Konzeption und Implementierung des Kommunikationsmanagers für das Workflow-Management-System MENTOR

-

Integration und Synchronisation der verteilten Workflow Engine

Ralf Schenkel

Diplomarbeit in Informatik
am Lehrstuhl Prof. Weikum der Universität des Saarlandes, Saarbrücken

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Arbeit vollständig selbst und nur unter Verwendung der im Literaturverzeichnis angegebenen Quellen erstellt habe.

Saarbrücken, im September 1997

Ich möchte mich bei Prof. Dr.-Ing. Gerhard Weikum bedanken, daß er es mir ermöglicht hat, dieses interessante Thema im Rahmen einer Diplomarbeit zu bearbeiten. Besonderer Dank gilt Dirk Wodtke für die vorbildliche und engagierte Betreuung und für die vielen Anregungen und Verbesserungsvorschläge. Allen Assistenten und Studenten am Lehrstuhl verdanken wir eine sehr gute Arbeitsatmosphäre.

Inhaltsverzeichnis

Inhaltsverzeichnis **4**

1. Motivation und Einordnung **7**
1.1. Was ist Workflow Management? 7
1.2. Einordnung der Arbeit ins MENTOR-Projekt 7
1.2.1. Zielsetzung des MENTOR-Projektes 7
1.2.2. Spezifikation von Workflows mit State- und Activitycharts 7
1.2.3. Konzepte zur Erreichung von Skalierbarkeit und Fehlertoleranz 12
1.2.4. Beschreibung der Ausführungsumgebung 13
1.3. Zielsetzung und Beitrag der Arbeit 15
1.4. Überblick über die Arbeit 15

2. Konzeption des Kommunikationsmanagers **16**
2.1. Anforderungen an den Kommunikationsmanager 16
2.1.1. Korrekte Ausführung des Workflows 16
2.1.2. Erkennen und Behandeln von Fehlern 16
2.1.3. Effizienz 16
2.1.4. Integration in das MENTOR-Gesamtsystem 17
2.2. Konzeption des Kommunikationsmanagers 17
2.2.1. Schichtenarchitektur 17
2.2.2. Die Workflow-Engine-Schicht 19
2.2.3. Die Communication-Management-Schicht 19
2.2.4. Die Workflow-Engine-Interface-Schicht 20
2.2.5. Die Data-Transport-Schicht 21
2.2.6. Die Data-Transport-Interface-Schicht 21
2.3. Konfigurationsprofil für Workflows 23
2.3.1. Identifikation eines verteilten Workflows 23
2.3.2. Workflow-Konfigurationsdatei 23
Der Workflow-Block 23
Der Role-Block 24
Der Entity-Block 25
Der Prozeß-Block 25
2.3.3. Datenbankrelationen für die Konfiguration 25

3. CM-Schicht und WEI-Schicht **27**
3.1. Konzeption der CM-Schicht 27
3.1.1. Abstraktion von den verwendeten Systemkomponenten 27
3.1.2. Aufbau der CM-Schicht aus Synchronisationsphase, Lesephase, Ausführungsphase und Schreibphase 28
3.1.3. Aufbau der Lesephase 28
3.1.4. Aufbau der Ausführungsphase 31
3.1.5. Aufbau der Schreibphase 32
3.1.6. Integration des Logmanagers 33
3.1.7. Starten von Aktivitäten in nicht aktiven Partitionen 34
3.1.8. Beenden einer Partition 35
3.2. Implementierung der CM-Schicht 36
3.2.1. Verwaltung der Variablen, Aktivitäten und Zustände 36

3.2.2. Grundsätzliche Codierung der CM-Schicht 38
3.2.3. Implementierung der Lesephase 39
3.2.4. Implementierung der Ausführungsphase 43
3.2.5. Implementierung der Schreibphase 44
3.2.6. Initialisierung des Kommunikationsmanagers 45
3.2.7. Beenden einer Partition 46
3.3. Implementierung der WEI für Statemate 47
3.3.1. Verwaltung der Variablen, Aktivitäten und Zustände 47
3.3.2. Implementierung von Callbacks 49
3.3.3. Implementierung der Änderungsfunktionen 50
3.3.4. Ausführen eines lokalen Schritts 50
3.4. Implementierung der WEI für den Chart Interpreter 51
3.4.1. Variablenverwaltung 52
3.4.2. Implementierung von Callbacks 52
3.4.3. Implementierung der Änderungsfunktionen 53
3.4.4. Ausführen eines lokalen Schritts 53
3.5. Diskussion von Entwurfsalternativen 53

4. Synchronisation 55
4.1. Notwendigkeit von Synchronisation 55
4.2. Strikte Synchronisation 56
4.3. Dynamische Synchronisation 58
4.4. Implementierung 61
4.5. Probleme bei der Synchronisation 64
4.6. Weitere Alternativen 66
4.6.1. Implizite Synchronisationsnachrichten 66
4.6.2. Verbesserung dynamischer Synchronisation 66
4.6.3. Synchronisation an Synchronisationspunkten 66
4.7. Das Starten von nicht aktiven Partitionen und das Beenden aktiver Partitionen . . 68
4.7.1. Der Algorithmus zur Einbindung von Partitionen 68
4.7.2. Der Algorithmus zum Beenden von Partitionen 70
4.7.3. Implementierungsaspekte 71
4.7.3.1. Startmaßnahmen bei dynamisch gestarteten Partitionen 71
4.7.3.2. Ergänzungen in der CM-Schicht 71
4.7.3.3. Implementierung des Dynamischen Prozeßmanagers 73
4.7.4. Optimierungsmöglichkeiten 73

5. Evaluation 75
5.1. Vergleich der Workflow Engines 75
5.1.1. Meßumgebung 75
5.1.2. Versuchsaufbau 75
5.1.3. Meßergebnisse 76
5.1.4. Auswertung der Meßergebnisse 78
5.2. Meßumgebung für die folgenden Messungen 78
5.2.1. Performance Monitoring 78
5.2.2. Aktivitätssimulation 80
5.2.3. Meßworkflow 81
5.3. Vergleich der Synchronisationsmethoden 81
5.4. Quantitativer Vergleich verschiedener Konfigurationen 85

6. Zusammenfassung und Ausblick 89
6.1. Zusammenfassung 89
6.2. Ausblick 89

A. Dateienstruktur des Kommunikationsmanagers 91

B. Struktur- und Funktionsdefinitionen zur Verwaltung der Konfiguration eines Workflows 92

C. Funktionsübersicht des Kommunikationsmanagers 96

D. Literaturverzeichnis 97

1. Motivation und Einordnung

1.1. Was ist Workflow Management?

Ein *Workflow* (Arbeitsablauf) ist die koordinierte Ausführung einer Menge zusammengehöriger Arbeitsschritte in einer verteilten Arbeitsumgebung [RS94]. Die einzelnen Arbeitsschritte eines Workflows werden von spezifischen *Ausführungsorganen* bearbeitet; dies können menschliche Sachbearbeiter und Entscheidungsträger wie auch Computersysteme oder eine Kombination davon sein. Ausführungsorgane übernehmen bezüglich der Arbeitszuteilung sogenannte *Rollen*; Ausführungsorgane in derselben Rolle sind austauschbar. Die fehlertolerante Ausführung solcher Arbeitsabläufe in einer unternehmensweit verteilten und hochgradig heterogenen Informationssystemlandschaft übernehmen Workflow-Management-Systeme. Außerdem sollte ein Workflow-Management-System die computergestützte Spezifikation, Verifikation, Steuerung und Überwachung von Workflows unterstützen.

Praktische Anwendungsgebiete für Workflow-Management-Systeme finden sich auf dem Dienstleistungssektor und in öffentlichen Verwaltungen. Beispiele für den Dienstleistungssektor sind Banken (Kreditvergabe, Kontoeröffnung, Akkreditivbearbeitung, Aktienemission etc.) und Versicherungen (Vertragsbearbeitung, Schadensfallbearbeitung, etc.), für die öffentliche Verwaltung sind es zum Beispiel Genehmigungsverfahren im Bauwesen und die Bearbeitung einer Steuererklärung.

1.2. Einordnung der Arbeit ins MENTOR-Projekt

1.2.1. Zielsetzung des MENTOR-Projektes

MENTOR ("Middleware for Enterprise-Wide Workflow Management") ist ein gemeinsames Projekt der Universität des Saarlandes, der Schweizerischen Bankgesellschaft (SBG) und der Eidgenössischen Technischen Hochschule (ETH) Zürich, das sich mit unternehmensweitem Workflow-Management beschäftigt. Das Projekt zielt auf die Entwicklung einer skalierbaren, hochgradig verfügbaren und fehlertoleranten Umgebung zur Ausführung und Überwachung von Workflows, die nahtlos in eine Spezifikations- und Verifikationsumgebung eingebunden ist.

1.2.2. Spezifikation von Workflows mit State- und Activitycharts

In MENTOR werden Workflows formal auf der Basis von State- und Activitycharts modelliert [WMS+95]. Workflows, die mit anderen Methoden spezifiziert wurden, können weitgehend automatisch auf State- und Activitycharts abgebildet werden [Wod96]. Wie in [WWK+97] beschrieben werden State- und Activitycharts für das gesamte Spektrum von der Erstellung der Workflow-Spezifikation über die Verifikation kritischer Eigenschaften bis hin zur Ausführung in einer verteilten Umgebung eingesetzt. Die Spezifikationsmethode der State- und Activitycharts wurde von David Harel zur Beschreibung von reaktiven Systemen entwickelt [Har87, HLN+90]. Sie kombiniert die Einfachheit und mathematische Rigorosität von Automatenmodellen mit Möglichkeiten zur Visualisierung von Spezifikationen und hat gleichzeitig eine mit Prädikattransitionsnetzen vergleichbare Ausdrucksmächtigkeit.

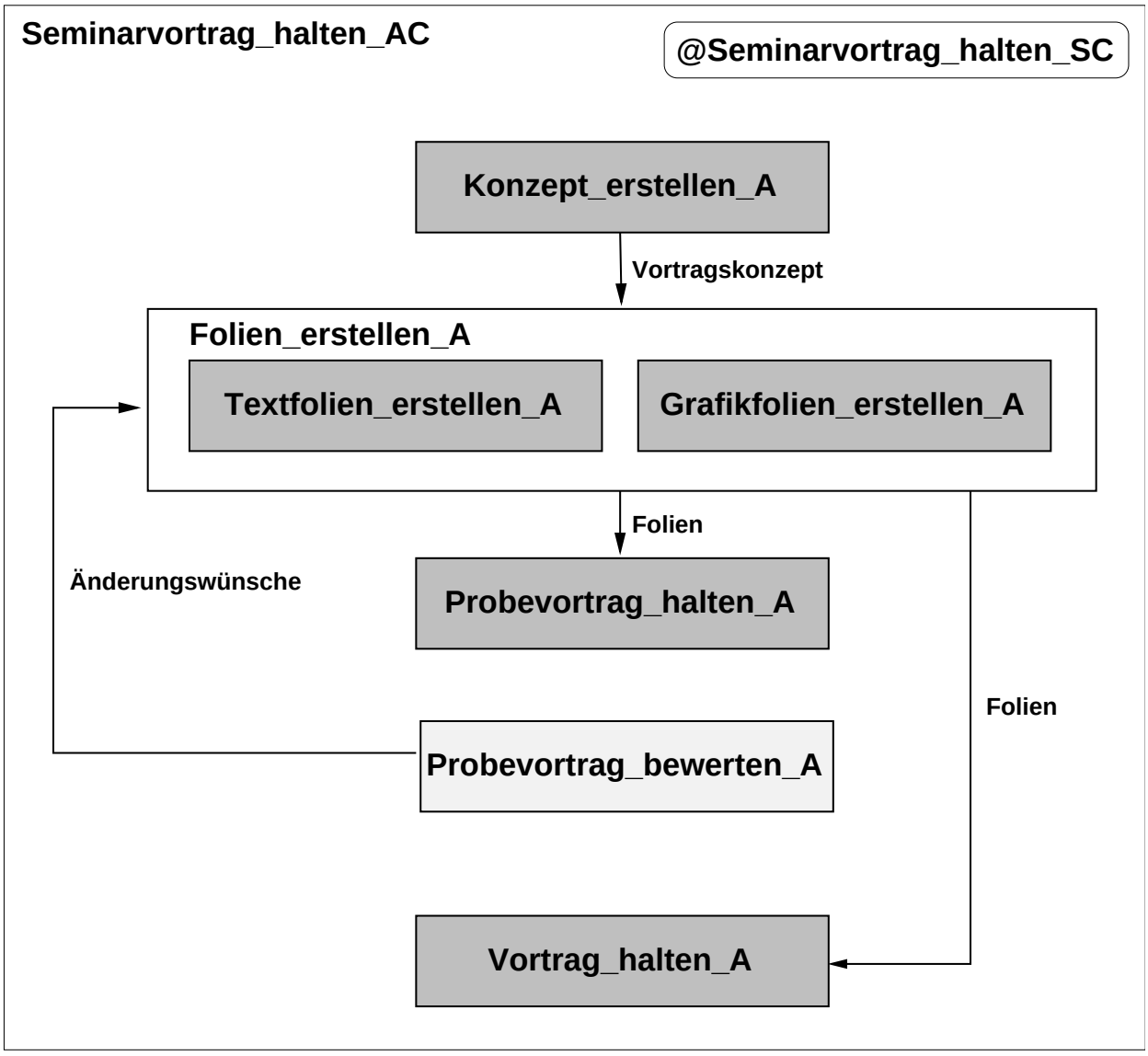

Abbildung 1 - Activitychart des Workflows “Halten eines Seminarvortrags”

State- und Activitycharts stellen die Spezifikation eines Workflows aus zwei verschiedenen Sichtweisen dar. Activitycharts beschreiben dabei den Workflow aus funktionaler Sicht, d.h. die einzelnen Arbeitsschritte des Workflows - auch Aktivitäten genannt - und den Datenfluß zwischen ihnen. Zu jeder Aktivität wird bestimmt, zu welcher Ausführungsrolle die Aktivität gehört. Ein Beispiel für ein Activitychart ist in Abbildung 1 dargestellt. Dieses Activitychart gehört zum Workflow "Halten eines Seminarvortrags". Die Aktivitäten der beiden Rollen "Student" und "Betreuer" sind durch unterschiedliche Graustufen kenntlich gemacht. Der Student erstellt dabei zunächst ein Konzept (`Konzept_erstellen_A`), danach fertigt er Text- und Grafikfolien an (`Textfolien_erstellen_A` und `Grafikfolien_erstellen_A`). Nach dem Halten eines Probevortrages (`Probevortrag_halten_A`) wird dieser durch den Betreuer bewertet (`Probevortrag_bewerten_A`). Wenn die Bewertung negativ ausfällt, muß der Student die Text- und Grafikfolien überarbeiten, ansonsten kann er den Vortrag halten (`Vortrag_halten_A`).

Statecharts beschreiben das Verhalten einer Spezifikation, indem sie den Steuerfluß zwischen Aktivitäten als Transitionen zwischen Zuständen spezifizieren. Jede Aktivität ist dabei genau einem Zustand zugeordnet. Abbildung 2 zeigt das zu dem Activitychart aus Abbildung 1 gehörende Statechart. Diese Zuordnung wird im Activitychart durch die ausgezeichnete Kontrollaktivität

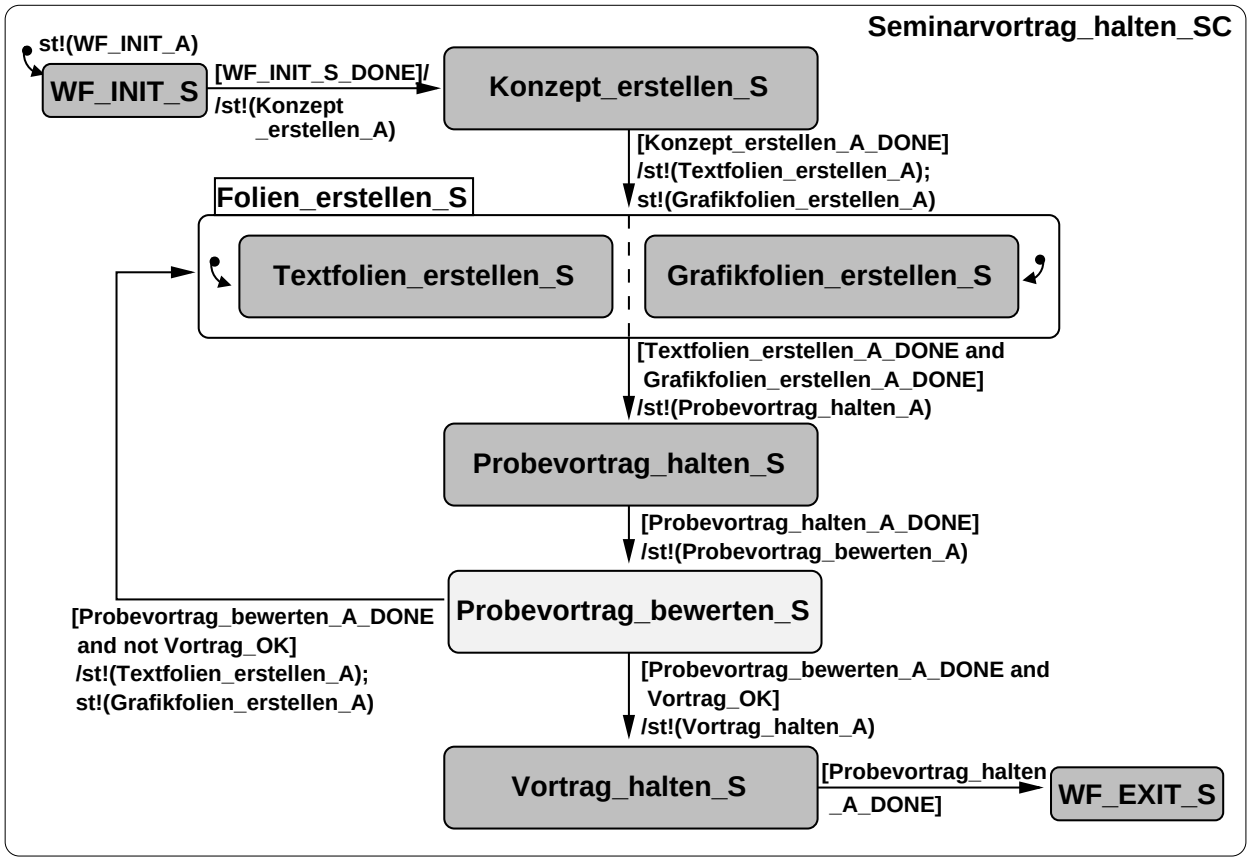

Abbildung 2 - Statechart des Workflows “Halten eines Seminarvortrags”

`@Seminarvortrag_halten_SC` ausgedrückt. Die Schaltvoraussetzungen werden in Form sogenannter *Event-Condition-Action-Regeln*, kurz ECA-Regeln, formuliert. Eine Transition mit der Beschriftung `E[C]/A` schaltet genau dann, wenn ihr Quellzustand, also der Zustand, von dem sie ausgeht, betreten ist, das Ereignis `E` generiert worden ist und die Bedingung `C` erfüllt ist. Beim Schalten der Transition wird die Aktion `A` ausgeführt, wobei Aktionen u.a. der Start einer Aktivität (z.B. `st!(Probevortrag_halten_A)`) oder das Generieren eines Ereignisses sein können. Transitionen, die keinen Quellzustand haben, kennzeichnen initiale Zustände, zum Beispiel `Konzept_erstellen_S`. Neben Ereignissen und Bedingungen können bei der Spezifikation auch Variablen, sogenannte *Data-Items*, verwendet werden. Data-Items enthalten entweder ganze Zahlen, Fließkommazahlen oder Zeichenketten, sind also typisiert.

Zustände können ihrerseits Zustände enthalten, bieten also die Möglichkeit der Schachtelung von Zuständen. Außerdem kann ein Zustand in orthogonale Komponenten aufgeteilt werden; dies sind Unterzustände, die parallel zueinander ausgeführt werden. Im Beispiel aus Abbildung 2 enthält der Zustand `Folien_erstellen_S` zwei durch eine gestrichelte Linie voneinander getrennte orthogonale Komponenten. Zwei ausgezeichnete Zustände, `WF_INIT_S` und `WF_EXIT_S`, dienen der Initialisierung bzw. dem Erkennen der Beendigung des Workflows.

Die Abbildungen 3 und 4 zeigen Ausschnitte aus der Beschreibung des Workflows zur Erstellung eines universitären Lehrberichts. Die Charts des Workflows wurden automatisch erzeugt. Die ursprüngliche Beschreibung des Workflows lag als im ARIS Toolset [SJ96] eingegebene ereignisgesteuerte Prozeßkette (EPK) vor und wurde vom Institut für Wirtschaftsinformatik der Universität des Saarlandes zur Verfügung gestellt. Da schon die Originalspezifikation über keine Datenflußinformationen verfügt, enthält auch das erzeugte Activitychart keinen Datenfluß.

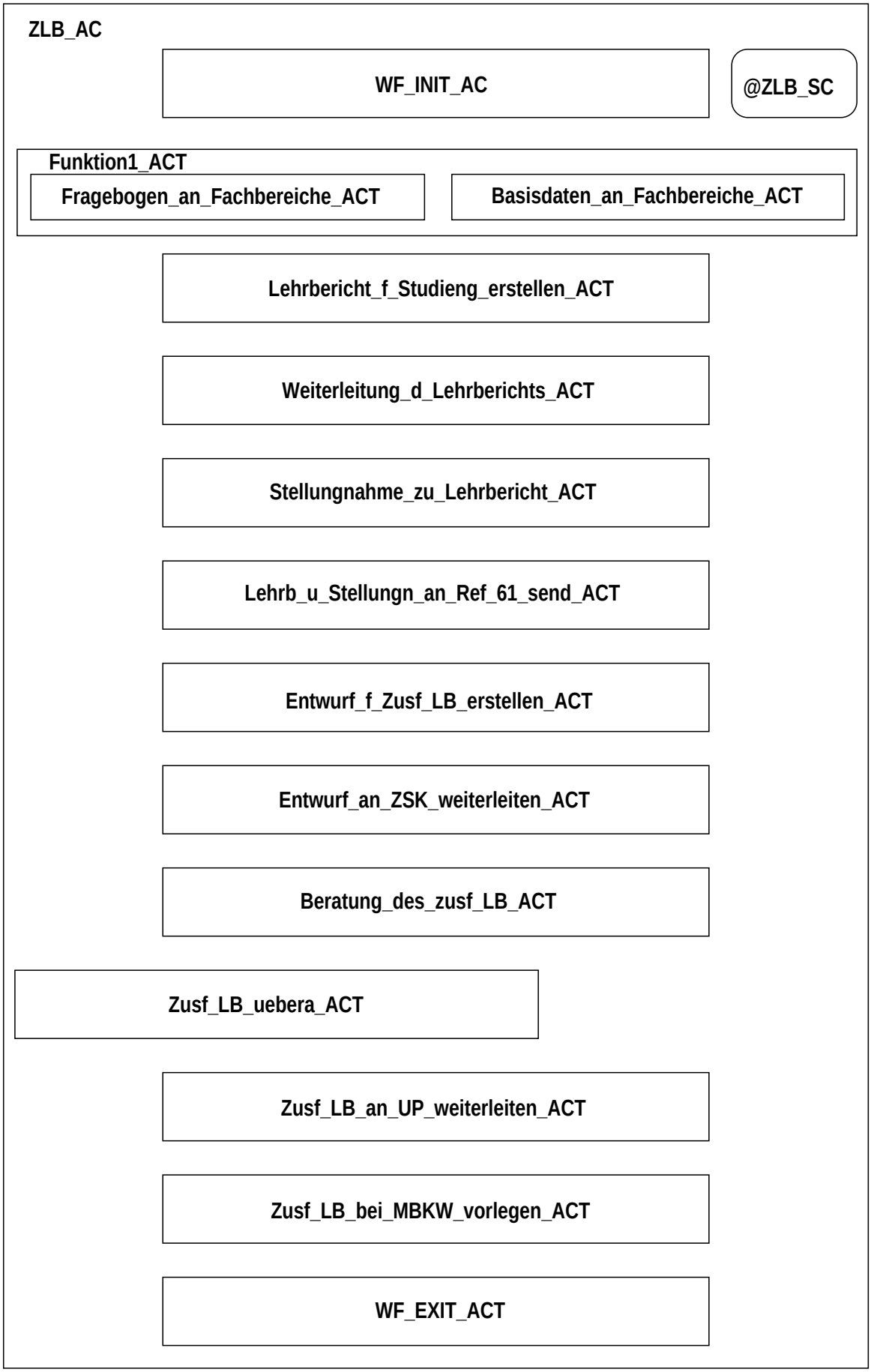

Abbildung 3 - Ausschnitt aus dem Activitychart des Workflows “Erstellen eines Lehrberichts”

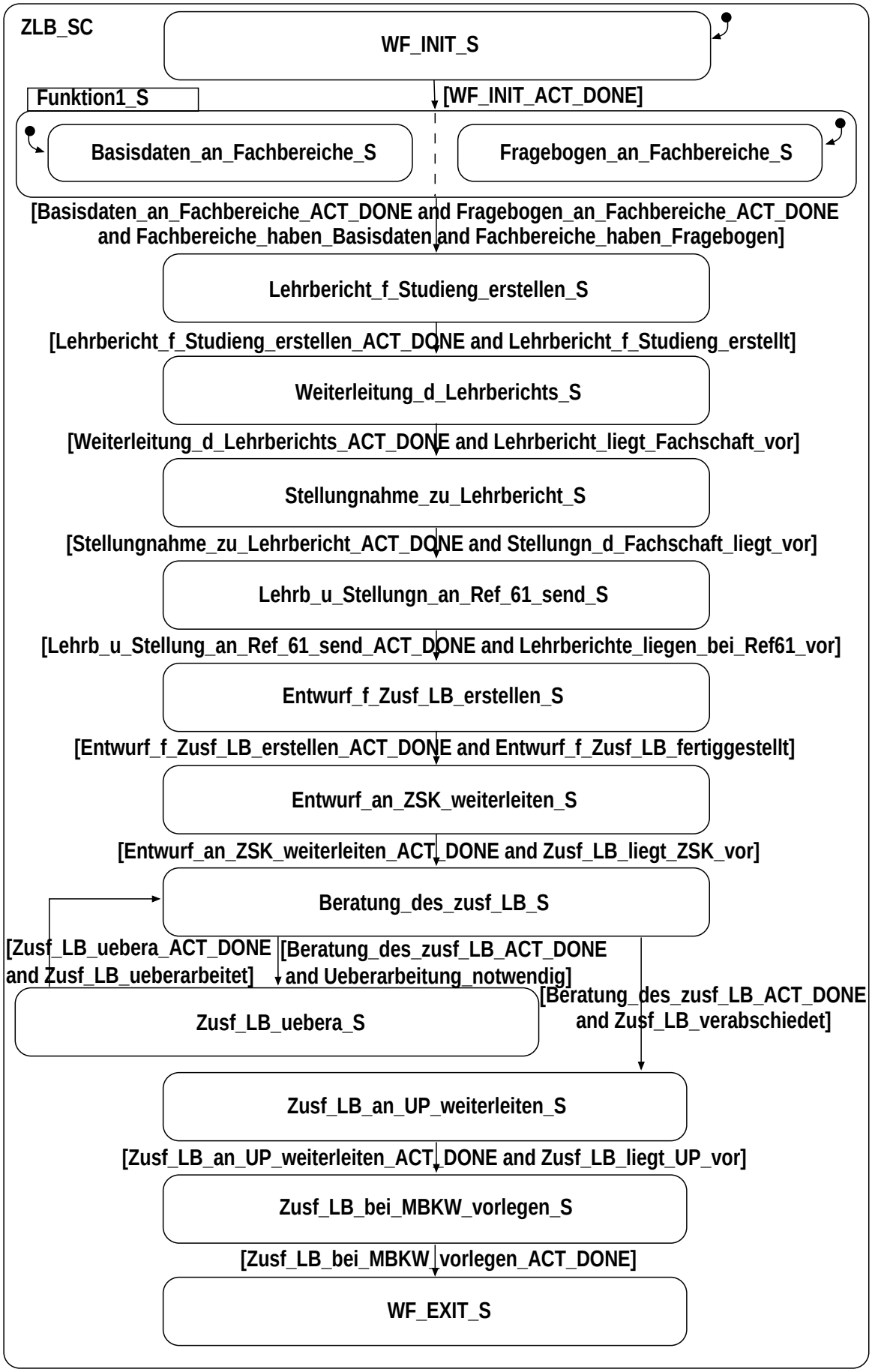

Abbildung 4 - Ausschnitt aus dem Statechart des Workflows “Erstellen eines Lehrberichts”

Sogar wesentlich aufwendigere Geschäftsprozesse lassen sich mit der Methode der State- und Activitycharts spezifizieren. In [Bie97] wird zum Beispiel gemäß den Anforderungen einer Bank ein realistischer Workflow für die Vergabe eines Kredits mit State- und Activitycharts spezifiziert. Dieser enthält 110 Aktivitäten, 434 Zustände, 266 Transitionen, 402 Bedingungen, 154 Ereignisse und 36 Data-Items. Das zeigt die Tauglichkeit von State- und Activitycharts für den Entwurf von großen Workflows in einem praktischen Anwendungskontext.

1.2.3. Konzepte zur Erreichung von Skalierbarkeit und Fehlertoleranz

In der praktischen Anwendung von Workflow-Management-Systemen werden in der Regel massenhaft viele und/oder auch komplexe Workflows gleichzeitig ausgeführt. Es darf dabei nicht zu überlangen Antwortzeiten für die Benutzer des Systems kommen, die zum Beispiel durch Rechnerüberlastung entstehen. Ein geeignetes Mittel zur Vermeidung solcher Überlastungen ist die Verteilung der Workflow-Ausführung auf mehrere Rechner und die Vermeidung zentralisierter Systemkomponenten in der Ausführungsumgebung. Aus diesem Grunde wurde das MENTOR-System *skalierbar* angelegt, d.h. das Gesamtsystem wird in Komponenten aufgeteilt, die nur die für sie notwendigen Informationen erhalten und nur bei Bedarf miteinander kommunizieren. Das System kann so bei steigender Last inkrementell wachsen.

Die Spezifikation eines Workflows wird zunächst zentralisiert in Form von State- und Activitycharts erstellt. Um den Workflow verteilt ausführen zu können, muß die Spezifikation anschließend *partitioniert* werden, d.h. in mehrere Teilspezifikationen, *Partitionen* genannt, aufgeteilt werden. Diese Partitionen bestehen aus orthogonalen Zustandskomponenten, die aus dem Original-Statechart erzeugt wurden und die dann auf verschiedenen Rechnern ausgeführt werden. Der Algorithmus zur Erzeugung dieser Partitionen ist in [Wod96] beschrieben worden. Informationen über Änderungen, die bei der Ausführung einer Partition auftreten, wie zum Beispiel das Betreten und Verlassen von Zuständen oder das Verändern von Variablen, werden durch den Kommunikationsmanager an die übrigen Partitionen propagiert.

Abbildung 5 zeigt eine Partition des vorher gezeigten Lehrbericht-Workflows. Die Zustände auf der rechten Seite der Abbildung enthalten Unterzustände, die schon im Original-Statechart aus Abbildung 4 vorhanden sind, dort aber aus Gründen der Übersichtlichkeit weggelassen wurden. Sie dienen der Steuerung der zugeordneten Aktivitäten. Die Struktur dieser Zustände wird in [Bie97] beschrieben und ist für diese Arbeit nicht weiter relevant.

Neben der Überlastung von Rechnern ist der Ausfall von einzelnen Rechnern oder Netzverbindungen ein weiteres Problem bei großen Anwendungen. In MENTOR wird diesem Problem durch Einsatz von geeigneten Middleware-Komponenten zur Erreichung von Fehlertoleranz begegnet. Andere Arbeiten, die sich mit fehlertolerantem Workflow-Management beschäftigen, sind [AKA+94], [KAGM96] und [Wä97].

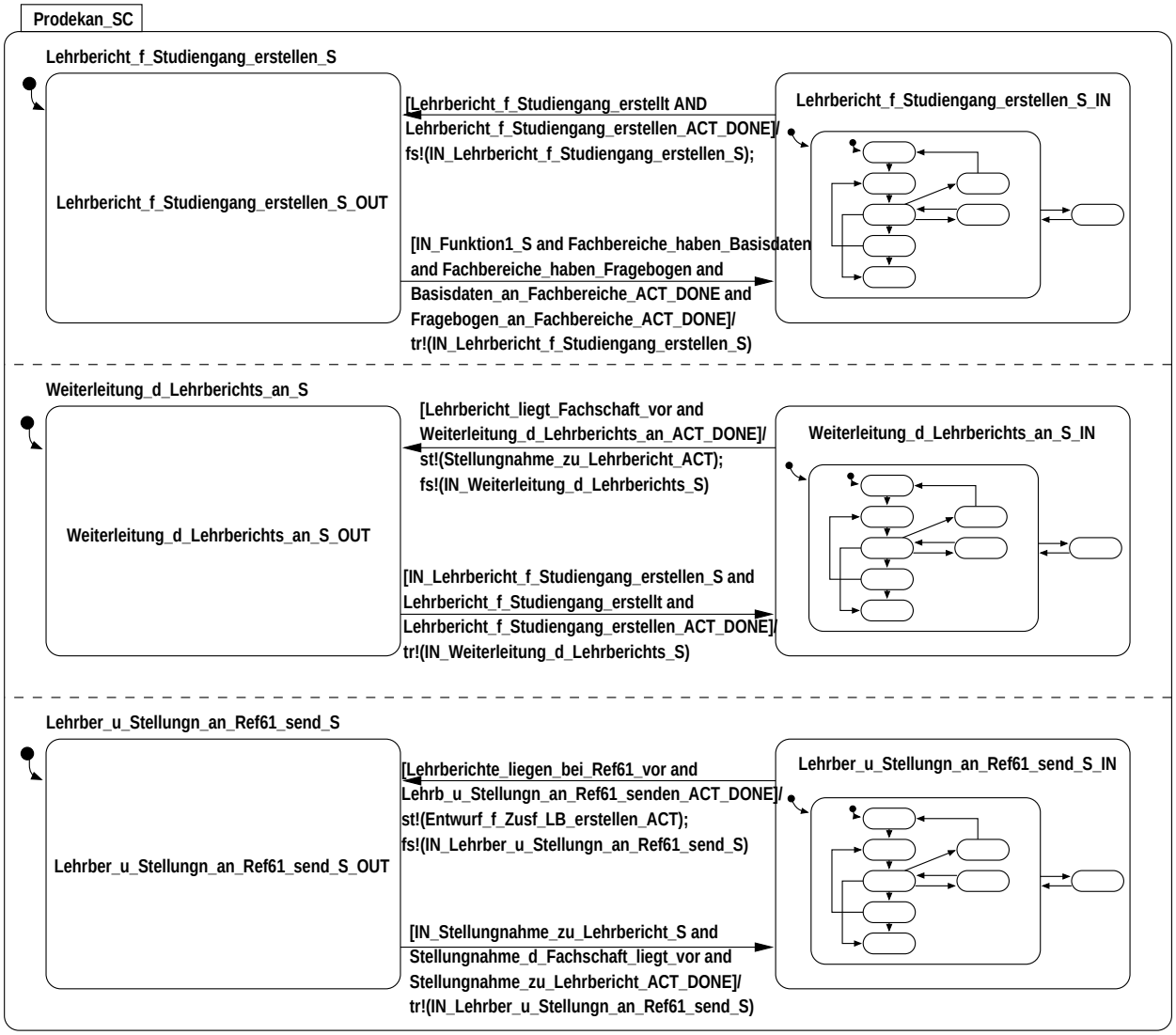

Abbildung 5 - Eine Partition des Workflows “Erstellen eines Lehrberichts”

1.2.4. Beschreibung der Ausführungsumgebung

In Abbildung 6 wird die Architektur einer verteilten Ausführungsumgebung, die in MENTOR realisiert wurde, gezeigt. Dargestellt sind exemplarisch zwei *Workflow-Server-Rechner*. Auf einem solchen Workflow-Server-Rechner arbeiten die folgenden Komponenten zusammen:

- Die *Workflow-Engine* dient der Ausführung einer Workflow-Partition.

- Eine Implementierung von *CORBA* (standardisierte und objektorientierte Ausführungsumgebung, mit der es möglich ist, transparent heterogene Applikationsprogramme aufzurufen) sorgt für die Anbindung von externen Applikationen wie zum Beispiel Textverarbeitungen und Tabellenkalkulationen an den Workflow.

- Der *Log-Manager* ([Klä97], [Sti97]) zeichnet alle Änderungen von Zuständen, Ereignissen, Bedingungen und kontrollfluß-relevanten Variablen in einer Log-Datenbank unter Zuhilfenahme eines TP-Monitors auf, um im Falle eines Fehlers die bereits ausgeführten Schritte der Workflow-Engine bis zum letzten konsistenten Schritt wiederholen zu können.

