Ein Transaktions-Service für tx+YAWL

Studienarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von:	Stefan Muderack
Matrikelnummer:	6201332
geboren am:	20.06.1986 in Rostock
Erstgutachter:	Holger J. Meyer
Betreuer:	Holger J. Meyer
	Sebastian Schick
Abgabedatum:	12. April 2012

Inhaltsverzeichnis

1 Einleitung

YAWL *(Yet Another Workflow Language)* ist eine, auf Workflow-Pattern basierende, Prozessmodellierungsprache. Die theoretische Grundlage von YAWL bilden Petrinetze [vdAtH05]. Es ist OpenSource und wurde unter der GNU Lesser General Public License und der Apache Software License veröffentlicht.

tx+YAWL ist eine YAWL-Erweiterung die transaktionale Workflows ermöglicht. Sie wurde an der Universität Rostock entwickelt und erlaubt den transaktionalen Zugriff auf externe Parameter aus dem Kontrollfluss eines Workcases heraus [SHA11]. Der Zugriff auf die externen Datenquellen soll mit tx+YAWL synchronisiert und konsistent durchgeführt werden. Um eine verbesserte Datensicherheit und Datenpersistenz zu erreichen, können transaktionale und nichttransaktionale Datenquellen in den Workflow integriert werden.

Für den Zugriff auf externe Parameter enthält tx+YAWL das *Data Access Framework (DAF)*. Dieses bietet eine Schnittstelle, die direkt in YAWL integriert ist und die Zugriffe auf externe Parameter an den *Data Gateway* weiterleitet. Über den Data Gateway wird dann die *Data Integration Chain (DIC)* gestartet.

Die DIC besteht aus verschiedenen Teilkomponenten, die in einer Apache Chain organisiert, den eigentlich Zugriff auf die externe Variable realisiert. Nach einem erfolgreichen Zugriff wird der Wert durch die DIC ans YAWL Framework zurückgeliefert und kann im Workflow verwendet werden.

In tx+YAWL wurde unter anderem das Konzept einer transaktionalen Sphäre aufgestellt. Eine transaktionale Sphäre stellt innerhalb eines Workflows einen Bereich dar, in dem transaktionale Prinzipien gelten. Wenn innerhalb einer transaktionalen Sphäre ein Zugriff auf einen externen Parameter durchgeführt wird, soll der Zugriff unter transaktionalen Gesichtspunkten erfolgen. Zugriffe auf externe Parameter werden im Verlauf einer transaktionalen Sphäre nur einmal gelesen und geschrieben.

Das DAF ermöglicht lesende Zugriffe auf den externen Datenspeicher, diese Zugriffe werden durch den Pluginmanager ausgeführt. Im Gegensatz zur Zielsetzung von tx+YAWL werden die Zugriffe jedoch noch nicht mit transaktionalen Grundprinzipien wie der Isolation durchgeführt. Auch das von tx+YAWL geforderte Read- und Writeset wurde noch nicht umgesetzt.

1.1 Ziele

Das Ziel dieser Studienarbeit ist es, die am Lehrstuhl entwickelte YAWL Erweiterung tx+YAWL um einen Transaktionsmanager zu erweitern. Dabei ist es notwendig, diesen Transaktionsmanager in das DAF zu integrieren.

Es soll eine Entscheidung bezüglich der Auswahl des verwendeten Transaktionsystemes getroffen werden. Dafür sollen die Vor- und Nachteile der in Betracht kommenden Systeme untersucht werden. Bereits existierende Transaktionsmanager sollen auf ihre Anwendbarkeit untersucht und bei Eignung ein Transaktionsmanager ausgewählt werden. Sollte kein bereits existierender Transaktionsmanager auf das Anforderungsprofil passen, soll eine eigene Lösung entwickelt werden.

Der Transaktionsmanager soll Zugriffe auf externe Parameter in einer transaktionalen Sphäre gegenüber Zugriffen auf den selben Parameter in anderen Sphären isolieren und dem Nutzer das Gefühl geben, dass er der einzige Nutzer der Datenbasis ist. Des Weiteren soll er das Read- und Writeset aus der Zielsetzungen von tx+YAWL umsetzen und nur noch ein Lese- und Schreibzugriff pro Parameter in einer transaktionalen Sphäre durchführen. Dafür ist es nötig, eine Zwischenspeicherschicht einzuführen, die die Werte der Variablen speichert.

Sämtliche Lese- und Schreibzugriffe, die innerhalb von transaktionalen Sphären auftreten, sollen durch den Transaktionsmanager mit transaktionalen Grundprinzipien wie Isolierbarkeit behandelt und protokolliert werden. Hierbei ist es gleichgültig, ob die Daten aus einer transaktionalen Datenquelle wie einer Datenbank oder einer nicht-transaktionalen Datenquelle wie einer XML Datei bezogen werden.

Zur Realisierung des Read-/Writesets wird in dieser Arbeit ein Transaktionsmanager vorgestellt, der über einen angeschlossenen Zwischenspeicher verfügt, in welchem Werte von Parametern gespeichert werden können und erst am Ende einer transaktionalen Sphäre in den persistenten Speicher geschrieben werden.

Der Transaktionsmanager soll im Verlauf der Arbeit prototypisch implementiert werden.

1.2 Struktur

Die vorliegende Ausarbeitung ist wie folgt aufgebaut. Kapitel 2 erläutert die wichtigsten Grundkonzepte. Die verwendeten Techniken werden in Kapitel 3 vorgestellt. In Kapitel 4 werden die verwendeten Techniken und die in Kapitel 2 betrachteten Grundlagen zu einem Konzept verbunden und präsentiert. Kapitel 5 beschreibt die Implementierung des Prototypens und die dabei

aufgetretenen Probleme. Das nachfolgende Kapitel 6 präsentiert die Zusammenfassung dieser Arbeit und zeigt einen Ausblick auf zukünftige Versionen des Transaktionsmanagers.

2 Grundlagen und Begriffsklärung

Die folgenden Abschnitte enthalten eine Einführung in die grundlegenden verwendeten Technologien und Konzepte dieser Arbeit. Die hier aufgeführten Beschreibungen basieren auf [SH99] und [Wei88].

2.1 ACID

Um die Korrektheit und Konsistenz der Daten in der Datenbank zu gewährleisten, müssen die ACID Prinzipien eingehalten werden. Transaktionen in Datenbanksystemen basieren auf folgenden grundlegenden vier Pfeilern:

Atomicity (Atomarität)
Eine Transaktion ist unteilbar. Sie wird entweder komplett oder gar nicht ausgeführt.

Consistency (Konsistenz)
Wenn der Datenspeicher vor der Transaktion in einem konsistenten Zustand war, so muss er es auch nach der Beendigung der Transaktion sein.

Isolation (Isolation)
Der Benutzer soll das Gefühl haben, dass er zu dem Zeitpunkt der einzige Nutzer des Datenbestandes ist. Die verschiedenen Transaktionen dürfen sich gegenseitig nicht beeinflussen.

Durability (Dauerhaftigkeit)
Nachdem die Transaktion abgeschlossen ist, werden die Daten persistent gespeichert. Die Auswirkungen der Transaktion sind permanent und dürfen nicht verloren gehen.

2.2 Transaktionen

Eine Transaktion ist eine geordnete Folge von Operationen (Aktionen), die zusammen eine logische Einheit bilden. Die Reihenfolge der Operationen der Transaktion darf nicht verändert werden.

Von Transaktionen im Datenbanksystem wird die Einhaltung der oben dargestellten ACID Regeln gefordert, um unerwünschte Seiteneffekte wie Lost-Updates oder Nonrepeatable Reads zu

verhindern. Transaktionen beginnen mit einem BOT (*begin of transaction*) Kommando und enden mit einem EOT (*end of transaction*) Befehl. Eine *vollständige Transaktion* muss mit einem **abort** *oder* **commit** Befehl terminieren. Nach dem Terminalbefehl darf kein weiterer Lese- oder Schreibzugriff erfolgen.

Grundlage dieser Arbeit ist das *Read/Write Modell* aus [SH99]. In diesem Modell besteht eine Datenbank aus Datenobjekten. Diese Objekte werden in *unteilbaren* Schritten gelesen (**read**) und geschrieben (**write**). Kleinbuchstaben mit Indizes sind Bezeichner für Datenobjekte. x_1 und z_i ($i \in$ Natürliche Zahlen) stellen demnach Datenobjekte dar.

In den folgenden Kapiteln wird die Aktion $r_i(x)$ oder $w_i(x)$ einer Transaktion auf einem Datenobjekt x durch ein Indizes markiert. Damit sind die Aktionen der verschiedenen Transaktionen in einem Schedule differenzierbar gestaltet. Eine Transaktion ist somit eine geordnete, endliche Folge von Operationen o_1 der Form $r_i(x)$ oder $w_i(x)$.

Formal kann eine Transaktion wie folgt ausgedrückt werden:

$$T = o_1 o_2 ... o_n \; mit \; o_i \in \{r_i(x), w_i(x)\}$$

Die im Verlaufe dieser Arbeit verwendeten vollständigen Transaktionen enthalten insgesamt 4 Aktionstypen. Das oben eingeführte Aktionsobjekt o wird um **abort** a_i und **commit** c_i ergänzt, sodass eine Transaktion über insgesamt 4 Aktionen verfügt: *read, write, commit, abort.*

In BNF [1] lässt sich eine Transaktion formal wie folgt ausdrücken, dabei wird der **BOT** nicht notiert und die **EOT** ist in der Terminierungsaktion impliziert:

$$T = [r|w]^+[a|c]$$

In dieser Arbeit wird die Transaktion am Beispiel einer Banküberweisung dargestellt:

Beispiel einer Transaktion:

```
1 begin of transaction
2   Kontostand von Konto A lesen
3   10 Euro von Konto A abbuchen
4   geänderten Wert zurückschreiben
5   Kontostand von Konto B lesen
6   10 Euro auf Konto B aufbuchen
```

[1] Backus-Naur-Form

```
7   geänderten Wert zurückschreiben
8 end of transaction
```

Die Transaktion stellt die Überweisung von 10 Euro von Konto A an Konto B dar. Diese Transaktion muss komplett ausgeführt werden. Wenn ein Fehler in der Abarbeitung der Transaktion auftritt müssen alle bisher ausgeführten Operationen rückgängig gemacht werden, um Inkonsistenzen vorzubeugen.

Transaktionen werden in Transaktionssystemen wie Schedulern verarbeitet und zu Schedules zusammengesetzt, die anschließend parallel im Mehrbenutzerbetrieb abgearbeitet werden können. Dabei ist das Ziel des Transaktionssystemes stets einen größtmöglichen Durchsatz bei gleichzeitiger Einhaltung der ACID Kriterien zu erhalten.

2.2.1 Zustand von Transaktionen

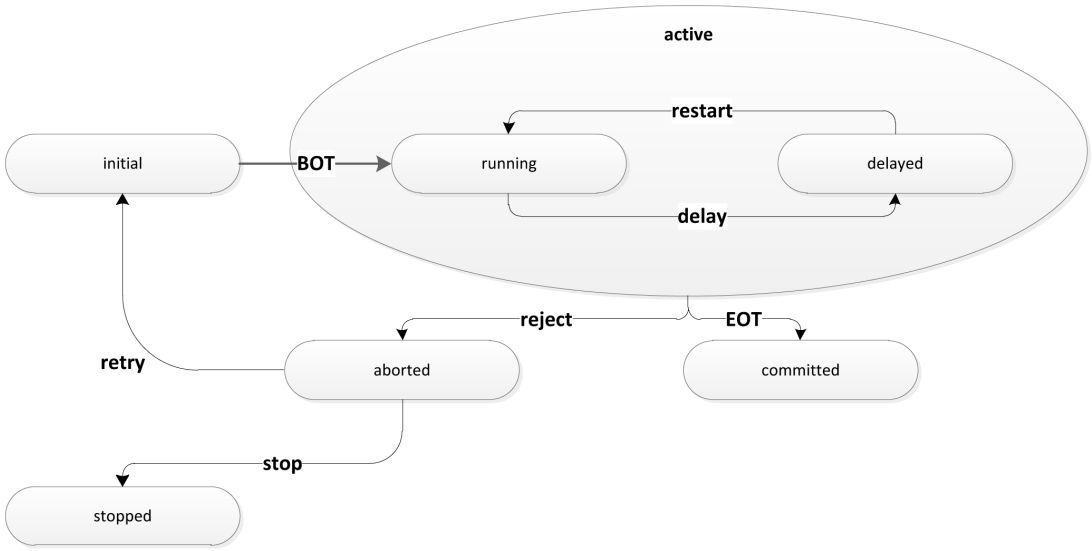

Abbildung 1: Mögliche Zustände einer Transaktion aus [SH99]

Eine Transaktion kann durch den Scheduler (siehe Kapitel 2.4.1) in verschiedene Zustände gesetzt werden. Die verschiedenen Zustände einer Transaktion sind in Abbildung 1 dargestellt. Nach der Initialisierung folgt der implizit durchgeführte Start (*BOT*) einer Transaktion. Eine Transaktion kann erfolgreich beendet (*commit*), zurückgewiesen (*reject*, dies entspricht den *abort* der Transaktion) oder in der Ausführung angehalten werden (*delayed*).

Eine pausierte Transaktion muss durch den Scheduler reaktiviert oder abgebrochen werden. Eine abgebrochene Transaktion wird durch den Scheduler neu gestartet oder komplett gestoppt.

