse wird zusammen mit der Kompensation der noch gültigen Einträge in einer ersten Phase des Zurücksetzens durchgeführt, in der zweiten Phase werden die *vor* dem Eintrag des Zielpunktes liegenden ungültigen Einträge wiederholt. Der Gesamtablauf des Zurücksetzens ist in Abbildung 4.6 graphisch dargestellt. Bei der ersten Phase wird das Log vom Ende bis zum Eintrag des Zielpunktes umgekehrt chronologisch gelesen, parallel dazu werden die noch gültigen Einträge kompensiert und anhand der Löscheinträge die zu wiederholenden Einträge ermittelt. In der zweiten Phase wird das Log chronologisch von einem Starteintrag bis zum Eintrag des Zielpunktes durchlaufen und dabei die relevanten Einträge wiederholt. Bei welchem Eintrag gestartet wird und welche Einträge für die Wiederholung relevant sind wird durch die vorangegangene Analyse ermittelt.

[10] Auch hier wurde eine verkürzte Sprechweise gewählt. Korrekterweise müßte man sagen, daß die in den gültigen Erzeugungs- und Änderungseinträge verzeichneten kompensierenden Operationen ausgeführt werden.

[11] Die ausführliche Sprechweise müßte lauten: die in den Einträgen verzeichneten *create*- oder *redo*-Funktionen werden wiederholt.

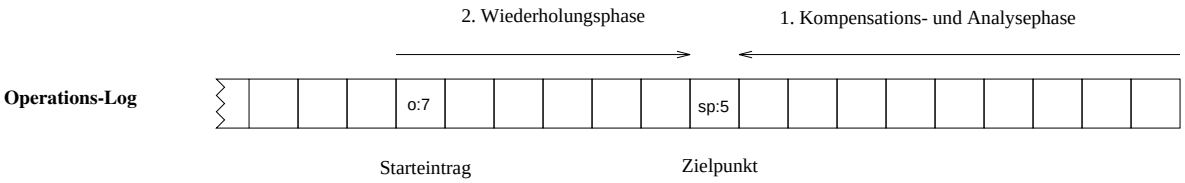

Abbildung 4.6: Die Phasen des Zurücksetzens

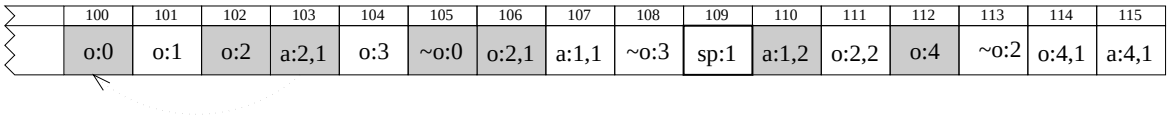

Abbildung 4.7: Beispielszenario für das Zurücksetzen

Bei der Durchführung der Kompensationen ergeben sich aus der hierarchischen Strukturierung der Objekte und allgemein der Semantik der Löschoperation noch Optimierungsmöglichkeiten. Muß beispielsweise ein Änderungseintrag kompensiert werden, könnte das auch durch Löschen des zugehörigen Objektes erfolgen. Wurde das Objekt nach dem Eintrag des Zielpunktes erzeugt, steht fest, daß es auch im Rahmen der Kompensationsphase gelöscht werden muß und es kann von der expliziten Kompensation der zugehörigen Änderungsoperationen abgesehen werden. Das gleiche Verfahren wird für Erzeugungseinträge für Subobjekte angewendet, die kompensiert werden müssen: muß auch das Vaterobjekt gelöscht werden, weil dessen Eintrag hinter dem Zielpunkt liegt, wird das Subobjekt nicht explizit gelöscht. Unabhängig zu löschende Subobjekte bilden die einzige Ausnahme.

Das Vorgehen ist in Abbildung 4.7 verdeutlicht. Diese Abbildung zeigt ein Log-Szenario, bei dem auf den Sicherungspunkt 1 (LSN 109) zurückgesetzt werden soll. Die explizit zu kompensierenden Einträge nach dem Eintrag des Zielpunktes sind grau unterlegt. Bei den Einträgen mit den LSNs 114 und 115 handelt es sich um gültige Einträge, die nicht explizit, sondern implizit mit dem Eintrag 112 kompensiert werden. Demhingegen muß Eintrag 110 explizit kompensiert werden, weil der zugehörige Objekteintrag 101 vor dem Eintrag des Zielpunktes liegt und folglich nicht kompensiert wird. Der Eintrag 111 gehört zu Objekt 2, das gelöscht wurde; folglich ist 111 ungültig und wird deshalb auch nicht kompensiert.

Bisher wurde noch nicht motiviert, warum die Kompensationsphase in umgekehrt chronologischer Log-Reihenfolge durchgeführt wird. Eine kompensierenden Operation kann Parameterobjekte erfordern, wie dies z.B. bei der Änderungsoperation *window.setTitle* in dem in Abschnitt 4.4 vorgestellten Beispiel der Fall war. Die kompensierende Operation hatte einen Wert vom Typ *SVString.T* als Parameter, wobei es sich ebenfalls um ein externes flüchtiges Objekt handelte. Dieses Objekt, der alte Fenstertitel, wurde vor dem Eintragen der Änderungsoperation erzeugt, damit lag auch dessen Erzeugungseintrag vor dem Änderungseintrag. Führte man die Kompensationsphase in chronologischer Reihenfolge durch, würde das SVString-Parameterobjekt *vor* der Kompensation von *setTitle* gelöscht werden und die Kompensation würde fehlschlagen.

Bei der zeitgleich zur Kompensation durchgeführten Analyse werden all jene ungültigen Einträge vor dem Zielpunkt markiert, die zu gelöschten Objekten gehören und in der zweiten Phase wiederholt werden müssen. Dabei handelt es sich nicht nur um die Erzeugungseinträge der gelöschten Objekte mit ihren Änderungseinträgen und den Einträgen der Subob-

jekte. Auch referenzierte Objekte müssen wiederhergestellt werden, wenn diese mittlerweile gelöscht sind. Ansonsten ist nicht sichergestellt, daß die primär zu wiederholenden Operationen ausgeführt werden können. Es werden also auch die Einträge der referenzierten Objekte markiert. Die Analyse wird anhand der vorgefundenen Löscheinträge vorgenommen, die auf den Erzeugungseintrag des gelöschten Objektes verweisen. Von diesem Eintrag aus sind alle zu markierenden Einträge für Änderungsoperatioinen und Subobjekte entweder direkt oder indirekt über die Erzeugungseinträge der Subobjekte erreichbar. Die Parameterobjekte von verzeichneten Operationen sind ebenfalls über die Log-Einträge erreichbar. Von den gelöschten Parameterobjekten werden alle zugehörigen Einträge rekursiv markiert: Hat ein solcher Eintrag weitere gelöschte Parameterobjekte werden auch diese markiert. Das Markierungsverfahren wird für jeden Löscheintrag einzeln durchgeführt.

Als Beispiel soll wieder Abbildung 4.7 dienen, wo die Löschung von Objekt 2, ersichtlich durch Eintrag 113, rückgängig gemacht werden muß. Neben dem Erzeugungseintrag 102 müssen auch die Einträge 103 und 106 wiederholt werden, weil sie implizit durch die Löschung von Objekt 2 kompensiert wurden. Um Eintrag 103 zu wiederholen ist das Parameterobjekt 0 erforderlich, was aber auch bereits in Eintrag 105 gelöscht wurde. Diese Löschung muß vollständig kompensiert werden, weshalb auch die Einträge 100 und 105 markiert werden. Hätte Eintrag 100 auch Parameterobjekte, die gelöscht sind, müssten auch deren Einträge markiert werden.

In der abschließenden zweiten Phase werden alle markierten Einträge in einem chronologischen Log-Durchlauf, angefangen beim frühesten markierten Eintrag (in Abbildung 4.7 ist das Eintrag 100), wiederholt. Auf diese Weise werden auch die Parameterobjekte für zu wiederholende Operationen neu erzeugt[12], wie im Beispiel Objekt 0, und auch wieder gelöscht, wenn das im Log so verzeichnet ist.

Das geschilderte zweiphasige Verfahren zum Zurücksetzen des externen flüchtigen Zustands entspricht somit den in Abschnitt 4.2 aufgestellten Effizienzkriterien:

- Hochfrequenten Operationen werden durch Maßnahmen des Anwendungsprogrammes nicht im Log erfaßt und damit auch nicht kompensiert oder wiederholt.
- Es werden nur Operationen kompensiert, die nach dem Zielpunkt ausgeführt wurden; somit müssen keine kompensierten Operationen in der zweiten Phase wiederholt werden. Ferner wird auch die Möglichkeiten zur impliziten Kompensation, die sich aus der Semantik der Löschoperationen ergeben, ausgenutzt, so daß nur eine minimale Anzahl von Operationen in der ersten Phase ausgeführt wird.
- Die Wiederholung wird nur für durch Objektlöschungen nach dem Zielpunkt kompensierte Einträge durchgeführt. Um die Ausführbarkeit der Operationen zu gewährleisten, werden auch die Einträge der Parameterobjekte in die Wiederholung eingeschlossen. Insgesamt wird eine minimale, aber hinreichende Menge von Einträgen in der zweiten Phase wiederholt.

Die beiden Phasen des Zurücksetzens des externen Zustandes werden vollständig vor dem Zurücksetzen des Tycoon-Objektspeichers durchgeführt. Für die erste Phase ist dies erforderlich, weil der vor ihr gelesene Log-Abschnitt nach dem Zielpunkt erzeugt wurde und deshalb

[12] Da alle zu einem Parameterobjekt gehörenden Einträge markiert wurden, werden auch alle Änderungsoperationen für das Parameterobjekt wiederholt und auch dessen Subobjekte neu erzeugt. Dies ist auch erforderlich, weil der Zustand der Parameterobjekte vollständig wiederhergestellt werden muß.

nach dem Zurücksetzen des Stores nicht mehr zur Verfügung stünde. Die zweite Phase wiederum ist auf die in der ersten Phase gesetzten Markierungen der Einträge angewiesen, die ebenfalls nach dem Zurücksetzen des Stores rückgängig gemacht worden wären. In Abschnitt 4.5.1 wurde beschrieben, wie die Zeigerwerte der flüchtigen Objekte außerhalb von Tycoon gespeichert werden. Die Notwendigkeit dazu ergibt sich aus dem Verfahren zum Zurücksetzen und der Tatsache, daß auch die zweite Phase vor dem Zurücksetzen des Stores durchgeführt wird. Werden in der zweiten Phase des Zurücksetzens Erzeugungsoperationen wiederholt, wird von diesen auch ein neuer Zeigerwert zurückgeliefert. Speichert man den Zeigerwert unmittelbar im Erzeugungseintrag, würde diese Modifikation durch das Zurücksetzen des Stores verlorengehen und stattdessen stünde wieder der alte, jetzt ungültige Zeigerwert im Eintrag.

Im Anschluß an das Zurücksetzen des Stores folgt noch die Ausführung der Restaurationsprozeduren. Diese Anordnung ist erforderlich, weil die Restaurationsprozeduren sich auf den Store-Zustand in Form von Tycoon-Variablen beziehen und vor Ausführung der Restaurationsprozeduren der Store im Zielzustand sein muß.

4.5.4 Bereinigungsverfahren für die Log-Struktur

Die Notwendigkeit der Bereinigung der Log-Struktur um überflüssige Einträge wurde bereits in Abschnitt 4.1 erwähnt; die Gründe sind ein ansonsten stetig anwachsender Speicherbedarf und damit einhergehend ein stetig anwachsender Aufwand beim Wiederanlauf.[13] In diesem Abschnitt wird ein Verfahren geschildert, mit dem die überflüssigen Einträge periodisch aus dem Log entfernt werden. Das Verfahren ist an *mark-and-sweep* Garbage-Collection-Algorithmen angelehnt: in einer ersten Phase werden die erforderlichen Log-Einträge markiert, im Anschluß daran werden die nicht markierten Einträge entfernt und die Referenzzähler der verbleibenden Einträge entsprechend gemindert. Die Bereinigung des Logs wird immer vor dem Setzen eines Synchronisationspunktes vorgenommen.

Die eigentliche Aufgabe des Bereinigungsverfahrens ist das Erkennen der erforderlichen Log-Einträge. Um die zu dem Verfahren führenden Überlegungen verständlicher zu machen, wird im ersten Schritt von einem vereinfachten Operationsmodell, welches die Existenz von Subobjekten nicht vorsieht, ausgegangen. Das für dieses vereinfachte Operationsmodell entwickelte Verfahren wird in einem zweiten Schritt um die Berücksichtigung von hierarchischen Objektbeziehungen erweitert.

Ein Log-Eintrag ist offensichtlich dann erforderlich, wenn er für eine eventuell durchzuführende transaktionale Operation noch einmal benötigt wird. Bei den Log-verarbeitenden transaktionalen Operationen handelt es sich um den Wiederanlauf und das Zurücksetzen: beim Wiederanlauf werden Log-Einträge wiederholt, beim Zurücksetzen sowohl kompensiert als auch wiederholt. Da beide Verfahren selektiv arbeiten und nicht alle vorhandenen Log-Einträge verarbeiten müssen,[14] sind auch nicht alle im Log vorhandenen Einträge erforderlich. Ein Log-Eintrag ist also dann erforderlich, wenn die Möglichkeit besteht, daß er kompensiert oder wiederholt werden muß. Für das vereinfachte Operationsmodell ohne Subobjekte lassen sich drei Kriterien aufstellen, von denen mindestens eines für erforderlich Einträge erfüllt sein muß. Diese sind:

[13] Dies sei als Vorgriff auf die Beschreibung des Wiederanlaufverfahrens im nächsten Abschnitt erwähnt.

[14] Die selektive Arbeitsweise des Verfahrens zum Zurücksetzen wurde im letzten Abschnitt deutlich. Das beim Wiederanlauf ebenfalls selektiv vorgegangen wird ergibt sich aus den in Abschnitt 4.2 aufgestellten Effizienzkriterien.

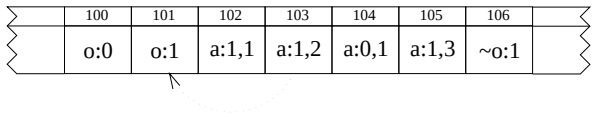

Abbildung 4.8: Beispiel für Kriterium 3 der Log-Bereinigung

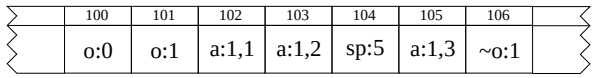

Abbildung 4.9: Beispiel für Kriterium 2 der Log-Bereinigung

Kriterium 1 Der Eintrag ist nicht kompensiert, weswegen er auf jeden Fall beim Wiederanlauf wiederholt werden muß und bei einem zukünftigen Zurücksetzen u. U. zur Kompensation herangezogen wird.

Kriterium 2 Dem Eintrag ist ein gelöschtes Objekt zugeordnet (es handelt sich also entweder um einen Erzeugungseintrag oder einen Änderungseintrag für ein bestimmtes Objekt), das zum Zeitpunkt einer der gültigen Synchronisationspunkte noch existierte. Bei Zurücksetzen auf diesen Synchronisationspunkt würde dieser Eintrag entweder zur Wiederholung oder zur Kompensation herangezogen werden. Da das Objekt zum Zeitpunkt eines Synchronisationspunktes existierte, müssen die Einträge für dieses Objekt einen oder mehrere Synchronisationspunkte umspannen.