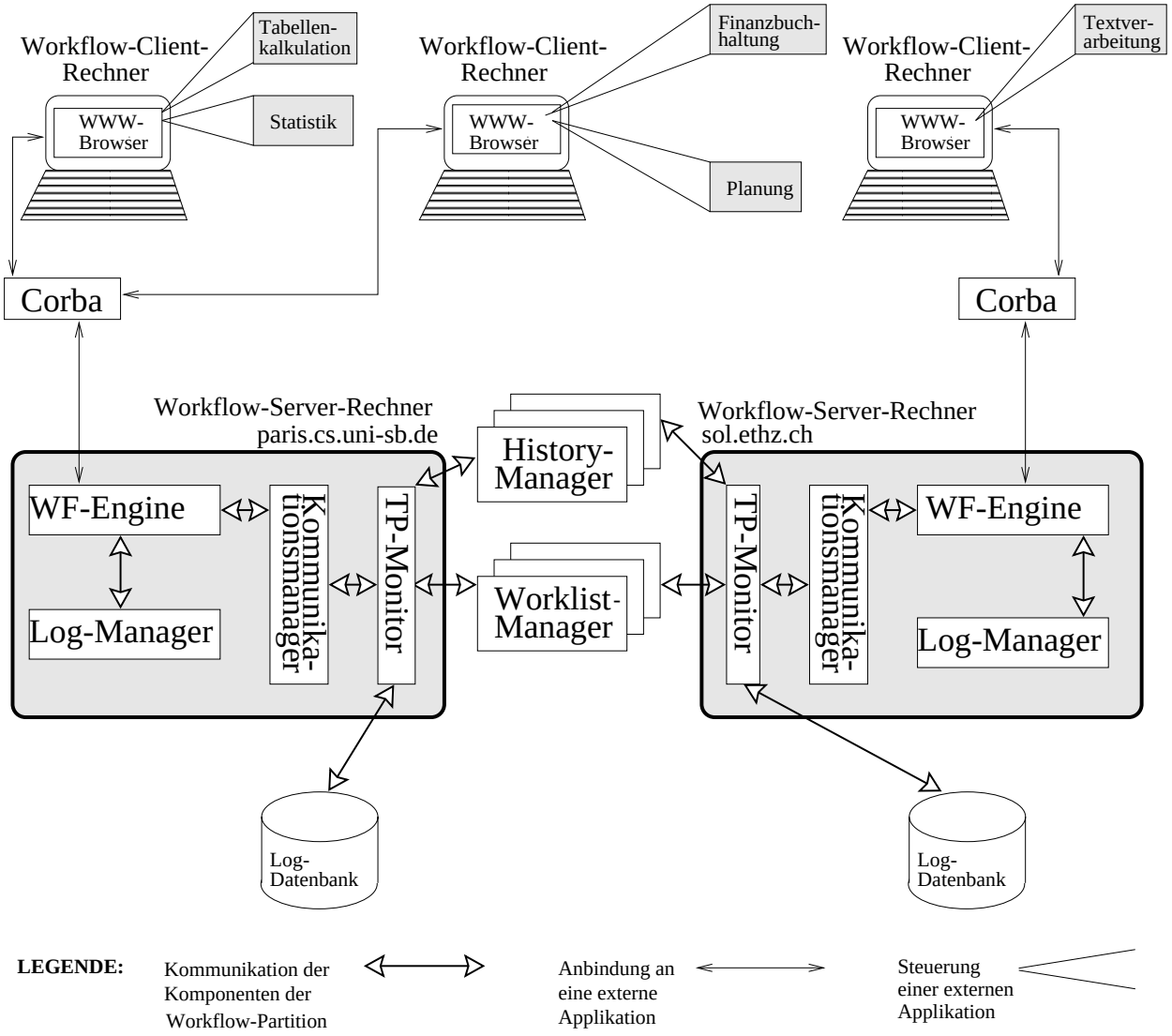

Abbildung 6 - Architektur der verteilten Ausführungsumgebung in MENTOR

- Der Kommunikationsmanager ist Teil der Ausführungsumgebung jeder Workflow-Partition. Er propagiert die genannten Änderungen an die übrigen Workflow-Server-Rechner, auf denen Partitionen dieses Workflows ausgeführt werden. Er bedient sich dazu des TP-Monitors. Dieser stellt zum einen *persistenten Speicher* in Form von *Reliable Message Queues*, zum anderen verteilte Transaktionen zur Verfügung. Durch den persistenten Speicher bleiben Nachrichten auch über Fehlersituationen hinweg erhalten; durch die verteilten Transaktionen wird sichergestellt, daß im Fehlerfall keine inkonsistenten Informationen an die übrigen Partitionen geschickt werden. Außerdem kann man den TP-Monitor in ein Konzept zur Replikation von Daten einbinden.

Ferner gibt es die folgenden Komponenten, die sowohl auf den Workflow- als auch auf anderen Server-Rechnern ablaufen können:

- Der History-Manager ([Klä97], [Sti97]) erlaubt zu Überwachungszwecken das Sammeln von workflow-weiten Informationen in einer Workflow-History-Datenbank, wie zum Beispiel Zustandsinformationen über laufende sowie bereits beendete Workflows. Auf dieser Datenbank können dann Statusabfragen und Langzeitanalysen durchgeführt werden.

- Der Worklist-Manager [Hof97] weist einer zur Bearbeitung anstehenden Aktivität eines der Ausführungsorgane zu, die die zugehörige Rolle innehaben. Er berücksichtigt bei dieser Auswahl zum Beispiel die Auslastung der einzelnen Ausführungsorgane.

1.3. Zielsetzung und Beitrag der Arbeit

Das Ziel dieser Doppel-Diplomarbeit ist es, eine Konzeption für den MENTOR-Kommunikationsmanager zu entwerfen und einen lauffähigen Prototypen zu implementieren. Die Diplomarbeit von Mark Wehrmann [Weh97] beschäftigt sich dabei mit Integration und Administration des Kommunikationsmanagers mit weiteren Systemkomponenten unter Benutzung des TP-Monitors Tuxedo. In der vorliegenden Diplomarbeit werden die Anbindung des Kommunikationsmanagers an die Workflow-Engine und Konzepte zur Synchronisation der Workflow-Engines, um eine korrekte Ausführung des Workflows zu gewährleisten, vorgestellt.

Die entwickelten Konzepte wurden prototypisch implementiert. Als Implementierungsbasis dienten Sun Sparcstations mit dem Betriebssystem Solaris in den Versionen 2.3 bis 2.5.1. Die als Code-Generator für die Workflow-Engine fungierende Spezifikationssoftware Statemate lag in Version 5.3 und der TP-Monitor Tuxedo in Version 5.0 vor. Als Datenbanksystem kam Oracle 7.2 zum Einsatz. Für die Compilation der Programme wurde der Gnu-C-Compiler in der Version 2.7.2 und der Gnu-C-Debugger in der Version 4.12 benutzt.

1.4. Überblick über die Arbeit

Im weiteren Verlauf dieser Arbeit folgt zunächst ein Kapitel, das mit dem Kapitel 2 der Diplomarbeit von Mark Wehrmann [Weh97] identisch ist und in dem die Konzeption des Kommunikationsmanagers vorgestellt wird. Kapitel 3 beschreibt Konzeption und Implementierung der Communication-Management-Schicht des Kommunikationsmanagers und die Anbindung der Workflow-Engines Statemate und Chart Interpreter. Anschließend wird in Kapitel 4 ein Konzept zur Synchronisation von verteilt laufenden Workflows dargestellt und seine Implementierung beschrieben. In Kapitel 5 werden verschiedene quantitative Betrachtungen und Messungen am implementierten Prototypen durchgeführt und die Ergebnisse interpretiert. Kapitel 6 faßt die gewonnenen Erkenntnisse zusammen und enthält einen Ausblick auf mögliche Erweiterungen des Kommunikationsmanagers für MENTOR.

2. Konzeption des Kommunikationsmanagers

2.1. Anforderungen an den Kommunikationsmanager

2.1.1. Korrekte Ausführung des Workflows

Der Kommunikationsmanager muß gewährleisten, daß Workflows in der verteilten Ausführungsumgebung korrekt ausgeführt werden. Die verteilte Ausführung muß also verhaltens-äquivalent zur zentralen Ausführung sein, das heißt bei der verteilten Ausführung müssen sich genau die Änderungen ergeben, die bei einer zentralen Ausführung auftreten würden. Dazu müssen insbesondere die folgenden Anforderungen erfüllt sein:

- Die Änderungen, die sich bei der Ausführung eines Schritts einer Partition ergeben, müssen an andere Partitionen propagiert werden. Änderungen in diesem Sinne sind Variablenänderungen, das Betreten und Verlassen von Zuständen, sowie das Starten und Beenden von Aktivitäten.
- Alle Änderungen eines Schrittes während der Statechart-Ausführung müssen atomar an die jeweiligen Empfänger-Partitionen weitergeleitet werden, damit kein inkonsistenter Zustand erreicht wird. Das heißt entweder werden alle Änderungen propagiert oder - im Fehlerfall - überhaupt keine.

2.1.2. Erkennen und Behandeln von Fehlern

Der Kommunikationsmanager muß in der Lage sein, Fehler die bei der Kommunikation auftreten, zu erkennen und zu behandeln. So soll zum Beispiel der Ausfall von Rechnern erkannt werden, wobei der Ausfall einer Komponente nicht zum Ausfall des Gesamtsystems führen darf. Vielmehr soll, nachdem die ausgefallene Teilkomponente oder eine Ersatzkomponente zur Verfügung steht, der Zustand vor dem Ausfall wiederhergestellt werden. Dafür muß es eine Schnittstelle zu einem Log-Manager geben, der während der Ausführung des Workflows den aktuellen Systemzustand (aktuell betretene Zustände, Belegung der Variablen und gestartete Aktivitäten) protokolliert hat. In anderen Fällen, wie zum Beispiel dem temporären Ausfall von Netzverbindungen, soll der Kommunikationsmanager fehlertolerant reagieren, also nach dem Erkennen eines solchen Ausfalls Maßnahmen zur Fehlerumgehung einleiten.

2.1.3. Effizienz

Im Hinblick auf den praktischen Einsatz von Workflow-Management soll auf Benutzereingaben schnell reagiert werden, damit ein effektives Arbeiten für den Benutzer möglich ist. Außerdem sollen Aktivitäten, die direkt auf eine abgeschlossene Aktivität folgen, schnell bearbeitet werden können.

Um viele Workflows gleichzeitig auf einer gegebenen Infrastruktur ausführen zu können, muß der Ressourcen-Verbrauch jedes Workflows möglichst gering gehalten werden. Da der Kommunikationsmanager als Systemkomponente in jeder Workflow-Partition enthalten ist, belastet er die Rechner, auf denen er ausgeführt wird, durch den Austausch von Änderungsinformationen. Wird die Anzahl der zwischen den Partitionen verschickten Nachrichten möglichst gering gehalten, verringert sich die Netzbelastung und verkürzt sich voraussichtlich die Antwortzeit.

2.1.4. Integration in das MENTOR-Gesamtsystem

Zur Integration des Kommunikationsmanager in das MENTOR-Gesamtsystem müssen verschiedene Schnittstellen definiert werden. Im einzelnen sind dies:

- Die Schnittstelle zum Log-Manager, der Änderungen protokolliert, um nach einem Ausfall den Systemzustand vor dem Ausfall wiederherstellen zu können.
- Die Schnittstelle zum History-Manager, der Statusabfragen und Langzeitanalysen über die Workflow-Ausführung ermöglicht.
- Die Schnittstelle zur Workflow-Engine, die die Spezifikation des Workflows ausführt. Im Falle von MENTOR ist die Workflow-Engine state- und activitychart-basiert.
- Die Schnittstelle zum Worklist-Manager, der den Aktivitäten zur Laufzeit ein Ausführungsorgan zuordnet.
- Für eine prinzipielle Erweiterbarkeit muß der Kommunikationsmanager so gestaltet sein, daß er das Hinzufügen von Schnittstellen für weitere Komponenten zuläßt.

2.2. Konzeption des Kommunikationsmanagers

2.2.1. Schichtenarchitektur

Dem Kommunikationsmanager liegt eine offene Architektur und ein Schichtenmodell zu Grunde. Für diesen Ansatz sprechen die folgenden Gründe:

- Die Unterteilung in möglichst unabhängige Einzelkomponenten reduziert die Komplexität des Entwurfs des Gesamtsystems. Die einzelnen Schichten können dadurch unabhängig voneinander implementiert und optimiert werden.
- Die Implementierungsdetails einer Schicht bleiben den anderen Schichten verborgen, da die einzelnen Schichten lediglich über eine definierte Schnittstelle miteinander kommunizieren.
- Im Kommunikationsmanager werden verschiedene externe Softwarekomponenten verwendet, zum Beispiel zum Übertragen von Informationen auf andere Rechner. Bei diesen Komponenten handelt es sich oft um autarke Softwaresysteme, die häufig keine Möglichkeiten zur leichten Integration mit anderen Systemen bieten. Spezielle Schnittstellen-Schichten binden diese Komponenten an das MENTOR-Gesamtsystem an.
- Eine Schichtenarchitektur ermöglicht, daß einzelne Komponenten des Systems austauschbar und durch die für den jeweiligen Anwendungszweck am besten geeigneten oder durch neuere Versionen dieser Komponenten ersetzbar werden.

In Abbildung 7 ist der Aufbau des Kommunikationsmanagers dargestellt. Er besteht aus den folgenden fünf Schichten:

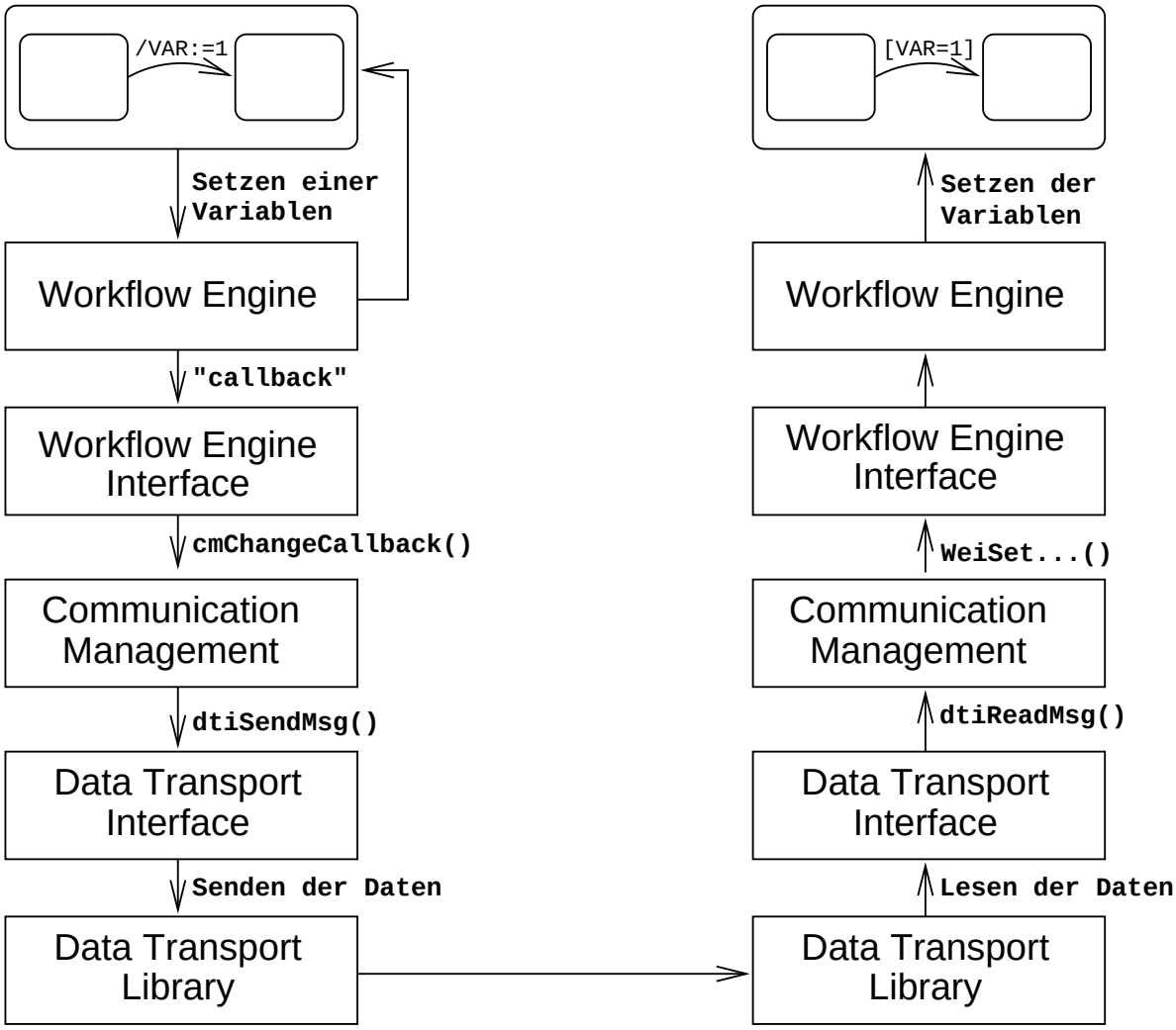

Abbildung 7 - Schichtenstruktur des Kommunikationsmanagers

- Die *Workflow-Engine-Schicht* (WE) ist das Laufzeitsystem, das die State- und Activity-charts ausführt. Sie gehört zu den oben genannten externen Softwarekomponenten, die in das System integriert werden müssen.

- Die *Workflow-Engine-Interface-Schicht* (WEI) stellt Standardfunktionen bereit, mit denen die CM-Schicht auf die WE-Schicht einwirken kann, sowie Callbacks, mit denen die CM-Schicht Informationen über Änderungen aus der WE-Schicht erhält.

- Die *Communication-Management-Schicht* (CM) propagiert Änderungsinformationen zwischen den Partitionen des Workflows und sorgt für die korrekte Abarbeitung des Workflows.

- Die *Data-Transport-Interface-Schicht* (DTI) stellt Standardfunktionen bereit, mit denen Nachrichten gelesen und geschrieben sowie Transaktionen gesteuert werden können, sofern die DT-Schicht diese vorsieht.

- Die *Data-Transport-Schicht* (DT) gewährleistet den Austausch von Nachrichten zwischen verschiedenen Partitionen des Workflows und sichert gegebenenfalls die Transaktionseigenschaften dieses Austausches. Sie ist ebenfalls eine der externen Softwarekomponenten des Systems.

Zu beachten ist, daß die WE- und DT-Schichten externe Softwarekomponenten sind, die über die jeweilige Schnittstellen-Schicht angebunden werden. Die in der Abbildung angedeuteten Funktionsaufrufe werden im folgenden zusammen mit den einzelnen Schichten näher beschrieben.

Im folgenden werden die einzelnen Schichten näher beschrieben.

2.2.2. Die Workflow-Engine-Schicht

Die Workflow-Engine-Schicht ist die Komponente des Systems, die die State- und Activitycharts ausführt. In MENTOR wird dazu entweder das Laufzeitsystem von Statemate [i-Log94a-c] oder ein im Rahmen des Projektes entwickelter Chart-Interpreter eingesetzt. Der Einsatz von Statemate im Bereich Workflow-Management, insbesondere zur Spezifikation und Ausführung, wurde zum ersten Mal in der Arbeit von [BSW95] betrachtet.

2.2.3. Die Communication-Management-Schicht

Die Communication-Management-Schicht ist als Schleife von fünf Verarbeitungsschritten realisiert, die in Abbildung 8 schematisch dargestellt sind.

Im ersten Schritt werden Maßnahmen zur Synchronisation der Partitionen des Workflows getroffen. Dies ist erforderlich, um die zur zentralen Ausführung äquivalente Abarbeitung des Workflows zu gewährleisten. Im Anschluß daran werden Änderungsinformationen, die von anderen Partitionen empfangen wurden, verarbeitet und lokal ausgeführt. Nun wird ein lokaler Schritt der Workflow-Engine ausgeführt; anschließend werden Informationen über die dabei aufgetretenen Änderungen an die anderen Partitionen des Workflows propagiert. Zuletzt werden die Änderungsinformationen an den Log-Manager, der sie für den Fall eines Rechner-Ausfalls in ein Logfile schreibt, und an den History-Manager, der sie für seine Zwecke auswertet, weitergeleitet.

Alle Verarbeitungsschritte werden in einer Transaktionen gekapselt. Dadurch wird erreicht, daß alle Schritte eine atomare Einheit bilden und entweder komplett oder - etwa beim Ausfall eines Rechners - gar nicht ausgeführt werden.

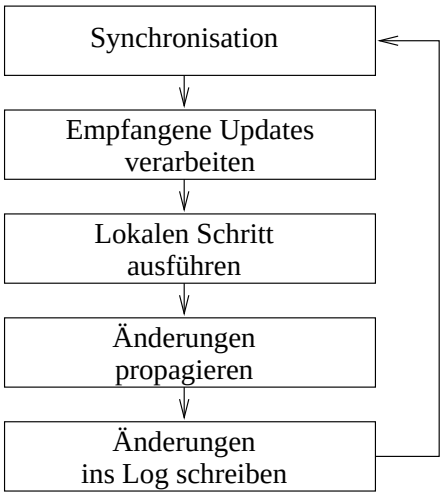

Abbildung 8 - Aufbau der Communication-Management-Schicht

2.2.4. Die Workflow-Engine-Interface-Schicht

Die Workflow-Engine-Interface-Schicht implementiert Funktionen zum Zugriff auf die Workflow-Engine und abstrahiert dadurch zur CM-Schicht hin von der verwendeten Workflow-Engine. Wenn eine andere Workflow-Engine eingesetzt werden soll, muß daher ebenfalls die WEI-Schicht ersetzt werden.

Die Funktionen der WEI-Schicht lassen sich in die folgenden vier Teilbereiche einteilen:

- Funktionen zum Ändern von Variablen der WE-Schicht:

  `weiSetInteger(...)` Setzen einer Integer-Variablen

  `weiSetDouble(...)` Setzen einer Double-Variablen

  `weiSetString(...)` Setzen einer String-Variablen

  `weiSetCondition(...)` Setzen einer Bedingung

  `weiGenerate(...)` Generieren eines Ereignisses

- Funktionen zum Initialisieren und Beenden der Workflow-Engine:

  `weiInitWorkflowEngine(...)`

  `weiExitWorkflowEngine(...)`

- Funktion zum Ausführen eines Schritts in der WE-Schicht:

  `weiMakeStep(...)`

- Aufruf von Callback-Funktionen der CM-Schicht bei Änderungen:

  `cmChangeCallback(...)` Benachrichtigung über einen geänderten Variablenwert

  `cmStartActivityCallback(...)` Benachrichtigung über eine gestartete Aktivität

2.2.5. Die Data-Transport-Schicht

Die Data-Transport-Schicht realisiert die Kommunikation der einzelnen Partitionen, sie muß daher die im folgenden geschilderten Anforderungen erfüllen. Sollten verschiedene der genannten Anforderungen nicht auf DT-Ebene realisiert sein, müssen sie durch die zugehörige DTI-Schicht implementiert werden.

- Die Data-Transport-Schicht muß das Senden von Nachrichten an einzelne Empfänger ermöglichen. Sie kann darüber hinaus Mechanismen zum Broadcast von Nachrichten enthalten.
- Es muß gewährleistet sein, daß jede der verschickten Nachrichten genau einmal ankommt, also nicht verloren geht oder verdoppelt wird.
- Die Reihenfolge der Nachrichten, die gesendet werden, muß mit der Reihenfolge übereinstimmen, in der die Nachrichten beim Empfänger eingehen.
- Verteilte Transaktionen, die über verschiedenen Medien wie zum Beispiel Datenbanken ausgeführt werden, müssen unterstützt werden.
- Die Data-Transport-Schicht muß Möglichkeiten zum Erkennen und Behandeln von Fehlern anbieten.

Als Datentransportschichten werden externe Softwarepakete oder Mechanismen des Betriebssystems verwendet, zum Beispiel TP-Monitore wie Tuxedo oder Encina, Sun-RPC, Sockets oder E-Mail. Da diese in der Regel nicht als Quellcode, sondern nur "black box"-artig als ausführbare Programme vorliegen, ist der Umweg über die DTI-Schicht zur Einbindung der DT-Schicht in den Kommunikationsmanager unumgänglich. In Abbildung 9 werden einige dieser Möglichkeiten hinsichtlich der Anforderungen aus Kapitel 2.1.1 verglichen. Daraus wird ersichtlich, daß ein TP-Monitor alle notwendigen Anforderungen erfüllt. Abstriche sind allerdings im Hinblick auf Performance und Administrationsaufwand zu machen. Letzteres wird jedoch durch Schaffung einer semiautomatischen Administrationsumgebung im Rahmen der Diplomarbeit von Mark Wehrmann [Weh97] erheblich abgemildert.

2.2.6. Die Data-Transport-Interface-Schicht

Die DTI-Schicht implementiert Standardfunktionen zum Lesen und Schreiben sowie zur Steuerung von Transaktionen für die jeweilige Datentransportschicht und abstrahiert dadurch zur CM-Schicht hin von der verwendeten DT-Schicht. Ein Wechsel der DT-Schicht ist daher nur durch gemeinsames Ersetzen von DT- und DTI-Schicht möglich. Die Funktionen der DTI-Schicht lassen sich in die folgenden vier Teilbereiche einteilen:

- Funktionen zum Austausch von Nachrichten:

`dtiReadMsg(...)`	Empfangen einer Nachricht
`dtiSendMsg(...)`	Gezieltes Senden einer Nachricht an eine Partition
`dtiBroadcast(...)`	Senden einer Nachricht an alle Partitionen

Eigenschaft	Mail	SunRPC	TP-Monitor	Sockets
* Senden und Empfangen von Nachrichten	ja	ja	ja	ja
* Jede verschickte Nachricht kommt an	nein	ja	ja	ja
* Nachrichten werden nicht verdoppelt	nein	ja	ja	ja
* Reihenfolge der Nachrichten bleibt erhalten	nein	ja	ja	ja
* Verteilte Transaktionen	nein	nein	ja	nein
* Möglichkeiten zur Fehlererkennung und -behandlung	nein	kaum	ja	kaum
* Performance	unver-läßlich	schnell	mittel bis langsam	schnell
* Aufwand zur Administrierung	gering	mittel	sehr groß	mittel

Abbildung 9 - Vergleich verschiedener Datentransportschichten

- Funktionen zur Unterstützung von Transaktionen:

`dtiBegin(...)`	Starten einer Transaktion
`dtiCommit(...)`	Erfolgreiches Abschließen einer Transaktion
`dtiAbort(...)`	Abbrechen einer Transaktion

- Funktionen zur Unterstützung von dynamischen Änderungen:

`dtiAddProc(...)`	Hinzufügen einer Partition zur Kommunikation
`dtiRemProc(...)`	Entfernen einer Partition aus der Kommunikation

- Funktionen zum Auf- und Abbau der Kommunikation:

`dtiInit(...)`	Initialisieren der Kommunikation
`dtiShutdown(...)`	Beenden der Kommunikation

2.3. Konfigurationsprofil für Workflows

2.3.1. Identifikation eines verteilten Workflows

Im praktischen Einsatz eines Workflow-Management-Systems werden viele verschiedene Geschäftsvorgänge gleichzeitig bearbeitet. Die Aktionen eines Ausführungsorgans dürfen sich nur auf den aktuell bearbeiteten Workflow auswirken, nicht aber direkt auf die anderen. Es muß also eine Möglichkeit geben, verschiedene Workflow-Typen und Instanzen eines Workflow-Typs zu unterscheiden.

Dazu wird jeder Workflow-Spezifikation eine eindeutige Zahl zugewiesen, der *Workflow-Typ*. Ein Workflow zur Kreditbearbeitung hätte also beispielsweise den Workflow-Typ 1, während ein Workflow für eine Kontoeröffnung, den Typ 2 haben könnte. Diese Zahl wird unternehmensweit von einem Administrator nach Abschluß der Spezifikation des Workflows festgelegt.

Beim Starten eines Workflow muß dessen Typ angegeben werden, damit eine neue Instanz des Workflows erzeugt werden kann. Da in der Regel mehrere Instanzen eines Typs gleichzeitig ablaufen können, muß man die einzelnen Instanzen unterscheiden können. Dies geschieht durch die innerhalb eines Workflow-Typs eindeutige Zahl *Workflow-Id*, die beim Start eines Workflows durch das Laufzeitsystem vergeben wird. In einer Bankanwendung könnte zum Beispiel die Bearbeitung des Kredits von Herrn Weikum, die Workflow-Id 1 haben, während die Bearbeitung des Kredits von Herrn Wilhelm, die Workflow-Id 2 hat.

Innerhalb eines Workflows gibt es im allgemeinen mehrere Teilprozesse, die jeweils eine Partition des Workflows ausführen. Jedem dieser Teilprozesse wird bei der Spezifikation eine eindeutige Zahl zugewiesen, die *Proc-Id* mit der die Prozesse zur Laufzeit angesprochen werden. In einer Kreditbearbeitung könnte zum Beispiel der Teilprozeß, der den Teil steuert, in dem die Kundendaten von Herrn Weikum erfaßt werden, die Proc-Id 1 haben, während die Risikobewertung von Herrn Weikum Proc-Id 2 hat.

Workflow-Typ, Proc-Id, Workflow-Id zusammen erlauben eine eindeutige Referenzierung eines Workflow-Teilprozesses innerhalb des gesamten Workflow-Management-Systems (*Workflow-Identifizierung*), um ihm zum Beispiel Nachrichten schicken zu können.

2.3.2. Workflow-Konfigurationsdatei

Zu jedem Workflow-Typ gibt es eine Konfigurationsdatei, die Informationen über die am Workflow beteiligten Rollen und Partitionen und die daran beteiligten Ausführungsorgane enthält. Sie wird zum Teil im Rahmen der Workflow-Spezifikation erstellt. Zur Laufzeit kann der Worklist-Manager weitere Ausführungsorgane eintragen. Die Konfigurationsdatei wird u.a. beim Instanziieren eines Workflows benutzt, um die notwendigen Teilprozesse zu starten und die Kommunikation zwischen ihnen aufzubauen. Außerdem dient sie der Zuordnung von Ausführungsorganen zu Ausführungsrollen zur Laufzeit. Abbildung 10 zeigt einen Ausschnitt aus der Konfigurationsdatei des Workflows zur Erstellung eines Lehrberichts, der bereits in Kapitel 1 vorgestellt wurde.

Im folgenden werden die Teilblöcke der Konfigurationsdatei und ihre Verwendung vorgestellt.

Der Workflow-Block

Dieser Block enthält allgemeine Beschreibungen über den Workflow-Typ wie zum Beispiel seinen Namen (Zeile 2), das Erstellungsdatum (Zeile 3-4) und den Namen des Autors (Zeile 5). Zudem kann er zusätzlich Parameter für Systemkomponenten enthalten wie zum Beispiel in Abbildung 10

```
1  workflow :
2     name : ZLB_MC
3     version : 835092142  --
4        Tue Jun 18 2:02:22 1996
5     author : wehrmann
6     type : 50
7     master : 5
8     backup : 6
9     cfgversion : 2
10 end workflow
11
12 [...]
13
14 role :
15    name : SB_61_2
16    roleid : 5
17    chart : SB_61_2_AC
18    fgfile : SB_61_2.fg
19    svbname : SB_61_2_AC.svb
20    homedir : /home/mentor/
21       workflows/lehrbericht/
22 end role
23
24 role :
25    name : Prodekan
26    roleid : 6
27    chart : Prodekan_AC
28    fgfile : Prodekan.fg
29    svbname : Prodekan_AC.svb
30    homedir : /home/mentor/
31       workflows/lehrbericht/
32 end role
33
34 [...]
35
36 entity :
37    name : Harald Schmidt
38    entityid : 5
39 end entity
40
41 entity :
42    name : Heinz Becker
43    entityid : 6
44 end entity
45
46 entity :
47    name : Prof. Raimund Seidel
48    entityid : 7
49 end entity
50
51 [...]
52
53  process :
54     location : Saarbruecken
55     orgunit : Uni-Verwaltung
56     name : SB_61_2
57     procid : 5
58     roleid : 5
59     hosts : frankfurt.cs.
60             uni-sb.de,
61             paris.cs.uni-sb.de,
62             london.cs.uni-sb.de
63     homedir : /home/mentor/
64        workflows/lehrbericht/
65     filename : dsitux
66     type : DYNAMIC
67     display     :     trier.cs.
68                uni-sb.de:0
69     entityid : 5
70  end process
71
72  process :
73     location : Saarbruecken
74     orgunit : Uni-Verwaltung
75     name : SB_61_2
76     procid : 6
77     roleid : 5
78     hosts : paris.cs.uni-sb.de
79     homedir : /home/mentor/
80        workflows/lehrbericht/
81     filename : dsitux
82     type : DYNAMIC
83     display : homburg.cs.
84                uni-sb.de:0
85     entityid : 6
86  end process
87
88  process :
89     location : Saarbruecken
90     orgunit : Informatik-FB
91     name : Prodekan
92     procid : 7
93     roleid : 6
94     hosts : paris.cs.uni-sb.de
95     homedir : /home/mentor/
96        workflows/lehrbericht/
97     filename : dsitux
98     type : STATIC
99     display : kyoto.cs.
100               uni-sb.de:0
101    entityid : 7
102 end process
```

Abbildung 10 - Ausschnitt aus dem Konfigurationsfile des Lehrbericht-Workflows

die `master`- (Zeile 7) und `backup`-Einträge (Zeile 8), die bestimmte Eigenschaften der Datentransportschicht festlegen.

Der Role-Block

Zu jeder Rolle eines Workflows muß es einen Role-Block geben (z.B. Zeilen 14-22). Neben dem Rollen-Namen (Zeile 15) enthält er die eindeutige Identifizierungsnummer der Rolle (Zeile 16),

sowie den Namen der der Rolle zugeordneten Partition der Workflow-Spezifikation (Zeile 17). Der Name der Datei, in der alle in der Partition verwendeten Variablen aufgeführt sind, steht im Eintrag `svbname` (Zeile 19). Der `fgfile`-Eintrag (Zeile 18) ist der Name der Datei, die notwendige Daten für den Aufbau einer Benutzeroberfläche speichert. Der `homedir`-Eintrag (Zeile 20-21) gibt das Verzeichnis an, in dem sich alle diese Dateien befinden.

Der Entity-Block

Für jedes Ausführungsorgan, gibt es einen Entity-Block (z.B. Zeilen 36-39), der neben dem Namen des Ausführenden (Zeile 37) eine eindeutige Identifizierungsnummer (Zeile 38) beinhaltet, über die der Ausführende im Workflow referenziert werden kann.

Der Prozeß-Block

Der Prozeß-Block (z.B. Zeilen 53-70) beschreibt einen zu startenden Teilprozeß und dessen Ausführungsumgebung. Er enthält daher Einträge für das beim Starten auszuführende Programm (`filename`, Zeile 65), eine Liste von Rechnern, auf denen dieser Teilprozeß gestartet werden kann (`hosts`, Zeilen 59-62) und den Rechner auf dessen Bildschirm die Ausgaben erscheinen sollen (`display`, Zeile 67-68). Die bereits genannte eindeutige Identifizierungsnummer des Prozesses wird durch den `procid`-Eintrag (Zeile 57) festgelegt. Die Felder `roleid` (Zeile 58) und `entityid` (Zeile 69) stellen den Bezug zur ausgeführten Rolle und dem Ausführenden her. Der `type`-Eintrag (Zeile 66) dient der Unterscheidung verschiedener Arten von Teilprozessen. Teilprozesse, in denen sich initial zu startende Aktivitäten befinden, werden mit dem Schlüsselwort `STATIC` kenntlich gemacht (Zeile 98), während solche, die erst im Verlauf der Workflow-Ausführung gestartet werden, durch das Schlüsselwort `DYNAMIC` gekennzeichnet werden (Zeile 66). Diese Unterscheidung dient der Reduzierung des Ressourcen-Verbrauchs bei der Workflow-Ausführung. Außerdem lassen sich hier Systemprozesse des Workflow-Management-Systems durch besondere Schlüsselworte konfigurieren, zum Beispiel Administrations- und Überwachungsprozesse. Die `location`- (Zeile 54) und `orgunit`-Einträge (Zeile 55) bestimmen die Orts- und Abteilungszugehörigkeit des Rechners, auf dem der jeweilige Teil des Workflows ausgeführt wird. Diese Informationen werden benötigt, um Performance- und Replikationsaspekte für die Ausführungsumgebung des Workflows berücksichtigen zu können.

2.3.3. Datenbankrelationen für die Konfiguration

Zusätzlich zu den Informationen in der Konfigurationsdatei wird zur Laufzeit in Datenbank-Tabellen protokolliert, welche Workflow-Instanzen und Workflow-Teilprozesse gerade aktiv sind.

Die Relation WFC_Workflows

In der Relation `WFC_Workflows` werden die aktiven Instanzen gespeichert, um zum Beispiel beim Starten einer neuen Workflow-Instanz eine unbenutzte WorkflowID zu finden. Das Attribut `Status` gibt den aktuellen Zustand der Workflow-Instanz an, das heißt ob die Instanz läuft, in der Startphase ist, beendet wird oder bereits beendet ist. Diese Tabelle erlaubt damit einen Überblick über die zur Zeit vom System verarbeiteten Workflows und ihre aktuellen Zustände.

```
TABLE WFC_Workflows
  WorkflowType INTEGER NOT NULL
  WorkflowID   INTEGER NOT NULL
  Status       INTEGER
PRIMARY KEY (WorkflowType,WorkflowID)
```

Die Relation WFC_Processes

In der Relation `WFC_Processes` werden die Workflow-Prozesse vermerkt, die zu den aktuell laufenden Workflowinstanzen gehören und bereits gestartet sind. Das Attribut `Status` gibt an, ob der Prozeß läuft, in der Startphase ist, beendet wird oder bereits beendet ist. Die Informationen in dieser Tabelle werden zum Beispiel dazu verwendet, beim Starten eines Prozesses festzustellen, welche anderen Teilprozesse der Workflow-Instanz aktiv sind.

```
TABLE WFC_Processes
  WorkflowType  INTEGER NOT NULL
  WorkflowID    INTEGER NOT NULL
  ProcID        INTEGER NOT NULL
  Status        INTEGER
PRIMARY KEY (WorkflowType,WorkflowID,ProcID)
FOREIGN KEY (WorkflowType,WorkflowID) REFERENCES WFC_Workflows
```

3. CM-Schicht und WEI-Schicht

In diesem Kapitel wird ausführlich der Kern des Kommunikationsmanagers, die *Communication-Management*-Schicht, sowie die Anbindung an verschiedene Workflow Engines dargestellt. Zunächst wird die Konzeption der CM-Schicht beschrieben, die als zentrale Komponente des Kommunikationsmanagers für den korrekten verteilten Ablauf des Workflows verantwortlich ist. Im folgenden Abschnitt wird die Implementierung dieser Schicht dargestellt. Anschließend wird die Integration der Workflow Engines Statemate und Chart Interpreter vorgestellt. Im abschließenden Abschnitt werden Alternativen für den Entwurf der CM-Schicht diskutiert.