2.3 Schedule

Ein Schedule (oft auch als Historie bezeichnet) ist eine Abarbeitungsreihenfolge der Aktionen von den beteiligten Transaktionen. Ein vollständiger Schedule enthält alle Operationen der involvierten Transaktionen genau einmal. Dabei müssen die Aktionen der Transaktion T_x im Schedule in einer identischen Reihenfolge auftreten; wie in der Transaktion T_x. Zusätzlich muss für jede beteiligte Transaktion ein Terminierungskommando (**commit, abort**) enthalten sein.

Die Operationen der verschiedenen Transaktionen können im Schedule gemischt auftreten und ermöglichen eine effizientere Ausnutzung gegebener Ressourcen, da Operationen, die auf verschiedene Datenbankobjekte zugreifen, parallel bearbeitet werden können.

"Ein Schedule ist ein Präfix eines vollständigen Schedules."[SH99]

Beispiel eines vollständigen Schedules

$$\mathrm{s} = r_1(a)r_2(b)w_2(b)w_1(a)r_1(c)a_2w_1(c)c_1$$

Die Transaktion T_1 terminiert erfolgreich, während die Transaktion T_2 abbricht.

Ein möglicher **Schedule** aus diesem *vollständigen Schedule* ist beispielsweise

$$\mathrm{s} = r_1(a)r_2(b)w_2(b)w_1(a)$$

Ein **serieller** Schedule ist eine spezielle Form eines Schedules. Die involvierten Transaktionen werden aufeinanderfolgend ausgeführt, wobei die Reihenfolge der Ausführung der Transaktionen willkürlich ist. Die Operationen einer Transaktion werden ohne Unterbrechung durch die Operation einer anderen Transaktion ausgeführt. Die Transaktionen werden nacheinander (seriell) abgearbeitet.

Ein serieller Schedule ist aufgrund der unterbrechungsfreien Ausführung der Operationen seiner Transaktion korrekt, aufgrund der Hintereinanderausführung der Aktionen allerdings verhältnismäßig ineffizient, da keinerlei Parallelisierung stattfinden kann.

Ein möglicher serieller Schedule für das obige Beispiel wäre beispielsweise:

$$\mathrm{s} = r_1(a)w_1(a)r_1(c)w_1(c)c_1r_2(b)w_2(b)a_2$$

Für einen vollständigen Schedule mit n Transaktionen existieren folglich $n!$ verschiedene serielle Schedules.

Die ursprüngliche Concurrency-Control-Theorie nennt einen Schedule s *serialisierbar*, wenn es einen äquivalenten seriellen Schedule s' gibt. Der Begriff Äquivalenz eines Schedules kann durch verschiedene Äquivalenzrelationen dargestellt werden. [SH99, Seite 430ff] erläutert die Sichtäquivalenz und die Konfliktäquivalenz.

Erweiterungen und Variationen der Transaktionsmodelle (wie die multiversionen Serialisierbarkeit oder geschachtelte Transaktionen) werden mit seriellen Schedules auf ihre Korrektheit verglichen.

2.4 Transaktionsmanager

Die Aufgabe des Transaktionsmanager, mit seiner Hauptkomponente dem Scheduler, ist es ausführbare Schritte für jede Transaktion zu erzeugen [SH99]. Diese Schritte werden in der, durch die Transaktion vorgegebenen, Reihenfolge ausgeführt.

2.4.1 Scheduler

Der Scheduler ist das zentrale Bauteil des Transaktionsmanagers. Er wandelt einen aus potentiell mehreren Transaktionen bestehenden, nicht zwingenderweise serialisierbaren Eingabeschedule in einen serialisierbaren Ausgabeschedule um. Der Scheduler muss sicherstellen, dass der Ausgabeschedule **serialisierbar** und **fehlersicher** ist. Ein Schedule mit abgebrochenen Transaktionen muss sich so verhalten wie ein Schedule ohne abgebrochene Transaktionen.

Der Ausgabeschedule muss äquivalent zum Eingabeschedule sein, alle Aktionen des Eingabeschedules müssen im Ausgabeschedule genau einmal enthalten sein. Derweil muss der Scheduler die interne Reihenfolge der involvierten Transaktionen beibehalten. Der Scheduler muss Konflikte im Eingabeschedule erkennen und lösen, dabei hat er verschiedene Möglichkeiten einen serialisierbaren Ausgabeschedule zu erzeugen. Er kann eine Operation des Schedules sofort ausführen, sie zurückstellen oder die zugehörige Transaktion abbrechen und neu starten. Wenn der Scheduler eine Transaktion abbricht, wird ein *abort* der Transaktion durchgeführt und versucht alle von ihr angestoßenen Veränderungen rückgängig zu machen, bevor die Transaktion zu einem späteren Zeitpunkt wieder neu gestartet wird.

Abbildung 2 zeigt einen Scheduler, der aus den beteiligten Transaktionen a, b und z einen Ausgabeschedule erzeugt und dabei ihre interne Reihenfolge beibehält.

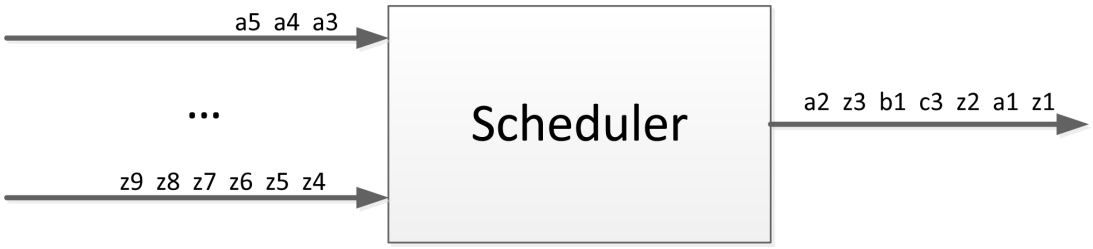

Abbildung 2: schematische Abbildung eines Schedulers

Scheduler haben verschiedene Verfahren, um Konflikte im Mehrbenutzerbetrieb zu vermeiden. Es wird grob zwischen optimistischen und pessimistischen Schedulern unterschieden, zusätzlich gibt es noch andere speziellere Scheduler wie die Mehrversionen-Scheduler.

Optimistische Scheduler erzielen eine hohe Parallelität im Mehrbenutzerbetrieb und somit eine hohe Transaktionsdurchsatzrate. Sie erreichen dies, indem sie Konflikte zulassen und eine Transaktion erst dann abbrechen, wenn sie den Konflikt, bei Commit der Transaktion, nicht auflösen konnten. Der Rücksetzaufwand der Transaktion erhöht sich, weil die Transaktion bereits ihre Berechnungen durchgeführt hat.

Pessimistische Scheduler verfolgen einen anderen konservativeren Ansatz. Sie versuchen Konflikte zu vermeiden und das Risiko eines Transaktionsabbruches zu minimieren. Dafür nehmen sie Verzögerungen im Transaktionsverlauf und eine Senkung der Parallelität in Kauf. Sie erreichen eine deutlich geringere Durchsatzrate, da sie im Worst-Case Szenario nur noch eine Transaktion ausführen.

Mehrversionen-Scheduler können zu jedem geschriebenen Datenobjekt mehrere Versionen speichern. Bei einer **read**-Operation müssen die Scheduler die korrespondierende Version des Datenelementes zuweisen. Mehrversionen-Protokolle sind beispielweise MVTO und MV2PL [WV02, Seite 203 ff.].

Scheduler besitzen verschiedene Sperrmodelle, um Datenbankobjekte (Seiten, Tabellen, Datensatz) für andere Transaktionen zu sperren. Diese Sperre kann verschiedene Ausprägungen haben. So kann die Sperre eine Schreib- oder eine Lesesperre auf dem Datenbankobjekt x sein. Die Sperre wird vom Scheduler bei Beginn der Ausführung gesetzt und spätestens vor dem **commit** der Transaktion wieder entfernt. Sollte vor Beginn der Ausführung bereits eine Sperre auf dem Objekt vorhanden sein, muss natürlich gewartet werden, bis die alte Sperre aufgehoben ist. Ein Scheduler hat verschiedene Sperrprotokolle und Sperrmodelle zur Auswahl, um Probleme wie LiveLocks oder Verklemmungen zu verhindern.

Nähere Informationen zu diesen Punkten findet der Leser in [SH99, Seite 486ff].

2.5 Serialisierbarkeit

Im Mehrbenutzerbetrieb müssen die Transaktionen der Benutzer teilweise auf die gleichen Daten zurückgreifen. Die Transaktionen konkurrieren um die Lese- und Schreibzugriffe auf diese Datensätze. Während eine Transaktion X auf einen Datensatz schreibend zugreift, kann eine Transaktion Y, wenn die Sperrprotokolle eingehalten werden, auf diesen Datensatz nicht lesend oder schreibend zugreifen, da ein Zugriff auf diese Ressource zu Seiteneffekten wie einem Dirty-Read führen könnte. Um solche Seiteneffekte und Fehler zu verhindern könnte die Transaktionsverwaltung die Transaktionen seriell ausführen.

Eine serielle Ausführung ist weitestgehend fehlerfrei, ist aber aus performancetechnischer Sicht nicht erstrebenswert, da der Durchsatz durch die Hintereinanderausführung der Transaktionen erheblich sinken würde. Außerdem hat das DBMS[2] keine Möglichkeit den Durchsatz durch eine parallele Abarbeitung der Transaktionen zu erhöhen.

Um einen maximalen Durchsatz zu erhalten, führt das DBMS die Transaktionen verschränkt aus. Das DBMS hat die Aufgabe, die parallel ablaufenden Transaktionen zu synchronisieren und zu isolieren. Ebenso soll sie fehlerhafte Abläufe und Seiteneffekte zwischen Transaktionen verhindern und eine maximale Parallelität bei gleichzeitiger Äquivalenz zu einem möglichen seriellen Schedule erreichen. Die Ergebnisse der verschränkten Ausführungen sollen äquivalent zu einer der möglichen seriellen Ausführungen sein.

Die Serialisierbarkeit ist eine Grundlage für die verschränkte Ausführung von Transaktionen im Mehrbenutzerbetrieb. Um das Prinzip der Serialisierbarkeit zu verdeutlichen, werden hier nun drei verschiedene Ausführungsvarianten zweier Transaktionen betrachtet.

Transaktion 1: **read** A; A = A - 5; **write** A; **read** B; B = B + 10; **write** B;
Transaktion 2: **read** B; B = B - 5; **write** B; **read** A; A = A + 10; **write** A;

In Tabelle 1 werden drei verschiedene mögliche Ausführungsvarianten dargestellt. Die erste Spalte enthält eine serielle Ausführung der beiden Transaktionen T1 und T2, wobei deutlich sein dürfte, dass die Ergebnisse der Ausführung von T1 und T2 äquivalent sind, egal ob T1 vor T2 ausgeführt wird. Deshalb wird auf eine weitere Darstellung der Ausführung von T2 vor T1 verzichtet. Die beiden folgenden Spalten enthalten unterschiedlich verschränkte Ausführung der

[2] Database Management System

Ausführung 1		Ausführung 2		Ausführung 3	
T1	T2	T1	T2	T1	T2
read A		**read** A		**read** A	
A = A - 5			**read** B	A = A - 5	
write A		A = A - 5		**write** A	
read B			B = B - 5	**read** B	
B = B + 10		**write** A			**read** B
write B			**write** B		B = B - 5
	read B	**read** B		**read** A	**write** B
	B = B - 5		**read** A		**read** A
	write B	B = B + 10			A = A + 10
	read A		A = A + 10	B = B + 10	
	A = A + 10	**write** B		**write** B	
	write A		**write** A		**write** A

Tabelle 1: verschiedene Ausführungmöglichkeiten von zwei Transaktionen [SH99]

beteiligten Transaktionen.

	A	B	**Summe**
Ausgangswert	20	20	40
nach Ausführung 1	25	25	50
nach Ausführung 2	25	25	50
nach Ausführung 3	25	30	55

Tabelle 2: Ergebnisse der unterschiedlichen Ausführungsreihenfolgen der Transaktionen

Tabelle 2 zeigt die Werte der Variablen A und B und eine Summenspalte nach der Ausführung der verschiedenen Schedules. Während die Ausführungen 1 und 2 die gleichen Werte aufzeigen, weicht die Ausführung 3 bei dem Wert der Variable B ab. Sämtliche Änderungen der Transaktion T2 an der Variable B werden durch Transaktion T1 überschrieben, da die Transaktion T1 den Wert von B vor der Ausführung von T2 liest und mit diesem Wert arbeitet.

Neben der *Konfliktserialisierbarkeit* gibt es die *Sichtenserialisierbarkeit* und die auf Mehrversionen-Schedules spezialisierte *1-Serialisierbarkeit*.

Zur Durchführung der Serialisierung wurden Sperrprotokolle eingeführt, die Objekte sperren und Lese- und Schreibzugriffe so reglementieren, dass eine fehlerhafte Ausführung wie Ausführung 3 in Tabelle 1 nicht mehr möglich ist. Prominente Sperrprotokolle sind das 2-Phasen Sperrprotokoll und das Hierarchische Sperren.

2.6 geschachtelte Transaktionen

Der bisher in dieser Arbeit verwendete Begriff der Transaktion implizierte ein flaches Modell ohne Hierarchien. Die geschachtelten Transaktionen (engl. nested transactions) bestehen aus Vater-/Kindtransaktionen, die hierarchisch organisiert werden. Eine ausgezeichnete Transaktion an der Spitze der Hierarchie wird als Wurzeltransaktion bezeichnet. Jede Kindtransaktion in der Hierarchie kann eine beliebige Anzahl an Subtransaktionen enthalten und somit auch als Vatertransaktion fungieren. Jede Subtransaktion kann wieder Subtransaktionen enthalten.