Kriterium 3 Das dem Eintrag zugeordnete Objekt wurde gelöscht, aber dessen Erzeugungseintrag wird von einem erforderlichen Log-Eintrag als Parameter referenziert. Um den erforderlichen Log-Eintrag zu wiederholen, müssen auch die Parameterobjekte anhand ihrer Log-Einträge vollständig wiederhergestellt werden können.

Das letzte Kriterium fordert, daß alle Einträge für ein als Parameter referenziertes Objekt zurückbehalten werden, um so den Zustand des Objektes zu jedem Zeitpunkt vollständig wiederherstellen zu können. Dieses Kriterium ist eine Vereinfachung, weil der Zustand tatsächlich nur bis zum Zeitpunkt der letzten Referenz vollständig wiederhergestellt werden muß. Die folgenden Änderungen des Objektes sind ohne Bedeutung, da das Objekt ohnehin gelöscht wurde. Dieser Sachverhalt ist in Abbildung 4.8 dargestellt. Um den Eintrag 103 zu wiederholen, muß Objekt 1 nur bis einschließlich Eintrag 102 wiederhergestellt werden, die Wiederholung des Eintrags 105 wäre nicht erforderlich. Diese Optimierungsmöglichkeit soll aber vernachlässigt werden, weil dadurch die Komplexität des Bereinigungsverfahrens noch weiter steigt.

Kriterium 2 stellt auch eine Vereinfachung dar, wie das Beispiel in Abbildung 4.9 zeigt. Laut diesem Kriterium sind alle Einträge zu Objekt 1 erforderlich, obwohl Eintrag 104 beim Zurücksetzen weder kompensiert noch wiederholt werden muß. Allgemein gesprochen sind Änderungseinträge von gelöschten Objekten nicht erforderlich, wenn sie nach dem letzten Sicherungspunkt, zu dem das Objekt noch existierte, auftreten. Eine Ausnahme tritt auf, wenn das gelöschte Objekt referenziert wird: dann sind wieder alle Änderungseinträge erforderlich. Diese Optimierung von Kriterum 2 soll in den folgenden Verfahren berücksichtigt werden.

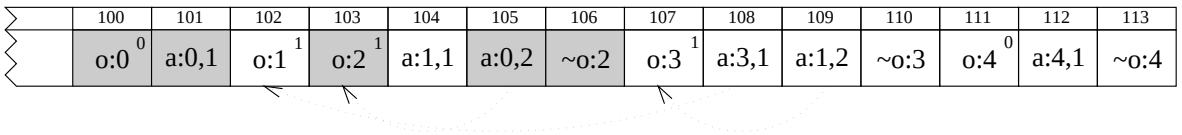

Abbildung 4.10: Markierung bei Commit

Logbereinigung beim Commit

Aus den angegebenen Kriterien läßt sich das Markierungsverfahren für die Log-Bereinigung bei Durchführung eines Commit ableiten:

1. Als Ausgangspunkt sind alle Einträge im Log unmarkiert.
2. Das Log wird vollständig in einer Richtung durchlaufen, dabei werden die gültigen, noch unmarkierten Einträge markiert
3. Handelt es sich bei einem gültigen, unmarkierten Eintrag um eine Objekterzeugung werden zusätzlich:
   a) alle dazugehörigen Änderungseinträge markiert
   b) die Einträge der vom Erzeugungs- und von den Änderungseinträgen referenzierten Objekte markiert, sofern diese noch unmarkiert sind. Diese Markierung wird rekursiv fortgesetzt, es werden also auch die Einträge der Parameterobjekte der referenzierten Objekte markiert usw.

Die durch die Markierung der Parameterobjekteinträge eingeführte Rekursion wird nicht unendlich forgesetzt, weil nur unmarkierte Einträge markiert und weiter durchlaufen werden.

Das Markierungsverfahren sei an einem Beispiel demonstriert. Der einzige gültige Erzeugungseintrag in Abbildung 4.10 ist der Eintrag 100, folglich werden auch die Einträge 101 und 105 markiert. Letzterer referenziert das Objekt 2, weshalb auch dessen Einträge, 103 und 106 markiert werden. Alle anderen Objekte sind gelöscht und ihre Erzeugungseinträge auch von keinem erforderlichen Eintrag referenziert, sie werden also nicht markiert. Das Beispiel zeigt auch, warum es nicht ausreicht, nur die Einträge gelöschter unreferenzierter Objekte zu markieren: zwischen den Objekten können ringförmige Referenzen bestehen (wie zwischen Objekt 1 und Objekt 3), wodurch deren Einträge nicht als überflüssig erkannt würden.

Die im Anschluß an die Markierung durchzuführende Bereinigung erfordert einen umgekehrt chronologischen Durchlauf des vollständigen Logs, bei dem die unmarkierten Einträge entfernt werden. Bevor ein Eintrag physikalisch aus der Log-Struktur entfernt wird, müssen die Referenzzähler in allen von dem Eintrag referenzierten Erzeugungseinträgen um 1 verringert werden. Auf diese Weise wird sichergestellt, daß die Referenzzähler immer mit den vorhandenden Referenzen übereinstimmt. Im Gegensatz zur Markierungsphase muß jetzt das Log umgekehrt chronologisch durchlaufen werden, weil sonst ein zu aktualisierender Erzeugungseintrag bereits entfernt sein könnte. Ein chronologischer Durchlauf würde in der Abbildung 4.10 bei Eintrag 108 scheitern, weil der referenzierte Erzeugungseintrag 102 bereits entfernt sein würde.

Das vorgestellte Verfahren für die Log-Bereinigung beim Setzen eines Commit-Punktes hält sich an die aufgestellten Kriterien. Nach Kriterium 1 erforderliche Einträge werden zurückbehalten, weil diese den Ausgangspunkt der Markierungsphase bilden. Nach Kriterium 3 erforderliche Einträge von Parameterobjekten bleiben auch bestehen, weil die Markierung eines erforderlichen Eintrags auf die Einträge referenzierter Objekte ausgeweitet wird. Weil nach dem Setzen eines Commit-Punktes keine weiteren Synchronisationspunkte mehr vorhanden sind, kann das Kriterium 2 ignoriert werden.

Logbereinigung beim Setzen eines Sicherungspunktes

Die Logbereinigung beim Setzen eines Sicherungspunktes ist nahezu identisch mit der beim Setzen eines Commit-Punktes; einzig das Markierungsverfahren muß erweitert werden, um jetzt auch Kriterium 2 zu berücksichtigen. Das Kriterium bezog sich auf gelöschte Objekte, deren Einträge einen oder mehrere Synchronisationspunkte umspannen und besagt, daß deren Einträge nicht entfernt werden dürfen. Bei der Log-Bereinigung zu Sicherungspunkten dürfen somit nur die Einträge temporärer Objekte, also solcher, die zwischen zwei benachbarten Synchronisationspunkten erzeugt und gelöscht wurden, entfernt werden. Es reicht deshalb aus, bei dieser Logbereinigung nur den Log-Ausschnitt bis zum letzten Synchronisationspunkt zu durchlaufen; eine Synchronisationspunkt-übergreifende Log-Bereinigung kann erst mit dem nächsten Commit durchgeführt werden, weil dann die gesetzten Sicherungspunkte wegfallen.

Das im vorigen beschriebene Markierungsverfahren muß für die Logbereinigung bei Sicherungspunkten konkret in zwei Punkten geändert werden. Wie bereits erwähnt, muß die optimierte Fassung des Kriteriums 2 berücksichtigt werden, indem die Markierung von Einträgen auf folgende zwei Fälle ausgedehnt wird:

- Löscheinträge, deren Erzeugungseintrag vor dem letzten Sicherungspunkt liegt
- Änderungseinträge, deren Erzeugungseintrag vor dem letzten Sicherungspunkt liegt und referenziert ist.

Die andere Abweichung betrifft die Tatsache, daß die Bereinigung nicht für das ganze Log durchgeführt wird. Das hat zur Folge, daß die rekursive Markierung von Parameterobjekten nicht über den letzten Synchronisationspunkt hinausgetragen werden muß.

Das Beispiel in Abbildung 4.11 zeigt, wie das erweiterte Markierungsverfahren angewendet wird. Die einzigen gültigen Einträge im Logausschnitt ab Eintrag 104 sind die zu Objekt 4 gehörenden, deswegen werden diese auch markiert. Der Erzeugungseintrag des Parameterobjektes 3 in Eintrag 101 wird nicht markiert, weil er vor dem Eintrag des letzten Synchronisationspunktes, Eintrag 104, liegt. Bei den Einträgen 115 und 116 handelt es sich um Löscheinträge, deren Erzeugungseinträge vor Eintrag 104 liegen, sie werden deshalb auch markiert. Eintrag 106 gehört zu einem referenzierten Erzeugungseintrag vor dem letzten Synchronisationspunkt; er wird deshalb zusammen mit den Einträgen seines Parameterobjektes auch markiert. Nicht markiert bleiben die Einträge für das temporäre und unreferenzierte Objekt 5 sowie die Einträge 108 und 112.

Erweiterung der Logbereinigung auf Subobjekte

Bei den bisher dargestellten Verfahren wurde von einem vereinfachten Operationsmodell ausgegangen, bei dem Objekte nicht hierarchisch strukturiert waren. Um bei den Bereinigungs-

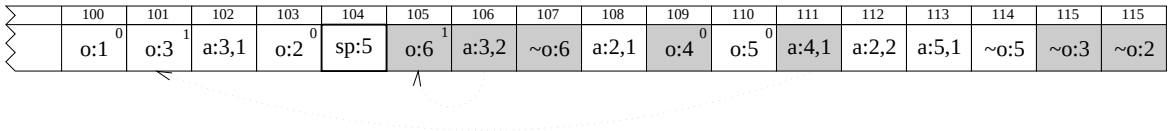

Abbildung 4.11: Markierung bei Sicherungspunkten

verfahren auch hierarchische Objektstrukturierung einzubeziehen, muß berücksichtigt werden, welche Anforderungen diese Möglichkeiten in bezug auf die Existenz von Parameterobjekten stellen und inwiefern der Zustand durch die Existenz von Subobjekten geändert wird. Beide Aspekte wirken sich auf die Menge der erforderlichen Einträge aus.

Um zu klären, inwiefern sich die Existenz von Subobjekten auf das Bereinigungsverfahren auswirkt, seien einige weitere Festlegungen bezüglich der Semantik von Vater- und Subobjekten getroffen.

1. Ein Subobjekt kann nicht unabhängig von seinem Vaterobjekt existieren, ebensowenig kann es auch nicht ohne die Existenz des Vaterobjektes neu erzeugt werden. Das Vaterobjekt wird folglich wie ein Parameterobjekt bei der Erzeugung des Subobjektes betrachtet. Ein Beispiel dazu war in Abbildung 4.4, Seite 49, zu sehen, wo ein Menüknopf als Subobjekt eines Fensters erzeugt wurde, wobei im Erzeugungsaufruf das Fenster als Parameter dient.

2. Es wird davon ausgegangen, daß eine Referenz eines Parameterobjektes sich automatisch auf dessen Subobjekte ausweitet, so daß diese auch transitiv referenziert werden. Ein solche Semantik findet man z.B. bei Anlegen eines tiefen Kopie eines Objektes. Natürlich trifft das nicht auf jede Operation zu, aber diese konservative Annahme ist notwendig, um zu verhindern, daß Einträge entfernt werden, die für irgendein Objekt relevant sind.

3. Ansonsten sind die Subobjekte eigenständige Objekte, insbesondere beeinflussen ihre Existenz und die für sie ausgeführten Änderungsoperationen nicht den Zustand des Vaterobjektes.

Mit dem Hintergrund dieser semantischen Festlegungen müssen zwei neue Kriterien formuliert werden, die die in Abschnitt 4.2 aufgestellten drei Kriterien zur Festlegung der Erforderlichkeit von Log-Einträgen ergänzen. Diese Kriterien sind:

Kriterium 4 Ein Objekt ist Vater eines von einem erforderlichen Eintrag referenzierten Subobjektes. Dann sind auch die Erzeugungs-, Änderungs- und Löscheinträge des Vaterobjektes erforderlich.

Kriterium 5 Ein Objekt wird von einem erforderlichen Eintrag referenziert. Dann sind auch alle Einträge von (transitiv erreichbaren) Subobjekten erforderlich.

Kriterium 4 reflektiert die eben getroffenen Festlegungen 1 und 3: das Markieren eines Parameterobjektes wird auf die Vorfahren ausgedehnt (Festlegung 1), aber nicht gleichzeitig auf alle Subobjekte der Vorfahren (Festlegung 3). Kriterium 5 reflektiert die Festlegung 2. Durch diese neuen Kriterien werden die alten nicht verändert, weil mit einem erforderlichen

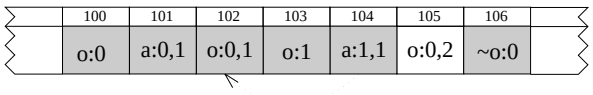

Abbildung 4.12: Markierung von Subobjekten

Vaterobjekt nicht automatisch die existierenden Subobjekte erforderlich sind. Insbesondere sind für existierende, aber unreferenzierte Objekte die Einträge der gelöschten, ebenfalls unreferenzierten Subobjekte nicht erforderlich.

Die neuen Kriterien lassen sich einfach in die vorigen Verfahren zur Log-Bereinigung durch Erweiterung des Markierungsvorgangs integrieren. Um Kriterium 4 zur berücksichtigen, werden bei Markierung des Erzeugungseintrags eines Parameterobjektes auch alle Einträge der Vorfahren markiert. Für Kriterium 5 wird die Markierung eines Parameterobjektes rekursiv an dessen Subobjekte weitergeleitet. Der erweiterte Markierungsvorgang sei an dem in Abbildung 4.12 zu sehenden Beispiel demonstriert. Als einziges noch existierendes Objekt werden die Einträge 103 und 104 von Objekt 1 markiert. Als Parameterobjekt von Eintrag 104 werden auch die Einträge von Subobjekt 1 und damit auch die Einträge des Vaterobjektes 0 markiert. Die Markierung von Objekt 0 wird aber nicht auf alle Subobjekte ausgedehnt, so daß der Erzeugungseintrag von Subobjekt 2 unmarkiert bleibt.

4.5.5 Algorithmen für den Wiederanlauf

Um nach einem Systemfehler, im Regelfall ein Systemabsturz, auch den externen flüchtigen Zustand zum Zeitpunkt des letzen Synchronisationspunktes zu regenerieren, muß für diesen ein Wiederanlauf durchgeführt werden. Dieser externe Wiederanlauf wird, ebenso wie das externe Zurücksetzen, vom das Modul *transaction* durch Aufruf der Funktion *volatile.doRestart* angestoßen.

Im Unterschied zum persistenten Tycoon-Zustand ist der externe flüchtige Zustand nach einem Systemfehler vollständig ausgelöscht und muß deshalb von Beginn der Applikation wieder aufgebaut werden. Das entsprechende Wiederanlaufverfahren besteht deshalb im Kern aus einer chronologischen Wiederholung der im Log verzeichneten Operationen bis hin zum Eintrag des letzten Sychronisationspunktes, gefolgt von einer Ausführung der registrierten Restaurationsprozeduren. Weil zu diesem Zweck die Log-Struktur und die Liste der Restaurationsprozeduren intakt sein müssen, wird zuerst der Store-Wiederanlauf durchgeführt. Dadurch wird der Tycoon-interne Zustand zum letzten Synchronisationspunkt regeniert, so daß der für den externen Wiederanlauf relevante Teil der Log-Struktur auch bei Beginn des externen Wiederanlaufs vorhanden ist.