3.1. Konzeption der CM-Schicht

3.1.1. Abstraktion von den verwendeten Systemkomponenten

Wie bereits im vorhergehenden Kapitel beschrieben, stellt die CM-Schicht den eigentlichen Kern des Kommunikationsmanagers dar. Sie hat die Aufgabe, durch Überwachen der lokalen Ausführung der Workflow-Engine und Versenden geeigneter Nachrichten an die übrigen Partitionen des Workflows die gegenüber der zentralen Ausführung verhaltensäquivalente Ausführung des Workflows zu gewährleisten. Dabei soll die Zahl der verschickten Nachrichten so gering wie möglich sein, um die Belastung des Verbindungsnetzwerkes möglichst gering zu halten.

Die CM-Schicht muß so konzipiert werden, daß sie unabhängig von der verwendeten Workflow-Engine und Datentransportschicht arbeitet, damit eine transparente Austauschbarkeit dieser Komponenten des Gesamtsystems gewährleistet bleibt. In der Schichtenarchitektur des Kommunikationsmanagers wurden dazu zwei besondere Interface-Schichten vorgesehen, die die Abstraktion von den verwendeten Komponenten gewährleisten. Abbildung 7 in Kapitel 2 zeigt die Workflow-Engine-Interface-Schicht (WEI-Schicht) zwischen WE-Schicht und CM-Schicht und die Data-Transport-Interface-Schicht (DTI-Schicht) zwischen CM-Schicht und DT-Schicht.

Um unabhängig von der verwendeten Workflow Engine zu bleiben, wird innerhalb der CM-Schicht ein vereinfachtes Modell von Variablen, Zuständen und Aktivitäten verwendet, das zunächst nichts mit den gleichen Elementen der Workflow-Engine zu tun hat. Die Abbildung der WE-Elemente auf die CM-Elemente und umgekehrt übernimmt ein Teil der Workflow-Engine-Interface-Schicht; insbesondere weiß die CM-Schicht nichts von der Implementierung der Variablen, Zustände und Aktivitäten in der Workflow Engine. Alle Manipulationen an Elementen auf der CM-Seite müssen durch Funktionen der WEI-Schicht vorgenommen werden, die diese Änderungen an die Workflow Engine weitergeben. Änderungen von Variablen, Zuständen und Aktivitäten der Workflow Engine werden durch den Aufruf von Callback-Funktionen an die CM-Schicht weitergegeben.

Durch diese konsequente Trennung von CM-Schicht und Workflow Engine ist es sogar denkbar, durch geeignete Konvertierungsmaßnahmen in der WEI-Schicht Workflow-Engines zu unterstützen, die nicht statechart-basiert arbeiten, aber mindestens die gleiche Ausdrucksmächtigkeit besitzen. Dieser Ansatz wurde in dieser Arbeit jedoch nicht weiter verfolgt.

Auf der anderen Seite darf die CM-Schicht natürlich auch keine besonderen Eigenschaften der Datentransportschicht verwenden, sondern muß sich ausschließlich auf eine definierte Schnittstelle stützen, die von der Data-Transport-Interface-Schicht (DTI-Schicht) zur Verfügung gestellt wird. Die CM-Schicht hat keine Informationen darüber, welche Art von Kommunikation benutzt wird, um Nachrichten zu senden und zu empfangen. Dadurch kann man für verschiedene Workflows

verschiedene Kommunikationsmechanismen verwenden. Sogar innerhalb einer einzigen Workflowinstanz könnte man durch geeignete Maßnahmen in der DTI-Schicht unterschiedliche Datentransportschichten zur Kommunikation mit verschiedenen Partitionen einsetzen, ohne dazu etwas in der CM-Schicht ändern zu müssen. Die Notwendigkeit solcher Maßnahmen kann sich etwa aus der Netztopologie oder den verwendeten Betriebssystemen ergeben. In dieser Arbeit wurden nur "homogene" verteilte Systeme betrachtet, so daß innerhalb einer Instanz nur eine Art der Kommunikation eingesetzt wird.

3.1.2. Aufbau der CM-Schicht aus Synchronisationsphase, Lesephase, Ausführungsphase und Schreibphase

Abbildung 11 zeigt den prinzipiellen Aufbau der CM-Schicht. Sie besteht im wesentlichen aus einer Schleife, in der nacheinander die folgenden Phasen ausgeführt werden:

- Durchführen von Maßnahmen zur Synchronisation der Partition mit den übrigen Partitionen des Workflows *(Synchronisationsphase*)
- Lesen von Änderungsinformationen, die andere Partitionen gesendet haben, und Verarbeiten derselben (*Lesephase*)
- Ausführen eines Schrittes der lokalen Workflow Engine (*Ausführungsphase*)
- Propagieren der dabei aufgetretenen Änderungen an die übrigen Partitionen (*Schreibphase*)

Ein Durchlauf durch diese Schleife wird im folgenden *ein Schritt der zugehörigen Partition* genannt. Der konzeptionelle Aufbau der einzelnen Phasen der CM-Schicht mit Ausnahme der Synchronisationsphase wird in den folgenden Abschnitten dargestellt; diese wird erst in Kapitel 4 beschrieben.

3.1.3. Aufbau der Lesephase

In der Lesephase müssen die Informationen über Änderungen von Variablen, Zuständen und Aktivitäten empfangen werden, die von den übrigen Partitionen des Workflows verschickt wurden. Insbesondere ist es für die korrekte Ausführung wichtig, daß alle Änderungen, die innerhalb eines Schritts einer Partition aufgetreten sind und daher verschickt wurden, auch auf einmal verarbeitet werden. Andernfalls könnte es bei der verteilten Ausführung des Workflows zu anderen Ergebnissen als bei der zentralen kommen, was natürlich vermieden werden muß.

Um dieses Ziel zu erreichen, wurde der Leseschritt der CM-Schicht wie in Abbildung 12 gezeigt konzipiert:

Zunächst wird eine Transaktion begonnen, damit das Auslesen und Verarbeiten der eingehenden Informationen atomar geschehen kann: Entweder werden alle Informationen verarbeitet, oder keine. Anschließend werden alle Nachrichten eingelesen, die in der Zwischenzeit von anderen Partitionen eingegangen sind. Unmittelbar nach dem Lesen einer Nachricht wird die in ihr enthaltene Information über einen geänderten Variablenwert usw. ausgewertet und an die lokale Workflow Engine weitergegeben. Dies geschieht mit den bereits erwähnten Funktionen der WEI-Schicht zur Beeinflussung von Variablen in der Workflow Engine.

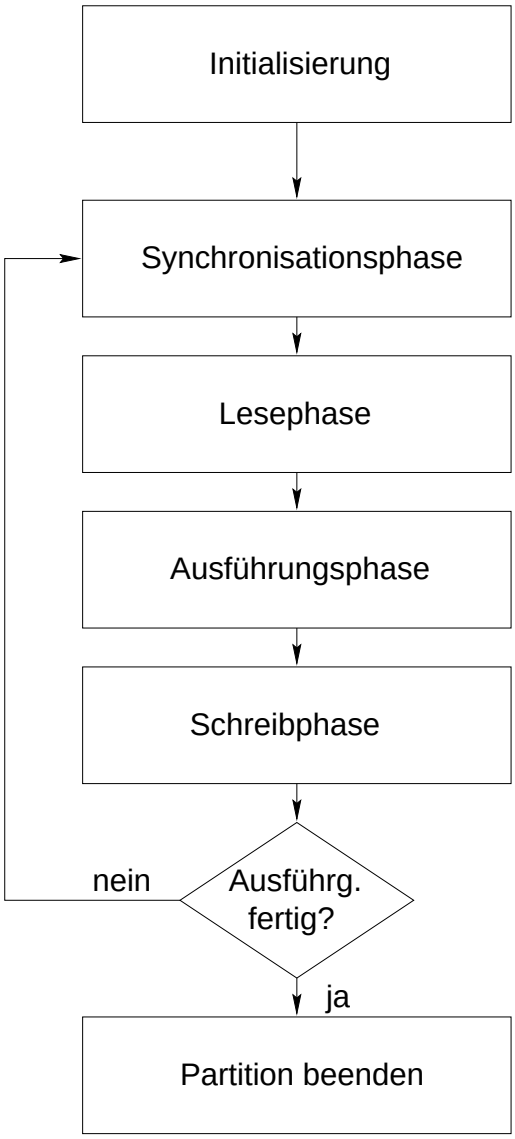

Abbildung 11: Prinzipieller Aufbau der CM-Schicht

Wenn alle Nachrichten gelesen und verarbeitet wurden, die bisher von den anderen Partitionen empfangen wurden, muß geprüft werden, ob *alle Nachrichten von allen Partitionen* eingegangen sind, oder ob noch Nachrichten von Partitionen ausstehen. Um dieses zu erkennen, müssen die beiden folgenden Probleme gelöst werden:

- Wie erkennt man, daß alle Nachrichten einer Partition eingegangen sind?
- Wie erkennt man, wenn eine Partition überhaupt keine Nachrichten geschickt hat, etwa weil sich bei der Ausführung eines Schrittes ihrer Workflow Engine keine Änderungen ergeben haben?

Das erste Problem läßt sich relativ einfach durch Klammern des Schreibens der Informationen über Änderungen an die übrigen Partitionen in Transaktionsklammern lösen. Dadurch ist gewährleistet, daß immer alle Nachrichten einer Partition gleichzeitig sichtbar werden. Da im Leseschritt immer alle anliegenden Nachrichten gelesen werden, kann es also nicht vorkommen, daß ein Teil der Nachrichten erst im folgenden Schritt bearbeitet wird.

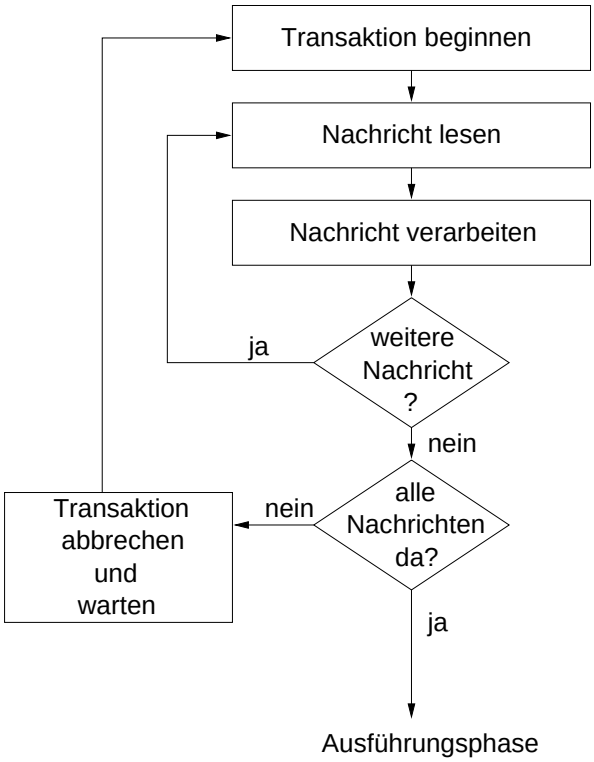

Abbildung 12 - Aufbau der Lesephase der CM-Schicht

Zur Behandlung des zweiten Problems sind Maßnahmen zur Synchronisation aller Partitionen des Workflows erforderlich. Es genügt dazu, wenn eine Partition ohne Änderungen im ausgeführten lokalen Schritt eine “leere” Nachricht ohne Nutzinhalt an alle übrigen Partitionen schickt. Im folgenden Kapitel wird sich zeigen, daß dieser Ansatz unmittelbar zum Konzept der strikten Synchronisation führt, hier wird daher nicht weiter darauf eingegangen.

Hat man erkannt, daß noch Nachrichten von Partitionen ausstehen, wird die am Anfang des Schrittes begonnene Transaktion zurückgesetzt. Da die Datentransportschicht selbst auch Transaktionen unterstützt, bedeutet das, daß alle bis dahin gelesenen Informationen erneut gelesen werden können, und zwar in der gleichen Reihenfolge wie vorher. Nach einer kurzen Wartezeit, während der weitere Nachrichten von anderen Partitionen eingehen können, wird wieder am Anfang der Leseschleife begonnen.

Konzeptionell ist es an dieser Stelle nicht unbedingt notwendig, die Transaktion abzubrechen und neu zu beginnen. Insbesondere muß man dadurch den Nachteil in Kauf nehmen, daß bereits ausgewertete Nachrichten anderer Partitionen erneut bearbeitet und die in ihnen enthaltenen Änderungen erneut lokal propagiert werden müssen. Alternativ könnte man auch die Transaktion erfolgreich beenden und nach der Wartezeit eine neue starten. Im Falle eines Absturzes wird dadurch allerdings die Recovery erschwert, die durch den in [Klä97] und [Sti97] beschriebenen Logmanager durchgeführt wird. Dort wird davon ausgegangen, daß eine abgestürzte Partition immer auf einem konsistenten Zustand am Anfang der Lese- oder der Schreibphase wieder aufgesetzt werden kann. Beendet man in der Lesephase die Transaktion, wird dabei auch der aktuelle Zustand der Partition ins Log geschrieben. Im Falle eines Absturzes an dieser Stelle befände sich die Partition also mitten in der Lesephase, was eine korrekte Recovery zumindest erschweren würde.

Ein weiterer Ansatz ist es, die Transaktion zwischenzeitlich nicht abzubrechen, sondern innerhalb der Transaktionsklammer auf das Eintreffen neuer Nachrichten zu warten. Dadurch würde man sich

sowohl die Probleme bei der Recovery als auch das wiederholte lokale Propagieren von Änderungen sparen. Es kann dadurch aber zu Transaktionen kommen, die sehr lange “offen” sind, etwa wenn eine Partition aufgrund eines Absturzes erst nach einer zeitaufwendigen Recovery ihre Nachrichten senden kann. Verschiedene Datentransportschichten, insbesondere Tuxedo, haben Probleme mit solchen langen Transaktionen: Man muß durch einen Parameter die maximale Dauer einer Transaktion spezifizieren, nach Ablauf dieser Zeit wird eine begonnene Transaktion automatisch abgebrochen. Dies ist gedacht, um Transaktionen zu erkennen, die von zwischenzeitlich abgestürzten Prozessen gestartet wurden. Wird dieser Parameter zu groß gewählt, um lange Transaktionen zu ermöglichen, wird dieser Mechanismus lahmgelegt. Außerdem erhöhen lange Blockierungen von Ressourcen durch Transaktionen die Wahrscheinlichkeit von Verklemmungen. Dies ist insbesondere deshalb ein Problem, weil der Logmanager, der beim lokalen Propagieren der Änderungsinformationen eingebunden ist, Informationen in der Datenbank verwaltet und dazu Schreibsperren hält, die die Ausführung anderer Logmanager in anderen Partitionen der gleichen Instanz behindern können.

3.1.4. Aufbau der Ausführungsphase

In dieser Phase wird ein Schritt in der lokalen Workflow-Engine ausgeführt. Um die Korrektheit der verteilten Ausführung zu gewährleisten, müssen die dabei auftretenden Änderungen von Variablen, Zuständen und Aktivitäten erkannt werden, damit sie in der folgenden Schreibphase an die übrigen Partitionen des Workflows propagiert werden können. Dies geschieht über Callbacks, die von der WEI-Schicht zur Verfügung gestellt werden. Jede Änderung einer Variablen hat den Aufruf einer Callback-Funktion der CM-Schicht zur Folge, die die entsprechende Variable im CM-Variablenmodell als geändert markiert und ihren Wert anpaßt.

Auch das Starten von Aktivitäten bewirkt den Aufruf einer Callback-Funktion der CM-Schicht, in der der Zustand der entsprechenden Aktivität im CM-Modell geändert wird. Für Zustände gibt es keine entsprechenden Callbacks, sie werden bereits bei der Partitionierung des Workflows durch zusätzlich eingeführte Bedingungsvariablen vollständig simuliert. Für einen Zustand `S` der Originalspezifikation wird dabei eine Bedingung `IN_S` geschaffen, die genau dann den Wert `true` hat, wenn `S` betreten ist. Abbildung 13 zeigt die Übersetzung von Ausdrücken, die Zustände verwenden, in äquivalente Ausdrücke mit dieser Bedingung.

Originalausdruck	Bedeutung	Übersetzung
`in(S)`	Zustand S betreten	`IN_S=true`
`entered(S)`	Zustand S wurde im letzten Schritt betreten	`true(IN_S)`
`exited(S)`	Zustand S wurde im letzten Schritt verlassen	`false(IN_S)`

Abbildung 13 - Simulation von Zuständen durch Bedingungen

Diese Konvertierung wurde vorgenommen, weil es in der Regel unproblematisch ist, Werte von Variablen in der Workflow Engine zu beeinflussen. Es ist bei den betrachteten Workflow Engines nicht möglich, die Workflow Engine von außen dazu zu veranlassen, Zustände zu betreten oder zu verlassen. Für eine korrekte Ausführung des verteilten Workflows ist aber genau dieses notwendig, da in Zustandsübergängen in einer Partition Zustände im Bedingungsteil der Beschriftung vorkommen können, die in einer anderen Partition liegen. Ohne Übersetzung nach Variablen wäre es also notwendig, Zustände in allen Partitionen zu "spiegeln", in denen sie verwendet werden, was aus den oben genannten Gründen nicht praktikabel ist.

3.1.5. Aufbau der Schreibphase

Nach Abschluß des Schrittes der lokalen Workflow Engine müssen Informationen über die geänderten Variablen und Aktivitäten, die während des Schrittes durch Callbacks gesammelt wurden, an die übrigen Partitionen propagiert werden. Ein erster Ansatz dazu ist, jeder Partition alle Informationen zu schicken; dadurch ist auf jeden Fall gewährleistet, daß jede Partition alle notwendigen Änderungsinformationen erhält. Neben diesen erforderlichen erhalten die Partitionen aber oft auch Informationen über Änderungen von Variablen und Aktivitäten, die sie überhaupt nicht benötigen. Eine Information über eine Variable ist insbesondere dann für eine Partition nicht notwendig, wenn die Variable in dem Statechart, daß zu der Partition gehört, nicht vorkommt. Die Ausführung der Partition kann nicht vom Wert einer solchen Variable abhängen. Dies führt unmittelbar zum Konzept des *gezielten Versendens* von Änderungsinformationen, das in der Schreibphase genutzt wird:

Eine Information über eine Änderung einer Variablen wird nur an die Partitionen propagiert, in deren Statechart die Variable verwendet wird.

Bereits während der Partitionierung des Workflows kann festgestellt werden, von welchen Partitionen eine Variable genutzt wird. Diese Informationen werden der CM-Schicht jeder Partition zur Verfügung gestellt, sie sind Teil des Variablenmodells der CM-Schicht. Auf diese Weise wird die Zahl der versendeten Nachrichten reduziert.

In der Schreibphase wird für jede Variable, die in der Ausführungsphase geändert wurde, an alle Partitionen eine entsprechende Nachricht geschickt, die diese Variable verwenden. In der versendeten Nachricht ist neben dem Namen der Variablen ihr neuer Wert enthalten, so daß die empfangenden Partitionen den Wert dieser Variablen in ihrer lokalen Workflow-Engine entsprechend ändern können.

Gestartete Aktivitäten werden anders behandelt. Grundsätzlich ist es nicht notwendig, Informationen über solche Ereignisse an andere Partitionen weiterzugeben, da gemäß den Regeln zur Partitionierung bereits entsprechende Maßnahmen getroffen wurden: Wie in Abbildung 14 gezeigt wird, werden bei der Partitionierung alle Start- und Stop-Anweisungen an partitionsübergreifenden Transitionen sowohl in der Ausgangs- als auch in der Zielpartition der Transition repliziert, so daß es genügt, die Änderung der die Transition auslösenden Variablen an die Zielpartition zu propagieren. Dies ist insbesondere deshalb ausreichend, weil eine Entwurfsregel für Workflows besagt, daß Aktivitäten nur beim Betreten des ihnen zugeordneten Zustandes gestartet werden dürfen, der in der gleichen Partition liegt, die die Aktivität ausführt. Auch dadurch werden wieder Nachrichten eingespart.

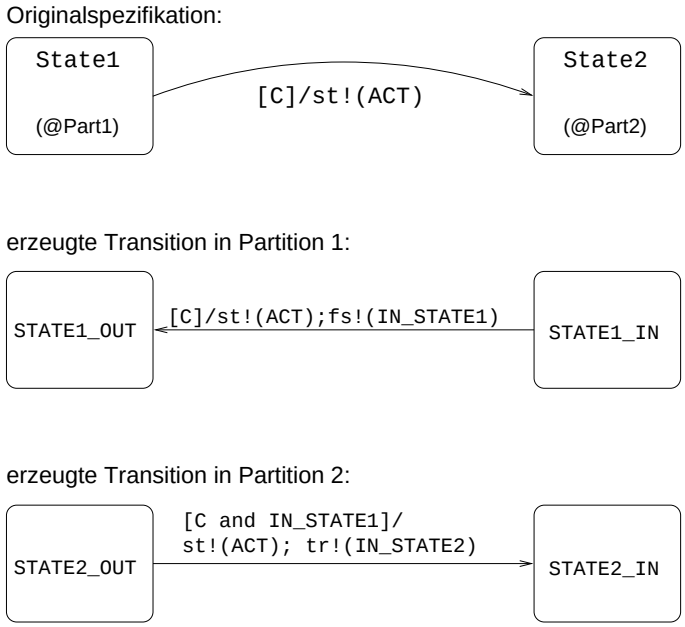

Abbildung 14 - Starten von Aktivitäten in unterschiedlichen Partitionen

Um die korrekte verteilte Ausführung des Workflows zu gewährleisten, ist es wichtig, daß immer alle Änderungen innerhalb eines Schrittes einer Partition an die übrigen Partitionen propagiert werden. Damit dieses gewährleistet ist, wird die am Anfang der Lesephase begonnene Transaktion erst erfolgreich beendet, wenn die Schreibphase abgeschlossen ist. Auf diese Weise wird der komplette Durchlauf der CM-Hauptschleife in einer Transaktion gekapselt.

3.1.6. Integration des Logmanagers

Der Logmanager von MENTOR, beschrieben in [Klä97] und [Sti97], hat die Aufgabe, alle Änderungen innerhalb einer Workflowinstanz mitzuprotokollieren, um im Falle des Ausfalls einer Partition diese wieder auf den letzten konsistenten Systemzustand zurücksetzen zu können. Der natürliche Ansatzpunkt für das Sammeln solcher Informationen liegt in der CM-Schicht des Kommunikationsmanagers, da hier ohnehin alle Informationen über Änderungen von Variablen und Aktivitäten festgestellt werden. Insbesondere muß der Logmanager an den folgenden Stellen über Änderungen informiert werden:

- Innerhalb der Callback-Funktion, die bei Änderungen von Variablen aufgerufen wird
- Innerhalb der Callback-Funktion, die beim Starten und Beenden von Aktivitäten aufgerufen wird

Zusätzlich muß der Logmanager informiert werden, wenn ein lokaler Schritt der Partition beginnt und endet, sowie aus technischen Gründen bei allen Transaktionsoperationen.

In der Praxis wurde der Logmanager nicht vollständig auf diese Weise implementiert, sondern greift unmittelbar in die unterliegende Workflow Engine ein. Dieser auf historischen Gründen beruhende Ansatz widerspricht zwar der modularen Gestaltung des Gesamtsystems, wurde aber deshalb gewählt, da der Logmanager auch für die fehlertolerante Ausführung von State- und Activitycharts außerhalb der Verwengung in MENTOR konzipiert wurde. Er unterstützt speziell in der Variante für Statemate wesentlich mehr Konzepte, als in MENTOR benötigt werden. Durch diese Vorgehensweise sind Änderungen in der Workflow Engine selbst notwendig. Die Workflow

Engine muß dazu entsprechende Mechanismen vorsehen oder Anpassungen an ihrem Quelltext erlauben, wie es bei Statemate und dem Chart Interpreter möglich ist.

Die Einbindung des Logmanagers in den Kommunikationsmanagers wird in [Weh97] ausführlich dargelegt und bleibt daher in dieser Arbeit weitgehend unberücksichtigt. Insbesondere auf das Erkennen des Ausfalls einer Partition und das Wiederherstellen des letzten konsistenten Systemzustandes wird hier nicht eingegangen.

3.1.7. Starten von Aktivitäten in nicht aktiven Partitionen

Für das Starten einer Aktivität in einer anderen Partition ist keine besondere Aktion außer dem Propagieren der üblichen Änderungsinformationen notwendig, da bereits beim Partitionieren entsprechende Vorkehrungen getroffen werden. Damit dieser Mechanismus funktioniert, ist es notwendig, daß alle Partitionen, in denen potentiell Aktivitäten gestartet werden können, auf Änderungsinformationen warten und diese lokal propagieren. Insbesondere bedeutet dies, daß auch Partitionen, in denen aktuell keine Aktivität ausgeführt wird, auf der Ebene des Betriebssystems als Prozeß laufen und Nachrichten von außen empfangen müssen, wozu wiederum entsprechende Ressourcen belegt werden müssen.

Im Zuge des sparsamen Umgangs mit den vorhandenen Ressourcen ist eine solche Vorgehensweise unbefriedigend, da auch im Moment eigentlich nicht benötigte Partitionen aktiv sein müssen, also Ressourcen belegen, die dann für andere Partitionen nicht mehr zur Verfügung stehen, in denen wirklich Arbeit verrichtet wird. An dieser Stelle ist es sinnvoll, eine besondere Behandlung solcher Partitionen vorzusehen. Ziel ist es dabei, daß nur solche Partitionen bearbeitet werden - das bedeutet, als Prozeß gestartet sind -, in denen es gestartete Aktivitäten gibt. Nur diese Partitionen tauschen Nachrichten aus. Alle anderen Partitionen bleiben inaktiv, das bedeutet insbesondere, daß sie weder auf der Systemebene als Prozeß gestartet werden noch in irgendeiner Art und Weise an der Kommunikation teilnehmen. Abbildung 15 zeigt ein Beispiel, in dem die Partitionen 1 und 3 aktiv, die übrigen drei Partitionen 2, 4 und 5 dagegen inaktiv sind.

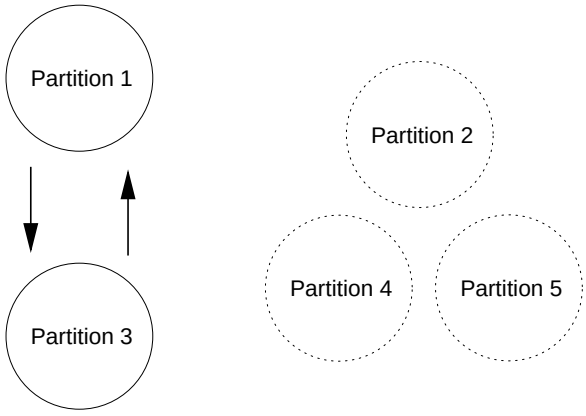

Abbildung 15 - aktive und inaktive Partitionen eines Workflows

Beim Starten einer Aktivität, die in einer nicht aktiven Partition liegt, ist es nun nicht einfach möglich, lediglich Änderungsinformationen an diese Partition zu propagieren, da sie nicht in die Kommunikation eingebunden ist. Zunächst ist es erforderlich, die Partition als solche zu aktivieren; anschließend kann dann der bereits beschriebene Mechanismus beim Start von Aktivitäten genutzt werden. Im einzelnen müssen die folgenden Bearbeitungsschritte durchgeführt werden:

- Zunächst muß die Partition auf einem der ihr zugeordneten Rechner als Prozeß instantiiert werden, das heißt, das ihr in der Konfigurationsdatei zugeordnete Programm muß auf diesem Rechner gestartet werden.
- Die Kommunikationsverbindungen aller bisher aktiven Partitionen mit der neu hinzugenommenen Partition müssen auf DT-Ebene eingerichtet werden.
- Der aktuelle Zustand des Workflows, das heißt die aktuellen Variablenwerte sowie Informationen über die betretenen Zustände und die gestarteten Aktivitäten jeder Partition, muß an die neue Partition übermittelt werden

Es genügt nicht, wenn dieser Startvorgang unmittelbar beim Starten einer Aktivität in einer nicht aktiven Partition angestoßen wird. Statt dessen ist eine Koordination aller solcher Vorgänge in der gesamten Workflowinstanz notwendig, da es zum Beispiel vorkommen kann, daß in zwei verschiedenen Partitionen innerhalb des gleichen Schritts eine Aktivität in der gleichen nicht aktiven Partition gestartet werden soll. Hier sind besondere Maßnahmen notwendig, damit nicht zwei Instanzen der gleichen Partition gestartet werden.

Der detaillierte Ablauf dieser Einbindung ist eng mit der Synchronisation der Partitionen verzahnt, er wird daher erst im folgenden Kapitel beschrieben, das sich mit der Synchronisation innerhalb einer Workflowinstanz befaßt.

3.1.8. Beenden einer Partition

Ein eng mit dem Starten von Partitionen zusammenhängendes Problem ist ihr Beenden. Man möchte nach Möglichkeit vermeiden, daß Partitionen, in denen sich keine gestartete Aktivität mehr befindet, bis zum Ende der Workflowausführung weiterläuft, da sie während der gesamten Zeit Ressourcen belegt und Nachrichten erhält, so daß die Ausführung der übrigen Partitionen potentiell behindert wird. Zunächst ist dazu die Frage zu klären, wann eine Partition beendet werden sollte, wenn sie keine gestarteten Aktivitäten mehr enthält:

- *Vorschlag 1*: unmittelbar nach Beenden der letzten gestarteten Aktivität und Verlassen des letzten betretenen Zustandes. Hier stellt sich das Problem, daß die Partition erneut instantiiert werden muß, wenn bei der folgenden Workflowausführung erneut eine Aktivität in dieser Partition gestartet wird. Dieser Vorgang kann möglicherweise ressourcenaufwendiger sein als das bloße Weiterlaufen der Partition, so daß man zwischen den beiden Möglichkeiten abwägen muß. Unmittelbares Abbrechen kann man auch als “optimistischen Ansatz” bezeichnen, weil man dann davon ausgeht, daß keine Aktivität in dieser Partition mehr aktiv werden wird.
- *Vorschlag 2*: erst am Ende der Workflowausführung. Untätige Partitionen werden hier in keinem Fall abgebrochen, sondern belegen Ressourcen immer bis zum Ende der Ausführung. Dadurch vermeidet man Reinstantiierungen von Partitionen, verschenkt aber möglicherweise Leistung des Gesamtsystems. Analog zur obigen Methode kann man hier von einem “pessimistischen Ansatz” sprechen, da man davon ausgeht, daß auf jeden Fall wieder Aktivitäten in dieser Partition gestartet werden und sich somit ein Beenden nicht lohnt.
- *Vorschlag 3*: wenn keine Aktivitäten dieser Partition jemals wieder aktiv werden. Diese Methode ist sicher in dem Sinn optimal, daß sie keine aufwendigen Reinstantiierungen verursacht. Sie erfordert aber einen “Blick in die Zukunft”, für den aufwendige Analysen der Spefizikation erforderlich sind. Sind zur Laufzeit dynamische Änderungen der Work-

flowspezifikation möglich, müssen diese Analysen außerdem nach jeder solchen Änderung erneut durchgeführt werden.

Der Kommunikationsmanager unterstützt prinzipiell alle diese Methoden, wenn ihm entsprechende Entscheidungsalgorithmen zur Verfügung gestellt werden. Im Rahmen dieser Arbeit wurde exemplarisch die pessimistische Methode implementiert. Ähnlich wie beim Starten von Partitionen ist auch für das Beenden ein vergleichbares Protokoll notwendig, das ebenfalls am Ende des Kapitels über Synchronsiation behandelt wird.

3.2. Implementierung der CM-Schicht

3.2.1. Verwaltung der Variablen, Aktivitäten und Zustände

Die CM-Schicht arbeitet mit eigenen Variablenstrukturen, die von denen der Workflow Engine vollständig unabhängig sind. Insbesondere werden nur fünf elementare Datentypen unterstützt: Zeichenketten (Strings), ganze Zahlen (Integers), Gleitkommazahlen (Doubles), Bedingungen (Conditions) und Ereignisse (Events). Zusammengesetzte Datentypen wie Arrays und Strukturen werden nicht unterstützt, sind aber in diesem Zusammenhang auch nicht erforderlich, da die beiden betrachteten Workflow Engines sie auch nur eingeschränkt anbieten: Der Chart Interpreter kennt keine zusammengesetzten Datentypen, und Statemate bildet bereits bei der Codegenerierung solche Datentypen auf elementare Datentypen ab.

Zur Verwaltung der Variablen wird die folgende Datenstruktur `cmVar` verwendet, die für jede Variable einer Partition einmal instantiiert wird:

```
typedef struct
{ LSNode   node;
  int      stype;
  char    *name;
  void    *address;
  int      type;
  int      changed;
  cmVarSubscriberPtr firstproc;
  union
  { char  *val_string;
    int    val_integer;
    double val_double;
    char   val_condition;
  } value;
  union
  { char  *val_string;
    int    val_integer;
    double val_double;
    char   val_condition;
  } newvalue;
} cmVar;
```

Die Strukturen sind über den Eintrag `node` in einer doppelt verketteten Liste miteinander verbunden. Die Funktionen und Strukturen für die Verwaltung dieser Listen wurden dem vom Autor vor längerer Zeit entwickelten ListSupport-Paket entnommen. Mittels des Eintrags `stype` werden die

Strukturen als Blätter der Suchbäume gekennzeichnet, sein Wert ist in diesem Fall `STYPE_LEAF`. Der Eintrag `name` enthält den eindeutigen Namen der Variablen, der Eintrag `address` zeigt auf die zugehörige Datenstruktur der Workflow Engine. Dieser Zeiger wird nicht von der CM-Schicht genutzt, sondern kann von der WEI-Schicht gesetzt werden, falls er dort benötigt wird. Der Eintrag type dient der Unterscheidung der elementaren Datentypen, dazu sind die Konstanten `cmVT_String`, `cmVT_Integer`, `cmVT_Double`, `cmVT_Condition` und `cmVT_Event` mit den offensichtlichen Bedeutungen definiert. Die Union `value` enthält den letzten Wert der Variablen, der an die übrigen Partitionen propagiert wurde, in `newvalue` steht der aktuelle Wert. Sollte sich dieser aktuelle Wert gegenüber dem letzten propagierten Wert geändert haben, wird das in `changed` vermerkt. Alle Partitionen, die über Änderungen einer Variablen informiert werden müssen, werden in einer einfach verketteten Liste von `cmVarSubscriber`-Strukturen gespeichert, auf die `firstproc` zeigt; diese Strukturen enthalten lediglich einen Zeiger auf die `cfgProcInfo`-Struktur (s. Anhang B) des entsprechenden Teilprozesses dieser Partitionen:

```
typedef struct
{ cmVarSubscriberPtr next;
  cfgProcInfoPtr     proc;
} cmVarSubscriber, *cmVarSubscriberPtr;
```

Wie sich im folgenden zeigt, ist es notwendig, effiziente Suchmöglichkeiten auf den Variablen der CM-Schicht zu haben. Insbesondere wird häufig mit dem Namen der Variablen als Schlüssel gesucht, etwa wenn eine Nachricht einer anderen Partition über eine Änderung der Variablen ausgewertet wird. Wie sich bei der Diskussion der WEI-Schicht für Statemate zeigt, wird dort außerdem oft mit der Adresse der zugehörigen Struktur in Statemate als Schlüssel gesucht. Da andere Suchzugriffe nicht vorkommen, ist es sinnvoll, den Suchalgorithmus für diese beiden Zugriffsmuster möglichst effizient zu gestalten. Effiziente Suche in logarithmischer Zeit erlauben zum Beispiel balancierte binäre Suchbäume. In diesem Fall ist es einfach, diese Bäume aufzubauen und balanciert zu halten, da sich die Menge der Schlüssel - zum Beispiel die Namen der in der Partition verwendeten Variablen - zur Laufzeit nicht mehr ändert. Um den Suchbaum mit dem Variablennamen als Schlüssel aufzubauen, genügt es, die Liste der Variablen nach dem Namen zu sortieren und daraus rekursiv einen binären Baum aufzubauen. Für die Sortierung wurde der Quicksort-Algorithmus ([CLR90]) benutzt, der gegenüber der in der Literatur beschriebenen Variante für die Sortierung von Listen angepaßt wurde. Die beiden Suchbäume werden blattorientiert aufgebaut, wobei die Blätter gerade die Strukturen selbst sind, die am Eintrag `stype` erkannt werden.

Die Liste der Variablen der CM-Schicht wird während der Initialisierung der WEI-Schicht aufgebaut, außerdem werden die beiden Suchbäume für Namen und Adressen eingerichtet. Die Liste selbst steht dann in der Variablen `cmVarList`, eine der erwähnten `LSList`-Strukturen. Die Suche auf diesen Datenstrukturen wird über die Funktionen `FindVarByName()` zum Suchen einer Variablen anhand ihres Namens und `FindVarByAddress()` zum Suchen anhand der Adresse durchgeführt.