Mithilfe von geschachtelten Transaktionen können lange oder komplexe Aufgaben (langlaufende Transaktionen) in verschiedene kleinere, parallel ablaufende Teilaufgaben (Subtransaktionen) zerlegt und parallel abgearbeitet werden.

Ein mögliches Anwendungsszenario von geschachtelten Transaktionen sind CAD Programme oder lang laufende Batchprozesse, bei denen eine *Alles oder Nichts*-Ausführung zu einem erhöhten Einsatz an Arbeitszeit führen könnte. Wenn ein laufender Batch-Prozess (wie zum Beispiel Zinsberechnung) abgebrochen wird, gehen sämtliche durch ihn berechnete Ergebnisse verloren. Der gesamte langlaufende Batchprozess müsste wiederholt werden.

Durch die Einführen einer Hierarchie in das vormals flache Modell muss überprüft werden, ob die geschachtelten Transaktionen noch den ACID-Prinzipien genügen:

- Die **Atomarität** einer geschachtelten Transaktion kann nur gewährleistet werden, wenn im Falle des *aborts* einer Transaktion alle Transaktionen der Hierarchie gemeinsam abgebrochen werden oder alle Transaktionen gemeinsam durch einen *commit* beendet werden. Durch das Einführen von kompensierenden Transaktionen kann das Scheitern einer (Sub)-Transaktion durch eine kompensierende Transaktion ausgeglichen werden. Der gesamte Transaktionsbaum muss nicht abgebrochen werden. Die Nutzung von kompensierenden Transaktionen verstößt jedoch gegen das Atomaritätsprinzip und entspricht somit nicht den ACID-Prinzipien.
- Um die **Isolation** in geschachtelten Transaktionen zu erhalten, müssen die Ergebnisse der Subtransaktionen an ihre jeweilige Vatertransaktion weitergeleitet werden. Wenn die Subtransaktionen ihre Ergebnisse bei *commit* auch an andere nebenläufige Transaktionen freigibt und sie nicht nur exklusiv an ihre Vatertransaktion vererbt, spricht man von Isolation auf Subtransaktionsebene.

Geschachtelte Transaktionen können in *Offen geschachtelte Transaktionen* und *Geschlossen geschachtelte Transaktionen* unterschieden werden, wobei die Begriffe 'offen' und 'geschlossen' die

Sichtbarkeit der Ergebnisse der Subtransaktionen beschreibt.

2.6.1 Offen geschachtelte Transaktionen

In Offen geschachtelten Transaktionen sind die Ergebnisse nach Commit der Subtransaktion für andere nebenläufige Transaktionen sichtbar. Durch die Freigabe der Ergebnisse von Subtransaktionen für andere (Kind-)Transaktionen erreichen Offen geschachtelte Transaktionen eine stark erhöhte Möglichkeit der Parallelität zwischen nebenläufigen Transaktionen. Durch die frühe Freigabe der Ergebnisse wird das Isolationsprinzip der geschachtelten Transaktionen verletzt. Die **Isolation** wird nur noch auf der Ebene der Subtransaktionen erreicht.

Die Einhaltung der Atomarität in Offen geschachtelten Transaktionen soll nun Anhand der Transaktionen T_V und T_S betrachtet werden. Wobei T_S die Subtransaktion und T_V die Vatertransaktion ist.

Die Vatertransaktion T_V kann nur nach dem **commit** all ihrer Kindtransaktionen T_S selbst **comitten**, während ein **abort** von T_V einen **abort** von T_S nach sich zieht.

Die Reaktion auf einen **abort** der Kindtransaktion kann durch 4 verschiedene Schritte skizziert werden:

1. Auslösen eines kaskadierenden Abbruches (**abort**) der Vatertransaktion und Weitergabe des **aborts** an die Vatertransaktion der Vatertransaktion

2. Neustart (**retry**) von T_S.

3. Starten einer Alternativtransaktion (engl. *contingency transaction*) (**try**) für T_S.

4. Ignorieren (**ignore**) des **aborts** wenn T_S für die Vatertransaktion eine nicht-vitale Subtransaktion darstellt.

Offen geschachtelte Transaktionen können *vitale* und *nicht-vitale* Subtransaktionen besitzen. Damit ergeben sich verschiedene Möglichkeiten zur Behandlung eines *abort* der Kindtransaktion T_S. Die Kindtransaktion kann im Falle der *nicht-Vitalität* ignoriert werden und im Falle der Vitalität neu gestartet (*retry*) oder durch eine *Alternativtransaktion* ausgeglichen werden. Die Atomarität wird jedoch in all diesen Optionen verletzt.

Die Atomarität einer Offen geschachtelten Transaktion kann bei **abort** der Kindtransaktion nur durch einen **retry** oder eines **aborts** der Vatertransaktion gewahrt werden.

2.6.2 Geschlossen geschachtelte Transaktionen

Geschlossen geschachtelte Transaktionen erfüllen das *Isolations-Prinzip* durch eine strikte Sperrenweitergabe. Kindtransaktionen vererben ihre Sperren an die Vatertransaktionen. Die Ergebnisse der Kindtransaktion werden der Vatertransaktion und den *Geschwistertransaktionen* zugänglich gemacht. Nebenläufige Transaktionen können die Ergebnisse aber erst mit dem *commit* der Wurzeltransaktion lesen. Aus dieser Geschlossenheit entstand der Name 'Geschlossen geschachtelte Transaktionen'.

Die Transaktionen der Geschlossen geschachtelten Transaktionen sind aufgrund der Einhaltung folgender Kriterien atomar:

1. Der Abbruch der Vatertransaktion führt zum Abbruch aller Kindtransaktionen.
2. Der Abbruch einer Kindtransaktion führt zum Abbruch der Vatertransaktion.
3. Eine Transaktion der Hierarchie kann nur erfolgreich abschließen, wenn alle ihre Subtransaktionen erfolgreich sind.

Der Abbruch der Subtransaktion T_S führt zum Abbruch seiner Vatertransaktion T_V. Dies wiederum bedeutet, dass alle Kindtransaktionen von T_V abgebrochen werden. Dieser Abbruch wird kaskadierend bis zum Wurzelknoten fortgesetzt.

2.7 Multiversionen-Serialisierbarkeit

2.7.1 Einleitung und Ziele

Die Multiversionen-Serialisierbarkeit (engl. *MultiVersion Concurrency-Control (MVCC)*) verfolgt einen anderen Ansatz zur klassischen Concurrency-Control Theorie. Die klassische CC-Theorie basiert auf der Annahme, dass von einem Objekt zu einem bestimmten Zeitpunkt genau eine Version existiert. Auf dieser Objektversion werden sämtliche Lese- und Schreibvorgänge ausgeführt. Die MVCC Theorie will die Parallelität und den Durchsatz der Abarbeitung der Transaktion erhöhen, indem sie, statt Datenbankobjekte zu überschreiben, neue Versionen dieser Objekte anlegt.

In gewöhnlichen Transaktionen müssen Aktionen vor ihrem Zugriff auf Objekten Lese- und Schreibsperren setzen. Wenn eine Aktion A_1 auf das Objekt x schreibend zugreifen will, setzt es eine Scheibsperre. Alle nachfolgenden Aktionen oder parallel laufenden Transaktionen, die x lesen wollen, müssen warten bis A_1 die Schreibsperre aufgelöst hat.

Die Folge aus solchen Sperren ist beispielsweise, dass der Transaktionsdurchsatz des Transaktionsmanagers sinkt. Eine andere Konsequenz können sogenannte Verklemmungen sein. Zwei Transaktionen (T_1 und T_2) benötigen beispielsweise einen exklusiven Zugriff auf Objekte (Datensatz, Zeile, Tabelle), welche die jeweils andere Transaktion hält. So kann T_1 auf die Freigabe eines Objektes durch T_2 warten. Sollte T_2 allerdings gleichzeitig auf die Freigabe eines Objektes durch T_1 warten, entsteht ein Deadlock. Dieser Deadlock kann nur noch durch den Abbruch einer Transaktion und anschließendem Neustart gelöst werden.

In MVCC werden die Objekte während der **write**-Operation überschrieben oder initial angelegt. Sie werden zusätzlich zu den alten Objekten hinterlegt und je nach Implementierung durch einen Timestamp oder einer ansteigenden Transaktionsnummer gekennzeichnet.

Die **read**-Operation der Transaktionen kann aus den verschiedenen Revisionen des Objektes auswählen und ist nicht an eventuelle Lese- oder Schreibsperren gebunden. Zu jedem Zeitpunkt steht eine lesbare Version eines Objektes zur Verfügung. Damit erhöht MVCC die Parallelität einer Verarbeitung. Die Auswahl der Version kann unter verschiedenen Gesichtspunkten wie der Datenbankkonsistenz erfolgen.

Das Ziel von MVCC ist es, nur solche Mehrversionen-Schedules auszuführen, die äquivalent zu einer seriellen (Ein-Versionen-)Ausführung sind. MVCC Transaktionen haben gegenüber normalen Transaktionen den Vorteil, dass sie stellenweise nicht-serialisierbare Schedules durch multiple Versionen von Objekten so serialisieren können, dass die vormals nicht-serialisierbaren Schedules durch die Einführung verschiedener Versionen serialisiert werden.

Durch das erstmalige Rückschreiben der Werte am Ende einer Transaktion ermöglicht MVCC auch eine sehr simple Implementierung der Rücksetzbarkeit. Sollte eine Aktion nicht erfolgreich abschließen, wird die gesamte Transaktion zurückgesetzt. Da im Fall von MVCC aber noch keine Daten geschrieben wurden, muss das System keine Ausgleichstransaktionen starten.

2.7.2 Beispiele

Um einen kurzen Einblick in die Funktionsweise der MVCC-Theorie zu geben, folgen hier zwei Beispiel (ohne und mit MVCC), welche die Funktionsweise und Prinzipien von MVCC darstellen.

Beispiel 1 enthält einen Schedule der an-sich nicht serialisierbar ist, durch den Einsatz von MV-Concurrency-Control, in Beispiel 2 mit MVCC jedoch ausgeführt werden kann.

Beispiel 1. $t = r_1(x)r_2(x)w_2(y)r_1(y)w_4(z)w_2(u)$

Diese Transaktion ist aufgrund eines zirkulären Konfliktes zwischen t_2 und t_1 ***nicht Konflikt-Serialisierbar****, t_2 schreibt den Wert von x bevor t_1 ihn gelesen hat. Damit existiert eine zirkuläre Abhängigkeit zwischen den beiden Transaktionen.*

Durch die Einführung mehrerer Versionen der beteiligten Objekte kann der Schedule konfliktserialisierbar gemacht werden. Der zirkuläre Konflikt zwischen den Transaktionen wird aufgebrochen.

Im folgenden Beispiel 2 steht der Index neben der Variable x für die benutzte Versionsnummer. So würde $r_1(x_1)$ bedeuten, dass die Transaktion die Version 1 der Variable x verwendet.

Beispiel 2. $t = r_1(x_1)r_2(x_1)w_2(y_2)r_1(y_1)w_4(z_1)w_2(u_1)$

Der zirkuläre Konflikt zwischen den beiden Transaktion kann in diesem Schedule durch die Einführung verschiedener Versionen (MVCC) gelöst werden. Der Schreibzugriff von t_2 legt eine neue Version mit der Kennnummer 2 an. Der lesende Zugriff von t_1 greift anschließend auf die ältere Version von y zurück.

Der oben dargestellte MultiVersionen-Schedule ist äquivalent zu dem folgenden konfliktfreien 1-Versionen Schedule:

$$\text{t} = r_1(\text{x})r_2(\text{x})r_1(\text{y})w_2(\text{y})w_4(\text{z})w_2(\text{u})$$

2.7.3 Limitierung der Versionen

MultiVersion-Concurrency-Control ermöglicht eine Steigerung der Performance durch die Anlage von verschiedenen Versionen eines Objektes. [WV02] beschreibt einige Probleme, die durch die verschiedenen Versionen entstehen. Der Gewinn beim Transaktionsdurchsatz oder der gesteigerten Parallelität kann durch den hohen Verwaltungsaufwand und den begrenzten Speicherplatz begrenzt oder sogar verhindert werden.

Aufgrund dieser Limitationen hält [WV02] es für sinnvoll, eine maximale Anzahl von Versionen eines Objektes einzuführen. Zu einem Zeitpunkt P würden von jedem verfügbaren Objekt maximal *n-Versionen* existieren.

Nach dem Einführen einer Versionen-Obergrenze untersucht [WV02] ob eine Transaktion die MultiVersionen-Serialisierbar ist, bei einer Beschränkung der Anzahl der Versionen diese Eigen-

schaft verliert. [WV02] kommt dabei zu folgendem Ergebnis:

Einige MultiVersionen-Serialisierbare Schedules verlieren diese Eigenschaft bei einer Limitierung der verfügbaren Versionen, folglich sind diese Schedules nicht *n-Versionen-Serialisierbar*.

2.7.4 Probleme und Fazit

MVCC erlaubt durch die Versionierung der Objekte eine höhere Parallelität und einen höheren Transaktionsdurchsatz. Die erhöhte Parallelität wird allerdings mit dem zusätzlichen Aufwand durch die Haltung der verschiedenen Versionen erkauft. Durch das Anlegen von multiplen Versionen eines Objektes wird mehr Speicherplatz verbraucht. Dieser Nachteil kann mit *Garbage Collection*, das heißt einem Mechanismus zum Löschen nicht mehr benötigter oder veralteter Versionen teilweise behoben werden. Gleichzeitig steigt der Verwaltungsaufwand durch die Haltung sowie Verwaltung der Versionen und dem Ausführen der Garbage Collection.