In Abschnitt 4.2 wurde ein Kriterium für die effiziente Ausführung der Wiederanlaufs gegeben, wonach nur die notwendigen Operationen wiederholt werden sollen, und nicht alle insgesamt verzeichneten. Der genaue Wortlaut des Kriteriums war:

„Zum Wiederanlauf werden nur alle zum Zeitpunkt des letzten Sicherungspunktes noch nicht kompensierten Operationen wiederholt. Damit dies korrekt durchgeführt werden kann, sollen auch die für Parameterobjekte verzeichneten Operationen wiederholt werden."

Die zu wiederholenden Einträge zeichnen sich demnach durch zwei Eigenschaften aus:

1. sie sind nicht kompensiert und demzufolge noch als gültig markiert

2. oder sie gehören zu Parameterobjekten von anderen zu wiederholenden Einträgen

Für die selektive Wiederholung werden zuerst die notwendig zu wiederholenden Einträge markiert und erst anschließend die markierten Einträge in chronologischer Reihenfolge wiederholt. Dabei ist zu beachten, daß die zwei Bedingungen, anhand derer die notwenigen Einträge erkannt werden, genau den für die Log-Bereinigung nach einem Commit angewendet Kriterien entsprechen (vergleiche Seite 53). Das zu diesem Zweck entwickelte Markierungsverfahren wird deshalb auch beim Wiederanlauf zur Kennzeichnung der zu wiederholenden Einträge verwendet.

4.6 Anwendungsgrenzen des Operations-Logging für flüchtige externe Daten

Das Operations-Logging erlaubt eine effiziente Simulation transaktionalen Verhaltens für externe Daten über die Wiederholung und Kompensation der ausgeführten externen Operationen, die dazu in der Log-Struktur verzeichnet werden. Allerdings beruht die Korrektheit der entwickelten transaktionalen Verfahren auf einigen impliziten Annahmen über die Beschaffenheit der Daten und ihren Operationen, die vor dem Einsatz der Verfahren in einem bestimmten Anwendungsfall überprüft werden müssen. Aus der Schnittstelle des für das Logging zuständigen Moduls *volatile* und der Implementation ergeben sich zwei weitere Einschränkungen in der Anwendung des Operations-Loggings.

Das Prinzip, beim Wiederanlauf den flüchtigen externen Zustand vollständig durch Wiederholung der ausgeführten Operationen zu rekonstruieren, basiert auf drei Annahmen, die in den vorangegangenen Abschnitten noch nicht zur Sprache kamen. Als erstes wird vorausgesetzt, daß die Seiteneffekte der Operationen nur von ihnen selbst und den übergebenen Parametern, nicht aber von externen Einflüssen abhängen, die sich außerhalb der Kontrolle von Tycoon befinden. Wird diese Voraussetzung verletzt, kann nicht garantiert werden, daß der ursprüngliche Zustand wiederhergestellt werden kann, weil die externen Parameter sich geändert haben können. Solches Verhalten ist zum Beispiel bei zeitabhängigen Funktionen anzutreffen; der Wiederanlauf würde durch die Operationswiederholung zu einem späteren Zeitpunkt einen neuen Zustand erzeugen, der mit dem ursprünglichen nicht übereinstimmt. Die Existenz externer Einflüsse auf den flüchtigen Zustand beeinträchtigt allgemein die Korrektheit der transaktionalen Algorithmen und es ist Aufgabe des Programmierers, diese Einflüsse auszuschließen.

Die nächste Annahme betrifft die Verwendung der Daten. Da beim Wiederanlauf die (relevanten) im Log verzeichneten Operationen wiederholt werden, muß sichergestellt sein, daß auch alle Operationen im Log verzeichnet sind. Diese Voraussetzung würde bei geteilter Nutzung der Daten zwischen mehreren Prozessen nicht mehr erfüllt sein: Jeder Tycoon-Prozeß würde die von ihm ausgeführten Operationen in seinem lokalen Log verzeichnen, so daß in jedem Log nur ein partielles Bild der vollständigen Operationssequenz vorhanden ist. Eng damit verknüpft ist die dritte Annahme, daß der flüchtige Zustand simultan mit der Unterbrechung des Tycoon-Prozesses zerstört wird, so daß beim Wiederanlauf die vollständige Rekonstruktion des Zustands überhaupt erforderlich ist. Wiesen die flüchtigen Daten ein vom Tycoon-Prozeß unabhängiges Fehlerverhalten auf, beispielsweise weil sie in einem separaten Adreßraum oder sogar auf einem entfernten Rechner liegen, blieben sie auch von einer Unterbrechung des Tycoon-Prozesses unberührt und müßten beim Wiederanlauf nicht rekonstruiert

werden. Zusammenfassend kann also festgestellt werden, daß für den Einsatz der Transaktionsalgorithmen nur private, prozeßlokale flüchtige Daten in Frage kommen.

Eine weitere Einschränkung betrifft die Programmierung der transaktionalen Anpassung einer externen Bibliothek. Diese ist für die Erzeugung der Log-Einträge zuständig und muß bei Aufruf der Funktionen *volatile.create()* und *volatile.log()* die Konvention beachten, daß die Parameter der verzeichneten Funktionen konstant sind. Als Beispiel sei die Funktion *setText()* aus Abschnitt 4.4 noch einmal näher betrachtet:

```
let setText(self :T str :String) :Ok =
  if volatile.loggingEnabled() then
    let oldSvString = ...
    volatile.log(:Ok
      fun() :Ok cSetText(volatile.toWord(self) str)
      fun() :Ok cSetText(volatile.toWord(self) svString.get(oldSvString))
      self
      array oldSvString end)
  else
    cSetText(volatile.toWord(self) str)
  end
```

Würde es sich bei *str* um einen variablen Parameter handeln, kann dieser einen neuen Wert angenommen haben, wenn die *redo*-Funktion z.B. im Rahmen eines Wiederanlaufs ausgeführt wird. Das Resultat wäre, daß das Fenster beim Wiederanlauf einen anderen Titel als den ursprünglich gesetzten erhielte. Dieses Verhalten resultiert aus der Trennung des Wiederanlaufs und des Zurücksetzens in separate Abschnitte für interne und externe Daten. Aufgrund dieser Trennung treten während des Wiederanlaufs und Zurücksetzens Zustände auf, die bei der Vorwärtsverarbeitung nicht vorlagen. Beispielsweise wird beim Wiederanlauf vor der Wiederherstellung des externen Zustandes vollständig der interne Zustand zum Zeitpunkt des letzten Synchronisationspunktes wiederhergestellt. Diejenigen Werte von Tycoon-Variablen, die nur zum Zeitpunkt der Erzeugung der Log-Einträge existierten, sind demnach bei Wiederherstellung des externen Zustandes verloren. Ähnlich verhält es sich beim Zurücksetzen, nur das dabei der externe vor dem internen Zustand zurückgesetzt wird.

Die Einschränkung läßt sich umgehen, indem für die Log-Einträge Kopien der variablen Parameter angelegt werden. Die Erfahrungen mit der StarView-Bibliothek haben allerdings gezeigt, daß die Notwendigkeit dazu sich nur sehr selten ergeben dürfte, weil überlicherweise nur einfach strukturierte konstante Werte aus Tycoon als Parameter an externe Funktionen übergeben werden. Wie im obigen Beispiel der Funktion /sl setText kann durch die Deklaration des Parameters erzwungen werden, daß es sich nicht um einen variablen Parameter handelt.

Die zweite Einschränkung besteht in der vom Modul *volatile* vorgegebenen hierarchischen Objektstrukturierung. Ist diese in den Objekten einer externen Bibliothek nicht anzutreffen, kann diese auch nicht angepaßt werden. Implementiert eine Bibliothek zum Beispiel ringförmig angeordnete Objekte, so daß bei Löschen eines beliebigen Elementes implizit alle Elemente des Rings gelöscht werden, läßt sich dieses Verhalten nicht mit hierarchischen Objekten und den dadurch implizierten Löschabhängigkeiten simulieren. Allgemein gesprochen kann es bei komplexeren Strukturierungen zu Löschabhängigkeiten zwischen Objekten kommen, die sich

nicht auf die von *volatile* implementierten abbilden lassen. Für die derzeit in Tycoon zur Verfügung stehenden externen Bibliotheken ist das allerdings nicht der Fall, deswegen wurde auch kein Weg gesucht, diese Einschränkung zu umgehen. Bei Bedarf könnte das Logging-Verfahren jedoch problemlos erweitert werden.

Der einfachste Ansatz wäre eine Erweiterung des Operationsmodells um die für eine bestimmte Strukturierungsmöglichkeit benötigte Semantik, insbesondere die Löschabhängigkeiten. Nachteilig wäre, daß dafür auch die Schnittstelle von volatile erweitert werden müßte, und das diese Erweiterung für jedes neue Objektmodell erforderlich wäre. Eine alternative Möglichkeit besteht in der Auslagerung eines bestimmten Objektmodells in ein eigenes Modul, so daß *volatile* vollkommen neutral bezüglich der Objektstrukturierung gehalten werden kann. Die Semantik eines neu hinzuzunehmenden Objektmodells würde in einem eigenen Modul mit standardisierter Schnittstelle implementiert werden, das die auftretenden Löschabhängigkeiten verwaltet. Das Anpassungsmodul einer externen Bibliothek müßte dann bei Erzeugung der Log-Einträge gegenüber *volatile* spezifizieren, um welches Objektmodell es sich handelt. Das Modul *volatile* würde die modellspezifischen Module aufrufen, um sich bei Bedarf über die bestehenden Löschabhängigkeiten zu informieren. Die in Abschnitt 4.1 vorgestellte Architektur des Operations-Loggings müßte dann um die modellspezifischen Module erweitert werden. Dieser Ansatz hat gegenüber der im vorigen Absatz erwähnten direkten Erweiterung von *volatile* den Vorteil, daß beliebige Objektmodelle auf einfache und vor allen Dingen standardisierte Weise einbezogen werden können und dadurch insbesondere auch keine Veränderung der *volatile*-Schnittstelle erforderlich ist.

5. Sicherungspunkte für transaktionale externe Dienste

In diesem Kapitel wird beschrieben, wie die letzte Ressourcenklasse, nämlich die der transaktionalen externen Daten, in das um Sicherungspunkte erweiterte Transaktionskonzept von Tycoon integriert wird. Die dabei angewendete Implementationstechnik—Abbildung von Sicherungspunkten auf externe abgeschlossene Transaktionen und Zurücksetzen durch kompensierende Operationen—wurde bereits in Abschnitt 2.3 skizziert und wurde auch für die Simulation von Sicherungspunkten mit dem Tycoon-Objektspeicher herangezogen. Umgesetzt wird diese Technik wiederum mit Hilfe einer Log-Struktur, die kompensierende Operationen verzeichnet und damit den Ausgangspunkt für das Zurücksetzen darstellt. Um zu gewährleisten, daß beim Setzen eines Sicherungspunktes in Tycoon die erfolgreiche Beendigung der externen Transaktion zuverlässig im Log verzeichnet wird, muß zwischen Tycoon und den beteiligten externen Servern ein 2-Phasen-Commit (2PC) durchgeführt werden. Die Durchführung eines 2PC-Protokolls setzt eine Transaktionsumgebung voraus, die u.a. die Elemente und den Ablauf des Protokolls genau spezifiziert. Als Grundlage der Transaktionsumgebung wurde das X/Open DTP-Modell ausgewählt, durch das die Portabilität und Interoperabilität von Tycoon sichergestellt wird. Dieses Modell stellt einen Standard im Bereich verteilter Transaktionsverarbeitung dar, der es erlaubt, Tycoon mit beliebigen, X/Open-konformen transaktionalen Diensten einzusetzen.

Der nächste Abschnitt beschreibt kurz das angewendete Logging-Verfahren und verdeutlicht anhand eines Beispiels die Notwendigkeit der Durchführung eines 2PC-Protokolls bei der Einbeziehung externer transaktionaler Ressourcen. Im Anschluß daran wird das X/Open-Modell mit seinen Komponenten und den standardisierten Schnittstellen vorgestellt. Darauf aufbauend enthält der folgende Abschnitt eine Übersicht der architekturellen Integration von Tycoon in das X/Open-Modell. Den Abschluß des Kapitels bildet die Darstellung der Algorithmen zum Zurücksetzen, Wiederanlauf und Setzen von Synchronisationspunkten unter Verwendung der relevanten X/Open-Schnittstellen.

5.1 Integration externer transaktionaler Dienste

Zur Simulation persistenter Sicherungspunkte durch die flachen Transaktionen der externen transaktionalen Dienste wird, genau wie bei der Erweiterung des TSP, jeder einzelne Sicherungspunkt auf eine abgeschlossene externe Transaktion abgebildet. Dadurch ergibt sich auch wieder die Unterscheidung in Tycoon-Transaktionen und externe Transaktionen, die sich genau wie in Abbildung 2.1 (Seite 16) zueinander verhalten. Das Zurücksetzen wird durch

Ausführung kompensierender Operationen erreicht, die ihrerseits auch in eine externe Transaktion eingefaßt sind. Um jederzeit in der Lage zu sein, auf einen beliebigen noch erreichbaren Synchronisationspunkt zurücksetzen zu können, müssen die kompensierenden Operationen persistent in einer Log-Struktur verzeichnet werden. Die Grundlage der Integration externer transaktionaler Daten bildet also ein Operations-Log, aber anders als bei flüchtigen Daten wird der Zustand der externen transaktionalen Ressourcen zu jedem Synchronsationspunkt durch Beendigung der externen Transaktion persistent gemacht. Aus diesem Grund ist beim Wiederanlauf für externe transaktionale Ressourcen keine Operationswiederholung erforderlich, so daß es genügt, nur die kompensierenden Operationen im Log zu verzeichnen. Wie auch für Tycoon-interne Zustände reicht also ein reines Undo-Log aus.

Ebenfalls können die im Operations-Log angesammelten kompensierenden Operationen mit jedem Tycoon-Commit entfernt werden, weil die bis dahin gesetzten Sicherungspunkte durch das Transaktionsende unerreichbar werden. Dadurch wird die Log-Struktur ständig verkürzt und ein aufwendiges Bereinigungsverfahren, wie im vorigem Kapitel für flüchtige Daten dargestellt, wird überflüssig. Damit entfällt aber auch die Notwendigkeit der Ausrichtung des Operationsmodells auf die Log-Bereinigung; die Operationen für externe transaktionale Daten werden im Log daher nicht weiter nach Erzeugung, Änderung und Löschung unterschieden.

Das für externe transaktionale Daten verwendete Logging-Verfahren ist demnach ein Hybridansatz, der Elemente aus den Verfahren für Tycoon-intern und flüchtige externe Daten in sich vereint. Das Verfahren hat folgende Charakteristika:

- Die Basis bildet ein zentrales Operations-Log, das jedoch nur Undo-Informationen in Form von kompensierenden Operationen aufzeichnet. Neben den Einträgen für kompensierende Operationen finden sich auch Einträge für Sicherungspunkte, die wieder beim Zurücksetzen als Zielmarken dienen.

- Wie auch bei flüchtigen externen Daten werden die Einträge mit den kompensierenden Operationen von den Anpassungsmodulen der externen Bibliotheken direkt bei Ausführung der externen Operationen geschrieben.