Es bleibt noch zu erwähnen, wie die Liste der “Abonnenten” einer Variable aufgebaut wird, also der Partitionen, die eine Variable ebenfalls verwenden und daher über Änderungen informiert werden müssen. Bereits beim Partitionieren wird dazu zu jeder Partition eine Datei erzeugt, in der alle verwendeten Variablen enthalten sind. Wenn im Rahmen der Initialisierung der CM-Schicht die Konfigurationsstrukturen der Partitionen, Ausführungsorgane und Teilprozesse aufgebaut werden, wird diese Datei für alle aktiven Partitionen ausgelesen und ein entsprechender Eintrag bei allen Variablen vorgenommen, die darin enthalten sind.

Neben den Variablen einer Partition müssen auch die Aktivitäten verwaltet werden, wofür die CM-Schicht die im folgenden gezeigte cmActivity-Struktur benutzt:

```
typedef struct cmActivity
{ LSNode node;
  int   stype;
  char *name;
  void *address;
  int   roleid;
  int   changed;
} cmActivity;
```

node dient auch hier einer doppelten Verkettung der Strukturen. Der Name der Aktivität wird in name, die Adresse der zugehörigen Struktur der Workflow Engine in address gespeichert. roleid enthält die Nummer der Rolle, die diese Aktitivät ausführt; in changed wird das Starten der Aktivität im aktuellen Schritt vermerkt. Die Liste dieser Strukturen wird ähnlich wie die Variablenstrukturen bei der Initialisierung der WEI-Schicht angelegt, sie steht anschließend in der Variablen cmActivityList. Zur Suche in dieser Liste existieren analog zur Variablenliste die Funktionen FindActByName() und FindActByAddress().

Die letzte Struktur, die in diesem Zusammenhang erwähnt werden muß, ist die cmState-Struktur. Sie dient, wie der Name schon nahelegt, der Verwaltung der Zustände der Partition:

```
typedef struct
{ cmStatePtr next;
  char *name;
  char *condition;
} cmState, *cmStatePtr;
```

Im Gegensatz zu den bisherigen Strukturen sind die Listenelemente nur über den Zeiger next einfach verkettet und auch nicht über Suchbäume zugreifbar, da die einzige Funktion, die diese Strukturen benutzt, immer linear über alle Strukturen iteriert. name zeigt wie üblich auf den Namen, condition zeigt auf den Namen der diesem Zustand zugeordneten Bedingung, die das Betretensein des Zustandes anzeigt. Durch die Benutzung dieser Bedingung kann vermieden werden, auf die Strukturen der Workflow Engine selbst zuzugreifen. Speziell im Fall von Statemate wäre das schwierig gewesen, da dort im generierten Code Zustände nicht durch einzelne Strukturen repräsentiert werden. Gemäß der später in Abschnitt 3.2.7 gezeigten Anwendung dieser Strukturen werden nicht für alle Zustände der Partition, sondern nur für die Zustände der Originalspezifikation, die dieser Partition zugeordnet sind, Strukturen angelegt. Für einen Zustand S, der in der Originalspezifikation vorkommt, wird also in genau einer Partition eine Struktur angelegt. Der Name der zugeordneten Bedingung ist in diesem Fall IN_S.

3.2.2. Grundsätzliche Codierung der CM-Schicht

Die CM-Schicht ist im wesentlichen als Endlosschleife konzipiert, in der nacheinander die Lese-, Ausführungs- und Schreibphase ausgeführt werden. Der prinzipielle Aufbau der Funktion cmMain(), die diese Funktionalität beinhaltet, ist in Abbildung 16 gezeigt. Dabei sind die einzelnen Phasen noch nicht als Code ausgeführt, ihre Codierung wird in den folgenden Abschnitten behandelt.

Der Abbruch der Hauptschleife bedeutet gleichzeitig das Beenden der Ausführung der Partition. Die Implementierung dieses Abbruchs wird im Rahmen der Diskussion der Implementierung des Beendens einer Partition in Abschnitt 3.2.7 behandelt.

```
static void cmMain(void)
{
 while (1) /* endlos wiederholen, bis expliziter Abbruch */
 {
   LESEPHASE();

   AUSFÜHRUNGSPHASE();

   SCHREIBPHASE();
 }

}
```

Abbildung 16 - Aufbau der Funktion `cmMain`()

3.2.3. Implementierung der Lesephase

In der Lesephase müssen solange die eingehenden Nachrichten gelesen werden, bis alle erwarteten Nachrichten erhalten worden sind. Wie bereits im vorangegangenen Kapitel gezeigt wurde, ist dazu eine Synchronisation der Partitionen notwendig, die erst im nächsten Kapitel behandelt wird. Dieser Abschnitt befaßt sich daher vorrangig mit der Struktur der Lesephase und der Auswertung der eingehenden Nachrichten.

Abbildung 17 zeigt den Aufbau der Lesephase. Sie besteht im wesentlichen aus zwei Schleifen: Die äußere Schleife wird solange durchlaufen, bis die Partitionen synchronisiert sind. Dieses stellt die Funktion `cmCheckSyncProcs()` fest, die im nächsten Kapitel über Synchronisation vorgestellt wird. In jedem Durchlauf der äußeren Schleife werden in der inneren Schleife solange transaktionsgeschützt Nachrichten gelesen und verarbeitet, bis keine mehr anstehen. Wurde mit den verarbeiteten Nachrichten noch keine Synchronisation erreicht, wird die Transaktion abgebrochen und nach einer Wartezeit, um die Belastung des Systems durch Lesevorgänge zu reduzieren, ein weiterer Versuch unternommen.

Bei der Verarbeitung der Nachrichten in der inneren Schleife werden verschiedene Typen von Nachrichten unterschieden, die sich im ersten Zeichen unterscheiden. Nachrichten, die der Synchronisation dienen, beginnen mit ‘*’ und werden von der Funktion `cmHandleSyncMsg()` ausgewertet, die im folgenden Kapitel über Synchronisation beschrieben wird. Nachrichten, die mit ‘$’ beginnen, sind Kontrollnachrichten, die etwa beim Einbinden von neuen Partitionen zur Laufzeit verwendet werden; sie werden in den entsprechenden Kapiteln bei Bedarf eingeführt. Zu beachten ist hierbei allerdings, daß sie nicht sofort ausgeführt, sondern zunächst durch die Funktion `cmAddCtrlMessage()` zwischengespeichert und erst nach Abschluß der Lesephase durch die Funktion `cmHandleCtrlMsgs()` ausgeführt werden. Im Gegensatz zu Änderungsinformationen dürfen nämlich Kontrollnachrichten (z.B. “Binde Partition x in die Kommunikation ein”) im Falle eines Abbruchs der Lesetransaktion nicht mehrmals ausgeführt werden, wenn sie im anschließenden Schleifendurchlauf erneut gelesen werden. Die Funktion `cmAddCtrlMessage()` hängt die übergebene Kontrollnachricht an die Liste der bereits eingegangenen Kontrollnachrichten an, auf

```
while (1)
{ dti_ta_begin();
  reading_messages=1;
  do
  {
    dti_readmsg(message);
    if (message[0]==0) /* keine Nachricht */
      reading_messages=0; /* Schleife beenden */
    else if (message[0]=='*') /* Sync-Nachricht */
      cmHandleSyncMessage(message);
    else if (message[0]=='$') /* Kontrollnachricht */
      cmAddCtrlMessage(message);
    else /* Nachricht ist Änderungsinformation */
      cmConsumeMessage(message);
  } while (reading_messages);
  if (cmCheckSyncProcs())
  {
   cmHandleCtrlMsgs();
    break;
  }
  else
  { dti_ta_abort();
    cmClearCtrlMsgs();
    sleep(SLEEP_AMOUNT);
  }
}
```

Abbildung 17 - Aufbau der Lesephase in `cmMain()`

die erste dieser Kontrollnachrichten zeigt die Variable `cmCTRLList`. Die Funktion und die Definition der von ihr genutzten Struktur wird im folgenden gezeigt:

```
typedef struct
{ cmCTRLPtr    next; /* nächste Struktur in der Liste */
  char        *msg;  /* Nachricht als String */
} cmCTRL, *cmCTRLPtr;
cmCTRLPtr cmCTRLList;

void cmAddCtrlMessage(char *message)
{
  struct cmCTRLPtr c,ctrl=malloc(sizeof(cmCTRL));

  ctrl->msg=strmove2(message);

  if (cmCTRLList==NULL) cmCTRLList=ctrl;
  else
  { c=cmCTRLList;
    while (c->next) c=c->next;
    c->next=ctrl;
  }
}
```

Man beachte dabei insbesondere, daß die Kontrollnachrichten in der Reihenfolge ihres Eintreffens in der Liste gespeichert werden. Die Funktion `strmove2()` ist Teil der Support-Bibliothek des Mentor-Prototypen und dient dem Kopieren einer Zeichenkette. Damit gewährleistet ist, daß Kontrollnachrichten wirklich nur einmal ausgeführt werden, muß die Liste der Kontrollnachrichten gelöscht werden, wenn die Lesetransaktion abgebrochen wird, dies erledigt die Funktion `cmClearCtrlMsgs()`:

```
void cmClearCtrlMsgs(void)
{ cmCTRLPtr c,c1;

  c=cmCTRLList;
  while (c)
  { c1=c->next;
    free(c->msg);
    free(c);
    c=c1;
  }
  cmCTRLList=NULL;
}
```

Weitere Nachrichtentypen, die eine besondere Behandlung erfordern, werden im folgenden bei Bedarf eingeführt werden.

Alle übrigen Nachrichten, die nicht in eine der genannten Kategorien fallen, sind Informationen über Änderungen von Variablen in einer anderen Partition. Sie werden unmittelbar ausgewertet, das bedeutet, die darin enthaltene Änderung wird sofort an die lokale Workflow Engine propagiert. Im Gegensatz zu Kontrollnachrichten kommt es dabei nicht zu Schwierigkeiten, wenn die Lesetransaktion abgebrochen wird, da dann beim nächsten Leseversuch die gleichen Nachrichten erneut verarbeitet und somit die gleichen Änderungen nochmals propagiert werden, was die Korrektheit nicht beeinflußt. Natürlich könnte man auch hier Nachrichten zwischenspeichern und die enthaltenen Änderungen erst nach Abschluß der Lesephase propagieren, es ist jedoch nicht erforderlich.

Nachrichten über Änderungen von Variablenwerden haben grundsätzlich das Format

Variablenname#neuer_Wert#Schrittnummer#AbsenderID

Die Bedeutung von `Variablenname` und `neuer_Wert` sind offensichtlich. `Schrittnummer` ist die Nummer des lokalen Schritts, in der die Änderung eingetreten ist; sie wird erst bei der Synchronisation verwendet. `AbsenderID` ist die ProcID des Teilprozesses, der die Nachricht verschickt hat; sie dient nur der Fehlersuche.

Die Auswertung dieser Nachrichten übernimmt die in Abbildung 18 gezeigte Funktion `cmConsumeMsg()`. Zu Beginn dieser Funktion werden zunächst die einzelnen Teile der Nachricht durch Suchen des Begrenzerzeichens '#' isoliert, dieser Teil der Funktion ist trivial und wird daher hier nicht wiedergegeben. Die Zeiger `name_str`, `val_str`, `step_str` und `sender_str` zeigen auf die entsprechenden Teile im Eingabestring, die Begrenzerzeichen wurden durch Nullbytes als Stringendekennzeichen ersetzt. Anschließend wird die Variablenstruktur des CM gesucht, die zu der geänderten Variable gehört; wird sie nicht gefunden, wird die Funktion verlassen[1]. Jetzt wird die

[1] Normalerweise wird hier eine Fehlermeldung ausgegeben, da es wegen des gezielten Versendens von Änderungsinformationen nicht vorkommen darf, daß eine Partition die Änderung einer ihr unbekannten Variable erhält. Dies deutet immer auf einen systematischen Fehler bei der Partitionierung hin.

```
void cmConsumeMsg(char *msg)
{
  cmVarPtr cv;
  char *name_str, *val_str, *step_str, *sender_str;

  /* hier werden diese Zeiger gesetzt */

  /* Suchen der zugeordneten Datenstruktur */

  cv=FindVarByName(name_str);

  if (cv==NULL) /* unbekannte Variable */
    return; /* wird ignoriert */

  /* jetzt wird die Änderung propagiert */

  switch (cv->type)
  { case cmVT_String:
      weiSetString(cv,val_str);
      break;
    case cmVT_Integer:
      weiSetInteger(cv,atoi(val_str));
      break;
    case cmVT_Double:
      weiSetDouble(cv,atof(val_str));
      break;
    case cmVT_Condition:
      weiSetCondition(cv,(char)atoi(val_str));
      break;
    case cmVT_Event:
      weiGenerate(cv);
      break;
    default:
      /* Fehlerbehandlung */
  }

  StepHasChanges++; /* Anzahl der Gesamtänderungen in
                       diesem Schritt mitzählen */
}
```

Abbildung 18 - Aufbau der Funktion `cmConsumeMsg()`

Änderung der Variable mit den entsprechenden WEI-Funktionen an die Workflow Engine propagiert. Abschließend wird noch der Zähler `StepHasChanges` aktualisiert, der die Gesamtzahl der Änderungen in der Partition in diesem Schritt zählt. Dieser Zähler wird später bei der Synchronisation eine wichtige Rolle spielen.

3.2.4. Implementierung der Ausführungsphase

In der Ausführungsphase wird zunächst eine Transaktion begonnen, die erst am Ende der Schreibphase beendet wird, damit lokaler Schritt und Propagation der Änderungen atomar ablaufen. Anschließend setzt die Funktion `cmResetVars()` die `changed`-Flags aller Variablen- und Aktivitätsstrukturen zurück, damit die in diesem Schritt geänderten Variablen und Aktivitäten nach der Ausführungsphase am gesetzten `changed`-Flag erkannt werden können. Nun wird die Funktion `weiStep()` der WEI-Schicht aufgerufen, was genau einen Schritt der Workflow Engine bewirkt. Sollten dabei Änderungen von Variablen auftreten oder Aktivitäten gestartet oder beendet werden, werden durch die WEI-Schicht Callback-Funktionen der CM-Schicht aufgerufen, die im folgenden vorgestellt werden.

Die Änderung einer Variablen führt zu einem Aufruf der Funktion `cmChangeCallback()`, die in Abbildung 19 gezeigt ist. Sie erhält als Parameter von der WEI-Schicht einen Zeiger auf die CM-Variablenstruktur der geänderten Variablen sowie einen Zeiger auf den neuen Wert der Variablen. Abhängig vom Variablentyp setzt sie den `newValue`-Eintrag der Struktur, außerdem markiert sie die Variable als geändert, indem sie das `changed`-Flag in der Struktur setzt. Abschließend wird noch der Zähler erhöht, der die Änderungen im aktuellen Schritt der Partition zählt.

```
void cmChangeCallback(cmVarPtr cv, void *value)
{
  switch (cv->type)
  { case cmVT_String:
      cv->newValue.val_string=*((char **)value);
      break;
    case cmVT_Integer:
      cv->newValue.val_integer=*((int *)value);
      break;
    case cmVT_Double:
      cv->newValue.val_double=*((double *)value);
      break;
    case cmVT_Condition:
      cv->newValue.val_condition=*((char *)value);
      break;
    case cmVT_Event:
      /* hier muß kein Wert gesetzt werden */
      break;
    default:
      /* Fehlerbehandlung */
  }

  cv->changed=1;

  StepHasChanges++;
```

Abbildung 19 - Die Funktion `cmChangeCallback()`

```
void cmStartActivityCallback(cmActivityPtr a)
{
  a->changed=cmActStarted;

  StepHasChanges++;
}
```

Abbildung 20 - Die Funktion `cmStartActivityCallback()`

Beim Starten von Aktivitäten ruft die Workflow-Engine-Interface-Schicht die Funktion `cmStartActivityCallback()` auf, die in Abbildung 20 dargestellt ist. Hier wird lediglich in der CM-Struktur der Aktivität, die als Parameter übergeben wird, das `changed`-Flag auf einen Wert gesetzt, der die Aktivität als neu gestartet identifiziert, sowie der bereits bekannte Zähler der Änderungen hochgezählt.

3.2.5. Implementierung der Schreibphase

In der Schreibphase, die in Abbildung 21 dargestellt ist, werden die Änderungen von Variablen, die in der Ausführungsphase erkannt wurden, an die übrigen Partitionen propagiert. Gestartete und beendete Aktivitäten, die ebenfalls erkannt wurden, müssen wie bereits vorher beschrieben nicht propagiert werden. Abschließend wird die vorher begonnene Transaktion erfolgreich beendet. Wenn es im aktuellen Schritt keine Änderung gab, was an der Variablen `StepHasChanges` erkannt werden kann, wird nun noch eine variable Pause eingelegt, bevor wieder in die Lesephase eingetreten wird. Dadurch wird erreicht, daß “untätige” Workflows, in denen auf Aktionen des Benutzers gewartet wird, wenig Ressourcen belegen. Die Wartezeit steigt von einem Minimalwert linear auf einen Maximalwert an und wird sofort auf null gesetzt, wenn sich in einem Schritt eine Änderung ergibt. `SLEEP_MAX` und `SLEEP_AMOUNT` sind dabei Konstanten, die passend definiert werden müssen.

```
cmSendUpdates();

dti_ta_commit();

if (StepHasChanges==0)
{ if (delay<SLEEP_MAX) delay++;
  sleep(delay*SLEEP_AMOUNT);
}
else
  delay=0;
```

Abbildung 21 - Implementierung der Schreibphase

Man muß dabei beachten, daß höhere maximale Wartezeiten zwar die Belastung des Gesamtsystems durch ständiges, vergebliches Auslesen von Nachrichten reduzieren. Andererseits wird dadurch aber auch die Zeit erhöht, die die Partition im Mittel braucht, um auf eingehende Änderungen zu reagieren. Der Benutzer erkennt dies an der Zeit, die etwa von einem Klick in einer Oberfläche bis zur Reaktion des Workflows vergeht. Praktische Erprobungen haben ergeben, daß maximale Verzögerungszeiten bis zu 10 Sekunden noch tolerierbar sind, längere Zeiten lassen die

Workflowausführung träge erscheinen oder gar den Verdacht aufkommen, die Ausführung sei stehengeblieben.

Abbildung 22 zeigt die Funktion `cmSendUpdates()`, die die Änderungen der Variablen an die übrigen Partitionen propagiert. In einer Schleife werden dort alle Variablen getestet, ob sie im letzten Schritt geändert wurden, was am changed-Flag erkannt werden kann. Bei geänderten Variablen wird durch die Funktion `cmMakeUpdateMsg()` eine Nachricht in dem Format erzeugt, das bereits bei der Beschreibung der Lesephase vorgestellt wurde. Diese Nachricht wird dann an alle Teilprozesse geschickt, die über Änderungen dieser Variable informiert werden wollen, das sind gerade die in der `firstproc`-Liste der Variablenstruktur.

Hier besteht noch Potential für Optimierungen. Insbesondere kann man alle Nachrichten an eine Partition zu einer einzigen bündeln, da die einzelnen Nachrichten selbst sehr kurz sind. Der Kommunikationsoverhead zum Senden einer Nachricht wird dann nur einmal je Prozeß erforderlich. Für den erstellten Forschungsprototypen ist allerdings bereits die implementierte Variante ausreichend leistungsfähig.

Nach dem Versenden der Variablen muß nun noch geprüft werden, ob Aktivitäten in nicht gestarteten Partitionen gestartet wurden. Gestartete Aktivitäten können durch Prüfen der `changed`-Flags in den `cmActivity`-Strukturen erkannt werden, bei dieser Suche können diese Flags auch wieder zurückgesetzt werden. Wird eine gestartete Aktivität gefunden, kann ihre Rolle dem Eintrag `role` ihrer Struktur entnommen werden. Die Funktion `cfgFindRoleByName()` (s. Anhang B) liefert dann die zugehörige `cfgRoleInfo`-Struktur der Rolle.Die Funktion `cfgFindProc4Role()` sucht dann den Prozeß, der dieser Rolle zugeordnet ist. Findet sie keinen, ist die Partition noch nicht aktiv und muß gestartet werden; der genaue Ablauf dieses Starts wird in Kapitel 4.7 beschrieben. Andernfalls müssen wie bereits geschildert keine weiteren Maßnahmen getroffen werden, allein die Propagation der geänderten Variablen genügt.

3.2.6. Initialisierung des Kommunikationsmanagers

In diesem Abschnitt werden die Maßnahmen vorgestellt, die bei der Initialisierung des Kommunikationsmanagers getroffen werden. Im einzelnen sind dies in der Reihenfolge der Ausführung:

- Aufbau der Strukturen der Rollen, Ausführungsorgane und Prozesse mit der Funktion `cmReadConfig()`; die entsprechenden Strukturen werden in Anhang B vorgestellt
- Initialisierung der Datentransportschicht, gleichzeitig Aufbau von Kommunikationskanälen zu jedem der aktiven Prozesse
- Setzen von Zeigern auf die Strukturen von ausgezeichneten Prozessen: Zum einen `ThisProcPtr`, der auf die Struktur des eigenen Prozesses zeigt, und zum anderen Zeiger auf die Strukturen verschiedener Systemprozesse, die im weiteren Verlauf der Arbeit eingeführt werden.
- Initialisierung der Workflow Engine
- Ausführung der Partition durch Einsprung in die CM-Hauptschleife `cmMain()`

Diese Aufgaben übernimmt die Funktion `cm()`.

```
void cmSendUpdates(void)
{
  cmVarPtr cv;
  cmVarSubscriberPtr vs;
  char *msg;

  cv=LSFirstNode(&cmVarList);

  while (LSMoreNodes(&cv->node))
  {
    if (cv->changed)
    {
      msg=cmMakeUpdateMsg(cv);
      vs=cv->firstproc;

      while (vs)
      { dti_sendmsg(msg,0,vs->proc->procid);
        vs=vs->next;
      }
      free(msg);
    }
    cv=(cmVarPtr)LSNextNode(&cv->node);
  }
}
```

Abbildung 22 - Die Funktion `cmSendUpdates()`

3.2.7. Beenden einer Partition

Zwei Vorkommnisse können zum Ende der Ausführung einer Partition führen: Entweder ist die Ausführung des Gesamtworkflows beendet, oder keine Aktivität der Partition ist mehr aktiv und nach den Kriterien, die in Abschnitt 3.1.8 beschrieben wurden, wird die Partition beendet.

Der erste Fall, also das Ende der Ausführung des gesamten Workflows, wird erkannt, wenn in einer Partition der ausgezeichnete Zustand `WF_EXIT_S` betreten wird. Eine Partition erkennt dies, indem am Ende der Ausführungsphase die Bedingung IN_WF_EXIT_S, die genau dieses anzeigt, auf ihren Wert geprüft wird. Zur Vereinfachung dieses Vorgangs wird bei der Initialisierung der WEI-Schicht der Zeiger `cmExitCondition` auf die entsprechende Struktur dieser Variablen gesetzt, falls sie Teil der Partition ist. Ist die Bedingung wahr, wird an alle Partitionen eine besondere Kontrollnachricht, die sogenannte “Shutdown-Nachricht”, vom Format “$SHUTDOWN_ProcID” geschickt. ProcID ist dabei die ProcID der Partition, die die Beendigung initiiert hat. Alle Partitionen, die diese Nachricht lesen, beenden durch Verlassen der CM-Hauptschleife ihre Ausführung, nachdem durch die Funktion `dti_shutdown()` die Kommunikationsverbindungen abgebaut und durch die Funktion `weiExitWorkflowEngine()` die Workflow-Engine ordnungsgemäß beendet wurde.

Um zu erkennen, ob noch Aktivitäten der Partition aktiv oder Zustände betreten sind, genügt es, dies allein für die Zustände zu prüfen. Nach der in Kapitel 1 beschriebenen Spezifikationsregel für Workflows ist nämlich jeder Aktivität ein Zustand zugeordnet, der genau dann betreten ist, wenn die Aktivität gestartet wurde. Das "Nichtbetretensein" eines Zustandes `S` bezieht sich dabei auf die Ausgangsspezifikation, in der partitionierten Variante ist es gleichbedeutend mit dem Betretensein

des entsprechenden Zustandes `S_OUT` oder, äquivalent, damit, daß die dem Originalzustand zugeordnete Bedingung `IN_S` den Wert falsch hat.

Der Test, ob alle Zustände verlassen sind, geschieht durch die Funktion `cmCheckActiveStates()`. Sie prüft für alle Zustände anhand der in der entsprechenden, im Eingangsabschnitt dargestellten `cmState`-Struktur abgelegten Bedingung, ob der Zustand betreten oder nicht betreten ist. Sind alle Zustände nicht betreten, gibt sie 1 zurück. In der CM-Schicht kann dann, ein positives Resultat eines entsprechenden Entscheidungsalgorithmus vorausgesetzt, das Beenden der Partition eingeleitet werden. Die Funktion `cmCheckActiveStates()` wird in der Schreibphase nach dem Senden der Änderungsinformationen, aber vor dem Beenden der Transaktion eingefügt.

3.3. Implementierung der WEI für Statemate

Statemate bietet die Möglichkeit, für State- und Activitycharts C-Code zu generieren, der sich bei Ausführung genau wie die Charts verhält [i-Log94-c]. Das bedeutet, er reagiert genau wie die Ausgangscharts auf generierte Ereignisse, geänderte Variablen usw. Die Laufzeitbibliothek von Statemate bietet dabei Funktionen an, um aus C-Programmen heraus Einfluß auf die Ausführung der Charts zu nehmen, etwa durch Setzen von Bedingungen oder durch Generieren von Ereignissen. Außerdem ist es möglich, den Aktivitäten der Spezifikation eigene C-Funktionen zuzuordnen, die beim Starten der jeweiligen Aktivität aufgerufen werden. Erzeugter Code, Laufzeitbibliothek und eigener Code zusammen bilden dann nach Compilierung ein ausführbares Programm.

Im folgenden wird zunächst die Behandlung von Variablen im von Statemate generierten Code und ihre Integration in die Variablenstrukturen der CM-Schicht beschrieben. Anschließend werden die implementierten Callback-Funktionen und die Funktionen zur Änderung von Variablen behandelt. Abschließend werden die erforderlichen Änderungen im Statemate-Laufzeitsystem dargestellt, die zur Ausführung eines einzelnen Schrittes der Spezifikation notwendig waren.

3.3.1. Verwaltung der Variablen, Aktivitäten und Zustände

Statemate generiert bei der Codeerzeugung für jede Variable und jedes Ereignis, die in der Spezifikation vorkommen, eine eigene C-Variable gleichen Namens im Code. Den elementaren Datentypen in den Charts, die in dieser Arbeit betrachtet werden, werden dabei die folgenden Typen in C zugeordnet:

```
String          char *
Double          double
Integer         int
Condition       char
Event           char
```

Um die Datenstrukturen der CM-Variablen zu füllen, die im vorangegangenen Kapitel beschrieben wurden, ist insbesondere eine Zuordnung von den Namen der Variablen und Ereignisse in der Spezifikation zu den Adressen im Speicher der zugehörigen C-Variablen im erzeugten Code notwendig. Weder das Laufzeitsystem von Statemate noch die Programmiersprache C bieten ein geeignetes Konstrukt an, um zur Laufzeit anhand des Variablennamens die Adresse der Variable herauszufinden oder umgekehrt. Um eine solche Funktion `getaddress("Variablenname")` zu implementieren, wäre es erforderlich, ähnlich wie ein Debugger Informationen im ausführbaren

Programm zu analysieren. Diese Vorgehensweise ist für diese Problemstellung hier wesentlich zu aufwendig. Einfacher ist es, die gewünschten Datenstrukturen bereits zum Zeitpunkt der Compilierung zu erzeugen:

Bereits der Partitionierer kennt alle Variablen und Ereignisse, die in einer Partition verwendet werden. Er kann daher die Variablenstrukturen als C-Code erzeugen. Da die Namen der Variablen in Charts und Code gleich sind, genügt der C-Adressoperator "&", um die Adresse der C-Variablen zu einer Variablen im Chart zu erhalten. Die Zuordnung zur Laufzeit erledigen dann die bereits beschriebenen Funktionen der CM-Schicht zur Suche nach Variablen anhand ihres Namens oder ihrer Adresse. Abbildung 23 zeigt einen Ausschnitt aus einem solchen automatisch generierten C-Code für eine Partition des Lehrbericht-Workflows aus dem ersten Kapitel. Am Ende sieht man außerdem die beim Erkennen des Endes der Workflowausführung benutzte Variable `cmExitCondition`, die in diesem Fall `NULL` ist, da die Variable `IN_WF_EXIT_S` in dieser Partition nicht vorkommt.

Statemate erzeugt für jede Aktivität einer Spezifikation eine eigene C-Datenstruktur. Analog zur Vorgehensweise bei den Variablen werden die entsprechenden Datenstrukturen der CM-Schicht als Quellcode erzeugt und beim Compilieren eingebunden.

Die Vorgehensweise bei den Strukturen zur Zustandsverwaltung ist einfacher, da dort keine Verweise auf Strukturen von Statemate notwendig sind. Der Partitionierer erzeugt für jeden Zustand der Originalspezifikation, der der Partition zugeordet ist, eine entsprechende Struktur als Quellcode, der dann in der üblichen Weise beim Compilieren eingebunden wird.

Abbildung 23 enthält neben Variablenstrukturen auch generierte Strukturen zu Aktivitäten und Zuständen.

```
/* Communication Manager Information File Version 1.4
   created for role Fachschaft_AC (#4) on Sat Aug 23 15:59:26 1997 */

#include <cmconfig.h>

extern condition IN_STELLUNGNAHME_ZU_LEHRBERICHT_S;
extern condition IN_WEITERLEITUNG_D_LEHRBERICHTS_AN_S;
(...)
svbitem svblist[] = {
{ &svblist[1].node,NULL,STYPE_LEAF,"IN_STELLUNGNAHME_ZU_LEHRBERICHT_S",
  &IN_STELLUNGNAHME_ZU_LEHRBERICHT_S,SVBT_CONDITION,0,NULL},
{ &svblist[2].node,&svblist[0].node,STYPE_LEAF,"IN_WEITERLEITUNG_D_LEHRBERICHTS_AN_S",
  &IN_WEITERLEITUNG_D_LEHRBERICHTS_AN_S,SVBT_CONDITION,0,NULL},
(...)
};
LSList svbl =
{&svblist[0].node,NULL,&svblist[21].node,22};

extern activity LEHRB_U_STELLUNGN_AN_REF_61_SEND_ACT;
(...)
activityitem activitylist[] = {
{ &activitylist[1].node,NULL,STYPE_LEAF,"LEHRB_U_STELLUNGN_AN_REF_61_SEND_ACT",
  &LEHRB_U_STELLUNGN_AN_REF_61_SEND_ACT,6},
(...)
};
LSList actl =
{&activitylist[0].node,NULL,&activitylist[3].node,4};

stateinfo statelist[] = {
{ NULL,NULL,STYPE_LEAF,"STELLUNGNAHME_ZU_LEHRBERICHT_S",&IN_Stellungnahme_zu_Lehrbericht_S}
};
LSList statel =
{ &statelist[0].node,NULL,&statelist[0].node,1};

char *cmExitCondition=NULL;
```

Abbildung 23 - Ausschnitt aus den automatisch generierten CM-Variablenstrukturen

3.3.2. Implementierung von Callbacks

Das Laufzeitsystem von Statemate bietet selbst die Möglichkeit, Callback-Funktionen zu definieren, die bei der Änderung bestimmter Variablen, beim Generieren bestimmter Ereignisse oder beim Starten bestimmter Aktivitäten aufgerufen werden. Diese Statemate-Callbacks haben aber zwei entscheidende Nachteile, die ihre Verwendung für den Kommunikationsmanager ausschließen:

- Zum einen können Callbacks nicht für eine ganze Klasse von Änderungen - etwa für alle Änderungen von Variablen -, sondern nur für jeweils ein ganz bestimmtes Statemate-Element eingesetzt werden. Für jede Variable, jedes Ereignis und jede Aktivität muß daher eine Callback-Funktion beim Laufzeitsystem angemeldet werden. Da eine Callback-Funktion auch mehreren Variablen zugeordnet werden kann, stellt dies kein wesentliches Problem da, macht aber die Initialisierung der Callbacks aufwendig.
- Zum anderen sind die vorhandenen Callbacks von ihrer Semantik her für diese Anwendung unbrauchbar. Wird zum Beispiel eine Variable geändert, feuert der Callback nicht sofort, sondern erst im folgenden Schritt, so daß die CM-Schicht erst einen Schritt zu spät über eine Änderung informiert wird.

Aus diesem Grund mußte ein eigener Callback-Mechanismus in das Laufzeitsystem von Statemate integriert werden. Das war ohne wesentliche Probleme möglich, weil der Quellcode vorlag und entsprechend behutsam erweitert werden konnte.

Für jeden Variablentyp gibt es in der Statemate-Laufzeitbibliothek eine Funktion, über die alle Änderungen von Variablen dieses Typs abgewickelt werden: `sets()` für `string`, `setd()`, `seti()`, `setc()` und `gen()` analog für `double`, `integer`, `condition` und `event`. Diese Funktionen wurden umbenannt, so daß sie nun alle den Präfix `stm` tragen, also in `stmsets()` usw. Statt dessen wurden neue Funktionen unter den alten Namen als Teil der WEI-Schicht für Statemate entworfen. Die neu geschaffene Funktion `seti()` wird exemplarisch in Abbildung 24 gezeigt.

```
int seti(int *var, int value)
{ cmVarPtr cm;

  stmseti(var,value); /* propagate locally */

  cm=FindVarByAddress(var);
  if (cm)
  {
    cmChangeCallback(cm);
    return 1;
  }
  else
  { /* error management skipped */
    return 0;
  }
}
```

Abbildung 24 - Die neue Funktion `seti()`

Hier wird zunächst die Originalfunktion der Laufzeitbibliothek aufgerufen, um die Änderung lokal zu propagieren. Anschließend wird die CM-Variablenstruktur gesucht, die zu der Variable gehört. Wird sie gefunden, wird die Callback-Funktion der CM-Schicht aufgerufen, andernfalls ein Fehler ausgegeben.

Da die Originalfunktionen in der Laufzeitbibliothek erhalten geblieben sind, ist es möglich, lokale Änderungen von Variablen am Kommunikationsmanager "vorbei" vorzunehmen. Diese Vorgehensweise ist zum Beispiel notwendig, wenn der Logmanager nach dem Ausfall einer Partition alle Änderungen, die in dieser vorgenommen wurden, nachfährt; dabei sollen keine Nachrichten mehr an die übrigen Partitionen geschickt werden, da das bereits vor dem Fehlerfall getan wurde. Diese Möglichkeit würde nicht mehr bestehen, wenn der Aufruf des CM-Callbacks unmittelbar in die Bibliotheksfunktionen von Statemate integriert worden wäre.

Die Behandlung von Aktivitäten erfolgt analog zur Behandlung von Variablen. Das Laufzeitsystem enthält eine Funktion `start()`, die beim Starten einer Aktivität aufgerufen wird. Diese wird wie oben in `stmstart()` umbenannt und eine eigene Funktion `start()` implementiert, die zunächst die Originalfunktion aus dem Laufzeitsystem aufruft, dann die zu der Aktivität gehörende `cmActivity`-Struktur sucht und anschließend die Callback-Funktion `cmStartActivityCallback()` der CM-Schicht aufruft.

3.3.3. Implementierung der Änderungsfunktionen

Die Implementierung der Änderungsfunktionen `weiSetInteger()` usw. ist sehr einfach, da die zum Aufruf der entsprechenden Funktion des Statemate-Laufzeitsystems benötigte Adresse bereits in der übergebenen CM-Variablenstruktur enthalten ist:

```
void weiSetInteger(cmVarPtr var, int value)
{
  stmseti(var->address,value);
  cm->value.val_integer=value;
}
```

Die übrigen Funktionen sind analog aufgebaut. Man beachte hier, daß die Originalfunktionen des Laufzeitsystems und nicht die geänderten der WEI-Schicht aufgerufen werden. Änderungen, die über die `weiSet*()`-Funktionen lokal ausgeführt werden, werden also nicht an die übrigen Partitionen propagiert.

3.3.4. Ausführen eines lokalen Schritts

Im Laufzeitsystem von Statemate gibt es die Funktion `pr_make_step()`, die das Ausführen genau eines Schritts in der Spezifikation ermöglicht. Diese Funktion wird normalerweise aus der Statemate-Hauptschleife heraus aufgerufen. In der vorliegenden Version kann sie jedoch nicht genutzt werden, es mußten Änderungen vorgenommen werden, die in Abbildung 25 zusammen mit der Originalfunktion gezeigt werden.

```
void pr_make_step()
{ /* original */
  lo_main();
  scheduler();
  sched_disable();
  update();
  update_CEs();
  sched_enable();
  pge_start_graphics();
  call_cbks();
  pge_end_graphics();
  scheduler();
}
```

```
void pr_make_step()
{ /* geändert */
  sched_disable()
  update();
  update_CEs()
  lo_main();
  sched_enable();
  scheduler();
  pge_start_graphics();
  call_cbks();
  pge_end_graphics();
  scheduler();
}
```

Abbildung 25 - Original und geänderte Funktion `pr_make_step()`

In der Originalfunktion wird zunächst ein Schritt in der Spezifikation mittels der Funktion `lo_main()` ausgeführt, die Teil des generierten Codes ist. Anschließend werden die dabei entstandenen Änderungen, die zuerst nur zwischengespeichert wurden, durch die Funktionen `update()` und `update_CEs()` ausgeführt. Die Aufrufe der Funktionen `scheduler()`, `sched_enable()` und `sched_disable()` steuern ein Funktionspaket des Laufzeitsystems, das Multithreading implementiert; diese Funktionalität wird im Rahmen der Verwendung als Workflow-Engine im Zusammenhang mit der Anbindung von Benutzeroberflächen [Bie97] genutzt. Die Aufrufe der mit `pge` beginnenden Funktionen dienen der Steuerung der Panels, das sind Benutzungsoberflächen, die von Statemate zur Verfügung gestellt werden. `call_cbks()` schließlich ruft die vereinbarten Statemate-Callbacks auf, die hier aus den zuvor genannten Gründen keine Verwendung finden.