MultiVersion Concurrency-Control ist vor allem in Applikationen, die viele lesende Zugriffe haben, sehr effektiv.

Aufgrund der Struktur und dem Aufbau des hier vorgestellenten Transaktionsmanagers wird eine Nutzung von multiplen Variablenversionen angestrebt.

2.8 Workflow

Ein Workflow ist ein automatischer, reglementierter Austausch von Infomationen, Dokumenten und Aufgaben zwischen den Teilnehmern [Hol95, Seite 6]. Jeder Workflow hat einen definierten Start- und Endpunkt. Die Bestandteile eines Workflows folgen einer vordefinierten Ablaufreihenfolge und repräsentieren einen zukünftigen oder bereits existierenden Arbeitsablauf.

Workflows werden oft synonym mit der Geschäftsprozess(Re-)modellierung genannt. Geschäftsprozessmodellierung beschäftigt sich mit der Analyse, der Modellierung und Definition von Geschäftsprozessen, welche meistens in einer grafischen Weise dargestellt werden [Hol95, Seite 6].

Die Darstellung eines Workflows erfolgt durch eine UML ähnliche Repräsentation. Sie besteht aus Eingabe- und Ausgabeparametern und den definierten Aktionen respektiver Algorithmen. Abbildung 3 zeigt einen Workflow zur Essenbestellung.

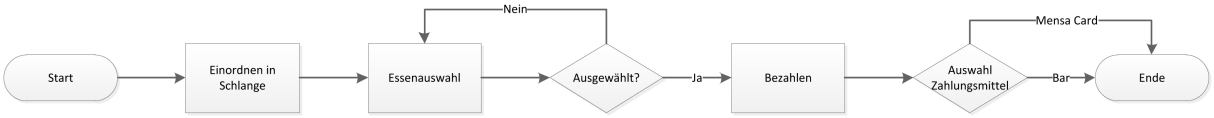

Abbildung 3: schematische Abbildung eines Workflows einer Essenbestellung in der Mensa

2.9 YAWL

YAWL (*Yet another workflow language*) ist eine auf Workflow-Pattern basierende Prozessmodellierungssprache. Das theoretische Fundament von YAWL bilden Petri Netze. Es erweitert die Petrinetze um Funktionalitäten, die die Modellierung von komplexen Workflows ermöglichen [vdAtH05]. YAWL kann durch die Erweiterungen von Petrinetzen, Patterns nutzen, die sich mit dem Abbruch, der Synchronisierung sowie der Ausführung multipler Instanzen einer Aufgabe beschäftigen [HAAR09].

Nach der Definition der YAWL Sprache begannen die Arbeiten an der Implementierung eines Unterstützugsframeworks. Das Acronym wurde Synonym für die Sprache YAWL und dem Framework [HAAR09], welches heute aus dem Editor, der Engine und dem Handler zur Ausführung der Workitems besteht. YAWL und das Framework sind OpenSource-Software und stehen unter der LGPL Lizenz.

Eine Workflow-Spezifikation in YAWL ist eine Hierarchie aus unteilbaren und zusammengesetzten Aufgaben (welche wiederrum aus Unteraufgaben bestehen können). Jede dieser Aufgaben kann multiple Instanzen mit einer Ober- und Untergrenze haben [vdAtH05].

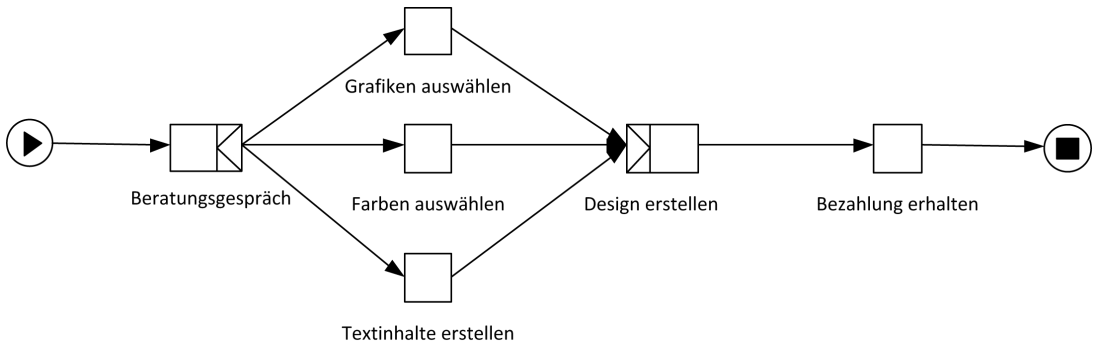

Abbildung 4: schematische Abbildung eines YAWL-Workflows einer Homepageerstellung

Durch einige Funktionen gewinnt YAWL auf dem Gebiet der Business Process Manager (BPM) eine Sonderstellung. So besitzt YAWL beispielsweise ein *formales Fundament*. Die Syntax und

Semantik von YAWL sind präzise definiert. Damit ist in YAWL die Verbindung zwischen der Sprache und der formalen Theorie definiert und verbindlich [HAAR09]. Bei der Entwicklung von YAWL hatte die intensive Unterstützung von Workflow-Pattern eine hohe Priorität, so dass nun eine große Anzahl von Workflow-Pattern unterstützt werden [HAAR09].

Durch die offene Natur von YAWL kann es in verschiedenen Projekten eingesetzt und individuell angepasst werden. Es kann um bestimmte Funktionaliäten wie die Unterstützung von verschachtelten Transaktionen erweitert werden [SHA11]. Das in dieser Arbeit verwendete *tx+YAWL* ist eine YAWL-Erweiterungen.

3 Techniken

Nachdem im vorangegangenen Kapitel die grundlegenden Konzepte, die in dieser Arbeit verwendet werden sollen, vorgestellt wurden, soll dieses Kapitel einen Blick auf die verwendeten Technologien, Paradigmen und Software werfen und diese in einen Kontext setzen.
Ziel dieser Studienarbeit ist das Vergleichen verschiedener Transaktionsmanager und die Überprüfung ob diese für das verwendete Szenario praktikabel sind. Hierfür werden in diesem Kapitel die Transaktionsmanager kurz vorgestellt und mit einem vom Autor aufgestellten und beschriebenen Forderungskatalog verglichen.

3.1 tx+YAWL

tx+YAWL ist eine an der Universität Rostock entwickelte Erweiterung des YAWL Frameworks. Es erweitert YAWL um den Zugriff auf externe Datenquellen direkt aus dem Kontrollfluss heraus und kombiniert dies mit der Einführung von transaktionalen Eigenschaften. Auch die Unterstützung der Modellierung von geschachtelten Transaktionen und MVCC wurde mit tx+YAWL ermöglicht [SHA11]. Andere ähnliche Projekte ermöglichen langlaufende Transaktionen, die auch auf verschiedene Webservices zugreifen, allerdings ohne dabei einen direkten Zugriff auf die Kontrolldaten aus den Webservices oder den Tasks zu gewährleisten.

3.1.1 Aufbau

tx+YAWL besteht aus 4 Schichten, die aufeinander aufbauend die Funktionalität bereitstellen.

Layer 0 - Basic data access Der Layer 0 ist verantwortlich für den Zugriff auf die externen Daten. Dieser Zugriff wird über die Plugin-Architektur realisiert. Der Layer 0 bietet dabei analog zu Datenbanktransaktionen vier verschiedene Grundoperationen. So kann eine bestimmte Variablenversion $T_i.r\{x\}$ gelesen oder $T_i.w\{x\}$ geschrieben werden. Neben

den Lese- und Schreiboperationen gibt es noch Operationen, um die Transaktion erfolgreich zu beenden $T_i.c()$ und die Transaktion abzubrechen $T_i.a()$. Das Verhalten von diesen Operationen ist abhängig von der transaktionalen Sphäre T_i und wird durch die darüber liegenden Schichten kontrolliert.

Layer 1 - Workflow tasks Im Layer 1 wird das Konzept, das Tasks als Grundbausteine von Transaktionen fungieren, beschrieben. Tasks können nicht direkt auf externe Datenquellen zugreifen, sondern müssen dafür ihre Ein- und Ausgabeparameter nutzen. Die externen Variablen werden auf die Parameter gemappt. Parameter werden durch ein eigenes $XSD_{Parameter}$ Schema beschrieben. Der Layer 1 ist verantwortlich für die Assoziierung der Parameter der atomaren oder komplexen Tasks mit den Lese- und Schreiboperationen, die im darunterliegenden Layer 0 durchgeführt werden.

Es können Lese- und Schreibrichtlinien hinterlegt werden, die kontrollieren, ob eine Variable beispielsweise bei jedem lesenden Zugriff aus dem externen Speicher erneut gelesen werden muss oder ob sie vor externen Zugriff geschützt aufbewahrt und bei einem lesenden Zugriff aus einem Cache erneut gelesen wird. In transaktionalen Sphären wird der Ursprungswert nur einmalig gelesen, jeder weitere Lesevorgang wird nicht mehr aus dem externen und vor Veränderungen ungeschützten Datenspeicher vorgenommen.

Gleiches gilt für den schreibenden Zugriff. Dort können die Richtlinien kontrollieren, ob eine geänderte Variable sofort oder erst später in den externen Speicher zurückgeschrieben werden muss. Auf dieser Ebene werden die lokalen Integritätskriterien pro Variable definiert und überprüft. Sollte gegen ein Integritätskriterium verstoßen werden, findet eine Ausnahmebehandlung statt. Diese Behandlung kann beispielsweise ein RollBack oder eine kompensierende Transaktion sein.

Layer 2 - Control flow and transactional spheres Die Ebene 2 beschreibt die Tasks zusammen mit den Kontrollflusspatterns, welche die Bausteine von (Sub-)Transaktionen sind und die Kontrollflussperspektive der Workflowbeschreibung enthalten. In dieser Ebene werden außerdem die durch tx+YAWL unterstützten Transaktionsarten beschrieben. Gewöhnliche Transaktionen sind eine geordnete Folge von Tasks. Mit dieser Art von Transaktion kann man nicht sämtliche in einem Workflowsystem zu betrachtenden Situation abdecken. Tx+YAWL unterstützt zusätzlich noch andere Transaktionen wie geschachtelte-, nichtvitale-, kompensierende-, read-only- Transaktionen [SHA11, Seite 7 ff].

Layer 3 - Case level and global transaction Die Ebene 3 ist die Ebene des gesamten Workflows und den damit assoziierten globalen Transaktionen. Sollte eine globale Transaktion

abbrechen, wird das Exceptionhandling auf Workflowebene ausgeführt und kann damit beispielsweise den Abbruch des Workflowprozesses bedeuten. Ein Workflow kann mehrere voneinander unabhängige transaktionale Sphären enthalten, die sich aber dennoch gegenseitig beeinflussen können. So würde beispielsweise der Abbruch einer globalen transaktionalen Sphäre konsequenterweise den Abbruch aller anderen Sphären bedeuten.

3.1.2 Transaktionale Sphären

Regionen des Workflows, welche transaktionalen Eigenschaften umsetzen, werden in tx+YAWL *transaktionale Sphären* genannt. Diese Sphären können aus einem einzigen simplen Task, einer Sequenz von Tasks oder aus zusammengesetzten Tasks bestehen. Die transaktionalen Sphären bekamen ihren Namen, weil sie im Kontrollfluss Sphären erschaffen in denen transaktionale Eigenschaften gelten. Transaktionale Sphären basieren auf dem Konzept der Offen geschachtelten Transaktionen. Sie sind in tx+YAWL auf der Ebene 2 angeordnet.

Transaktionale Sphären müssen unter anderem folgende definierte Parameter besitzen:

Transaktionstyp Mögliche Transaktionstypen sind: *nicht-transaktional*, *read-only-*, *nicht-vital-*, *kompensierende-* und *Kontingenttransaktionen.*

Lese- und Schreibset Ein Lese- und Schreibset $(readset(T_i), writeset(T_i))$ pro transaktionaler Sphäre.

Integritätskriterien der externen Variablen.

3.1.3 Bausteine

Ein tx+YAWL Workflow besteht aus 12 primitiven Objekten, die auf dem Workflowframework YAWL aufbauen und es stellenweise erweitern. Ein tx+YAWL Netz startet mit einem *Startbaustein* und endet mit einem *Stoppbaustein.* Um Daten über das DAF zu laden, werden *externe Inputs* utilisiert. Diese laden externe Datenobjekte in die Netzvariablen. Dafür müssen die Datenobjekte speziell gekennzeichnet sein und beispielsweise das Plugin und den Speicherort enthalten. Analog dazu funktioniert das Rückschreiben der Netzvariablen in den externen Datenspeicher über den *externen Output.*

In tx+YAWL Workflows können *transaktionale Sphären* definiert werden. In diesen Spähren findet durch den Transaktionsmanager gesteuert, ein transaktionales Verhalten statt. Eine transaktionale Sphäre kann bis auf den Start- und Stoppbaustein alle Bausteine von tx+YAWL enthalten.