- Anders als bei flüchtigen externen Daten kann die Log-Struktur mit jedem Tycoon-Commit abgeschnitten werden und braucht nicht durch wie ein in Abschnitt 4.5.4 beschriebenes Spezialverfahren bereinigt zu werden, so daß die im Log verzeichneten kompensierenden Operationen nicht weiter klassifiziert werden müssen. Weil beim Wiederanlauf die Wiederholung der ursprünglich ausgeführten Operationen entfällt, muß der Applikationsprogrammierer nicht in die Logging-Vorgänge eingreifen, indem er das Logging bei häufig wiederholten Operationen ausblendet.

- Ähnlich wie für Tycoon-internen Zustand erfolgt das Zurücksetzen der externen transaktionalen Daten durch einfache umgekehrt chronologische Ausführung der im Log verzeichneten kompensierenden Operationen. Aufgrund der fehlenden Operations-Klassifizierung kann auf das komplexe zweiphasige Verfahren zum Zurücksetzen des externen flüchtigen Zustandes (siehe Abschnitt 4.5.3) verzichtet werden.

Die bisher etablierte Architektur der Tycoon-Anwendungsebene wird um das Modul *transServers* wie in Abbildung 5.1 erweitert. Das neue Modul ist für die Verwaltung des Operations-Log für transaktionale externe Daten zuständig und hat die in Abbildung 5.2 dargestellte Schnittstelle.

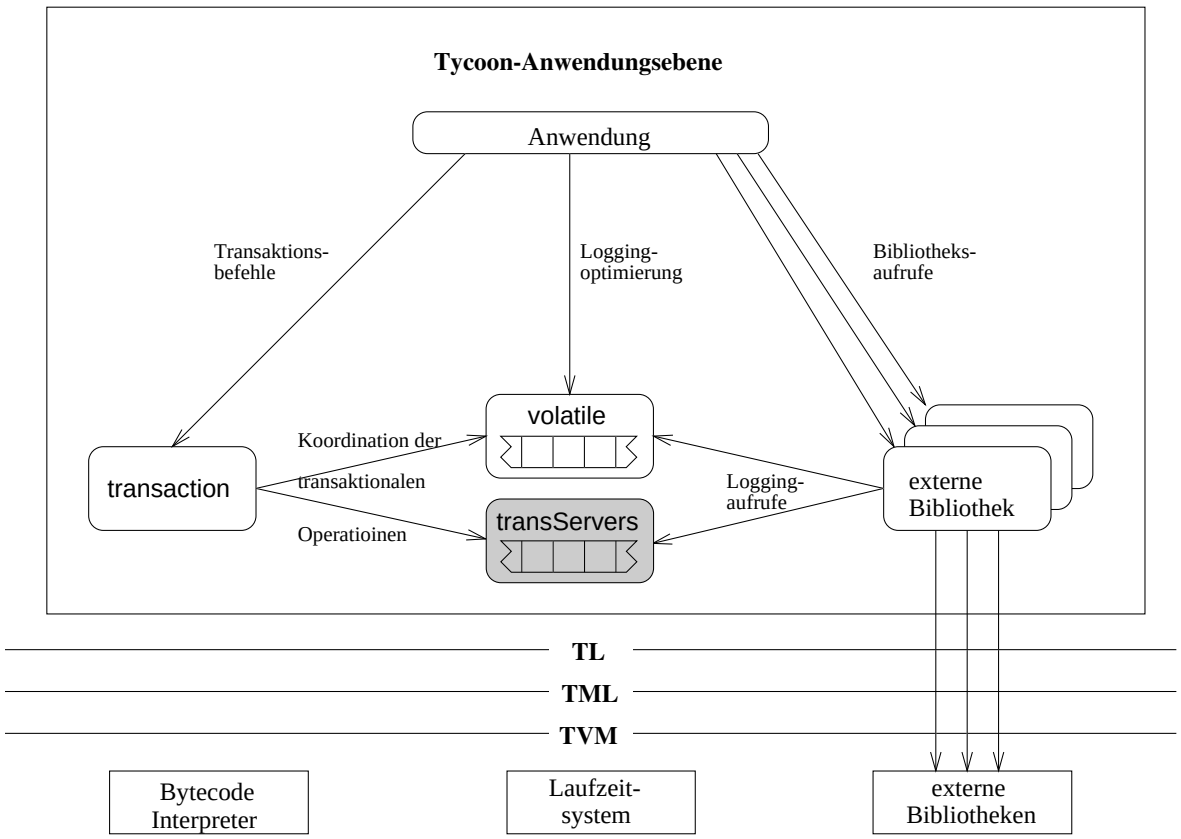

Abbildung 5.1: Die Architektur des Operations-Logging für transaktionale externe Daten

Die Funktionen *doSavepoint()*, *doCommit()* und *doRollbackTo()* haben dieselbe Bedeutung wie die namensgleichen Funktionen des Moduls *volatile* und werden auch zum gleichen Zeit-

```
interface TransServers

export

  log(undo() :Ok) :Ok

  doSavepoint() :Ok

  doCommit() :Ok

  doRollbackTo(syncPoint :Int) :Ok

end;
```

Abbildung 5.2: Definitionsmodul TransServers

punkt vom Modul *transaction* aufgerufen. Die Anpassungsmodule der externen Bibliothken benutzen die Funktion *log*, um die kompensierenden Operationen zu verzeichnen. Zur Illustration dieser Funktion sei das Beispiel aus dem vorigen Kapitel erweitert: neben der Anzeige der Meßwerte sollen diese jetzt auch in einer SQL-Datenbank gespeichert werden. Zu diesem Zweck wird die Anzeigeschleife um den Aufruf *messwert.speichern(m)* ergänzt. Diese Funktion hat die Aufgabe, den Parameterwert in einer Tabelle „Messwerte“ einzufügen und gleichzeitig eine diese Einfügeoperation kompensierende Löschoperation im Log zu verzeichnen:

```
let messwerte.speichern(wert :Int) :Ok =
begin
  let currentTime =
  let insertStmt = string.concat("INSERT INTO Messwerte VALUES("
    currentTime.toString ", " intOp.Format(m) ")")
  let deleteStmt = string.concat("DELETE Messwerte WHERE timestamp = "
    currentTime.toString)
  sql.execute(insertStmt)
  transServers.log(fun() sql.execute(deleteStmt))
end
```

In diesem Beispiel wird deutlich, daß die kompensierende Operation die Aufgabe hat, genau den Seiteneffekt der gerade ausgeführten Einfügeperation zu kompensieren; dies wird durch Verwendung des Zeitstempels als eindeutiges Attribut erreicht.

Um inkonsistente Zustände zu vermeiden, muß der Zustand von Tycoon synchron mit der Beendigung der externen Transaktion gesichert werden. Die durch eine nicht-synchrone Durchführung des Store-Commit und des externen Commit verursachten Inkonsistenzen lassen sich leicht an einem einfachen Beispiel eines Einzahlungsvorganges erkennen. Dieser Einzahlungsvorgang wird von einem Sachbearbeiter über die Bildschirmmaske einer Tycoon-Applikation angestoßen und durch Anzeige einer Statusmeldung und Setzen eines Sicherungspunktes abgeschlossen. Applikations-intern wird für die Einzahlung die Funktion *gutschreiben()* aufgerufen, die den entsprechenden Geldbetrag in einer Datenbanktabelle verbucht. Stürzt das System nach dem Store-Commit aber vor Beendigung der externen Transaktion ab, sieht es für den Sachbearbeiter so aus, als ob die Zahlung in der Datenbank verzeichnet sei. Stürzt das System nach Beendigung der externen Transaktion aber vor dem Store-Commit ab, wird der Sachbearbeiter nach dem Wiederanlauf die Einzahlung fälschlich wiederholen und dem Kunden zuviel gutschreiben.

Die beiden einzelnen Transaktionen, die im Tycoon-Store und die in der Datenbank, müssen demnach als eine unteilbare, zusammenhängende Transaktion durchgeführt werden. Nach deren Ausgang müssen entweder beide Speicher die Transaktion beendet haben oder keiner von ihnen. Für diesen Sachverhalt hat sich die Bezeichnung *verteilte Transaktion* eingebürgert; die dafür üblicherweise eingesetzte Technik ist das zweiphasige Commit (2PC) ([Gray 78; Lampson, Sturgis 76; Lindsay 79]).

Beim 2PC wird dem Commit eine von einem zentralen Koordinator initiierte Wahlphase (*Prepare*-Phase) vorangestellt, in der jeder an der verteilten Transaktion teilnehmende persistente Speicher darüber Auskunft gibt, ob die Transaktion von ihm erfolgreich beendet werden kann. Die Antworten werden an den Koordinator geschickt, der daraufhin bei durchgehend positiven Antworten den Teilnehmern in der zweiten Phase die erfolgreiche Beendigung mitteilt. Bei mindestens einer negativen Antwort muß allen übrigen Teilnehmern der Abbruch der

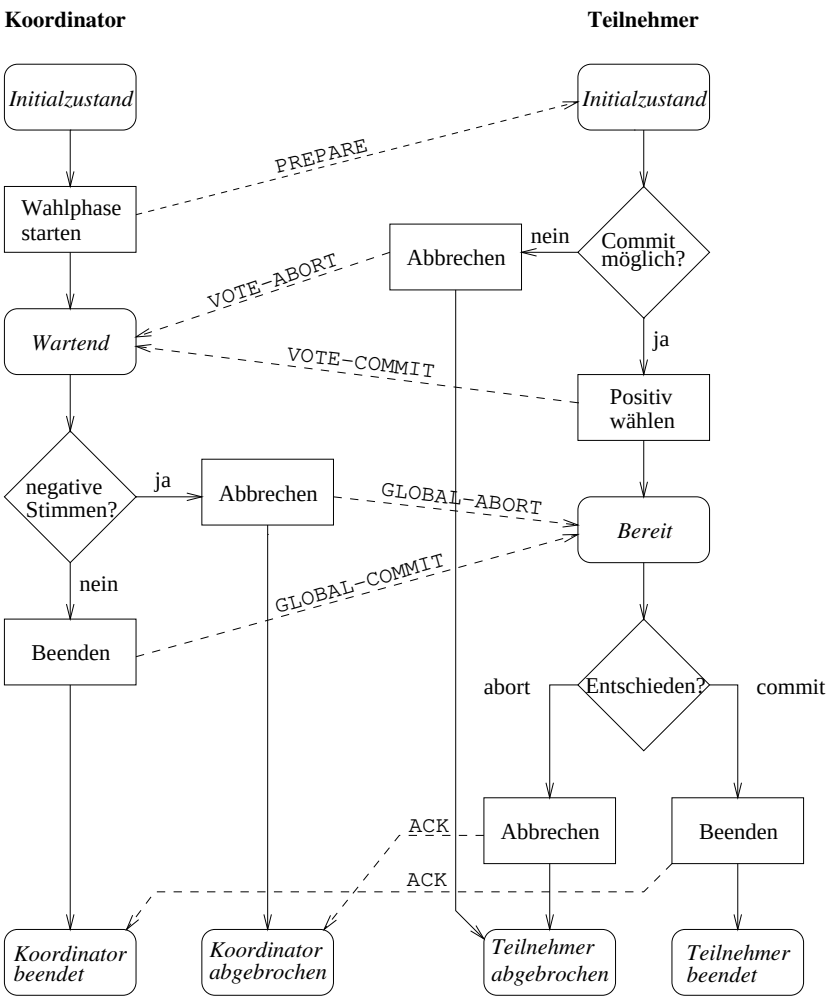

Abbildung 5.3: Zustände und Nachrichten beim 2PC

Transaktion mitgeteilt werden. Die Abbildung 5.3 (angelehnt an eine Abbildung aus [Özsu, Valduriez 91]) zeigt die eingenommenen Zustände des Koordinators und der Teilnehmer und die zwischen ihnen ausgetauschten Nachrichten. Durch dieses Protokoll läßt sich selbst in komplexen Fehlersituationen garantieren, daß alle Teilnehmer einen gemeinsamen Ausgang der Transaktion erreichen. Das geschilderte Verfahren entspricht dem zentralisierten 2PC und ist z.B. in [Bernstein et al. 87] noch genauer beschrieben, daneben existieren auch noch andere Varianten ([Mohan, Lindsay 83]) sowie ein dreiphasiges Commit-Protokoll (Ref Skeen).

Um ein 2PC zwischen mehreren persistenten Speichern durchzuführen, sind gewissen Vorbedingungen zu erfüllen. Einmal muß eine Koordinatorkomponente, auch vielfach Transaktions-Manager (TM) genannt, vorhanden sein. Zum anderen müssen sich aber auch die Teilnehmer untereinander auf einen Protokollablauf und das Format der auszutauschenden Nachrichten geeinigt haben. Als Grundlage für die verteilte Transaktionsverarbeitung unter Tycoon wurde das X/Open-Modell ausgewählt, welches die notwendige Infrastruktur in einer standardisierten Umgebung bereitstellt. Innerhalb des X/Open-Modells werden die für die verteilte Transaktionsverarbeitung notwendigen Komponenten und insbesondere deren Programmierschnittstellen (*Application Programming Interfaces*, API) standardisiert, ebenso wie die zwi-

schen den Komponenten ausgetauschten Nachrichten.

Durch Verwendung eines Standards als Basis für eine verteilte Transaktionsverarbeitung ergeben sich einige Vorteile gegenüber dem Einsatz einer herstellerspezifischen Umgebung wie z.B. CICS, Tuxedo oder ACMS. Zum einen führt die Standardisierung der Komponentenschnittstellen zur Portabilität der Applikationsprogramme, da diese nun nicht mehr für die speziellen Umgebungen der einzelnen Hersteller angepaßt werden müssen. Durch die Standardisierung der Interaktionen erreicht man eine Interoperabilität von Komponenten verschiedener Hersteller, so daß Komponenten wie Transaktionsmanager und Datenbanksysteme beliebiger Hersteller gemeinsam an verteilten Transaktionen teilnehmen können. Die Interoperabilität bietet dem Anwender ein erhöhtes Maß an Flexibilität, weil er nicht mehr an die Produktpalette eines einzelnen Herstellers gebunden ist und seine verteilte Transaktionsumgebung komponentenweise zusammenstellen kann. Auch die Hersteller profitieren von von der Standardisierung auf Schnittstellen- und Protokollebene. Durch die standardisierten Schnittstellen werden auch ihre Produkte, wie z.B. Transaktionsmanager, von Hardware- und Betriebssystemen unabhängig; durch die Interoperabilität vergrößern sich die Einsatzmöglichkeiten für die Anwender und damit auch der Markt für die Produkte.

5.2 Übersicht des X/Open-Standards für verteilte Transaktionsverarbeitung

Die X/Open Company ist eine weltweit operierende Organisation, die von vielen Hard- und Software-Herstellern unterstützt wird und sich die Standardisierung einer Umgebung zur Unterstützung offener Systeme zum Ziel gesetzt hat. Diese Umgebung—die Common Applications Environment (CAE)—soll existierende Standards aus allen für offene System relevanten Bereichen in sich zusammenfassen und gegebenenfalls um neu zu erschaffende Standards für noch nicht abgedeckte Bereiche ergänzen. Der Bereich der verteilten Transaktionsverarbeitung wird durch das X/Open Distributed Transaction Processing (DTP) Modell abgedeckt. Dieses Modell spezifiziert eine Software-Umgebung, die mehreren Applikationen den gleichzeitigen Zugriff auf transaktionale Server, auch *Ressource-Manager* genannt, ermöglicht und die Zusammenfassung der von den Ressource-Managern ausgeführten Operationen zu verteilten Transaktionen unterstützt.