Damit diese Funktion als Schrittfunktion verwendet werden kann, muß die Reihenfolge von `lo_main()` und den beiden Update-Funktionen vertauscht werden. Das liegt daran, daß zunächst die von anderen Partitionen empfangenen Änderungen, die während der Lesephase lokal propagiert werden, noch ausgeführt werden müssen, bevor ein Schritt gemacht werden kann. In der Originalreihenfolge würde zuerst ein Schritt mit den alten Variablenwerten gemacht, bevor anschließend durch die beiden Update-Funktionen die neuen Werte angenommen würden. Durch die Verschiebung der Update-Funktionen vom Ende an den Anfang der Schrittfunktion geben die Statemate-Variablenstrukturen während der Schreibphase nicht unbedingt den aktuellen Wert der Variablen wieder. Das schadet aber nichts, da der Kommunikationsmanager in seinen eigenen Strukturen selbst Buch über den aktuellen Wert der Variablen führt.

3.4. Implementierung der WEI für den Chart Interpreter

Der Chart Interpreter wurde als Teil von MENTOR speziell für die interpretative Ausführung von Workflowpartitionen in Zusammenarbeit mit dem Kommunikationsmanager konzipiert. Viele Schwächen von Statemate wurden dabei umgangen, so daß der Interpreter ideal als Workflow Engine integriert werden kann. Wie im vorangegangenen Abschnitt werden nun die einzelnen Aspekte dieser Integration und damit der Aufbau der WEI-Schicht für den Interpreter beschrieben.

3.4.1. Variablenverwaltung

Der Interpreter baut erst zur Laufzeit interne Datenstrukturen auf, in denen sich die Charts, das heißt Zustände, Zustandsübergänge usw., sowie Variablen und Ereignisse widerspiegeln. Zusätzlich bietet er besondere Funktionen zum Zugriff auf Variablenstrukturen sowohl anhand ihres Namens, als auch anhand ihrer Adresse. Liegt etwa die Struktur, die zur Variable TEST vom Typ integer gehört, an Adresse 0xf5632, kann der Wert der Variablen sowohl durch den Funktionsaufruf

```
siSetInteger(0xf5632,1);
```

als auch durch den Funktionsaufruf

```
siSetIntegerByName(“TEST”,1);
```

auf den Wert 1 geändert werden. Zusätzlich bietet der Interpreter eine Suchfunktion, um eine Variablenstruktur anhand des Namens der Variablen zu finden; ist die Adresse einer Variablenstruktur bekannt, kennt man auch den Namen der Variablen, der in der Struktur abgelegt ist.

Die CM-Variablenstrukturen der CM-Schicht können nun natürlich nicht mehr bei der Partitionierung erzeugt und compiliert werden, da die Strukturen der Variablen des Interpreters erst zur Laufzeit angelegt werden. Statt dessen werden diese Strukturen zur Laufzeit aus den Variablenstrukturen des Interpreters generiert, der dies durch eine verkettete Liste dieser Strukturen unterstützt.

Für die Aktivitäten gelten die analogen Aussagen. Die Zustandsstrukturen können durch Auswertung der entsprechenden Strukturen des Interpreters einfach gewonnen werden, da der Interpreter die ausgeführten Charts in von außen zugänglichen Strukturen hält.

3.4.2. Implementierung von Callbacks

Der Chart Interpreter unterstützt neben der Callbacksemantik von Statemate auch eigene, sogenannte “immediate”-Callbacks. Wie dieser Name schon nahelegt, “feuern” diese Callbacks unmittelbar, nachdem sich eine Änderung einer Variablen ereignet hat, also insbesondere noch während des gleichen Schritts. Darüber hinaus können Callbacks für ganze Klassen von Änderungen vereinbart werden, etwa solche, die bei jeder Änderung einer Variablen oder bei jedem Start einer Aktivität feuern. Diese beiden Eigenschaften zusammen erlauben einen sehr einfachen Aufbau der Callback-Funktionen, die vom Interpreter aufgerufen werden und dann ihrerseits die CM-Schicht aufrufen. Genaugenommen sind hier spezielle Funktionen in der WEI-Schicht nur noch wegen der Umsetzung auf die CM-Variablenstrukturen notwendig.Die folgende Funktion `weiChangeIntCallback()` wird zum Beispiel bei Änderungen von Integer-Variablen aufgerufen:

```
void weiChangeIntCallback(dataitemptr di,int value)
{
  cmVarPtr cm=FindVarByAddress(di); /* Umsetzung */

  cmChangeCallback(cm,value);
}
```

Entsprechende Callbacks sind im Interpreter auch für das Starten von Aktivitäten verfügbar, so daß hier analog vorgegangen werden kann.

3.4.3. Implementierung der Änderungsfunktionen

Wie schon bei Statemate sind die Änderungsfunktionen sehr einfach, hier als Beispiel die Funktion zur Änderung einer Integer-Variablen aus der CM-Schicht:

```
void weiSetInteger(cmVarPtr cm,int value)
{
  dataitemptr di=(dataitemptr)cv->address;

  siSetInteger((dataitemptr)cv->address,value);
}
```

Alternativ hätte man hier auch auf die Funktion `siSetIntegerByName()` zurückgreifen können und könnte sich so das Speichern der WEI-Strukturadresse in den CM-Variablenstrukturen sparen. Allerdings sind die namenbasierten Funktionen des Interpreters weniger performant als die adreßbasierten, da sie zunächst intern anhand des Namens die zugehörige Variablenstruktur suchen müssen.

3.4.4. Ausführen eines lokalen Schritts

Da der Interpreter speziell für den Einsatz mit dem Kommunikationsmanager erstellt wurde, ist es natürlich naheliegend, daß er bereits eine besondere Funktion `siStep()` bietet, die genau einen Schritt im Chart ausführt. Insbesondere hält sich diese Funktion auch an die bereits bei Statemate angesprochene Reihenfolge von Update- und Step-Phase, das heißt, sie aktualisiert zunächst die Werte der Variablen und führt erst dann den eigentlichen Schritt im Chart aus. Diese Funktion wird daher als Schrittfunktion von der WEI-Schicht aus aufgerufen.

3.5. Diskussion von Entwurfsalternativen

Das im Kommunikationsmanager verwendete Konzept basiert darauf, Änderungen von Elementen der Spezifikation an alle Partitionen zu propagieren, die dieses Element verwenden. Der Anstoß zur Übertragung der Änderungsnachricht geht also von der Partition aus, in der sich Änderungen ergeben.

Um alternative Möglichkeiten aufzuzeigen, kann man vorliegende Situation mit einer verteilten Datenbank vergleichen, in der die Daten über verschiedene Server verteilt sind, wobei mehrere Kopien eines Datensatzes möglich sind. Die Datensätze sind in diesem Fall die in der Spezifikation vorkommenden Variablen und Aktivitäten, die einzelnen Server sind die Partitionen. Gesucht sind nun Protokolle, die die Konsistenz der Daten im verteilten System erhalten. Insbesondere sollen zu einem Zeitpunkt alle Server den gleichen Wert eines Datensatzes sehen, nämlich den zuletzt geschriebenen. Algorithmen aus dem Bereich der verteilten Datenbanksysteme lassen sich entsprechend auf die Situation der verteilten Workflowausführung übertragen.

Denkbar ist zum Beispiel, Änderungen überhaupt nicht zu propagieren. Statt dessen speichern die Kommunikationsmanager aller Partitionen in ihren CM-Variablenstrukturen zusätzlich zum aktuellen Wert auch als Zeitstempel die Nummer des Schrittes, in dem die Variable zum letzten Mal geändert wurde. Im folgenden Kapitel wird sich zeigen, daß bei einem vollständig synchronisierten Ablauf der Workflowausführung eine solche eindeutige Schrittnummer definiert ist. Wenn eine Partition nun den aktuellen Wert einer Variablen benötigt, fordert sie die aktuellen Werte zusammen mit den Zeitstempeln an. Der Wert der Variablen ist dann der Wert mit dem größten Zeitstempel.

Die hier verwendeten generierten Statecharts erfordern aber ständiges Abfragen des aktuellen Werts von Bedingungen, da alle Transitionen mit Bedingungen versehen sind, die oft nicht lokal sind, sondern mindestens mit einer weiteren Partition geteilt werden. Das in Abbildung 5 in Kapitel 1 gezeigte Beispiel verdeutlicht dies. Gegenüber der im Kommunikationsmanager verwendeten Methode verursachte diese Alternative also wesentlich mehr Kommunikationsaufwand.

Andere Algorithmen aus dem Datenbankbereich lassen sich ähnlich für den Kommunikationsmanager einsetzen. Die gewählte Methode erfüllt bei moderaten Kommunikationskosten die gewünschten Anforderungen, so daß im Rahmen dieser Arbeit keine weiteren Algorithmen untersucht wurden.

4. Synchronisation

In diesem Kapitel werden verschiedene Möglichkeiten zur Synchronisation der verschiedenen verteilt laufenden Partitionen einer Workflowinstanz vorgestellt. Zunächst wird die Verwendung von Synchronisation anhand eines einfachen Beispiels motiviert. Anschließend werden zwei unterschiedlich effiziente Synchronisationskonzepte dargestellt und ihre Implementierung beschrieben. Nach der Diskussion weiterer alternativer Synchronisationskonzepte wird im abschließenden Abschnitt der Algorithmus zur dynamischen Einbindung von Partitionen in die Workflowausführung behandelt.

4.1. Notwendigkeit von Synchronisation

Wie bereits im Eingangskapitel dargelegt wurde, soll die verteilte Ausführung eines Workflows verhaltensäquivalent zur Ausführung der ursprünglichen, nicht partitionierten Spezifikation sein. Dadurch ist es zum Beispiel möglich, Eigenschaften einer Spezifikation durch formale Methoden, die in [Wod96] näher beschrieben sind, zu beweisen, die sich dann unmittelbar auf die verteilte Ausführung der Spezifikation übertragen lassen. Die Korrektheit der Partitionierung von Spezifikationen wurde formal in [Wod96] gezeigt, allerdings nur für die Aufteilung der Ausgangsspezifikation in parallele Komponenten, die dann zentralisiert ausgeführt werden. Der Kommunikationsmanager muß entsprechende Maßnahmen treffen, damit sich diese Komponenten auch dann noch äquivalent zur Ausgangsspezifikation verhalten, wenn sie verteilt ausgeführt werden.

Die in Kapitel 3 vorgestellten Methoden sorgen dafür, daß alle Informationen über Änderungen von Variablen, Zuständen und Aktivitäten innerhalb eines Schrittes einer Partition an alle Partitionen weitergeleitet werden, die diese Informationen benötigen. Bereits das Studium einfacher Beispiele zeigt allerdings, daß dieses allein nicht genügt, um die korrekte Ausführung von Workflows zu gewährleisten. Abbildung 26 zeigt eine solche, sehr einfach gehaltene Situation, in der lediglich zwei Partitionen an der Ausführung beteiligt sind.

Befindet sich zum Beispiel Partition 1 im Zustand `state1_in` und Partition 2 in Zustand `state2_out`, so folgt aus dem Partitionierungsalgorithmus, daß außerdem die Bedingung `in_state1_in` wahr sein muß. Wenn nun die Aktivität `act1` in einem Schritt beendet wird, wird gleichzeitig die Bedingung `act1_done` auf wahr gesetzt. Bei der Ausführung in der Originalspezifikation werden dann im nächsten Schritt `state1_out` und `state2_in` betreten, weil die entsprechenden Transitionen schalten. Bei der verteilten Ausführung kann es vorkommen, daß die Nachricht über die Änderung der Variablen `act1_done` relativ lange unterwegs ist, so daß die Transition in Partition 1 vor der in Partition 2 schaltet und damit eine Zustandskombination erreicht wird, die bei der zentralisierten Ausführung nicht vorkommt. Im schlimmsten Fall kann sogar die Nachricht über die Änderung der Bedingung `in_state1_in`, die durch das Schalten der Transition in Partition 1 ausgelöst wird, gleichzeitig mit der Nachricht über die Änderung der Bedingung `act1_done` von Partition 2 verarbeitet werden, was bewirkt, daß die Transition in Partition 2 nicht schaltet, da `in_state1_in` nicht mehr wahr ist.

Dieses Beispiel verdeutlicht die Problematik der unsynchronisierten Ausführung von verteilten Partitionen. Gleichzeitig zeigt es aber auch, welche Art von Problemen auftreten können: Partition 1 macht einen lokalen Schritt, während Partition 2 untätig bleibt; dies führt zu ungültigen Zustandskombinationen, die bei der Ausführung der Originalspezifikation nicht auftreten. Anschließend führt Partition 2 Änderungen aus zwei vorangehenden Schritten von Partition 1 gleichzeitig aus,

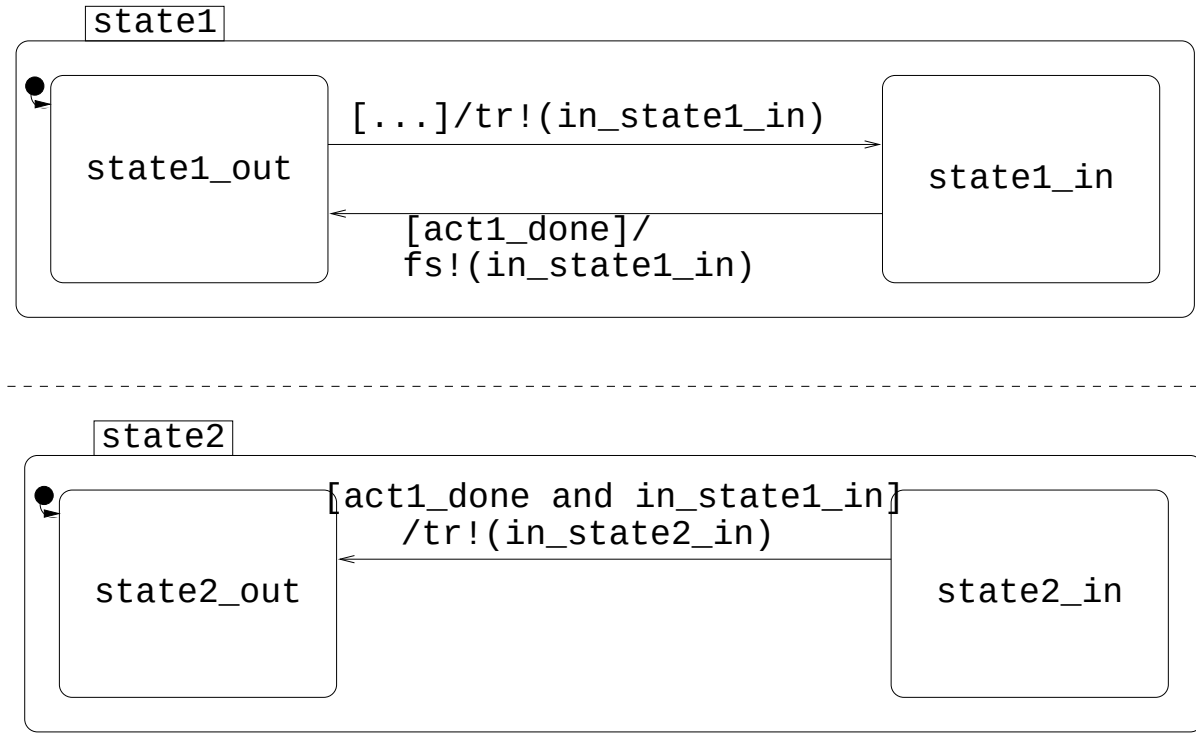

Abbildung 26 - Beispiel für fehlerhafte Ausführung einer Spezifikation ohne Synchronisation

was eine falsche Reaktion der Workflow Engine zur Folge hat. Solche Situationen gilt es also zu vermeiden, was unmittelbar zu dem im folgenden Abschnitt geschilderten Konzept führt.

4.2. Strikte Synchronisation

Die Methode der *strikten Synchronisation*, die in diesem Abschnitt vorgestellt wird, vermeidet die genannten Probleme, indem sie die folgenden beiden Invarianten gewährleistet:

- Alle Partitionen einer Workflowinstanz machen nach dem Abschluß eines lokalen Schritts erst dann einen weiteren, wenn alle übrigen ihren Schritt ebenfalls beendet haben.
- Alle während eines Schrittes gesendeten Nachrichten werden vom Empfänger verarbeitet, bevor ein weiterer Schritt gemacht wird.

Die Einhaltung dieser beiden Invarianten löst genau die oben gezeigten Probleme. Im Anschluß an die Beschreibung des Protokolls wird ein Beweis für die Korrektheit des Protokolls in dem Sinne, daß es die zur zentralisierten Ausführung verhaltensäquivalente Ausführung der Spezifikation gewährleistet

Zunächst zum Ablauf des Protokolls: Jede sendet Partition am Ende ihrer Schreibphase nach dem Senden der Änderungsinformationen an jede andere Partition eine Synchronisationsnachricht. Sie zeigt damit, daß ihr Schritt beendet ist und alle Nachrichten über Änderungen verschickt wurden. Insbesondere erhalten auch solche Partitionen eine Synchronisationsnachricht, denen keine Änderungsnachricht geschickt wird. Am Anfang der Lesephase wird dann, wie schon in Kapitel 3.1 beschrieben, so lange gewartet, bis die Synchronisationsnachrichten aller übrigen Partitionen eingetroffen sind, bevor ein weiterer Schritt in der lokalen Teilspezifikation unternommen wird.

Am Anfang der Ausführung einer Workflowinstanz ist es erforderlich, daß alle Partitionen einmal eine Synchronisationsnachricht verschicken, damit der Algorithmus korrekt abläuft. Sie zeigen damit den übrigen Partitionen, daß sie ihre Initialisierung beendet haben und loslaufen können.

Abbildung 27 zeigt den Ablauf einer Runde dieses Synchronisationsalgorithmus an einem einfachen Beispiel mit nur zwei Partitionen. Am Anfang senden beide Partitionen ihre Synchronisationsnachricht und machen anschließend einen Schritt in ihrer Spezifikation. Ist dieser Schritt abgeschlossen, senden beide Informationen über eventuelle Änderung von Variablenwerten an die andere Partition sowie zum Abschluß eine Synchronisationsnachricht, dann warten sie auf die entsprechenden Nachrichten der anderen Partition.

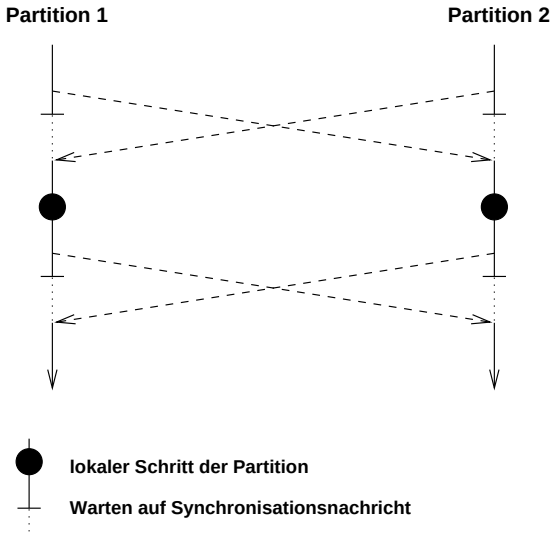

Abbildung 27 - Ein Beispiel für den Ablauf der strikten Synchronisation

Der Algorithmus gewährleistet die beiden geforderten Invarianten:

- Nach der Ausführung eines Schrittes wird der nächste erst begonnen, wenn alle Synchronisationsnachrichten eingegangen sind. Das bedeutet aber gerade, daß alle übrigen Partitionen ebenfalls die Ausführung ihres Schrittes beendet haben.
- Wenn eine Synchronisationsnachricht einer Partition gelesen wurde, wurden auch alle Nachrichten über Änderungen in dieser Partition gelesen und lokal propagiert. Dies gilt, da die Synchronisationsnachricht immer nach allen Änderungsnachrichten gesendet wird und die Datentransportschicht die Reihenfolge der Nachrichten erhält.

Die Korrektheit des Protokolls folgt nun mittels vollständiger Induktion über die Anzahl der Schritte. Am Anfang der Workflowausführung sind die Charts aller Partitionen im gleichen Zustand wie bei der Ausführung der bereits in orthogonale Komponenten aufgeteilten, aber noch nicht verteilten Version. Das bedeutet, bei beiden Ausführungen sind die gleichen Zustände betreten und die gleichen Aktivitäten aktiv, außerdem haben die Variablen die gleichen Werte. Dies zeigt die Induktionsvoraussetzung.

Wurden nun bei der Workflowausführung bereits eine Anzahl Schritte gemacht, sind die Charts aller Partitionen nach Induktionsvoraussetzung im gleichen Zustand wie in der orthogonalisierten Version der Ausgangsspezifikation. Aufbauend auf diesem Zustand berechnen nun alle Partitionen den Zustand ihres Teils der Spezifikation nach dem nächsten Schritt, indem sie einen Schritt in

ihrer Teilspezifikation machen. Wegen Invariante 1 wird dann gewartet, bis alle fertig sind, anschließend wird dieser neue Zustand an alle Partitionen propagiert. Wegen Invariante 2 werden dabei alle Informationen verarbeitet und somit der gesamte neue Zustand angenommen, bevor ein weiterer Schritt ausgeführt wird. Wegen der Semantik von parallelen Komponenten in Statecharts ist dieser neue Zustand der gleiche, der bei der Ausführung eines Schritts im orthogonalisierten Statechart erreicht wird, da die Statecharts der einzelnen Partitionen dort als parallele Komponenten enthalten sind.

Strikte Synchronisation löst also auf sehr einfache Weise das Problem der Synchronisation der verteilten Partitionen einer Workflowinstanz. Der Preis für die Einfachheit ist aber ein hoher Kommunikationsaufwand, da in jedem Schritt eine Nachricht an alle Partitionen geschickt werden muß. Dies gilt sogar dann, wenn sich im gesamten Workflow während eines Schrittes keine Änderungen ergeben haben. Bei der Ausführung von Workflows ist das aber der Normalfall, da in der Regel auf Eingaben von Benutzern gewartet wird. Offenbar lassen sich diese “unnötigen” Synchronisationen sparen, was zum Konzept der *dynamischen Synchronisation* führt, die im folgenden Abschnitt dargestellt wird.

Ein weiterer Nachteil strikter Synchronisation ist, daß die Geschwindigkeit der Ausführung einer Workflowinstanz immer von der langsamsten Partition bestimmt wird. Alle übrigen Partitionen warten, bis auch die langsamste ihren Schritt beendet hat. Solche Probleme können zum Beispiel durch überlastete Rechner, aber auch durch überlastete oder gestörte Kommunikationsverbindungen auftreten. Vollständig unbrauchbar ist dieses Synchronisationskonzept, wenn die Rechner, auf denen Partitionen ausgeführt werden, nicht immer erreichbar sind - etwa der Laptop eines Außendienstmitarbeiters. Für diesen Fall, der hier nicht weiter betrachtet wird, muß man andere Lösungsmöglichkeiten vorsehen.

Die Implementierung strikter Synchronisation wird nicht gesondert vorgestellt, da sie analog zur Implementierung von dynamischer Synchronisation verläuft, die im folgenden gezeigt wird. An den entsprechenden Stellen wird kurz auf die Änderungen verwiesen, die für strikte Synchronisation vorgenommen werden müssen.

4.3. Dynamische Synchronisation

Um den hohen Kommunikationsaufwand der strikten Synchronisation zu verringern, beschränkt sich *dynamische Synchronisation* darauf, alle Partitionen genau dann zu synchronisieren, wenn sich bei mindestens einer Partition bei Ausführung eines Schrittes Änderungen ergeben. Ab diesem Zeitpunkt laufen alle Partitionen solange synchron zueinander, bis ein Schritt keine Änderungen ergeben hat. Danach arbeiten die Partitionen unabhängig voneinander weiter, bis sich erneut in einer Partition eine Änderung ergibt.

An dieser Stelle tritt die Frage auf, warum Partitionen “im Leerlauf”, also in deren Spezifikation sich nichts tut, überhaupt regelmäßig einen Schritt in ihrer Workflow Engine anstoßen müssen. Das liegt daran, daß Benutzeroberflächen, über die Anwender Eingaben in den Workflow machen können, über die Workflow Engine angebunden werden. Statemate bietet dazu eigene Oberflächen, sogenannte *Panels*, an; der Interpreter unterstützt ein Interface zum World Wide Web mittels *Java* ([Hof97]); beide können mit *integrierten Worklist-Oberflächen* ([Bie97]) gekoppelt werden. Diese Oberflächen müssen regelmäßig abgefragt werden. Dazu ist ein Aufruf der Schrittfunktion der Workflow Engine notwendig, da alle Oberflächen unmittelbar in die Workflow Engine und nicht in den Kommunikationsmanager integriert sind. Auf diese Weise ist es möglich, die Oberflächen auch bei der zentralisierten Ausführung eines Workflows zu benutzen.

Abbildung 28 zeigt ein Beispiel für eine dynamische Synchronisation zweier Partitionen, an dem nun die Details dieses Synchronisationskonzeptes dargestellt werden.

Zunächst laufen beide Partitionen unsynchronisiert. In Partition 1 ergeben sich nun Änderungen bei der Ausführung eines lokalen Schrittes, die an Partition 2 propagiert und mit der bereits bekannten Synchronisationsnachricht abgeschlossen werden. Partition 2 liest diese Nachrichten. Bevor nun ein lokaler Schritt in Partition 2 ausgeführt wird, wird eine zusätzliche Synchronisationsnachricht an Partition 1 verschickt. Durch diese Nachricht wird das Gesamtsystem in den gleichen Zustand versetzt, als sei am Ende des vorangehenden Schrittes von Partition 2 strikt synchronisiert worden. Anschließend arbeiten die beiden Partitionen die nächsten Schritte strikt synchronisiert zueinander ab. In Partition 1 ergeben sich im folgenden Schritt keine Änderungen. Im Unterschied zum strikten Protokoll wird nun eine sogenannte “Unsynchronisationsnachricht” verschickt, die bedeutet, daß sich in Partition 1 nichts geändert hat und somit wieder im asynchronen Lauf weitergemacht werden kann. Da sich gleichzeitig in Partition 2 Änderungen ergeben haben, folgt noch ein weiterer synchronisierter Schritt. Nun ändert sich in keiner der beiden Partitionen etwas, beide verschicken die Unsynchronisationsnachricht, und die Workflowausführung geht asynchron weiter.

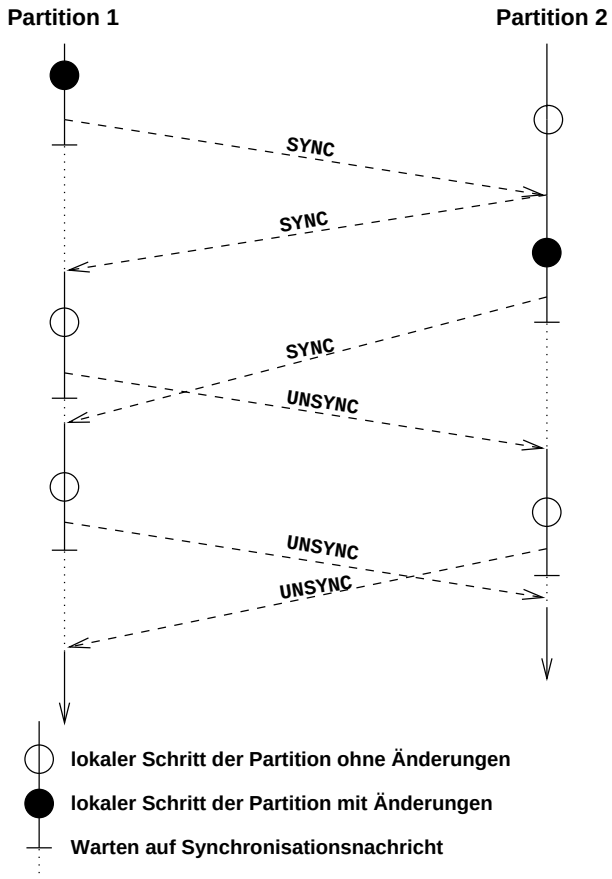

Abbildung 28 - Ablauf der dynamischen Synchronisation

Welche Art von Synchronisationsnachricht schickt eine Partition? Für diese Entscheidungen gibt es vier einfach zu unterscheidende Kriterien:

- Wenn sich bei der Ausführung des lokalen Schritts Änderungen ergeben haben, wird immer eine Synchronisationsnachricht geschickt (Fall *S*).

- Wenn in der Synchronisationsphase Synchronisationsnachrichten empfangen wurden, sich aber keine Änderungen beim Ausführen des lokalen Schritts ergeben haben, wird eine Unsynchronisationsnachricht geschickt (Fall *U1*).
- Wenn die Partition nach dem vorherigen lokalen Schritt eine Synchronisationsnachricht geschickt hat, in der Synchronisationsphase nur Unsynchronisationsnachrichten erhalten hat und sich bei der Ausführung des lokalen Schritts keine Änderungen ergeben haben, wird trotzdem eine Unsynchronisationsnachricht geschickt (Fall *U2*).
- In allen anderen Fällen wird keine Nachricht geschickt und die Ausführung geht asynchron weiter (Fall *N*).

Im Fall U1 ist die Unsynchronisationsnachricht die Antwort auf eine Synchronisationsnachricht einer anderen Partition. Im letzten Schritt dieser Partition gab es also eine Änderung, so daß die Ausführung synchron weitergehen muß. Der Fall U2 ist der symmetrische Fall in der anderen Partition: Den Partitionen, die sich im letzten Schritt gemäß U1 für eine weitere synchrone Ausführung entschieden haben, wird nunmehr die asynchrone Ausführung ermöglicht. Im Fall N gibt es keine Gründe, die gegen eine asynchrone Ausführung sprechen würden, außerdem warten keine anderen Partitionen mehr auf Synchronisationsnachrichten.

Empfängt eine Partition während der asynchronen Abarbeitung eine Synchronisationsnachricht, verschickt sie ihrerseits Synchronisationsnachrichten an alle übrigen Partitionen. Dieser Vorgang heißt *Resynchronisation*. Durch ihn werden die Voraussetzungen geschaffen, daß die Synchronisation der folgenden Schritte wie im strikten Fall erfolgen kann. Zur Verdeutlichung dieses Vorgangs zeigt Abbildung 29 im linken Teil noch einmal den Anfang des in Abbildung 28 gezeigten Ablaufs. Auf der rechten Seite ist der entsprechende Ablauf in strikter Synchronisation zu sehen. Man erkennt sofort, daß die beiden Abläufe identisch weitergehen. Die Korrektheit des Protokolls in dem Sinn, daß die beiden eingangs genannten Invarianten erhalten bleiben, folgt daher unmittelbar aus der Korrektheit der strikten Variante.

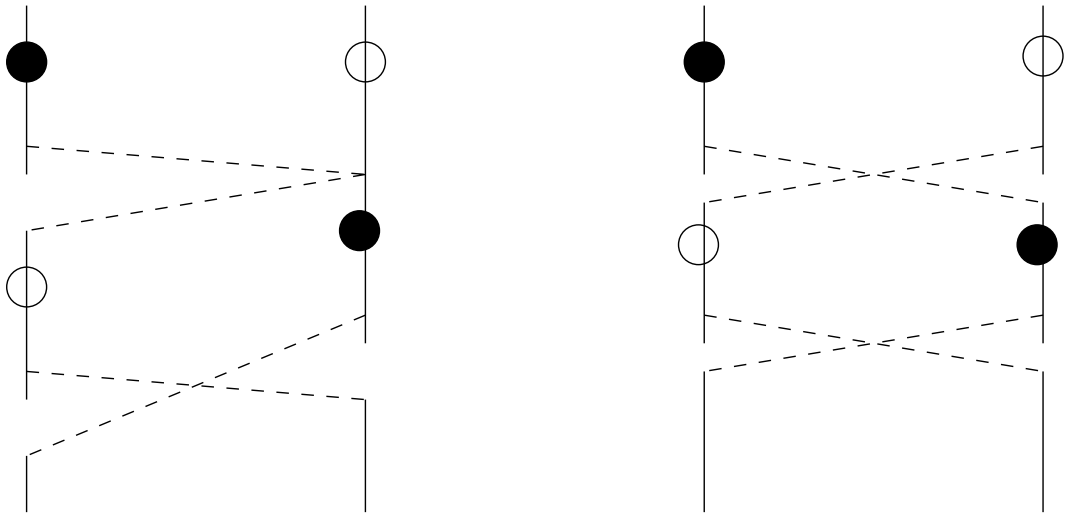

Abbildung 29 - Resynchronisation als Spezialfall strikter Synchronisation

	synchron/ asynchron	zuletzt gesendeter Typ	empfangener Typ	Änderung im lokalen Schritt	gesendeter Typ	synchron/ asynchron
Fall S	synchron	beliebig	beliebig	ja	SYNC	synchron
Fall U1	synchron	beliebig	SYNC	nein	UNSYNC	synchron
Fall U2	synchron	SYNC	UNSYNC	nein	UNSYNC	synchron
Fall N	synchron	UNSYNC	UNSYNC	nein	--	asynchron
-	asynchron	beliebig	SYNC	ja	Resync + SYNC	synchron
-	asynchron	beliebig	SYNC	nein	Resync + UNSYNC	synchron
-	asynchron	beliebig	--	ja	SYNC	synchron

Abbildung 30 - Tabellarische Darstellung der vier Fälle der dynamischen Synchronisation

Zu zeigen bleibt noch, daß das Protokoll terminiert, das heißt, daß die Workflowinstanz irgendwann wieder asynchron ausgeführt wird, wenn sich keine Änderungen mehr ergeben. Abbildung 30 zeigt dazu noch einmal die genannten Fälle in tabellarischer Form. Man sieht zunächst, daß eine Änderung in einer Partition nach Fall S immer eine Synchronisationsnachricht und damit einen weiteren synchronisierten Schritt nach sich zieht, so daß das Protokoll in keinem Fall zu früh abbrechen kann. Sobald sich in einem Schritt nirgends mehr Änderungen ergeben, werden nach Fall U1 oder U2 von allen Partitionen Unsynchronisationsnachrichten verschickt. Im darauf folgenden Schritt schließlich wird die Ausführung nach Fall N wieder asynchron, das heißt, das Protokoll terminiert.

4.4. Implementierung

Für die Einbindung von Synchronisation in den Kommunikationsmanagers, insbesondere in die Hauptschleife der CM-Schicht, sind vor allem Ergänzungen in der Lesephase, in der jetzt auch Synchronisationsnachrichten verarbeitet werden müssen, und in der Schreibphase, wo nun auch solche Nachrichten geschrieben werden müssen, notwendig.

Vor der Beschreibung der notwendigen Funktionen werden zunächst noch einige weitere Ergänzungen und Definitionen betrachtet. Während der Workflowausführung zählt die *SyncSequenceNumber* die Anzahl der bisher durchgeführten synchronisierten Schritte. Synchronisationsnachrichten haben das Format

```
$SYNC_ProcID_SSN,
```

wobei ProcID die ProcID des abschickenden Prozesses und SSN die SyncSequenceNumber des Schrittes ist, in dem die Nachricht geschickt wird. Unsynchronisationsnachrichten haben analog das Format

```
$UNSYNC_ProcID_SSN.
```

```
void cmHandleSyncMessage(char *msg)
{ cfgProcInfoPtr p;
  int procid;
  int ssn;

  /* zunächst werden procid und ssn aus der Nachricht
     extrahiert, das wird hier weggelassen */

  p=cfgFindProc(procid);
  if (p==NULL)
  { /* Fehlerbehandlung weggelassen */ }

  if (strncmp(&msg[1],"SYNC",4)==0) /* es ist SYNC */
  {
    if (cmSyncMode==0) /* bisher asynchron, daher jetzt
                          Resynchronisation */
    {
      dti_abort();
      dti_begin();
      cmSendSyncMsg();
      dti_commit();
      dti_begin();
      cmSyncMode=1; /* synchron weiterlaufen */
      return;
    }
    else
      cmSyncReceived=1;
  }

  /* bei UNSYNC sind keine besonderen Maßnahmen notwendig */

  p->synced=1; /* Prozeß als synchronisiert markieren */

}
```

Abbildung 31 - Die Funktion `cmHandleSyncMessage()`

In der Variablen `cmSyncReceived` wird gespeichert, ob in diesem Schritt SYNC empfangen wurde, in diesem Fall ist die Variable 1, sonst 0. Sie wird zu Beginn der Lesephase initialisiert. `cmSyncSent` ist 1, wenn in diesem Schritt SYNC gesendet wurde, und 0 sonst. Der Wert dieser Variable wird nach dem eventuellen Schreiben der Synchronisationsnachrichten angepaßt. `cmSyncMode` ist 1, wenn der Workflow synchron läuft, und 0 sonst; zu Beginn des Workflows wird diese Variable mit 1 initialisiert, da am Anfang immer synchronisiert wird. `cmSSN` schließlich enthält die aktuelle SyncSequenceNumber.