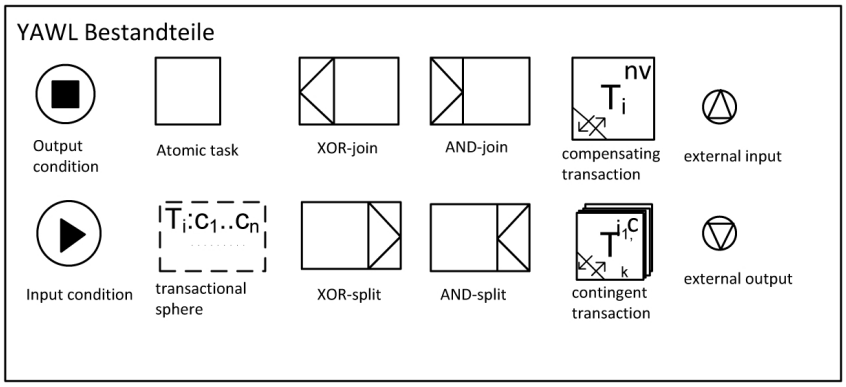

Abbildung 5: Darstellung der Elemente von tx+YAWL aus [SHA11]

Wie YAWL enthält auch tx+YAWL *atomare Task*. Diese sind äquivalent zur Aktion einer Transaktion. Sie führen eine atomare, unteilbare Aufgabe aus. Neben den atomaren Tasks gibt es noch weitere Taskkomponenten wie die *kompensierenden* und *Kontingenttransaktionen*. Kontingenttransaktionen vereinen viele Transaktionen, von denen eine Transaktion erfolgreich abschließen muss. Eine kompensierende Transaktion wird ausgeführt, wenn die vorhergehende Aktion nicht erfolgreich terminierte.
Vor den möglichen Operationen können XOR- und AND-Joins zwei vorhergehende Berechnungspfade zusammenführen. Nach den Operationen kann ein XOR- und AND-Split eine Aufteilung auf zwei parallele Berechnungspfade bereitstellen.

3.1.4 Funktionsweise des DAF

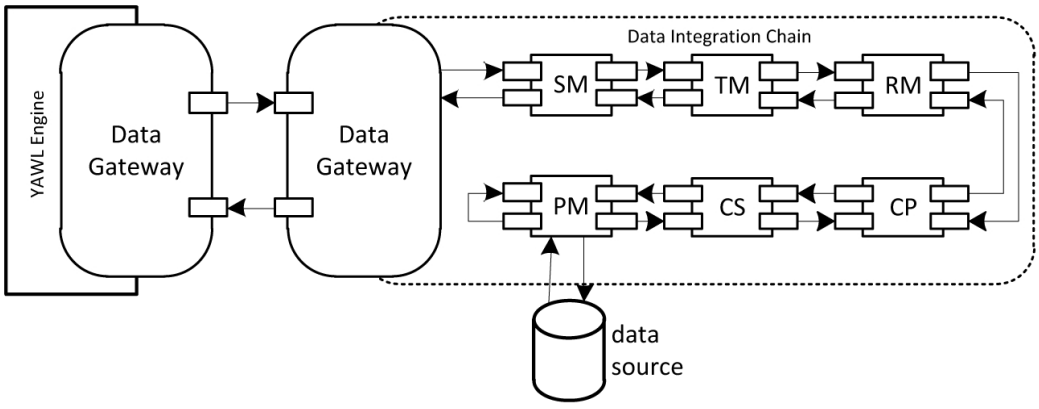

Abbildung 6: Darstellung des Data Access Frameworks [SHA11]

Das Data Access Framework wurde entwickelt, um YAWL Zugang zu externen Variablen zu ermöglichen. Das DAF verknüpft extern angebundene YAWL Variablen mit den entsprechenden

Datenquellen. Im Folgenden sollen die verschiedenen Komponenten des DAF vorgestellt und in einen Kontext gesetzt werden. Als Grundlagenwerk zu dieser Sektion wird sich auf [SHA11] bezogen.

Die Komponenten des DAFs schließen mit dem *Data Gateway* direkt an die *YAWL-Engine* an und sind über die *Data Integration Chain* direkt mit den externen Datenquellen verbunden. Der **Data Gateway** ist ein Interface in der YAWL-Engine, welches den Aufruf an externe Variablen abfängt und an Services außerhalb der YAWL-Engine weiterleitet. Dort werden die Anfragen bearbeitet und die Resultate in die YAWL-Engine zurück gegeben.

Der **Data Source Manager** (DSM) führt eine Vorauswahl des empfangenen Prozesses aus und kontrolliert die weitergehende Verarbeitung. So wird entschieden, welches Plugin für den Zugriff auf die Daten der angeforderten externen Variable initialisiert und genutzt werden muss.

Die **Date Integration Chain** besteht aus verschiedenen Services, die dazu dienen, den Aufruf den Umständen entsprechend abzuarbeiten. Wenn beispielsweise eine Variable außerhalb einer transaktionalen Sphäre gelesen werden soll, so ist das Aktivieren des *Transaktions-* und *Recoverymanagers* nicht notwendig. Sollte allerdings ein Aufruf zum Lesen einer externen Variable, die zu einem vorhergehenden Zeitpunkt schon gelesen wurde, erfolgen - so kann anhand des Status des Caches im Transaktionsmanager entschieden werden, ob die Variable aus dem Cache gelesen wird oder ob die Bearbeitungskette weitergeführt wird und die Variablen dann mittels der **Plugins** aus dem persistenten Speicher geladen wird.

3.2 verwendete Transaktionstypen

Wie im vorherigen Kapitel erwähnt, gibt es diverse Erweiterungen des Transaktionsbegriffs. Das Grundmodell von Transaktionen verwendet ein flaches Modell, welches keinerlei Hierarchien unterstützt. Eine Transaktion besteht in diesem Modell aus aufeinanderfolgenden hintereinander ausgeführten Aktionen. Sollte eine Aktion scheitern, ist ein festes Fehlermanagement, das Abbruch und Neustart der gesamten Transaktion beinhaltet, festgelegt.

Workflows modellieren (zukünftige) komplexe Geschäftsprozesse oder abstrahieren reale Arbeitsabläufe. Diese Darstellungen sind oftmals hierarchischer Natur. Eine flache Transaktion könnte nur einen Teilaspekt des Workflows umfassen.

Im Rahmen dieser Arbeit musste für tx+YAWL eine Transaktionserweiterung gefunden werden, die Hierarchien und langlaufende Prozesse unterstützt.

tx+YAWL Transaktionen unterstützt *nicht-vitale Aktionen und nicht-vitale Subtransaktionen* [SHA11]. Der Transaktionsmanager muss diese ebenfalls unterstützen. So könnte auf einen Abbruch einer Subtransaktion auf verschiedene Arten reagiert werden:

- Ersatztransaktion ausführen
- Ignorieren des Abbruchs der Subtransaktionen
- Neustart der Subtransaktion
- Abbruch der gesamten Transaktion

3.2.1 Problemstellung in Workflows

Bei der Erstellung des Transaktionsmanager konnten zwei Problemstellungen mit Transaktionen herausgearbeitet werden:

Gewöhnliche Transaktionen, wie in Kapitel 2 dargestellt, sind durch die Abwesenheit von Hierarchien und die strikten Einhaltung der ACID Kriterien in Workflowumgebungen nicht geeignet. Die oben aufgeführte Behandlung des Abbruches einer Subtransaktion würde beispielsweise die Atomarität verletzen.

Isolation bei Datenbanken gibt dem Benutzer das Gefühl, dass er der einzige Nutzer der Datenbank ist. Zur Erreichung dieses Ziels werden Datenzugriffe (Lese- und Schreibvorgänge) mit ausgeklügelten Sperrprotokollen wie dem 2-Phasen-Sperrprotokoll realisiert. Ergebnisse der Berechnung werden nach Abschluss der Transaktion veröffentlicht und können von nebenläufigen Transaktionen oder Geschwistertransaktionen benutzt werden.

Workflows stellen oftmals komplexe Geschäftsmodelle oder vordefinierte Abläufe dar. Diese Abläufe enthalten zum Teil langlaufende Tasks, die über einen langen Zeitrahmen parallel neben anderen Tasks laufen. In *tx+YAWL* können diese langlaufenden Tasks transaktionale Sphären sein, die Berechnungen vornehmen. Wenn die Resultate dieser Berechnung in anderen parallelen Tasks oder anderen Prozessen verwendet werden sollen, muss eine Alternative zur strikten Isolation gefunden werden.

Mit einer strikten Isolierung müsste die Publizierung der Resultate und subsequent der Beginn der anderen Workflowprozesse oder Tasks verschoben werden bis der gesamte Workflow abgeschlossen ist und somit sequentiell abgearbeit wurde. Damit wären nebenläufige Workflows oder auch langlaufende Workflows nicht mehr realisierbar.

Das andere Problemfeld betrifft die *Atomarität* von Transaktionen. Atomarität bedeutet, dass von einer Transaktion entweder *alles oder nichts* ausgeführt wird. tx+YAWL unterstützt neben kompensierenden Transaktionen auch Kontingenttransaktionen [SHA11]. Beide eint, dass sie das Kriterium der Atomarität verletzten. Durch die Ausführung einer Kontigenttransaktion, bei der nur eine von x Transaktionen abschließen muss, wird das *Alles-oder-nichts* Prinzip verletzt. Auch die kompensierende Transaktion, welche ausgeführt wird, nachdem eine Transaktion abgebrochen wird, verletzt die Atomarität. Ein Abbruch einer Aktion in einer gewöhnlichen Transaktion würde unweigerlich zum Abbruch und etwaigen Neustart der gesamten Transaktion führen.

Flache Transaktionen und Geschlossen geschachtelte Transaktionen unterstützen durch strenge Einhaltung der Atomarität keine nicht-vitale-Subtransaktionen und kompensierende Transaktionen [SH99]. Flache Transaktionen können nicht geschachtelt werden und Geschlossen geschachtelte Transaktionen ermöglichen keine kompensierenden Transaktionen, da Geschlossen geschachtelte Transaktionen die Atomarität erfüllen. Der Abbruch einer Kindtransaktion hätte somit den Abbruch der Vatertransaktion zufolge, welche wiederum ihre Vatertransaktion abbricht. Diese Kaskade wird solange durchgeführt, bis die gesamte Transaktion abgebrochen ist.

Offen geschachtelte Transaktionen empfehlen sich für den Einsatz im Transaktionsmanager von tx+YAWL durch den möglichen Einsatz von kompensierenden Transaktion und Kontingenttransaktionen, welches durch die aufgeweichte Atomarität[SH99] ermöglicht wird. Hier wird ein Abbruch einer Kindtransaktion mit dem im Anfang des Kapitels erwähnten Optionen *(retry, abort, compensating, ignore)* behandelt.

ONT ermöglichen durch die aufgeweichte Isolation auch die Freigabe der Berechnungsergebnisse und Aufhebung der Sperrung, um auf Objekten wie Netzvariablen bereits vor Abschluss der Workflowtransaktion zugreifen zu können. Somit können die Berechnungsergebnisse bereits mit Abschluss einer transaktionalen Sphäre von anderen Sphären genutzt werden. Damit wird eine deutlich höhere Parallelität erreicht.

ONT bietet durch die aufgeweichte Isolation und der Atomarität Vorteile im Bereich der Parallelität, der Sperrpolitik und zusätzlich ermöglicht es gegenüber CNT auch kompensierende Transaktionen und nicht-vitale Subtransaktionen. Auf Grundlage dieser Erkenntnisse wird der Transaktionmanager *Offen geschachtelte Transaktionen* nutzen, da nur diese Transaktionserweiterung den Anforderung bezüglich Isolation und Atomarität genügt. So erreicht man mit ONT eine deutlich höhere Parallelität, da Isolation nur noch auf subtransaktionaler Ebene erreicht wird. Sobald eine transaktionale Sphäre abgeschlossen wird, kann demnach nicht nur der aktuelle Workflowprozess die Resultate benutzen sondern auch andere Workflowinstanzen.

	Transaktionsart		
	flache	**CNT**	**ONT**
Atomarität	voll	voll	keine Atomarität
Isolation	voll	voll	subtransaktionale Ebene
Sperrfreigabe			
Möglichkeiten des Einsatzes			
Subtransaktionen	nein	ja	ja
Freigabe der Berechnungsergebnisse	Transaktionsende	Transaktionsende	mit Ende der Subtransaktion
Kontingenttransaktion	nein	nein	ja
kompensierende Transaktionen	nein	nein	ja
nicht-vitale Subtransaktionen	nein	nein	ja

Tabelle 3: Vergleich der Möglichkeiten der verschiedenen Transaktionsarten

In einem Workflowsystem wie YAWL und der Erweiterung tx+YAWL, die wie oben aufgeführt Zugriffe auf den persistenten Speicher über das DAF und subsequent Plugins ausführt bietet es Vorteile eine Zwischenebene wie MVCC einzuführen um den Overhead bei Schreib- und Leseoperationen zu verringern. Durch die Versionshistorie von MVCC kann der Zugriff auf die persistenten Speicher (Datenbank, direktes Dateisytem bspw. XML) minimiert werden, daraus resultierend steigt natürlich auch die Parallelität der Verarbeitung an.

Gleichzeitig ist es beispielsweise auch vorteilhaft, dass die Schreibzugriffe nicht direkt auf den persistenten Speicher geschrieben werden sondern sich in einer Art Zwischenspeicher befinden. Die Wiederherstellung im Fehlerfall wird erleichtert und durch die geringere Anzahl an Schreibzugriffen auf den persistenten Speicher steigt die Performance.

Der Einsatz von MVCC im Transaktionsmanager ermöglicht eine höhere Flexibilität in langlaufenden Workflows. Da die Vorteile gegenüber den Nachteilen überwiegen wird der Transaktionsmanager MVCC einsetzen.

3.3 Transaktionsmanager

In diesem Teilabschnitt der Arbeit werden die Transaktionsmanager kurz vorgestellt und dann verglichen.