Eine Transaktion ist in diesem Modell als Arbeitseinheit definiert, deren Wirkungsbereich nicht auf Datenbanken beschränkt sein muß, sondern z.B. auch Kommunikationseinrichtungen oder Benutzeroberflächen, kurzum jeden beliebigen Zustand einbeziehen kann. Das Modell ist also flexibel genug, um auch den Zustand eines Tycoon-Programmes in eine Transaktion einbeziehen zu können. In einer verteilten, auch global genannten, Transaktion werden alle von den teilnehmenden Ressource-Managern ausgeführten Arbeitsschritte zusammengefaßt. Die Sequenz der Arbeitsschritte kann nicht weiter innerhalb der Transaktion strukturiert werden, es wird also nur das flache Transaktionsmodell unterstützt. Das Modell spezifiziert folgende Komponenten:

- das Applikationsprogramm (*Application Program*, AP), die Operationen auf den Ressource-Managern durchführt und Transaktionsgrenzen vorgibt
- die Ressource-Manager (*Resource Manager*, RM), die die Ressourcen verwalten

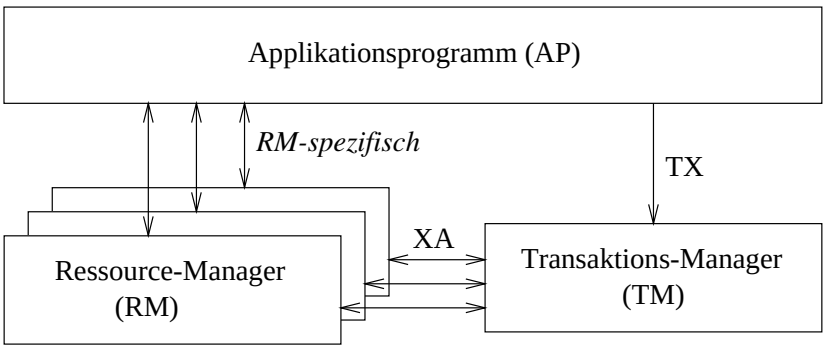

Abbildung 5.4: Komponenten und Schnittstellen einer Instanz

- der Transaktions-Manager (*Transaction Manager*, TM), der die Ausführung von Transaktionen überwacht und ihre Beendigung bzw. ihren Abbruch koordiniert
- der Kommunikationsressourcen-Manager (*Communication Resource Manager*, CRM), der für die Kommunikation zwischen Applikation und anderen Komponenten zuständig ist

Wie diese Komponenten im einfachsten, nicht-verteilten Fall zusammenwirken zeigt die Abbildung 5.4 (angelehnt an [X/Open Ltd. 93a], Abbildung 2–1). Man sieht hierin die Komponenten einer *Instanz des Modells*: ein Applikationsprogramm, ein Transaktions-Manager und ein oder mehrere Ressource-Manager. Eine Instanz ist an einen Rechner gebunden und kann nicht über mehrere Rechner im Netz verteilt sein. Jeder Rechner kann mehrere Instanzen beherbergen, die sich aber jeweils den Transaktions-Manager teilen—aus diesem Grund wird ein Rechner auch als Transaktions-Manager-Domäne angesehen. Anhand dieses Szenarios lassen sich die Rollen der Komponenten genauer erläutern:

Applikationsprogramm Das Applikationsprogramm implementiert die vom Endanwender geforderte Funktionalität. Dazu führt sie Operationen mit den Ressource-Managern über deren spezifische Schnittstellen durch und faßt diese Operationen zu globalen Transaktionen zusammen. Die Applikation trifft die Entscheidung, ob eine Transaktion erfolgreich beendet oder abgebrochen werden soll; diese Entscheidung wird dem Transaktions-Manager über die *TX*-Schnittstelle mitgeteilt.

Transaktions-Manager Der Transaktions-Manager verwaltet die globalen Transaktionen und führt Buch über die in seiner Domäne teilnehmenden Ressourcen-Manager pro Transaktion. Der Transaktions-Manager koordiniert die erfolgreiche Beendigung oder den Abbruch einer Transaktion mit Hilfe eines 2PC-Protokolls, welches über die XA-Schnittstelle mit den Ressource-Managern durchgeführt wird.

Ressource-Manager Der Ressource-Manager ist für die Verwaltung einer Ressource verantwortlich und bietet dafür eine spezifische Schnittstelle an, die von den Applikationen oder anderen Ressource-Managern genutzt wird. Prominente Beispiele für Ressource-Manager sind relationale Datenbanksysteme. Ein anderes Beispiel ist das Tycoon-System, welches den Ausführungszustand einer Applikation verwaltet. Ein wichtige Eigenschaft eines Ressource-Managers ist, daß er die Ressource transaktional verwaltet,

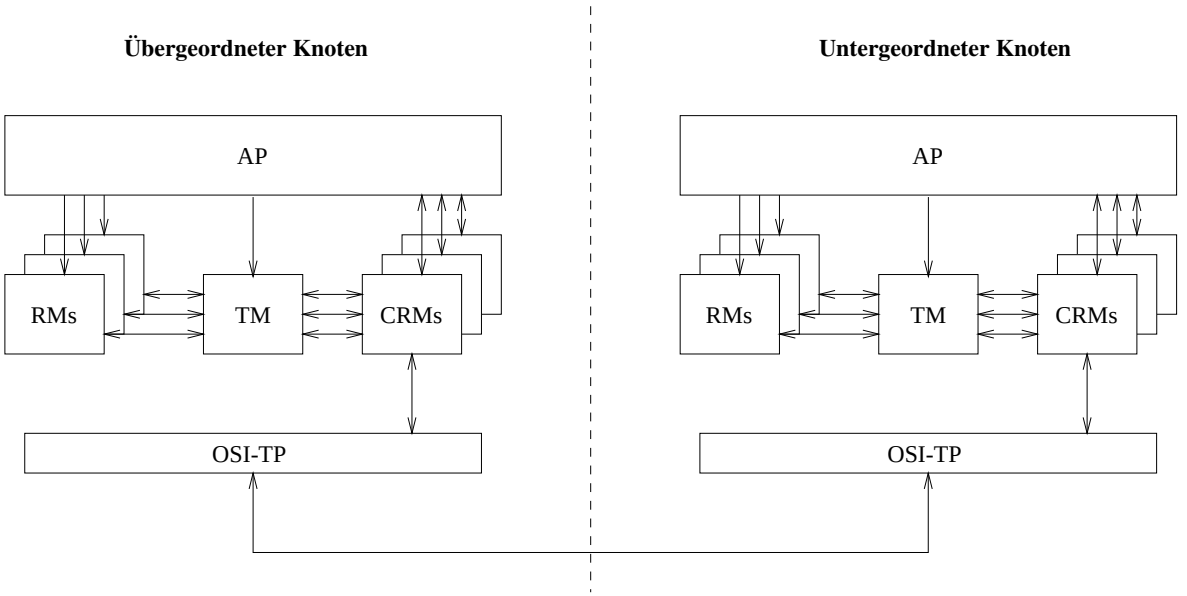

Abbildung 5.5: Komponenten und Schnittstellen einer verteilten Anwendung

und daß er in der Lage ist, die Koordinationsanweisungen des Transaktions-Managers umzusetzen.

Im Fall einer verteilten Anwendung arbeiten zwei oder mehrere Applikationsprogramme zusammen, wobei immer eines als übergeordnetes, das andere als untergeordnet angesehen wird. Zur Kommunikation zwischen den Applikationsprogrammen wird auch das Hinzuziehen der Kommunikationsressource-Manager notwendig, und es ergibt sich das in Abbildung 5.5 dargestellte Bild (aus [X/Open Ltd. 93a], Abbildung 3–1). Die Kommunikationsressource-Manager unterstützen die Kommunikation zwischen Instanzen, wobei diese sich auch innerhalb derselben Domäne befinden können, sie also nicht räumlich verteilt sein müssen. Für die Kommunikation zwischen den Applikationsprogrammen sind mehrere Paradigmen vorgesehen (z.B. RPC oder Peer-to-Peer), deswegen kann eine Instanz auch mehrere Kommunikationsressource-Manager umfassen. Um eine globale Transaktion zwischen mehreren Transaktions-Manager-Domänen auszuführen, ist auch Kommunikation zwischen den Transaktions-Managern der jeweiligen Domänen notwendig. Das X/Open-Modell spezifiziert hier die Verwendung der Dienste der OSI-Transaktionsverarbeitung (OSI-TP), also der OSI-Standards CCR ([OSICCR 89]), TP ([OSITP 92]) und so fort. Die zweistufe hierarchische Anordnung der Instanzen (eine übergeordnete, mehrere untergeordnete) kann auch zu einer allgemeinen mehrstufigen Hierarchie ausgebaut werden. Den untergeordneten Instanzen können dann noch weitere Instanzen untergeordnet sein und es ergibt sich so eine Baumstruktur. Bei den einzelnen beteiligten Instanzen spricht man deshalb auch von den *Zweigen* eines Transaktionsbaumes.

Im vorigen Abschnitt wurde erwähnt, daß durch das X/Open DTP-Modell sowohl die Protokollelemenmte als auch die Schnittstellen standardisiert werden. Durch den Einsatz der OSI-TP-Standards wird automatisch eine Standardisierung auf den Protokollebene erreicht. Die bereits erwähnten Schnittstellen sind auch größtenteils standardisiert und sollen an dieser Stelle etwas genauer beschrieben werden; die folgenden Beschreibungen sind nach Komponentenpaaren geordnet.

AP–TM Die TX-Schnittstelle des Transaktions-Managers erlaubt der Applikation die Markierung der Transaktionsgrenzen. Die wichtigsten Aufrufe dieser Schnittstelle sind *tx_begin()*, *tx_commit()* und *tx_rollback()*; für die Details dieser Schnittstelle siehe [X/Open Ltd. 92]).

AP–RM Diese Schnittstelle ist auf die spezifischen Fähigkeiten des Ressource-Managers ausgerichtet und entzieht sich deshalb einer generellen Standardisierung. Für eine prominenten Klasse von Ressource-Managern, den relationalen Datenbanksystemen, existiert aber auch eine standardisierte Form einer SQL-Schnittstelle.

AP–CRM Diese Schnittstelle stellt für das nutzende Applikationsprogramm auch gleichzeitig die indirekte Schnittstelle zum Partnerapplikationsprogramm dar, da das Applikationsprogramm über die Schnittstellen der Kommunikationsressourcen-Manager die Kommunikation abwickelt. Um die Kommunikationsformen der Anwendung möglichst wenig einzuschränken, sieht das X/Open-Modell drei Kommunikationsparadigmen vor:

1. die TxRPC-Schnittstelle, die einem Applikationsprogramm, dem Klienten, den Aufruf von Prozeduren in einer entfernten Server-AP ermöglicht. Die Aufrufsemantik ist synchron und ähnelt damit einem lokalen Prozeduraufruf. Der Wirkungsbereich der Klienten-Transaktion kann auch auf das Server-Applikationsprogramm ausgedehnt werden, man spricht dann von einem transaktionalen RPC. Die Details dieser Schnittstelle sind in [X/Open Ltd. 93b] zu finden.

2. die XATMI-Schnittstelle, die ebenfalls dem Client/Server-Paradigma zuzurechnen ist. Sie unterstützt Dienste, die entweder nach dem *request/response*-Prinzip oder als Konversation zwischen Klient und Server ablaufen.

3. die Peer-to-Peer-Schnittstelle, die einen Dialog zwischen zwei Applikationsprogrammen ermöglicht, wobei ein Applikationsprogramm der Initiator, das andere der Empfänger ist. Um die zuvor erwähnte Baumstruktur zwischen Instanzen, die an derselben globalen Transaktion teilnehmen, aufzubauen, kann ein Applikationsprogramm gleichzeitig Empfänger des einen Dialogs und Initiator eines anderen sein.

TM–RM Die bidirektionale XA-Schnittstelle ermöglicht die Koordination der globalen Transaktionen durch den Transaktions-Manager und die Teilnahme von Ressource-Managern an globalen Transaktionen. Weil diese Schnittstelle zentral für die Integration von Tycoon in das X/Open-Modell ist, sollen die einzelnen Aufrufe hier etwas ausführlicher behandelt werden. Da die Schnittstelle bidirektional ist, wird anhand des Prozedur-Präfixes die Richtung unterschieden: die *ax_**-Aufrufe gehen vom Ressource-Manager an den Transaktions-Manager, die *xa_**-Aufrufe vom Transaktions-Manager an den Ressource-Manager.

- Um die Arbeit eines Ressource-Managers mit einer bestimmten globalen Transaktion zu verbinden, muß der Ressource-Manager für die Transaktion bei einem Transaktions-Manager registiert sein. Dies geschieht entweder beim Start der Transaktion, indem der Transaktions-Manager die Ressource-Manager durch Aufruf von *xa_start()* benachrichtigt und die Transaktionskennung mitteilt; alternativ kann sich auch der Ressource-Manager selbständig vor dem ersten Aufruf durch das Applikationsprogramm beim Transaktions-Manager mit *ax_reg()* registrieren.

- Zu Beginn des 2PC ruft der Transaktions-Manager die Funktion *xa_prepare()* jedes beteiligten Ressource-Managers auf, diese liefern ihre Antwort als Funktionswert.
- Daraufhin wird der Ressource-Manager durch die Aufrufe *xa_commit()* und *xa_rollback()* von Ausgang verständigt.
- Unmittelbar vor dem Beginn des 2PC ruft der Transaktions-Manager die Funktion *xa_end()* für die registrierten Ressource-Manager auf, um ihnen die bevorstehende Terminierung der Transaktion mitzuteilten.
- Der Aufruf *xa_recover()* liefert die Liste der Transaktionen, die sich beim Ressource-Manager im *Prepare*-Zustand befinden und noch nicht vollständig abgeschlossen sind. Der Transaktions-Manager ruft diese Funktion nach dem Wiederanlauf eines Ressource-Managers auf, um den Status von dessen *Prepare*-Transaktionen zu klären.
- Das X/Open-Modell sieht auch die unilaterale heuristische Beendigung einer Transaktion durch einen Ressource-Manager vor, mit dem ein übermäßig langes Blockieren einer Ressource durch eine *Prepare*-Transaktion verhindert werden kann. Einzige Anforderung ist, daß der Ressource-Manager die bezüglich einer heuristisch terminierten Transaktion getroffene Ausgangsentscheidung solange zurückbehält, bis der Transaktions-Manager die Entscheidung durch Aufruf von *xa_forget()* freigibt.

Die Einstiegspunkte für die *xa_**-Aufrufe werden dem Transaktions-Manager über die sogenannte XA-Switch-Struktur mitgeteilt, die auch noch weitere Ressource-Manager-spezifische Informationen für den Transaktions-Manager bereithält.

TM-CRM : Da es sich bei Kommunikationsressource-Managern auch um Ressource-Manager handelt, ist die zwischen Transaktions-Manager und Kommunikationsressource-Manager benutzte XA+-Schnittstelle eine Obermenge der XA-Schnittstelle. Zu den für die XA-Schnittstelle erwähnten Aufrufen kommen noch weitere Aufrufe zum Weiterleiten von Transaktionsinformation an untergeordnete Transaktions-Manager und dergleichen (für Details siehe [X/Open Ltd. 93c]).

CRM-OSI TP Mit der XAP-TP-Schnittstelle wird eine Standardschnittstelle für die OSI-TP-Dienste vorgegeben.

5.3 Integration von Tycoon in das X/Open-Modell

Der vorige Abschnitt hat das X/Open-Modell für verteilte Transaktionsverarbeitung mit seinen Komponenten und für Tycoon relevanten Schnittstellen dargestellt. In diesem Abschnitt soll gezeigt werden, wie sich ein Tycoon-System in dieses Modell einfügen läßt, daß heißt welche Rollen es wahrnimmt und wie die Schnittstellen verwendet werden.