In der Prozeßstruktur `cfgProcInfo`, die in Anhang B beschrieben ist, zeigt das Flag `synced` an, ob in diesem Schritt eine bereits eine Synchronisationsnachricht - SYNC oder UNSYNC - von diesem Prozeß verarbeitet wurde.

```
void cmSendSyncMsg(void)
{ char cmSyncMsg[200];
  char cmUnsymcMsg[200];
  int flag;
  cfgProcInfoPtr p=proclist;

  sprintf(cmSyncMsg,"$SYNC_%d_%d",ThisProc,++cmSSN);
  sprintf(cmUnsyncMsg,"$UNSYNC_%d_%d',ThisProc,cmSSN);

  if (StepHasChanges) /* Änderungen, also Fall S */
    flag=1;
  else if (cmSyncReceived) /* Sync von außen erhalten, also
                              Fall U1 */
    flag=2;
  else if (cmSyncSent) /* Sync im letzten Schritt geschickt,
                          also Fall U2 */
    flag=2;
  else /* Fall N, keine Synchronisation notwendig */
    flag=0;

  if (flag==0)
  { cmSyncMode=0; /* asynchron weiterlaufen */
    cmSyncSent=0;
    return;
  }
  else if (flag==1)
    cmSyncSent=1;
  else
    cmSyncSent=0;

  while (p) /* an alle Prozesse schicken */
  { if (flag==1)
      dti_sendmsg(cmSyncMsg,0,p->procid);
    else
      dti_sendmsg(cmUnsyncMsg,0,p->procid);
    p=p->next;
  }
```

Abbildung 32 - Die Funktion `cmSendSyncMsg()`

In Kapitel 3.2.3 wurde die Plazierung der notwendigen Funktionen in der Leseschleife teilweise bereits in Abbildung 17 gezeigt, daher wird hier lediglich die Implementierung dieser Funktionen vorgestellt. Zu Beginn der Lesephase müssen die `synced`-Flags aller Prozesse zurückgesetzt werden, dies übernimmt die Funktion `cmResetSyncProcs()`, die in der Lesephase unmittelbar nach dem Beginn der Transaktion eingebaut wird. Die Verarbeitung einer eingehenden Synchronisationsnachricht, die daran erkannt wird, daß das erste Zeichen der Nachricht ein `$` ist, erledigt die Funktion `cmHandleSyncMessage()`, die in Abbildung 31 gezeigt ist. Gleichzeitig leitet sie auch die Resynchronisation ein, wenn eine SYNC-Nachricht empfangen wurde, der Workflow aber bisher asynchron ausgeführt wurde, was der Variablen `cmSyncMode` entnommen werden kann. Dazu wird zunächst die aktuelle Transaktion abgebrochen und eine neue begonnen, in der lediglich die

Synchronisationsnachrichten verschickt werden. Dies hat die gleiche Wirkung, als habe die Partition bereits am Ende der vorangegangenen Schreibphase eine Synchronisation durchgeführt. Anschließend wird diese Transaktion beendet, eine neue begonnen und der Synchronisationsmodus auf synchron geschaltet. Nach dem Rücksprung aus der Funktion `cmHandleSyncMessage()` werden dann die bereits gelesenen Nachrichten erneut gelesen, die Partition verhält sich so, als sei sie gerade erst in die Lesephase eingetreten.

Die Implementierung der Funktion `cmCheckSyncProcs()` ist trivial. Läuft der Workflow asynchron, gibt sie sofort 1 zurück, so daß die Lesephase der CM-Schicht unmittelbar verlassen wird. Sonst wird für jeden Prozeß anhand des Flags in der `cfgProcInfo`-Struktur geprüft, ob bereits eine Synchronisationsnachricht von ihm erhalten wurde. Ist das für mindestens einen nicht der Fall, gibt die Funktion den Wert 0 zurück, so daß die while-Schleife in Abbildung 17 erneut durchlaufen wird.

Die Änderungen in der in Abbildung 21 gezeigten Schreibphase sind einfach: Lediglich vor dem Beenden der Transaktion muß die Funktion `cmSendSyncMsg()` eingebaut werden, die in Abbildung 32 dargestellt ist. Sie sorgt für das Senden der entsprechenden Art von Synchronisationsnachricht nach der Tabelle in Abbildung 30. Außerdem setzt die Funktion die Variablen `cmSyncSent` und `cmSyncMode` entsprechend neu.

Zum Abschluß der Betrachtungen über die Implementierung der Synchronisation wird nun noch gezeigt, wie ohne umfangreiche Änderungen auch strikte Synchronisation implementiert werden kann. Dazu genügt es, zum einen die Variable `cmSyncMode` immer auf 1 zu halten und damit den Workflow synchron ablaufen zu lassen, und zum anderen das Senden von UNSYNC-Nachrichten zu vermeiden. In der Funktion `cmSendSyncMsg()` fällt daher die gesamte Fallunterscheidung zu Beginn der Funktion weg. Statt dessen wird immer eine SYNC-Nachricht an alle Prozesse geschickt. `cmSyncMode` ist so immer 1, da der einzige Ort, wo diese Variable verändert wurde, gerade die Fallunterscheidung in `cmSendSyncMsg()` war, die jetzt wegfällt. Die Workflowausführung geschieht also immer synchron. In der Funktion `cmHandleSyncMsg()` schließlich kommt es nie zu einer Resynchronisation, da `cmSyncMode` immer 1 ist.

4.5. Probleme bei der Synchronisation

Die beiden vorgestellten Protokolle erfüllen in der Theorie genau die in sie gesetzten Erwartungen, das heißt, sie gewährleisten eine korrekte Ausführung des Workflows in dem vorher vorgestellten Sinne. Im praktischen Einsatz zeigen sich aber im wesentlichen zwei Probleme, die im folgenden zusammen mit Lösungsansätzen kurz dargestellt werden.

Das erste Problem sind Nachrichten, die fehlerhaft übertragen werden oder gar verlorengehen. Man sieht sofort, daß eine verlorene Synchronisationsnachricht das gesamte Synchronisationsprotokoll zum Erliegen bringt. Um dieses Problem gar nicht erst entstehen zu lassen, wurde von der Datentransportschicht gefordert, keine Nachrichten zu verlieren. Reale Implementierungen von solchen Datentransportschichten, etwa Tuxedo, erfüllen diese Anforderung nur bedingt, da sie zum Beispiel gelegentlich bei der Übertragung “hängenbleiben” und nur durch “gewaltsames” Beenden von außen unter Verlust der übertragenen Daten wieder zum Leben zu erwecken sind. Dieser Fall ist natürlich kein Teil ihrer Spezifikation, sondern auf interne Fehler zurückzuführen. Die Behandlung solcher Fehlerfälle wurde im Kommunikationsmanager vollständig in die Datentransport-Interface-Schicht verlagert, da dort wesentlich spezifischere Maßnahmen getroffen werden können als in der allgemein gehaltenen CM-Schicht. Die Praxis zum Beispiel mit Tuxedo zeigt, daß solche Fehler bei korrekter Konfiguration nur sehr selten auftreten.

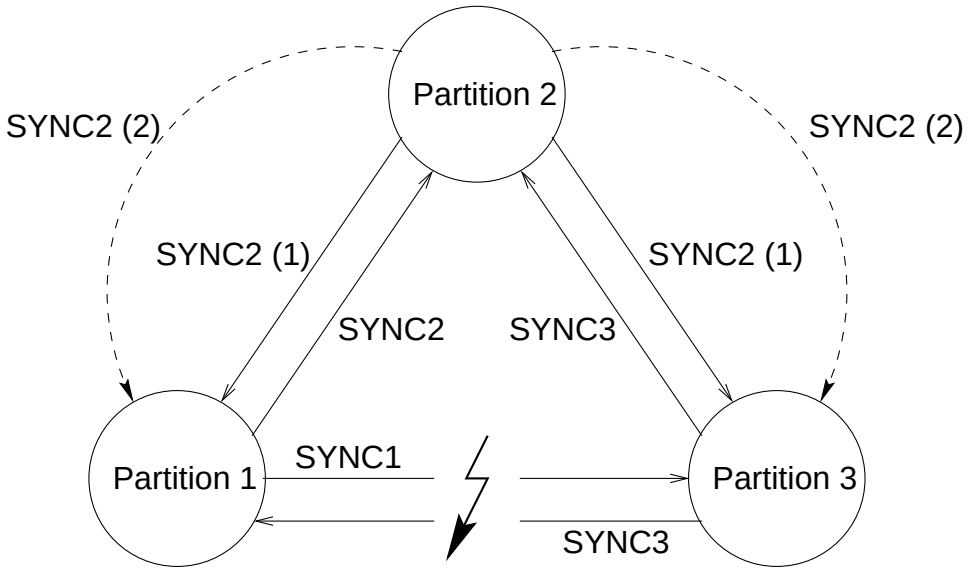

Abbildung 33 - Synchronisationsnachrichten des falschen Schritts

Das zweite Problem, das hier behandelt wird, tritt wesentlich häufiger auf, ist aber auch einfacher zu beheben. Bei ungleichmäßiger Verteilung schneller und langsamer Kommunikationswege zwischen den Rechnern, die an der Workflowausführung beteiligt sind, kann es vorkommen, daß insbesondere Synchronisationsnachrichten aus verschiedenen Schritten gemischt werden. Das gleiche Problem tritt bei zeitweiligen Störungen der Kommunikation zwischen verschiedenen Partitionen auf. Abbildung 33 zeigt ein einfaches Beispiel mit drei Partitionen. Zunächst schicken alle Partitionen in Schritt 1 an alle übrigen jeweils eine Synchronisationsnachricht. Die Kommunikation zwischen Partition 1 und Partition 3 ist dabei zeitweise gestört, so daß die beiden Nachrichten verzögert werden. Die beiden Partitionen erhalten zwar die Synchronisationsnachricht von Partition 2, warten aber auf die von Partition 3 bzw. Partition 1. Partition 2 dagegen hat beide erwarteten Synchronisationsnachrichten erhalten, führt einen lokalen Schritt aus und schickt Synchronisationsnachrichten des Schritts 2 an die beiden übrigen. Partition 1 zum Beispiel hat dann schon eine Nachricht aus Schritt 2 erhalten, obwohl sie noch auf eine Nachricht aus Schritt 1 wartet.

Die “überzählige” Nachricht kann nun nicht einfach verarbeitet werden, da sie sonst im nächsten Schritt fehlen würde. Darüber hinaus könnte es sein, daß zusammen mit dieser Synchronisationsnachricht Informationen über Änderungen in Partition 2 im zweiten Schritt eingegangen sind. Diese dürfen natürlich in keinem Fall lokal propagiert werden, da Partition 1 diesen Schritt selbst noch nicht ausgeführt hat und somit die Korrektheit nicht mehr gewährleistet wäre.

Die Lösung für dieses Problem, die auch in der CM-Schicht implementiert ist, besteht darin, solche “verfrühten” Nachrichten bis zum Ende der Lesephase zwischenzuspeichern, aber nicht auszuwerten. Aus diesem Grund wird auch mit den Nachrichten über Änderungen die SyncSequenceNumber des Schrittes verschickt, in dem die Änderung eingetreten ist. In den Synchronisationsnachrichten ist diese Information ohnehin enthalten. Am Ende der Lesephase, vor dem Beenden der Transaktion, werden diese Nachrichten dann zurückgeschrieben, indem sie mittels `dti_sendmsg()` an den eigenen Prozeß geschickt werden. In der nächsten Lesephase können sie dann wieder wie üblich als Nachrichten empfangen werden. Durch diese Vorgehensweise sind auch bei einem Absturz die Nachrichten, die den folgenden Schritt betreffen, nicht verloren; für die Restaurierung der Nachrichten des aktuellen Schritts ist der Logmanager zuständig.

4.6. Weitere Alternativen

In diesem Abschnitt werden zunächst zwei Optimierungsmöglichkeiten für die dynamische Synchronisation aufgezeigt. Anschließend wird mit dem Konzept der *Synchronisationspunkte* ein Ansatz vorgestellt, der sich von der schrittweisen Synchronisation der bisherigen Protokolle unterscheidet.

4.6.1. Implizite Synchronisationsnachrichten

Schaut man sich den Ablauf der Schreibphase genauer an, erkennt man, daß die Synchronisationsnachricht an eine Partition immer in einer Transaktion nach den Nachrichten über Änderungen geschickt wird. Das bedeutet insbesondere, daß alle diese Nachrichten gleichzeitig beim Empfänger sichtbar werden. In der Lesephase werden immer alle anstehenden Nachrichten gelesen. Wenn Änderungsnachrichten gelesen werden, wird also immer auch die nachfolgende Synchronisationsnachricht gelesen. Hier kann die Synchronisationsnachricht wegoptimiert werden, allein eine gelesene Änderungsnachricht genügt wegen der Transaktionseigenschaft, um die sendende Partition als synchronisiert zu markieren. Die Änderungsnachricht enthält also implizit die Synchronisationsnachricht.

Wenn an eine Partition keine Änderungsnachrichten geschickt werden, muß immer eine Synchronisationsnachricht geschickt werden. UNSYNCs können also insbesondere nicht implizit dargestellt werden, da beim Senden eines UNSYNC ja gerade keine lokalen Änderungen entstanden sind.

Es ist offensichtlich, daß dieses Konzept nicht funktioniert, wenn die Datentransportschicht keine Transaktionen unterstützt.

4.6.2. Verbesserung dynamischer Synchronisation

Eine weitere Möglichkeit, die Zahl der verschickten Nachrichten zu verringern, ist es, nicht alle, sondern nur die unbedingt notwendigen Partitionen zu synchronisieren. Hat ein Workflow zum Beispiel zehn Partitionen, von denen lediglich zwei Daten austauschen, ändert sich offensichtlich nichts an der Ausführung des Workflows, wenn nur die beiden synchron, die übrigen dagegen asynchron laufen. Sobald eine Änderung jedoch an eine der asynchronen Partitionen propagiert wird, muß auch diese in den synchronen Lauf eintreten.

Diese Optimierung ist insbesondere deshalb möglich, weil Änderungen von Variablen nur an die Partitionen propagiert werden, in denen die Variablen genutzt werden. Wenn Änderungen immer an alle geschickt würden, wäre es unmöglich, die Partitionen herauszufinden, die synchronisiert werden müssen.

4.6.3. Synchronisation an Synchronisationspunkten

Während dynamische Synchronisation versucht, die Zahl der für die korrekte Ausführung notwendigen Synchronisationsnachrichten durch geeignete Protokolle möglichst gering zu halten, verläßt sich die nachfolgend vorgestellte Methode weitgehend auf einen menschlichen Spezifikator. Dieser bestimmt besondere Zustände in der Spezifikation, sogenannte *Synchronisationspunkte*, bei deren Erreichen sich die Partitionen synchronisieren. Zwischen diesen Punkten laufen alle Partitionen

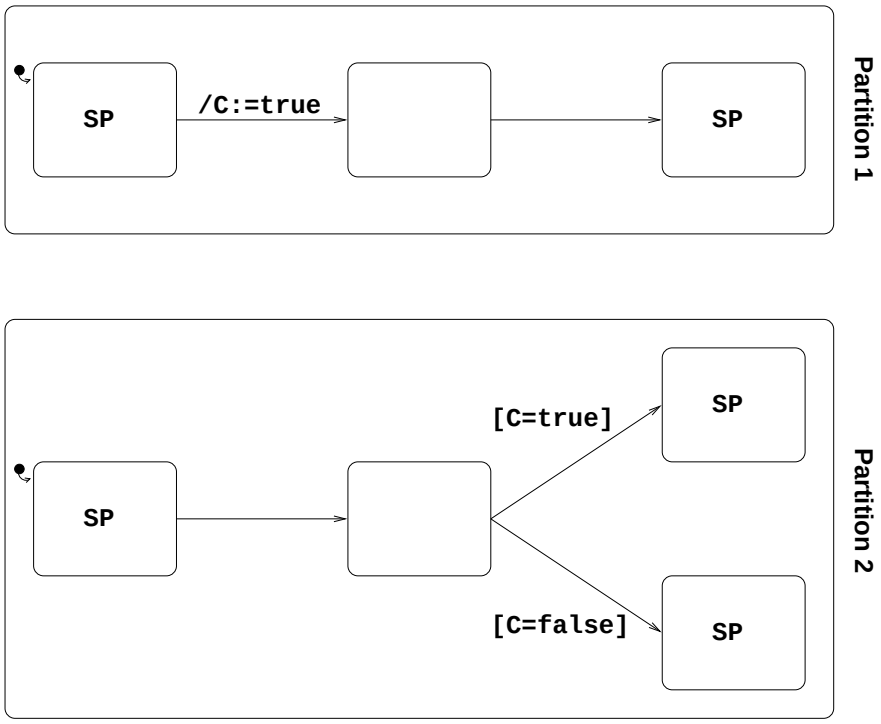

Abbildung 34 - Ungeeignete Wahl von Synchronisationspunkten

asynchron, erst im Synchronisationspunkt werden Informationen über Änderungen ausgetauscht, die sich zwischenzeitlich ergeben haben.

Strikte Synchronisation ist ein Spezialfall dieses Konzepts. Dazu muß jeder Zustand als Synchronisationspunkt ausgezeichnet werden.

Wie das in Abbildung 34 gezeigte Beispiel zeigt, führt nicht jede mögliche Wahl von Synchronisationspunkten zu einer korrekten Ausführung des Workflows im vorher eingeführten Sinne. In diesem (konstruierten) Beispiel wurden die Synchronisationspunkte maximal weit voneinander entfernt gewählt. Dadurch wird die Änderung der Bedingung `C` von Partition 1 erst am Ende der Ausführung an Partition 2 propagiert. Partition 2 hat aber zwischenzeitlich schon aufgrund des falschen Werts von C einen falschen Endzustand betreten. Für eine korrekte Ausführung wäre es notwendig gewesen, in beiden Partitionen auch einen Zustand früher auszuzeichnen - dann hat man aber jeden Zustand markiert, das heißt, der Workflow wird strikt synchronisiert ausgeführt.

Die geeignete Wahl von Synchronisationspunkten ist selbst bei solch einfachen Beispielen nichttrivial, wenn auch wegen der Überschaubarkeit des Beispiels zu lösen. Wesentlich aufwendiger wird die Bestimmung bei großen Workflowspezifikationen, etwa dem in Kapitel 1 gezeigten Lehrbericht. Ohne unterstützende Algorithmen ist es hoffnungslos, von Hand korrekte Synchronisationspunkte zu finden, die nicht zu strikter Synchronisation führen. In dieser Arbeit wurden Synchronisationspunkte daher nicht weiter betrachtet.

4.7. Das Starten von nicht aktiven Partitionen und das Beenden aktiver Partitionen

Wie bereits in Kapitel 3 beschrieben, wird zum Starten von Partitionen, die bisher nicht aktiv sind, ein besonderer Algorithmus benötigt, der in diesem Abschnitt vorgestellt wird. In den nachfolgenden Abschnitten werden einige wichtige Aspekte der Implementierung im Hinblick auf die Änderungen in der CM-Schicht und den Aufbau des *Dynamischen Prozeßmanagers* betrachtet. Im letzten Abschnitt werden zwei Optimierungsmöglichkeiten der vorgestellten Konzepte diskutiert.

4.7.1. Der Algorithmus zur Einbindung von Partitionen

Abbildung 35, 35 zeigt den Ablauf der Einbindung einer neuen Partition anhand eines einfachen Beispiels. Gezeigt ist eine Partition A, der Dynamische Prozeßmanager M und die zu startende Partition B. Der Algorithmus läßt sich einfach auf mehrere aktive Partitionen und mehrere zu startende Partitionen erweitern.

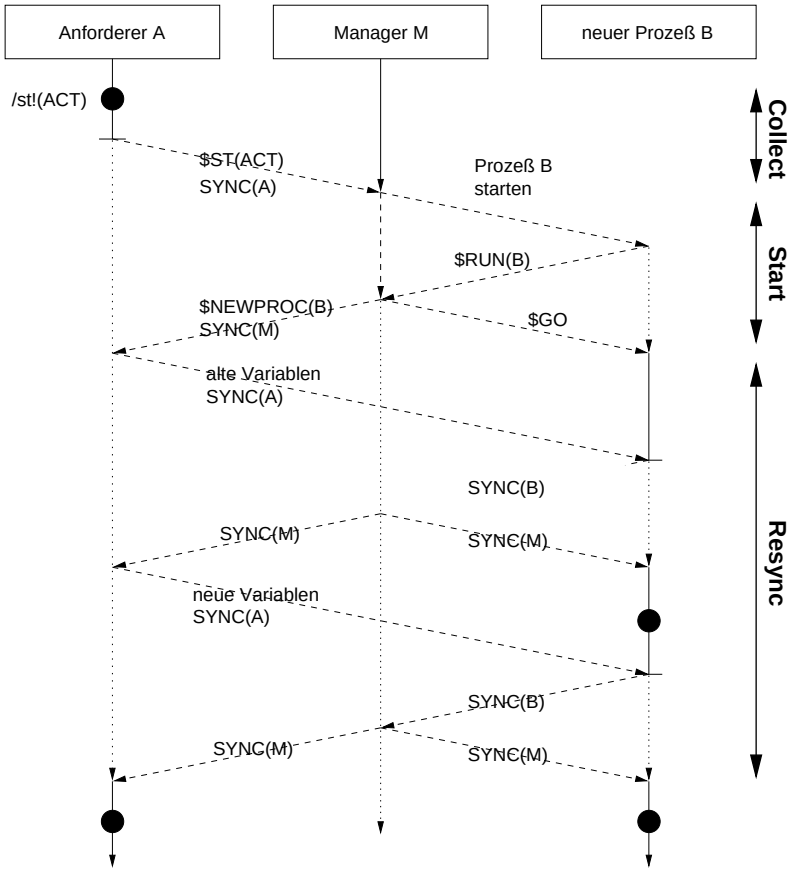

Abbildung 35 - Der dynamische Einbindungsalgorithmus an einem Beispiel

Der Dynamische Prozeßmanager M hat die Aufgabe, das Starten und Beenden von Partitionen einer Workflowinstanz zu koordinieren. Er ist einer der in Abschnitt 2.3 erwähnten Systemprozesse und nimmt aktiv an der Synchronisation der Partitionen teil. Insbesondere bedeutet das, daß die Synchronisation eines Schrittes erst dann abgeschlossen ist, wenn auch M seine Synchronisationsnachricht geschickt hat.

Im Beispiel in der Abbildung trifft Partition A bei der Ausführung eines Schritts in der Spezifikation auf die Anweisung, eine Aktivität zu starten, die in einer noch nicht gestarteten Partition liegt.

In der folgenden Schreibphase sendet sie eine entsprechende Nachricht an den Manager M mit der Bedeutung, die entsprechende Partition zu starten; anschließend verschickt A wie üblich Synchronisationsnachrichten. Nun wartet A auf den Abschluß der Synchronisation. M sammelt alle derartigen Nachrichten eines Schrittes, deshalb heißt diese Phase des Algorithmus *Collect-Phase*. Nachdem alle Partitionen ihre Synchronisationsnachricht geschickt haben, bestimmt M nun aus der in Kapitel 2.3 beschriebenen Konfigurationsdatei des Workflows einen Sachbearbeiter, der die Rolle einnehmen kann, zu der die zu startende Aktivität gehört. Zu jeder möglichen Sachbearbeiter-Rollen-Kombination gibt es einen zugehörigen Teilprozeß; M wählt aus den Teilprozessen, die einer Rolle zugeordnet sind, einen möglichen aus. Im allgemeinen gibt es dazu mehrere Möglichkeiten. M kann zum Beispiel den Teilprozeß wählen, dessen zugeordneter Sachbearbeiter am wenigsten ausgelastet ist, wozu er natürlich über entsprechende Informationen verfügen muß. In diesem Fall übernimmt M die Rolle eines *Worklist Managers*, der die anfallende Arbeit möglichst günstig auf die möglichen Sachbearbeiter verteilt. In dieser Arbeit wird auf diesen Aspekt nicht weiter eingegangen, sondern statt dessen einer der möglichen Prozesse zufällig ausgewählt. Der Worklist-Manager von mentor wird in [Hof97] beschrieben, seine Einbindung in den Kommunikationsmanager in [Weh97].

Nachdem der Prozeß ausgewählt wurde, wird das zugehörige Programm gestartet. Der neue Prozeß B meldet sich nach dem Starten sofort mit einer `$RUN`-Nachricht beim Manager, um anzuzeigen, daß der Start erfolgreich war. Diese Phase des Algorithmus, die *Start-Phase*, ist abgeschlossen, sobald alle gestarteten Prozesse sich in dieser Art gemeldet haben.

Der Manager erlaubt B nun durch eine `$GO`-Nachricht, die Ausführung fortzusetzen. Gleichzeitig teilt er allen anderen Partitionen - in diesem Fall nur A - mit, daß es nun einen neuen Prozeß B im Workflow gibt, und schließt seine Nachricht mit einer Synchronisationsnachricht ab, sobald er alle neuen Prozesse gemeldet hat, die in diesem Schritt eingebunden werden. Die Partitionen binden B nun in ihre Kommunikation ein und schicken ihm die Werte der Variablen vor dem letzten ausgeführten Schritt. Der Grund für diese Maßnahme wird im Anschluß erläutert. Als nächstes schalten die Partitionen in einen besonderen Modus, in dem keine lokalen Schritte ausgeführt werden, sondern nur Nachrichten ausgewertet und die Synchronisation durchgeführt werden.

B hat in der Zwischenzeit seine Ausführung fortgesetzt, hat seine erste Synchronisationsnachricht geschrieben und wertet nun die Variablenwerte aus, die er von A erhalten hat. M wiederum betrachtet die Synchronisationsnachricht von B als Abschluß einer Synchronisationsrunde und schickt seinerseits Synchronisationsnachrichten an alle Partitionen. B führt daraufhin einen ersten Schritt mit den “alten” Variablenwerten von A aus und schickt anschließend eine Synchronisationsnachricht. A schickt seinerseits die aktuellen Variablenwerte an B und an alle Partitionen eine Synchronisationsnachricht; anschließend schaltet A wieder in den normalen Modus. Als letzter Prozeß schickt nun auch M seine Synchronisationsnachricht an die übrigen, worauf A und B ihre Ausführung fortsetzen.

Es bleibt nun noch zu klären, warum A zunächst die alten Variablenwerte und erst im folgenden Schritt die aktuellen Werte senden muß. Abbildung 36 zeigt ein typisches Beispiel, das diese Vorgehensweise begründet. Das obere Zustandspaar ist eine parallele Komponente in Partition A, das untere eine in der neu zu startenden Partition. Der Start der Aktivität wird erkannt, sobald die Transition von `ACT1_S_IN` nach `ACT1_S_OUT` schaltet. Gleichzeitig wird die Bedingung `IN_ACT1_S` auf falsch gesetzt. Würden nun nur die aktuellen Werte übertragen, könnte die untere Transition nie schalten, da `IN_ACT1_S` ja jetzt falsch ist, das würde aber unmittelbar zu einer Ausführung führen,

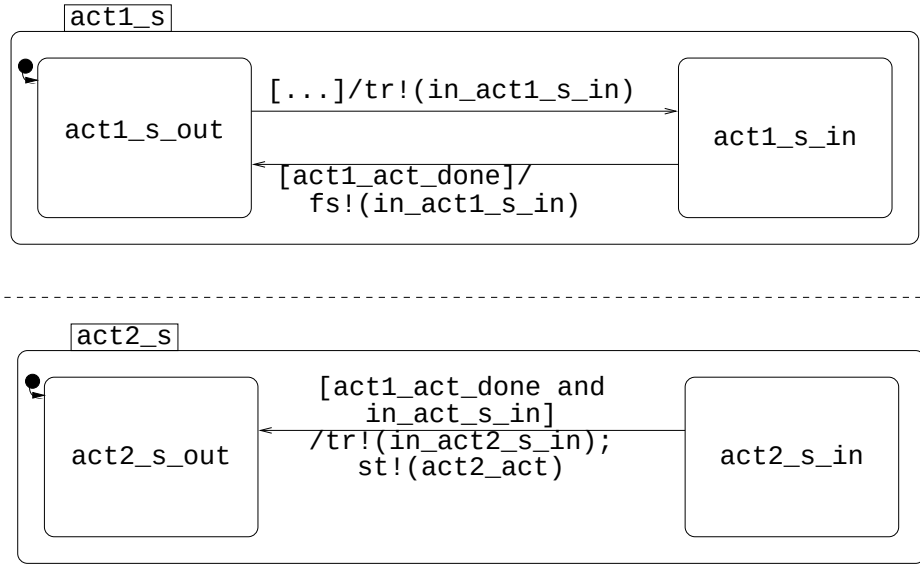

Abbildung 36 - Notwendigkeit des Sendens aktueller Variablenwerte

die nicht verhaltensäquivalent zur zentralisierten Ausführung ist und damit vermieden werden muß. Werden zunächst die alten Werte übertragen, schaltet die untere Transition. Vor dem folgenden Schritt werden dann die aktuellen Werte propagiert. Es ist dabei wichtig, daß A während dieser Zeit selbst keinen lokalen Schritt ausführt, so daß die Variablenwerte erhalten bleiben; deshalb schaltet A in den besonderen Modus.

4.7.2. Der Algorithmus zum Beenden von Partitionen

Es wurde bereits gezeigt, daß eine Partition selbst entscheidet, ob sie beendet werden soll. Die dabei verwendeten Kriterien wurden in Kapitel 3.1.8 dargestellt. Der Algorithmus, der verwendet wird, um eine Partition, die sich beenden will, aus der Workflowausführung zu entfernen, im folgenden Stop-Algorithmus genannt, wird in Abbildung 37 anhand eines Beispiels gezeigt. Er ist analog zum Algorithmus zum Einbinden einer Partition.

Partition A, die sich beenden will, erkennt dies am Ende ihrer Ausführungsphase und schickt daraufhin dem Manager M eine entsprechende Nachricht, außerdem wie üblich eine Synchronisationsnachricht an alle Partitionen, die hier aus Gründen der Übersichtlichkeit nicht dargestellt ist. M sammelt auch hier in der *Collect-Phase* alle derartigen Nachrichten, bis alle Partitionen ihre Synchronisationsnachricht geschickt haben. In der *Stop-Phase* wird anschließend zum einen Partition B mitgeteilt, daß Partition A nicht mehr aktiv ist, und zum anderen Partition A mittels einer `STOP`-Nachricht erlaubt, sich zu beenden. Partition B baut daraufhin die Kommunikationsverbindung zu A ab und löscht die zu A gehörenden Konfigurationsstrukturen. Nun arbeitet B weiter.

Falls gleichzeitig Start- und Stopalgorithmus arbeiten, fallen die Collectphasen der beiden Algorithmen zusammen. Der erste Teil der Resync-Phase wird gleichzeitig mit der Stop-Phase ausgeführt. Die Partitionen, die nicht beendet wurden, nehmen nach Abschluß der Stop-Phase aber nicht ihren regulären Ablauf auf, sondern beenden die Resynchronisationsphase.

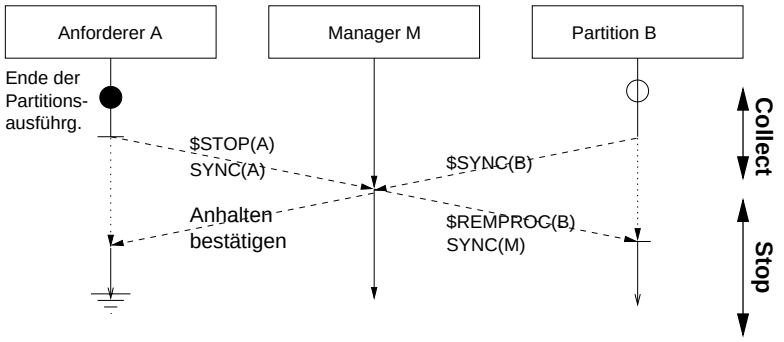

Abbildung 37 - Der Algorithmus zum Beenden von Partitionen

4.7.3. Implementierungsaspekte

In diesem Abschnitt werden einige wichtige Aspekte der Implementierung der beiden vorgestellten Algorithmen dargestellt. Zunächst werden die besonderen Maßnahmen beim Starten einer dyna misch gestarteten Partition aufgezeigt, anschließend die notwendigen Ergänzungen in der CM-Schicht behandelt. Abschließend wird kurz auf die Implementierung des Dynamischen Prozeßmanagers eingegangen.

4.7.3.1. Startmaßnahmen bei dynamisch gestarteten Partitionen

Der beim Start einer dynamisch eingebundenen Partition ausgeführte Code muß neben den in Kapitel drei geschilderten Maßnahmen beim Start einer Partition auch den vorgestellten Algorithmus implementieren. Man muß dabei auch den Fall eines Fehlers während des Starts berücksichtigen, der eine Recovery notwendig macht. Es darf zum Beispiel nicht passieren, daß zunächst die `RUN`-Nachricht an den Managerprozeß geschickt wird, anschließend die Partition wegen eines Fehlers abstürzt und nach der Recovery erneut eine `$RUN`-Nachricht schickt.

Aus diesem Grund wird während des Starts in der Datenbank in dem zum gestarteten Prozeß gehörenden Tupel in der Relation `WFC_Processes` festgehalten, wie weit die Startphase bereits fortgeschritten ist. Diese Relation wird in Abschnitt 2.3.3 beschrieben. Der Dynamische Prozeßmanager trägt dort, bevor er den Prozeß instantiiert, als Status den Wert 2 (“started”) ein. Während der Initialisierung des neuen Prozesses wird dieser Wert abgefragt. Ist er noch 2, wird die `$RUN`-Nachricht an den Manager geschickt und noch in der gleichen Transaktion der Status in der Datenbank auf 3 (“run sent”) geändert. Anschließend wird eine neue Transaktion begonnen, auf die `$GO`-Nachricht des Managers gewartet, der Status auf 1 (“running”) gesetzt und nach dem Beenden der Transaktion in die CM-Hauptschleife eingetreten. Wird dagegen beim Start der Statuswert 3 vorgefunden, wird unmittelbar auf die `$GO`-Nachricht gewartet - die kann noch nicht eingegangen sein, sonst wäre wegen des Zusammenschlusses zu einer Transaktion der Wert des Status schon auf 1 gesetzt worden. Entsprechend wird bei einem Statuswert von 1 sofort in die CM-Hauptschleife eingetreten, wo dann der Logmanager mit der Recovery beginnt.

4.7.3.2. Ergänzungen in der CM-Schicht

Um die gezeigten Algorithmen zu implementieren, sind einige wesentliche Änderungen in der CM-Schicht notwendig, die hier kurz dargestellt werden.

Zunächst muß der besondere Modus herbeigeführt werden können, in den eine Partition während der Resync-Phase geschaltet wird. Dazu wird die Variable `cmPhase` eingeführt. Ist diese Variable

0, so arbeitet die CM-Schicht wie bisher auch. Sobald eine neue Partition hinzugenommen werden soll, dies durch eine entsprechende Nachricht vom Dynamischen Prozeßmanager bekanntgegeben wird und die “alten” Werte der Variablen verschickt wurden, wird diese Variable auf 1 gesetzt. Dadurch wird vermieden, daß ein lokaler Schritt ausgeführt wird. Wenn die nun folgende Synchronisationsrunde abgeschlossen ist, wird die Variable auf 2 gesetzt. So erreicht man, daß die aktuellen Variablenwerte verschickt und die Variable wieder auf 0 gesetzt werden, so daß nach der anschließende Synchronisationsphase wieder die übliche CM-Funktionalität einsetzt. Hier noch einmal schematisch der neue Aufbau der CM-Hauptschleife als Pseudocode:

```
while (1)
{ LESEN_UND_SYNCHRONISIEREN();
  if (neue_prozesse)
  { cmPhase=1;
  }
  if (cmPhase==0)
  { weiStep();
    Änderungen_versenden();
  }
  else if (cmPhase==1)
  { Neue_Prozesse_einbinden_und_alte_Vars_schicken();
    cmPhase=2;
  }
  else if (cmPhase==2)
  { neue_Vars_schicken();
    cmPhase=0;
  }
  SendSyncMsg();
}
```

Die Auswertung der Nachrichten des Prozeßmanagers übernimmt die Funktion `cmHandleCtrlMessage()`, die bereits in Kapitel 3.2 kurz angesprochen wurde. Ihre Hauptaufgabe ist es, anhand der `NEWPROC`-Nachrichten eine Liste aller neu gestarteten Prozesse anzulegen und `cmPhase` auf 1 zu setzen, wenn eine solche Nachricht eingegangen ist. An die Prozesse in dieser Liste werden dann die Variablenwerte verschickt. Die Einbindung der neuen Prozesse in die Kommunikation schließlich wird mittels der DTI-Funktion `dti_addproc()` durchgeführt.

Der Stop-Algorithmus fügt sich nahtlos in dieses Gerüst ein. Hier wird ebenfalls durch die Funktion `cmHandleCtrlMessage()` eine Liste aller zu stoppenden Prozesse angelegt. Anschließend wird in Phase 1 durch die Funktion `dti_remproc()` die Kommunikationsverbindungen zu diesen Prozessen abgebaut, außerdem werden die Konfigurationsstrukturen dieser Prozesse gelöscht. Wenn gleichzeitig keine weiteren Prozesse gestartet wurde, wird nach Phase 1 unmittelbar mit Phase 0 fortgefahren.

Zum Abschluß dieses Abschnittes wird nun noch gezeigt, wie die `START`- und `STOP`-Nachrichten an den Managerprozeß verschickt werden. Die `START`-Nachricht wird in der Schreibphase generiert, wenn festgestellt wird, daß eine Aktivität gestartet wurde, die in einer nicht aktiven Partition liegt. Wie dieses festgestellt wird, ist in Abschnitt 3.2.5 beschrieben. Die `STOP`-Nachricht schließlich wird von einer Partition generiert, wenn sie am Ende ihrer Ausführungsphase nach den in Abschnitt 3.1.8 beschriebenen Kriterien feststellt, daß sie beendet werden kann. Die Partition wartet nach dem Absenden dieser Nachricht noch auf die entsprechenden Bestätigung durch den Prozeßmanager und beendet sich anschließend.