Folgende Forderungen werden an die Transaktionsmanager gestellt:

3.4 Kriterienkatalog

Um die Auswahl zwischen den bestehenden Transaktionsmanagern zu erleichtern, wurde ein Forderungskatalog aufgebaut, mit dem die bestehenden Transaktionsmanager verglichen werden können. Nach der Aufstellung und Erläuterung der Kriterien werden die Transaktionsmanager gegen den Forderungskatalog verglichen

leichte Einbindung Um den möglichen Nutzern von tx+YAWL das Einrichten und den initialen Einsatz so reibungslos wie möglich zu gestalten, sollte sich der Transaktionsmanager leicht in die Workflowumgebung integrieren lassen.

Es ist vorteilhaft, dass er sich über die Nutzung von JAR-Datein einbinden lässt und möglichst wenig Abhängigkeiten zu anderen Projekten besitzt.

Einbindung in J2SE tx+YAWL wird in einer Standard Java Umgebung ohne Funktionen der Java Enterprise Edition (J2EE) ausgeführt. Der Transaktionsmanager soll keine höheren Ansprüche an die Umgebung stellen als das Framework.

leichte Erweiterbarkeit Da vorhandene Transaktionsmanager häufig auf die Verwendung in Zusammenhang mit Datenbanken wie MySQL, MsSQL konzipiert sind, unterstützen die hier betrachteten TM oftmals Protokolle wie JDBC zur Kommunikation mit dem persistenten Speicher. Aufgrund der Architektur von tx+YAWL, die eine Nutzung von verschiedenen, nicht transaktionalen Datenquellen ermöglichen will, ist die Erstellung eines JDBC-Treibers nicht möglich.

Gleichzeitig sind die hier betrachteten Transaktionsmanager seit Jahren entwickelte und ständig erweiterte Produkte, die durch den professionellen Einsatz in etlichen Applikationen chronologisch gewachsen sind und häufig einen Überfluss an Funktionen entwickelt haben. Von diesen Funktionen würde der hier entwickelte Transaktionsmanager jedoch nur wenige nutzen, weshalb es von Vorteil ist, wenn die Architektur des TMs sich leicht anpassen lässt.

gute Dokumentation aufgrund der anstehenden Anpassungs- und Erweiterungsaufgaben, sowie zur Erleichterung des Einstieges in die Entwicklung mit dem ausgewählten Transaktionsmanagers sollte dieser gut dokumentiert sein und eine aktive Entwicklergemeinde innehaben.

geringer Speicherplatzbedarf tx+YAWL soll nur so wenig Speicher verbrauchen wie nötig. Dazu ist es notwendig, dass der Transaktionsmanager als Komponente des DAF möglichst wenig Speicherplatz verbraucht.

Recoverykomponenten Mit einer Recoverykomponente können im Fehlerfall Daten restauriert und die Ausführung an der letzten erreichten Aktion des Workflows fortgeführt werden. Traditionellerweise erfordert die Wiederherstellung die Beachtung vieler Eckpunkte, um eine konsistente weitere Ausführung zu gewährleisten. Eine bereits implementierte Recovery wäre eine logische Ergänzung des Transaktionsmanagers.

Unterstützung von MVCC Wie in diesem Kapitel beschrieben, bieten MVCC-Transaktionen Vorteile gegenüber gewöhnlichen Transaktionen.

Unterstützung von geschachtelten Transaktionen Die Unterstützung von geschachtelten Transaktionen ist ebenfalls eine Bedingung an den Transaktionsmanager.

3.5 Betrachtete Transaktionsmanager

Nachdem im vorhergehenden Kapitel der Forderungskatalog an die Transaktionsmanager aufgestellt wurde, sollen in diesem Kapitel die Transaktionsmanager Bitronix, Atomikos TransactionEssential und JBoss TS gegen den Katalog verglichen werden.

3.5.1 Bitronix

Die Entwicklung am Bitronix Transaktionsmanager (BTM) begann 2005. Bitronix war zu dem Zeitpunkt ein *Ein-Personen Projekt*. BTM kann unter der GNU LGPL[3] bezogen werden. Für die derzeit aktuellste Version 2.1.2 wird das JDK 1.5 benötigt. Bitronix konzentrierte sich auf XA mit JDBC und/oder JMS. Andere Services wie die JTS werden vom BTM nicht unterstützt. Bitronix fehlt die Unterstützung von geschachtelten Transaktionen und hat nur eine kleine Entwicklungs- und Supportgemeinde.

3.5.2 Atomikos - TransactionEssentials

Atomikos bietet ihren Transaktionsmanager in zwei Varianten an: *Transaction Extreme* ist die kostenpflichtige, durch Support unterstützte Variante, die hier nicht betrachtet wird und *Tran-*

[3]GNU Lesser General Public License : http://www.gnu.org/licenses/lgpl.txt

Kriterium	**Bitronix**
Einbindung	einfache Einbindung via Eclipse durch den Import diverser JAR-Files
J2SE	ja
Erweiterbarkeit	durch die kleine Entwicklungsgemeinde können Fragen eventuell nicht zeitnah beantwortet werden
Dokumentation	Dokumentation zur Installation, Nutzung auf der Homepage
Speicherplatzbedarf	Download der Installtionsdatei ist etwa 2,9 Mb groß
Recoverykomponente	Crash Recovery
MVCC	keine Informationen
geschachtelte Transaktionen	keine Unterstützung von NT

Tabelle 4: Bitronix vs. Kriterienkatalog

saction Essentials, welche seid 2006 unter der Apache Lizenz[4] steht. In diesem Vergleich wird *TransactionEssentials* betrachtet.

Transaction Essentials kann in einer Standard-Java Umgebung eingesetzt werden und bietet eine eingebaute Unterstützung von geschachtelten Transaktionen sowie eine gute Dokumentation und eine aktive Community [Ato11b]. Ein großer Vorteil ist die in TransactionEssentials direkt integrierte Recoverykomponenten, für die man allerdings das JDBC-Interface nutzen muss. Außerdem spricht die leichte Integration in ein Projekt ohne zusätzliche andere Services und die Integration mit anderen Java Produkten und Webapplication Servern wie TomCat für TransactionEssentials [Ato11a].

3.5.3 JBoss TS

Der Transaktionsmanager *JBoss Transactions* (JBoss TS) wurde Ende 2005 von HP und Arjuna Technologies entwickelt und durch JBoss erworben und unter Open Source gestellt. Zu diesem Zeitpunkt konnte der Transaktionsmanager schon auf eine 20-jährige Geschichte zurückblicken. JBoss TS unterstützt seit JBoss AS 5.0 auch die J2SE 1.5 [JBo11b]. Vorher wurden nur ältere Versionen von JAVA unterstützt.

JBoss TS unterstützt geschachtelte Transaktionen und hat ein eingebautes automatisches Recovery-

[4]Lizenz von Atomikos: http://www.atomikos.com/licenses/apache-license-2.0.txt

Kriterium	**Atomikos - TransactionEssentials**
Einbindung	sehr einfach
J2SE	ja
Erweiterbarkeit	erweiterbar und gute Community Support vom Hersteller erfordert den Kauf von TransactionExtreme
Dokumentation	sehr gute Online-Dokumentation sehr gutes herunterladbares Handbuch
Speicherplatzbedarf	<1 Mb
Recoverykomponente	eingebaut
MVCC	Mit verwendung von InnoDB in MySQL unterstützt
geschachtelte Transaktionen	unterstützt

Tabelle 5: Atomikos vs. Kriterienkatalog

System. Es lässt sich leicht mit anderen JBoss Produkten erweitern.[JBo11a]

Die Einbindung von JBoss TS erfolgt über die Installation von zwei Services in Windows und der Einbindung von diversen JAR-Files in das Eclipse Projekt. Das JBoss TS System ist das in diesem Vergleich am schwersten einsetzbare Produkt. Andererseits glänzt es durch einen extrem hohen Funktionsumfang, einer sehr guten Dokumentation und guten Erweiterungsmöglichkeiten.

3.5.4 Vergleich der Transaktionsmanager

Der Vergleich der Transaktionsmanager hat gezeigt, dass die betrachteten vollwertigen Transaktionsmanager wie Bitronix, Atomikos und JBoss TS viele der benötigten Funktionen anbieten.

Aufgrund der fehlenden Unterstützung von geschachtelten Transaktionen und der kleinen Entwicklungsgemeinde um Bitronix fällt dieser TM aus der weiteren Betrachtung heraus.

JBoss TS und Atomikos können alle benötigten Funktionen entweder von Haus aus oder durch Erweiterungen anbieten. Beide Transaktionsmanger eint, dass man im Internet eine Fülle an Informationen über sie findet. Gleichzeitig gibt es durch die hohe Verbreitung der beiden Transaktionsmanger eine große Community, die den Entwickler bei Fragestellungen rund um die TMs unterstützen können.

Kriterium	**JBoss TS**
Einbindung	Installation von Services, Veränderung des Classpathes Einfügen diverser JAR-Files
J2SE	J2SE 1.5 seit JBoss AS 5.0 unterstützt.
Erweiterbarkeit	sehr gute Erweiterbarkeit und Integration mit anderen JBoss Services
Dokumentation	sehr gute Dokumentation, sehr gute Community
Speicherplatzbedarf	Download der Installtionsdatei ist etwa 5.6 Mb groß
Recoverykomponente	eingebaut
MVCC	durch Einbindung eines Cache-Systems wie JBoss Cache realisierbar
geschachtelte Transaktionen	durch JBoss TS unterstützt

Tabelle 6: JBoss TS vs. Kriterienkatalog

Das DAF soll in der Lage sein, Daten aus einer Menge von Datenquellen (XML, Datenbanken) über die Plugin-Architektur zu beziehen. Um die hier diskutierten Transaktionsmanager nutzen zu können, müsste die Plugin-Architektur des DAFs transaktional (XA) aufgebaut sein und JDBC-Treiber unterstützen. Aufgrund der bereits besprochenen Vielzahl von Datenquellen ist es unpraktikabel eine transaktionale Architektur für die Plugins zu konzipieren und zu implementieren. Die Plugin-Architektur des DAFs kann nicht ohne weitere Anpassungen durch die Transaktionsmanager angesprochen werden.

Auch der **enorme Umfang** von *Atomikos TransactionEssentials* und *JBoss TS* spricht gegen diese TMs. Der von tx+YAWL benötigte Transaktionsmanager nutzt nur einen winzigen Teil der bereitgestellten Funktionalitäten, benötigt aber weitere Anpassungen. Es muss viel Arbeit in die Anpassung der betrachteten TMs investiert werden.

Wie in der Aufgabenstellung erwähnt, sollte ein Weg gefunden werden, wiederholte Lesezugriffe auf Netzvariablen zu vermeiden. Die konventionellen Transaktionsmanger (JBoss TS, etc.) konzentrieren sich auf das Verwalten von Transaktionen und sorgen dafür, dass Transaktionen vollständig ausgeführt oder abgebrochen werden. Sie stellen **keine Datenhaltungsschicht** zur Verfügung. Dies würde bedeuten, dass ein gesondertes Datenhaltungssystem implementiert werden müsste.

Desweiteren beschränkt sich die **MVCC-Unterstützung** der beiden Transaktionsmanager Atomikos und JBoss TS auf das Schreiben in die Datenbank. Dafür ist es allerdings notwendig, dass die Daten in der Datenbank in einem MVCC kompatiblen Datenbankformat (MySQL: InnoDB) vorliegen. Damit wären aber nur die Schreibzugriffe auf den persistenten Speicher auf Basis von MVCC nicht aber die Zugriffe in der Zwischenschicht abgedeckt.

Alle hier vorgestellten Transaktionsmanger sind Zwischenschichten die selbst keine Daten halten. Stattdessen agieren sie als Mittler zwischen verschiedenen Programmbauteilen und ermöglichen so den Zugriff auf eine Datenbank. Durch diese Ausrichtung auf Datenbanksysteme und der daraus resultierenden Beschaffung von Informationen aus einer Datenquelle - der Datenbank, sind sie nicht ohne Anpassungen auf die hier diskutierte Problemstellung anwendbar.

3.5.5 Auswahl des Transaktionsmanagers

Die von den betrachteten Transaktionsmanagern bezogenen Daten kommen aus einer Datenquelle, die über ein JDBC Interface angesprochen wird. Der in dieser Arbeit zu erstellende Transaktionsmanager muss mit einer Vielzahl an Quellen umgehen können. Das ist mit Atomikos und JBoss nur schwer umsetzbar, da für diese Lösungen pro Datenquelle, ein eigener XA-Kompatibler JDBC-Treiber nötig ist.

Auch die unzureichende MVCC-Unterstützung durch die vorhandenen Transaktionsmanager und die fehlende Datenhaltungsschicht sprachen gegen einen vorhanden Transaktionsmanager.

Da keiner der bereits vorhandenen Transaktionsmanager auf das Anforderungsprofil (siehe Kapitel 3.5.4) passte, wurde aufgrund der oben genannten Nachteile (MVCC-Unterstützung, keine Datenhaltungsschicht, enormer Umfang, JDBC-Driver für Pluginsystem) entschieden, einen eigenen Transaktionsmanager mit angeschlossener Datenhaltungsschicht zu entwickeln.

4 Konzept

Nachdem in Kapitel 2 die grundlegenden Konzepte von Transaktionen erläutert und ein Fokus auf die verschiedenen Teilaspekte von Transaktionen *(ACID, Scheduler, Serialisierbarkeit)* und Workflows gelegt wurde, hatte Kapitel 3 die verwendete Software und Technologien vorgestellt. In diesem Kapitel wird das Konzept zur Umsetzung des Transaktionsmanagers dargestellt. Hierbei werden die in den vorhergehenden Kapiteln vorgestellten Technologien, Programme und Theorien kombiniert.