Auf der einen Seite verhält sich ein Tycoon-System gegenüber einer X/Open-Umgebung wie eine Anwendung: Ressource-Manager werden über Anpassungsmodule als externe Bibliotheken aufgerufen und zur Simulation der Tycoon-internen Synchronisationspunkte werden Transaktionsgrenzen markiert. Diese Anwendung kann möglicherweise verteilt sein, wenn zwei

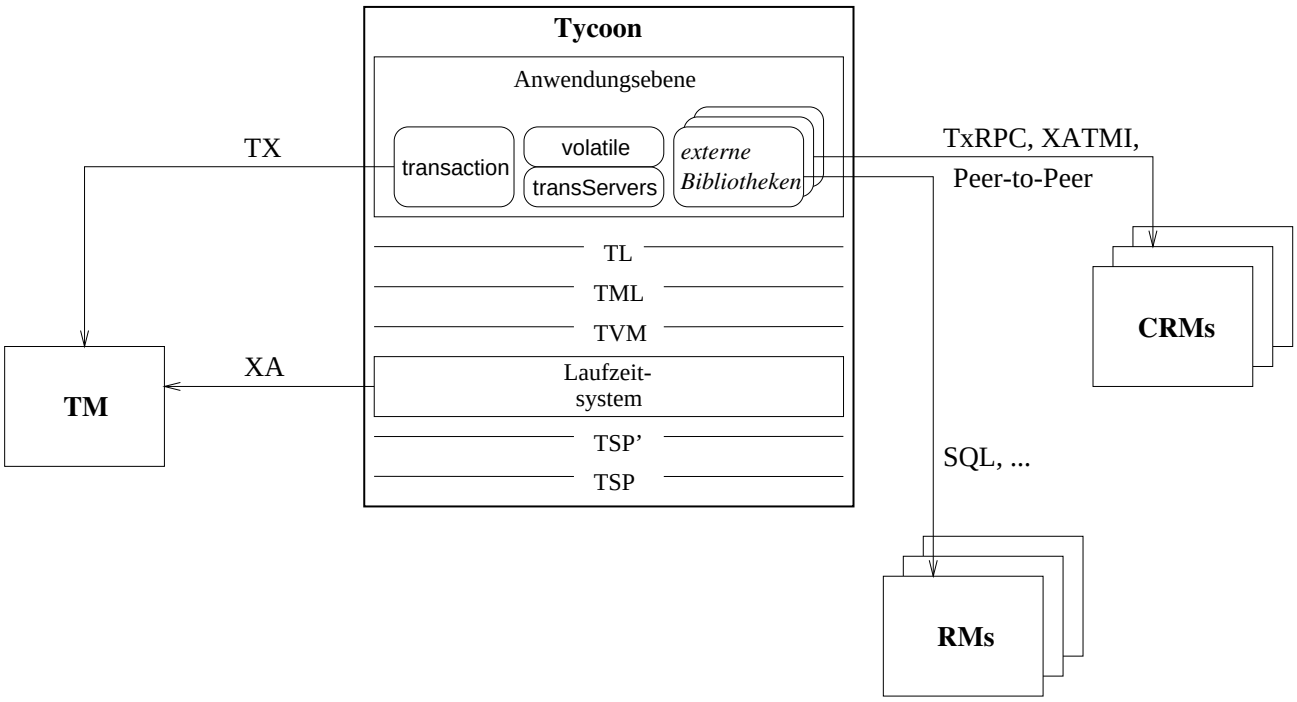

Abbildung 5.6: Integration eines Tycoon-Systems in das X/Open-Model

Tycoon-Systeme kooperieren, und würden dann im X/Open-Modell mehrere Instanzen in verschiedenen Transaktions-Manager-Domänen einnehmen. Diese Rolle als Anwendung reicht jedoch für das gesamte Tycoon-System nicht aus, weil, wie schon in Abschnitt 5.1 begründet wurde, die Sicherung des Store-Zustandes ebenfalls in das 2PC-Protokoll eingebunden werden muß, um Inkonsistenzen zu vermeiden. Aus diesen Überlegungen ergibt sich das in Abbildung 5.6 dargestellte Gesamtbild der Integration eines Tycoon-Systems in eine X/Open-Umgebung. Der Einfachheit halber wurde dabei von einer zentralen Anwendung ausgegangen, so daß die Darstellung einer einzelnen Instanz ausreicht.

Die Setzen von Synchronisationspunkten wird wie zuvor von der Tycoon-Anwendung über das Modul *transaction* durchgeführt, nur wird anstelle der TSP-Funktion die Elemente der TX-Schnittstelle (*tx_begin*, *tx_commit* und *tx_rollback*) verwendet. Um die Ressource-Manager und die Kommunikationsmechanismen in Tycoon einzusetzen, müssen Anpassungsmodule zur Verfügung stehen; diese werden dann auf dieselbe Weise wie andere Bibliotheken aufgerufen. Die Anwendungsebene des Tycoon-Systems nimmt also im X/Open-Modell die Rolle der Applikationsprogramm-Komponente wahr.

Zur Durchführung des 2PC-Protokolls mit dem Transaktions-Manager muß die XA-Switch-Komponente implementiert werden; diese wird vom Laufzeitsystem bereitgestellt, wobei die *xa_*-Aufrufe in einem von der Applikation unabhängigem Thread bearbeitet werden. Diese Konstruktion ist notwendig, weil bei der Durchführung des 2PC der Applikations-Thread mit dem *tx_commit*-Aufruf blockiert ist. Die Rolle des Ressource-Managers wird im Tycoon-System also vom Laufzeitsystem wahrgenommen. Abbildung 5.7 faßt noch einmal zusammen, welche Rollen ein Tycoon-System in dem X/Open-Modell wahrnimmt.

Wie bereits erwähnt verhalten sich die transaktionalen Algorithmen für externe transak-

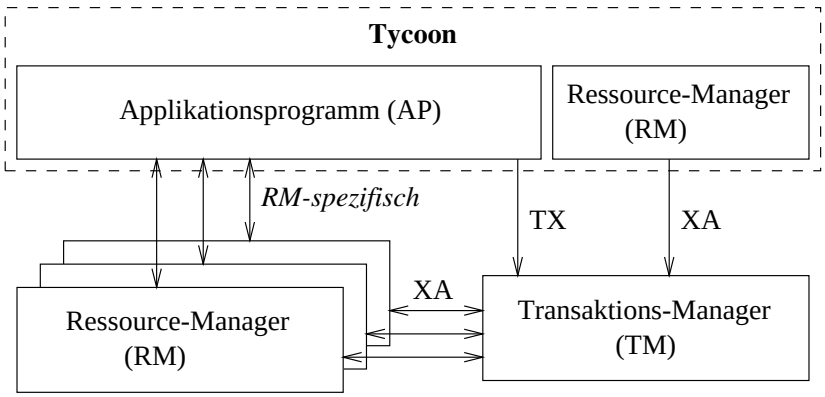

Abbildung 5.7: Die Rollen von Tycoon im X/Open-Modell

tionale Daten ähnlich wie für Tycoon-interne Daten. Bei der Einbeziehung externer Dienste durch ein 2PC-Protokoll muß aber besonders darauf geachtet werden, daß die drei unterschiedlichen Zustände—interner, externer flüchtiger und transaktionaler—immer synchron sind. Dies wird jetzt erschwert, weil der externe transaktionale Zustand auch extern, d.h. ohne Anweisung von Tycoon, zurückgesetzt werden kann. Der Fall tritt ein, wenn in der zweiten Phase des 2PC Tycoon im *Prepare*-Zustand die Abbruchaufforderung erhält. Welche Konsequenzen das hat, wird in der folgenden detaillierten Darstellung der Algorithmen zum Setzen von Synchronisationspunkten, Zurücksetzen und Wiederanlauf erläutert. Bei der Darstellung wird davon ausgegangen, daß der verwendete Store die optionale 2PC-Fähigkeit des TSP unterstützt.

5.3.1 Setzen eines Synchronisationspunktes

Beim Setzen eines Synchronisationspunktes wird die Log-Struktur für externe Daten in der gleichen Weise wie der Store-Log geändert: beim Commit wird sie abgeschnitten, ansonsten wird ein Eintrag mit der Nummer des Sicherungspunktes an das Log angehängt. Danach wird vom Modul *transaction* der Synchronisationspunkt durch Beendigung der globalen Transaktion mit einem Aufruf von *tx_commit* gesichert. Bei einem vollständig fehlerfreien Ablauf des daraufhin zwischen dem Transaktions-Manager und dem Tycoon-Laufzeitsystem ausgeführten 2PC werden folgende Schritte ausgeführt:

1. Beim Aufruf von *xa_prepare()* wird dem Laufzeitsystem die Kennung der globalen Transaktion (*XID*) übergeben. Die XID wird persistent im Store gespeichert, ebenso wird persistent ein Flag gesetzt, was den jetzt eingenommenen *Prepare*-Zustand signalisiert. Erst dann wird *tsp_prepareCommit* aufgerufen; durch das TSP wird garantiert, daß auch bei einem unmittelbar darauf folgenden Absturz die Store-Transaktion durch *tsp_commit* noch erfolgreich beendet werden kann. Mit der Rückgabe von *XA_OK* wird *xa_prepare* beendet und dem Transaktions-Manager der erfolgreiche Abschluß der *Prepare*-Phase in Tycoon signalisiert.

2. Beim folgenden Aufruf von *xa_commit* wird die Store-Transaktion mit *tsp_commit* erfolgreich beendet[1] und dem Transaktions-Manager dies durch Rückgabe von *XA_OK*

[1] Das TSP garantiert nach einem erfolgreichen *tsp_prepareCommit()* die Ausführbarkeit von *tsp_commit()*..

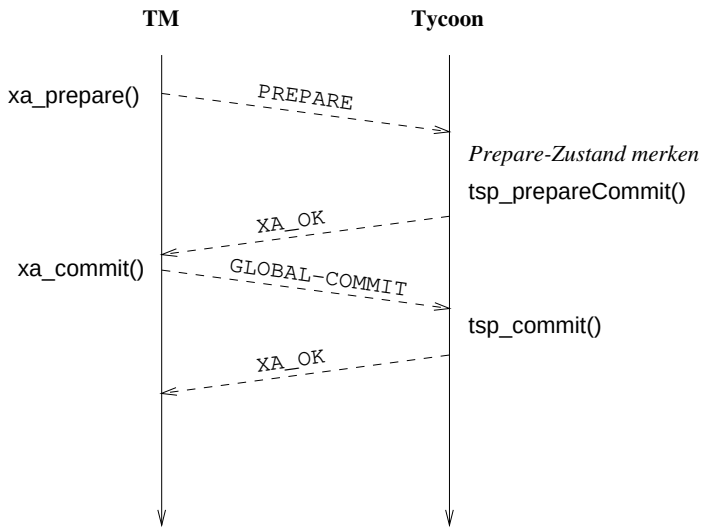

Abbildung 5.8: Ablauf des 2PC im Erfolgsfall

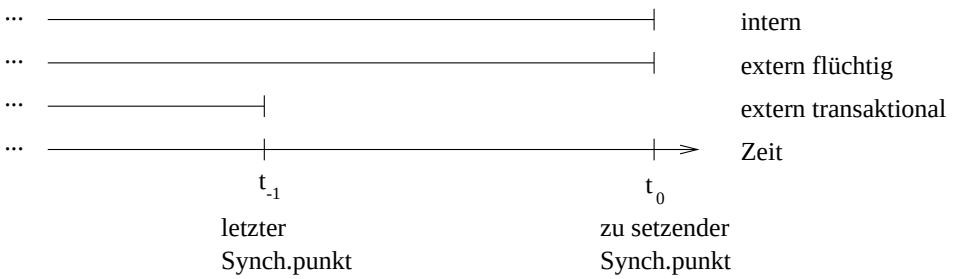

Abbildung 5.9: Zustand nach dem Abbruch der globalen Transaktion beim Setzen eines Synchronisationspunktes

angezeigt.

Abbildung 5.8 zeigt diesen Ablauf schematisch.

Tritt in der *Prepare*-Phase ein Store-Fehler bei Aufruf von *tsp_prepareCommit* auf, kann die Transaktion nicht erfolgreich beendet werden und das Laufzeitsystem initiiert den Abbruch der globalen Transaktion durch Rückgabe von *XA_RBROLLBACK* an den Transaktions-Manager. Da nach dem Store-Fehler kein Weiterarbeiten in Tycoon mehr möglich ist, wird das Tycoon-System angehalten. Das TSP setzt in diesem Fall die Store-Transaktion automatisch zurück, so daß interner und externer persistenter Zustand synchron sind.

Verläuft die *Prepare*-Phase erfolgreich und wird erst in der zweiten Phase der Abbruch vom Transaktions-Manager durch Aufruf von *xa_rollback* signalisiert, müssen der interne und externe flüchtige wieder mit dem externen transaktionalen Zustand synchronisiert werden. Letzterer wird automatisch durch den Transaktions-Manager zurückgesetzt, wodurch sich die in Abbildung 5.9 dargestellte Konstellation ergibt. Die einfachste Möglichkeit zur Resynchronisation besteht im Abbruch der Store-Transaktion und anschließendem Anhalten des Tycoon-Systems; Abbildung 5.10 zeigt diese Variante. Durch den Abbruch der Store-Transaktion wird der interne mit dem externen transaktionalen Zustand synchronisiert; der externe flüchtige Zustand wird dann beim Wiederanlauf korrekt wiederhergestellt. Alternativ

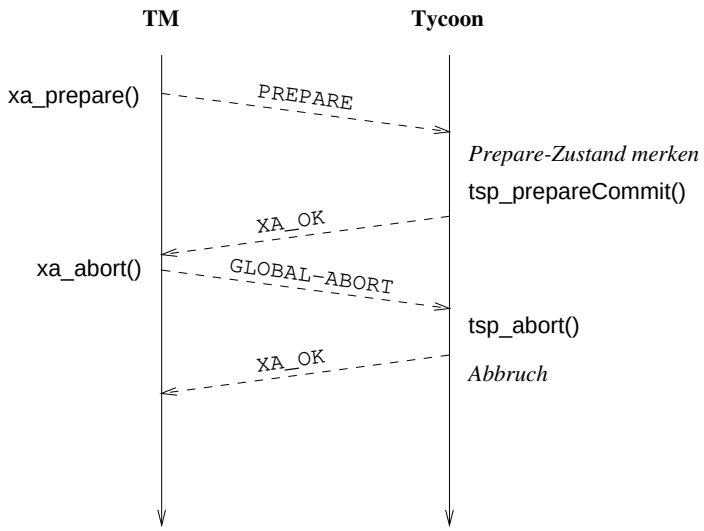

Abbildung 5.10: Ablauf des 2PC im Fehlerfall

zum Anhalten von Tycoon kann auch der externe flüchtige Zustand zurückgesetzt werden. Dies muß aber vor dem Abbruch der Store-Transaktion geschehen, weil ansonsten die dafür notwendigen Log-Einträge fehlen. Der Applikation wird der erzwungene Abbruch der globalen Transaktion durch einen entsprechenden Rückgabewert (*viaForcedRollback*) angezeigt.