4.7.3.3. Implementierung des Dynamischen Prozeßmanagers

Der Dynamische Prozeßmanager (DPM) durchläuft eine ähnliche Hauptschleife wie die CM-Schicht in den Workflowpartitionen, die aus Lese- und Ausführungsphase besteht. Die Details der Implementierung, insbesondere der Aufbau der verschiedenen Lese- und Auswertfunktionen sowie die Behandlung der Synchronisationsnachrichten, sind analog zu den entsprechenden Funktionen der CM-Schicht und werden daher weggelassen.

Die Lesephase hat exakt den gleichen Aufbau wie in der CM-Schicht. Hier nimmt der DPM Anforderungen anderer Partitionen entgegen, entweder neue Partitionen zu starten oder bereits laufende zu beenden. Außerdem erwartet er die Synchronisationsnachrichten der Partitionen. Es ist wichtig, daß der DPM hier die gleiche Synchronisationsmethode verwendet wie die übrigen Partitionen, damit die Synchronisation nicht außer Tritt kommt. Der DPM merkt sich die eingegangenen Anforderungen in zwei Listen: Die Liste `dpmStartMsgs` enthält die Startanfragen, `dpmStopMsgs` entsprechend die Stopanfragen.

In der Ausführungsphase implementiert der DPM die beiden oben gezeigten Algorithmen. Hat er Startanfragen erhalten, wählt er für jede zu startende Partition anhand der Konfigurationsdatei einen Prozeß aus und hält diese Auswahl wie oben gezeigt in der Relation WFC_Processes in der Datenbank fest. Anschließend instantiiert er alle gewählten Prozesse, indem er die Funktion `dti_startproc()` aus der DTI-Schicht für jeden Prozeß aufruft. Dieser Aufruf ist hier notwendig, da je nach verwendeter DT-Schicht unterschiedliche Mechanismen zum Starten einer Partition notwendig sind, im Falle von Tuxedo etwa muß dazu ein Tuxedo-Serverprozeß aufgerufen werden ([Weh97]).

Nun wartet der DPM in einer der Lesephase vergleichbaren Schleife auf die Startbestätigungen aller Partitionen. Hat er diese erhalten, erteilt er zum einen den Partitionen durch eine START-Nachricht die Starterlaubnis und schickt zum anderen die nach dem Algorithmus geforderten Nachrichten an die übrigen Partitionen. Falls STOP-Anfragen zu bearbeiten sind, werden diese ebenfalls hier durch Versand der entsprechenden Nachrichten erledigt. Anschließend wird der gezeigte Algorithmus zu Ende ausgeführt.

Bei der Implementierung des DPM wurden keine Aspekte der Fehlertoleranz berücksichtigt. Insbesondere wurden keine Mechanismen vorgesehen, durch geeignete Recoverymaßnahmen nach einem Ausfall des DPM während des Startens einer neuen Partition diesen Vorgang nach dem Neustart des DPM fortzusetzen. Im Sinne eines fehlertoleranten Gesamtsystems sollte diese Funktionalität natürlich noch eingebaut werden.

4.7.4. Optimierungsmöglichkeiten

Im bisherigen Konzept gibt es für jede Instanz eines Workflows einen eigenen Dynamischen Prozeßmanager, der sich in jedem synchronisierten Ausführungsschritt mitsynchronisiert. Hier stellen sich zwei mögliche Optimierungsansätze:

- Zum einen kann man einen Prozeßmanager für mehrere Instanzen vorsehen, unter Umständen sogar für einen ganzen Workflowtyp. Dieser Ansatz erfordert einen erhöhten Verwaltungsaufwand im DPM, da dieser nun mehrere Einbindungsalgorithmen gleichzeitig durchlaufen muß. Außerdem steigt die Belastung dieses einzigen Managers natürlich wesentlich an, wodurch die Leistung beeinträchtigt wird. Im Falle des Ausfalls eines solchen Managers schließlich kann nicht nur ein einziger Workflow bis zum Neustart nicht weiter ausgeführt werden, sondern viele Workflows.

- Eine andere Möglichkeit ist es, die Funktionalität des DPM in die Partitionen selbst zu integrieren. Wenn dies gelingt, können insbesondere viele Nachrichten gespart werden, da ein Teilnehmer weniger synchronisiert werden muß. Dabei muß eine Lösung für das eingangs geschilderte Problem gefunden werden, daß mehrere Partition gleichzeitig dieselbe neue Partition instantiieren wollen.

5. Evaluation

Nachdem in den vorangehenden Kapiteln Konzeption und Implementierung wesentlicher Elemente des Kommunikationsmanagers vorgestellt wurden, werden in diesem Kapitel quantitative Betrachtungen angestellt. Eine vollständige Betrachtung aller möglichen Aspekte in diesem Bereich wäre dabei zu umfangreich und würde den Rahmen dieser Arbeit sprengen. Statt dessen werden hier einige ausgewählte Punkte herausgegriffen:

- Die beiden Workflow Engines Statemate und Chart Interpreter werden hinsichtlich ihrer Leistungsfähigkeit verglichen.
- Die beiden vorgestellten Synchronisationsmethoden werden hinsichtlich der zur Synchronisation notwendigen Nachrichten und der erzielten Ausführungszeiten verglichen.
- Verschiedene Konfigurationen eines Workflowtyps werden hinsichtlich ihrer Ausführungszeiten verglichen.

Weitere quantitative Aussagen, insbesondere zu dem TP-Monitor Tuxedo, können in [Bur97] nachgelesen werden.

5.1. Vergleich der Workflow Engines

5.1.1. Meßumgebung

Die einzelnen Messungen wurden mittels eines Programms durchgeführt, das die Ausführungszeit von 1000 aufeinanderfolgenden Schritten in einem Chart mißt und anschließend die durchschnittliche Ausführungszeit für einen Schritt ausgibt. Jede Messung wurde zehn Mal wiederholt und anschließend der Mittelwert aller Messungen gebildet.

Die Messungen wurden auf einer SparcStation 5 mit 74 Megabyte Hauptspeicher unter dem Betriebssystem Solaris 2.5.1 durchgeführt, die außer dem Meßprogramm keine Anwenderprogramme ausführte. Die Codegröße des Meßprogramms lag unter einem Megabyte, die gesamte Messung lief vollständig im Arbeitsspeicher ohne Swapping ab.

5.1.2. Versuchsaufbau

Im Rahmen dieser Arbeit soll kein universeller Vergleich der beiden betrachteten Workflow-Engines angestellt werden. Ziel ist vielmehr, ihre Eignung zur Ausführung von partitionierten Workflowspezifikationen unter quantitativen Gesichtspunkten zu untersuchen. Die dafür ausschlaggebende Größe ist die Zeit, die für die Ausführung eines Schrittes in der Spezifikation benötigt wird, da sie unmittelbare Auswirkung auf die Dauer der Ausführungsphase in der CM-Hauptschleife hat. Die Zeit für das lokale Propagieren einer Änderung, die in der Lesephase gelesen wurde, ist im Gegensatz dazu bei beiden Engines vernachlässigbar klein und wird daher hier nicht betrachtet.

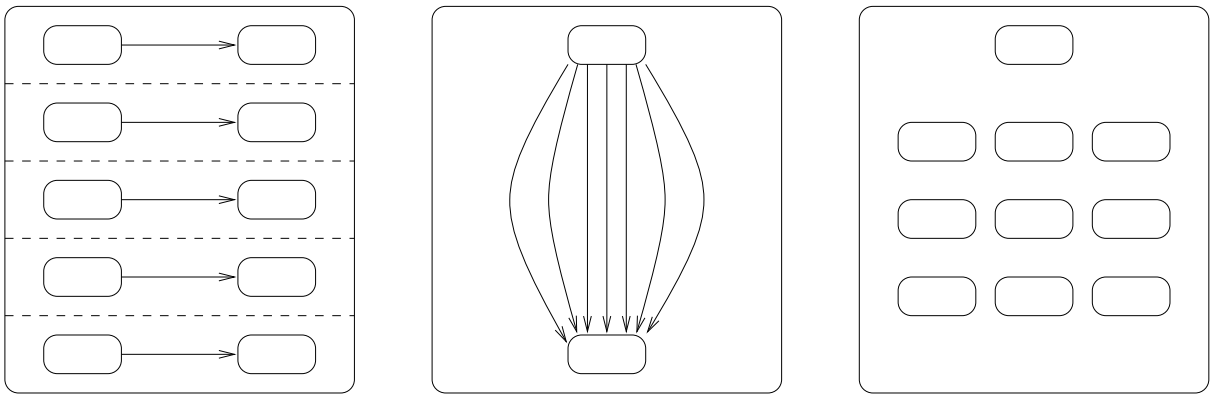

Abbildung 38 - Generische Spezifikationen zum Performancevergleich der Workflow Engines

Im folgenden wird nun die Abhängigkeit der Dauer der Ausführung eines lokalen Schrittes von verschiedenen Parametern der Spezifikation untersucht. Im einzelnen sind dies:

(1) Die Zahl der parallelen Komponenten in der Spezifikation

(2) Die Zahl der Transitionen, die aus betretenen Zuständen heraus gehen

(3) Die Zahl der Variablen in der Spezifikation

(4) Die Zahl der auf gleicher Ebene liegenden OR-Zustände in der Spezifikation

Alle Messungen werden anhand von generischen Spezifikationen durchgeführt, die sich nur in dem betrachteten Parameter ändern, die übrigen Parameter bleiben unverändert. Dadurch kann gezielt die Abhängigkeit von einer bestimmten Größe berechnet werden.Beispiele für diese Spezifikationen sind in Abbildung 38 dargestellt.

5.1.3. Meßergebnisse

Zur Messung des Falles (1) werden Statecharts mit n parallelen Komponenten verwendet. Jede Komponente enthält dabei einen IN- und einen OUT-Zustand, die mit einer Transition verbunden sind, an der eine nie erfüllte Schaltbedingung steht. Bei der Messung ergeben sich die folgenden Werte in Millisekunden:

Anzahl paralleler Komponenten	2	10	100	1000
Statemate	0,0046	0,0066	0,065	0,650
Interpreter	0,086	0,27	2,6	26,0

Die Ausführungszeit hängt also bei beiden Engines ab einer gewissen Größe linear von der Zahl der parallelen Komponenten ab. Statemate führt die Spezifikation dabei etwa um den Faktor 40 schneller aus als der Interpreter.

Im Fall (2) wird ein Statechart mit zwei Zuständen verwendet. Aus dem betretenen Zustand gehen n Transitionen aus, die eine nie erfüllbare Schaltbedingung haben. Bei der Messung ergeben sich die folgenden Werte in Millisekunden:

Anzahl Trans.	1	10	100	1000
Statemate	0,0040	0,0050	0,0151	0,150
Interpreter	0,034	0,170	1,770	18,05

Auch hier zeigt sich wieder eine lineare Abhängigkeit. Statemate liegt hier mit Faktoren zwischen 34 und 120 vorne. Zusammen mit Fall (1) ergibt sich: Verdoppelt man in jeder parallelen Komponente die Zahl der Transitionen zwischen den Zuständen, verdoppelt sich die Ausführungszeit höchstens[2].

Zur Betrachtung des Falles (3) wird ein Statechart mit nur einem Zustand verwendet, in dem aber n Variablen definiert, nach einer Initialisierung aber nicht mehr benutzt werden. In Workflowspezifikationen werden, speziell wegen der generischen Kontrollstrukturen, viele Variablen definiert, deshalb ist eine solche Betrachtung sinnvoll. Es ergeben sich die folgenden, teilweise überraschenden Werte in Millisekunden:

Anzahl Var.	1	10	100	1000
Statemate	0,0040	0,0040	0,0040	0,0040
Interpreter	0,032	0,042	0,142	1,410

Die Ausführungszeit des Interpreters hängt also im wesentlichen linear von der Anzahl der definierten Variablen ab, obwohl die Variablen überhaupt nicht verwendet werden. Dies liegt aber lediglich an der ineffizienten Implementierung; durch Optimierungen kann auch der Interpreter zu einer Laufzeit gebracht werden, die nur von der Anzahl der tatsächlich verwendeten Variablen abhängt.

Zum Abschluß wird in Fall (4) ein Statechart betrachtet, daß n OR-Zustände enthält, von denen einer betreten ist und zwischen denen es keine Transitionen gibt. Dieser Charttyp kommt im Zusammenhang mit der Verwendung als Workflow Engine nicht vor, er wird aber der Vollständigkeit wegen trotzdem untersucht (Werte in Millisekunden):

Anzahl OR-Z.	1	10	100	1000
Statemate	0,0040	0,0040	0,0040	0,0040
Interpreter	0,021	0,077	0,70	7,15

Man erwartet hier konstante Ausführungszeit, da sich bei der Abarbeitung der Spezifikation nichts ändern kann, da es keine Transitionen gibt. Die Laufzeit des Interpreters ist aber auch hier wieder praktisch linear in der Anzahl der Zustände. Auch das liegt an der einfachen Implementierung, durch Optimierungen kann auch hier konstante Laufzeit erreicht werden.

[2]Es kann auch Transitionen geben, die vom nicht betretenen Zustand ausgehen und daher die Ausführungszeit nicht beeinflussen.

5.1.4. Auswertung der Meßergebnisse

Beim Vergleich der Meßergebnisse der beiden Workflow Engines stellt man Faktoren von 34 bis 120 fest, die Statemate schneller als der Chart Interpreter ist. Die Ausführungszeit ist bei Statemate im Gegensatz zum Interpreter sogar unabhängig von der Zahl der Data Items und der Zahl der nicht betretenen OR-Zustände. Das liegt daran, daß Statemate unmittelbar compilierten C-Code ausführt, während der Interpreter mehr Zeit für das Interpretieren der Spezifikation benötigt. Allerdings ist der Interpreter aufgrund seiner Konzeption als Forschungsprototyp nicht performance-optimiert.

Absolut ist die Ausführungszeit bei in der Praxis vorkommenden Größen, die unter 100 parallelen Komponenten und unter 1000 Transitionen liegen, auch beim Interpreter immer noch unwesentlich gegenüber dem Kommunikationsaufwand. Die Ausführungszeit liegt nämlich im Bereich von einigen 10ms, während für die Kommunikation mitunter einige Sekunden benötigt werden, wie die Meßergebnisse in den folgenden Abschnitte zeigen. Es genügt daher, bei den folgenden Messungen ausschließlich die Variante mit dem Chart Interpreter zu benutzen, die einfacher handzuhaben ist, da keine langwierigen Compilierungsvorgänge wie bei Statemate erforderlich sind.

5.2. Meßumgebung für die folgenden Messungen

5.2.1. Performance Monitoring

In diesem Abschnitt werden die Möglichkeiten beschrieben, die der Kommunikationsmanager zum *Performance Monitoring* bietet. Der Begriff bedeutet in diesem Zusammenhang die Überwachung der Leistungsdaten des Kommunikationsmanagers selbst, also zum Beispiel die mittlere Ausführungszeit eines Schrittes der CM-Hauptschleife, die Zahl der verschickten Nachrichten usw. Er bezieht sich also auf die technische Seite der Workflowausführung. Im Gegensatz dazu steht das *Workflow Monitoring*, das die Überwachung der Ausführung des Workflows auf semantischer Ebene beinhaltet. Dabei geht es zum Beispiel um die mittlere Bearbeitungsdauer, die ein Ausführungsorgan für die Ausführung einer Aktivität benötigt, oder um das Finden von Engpässen bei der Workflowausführung. Diese Art von Monitoring wird hier nicht betrachtet.

Um an Informationen über die Leistungsdaten des Kommunikationsmanagers zu gelangen, muß eine Möglichkeit gefunden werden, diese aus dem laufenden System heraus von außen zugänglich zu machen. Damit aus den gewonnenen Daten sinnvolle Aussagen über die tatsächliche Workflowausführung gemacht werden können, muß der Übertragungsvorgang dabei so beschaffen sein, daß er möglichst wenig Einfluß auf die Ausführung selbst nimmt. Im Rahmen dieser Arbeit wurden dabei zwei Möglichkeiten vorgesehen:

- Ein Systemprozeß, der *Performance Monitor (PM)*, wird in die Workflowausführung eingebunden und erhält über die Kommunikationsverbindungen, die von der DT-Schicht zur Verfügung gestellt werden, laufend die aktuellen Leistungsdaten des Kommunikationsmanagers jeder Partition. Diese Methode hat den Nachteil, daß viele zusätzliche Nachrichten das Kommunikationsmedium belasten und damit die Leistungsdaten verfälschen. Sie steht damit im Widerspruch zu der Forderung, daß die Gewinnung von Leistungsdaten möglichst wenig Einfluß auf die Ausführung selbst haben soll.
- Die Leistungsdaten werden während der Workflowausführung gesammelt und erst an deren Ende in eine Datei geschrieben. Dadurch entstehen während der Ausführung keine verfälschenden Einflüsse, gleichzeitig sind aber auch zur Laufzeit keine Aussagen über die Leistungsdaten möglich.

Beide Methoden werden vom Kommunikationsmanager unterstützt. Bevor ihre Implementierungen kurz dargestellt werden, werden zunächst die zu ermittelnden Leistungsdaten und ihre Bestimmung im Kommunikationsmanager vorgestellt.

- Zunächst ist die Zahl der verschickten Nachrichten von Interesse, etwa um verschiedene Synchronisationsmethoden zu vergleichen. Zu diesem Zweck zählt der Kommunikationsmanager eingehende und versendete Nachrichten, getrennt nach Änderungs-, Synchronisations- und Kontrollnachrichten.
- Weiter ist die Zahl der ausgeführten Schritte wichtig. Der Kommunikationsmanager zählt dazu synchrone und asynchrone Schritte sowie, getrennt davon, Schritte mit und ohne lokale Änderungen.
- Für die Beurteilung der Leistung einer gesamten Rechnerkonfiguration sind außerdem die Ausführungszeiten jedes Schritts notwendig. Der Kommunikationsmanager mißt mittels der Systemfunktion `gethrtime()` die Ausführungszeiten von Lese-, Schreib- und Ausführungsphase, außerdem die Gesamtzeit für die Ausführung eines Schritts der CM-Hauptschleife. Die Auflösung dieser Messung liegt im Bereich von Mikrosekunden, was für eine Leistungsbeurteilung völlig ausreichend ist.
- Ebenfalls gezählt wird die Zahl der Transaktionsabbrüche.
- Die Zahl der lokalen Änderungen, die sich bei der Workflowausführung ergeben, ist zusammen mit der Zahl der verschickten Änderungsinformationen ein Maß für die Lokalität einer Spezifikation.

Viele dieser Daten sind nur bei der Bewertung verschiedener Algorithmen im Kommunikationsmanager von Interesse, etwa die Nachrichtenzahlen. Es genügt also, diese Daten am Ende der Workflowausführung zur Verfügung zu stellen. Andere Daten, zum Beispiel die Ausführungszeiten, können bereits während der Workflowausführung genutzt werden, um Engpässe zu erkennen und - zum Beispiel durch Migration einer Partition auf einen anderen Rechner - zu beheben. Dazu müssen sie natürlich bereits während der Workflowausführung zur Verfügung stehen; ansonsten kann man erst zukünftige Workflowausführungen beeinflussen. Konzeptionell können solche Rekonfigurationen vom Performance Monitor angestoßen werden, der über alle notwendigen Leistungsdaten verfügt. In dieser Arbeit wurde diese Seite des Performance Monitoring nicht betrachtet.

Der PM ist wie eine Partition in die Kommunikation eingebunden, synchronisiert sich aber nicht mit den übrigen. Das ist nicht erforderlich, da er nicht an der Workflowausführung selbst teilnimmt. Er wird in das Workflow-Konfigurationsfile eingetragen, das in Abschnitt 2.3 vorgestellt wurde, und hat dort als Prozeß den Typ `PERFMON`. Nach dem Start öffnet er für jede gestartete Partition des Workflows ein Überwachungsfenster; ein solches Fenster ist in Abbildung 39 dargestellt. Für die Realisierung dieser Fenster wurde das grafische Funktionspaket `xforms` ([Zha96]) benutzt. Am unteren Rand stellen Diagramme verschiedene Leistungsdaten grafisch dar, zum Beispiel die Anzahl der Transaktionsabbrüche, die Zahl der versendeten Nachrichten und die Schrittdauer.

Die Kommunikationsmanager jeder Partition schicken nun am Ende jedes Schritts eine Nachricht an den PM, die Daten enthält, die den Schritt charakterisieren. Der PM wertet diese Nachrichten aus und aktualisiert die Anzeigen im entsprechenden Fenster. Zusätzlich ist es möglich, daß eine Partition Ausgaben in das ihr zugeordnete Fenster des PM macht, in dem sie eine entsprechende Nachricht an den PM schickt. Diese Funktionalität ist zum Beispiel zur Fehlersuche im Kommunikationsmanager sehr nützlich. Allgemein erkennt man an den Ausgaben in den PM-Fenstern sofort, ob eine Partition noch arbeitet oder durch einen Fehler zum Stillstand gekommen ist. In diesem Fall

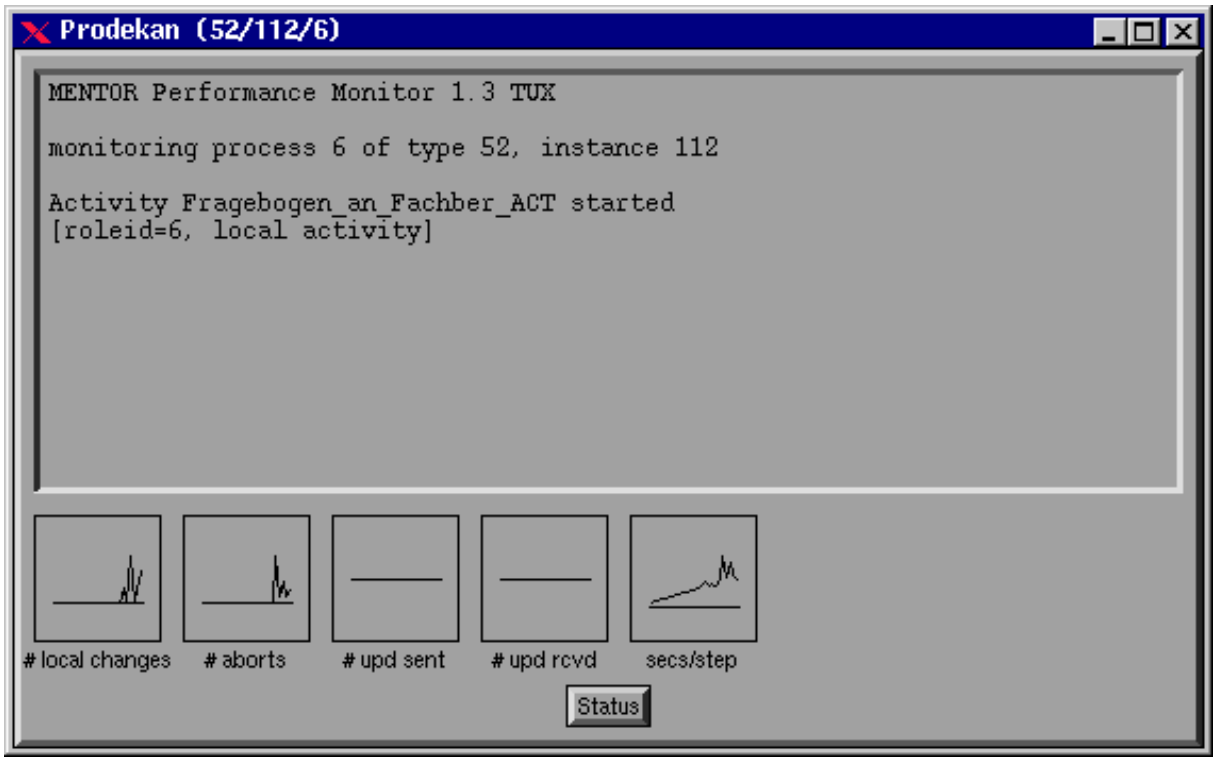

Abbildung 39 - Der Performance Monitor

wird das PM-Fenster nicht mehr aktualisiert, da keine entsprechenden Nachrichten von dieser Partition mehr kommen.

Für manche Anwendungen ist eine nachträgliche Auswertung der Leistungsdaten notwendig. Der Kommunikationsmanager schickt dazu die Nachrichten mit diesen Daten nicht sofort an den PM, sondern speichert sie zwischen. Erst am Ende der Workflowausführung werden diese Nachrichten zentral in eine Datei geschrieben, die anschließend von entsprechenden Analyseprogrammen verarbeitet werden kann. Insbesondere wurden Tools implementiert, die die in den folgenden Kapiteln beschriebenen Mittelwerte der Ausführungszeiten berechnen, die Zahl synchroner und asynchroner Schritte bestimmen, die grafische Ausgabe der Leistungsdaten durch `gnuplot` vorbereiten und die Zahl der Transaktionsabbrüche extrahieren. Als weitere Auswertungsmöglichkeit ist zum Beispiel die Bestimmung der Varianz der Ausführungszeiten denkbar. Auf die Darstellung der exakten Implementierung dieser Tools wird verzichtet.

5.2.2. Aktivitätssimulation

Um bei der Messung verschiedener Workflowausführungen vergleichbare Ergebnisse zu erzielen, können die Aktivitäten des Workflows nicht von einem menschlichen Benutzer ausgeführt werden, da von diesem unmöglich immer die gleichen Eingaben im gleichen zeitlichen Abstand gemacht werden können. Es ist daher erforderlich, statt der Ausführung der realen Aktivitäten besonderen Simluationscode zu benutzen, der Eingaben eines imaginären Anwenders in genau festgelegter Weise einspielt.

Für die Messungen im Rahmen dieser Arbeit wurde eine solche Funktionalität in den Chart Interpreter implementiert. Durch Setzen eines entsprechenden Flags schaltet man den Interpreter in den Evaluationsmodus, indem während der Ausführung eines Schritts der Spezifikation für jede

aktive Aktivität die besondere Funktion `cmEvaluateActivity()` aufgerufen wird. Diese Funktion vollzieht die Aktionen nach, die im normalen Ablauf des Workflows aufgrund von Benutzerinteraktionen von dem Code ausgeführt werden, der das der Aktivität zugeordnete Panel steuert: Dabei wird nach dem Starten einer Aktivität zunächst auf das Signal des Benutzers gewartet, diese Aktivität bearbeiten zu wollen. Dann wird das Panel auf dem Bildschirm dargestellt, in dem der Benutzer Eingaben vornehmen kann. Nachdem er seine Eingaben bestätigt hat, werden die zugehörigen Variablenwerte der Spezifikation aktualisiert und die Aktivität beendet. Auf der Spezifikationsseite enthält der Zustand, der der Aktivität zugeordnet ist, ein besonderes Unter-Statechart, das Zustände enthält, die den Ablauf der Bearbeitung der Aktivität beschreiben. Einzelheiten können der Arbeit von Holger Bielenstein [Bie97] entnommen werden. Die Funktion `cmEvaluate-Activity()` wird hier aus Verständnisgründen nur vereinfacht dargestellt, die reale Implementierung vollzieht tatsächlich verwendeten Mechanismus exakt nach.

Um die gewünschte Funktionalität zu realisieren, merkt sich die Funktion in eigens dazu eingeführten Strukturelementen der Activity-Struktur des Interpreters den Zeitpunkt des Starts der Aktivität. Ist seit diesem Start mehr als eine einstellbare Zeitspanne vergangen, wird das Startsignal des Benutzers simuliert, dies entspricht auf der Ebene der Spezifikation dem Generieren eines `ENTER`-Ereignisses. Gleichzeitig merkt sich die Funktion diesen Zeitpunkt. Nun geht die Workflowausführung normal weiter. Wird die Funktion in einem späteren Schritt erneut aufgerufen und sind ist dem Generieren des `ENTER`-Ereignisses mehr als eine ebenfalls einstellbare Zeit vergangen, wird die Bestätigung des Benutzers durch Setzen einer `OK`-Bedingung simuliert. Auf der Spezifikationsseite führt dies schließlich zum Beenden der Aktivität.

Einstellbar sind also die Zeit, die vom Starten der Aktivität bis zum Anfang ihrer Bearbeitung durch den Benutzer vergeht, sowie die Zeit zwischen Anfang und Ende der Bearbeitung. Bei den nachfolgenden Messungen wurde für beide Zeiten jeweils eine Minute vorgegeben, um die Gesamtzeit für die Messungen nicht zu groß werden zu lassen; reale Bearbeitungszeiten werden im allgemeinen höher liegen.

5.2.3. Meßworkflow

In den folgenden Abschnitten werden verschiedene Aspekte des Kommunikationsmanagers auf ihre Leistungsfähigkeit hin untersucht. Um dabei Einflüsse durch die ungeschickte Wahl einer realen Workflowspezifikation zu vermeiden, werden dazu sehr einfache Workflowspezifikationen verwendet, die zudem automatisch in der passenden Größe und mit der passenden Zahl von Partitionen generiert werden können.

Abbildung 40 zeigt das Statechart eines solchen generischen Workflows mit zwei Aktivitäten. Es steuert den Ablauf so, daß die Aktivitäten der Reihe nach abgearbeitet werden. Die Aktivitäten werden dabei reihum auf die Partitionen verteilt. Gibt es im Workflow also zum Beispiel zehn Aktivitäten und fünf Partitionen, sind Aktivität 1 und 6 in Partition 1, Aktivität 2 und 7 in Partition 2 usw. Dadurch wird erreicht, daß nach jedem Beenden einer Aktivität eine Aktivität in einer anderen Partition gestartet wird.

5.3. Vergleich der Synchronisationsmethoden

In diesem Abschnitt werden die beiden implementierten Synchronisationsmethoden, strikte und dynamische Synchronisation, im Hinblick auf die Zahl der zur Synchronisation notwendigen Nachrichten verglichen. Zur Messung wird dabei der bereits angesprochene generische Workflow

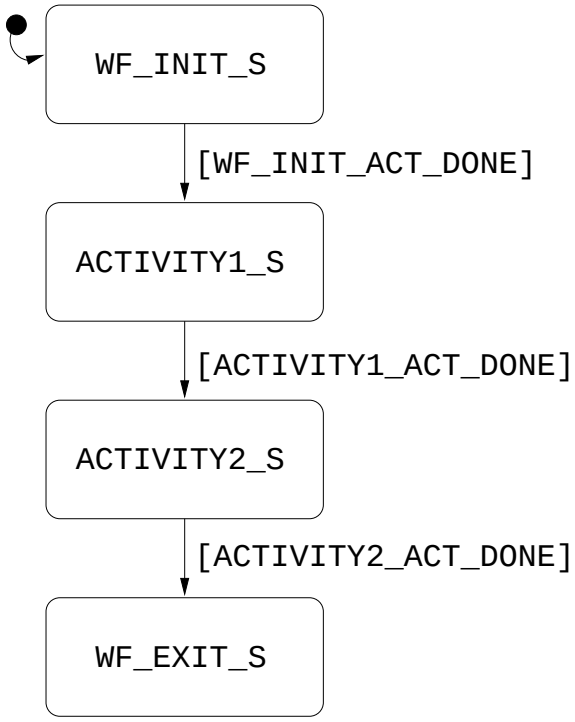

Abbildung 40 - Statechart eines automatisch erzeugten Workflows

mit zehn Aktivitäten und unterschiedlicher Zahl von Partitionen verwendet. Die Partitionen laufen auf einem ansonsten unbelasteten SparcStation10-Rechner ab.

Zunächst werden zwei Partitionen betrachtet. Die erste der beiden folgenden Tabellen stellt für diesen Fall die Gesamtzahl der Schritte sowie ihre Aufteilung in synchrone und asynchrone Schritte dar, die zweite die Gesamtzahl der Nachrichten sowie ihre Aufteilung in Synchronisations- und Änderungsnachrichten:

Typ	Schritte gesamt	synchron	asynchron	Anteil sync
strikte Synchron.	775	775	0	100%
dynamische Synchron.	569	116	453	20,4%

Typ	Nachrichten gesamt	Sync	Änderung	Anteil sync
strikte Synchron.	795	775	20	97,5 %
dynamische Synchron.	136	116	20	85,3%

Es ergibt sich wie bereits in Kapitel 4 angekündigt, daß dynamische Synchronisation wesentlich weniger Synchronisationsnachrichten benötigt als strikte Synchronisation, auch das Verhältnis der Zahl der Synchronisationsnachrichten zu der Gesamtzahl der Nachrichten ist besser. Mißt man zusätzlich noch die Gesamtzahl der Änderungen, die sich bei der Ausführung der Partition lokal ergeben, erhält man für beide Synchronisationsmethoden den Wert 111. Von diesen Änderungen

müssen wegen des in Kapitel 3 geschilderten Konzepts des gezielten Versendens von Änderungsnachrichten nur 20 an die andere Partition propagiert werden.

Vergleicht man die Schrittzahlen der beiden Methoden, fällt auf, daß mit dynamischer Synchronisation insgesamt weniger Schritte als mit strikter Synchronisation berechnet werden. Allerdings ist die Gesamtausführungszeit bei dynamischer Synchronisation mit 375 Sekunden höher als bei strikter mit nur 333 Sekunden. Erwartet hätte man das Gegenteil, da bei der dynamischen Methode weniger Nachrichten verschickt werden und somit weniger Kommunikationsaufwand nötig ist. Die naheliegende Lösung, daß dies an der dynamischen Verzögerung liegt, die in asynchrone Schritte eingebaut wird, trifft nicht zu. Eine genaue Messung der Mittelwerte der Ausführungszeiten für synchrone und asynchrone Schritte ergibt, daß dieser Unterschied statt dessen in der Synchronisationsphase begründet liegt. Zunächst die Werte der Messung bei der synchronen Methode, jeweils in Sekunden:

Typ	synchroner Schritt
Synchr./Lesephase	0,127
Ausführungsphase	0,011
Schreibphase	0,065
gesamt	0,203

Die Varianz der gemessenen Schrittlängen war 0,019. In der folgenden Tabelle sind die Werte für den dynamischen Fall aufgeführt. Zusätzlich wurden dabei die Werte für Resynchronisationsschritte gemessen, das sind die jeweils ersten synchronen Schritte nach einer Phase der asynchronen Ausführung.

Typ	synchroner Schritt	Resync-Schritt	asynchroner Schritt
Synchr./Lesephase	0,441	0,601	0,032
Ausführungsphase	0,011	0,011	0,011
Schreibphase	0,069	0,070	0,000
gesamt	0,521	0,682	0,043

Bei den asynchronen Schritten ist die dynamische Wartezeit nicht enthalten. Die Varianz der Schrittlängen ist 0,178 bei synchronen Schritten, 0,171 bei Resynchronisationsschritten und 0,004 bei asynchronen Schritten. Die Ausführungszeiten synchroner Schritte variieren also bei dynamischer Synchronisation wesentlich stärker als bei strikter Synchronisation. Außerdem fällt auf, daß die hier Synchronisationsphase im Mittel wesentlich länger dauert als bei strikter Synchronisation. Weitere Messungen ergeben im Mittel 3,4 Abbrüche der Transaktion in der CM-Leseschleife, bevor Synchronisation erfolgt ist. Im strikten Fall sind im Mittel lediglich 0,57 Abbrüche notwendig, also nur etwa in jedem zweiten Schritt. Dieses Phänomen zeigt sich auch bei anderen Vergleichsmessungen und ist also nicht in einem Meßfehler begründet. Im Rahmen dieser Arbeit wurde keine schlüssige Erklärung dafür gefunden.

Die höhere mittlere Ausführungszeit der Synchronisationsphase bei Resynchronisationsschritten hat ihre Ursache darin, daß in diesem Fall gemäß dem im Kapitel 4 vorgestellten Algorithmus ein

Transaktionsabbruch, das Verschicken einer Synchronisationsnachricht an alle übrigen Partitionen, ein Transaktionscommit und ein Beginnen einer neuen Transaktion zusätzlich zur normalen Synchronisationsphase ausgeführt werden müssen.

Im folgenden werden nun die gleichen Messungen mit fünf Partitionen durchgeführt. Hier zunächst die Messungen der Schritt- und Nachrichtenzahlen:

Typ	Schritte gesamt	synchron	asynchron	Anteil sync
strikte Synchron.	178	178	0	100%
dynamische Synchron.	355	116	195	32,7%

Typ	Nachrichten gesamt	Sync	Änderung	Anteil sync
strikte Synchron.	720	712	8	98,9 %
dynamische Synchron.	472	464	8	98,3%

Man sieht auch hier einen absoluten Vorteil für die dynamische Methode, während das Verhältnis der Zahl der Synchronisationsnachrichten zur Gesamtzahl der Nachrichten bei beiden Methoden etwa gleich ist. Allgemein kann man sagen, daß der Anteil der Synchronisationsnachrichten mit steigender Partitionszahl immer zunimmt, bis die Zahl der Änderungsnachrichten praktisch vernachlässigbar ist.

Auch in diesem Fall kann man Mittelwerte und Varianz der Ausführungszeiten betrachten. Zunächst wieder die der strikt synchronisierten Variante:

Typ	synchroner Schritt
Synchr./Lesephase	1,302
Ausführungsphase	0,010
Schreibphase	0,722
gesamt	2,034

Die Zeiten sind hier nicht nur viermal so hoch, wie man es erwartet, da viermal soviele Synchronisationsnachrichten verschickt werden müssen. Statt dessen dauern Synchronisations- und Schreibphase nun im Mittel etwa zehnmal so lange wie mit zwei Partitionen. Dies läßt sich damit erklären, daß alle Partitionen auf dem gleichen Rechner laufen, der stark durch die Kommunikation belastet ist. Die Varianz der Schrittdauer ist 0,369 ebenfalls wesentlich größer.