4.1 Prämisse

Im ersten Teil des Konzeptes werden nun einige Vorbedingungen aufgestellt. Aufgrund der veränderten Anforderungen, die durch transaktionale Workflows gestellt werden, beispielsweise das langlaufende Transaktionen unterstützt werden müssen, wurde entschieden, dass der Transaktionsmanager keinen gewöhnlichen Datenbankscheduler enthalten kann.

Ein Scheduler in einem Datenbanksystem ist, wie in den vorhergehenden Kapiteln beschrieben, eine der Hauptkomponenten des Transaktionsmanagers, der aus (mehreren) Eingabeschedules einen (serialisierbaren) Ausgabeschedule generieren kann. Der Datenbankscheduler kann eine Transaktion kurzzeitig anhalten, abbrechen und neu starten. Ein OLTP-Datenbankscheduler (*Online Transaction Processing*) trifft die Entscheidung aufgrund der aktuellen Aktion, er hat keinen Zugriff auf die kommenden Aktionen der Eingabeschedules.

Der Transaktionsmanager von tx+YAWL weiß bereits mit Beginn des Workflows, welche Tasks das Netz enthält und welche Tasks (externe) Variablen schreiben werden. Außerdem muss er den Tasks transaktionalen Sphären zuordnen können.

Zusätzlich soll er auch den Pfad identifizieren können, indem sich ein Task befindet.

Zur Erfüllung diese Anforderungen muss das tx+YAWL Netz nach der Erstellung oder einer Veränderung analysiert werden. Diese Strukturinformationen müssen gesichert und dem Transaktionsmanager zugänglich gemacht werden.

Dies bedeutet einen entscheidenden Schritt weg vom klassischen Scheduler, wie er aus Datenbanksystemen bekannt ist. Der in dieser Arbeit entwickelte Transaktionsmanager weiß bereits mit Beginn des Netzes, wann welche Variable geschrieben werden muss. Er kann Konflikte somit frühzeitig erkennen und die Anforderung des Read- und Writesets von tx+YAWL erfüllen, indem jede Variable in einer transaktionalen Sphäre nur einmal aus dem persistenten Speicher gelesen oder in diesen geschrieben wird. Das analysierte Netz wird gespeichert und kann vom Transaktionsmanager abgerufen werden.

4.2 Einordnung in tx+YAWL

Die YAWL Erweiterung tx+YAWL erweitert YAWL um transaktionale Sphären und ermöglicht transaktionale Workflows. Der Zugriff auf externe Parameter wird in tx+YAWL durch das Data Access Framework realisiert.

Das *Data Access Framework* ist eine Komponente von tx+YAWL. Es besteht aus dem *Data Gateway* und der *Data Integration Chain*, welche durch eine *Apache Chain* realisiert wurde.

Die Data Integration Chain besteht aus verschiedenen durch eine Apache Chain miteinander verbundenen Bauteile. In der Abbildung 7 ist das DAF mit den verschiedenen Komponenten der Data Integration Chain abgebildet.

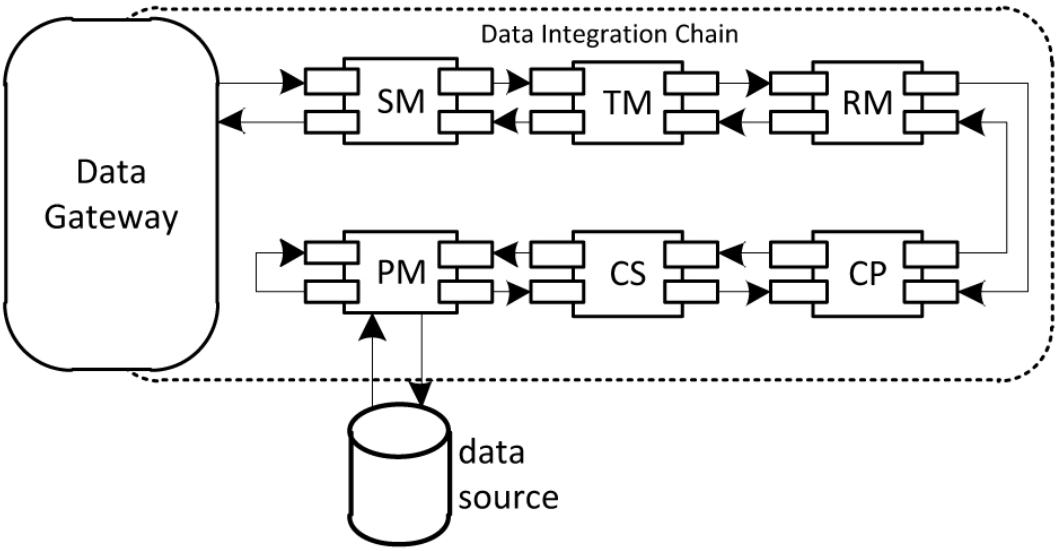

Abbildung 7: Aufbau der Data Integration Chain

Begleitend bietet die Chain ein Kontextobjekt, welches durch die verschiedenen Instanzen der Chain durchgereicht wird. Aus diesem Kontextobjekt können alle Kettenglieder Informationen beziehen und auch Informationen zurückschreiben. Der in dieser Arbeit entwickelte Transaktionsmanager ist Bauteil der Chain und ordnet sich zwischen dem Synchronisierungs- und Recoverymanager ein.

4.2.1 Zugriffe auf externe Parameter

Sollte ein Task eines tx+YAWL Netzes auf einen externen Parameter lesend oder schreibend zugreifen, wird der Zugriff im YAWL Netz über den *Data Gateway* abgefangen. Der Data Gateway leitet den Zugriff zum Data Source Manager, welcher sich außerhalb der YAWL Engine befindet, weiter. Der *Data Source Manager* (DSM) startet die Bearbeitung der Apache Chain.

Bei einem Zugriff auf externe Parameter werden die Komponenten der Apache Chain nacheinander durchlaufen. Zwischen den Komponenten wird das durch die Apache Chain bereitgestellte Kontextobjekt durchgereicht. Wenn eine Komponente der Chain den Lesezugriff ausführt, werden die Resultate in das Kontextobjekt geschrieben und die Chain rückwärts erneut durchlaufen. Bei dem Rücklauf der Chain können die höher angeordneten Komponenten der Chain auf das Kontextobjekt zugreifen und Daten extrahieren.

Bei einem Zugriff innerhalb eines tx+YAWL Netzes auf einen externen Parameter wird der entwickelte Transaktionsmanager prüfen, ob er den Zugriff ausführen kann. Ist das der Fall, liest oder schreibt er die angeforderten Daten und schreibt das Resultat in das Kontextobjekt, dass durch den *Synchronisierungsmanager* und *Constraint Service* zurück an die YAWL Engine übergeben wird.

Wenn der Transaktionsmanager den lesenden Zugriff nicht ausführen kann, weil die Daten noch nicht in dem Zwischenspeicher vorhanden sind, wird die Chain weiter durchlaufen bis der *Pluginmanager* (PM) den Lesezugriff aus der externen Datenquelle ausführt. Das vom PM zurückgegeben Resultat wird im Transaktionsmanager gespeichert und für zukünftige Lesezugriffe vorgehalten.

In den folgenden Subkapiteln wird der genaue Ablauf eines lesenden- und schreibenden Zugriffes aufgeführt.

4.3 Aufbau und Funktionsweise des Transaktionsmanagers

In diesem Abschnitt soll die Funktionsweise der lesenden und schreibenden Zugriffe durch den Transaktionsmanager erläutert werden. Da kein vorhandener Transaktionsmanager genutzt werden konnte, mussten im Rahmen dieser Arbeit eigene Funktionen eingeführt werden, die die gewünschte Funktionalität umsetzen. Der Transaktionsmanager ist auf die Bedürfnisse von tx+YAWL zugeschnitten und setzt beispielsweise lesende und schreibende Zugriffe aus verschiedenen Quellen um.

4.3.1 Aufbau des Transaktionsmanagers

In Abbildung 8 ist der hier konzipierte Transaktionsmanager abgebildet. In den folgenden Kapiteln werden die Komponenten im Detail erläutert. Die Komponente *Konfiguration* wird im Subkapitel 4.3.3 *Konfiguration des TM* beschrieben. Das Zusammenführen von Parametern wird in dem Subkapitel 4.3.5 *Taskvorbereitung* erläutert, die *Zugriffe* werden im Abschnitt Lese- und Schreibzugriffe dargestellt. Das Ende einer transaktionalen Sphäre (EOS) wird im gleichnamigen

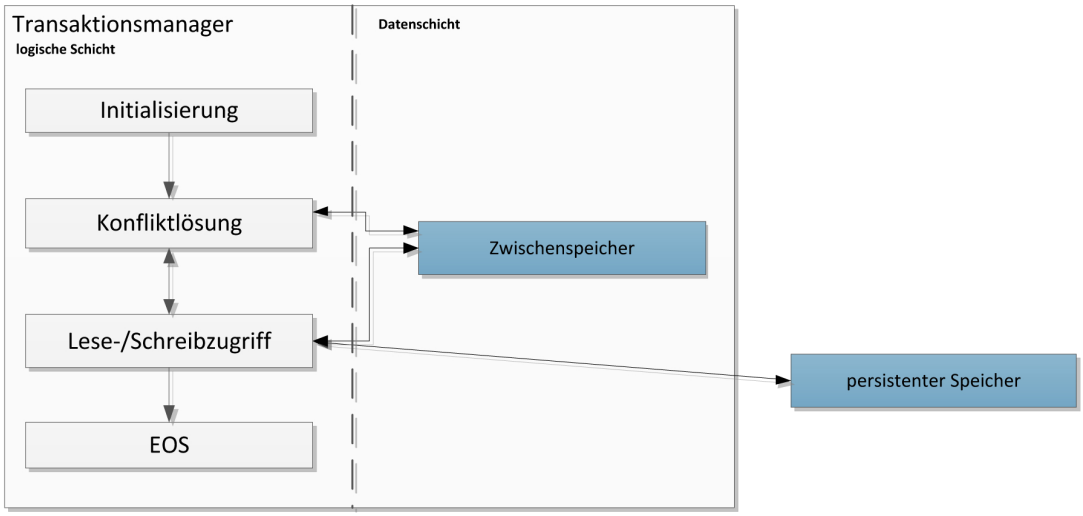

Abbildung 8: Architektur des Transaktionsmanager

Subkapitel 4.3.9 behandelt.

4.3.2 Strukturinformationen

Wie in der Prämisse (Kapitel 4.1) beschrieben soll der Transaktionsmanager bereits bei Beginn des Workcases wissen, welche Sphären und Tasks im Netz enthalten sind. Dafür ist es notwendig, dass eine Analyse und Speicherung der Netzinformationen nach der Erstellung oder Veränderung eines Netzes im YAWL-Editor durchgeführt wird. Die vom Transaktionsmanager benötigten Informationen sind in Abbildung 9 dargestellt.

In den Strukturinformationen sollen die Sphären des Netzes aufgeführt sein. Zusätzlich sollte im Hinblick auf die mögliche Verschachtelung von transaktionalen Sphären ein Attribut vorhanden sein, um eine mögliche Vatersphäre identifizieren zu können.

Zusätzlich zu den Sphären müssen die im Netz vorhanden Tasks gespeichert werden. Da das DAF derzeit keine Informationen darüber liefert, in welcher Sphäre sich der aktuelle Task befindet, muss diese Informationen in den Strukturinformationen gespeichert sein. Während der Ausführung eines Netzes besteht ebenfalls keine Möglichkeit zu erkennen, ob der aktuelle ausgeführte Task der letzte Task einer Sphäre ist. Deshalb sollten in den Netzinformationen auch der jeweils letzte Task gespeichert werden (*EOT-Attribut*).