Um den Verlust der geleisteten Arbeit zu vermeiden, wäre es auch denkbar, als Reaktion auf die Abbruchmeldung die ausgeführten externen Operationen zu wiederholen, um so den externen transaktionalen Zustand mit dem internen und externen flüchtigen zu synchronisieren. Ein solches Vorgehen wäre nach entsprechender Erweiterung der Log-Struktur um die nötige *Redo*-Information machbar, hat jedoch einen gravierenden Nachteil. Es kann passieren, daß auch nach automatischer Wiederholung der Operationen die globale Transaktion immer noch nicht erfolgreich beendet werden kann. Auf diese Weise würde sich dann eine nicht mehr zu unterbrechende Endlosschleife aus Operationswiederholung und Transaktionsabbruch einstellen.

5.3.2 Zurücksetzen

Das Zurücksetzen des externen transaktionalen Zustands wird auch analog zum Zurücksetzen des Stores durchgeführt. Soll nur zum letzten gesetzten Synchronisationspunkt zurückgesetzt werden, reicht dafür der Abbruch der aktuellen globalen Transaktion durch Aufruf von *tx_rollback*. Beim darauf folgenden Aufruf von *xa_rollback* wird mit *tsp_rollback* die Store-Transaktion abgebrochen.

Sollen beim Zurücksetzen Synchronisationspunkte übersprungen werden, müssen die verzeichneten kompensierenden Operation ausgeführt und der dann erreichte Synchronisationspunkt durch Beendigung der globalen Transaktion mit einem 2PC gesichert werden. Wie auch beim Setzen eines Synchronisationspunktes wird das 2PC durch den Aufruf von *tx_commit* durch das Modul *transaction* angestoßen und durchläuft im fehlerfreien Fall dieselben zwei Schritte: Beim Aufruf von *xa_prepare* wird erfolgreich *tsp_prepareCommit* ausgeführt und der Status *XA_OK* zurückgemeldet; danach beendet *xa_commit* die Store-Transaktion durch Aufruf von *tsp_commit*.

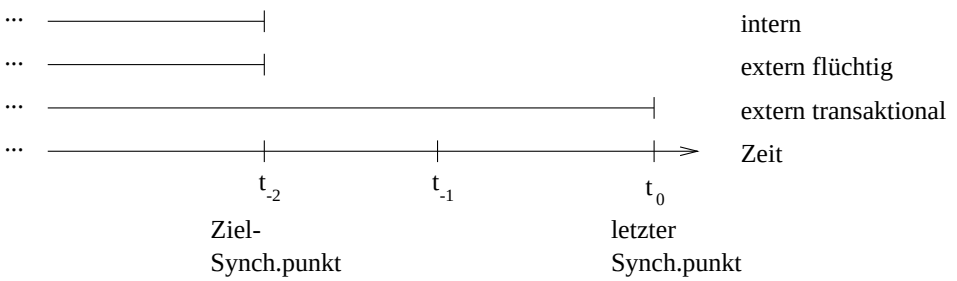

Abbildung 5.11: Zustand nach dem Abbruch der globalen Transaktion beim Zurücksetzen

Auf das Eintreten eines Store-Fehlers beim Aufruf von *tsp_prepareCommit* wird wieder genauso wie auch beim Setzen eines Synchronisationspunktes reagiert. Wird demhingegen nach erfolgreichem *Prepare* die globale Transaktion abgebrochen, müssen der interne und externe flüchtige Zustand wieder resynchronisiert werden. Im Gegensatz zum Abbruch während des Setzens eines Synchronisationspunktes befindet sich jetzt der externe transaktionale Zustand zeitlich „hinter“ dem internen und externen flüchtigen, wie die Abbildung 5.11 zeigt.

Die einfachste Möglichkeit der Synchronisation aller Zustände besteht wiederum im Abbruch der Store-Transaktion gefolgt vom Anhalten des Tycoon-Systems. Nach dem Wiederanlauf befinden sich dann alle Zustände beim letzten Synchronisationspunkt (t_0 in der Abbildung). Um das Anhalten von Tycoon zu vermeiden, müßte der externe flüchtige Zustand durch ein *Rollforward* auf den Stand t_0 gebracht werden. Dies ist im Modul *volatile* nicht vorgesehen, kann aber mit entsprechendem Aufwand nachgeführt werden. Die automatische Wiederholung der kompensierenden Operationen ist auch hier aus demselben Grund wie vorher abzulehnen.

5.3.3 Wiederanlauf

Beim Wiederanlauf des Tycoon-Systems muß sichergestellt werden, daß Store-Zustand und externer transaktionaler Zustand synchronisiert sind, bevor der Wiederanlauf des flüchtigen Zustands begonnen und das System zur Benutzung freigegeben wird. Dank der Ausführung des 2PC ist fast immer gewährleistet, daß die persistenten Zustände synchronisiert sind, so daß üblicherweise sofort mit der Wiederherstellung des flüchtigen Zustands begonnen werden kann.

Die einzige Ausnahme tritt auf, wenn die letzte Store-Transaktion noch im *Prepare*-Zustand ist. In diesem Fall hat Tycoon den Ausgang der globalen Transaktion nicht miterlebt, weil es nach dem erfolgreichen Ausführen von *tsp_prepareCommit* und vor der Benachrichtigung über den Transaktionsausgang abgestürzt ist. Um die unvollendete Transaktion terminieren zu können, wird der Wiederanlauf mit Hilfe des Transaktions-Managers durchgeführt. Zuerst muß dem Transaktions-Manager die Kennung der unvollendeten Transaktion mitgeteilt werden. Dies geschieht über einen vom Transaktions-Manager durchgeführten Aufruf von *xa_recover()*, der eine Liste der *Prepare*-Transaktionen—hier nur mit einem Eintrag—an den Transaktions-Manager liefert. Der Transaktions-Manager kann jetzt Tycoon über den Ausgang der globalen Transaktion durch Aufruf von *xa_commit()* oder *xa_rollback()* unterrichten. Entsprechend des Ausgangs wird dann *tsp_commit()* oder *tsp_rollback()* aufgerufen; danach sind die persistenten Zustände wieder synchron und es kann mit der Wiederherstellung des

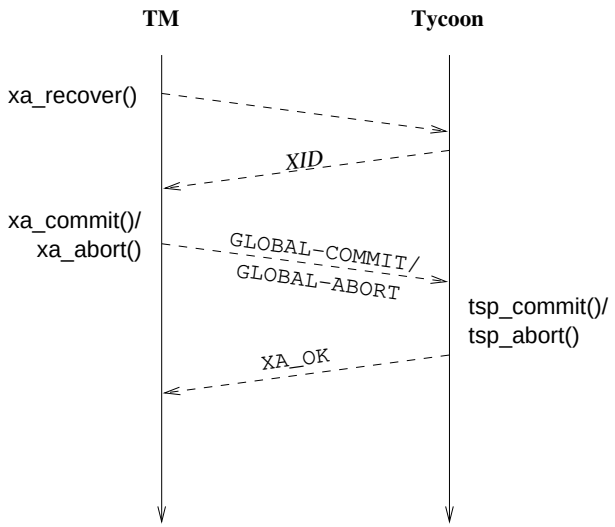

Abbildung 5.12: Wiederanlauf mit dem Transaktions-Manager

flüchtigen externen Zustandes begonnen werden. Abbildung 5.12 veranschaulicht die Interaktion zwischen Transaktion-Manager und Tycoon beim Wiederanlauf.

6. Zusammenfassung

In der vorliegenden Arbeit werden Transaktionsmechanismen für eine offene persistene Programmiersprache als Basistechnologie zur Unterstützung langlebiger Aktivitäten entwickelt. Die Grundlage dafür bildet das Tycoon-System, eine Programmierumgebung mit einer funktionalen, statisch typisiserten Programmiersprache und der Fähigkeit der Interoperabilität zu externen Systemkomponenten. Ein Kernpunkt ist die Erweiterung der in Tycoon vorhandenen Transaktionsmechanismen auch auf externe Daten. Dadurch wird die in Tycoon schon bestehende uniforme Sichtweise der Daten—interne wie externe Daten werden mit denselben sprachlichen Mitteln behandelt—auch auf das Transaktionskonzept ausgeweitet. Es entsteht so eine integrierte Entwicklungsumgebung mit orthogonaler Transaktionsfähigkeit.

Um dem zweiten Aspekt der Zielsetzung, nämlich der Unterstützung langlebiger Aktivitäten, gerecht zu werden, war eine Ablösung des bisher vertretenen und für langlebige Aktivitäten ungeeigneten flachen Transaktionskonzeptes erforderlich. Das in der Arbeit vorgestellte und verwendete Konzept der persistenten Sicherungspunkte zeichnet sich durch eine höhere Flexibilität bei der Strukturierung von Transaktionen und bei der Fehlererholung aus. Es hat zudem den Vorteil, daß über eine Abbildung von Sicherungspunkten auf eine Kette von Transaktionen auch externe transaktionale Ressourcen eingebunden werden können.

Im Hinblick auf dieses Transaktionsmodell wurden Transaktionsmechanismen entwickelt, die die Atomarität und Dauerhaftigkeit der Änderungen einer Applikation sicherstellen, unabhängig davon, ob diese sich auf interne oder externe, flüchtige oder persistente Daten beziehen. Um den Rahmen einer Diplomarbeit einzuhalten, wurde dabei auf die Berücksichtigung der Isolationsaspekte des Transaktionsmodells verzichtet und die Aufgabenstellung auf Mechanismen für ein Einbenutzersystem beschränkt.

Der Hauptteil der Arbeit beschäftigt sich mit der technischen Realisierung der transaktionalen Eigenschaften der Atomarität und Dauerhaftigkeit und gliedert sich entsprechend der Charakteristika der zu behandelnden Ressourcen. Diese wurden unterschieden in:

- Tycoon-interne, vom Tycoon-Objektspeicher persistent verwaltete Ressourcen;
- externe flüchtige Ressourcen, wie z.B. GUI-Fenster oder Netzwerkverbindungen, deren Zustand aus externen hauptspeicherresidenten Datenstrukturen besteht;
- externe transaktionale Ressourcen, wie z.B. relationale Datenbanken, die mit Persistenz und transaktionalen Eigenschaften bereits ausgestattet sind.

Bei der Implementierung von persistenten Sicherungspunkten für Tycoon-interne Daten war zu beachten, daß der Objektspeicher über das abstrakte Speicherprotokoll TSP angebunden wird, welches nur das flache Transaktionsmodell unterstützt. Um die Unabhängigkeit

von Tycoon zum Objektspeicher zu gewährleisten, mußte das Speicherprotokoll auf der untersten Ebene beibehalten werden. Die Erweiterung um persistente Sicherungspunkte wurde durch eine Anwendung des durch das TSP geförderten Schichtungsansatzes erreicht: Eine Softwareschicht, die unmittelbar oberhalb des TSP angesiedelt ist, implementiert ein um Sicherungspunkte erweitertes, ansonsten aber mit TSP identisches Speicherprotokoll. Grundlage der Implementation ist das chronologische Verzeichnen von geänderten Objekten in einer Log-Struktur. Analog zu dem Verfahren für externe transaktionale Ressourcen werden die Sicherungspunkte im Objektspeicher auf abgeschlossene Transaktionen abgebildet und das Zurücksetzen auf einen Sicherungspunkt durch die Ausführung von kompensierenden Operationen innerhalb des Objektspeichers erreicht.

Auch für externe flüchtige Ressourcen bildet eine Log-Struktur die Basis der Transaktionsmechanismen, nur werden in dieser nicht geänderte Objekte, sondern die für diese Objekte ausgeführten Operationen verzeichnet. Die „Dauerhaftigkeit“ des externen flüchtigen Zustandes wird durch Wiederholung der verzeichneten Operationen beim Wiederanlauf simuliert, beim Zurücksetzen werden die gleichermaßen im Log verzeichneten kompensierenden Operationen ausgeführt.

Da der naive Ansatz, alle jemals ausgeführten Operationen zu wiederholen bzw. zu kompensieren, besonders bei langlebigen Aktivitäten zu unakzeptabler Performanz führen würde, wurde ein abstraktes Operationsmodell entwickelt, welches die für eine effiziente Ausführung von Wiederanlauf und Zurücksetzen semantischen Charakteristika der Operationen enthält. Dieses Operationsmodell liegt der Log-Struktur zugrunde und erlaubt, in Kooperation mit dem Programmierer, eine selektive Ausführung der zur Wiederanlauf und Zurücksetzen notwendigen Operationen. Durch die selektive Wiederholung bzw. Kompensation wird der Aufwand der Transaktionsmechanismen gegenüber dem naiven Ansatz drastisch reduziert. Um das permanente Anwachsen der Log-Strukur zu vermeiden, wurde ein Verfahren entwickelt, mit dem die Log-Struktur durch Herauslöschen kompensierter Operationen bereinigt wird.

Für die letzte Gruppe von Ressourcen, die externen transaktional verwalteten Ressourcen, wurde ebenfalls ein Ansatz gewählt, der auf einem Operations-Log basiert und Sicherungspunkte auf abgeschlossene externe Transaktionen abbildet, die durch kompensierende Operationen zurückgesetzt werden. Um bei einem Sicherungspunkt die erfolgreiche Beendigung der externen Transaktion zuverlässig im Log zu verzeichnen, muß zwischen Tycoon und den beteiligten externen Servern ein zweiphasiges Commit (2PC) durchgeführt werden. Zur Wahrung der Portabilität und Interoperabilität von Tycoon wurde anstelle eines bestimmten herstellernahen Produktes ein standardisiertes Modell zur verteilten Transaktionsverarbeitung, das X/Open DTP Modell, ausgewählt. Dieses spezifiziert eine Umgebung, die den Zugriff auf lokale und entfernte Serversysteme unterstützt und die Durchführung eines 2PC zwischen allen beteiligten Parteien ermöglicht. Die Spezifikation der Transaktionsmechanismen zum Zurücksetzen und Wiederanlauf bezieht sich auf die relevanten X/Open-Schnittstellen.

6.1 Stand der Implementation

Die entwickelten Transaktionsmechanismen wurden größtenteils auch im Tycoon-System implementiert, einzig die Einbindung externer transaktionaler Ressourcen wurde wegen der hohen Komplexität der X/Open-Umgebung und der noch geringen Verfügbarkeit von Umsetzungen dieses Standards unterlassen.

Die bei der Implementierung gesammelten Erfahrungen haben die Realisierbarkeit und die Leistungsfähigkeit der erarbeiteten Konzepte und Algorithmen bestätigt. Bei der Erweiterung des TSP um persistente Sicherungspunkte durch die Logging-Zwischenschicht bestand anfangs der Verdacht, daß der dadurch eingeführte Zusatzaufwand unter Umständen zu hoch sein könnte. Dieser Verdacht hat sich aber als unbegründet erwiesen und selbst Vergleichsmessungen zeigen, daß der Performanzunterschied zu einem System ohne TSP-Erweiterung unter 30% liegt. Der Implementationsaufwand war mit circa 1300 Zeilen C-Code noch recht moderat, was auch der durch das TSP möglichen Schichtungsarchitektur zu verdanken ist.