Die Meßwerte für den dynamischen Fall ergeben ein ähnliches Bild wie mit zwei Partitionen:

Typ	synchroner Schritt	Resync-Schritt	asynchroner Schritt
Synchr./Lesephase	1,571	2,396	0,265
Ausführungsphase	0,010	0,010	0,010
Schreibphase	0,829	0,862	0,000
gesamt	2,410	3,268	0,275

Die Varianzen sind hier 0,867 für synchrone Schritte, 0,924 für Resynchronisationsschritte und 0,316 für asynchrone Schritte. Auch hier zeigt sich wieder ein Laufzeitnachteil für die dynamische Methode, der aber kleiner als im Fall von zwei Partitionen ist. Die Zahl der Transaktionsabbrüche liegt nun näher beeinander: Im strikten Fall wird die Lesetransaktion etwa einmal je Schritt abgebrochen, während es bei der dynamischen Methode etwa 1,9 Abbrüche je Schritt gibt.

Zum Abschluß dieses Abschnittes zeigt Abbildung 41 grafisch die Dauer der einzelnen Schritte bei der Ausführung einer Partition mit dynamischer Synchronisation. Man erkennt zwischen den synchron laufenden Teilen, die sich durch ungefähr konstante, große Schrittlängen auszeichnen, die asynchronen Bereiche, in denen die Schrittlänge ungefähr linear anwächst. Dies kommt von der dynamischen Wartezeit, die bei der asynchronen Ausführung eingebaut wird. In diesem Fall ist die Wartezeit im Verhältnis zu der zur Ausführung eines synchronen Schritt benötigten Zeit sehr klein, sie beträgt zwischen 10 und 1000 Millisekunden. Man hätte ihren Maximalwert auch bis zu einigen Sekunden wählen und so die Zahl der asynchronen Schritte weiter reduzieren können, da die Reaktionszeit des Systems auf Benutzereingaben von der Zeit bestimmt wird, die ein synchroner Schritt benötigt.

Zusammenfassend bestätigen die in diesem Abschnitt vorgestellten Messungen, daß dynamische Synchronisation im Vergleich zu strikter wesentlich weniger Nachrichten benötigt, um einen korrekten Ablauf des Workflows zu gewährleisten.Mit zunehmender Partitionszahl steigt bei beiden untersuchten Methoden der Anteil der Synchronisationsnachrichten an der Gesamtzahl der verschickten Nachrichten stark an, so daß schließlich die Nachrichten, die Änderungsinformationen enthalten, vernachlässigbar werden. Die Gesamtnachrichtenzahl wird von der Zahl der Synchronisationsnachrichten dominiert.

5.4. Quantitativer Vergleich verschiedener Konfigurationen

In diesem Abschnitt wird wieder der generische Workflow mit fünf Partitionen und dynamischer Synchronisation benutzt. Zunächst wird untersucht, ob und inwieweit die mittlere Ausführungszeit von der verteilten Ausführung des Workflows abhängt. Anschließend wird der Einfluß eines parallel laufendenden Workflow auf die Ausführungszeiten betrachtet.

Die Partitionen des Workflows wurden für die Messungen auf fünf Sun-SparcStations unterschiedlicher Leistungsfähigkeit verteilt, die über ein 10MBit-Ethernet verbunden sind. Der leistungsärmste der Rechner war dabei eine SparcClassic, der leistungsstärkste eine SparcStation 10. Alle Rechner waren zum Zeitpunkt der Messung nicht mit anderen Prozessen belastet.

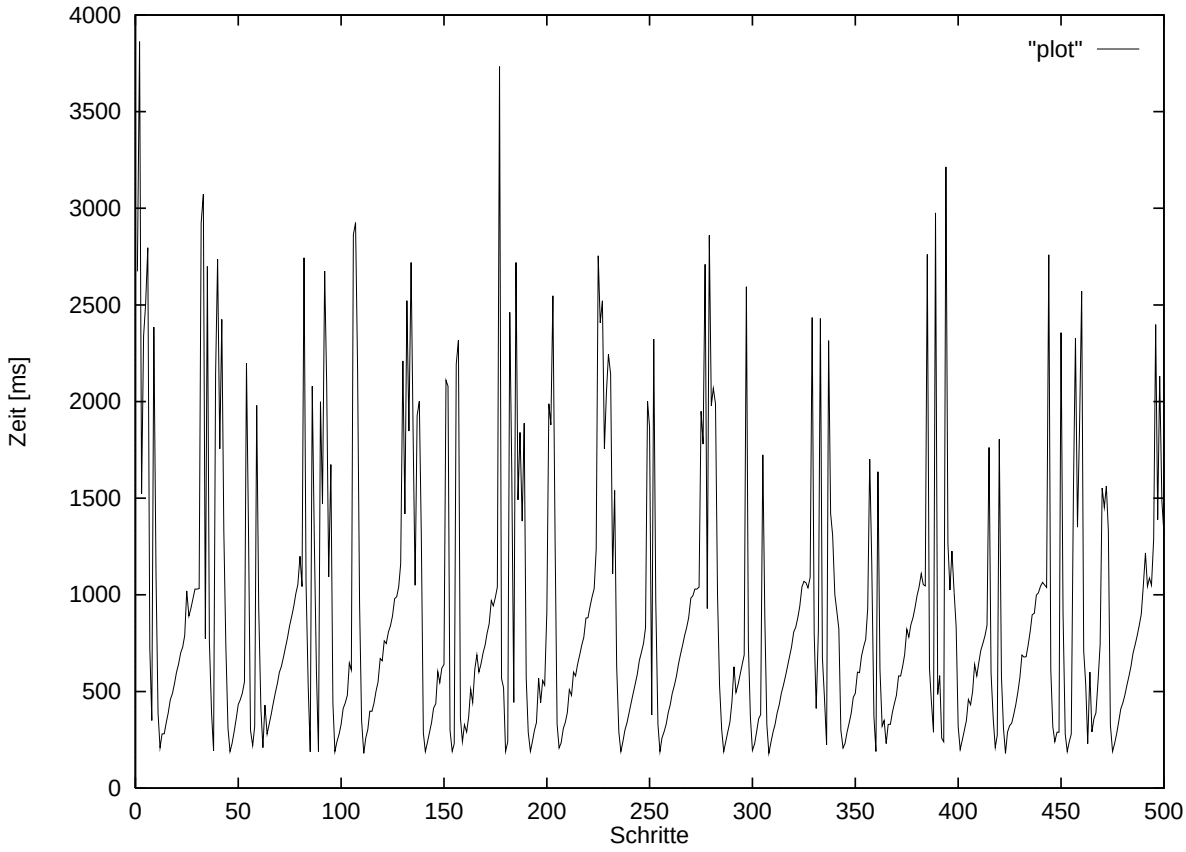

Abbildung 41 - Schrittlängengrafik einer Partitionsausführung

Mißt man nun die mittleren Ausführungszeiten auf der SparcStation 10, erhält man die folgenden Werte:

Typ	synchroner Schritt	Resync-Schritt	asynchroner Schritt
Synchr./Lesephase	0,573	1,247	0,104
Ausführungsphase	0,010	0,010	0,010
Schreibphase	0,372	0,393	0,000
gesamt	0,955	1,650	0,114

Man erhält etwa die halben Ausführungszeiten gegenüber dem Fall, in dem alle Partitionen auf diesem Rechner ablaufen. Dies erklärt sich damit, daß der Rechner weniger belastet ist. Nun müssen aber alle Nachrichten über das Netz transportiert werden, was vorher nicht notwendig war, so daß keine besseren Zeiten zustande kommen. Die Varianzen sind ebenfalls wesentlich niedriger (0,162 für synchrone Schritte, 0,261 für Resynchronisationsschritte und 0,017 für asynchrone Schritte).

Nun werden die Ausführungszeiten auf der SparcClassic betrachtet:

Typ	synchroner Schritt	Resync-Schritt	asynchroner Schritt
Synchr./Lesephase	0,594	1,210	0,106
Ausführungsphase	0,010	0,010	0,010
Schreibphase	0,360	0,420	0,000
gesamt	0,964	1,640	0,116

Es zeigt sich, daß die Ausführungszeiten auf dem schwächeren Rechner im wesentlichen die gleichen wie auf dem leistungsstarken Rechner sind, die Varianzen sind ebenfalls vergleichbar. Ähnliche Ergebnisse liefern die Analysen der Ausführungszeiten der übrigen an der Workflowausführung beteiligten Rechner. Synchronisation bremst also wie erwartet die schnelleren Rechner auf die Geschwindigkeit des langsamsten.

Startet man parallel einen weiteren Workflow gleichen Typs auf den gleichen Rechnern, ergibt sich eine höhere Ausführungszeit bei allen Partitionen, was durch die höhere Belastung der Rechner und der Kommunikationsinfrastruktur begründet werden kann. Hier als Beispiel die mittleren Ausführungszeiten auf der SparcStation 10:

Typ	synchroner Schritt	Resync-Schritt	asynchroner Schritt
Synchr./Lesephase	0,853	1,600	0,127
Ausführungsphase	0,010	0,010	0,010
Schreibphase	0,552	0,470	0,000
gesamt	1,415	2,080	0,137

Die Varianzen (0,237 für synchrone Schritte, 0,913 für Resynchronisationsschritte und 0,063 für asynchrone Schritte) deuten darauf hin, daß die tatsächlichen Ausführungszeiten dabei stark um die Mittelwerte schwanken können. Erschwerend kommt dabei hinzu, daß die beiden Workflows in der Regel zur gleichen Zeit synchrone Schritte machen, wenn sie ungefähr zur gleichen Zeit gestartet wurden. Wählt man den Startzeitpunkt des zweiten Workflows geschickt, so daß die synchronen Schritte des einen Workflows gerade in die Phase der asynchronen Ausführung des anderen Workflows fallen, verändern sich die mittleren Ausführungszeiten nur unwesentlich.

Bei einem durchgeführten Belastungstest mit vier derartigen Workflows ergaben sich bei der verwendeten Version von Tuxedo erhebliche Probleme, die sich in wiederholten Nachrichtenverlusten, fehlgeschlagenen Transaktionscommits und ähnlichen Fehlern äußerten. Es muß noch untersucht werden, ob diese Schwierigkeiten an der genutzten Version liegen oder ein inhärentes Problem dieses TP-Monitors sind.Wegen dieser Probleme, und da die verwendete Tuxedo-Version nur eine begrenzte Zahl gleichzeitig laufender Prozesse unterstützt, konnten keine Messungen mit mehreren parallel laufenden Workflows durchgeführt werden.

Zusammenfassend zeigen die Messungen in diesem Abschnitt, daß sich die mittleren Ausführungszeiten eines Workflows wesentlich verbessern lassen, wenn die Partitionen auf verschiedenen Rechnern statt auf dem gleichen Rechner ausgeführt werden. Die Ausführungszeit der synchronen Schritte orientiert sich dabei an der Partition, die auf dem langsamsten Rechner läuft. Parallel laufende Workflows, die gleichzeitig synchrone Schritte machen, beeinflussen sich gegenseitig relativ stark; parallel laufende asynchrone Workflows stören den Ablauf nur unwesentlich.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

In der vorliegenden Diplomarbeit wurde die Konzeption des Kommunikationsmanagers für MENTOR und die Implementierung eines lauffähigen Prototypen vorgestellt. Das Ziel war es, eine verteilte Umgebung für die korrekte Ausführung von Workflows zu schaffen. Als wesentliche Eigenschaften dieser Ausführungsumgebung waren dabei Skalierbarkeit und Fehlertoleranz gefordert. Außerdem sollte der Kommunikationsmanager eine hohe Performance erreichen und leichte Bedienbarkeit garantieren.

Der Kommunikationsmanager basiert auf einer Schichtenarchitektur, die die Intergation verschiedener Workflow-Engines zur Ausführung der partitionieren Spezifikationen und unterschiedlicher Funktionspakete zur Gewährleistung sicherer, transaktionsorientierter Kommunikation ermöglicht. Dazu wurde ein Kernmodul geschaffen, daß die korrekte verteilte Ausführung von Workflows durch Propagation von Änderungsinformation bei möglichst geringen Kommunikationskosten gewährleistet. Als Workflow Engines wurde die kommerzielle Ausführungsumgebung Statemate und der im Rahmen des Projektes entwickelte Chart Interpreter an den Kommunikationsmanager angebunden. Verschiedene Möglichkeiten zur Synchronisation wurden implementiert, die eine korrekte verteilte Workflowausführung ermöglichen, außerdem wurde ein Konzept zum dynamischen Starten und Beenden von Workflowpartitionen vorgestellt. Schließlich wurden verschiedene Leistungsaspekte des Gesamtsystems aus quantitativer Sicht untersucht.

In der Implementierung lag der Schwerpunkt bei der Schaffung eines Prototypen, mit dem beliebige Workflowspezifikationen verteilt ausgeführt werden können. Besonderer Wert mußte dabei auf die korrekte Zusammenarbeit des Kommunikationsmanagers mit anderen, im Rahmen des MENTOR-Projektes entwickelten Systembestandteilen gelegt werden, wie zum Beispiel dem Partitionierer, dem Chart Interpreter oder dem Logmanager.

Da bei der Implementierung viele verschiedenen Softwarekomponenten integriert werden mußten, die von den an MENTOR Beteiligten entwickelt wurden, konnten wertvolle Erfahrungen im koordinierten Arbeiten in einem Team gesammelt werden. Unerklärliche Phänomene im Zusammenhang mit fremder Software machte es immer wieder erforderlich, Fehler darin zu suchen und zu umgehen. Durch die weite Streuung der verwendeten Softwarekomponenten konnten Erfahrungen mit den verschiedensten Softwarekomponenten gesammelt werden, im einzelnen mit dem TP-Monitor Tuxedo, der Oracle-Datenbank, dem Spezifikationssystem Statemate, der grafischen Funktionensammlung xforms, aber auch mit der Versionskontrollsoftware RCS, den Programmiertools von GNU und diversen UNIX-Systemprogrammen.

6.2. Ausblick

Weil es den Rahmen dieser Diplomarbeit gesprengt hätte, wurden von den in Kapitel 4 vorgestellten Synchronisationsprotokollen nur strikte und dynamische Synchronisation implementiert. Insbesondere wurde die Bestimmung von möglichst günstigen Synchronisationspunkten nicht betrachtet, die sicher ein interessanter Ansatzpunkt für zukünftige Arbeiten ist. Ebenso wurden die in Abschnitt 4.7.4 gezeigten Optimierungsmöglichkeiten beim dynamischen Starten von Partitionen nicht weiter verfolgt; die genannten Vorteile im Bezug auf die insgesamt notwendige Nachrichtenmenge rechtfertigen hier eine genauere Untersuchung. Wenn auch die fehlertolerante dynamische Einbindung von Partitionen gefordert ist, müssen entsprechende Konzepte in den Dynamischen

Prozeßmanager bzw. in die entsprechenden Algorithmen integriert werden. Nicht betrachtet wurde außerdem die Anbindung des Worklist Managers, durch die vorhandene Schnittstelle im DPM ist sie aber ohne wesentlichen Aufwand möglich.

Die in Kapitel 3.5 vorgestellten alternativen Konzepte für den Datenaustausch zwischen den Partitionen wurden aus Zeitgründen nicht implementiert. Insbesondere im Hinblick auf eine Verbesserung der Gesamtleistung des Kommunikationsmanagers sollten die genannten, aber auch weitere Datenkonsistenzprotokolle aus dem Bereich der verteilten Datenbanksysteme auf ihre Verwendbarkeit in diesem System untersucht werden.

Der modulare Aufbau des Kommunikationsmanagers erlaubt eine reibungslose Integration der verschiedensten Datentransportschichten und Workflow Engines. Besonders interessant ist die in Abschnitt 3.1.1 angesprochene Anbindung von nicht statechart-basierten Workflow Engines, was eine entsprechende Konvertierung der Spezifikationen erforderlich macht, und die Verwendung alternativer TP-Monitore bzw. von CORBA-Implementierungen, die einen vergleichbaren Funktionsumfang bieten, aber möglicherweise einfacher handzuhaben sind.

Die Evaluation des implementierten Prototypens, die in Kapitel 5 in einigen wichtigen Punkten durchgeführt wurde, kann auf weitere Aspekte erweitert werden. Insbesondere sollte ein Vergleich der durch Messungen erhaltenen Werte mit den Simulationsergebnissen in [Bur97] angestellt werden, um die Güte der Simulation bestimmen zu können. Außerdem sollte versucht werden, Engpässe und Schwachstellen herauszufinden, auf die es bei der Konfiguration der Ausführungsumgebung für Workflows zu achten gilt.

A. Dateienstruktur des Kommunikationsmanagers

Abbildung 42 zeigt die Aufteilung des Kommunikationsmanagers in einzelne Dateien und Bibliotheken sowie die Struktur der Aufrufe zwischen diesen.

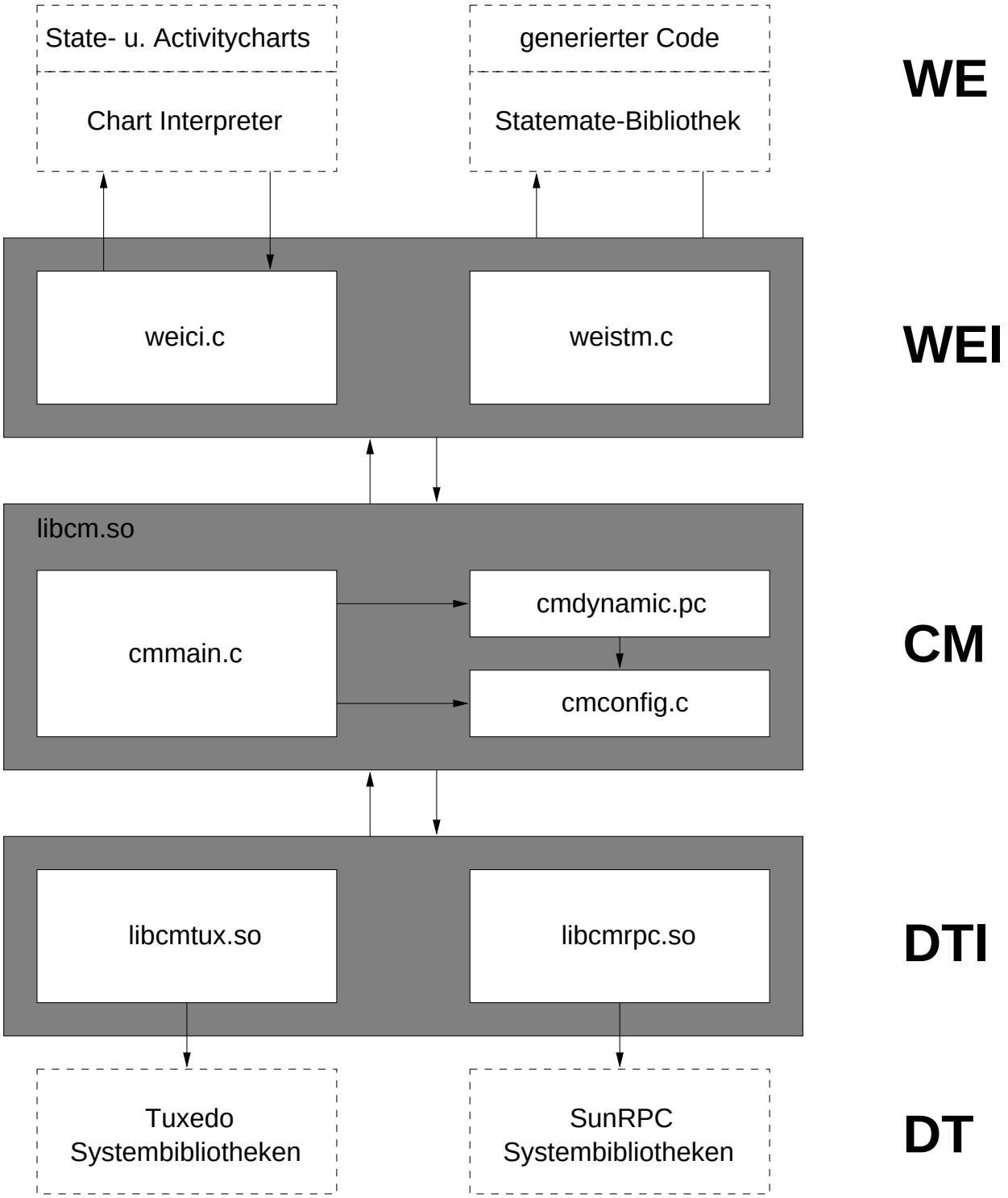

Abbildung 42 - Aufruf- und Dateistruktur des Kommunikationsmanagers

B. Struktur- und Funktionsdefinitionen zur Verwaltung der Konfiguration eines Workflows

Die im folgenden beschriebenen Strukturen und Funktionen sind kein Bestandteil des eigentlichen Kommunikationsmanagers, sondern dienen der Verwaltung der Konfiguration einer Workflowinstanz., das heißt der an einer Instanz beteiligten Partitionen, Teilprozesse und Ausführungsorgane. Sie werden hier kurz vorgestellt, weil wesentliche Module des Kommunikationsmanagers regen Gebrauch von ihnen machen.

Alle genannten Strukturen und Listen werden beim Start eines Workflow-Teilprozesses ihm Rahmen der Initialisierung des Kommunikationsmanagers mit Werten gefüllt. Bei statischen Teilprozessen genügt es, die Daten aller System- und statischen Teilprozesse zu laden, da solche Teilprozesse immer beim Start einer Instanz gestartet werden und zu diesem Zeitpunkt keine dynamischen Teilprozesse aktiv sind. Dynamische Teilprozesse wiederum entnehmen den in Kapitel 2.3 dargestellten Relationen in der Datenbank, welche Teilprozesse gerade aktiv sind, und füllen die Strukturen mit deren Daten.

B.1. Workflowkonfiguration - cfgWorkflowInfo

Grundlegende Informationen zum ausgeführten Workflowtyp werden in der Struktur `cfgWorkflowInfo` gespeichert, auf die zur Laufzeit die Variable `workflowinfo` zeigt.

```
typedef struct
{ char    *name;        /* Name des Workflows */
  char    *version;     /* Versionsbezeichnung des Workflows */
  char    *author;      /* Autor der Workflowspezifikation */
  char    *creator;     /* Partitionierer des Workflows */
  char    *partitioned; /* Datum der Partitionierung */
  int      type;        /* WorkflowTyp des Workflows */
 /* Ergänzungen für Tuxedo */
  int      master;      /* Master-Rechner der Tuxexo-
                           Konfiguration */
  int      backup;      /* Backup-Master-Rechner der Tuxedo-
                           Konfiguration */
  int      configid;    /* Nummer der Tuxedo-Systemkonfiguration,
                           in der der Workflow läuft */
} cfgWorkflowInfo, *cfgWorkflowInfoPtr;
```

B.2. Rollenkonfiguration - cfgRoleInfo

Die Partitionen bzw. Rollen, die an der Ausführung einer Workflowinstanz beteiligt sind, werden in einer einfach verketteten Liste der nachfolgend beschriebenen `cfgRoleInfo`-Strukturen verwaltet. Die Variable `rolelist` zeigt dabei auf die erste Struktur in dieser Liste, die Variable `ThisRole` auf die Struktur der Partition des eigenen Teilprozesses.

```
typedef struct
{ cfgRoleInfoPtr   next;      /* nächste Rolle in dieser
                                 Workflowspezifikation, einfach
                                 verkettete Liste */
  char            *name;      /* Bezeichnung dieser Rolle */
  int              roleid;    /* ID dieser Rolle */
  roleprocptr      firstproc; /* Zeiger auf Liste von Prozessen,
                                 die diese Rolle wahrnehmen
                                 können */
  int              numprocs;  /* Anzahl der Prozesse in dieser
                                 Liste */
  char            *homedir;   /* Verzeichnis, in dem die weiteren
                                 Dateien stehen, die diese Rolle
                                 beschreiben (s.u.) */
  char            *chart;     /* Activitychart für diese Rolle */
  char            *svbfile;   /* Datei, in der die in den Charts
                                 dieser Rolle verwendeten
                                 Variablen stehen */
  char            *fgfile;    /* Datei, in der Informationen über
                                 die von den Aktivitäten dieser
                                 Rolle verwendeten Oberflächen
                                 stehen */
} cfgRoleInfo,*cfgRoleInfoPtr;
```

B.3. Konfiguration der Ausführungsorgane - cfgEntityInfo

Die Ausführungsorgane, die an der Ausführung einer Workflowinstanz beteiligt sind, werden in einer einfach verketteten Liste der nachfolgend beschriebenen `cfgEntityInfo`-Strukturen verwaltet. Die Variable `entitylist` zeigt dabei auf die erste Struktur in dieser Liste, die Variable `ThisEntity` auf die Struktur des Ausführungsorgans, das den eigenen Teilprozeß ausführt.

```
typedef struct
{ cfgEntityInfoPtr   next;    /* nächstes Ausführungsorgan in
                                 dieser Workflowinstanz */
  char              *name;    /* Bezeichnung des Ausführungs-
                                 organs */
  int                entityid;/* eindeutige ID des Ausführungs-
                                 organs */
} cfgEntityID, *cfgEntityIDPtr;
```

B.4. Prozeßkonfiguration - cfgProcInfo

Die jeweils aktiven Teilprozesse innerhalb einer Instanz eines Workflowtyps werden in einer einfach verketteten Liste von den nachfolgend beschriebenen `cfgProcInfo`-Strukturen verwaltet. Die Variable `proclist` zeigt auf die erste Struktur in dieser Liste, die Variable `ThisProcPtr` auf die Prozeßstruktur des eigenen Teilprozesses.

```
typedef struct
{ cfgProcInfoPtr   next;      /* nächster aktiver Prozeß in
                                 dieser Workflowinstanz */
  char            *name;      /* Bezeichnung dieses Prozesses */
  int              type;      /* Typ dieses Prozesses (s.u.) */
  int              procid;    /* eindeutige ID dieses
                                 Prozesses */
  int              roleid;    /* ID der Rolle, zu der dieser
                                 Prozeß gehört */
  int              entityid;  /* ID des Ausführungsorgans, das
                                 diesen Prozeß ausführt */
  char            *host;      /* Rechner, auf dem dieser Prozeß
                                 ausgeführt wird */
  hostlistptr      hosts;     /* Liste von Rechnern, auf denen
                                 dieser Prozeß ausgeführt werden
                                 kann */
  char            *display;   /* Display, auf dem die Ausgaben
                                 dieses Prozesses erscheinen */
  char            *filename;  /* Name der ausführbaren Programm-
                                 Datei dieses Prozesses */
 /* Synchronisation */
  int              synced;    /* Prozeß hat Synchronisations-
                                 Nachricht geschickt */
  int              syncsent;  /* An diesen Prozeß wurde bereits
                                 eine Synchronisationsnachricht
                                 geschickt */
/* Tuxedo-spezifische Einträge */
  int              qspid;     /* ID des Queuespaces, der diesem
                                 Prozeß zugeordnet ist */
  char            *machinename;
  char            *location;
  char            *orgunit;   /* Informationen über den Standort
                                 des Rechners, der diesen
                                 Prozeß ausführt */
} cfgProcInfo, *cfgProcInfoPtr;
```

Mögliche Prozeßtypen:

PT_STATIC	statischer Prozeß, der während der ganzen Ausführung des Workflows läuft
PT_DYNAMIC	Prozeß, der erst zur Laufzeit des Workflows dynamisch eingebunden wird und auch wieder zur Laufzeit beendet wird
PT_DPM	Dynamischer Prozeßmanager (Systemprozeß)
PT_PERFMON	Performance Monitor (Systemprozeß)
PT_HISTORYMANAGER	History Manager (Systemprozeß)

Nur die ersten beiden Typen werden für Teilprozesse einer Workflowinstanz benutzt, die übrigen sind Prozesse des Mentorsystems. Der Dynamische Prozeßmanager DPM wird im Rahmen der

dynamischen Einbindung von Workflowprozessen in Kapitel 4, der Performance Monitor als Hilfsmittel zur Evaluation in Kapitel 5 beschrieben. Auf den History Manager wird in [Klä97] eingegangen.

B.5. Funktionsdeklarationen

Funktionen zur Verwaltung der Konfiguration des Workflows:

```
int cfgReadProc(int procid, int flag);
     /* cfgProcInfo-Struktur des Prozesses mit der angegebenen
        ID aus der Konfigurationsdatei lesen und an die proclist
        anhängen. Falls flag==0, werden außerdem die in der SVB-
        Datei des Prozesses genannten Variablen für diesen Prozeß
        abonniert */

int cfgReadStaticProcs(void);
     /* Einlesen aller statischen Prozesse */

int cfgReadProcs(void);
     /* Prozeßstrukturen aller aktiven Prozesse aus der Konfi-
        gurationsdatei einlesen und Prozeßliste aufbauen, außerdem
        Variablen entsprechend den SVB-Dateien abonnieren */

int cfgReadRoles(void);
     /* Alle Rollen aus der Konfigurationsdatei einlesen und
        Rollenliste aufbauen */

int cfgReadEntities(void)
     /* Alle Ausführungsorgane aus der Konfigurationsdatei
        einlesen und entsprechende Liste aufbauen */
```

Funktionen zum Suchen von Strukturen

```
cfgProcInfoPtr cfgFindProc(int procid);
     /* Suchen des Prozesses mit der ID procid */
cfgProcInfoPtr cfgFindProc4Role(int roleid);
     /* Suchen des Prozesses, der Rolle roleid ausführt */

cfgRoleInfoPtr cfgFindRole(int roleid);
     /* Suchen der Rolle mit der ID roleid */
cfgRoleInfoPtr cfgFindRoleByName(char *name);
     /* Suchen der Rolle mit der Bezeichnung name */

cfgEntityInfo cfgFindEntity(int entityid);
     /* Suchen des Ausführungsorgans mit der ID entityid */
```

C. Funktionsübersicht des Kommunikationsmanagers

In diesem Abschnitt werden die Funktionen der Komponenten von Mentor tabellarisch aufgeführt, die in dieser Arbeit angesprochen wurden. Bei allen Funktionen werden dabei die Schicht im Kommunikationsmanager, zu der sie gehören, die Programmdatei, in der sie definiert werden, sowie der Abschnitt in dieser Arbeit, in der sie vorgestellt werden, aufgelistet.

Funktionsname	**Schicht**	**Datei**	**Abschnitt**
`cmMakeMsg()`	CM	cmmain.c	3.2.4
`cmMakeUpdateMsg()`	CM	cmmain.c	3.2.4
`cmConsumeMsg()`	CM	cmmain.c	3.2.2
`cmResetVars()`	CM	cmmain.c	3.2.3
`cmStartActivityCallback()`	CM	cmmain.c	3.2.3
`cmChangeCallback()`	CM	cmmain.c	3.2.3
`cmSendSyncMsg()`	CM	cmmain.c	4.2.2, 4.3.2
`cmSendUpdates()`	CM	cmmain.c	3.2.4
`cmHandleSyncMessage()`	CM	cmmain.c	4.2.2, 4.3.2
`cmRequeueSyncs()`	CM	cmmain.c	4.2.2
`cmClearRequeueSyncs()`	CM	cmmain.c	4.2.2
`cmHandleCtrlMessage()`	CM	cmmain.c	3.2.2, 4.2.2
`cmAddCtrlMessage()`	CM	cmmain.c	3.2.2, 4.2.2
`cmHandleCtrlMsgs()`	CM	cmmain.c	3.2.2
`cmResetSyncProcs()`	CM	cmmain.c	4.2.2
`cmClearMessages()`	CM	cmmain.c	3.2.6
`cmMain()`	CM	cmmain.c	3.2
`cm()`	CM	cmmain.c	3.2.5

D. Literaturverzeichnis

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, El Abbadi, R. Günthör, C. Mohan. *Failure Handling in Large Scale Workflow Management Systems*. IBM Research Report RJ 1993. San Jose, November 1994

[Bie97] H. Bielenstein. *Entwicklung einer Spezifikationsumgebung für unternehmensweite Workflows*. Diplomarbeit. Universität des Saarlandes. Saarbrücken, 1997

[BSW95] H. Bielenstein, R. Schenkel, M. Wehrmann. *Dokumentation zum Fortgeschrittenenpraktikum "Workflowmanagement mit Statemate"*, Universität des Saarlandes. Saarbrücken, 1995

[Bur97] J. Bur. *Entwicklung eines Konfigurationswerkzeuges zur optimalen Plazierung der Komponenten eines verteilten Workflow-Management-Systems*. Diplomarbeit in Vorbereitung. Universität des Saarlandes. Saarbrücken, 1997

[CLR90] T. Corman, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press, 1990

[GR93] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers. San Mateo, 1993

[Har87] D. Harel. Statecharts: *A Visual Formalism for Complex Systems*. Science of Computer Programming Vol 8, 1987

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot. *Statemate: A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering Vol 16 No 4, April 1990

[Hof97] U. Hoffmann. *MENTOR at the Web*. Diplomarbeit in Vorbereitung. Universität des Saarlandes. Saarbrücken, 1997

[i-Log94-a] I-Logix Inc. *Statemate User and Reference Volume I*. Burlington, MA, 1994

[i-Log94-b] I-Logix Inc. *Statemate User and Reference Volume II*. Burlington, MA, 1994

[i-Log94-c] I-Logix Inc. *Statemate Prototyper*. Burlington, MA, 1994

[KAGM96] M. Kamath, G. Alonso, R. Günthör, C. Mohan. *Providing High Availability in Very Large Workflow Management Systems*. International Conference on Extending Database Technology, Avignon, 1996.

[Klä97] A. Kläser. *Konzeption und Implementierung eines Workflow-Log-Managers und eines Workflow-History-Managers - Teil 1*. Diplomarbeit in Vorbereitung. Universität des Saarlandes. Saarbrücken, 1997

[Lan92] H. Langendörfer. *Leistungsanalyse von Rechensystemen: Messen, Modellieren, Simulation*. Hanser Studienbücher der Informatik. München, 1992

[Nov94] Novell Inc. *Tuxedo System 5. System Documentation*. 1994

[Ora94] Oracle Inc. *Oracle System Documentation*. 1994

[Pri95] F. Primatesta. *TUXEDO - An open approach to OLTP*. Prentice-Hall. London, 1995

[RS94] M. Rusinkiewicz, A. Sheth. *Specification and Execution of Transactional Workflows*. In: W. Kim (Herausgeber). Modern Database Systems: The Object Model, Interoperability and Beyond. ACM Press, 1994

[Sh96] A. Shet (ed.). *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*. Athens GA, May 1996

[SJ96] A.-W. Scheer, W. Jost. *Geschäftsprozeßmodellierung innerhalb einer Unternehmensarchitektur*. In: [VB96]

[Sti97] K. Stichter. *Konzeption und Implementierung eines Workflow-Log-Managers und eines Workflow-History-Managers - Teil 2*. Diplomarbeit in Vorbereitung. Universität des Saarlandes. Saarbrücken, 1997

[Sun93] Sun Inc. *Sun-RPC. Online Manuals*. San Francisco, 1993

[VB96] G. Vossen (Herausgeber), J. Becker (Herausgeber). *Geschäftsprozeßmodellierung und Workflow-Management, Modelle,Methoden, Werkzeuge*. International Thomson Publishing. Bonn, 1996

[VGH93] G. Vossen, M. Groß-Hardt. Grundlagen der Transaktionsverarbeitung. Addison-Wesley Publishing. Bonn, 1993

[Wä97] H. Wächter. *Fehlertolerantes Workflow-Management. Eine Architektur für die zuverlässige Abwicklung verteilter Geschäftsprozesse*. Dissertation. Universität Stuttgart, 1997

[Weh97] M. Wehrmann. *Konzeption und Implementierung des Kommunikationsmanagers für das Workflow-Management-System MENTOR - Integration und Administration mit dem TP-Monitor Tuxedo*. Diplomarbeit. Universität des Saarlandes. Saarbrücken, 1997

[WMS+95] D. Wodtke, P. Muth, M. Sinnwell, G. Weikum, A. Kotz-Dittrich. *MENTOR: Entwurf einer Workflow-Management-Umgebung basierend auf State- und Activitycharts*. In: 6. Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft". Dresden, März 1996

[Wod96] D. Wodtke. *Modellbildung und Architektur von verteilten Workflow-Management-Systemem*. Dissertation. Universität des Saarlandes. Saarbrücken, 1996

[WW97] D. Wodtke, G. Weikum. *A Formal Foundation For Distributed Workflow Execution Based on State Charts*. In: Proc. International Conference on Database Theory. Delphy, Greece, 1997.

[WWK+97] G. Weikum, D. Wodtke, A. Kotz-Dittrich, P. Muth, J. Weißenfels. *Spezifikation, Verifikation und verteilte Ausführung von Workflows in MENTOR*. In: Informatik - Forschung und Entwicklung. Springer Verlag, 1997

[WWW+97] D. Wodtke, J. Weißenfels, G. Weikum, A. Kotz-Dittrich, P. Muth. *The MENTOR Workbench for Enterprise-Wide Workflow Management*. Proc. of the ACM SIGMOD Int. Conf. on Management of Data. Tuscon, Arizona, 1997

[WWWK96a] D. Wodtke, J. Weißenfels, G. Weikum, A. Kotz-Dittrich. *The MENTOR Project: Steps Towards Enterprise-Wide Workflow Management*. In: Proc. of the 12th IEEE International Conference on Data Engineering. New Orleans, LA, 1996

[WWWK96b] J. Weißenfels, D. Wodtke, G. Weikum, A. Kotz-Dittrich. *The MENTOR Architecture for Enterprise-Wide Workflow Management*. In: [Sh96]

[Zha96] T. C. Zhao. *Forms Library: a Graphical User Interface Toolkit for X*. V0.86. September 1996