Die transaktionalen Algorithmen für externe flüchtige Daten wurden einschließlich der Log-Bereinigung in einem Modul der Anwendungsebene vollständig umgesetzt und stehen für beliebige externe Bibliotheken zur Verfügung. Gerade die neuartige Fähigkeit zur transaktionalen Verwaltung flüchtiger Ressourcen zeigt deutlich die Vorteile eines orthogonalen Transaktionskonzeptes. Da nunmehr sämtliche von einer Applikation genutzten Ressourcen in das Transaktionskonzept integriert sind, müssen auch flüchtige Daten nicht mehr als Sonderfälle im Code behandelt werden. Zudem zeigte sich bei einer prototypischen Applikation, daß die Möglichkeit des kontrollierten Zurücksetzens durch persistente Sicherungspunkte auch bei rein interaktiven Programmen mit graphischer Benutzeroberfläche vorteilhaft eingesetzt werden kann. Ein Nachteil des Verfahrens besteht allerdings darin, daß es nicht vollkommen unsichtbar für den Applikationsprogrammierer ist; dieser muß vielmehr zum Zweck der Effizienzverbesserung in die Logging-Vorgänge eingreifen. Die Prototypenapplikation hat jedoch gezeigt, daß diese Eingriffe minimal und bei ausreichender Performanz entbehrlich sind.

6.2 Ausblick

Der Bereich der transaktionalen Unterstützung persistenter Programmiersysteme ist mit dieser Arbeit offensichtlich ängst nicht erschöpfend behandelt. Eine bereits in Kapitel 4 angesprochene Verbesserung betrifft das Operationsmodell, das den transaktionalen Verfahren für externe flüchtige Ressourcen zugrunde liegt. In dem Modell wurde die hierarchische Strukturierung der externen Objekte fest verankert, ohne weitere Strukturierungsalternativen anzubieten oder zu ermöglichen. Am flexibelsten wäre eine Architektur, in der die Strukturierung der externen Objekte in zusätzlichen Modulen untergebracht ist und entsprechend einfach umkonfiguriert und ergänzt werden kann.

Das Transaktionsmodell kann im Hinblick auf das Multithreading und die Migrationsfähigkeit von Applikationen noch verbessert werden. Mit ihrer gegenüber flachen Transaktionen erhöhten Flexibilität und Kontrolle weisen die persistenten Sicherungspunkte in die richtige Richtung, allerdings sind sie eher für zentralisierte Applikationen ohne interne Nebenläufigkeit und Verteilung gedacht. Die durch Threads erzielbare interne Nebenläufigkeit einer Applikation erfordert Synchronisationskonzepte, die bei persistenten Sicherungspunkten nicht vorgesehen sind und die vielleicht eine Erweiterung des Transaktionsmodells in Richtung genereller gesch chtelter Transaktionen sinnvoll werden lassen. Meines Wissens existiert derzeit auch noch kein erweitertes Transaktionsmodell, das die Migration einzelner Threads einer Applikation unterstützt. Wie ein Transaktionsmodell für durch migrierende Threads verteilte Applikationen aussehen muß, welches einerseits die Konsistenz der Daten wahrt, andererseits die Autonomie der Threads nicht zu stark behindert, ist eine Frage, die durch weitere Forschung zu klären ist.

A. Schnittstelle Volatile

```
interface Persistent

export

  nilError :Exception
  (* Raised if an attempt is made to execute the 'get' function of the
     persistent 'nil' value. *)

  error :Exception
  (* Raised if an attempt is made to access a destroyed pesistent.T. *)

  Let T(E <:Ok) = Tuple get():E end
  (* A persistent value of type 'persistent.T(E)' encapsulates a volatile
     object of type 'E' that is not stored within the Tycoon persistent object
     store (for example a window or a file handle). The 'get' function
     retrieves the encapsulated value. Creation, destruction and
     state changes of the encapsulated value are recorded in a persistent log.
     This log is used to propagate commit, savepoint and rollback
     operations on the Tycoon store to all volatile objects of type
     'persistent.T(X)'. As a consequence, application programmers can
     treat Tycoon store objects and persistent.T values uniformly.
     Note that A<:B implies persistent.T(A) <:persistent.T(B).
     ## should become  T(E <:+ Ok) <: Tuple get():E end in future
     versions of the type system :-) *)

  Let Any = T(Ok)
  (* The supertype of all persistent.T values. *)

  nil :T(Nok)
  (* A polymorphic nil value. 'nil.get()' raises 'nilError'. *)

(* – Operation Logging: *)

  create(E <:Ok  create():E  destroy(:E):Ok
    indepDestroy :Bool  parent :Any  references :Array(Any)) :T(E)
  (* Create and register a new persistent.T value by calling 'create' and
```

```
    inserting it into the operation log. The volatile value is assumed to be
    destroyed automatically ('indepDestroy == false') if 'parent' is destroyed
    (either explicitly or via a cascaded 'destroy'). If 'indepDestroy==true',
    a 'destroy' on the parent volatile triggers a call to 'destroy' of this
    persistent.T. If 'create' refers to other volatiles, these have to
    be passed as 'references'. During restart, these referenced
    volatiles are recreated, even if they are only temporary objects.
    Raise 'error' if 'create' returns a value that is not of type
    'word.T'. *)

  log(V <:Ok  redo() :V  undo() :Ok  parent :Any  references :Array(Any)) :V
  (* Log a side effecting operation on a persistent.T value.
     Execute 'redo' and return its result. For the meaning of 'parent' and
     'references' see the explanation of the 'create' function. *)

  destroy(E <:Ok  p :T(E)) :Ok
  (* Destroy 'p'. Further access to 'p' is not allowed. *)

  enableLogging() :Ok
  disableLogging() :Ok
  restoreLogging() :Ok
  (* Switch operation logging on/off or restore it to its prior state. If
     switched off, operations on persistent.T values are not logged, but
     creation is still logged. For 'restoreLogging' to work properly,
     every enable/disable must be followed by a matching 'restoreLogging'.
     Logging is initially switched on. *)

  loggingEnabled() :Bool
  (* Return true if logging is enabled. *)

  unlogged(execute() :Ok) :Ok
  (* Convenience function: execute 'execute' bracketed by
     'disable/restoreLogging' *)

(* – State Restauration: *)

  RestoreProc <:Ok
  (* The type of handles for restore procedures. *)

  registerRestore(restore():Ok  execNow :Bool  parent :Any) :RestoreProc
  (* Register a restore procedure that is executed after a rollback or a
     restart to restore the state of volatile objects which is not
     covered by 'create' and 'log' operations. A restore procedure may only
     only reference Tycoon store variables and must not reference
     non-local persistent.T values. The restore procedure is
     deregistered automatically if 'parent' is destroyed. *)
```

```
  deregisterRestore(restore :RestoreProc) :Ok
  (* Deregister a restore procedure. *)

(* – Callbacks for the Module 'Transaction': *)

  doSavepoint() :Ok
  (* Called before next savepoint is established. Append a savepoint entry
     to the operation log. *)

  doCommit() :Ok
  (* Called before next commit is performed. Purge the operation log and
     append a commit entry. *)

  doRollbackTo(syncPoint :Int) :Ok
  (* Called before module 'transaction' initiates rollback to the given sync
     point (either savepoint or commit point). The sync point is given as an
     integer, where '0' represents a commit point and ¿0 represents
     savepoints in chronological order of establishment. After a store
     rollback, 'doRollbackTo' has to be followed by a 'doRestart(true)'. *)

  doRestart(afterRollback :Bool) :Ok
  (* Re-execute the opeation log starting from the beginning and walk
     down restore list (if 'afterRollback==true', just walks down
     restore list) *)

end;
```

B. Schnittstelle Transaction

```
interface Transaction

export

  Via <:Ok

  Savepoint <: Ok

  error :Exception
  (* Raised when commit or savepoint fails. *)

  commitVia, rollbackVia, restartVia :Via

  savepoint() :Tuple s :Savepoint v :Via end

  commit() :Via

  rollbackTo(s :Savepoint) :Ok

  rollbackToLast() :Ok

  rollback() :Ok

end;
```

Literaturverzeichnis

Alonso et al. 94: Alonso, G., Kamath, M., Agrawal, D., El Abadi, A., Günthör, R., und Mohan, C. „Failure Handling in Large Scale Workflow Management Systems". Technical Report Research Report RJ9913, IBM, November 1994.

Astrahan et al. 79: Astrahan, M., Blasgen, M., Chamberlin, D., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, R., Mehl, J., Putzolu, G., Slutz, D., Strong, H., Tiberio, P., Traiger, I., und Yost, B. „An Overview of System R: A Relational Database System". *IEEE Computer*, Jg. 13, 1979, Nr. 4, S. 43–55.

Bancilhon et al. 92: Bancilhon, F., Delobel, C., und Kanellakis, P. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

Bernstein et al. 87: Bernstein, P.A., Hadzilacos, V., und Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

Copeland, Maier 84: Copeland, G. und Maier, D. „Making Smalltalk a database system". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, Juni 1984, S. 316–325.

Dearle et al. 89: Dearle, A., Connor, R., Brown, F., und Morrison, R. „Napier88 – A Database Programming Language?". In: *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Juni 1989.

Eswaran et al. 76: Eswaran, K.P., Gray, J.N., Lorie, R.A., und Traiger, I.L. „The Notion of Consistency and Predicate Locks in Database Systems". *Communications of the ACM*, Jg. 19, November 1976, Nr. 11, S. 624–633.

Garcia-Molina, Salem 87: Garcia-Molina, H. und Salem, K. „Sagas". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, Mai 1987, S. 249–259.

Georgakopoulos et al. 93: Georgakopoulos, D., Hornick, M., Manola, F., Brodie, M., Heiler, S., Nayeri, F., und Hurwitz, B. „An Extended Transaction Environment for Workflows in Distributed Object Computing". *IEEE Data Engineering Bulletin*, Jg. 16, 1993, Nr. 2.

Gray et al. 81: Gray, J., McJones, R., P. Blasgen, W., M. Lindsay, B., Lorie, A., R. Price, G., T. Putzolu, R., G. und Traiger, L., I.˙„The Recovery Manager of the System R Database Manager". *ACM Computing Surveys*, Jg. 13, 1981, Nr. 2, S. 223–242.

Gray, Reuter 93: Gray, J. und Reuter, A. *Transaction Processing – Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1993.

Gray 78: Gray, J. „Notes on Database Operating Systems". In: *Operating Systems – An Advanced Course*, *Lecture Notes in Computer Science*, Bd. 60. Springer-Verlag, 1978.

Härder, Reuter 83: Härder, T. und Reuter, A. „Principles of Transaction-Oriented Database Recovery". *ACM Computing Surveys*, Jg. 15, 1983, Nr. 4, S. 287–317.

Hsu et al. 93: Hsu, M., Obermarck, R., und Vuurboom, R. „Workflow Model and Execution". *IEEE Data Engineering Bulletin*, Jg. 16, 1993, Nr. 2.

Jablonski 94: Jablonski, S. „MOBILE: A Modular Workflow Model and Architecture". In: *Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems, Noordwijkerhout, The Netherlands*, November 1994.

Jablonski 95: Jablonski, S. „Workflow-Management-Systeme: Motivation, Modellierung, Architektur". *Informatik Spektrum*, Jg. 18, 1995, Nr. 1, S. 13–24.

Kent et al. 85: Kent, J., Garcia-Molina, H., und Chung, J. „An experimental evaluation of crash recovery mechanisms". In: *Proceedings of the Fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1985, S. 113–122.

Lamb et al. 92: Lamb, C., Landis, G., Orenstein, J., und Weinreb, D. „The ObjectStore Database System". *Communications of the ACM*, Jg. 34, 1992, Nr. 10, S. 50–64.

Lampson, Sturgis 76: Lampson, B. und Sturgis, H. „Crash Recovery in a Distributed Data Storage System". Technical report, Xerox Palo Alto Research Center, California, 1976.

Lindsay 79: Lindsay, B. „Notes on Distributed Databases". Technical Report Research Report RJ2517, IBM, Juli 1979.

Maier et al. 86: Maier, D., J., Stein, A., Otis, und Purdy, A. „Development of an Object-Oriented DBMS". In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, Oktober 1986.

Mathiske et al. 93: Mathiske, B., Matthes, F., und Müßig, S. „The Tycoon System and Library Manual". DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

Mathiske et al. 95: Mathiske, B., Matthes, F., und Schmidt, J.W. „On Migrating Threads". In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, Juni 1995. (to appear).

Matthes et al. 92: Matthes, F., Müller, R., und Schmidt, J.W. „Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience". FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1992.

Matthes, Schmidt 92: Matthes, F. und Schmidt, J.W. „Definition of the Tycoon Language TL – A Preliminary Report". Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes, Schmidt 93: Matthes, F. und Schmidt, J.W. „System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways". In: Spies, P.P. (Hrsg.), *Proceedings of Euro-Arch'93 Congress*. Springer-Verlag, Oktober 1993, S. 301–317.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German.).

McCarthy, Dayal 89: McCarthy, D. und Dayal, U. „The Architecture of an Active Database Management System". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, 1989.

McCarthy, Sarin 93: McCarthy, D. und Sarin, S. „Workflow and Transaction in InConcert". *IEEE Data Engineering Bulletin*, Jg. 16, 1993, Nr. 2.

Mohan et al. 92: Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., und Schwarz, P. „ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". *ACM Transactions on Database Systems*, Jg. 17, Maerz 1992, Nr. 1.

Mohan, Lindsay 83: Mohan, C. und Lindsay, B. „Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions". Technical Report Research Report RJ3881, IBM, Maerz 1983.

Moss 82: Moss, J.E.B. „Nested Transactions and Reliable Distributed Computing". In: *Symposium on Reliability in Distributed Software and Database Systems*, 1982, S. 33–39.

Niederée 92: Niederée, C. „Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

OSICCR 89: „Open Systems Interconnection: Commit, Concurrency Control and Recovery (OSI-CCR); Model: ISO IS 9804-1, Service Definition: ISO IS 9804-2, Protocol Specification: ISO IS 9804-3". New York: ANSI, 1989.

OSITP 92: „Open Systems Interconnection: Distributed Transaction Processing (OSI-TP); Model: ISO IS 10026-1, Service Definition: ISO IS 10026-2, Protocol Specification: ISO IS 10026-3". New York: ANSI, 1992.

Özsu, Valduriez 91: Özsu, M.T. und Valduriez, P. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Sheth, Rusinkiewicz 93: Sheth, A. und Rusinkiewicz. „On Transactional Workflows". *IEEE Data Engineering Bulletin*, Jg. 16, 1993, Nr. 2.

Stard93 93: StarView C++ Class Library, Version 2.0. Postfach 2830, D-2120 Lüneburg, FRG, 1993.

Stonebraker et al. 87: Stonebraker, M., Hanson, E., und Hong, C. „The Design of the POSTGRES Rule System". In: *Proceedings of the IEEE Third International Conference on Data Engineering, Los Angeles, California*, 1987.

Wächter, Reuter 91: Wächter, H. und Reuter, A. „The ConTract Model". In: Elmagarmid, A.K. (Hrsg.), *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1991, S. 219–263.

White, DeWitt 95: White, S. und DeWitt, D. „Implementing Crash Recovery in QuickStore: A Performance Study". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Jose, California*, 1995.

X/Open Ltd. 92: X/Open Ltd. *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*. X/Open Company Ltd., U.K., Berkshire, England, Oktober 1992.

X/Open Ltd. 93a: X/Open Ltd. *Distributed Transaction Processing Reference Model Version 2*. X/Open Company Ltd., U.K., Berkshire, England, November 1993.

X/Open Ltd. 93b: X/Open Ltd. *Distributed Transaction Processing: The TxRPC Specification*. X/Open Company Ltd., U.K., Berkshire, England, Juli 1993.

X/Open Ltd. 93c: X/Open Ltd. *Distributed Transaction Processing: The XA+ Specification*. X/Open Company Ltd., U.K., Berkshire, England, Maerz 1993.

Hiermit versichere ich, die vorliegende Arbeit ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel selbständig angefertigt zu haben.

Hamburg, den 22. August 1995

Marcel Kornacker