Sollte das Netz parallele Pfade beinhalten, deren Tasks zum selben Zeitpunkt schreibend auf einen externen Parameter zugreifen wollen, werden diese Tasks synchronisiert. Hier muss die

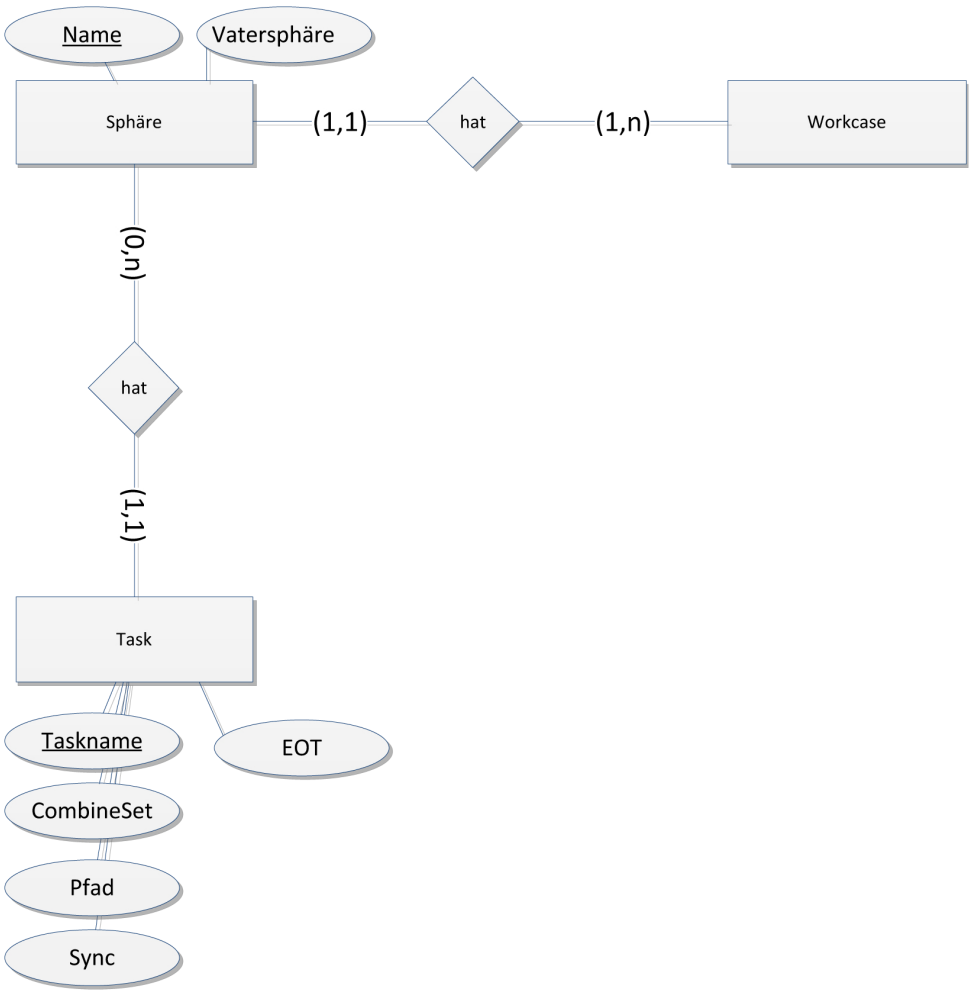

Abbildung 9: ER-Diagramm der Strukturinformationen

parallele Verarbeitung, die mit MVCC möglich ist, sequentialisiert werden, denn ohne eine Synchronisierung wäre es zufällig, welcher Task den letzten schreibenden Zugriff auf einen Parameter durchführt. Somit wäre ein nichtdeterministische unwiederholbarer Wert im persistenten Speicher hinterlegt. Um dies zu verhindern, kann ein Task ein *Sync-Attribut* beinhalten.

Um zu erkennen, in welchem parallelen Abarbeitungspfad sich ein Task derzeit befindet, wird das *Pfad-Attribut*, eine Erkennung für den Pfad, hinterlegt.

4.3.3 Konfiguration des Transaktionsmanagers

Sollten in dem aktuellen Workcase externe Variablen enthalten sein, werden die Zugriffe von YAWL auf das DAF weitergeleitet. Im DAF wird das Kontextobjekt aufgebaut und mit den

benötigten Informationen wie der CaseID, dem Tasknamen, dem Parameternamen und weiteren Informationen befüllt. Im nächsten Schritt werden Informationen über den Parameter bezogen und es wird beispielsweise analysiert, aus welcher Quelle gelesen werden soll.

Diese Informationen werden dem Transaktionsmanager übergeben, der die gespeicherten Strukturinformationen des Cases untersucht, um den aktuellen Bearbeitungsstand des Netzes zu analysieren. Hierfür ist es nötig, dass der Transaktionsmanager aus den Strukturinformationen sämtliche Sphären und Tasks erkennt und ihren jeweiligen Attributen zuordnet.

Die Metainformationen der Sphären und Tasks sollen in einer geeigneten Datenstruktur gespeichert werden. Da es derzeit nicht möglich ist, Informationen über transaktionalen Sphären über YAWL ins DAF zu übertragen, muss der Transaktionsmanager die Positionierung innerhalb des Netzes bzw. der Sphäre anderweitig vornehmen.

Ausgehend von den vorhandenen Informationen über einen Task sollte der Transaktionsmanager erkennen, in welcher Sphäre sich der Task befindet. Daraus kann geschlussfolgert werden, dass die Struktur auf lesende Zugriffe optimiert und ein schneller Zugriff auf die Daten gewährleistet sein muss. Der Fokus liegt daher auf den Tasks.

Die Datenstruktur spiegelt die gegebene Anfordungen wieder und bietet einen schnellen Zugriff mit den Namen des Tasks oder der Sphäre an. Dafür könnte die folgende in BNF notierte Speicherstruktur der Sphären und Tasks genutzt werden.

Sphäre Die Sphäre beinhaltet als Attribut, wie im vorigen Abschnitt beschrieben, ob sie eine Vatersphäre besitzt oder nicht. Diese Information muss in der Datenstruktur übertragen und gespeichert werden. Zusätzlich sollte in der Sphäre vermerkt sein, welcher der letzte Task der Sphäre ist. Bei der Notation des letzten Tasks ist es nicht ausreichend den letzten Task der Sphäre zu speichern. Stattdessen sollten in diesem Attribut der Task oder im Fall einer parallelen Abarbeitung die Tasks gespeichert werden, die die letzten Zugriffe auf externe Variablen innerhalb der Sphäre vornehmen.

Die Datenstruktur der Map kann wie folgt in BNF notiert werden:

```
1 sphären = sphäre+
2 sphäre = sphärenname sphärenobjekt
3 sphärenobjekt = sphärenname lastTasks
4 lastTasks = taskname ','+
5 sphärenname = AlphaNumerischeZeichen+
```

```
6 taskname = AlphaNumerischeZeichen+
```

Task Das Taskobjekt muss im Gegensatz zu dem Sphärenobjekt allerdings mehr Informationen vorhalten. Die vom Taskobjekt gehaltenen Attribute sind die folgenden *Sync, Path, EOT, CombineSet, Sphere* und *Taskname*.

In der Analyse werden Tasks bei denen das *EOT*-Attribut gesetzt ist, dem *lastTasks*-Attribut ihrer korrespondierenden Sphäre hinzugefügt. Das Taskobjekt kann ebenfalls in BNF notiert werden:

```
1 Tasks = Task+
2 Task = Taskname Taskobjekt
3 Taskobjekt = Sphere Path Parametername CombineSet EOT Sync
4 CombineSet = Parametername*
5 EOT = [TRUE | FALSE]
6 Sync = [Taskname ',']*
7 Path = Ziffer+
8 Taskname = AlphaNumerischeZeichen+
9 Sphere = AlphaNumerischeZeichen+
10 Ziffer = '0' | '1' | '2' | '3' | 5 | ? | '9'
11 Parametername = AlphaNumerischeZeichenUndSonderzeichen+
```

Nachdem die Analyse des Netzes abgeschlossen ist, kann der Transaktionsmanager mithilfe dieser Informationen lesende und schreibende Zugriffe gestalten.

4.3.4 Vorbereitung des Tasks

Nach der Analyse des Netzes muss untersucht werden, ob der aktuelle Task Teil einer transaktionalen Sphäre ist. Nur Tasks, die innerhalb eine transaktionalen Sphäre liegen, erhalten transaktionales Verhalten. Lese- und Schreibzugriffe, die ausserhalb einer transaktionalen Sphäre auftreten, werden sofort an den Pluginmanager von tx+YAWL durchgereicht.

Sollte der Task Teil einer transaktionalen Sphäre sein, wird der Zwischenspeicher geladen, in welchem sich die Werte und Resultate von Variablen des aktuellen Cases befinden. Nähere Informationen zum Zwischenspeicher sind im *Kapitel 4.4 Datenhaltung* aufgeführt.

4.3.5 Konfliktauflösung

In dieser Arbeit wird die Snapshot Isolation (SI) in Kombination mit dem *MultiVersion Concurrency Control* verwendet.

Eine Transaktion x, die SI anwendet, liest einen konsistenten Wert des Parameter aus der Datenquelle. Dieser Wert ist der letzte `comittete` Werte in der Datenquelle. Der Wert wird während der Abarbeitung verwendet, ohne nochmals auf die Datenquelle zuzugreifen. Bei Ende der Transaktion x werden die Werte zurück in den persistenten Speicher geschrieben. Damit ermöglicht SI wie MVCC einen deutlich höheren Transaktionsdurchsatz im Vergleich zum *2-Phasen Sperrprotokoll*. Nachteilig an SI ist, dass es in einigen Fällen Transaktionen ausführt, die nicht serialisierbar sind und durch MVCC abgebrochen würden [CRF06]. Diese Fälle müssen im Rahmen des Transaktionsmanagers erkannt und behandelt werden.

write/write Konflikt
Sollten in einer parallelen Abarbeitung in zwei Tasks **gleichzeitig** schreibend auf eine Variable zugegriffen werden, entsteht ein **Write/Write-Konflikt**. Um einen Datenverlust zu verhindern, wurde entschieden, dass die schreibenden Zugriffe von verschiedenen Tasks auf einen Parameter gegeneinander synchronisiert und sequentialisiert werden.

Nach dem FCW-Prinzip[5] müssten alle parallelen Tasks, die ebenfalls einen Schreibzugriff auf den Parameter unternehmen, abgebrochen werden, sobald der erste Task abgeschlossen ist. Dies ist in einer Workflowumgebung mit langlaufenden Aktionen nicht akzeptabel, weil eventuell die Resultate anderer langlaufender Aktivitäten verworfen würden.

Da der hier konzipierter Transaktionsmanager wie in Kapitel 4.3.2 beschrieben im vornherein über den Aufbau des Workcases informiert ist, kann er frühzeitig eine Situation wie in Abbildung 10 durch **Sequentialisierung** der betreffenden Tasks lösen.

read/write Konflikt
Ein **read/write Konflikt** tritt auf, wenn zwei Tasks lesend und schreibend auf einen Parameter zugreifen. Dabei kann der schreibende Zugriff beispielsweise vor dem lesenden Zugriff auftreten. Durch die Kombination von SI und MVCC wird der geschriebene Wert erst mit Ende der transaktionalen Sphäre in die persistente Datenquelle publiziert. Die lesende Transaktion liest dann den veralteten Wert des Parameters.

[5] first committer wins

Dieses Problem kann in mehreren Ausführungen auftreten: in parallelen Workcases, Pfaden oder parallelen transaktionalen Sphären. Es lässt sich beheben, indem der lesende Task über den Schreibzugriff informiert wird und erneut liest oder der Read/Write Konflikt zu einem Write/Write Konflikt befördert wird und eine oben genannte Sequentialisierung gestartet wird [CRF06].

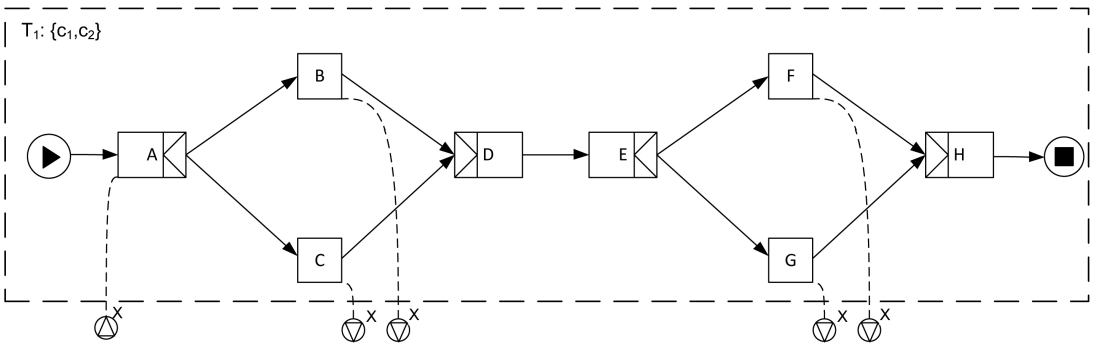

Abbildung 10: Parallel schreibender Zugriff auf Variable x

Beispiel write/write Konflikt
Das in Abbildung 10 abgebildete Netz stellt einen Kontozugriff dar. Task B und Task C sowie Task F und Task G versuchen einen schreibenden Zugriff auf die Variable x vorzunehmen. Die Variable x steht in diesem Beispiel für eine Überweisung auf ein Konto.

Wenn Task B und Task C jeweils 100 Euro auf ein Konto überweisen, dann muss der Kontostand in Task D 200 Euro betragen. Sollten Task F 100 Euro abbuchen und Task G 100 Euro aufbuchen, muss der finale Kontostand am Ende des Workcases bei 200 Euro liegen.

Dies bedeutet, dass Task A gegen Task B und Task F gegen Task G synchronisiert werden muss. Dieser Workcase enthält also zwei *Synchronisierungssets s* mit

$s = \{(TaskA, TaskB), (TaskF, TaskG)\}$

Die Snapshot Isolation ist nicht Teil des prototypischen Transaktionsmanagers und wird im Transaktionsmanager durch einen Logeintrag angezeigt. Stattdessen wurde die im Kapitel 4.3.6 beschriebe Pfadlösung für den Prototypen eingeführt.

Sollte der Task ein CombineSet besitzen, wird vor jeder weiteren anderen Aktion die Kombination der Variablen vorgenommen, die im CombineSet aufgeführt sind. Erst nach Abschluss dieser

Aktion kann der Task seine Lese- oder Schreibzugriffe ausführen.

4.3.6 Pfade

Mit dem *Pfad*-Attribut ist es möglich parallele Pfade, die im YAWL Editor modelliert wurden, dem Transaktionsmanager deutlich zu machen. Dies ist notwendig, da in YAWL-Netzen auch innerhalb von parallelen Pfaden Berechnungen auf einer Variable erfolgen können. Jeder Pfad in Abbildung 11 kann demnach auf einer eigenen Version von x Operationen durchführen ohne mit den anderen Pfaden zu interagieren. Der Zugriff auf eine Variable innerhalb von parallelen Pfaden birgt Risiken:

1. Welche Variable wird übernommen?
2. Wie wird der Datenverlust behandelt?
3. Wurde die Variable schon initialisiert?

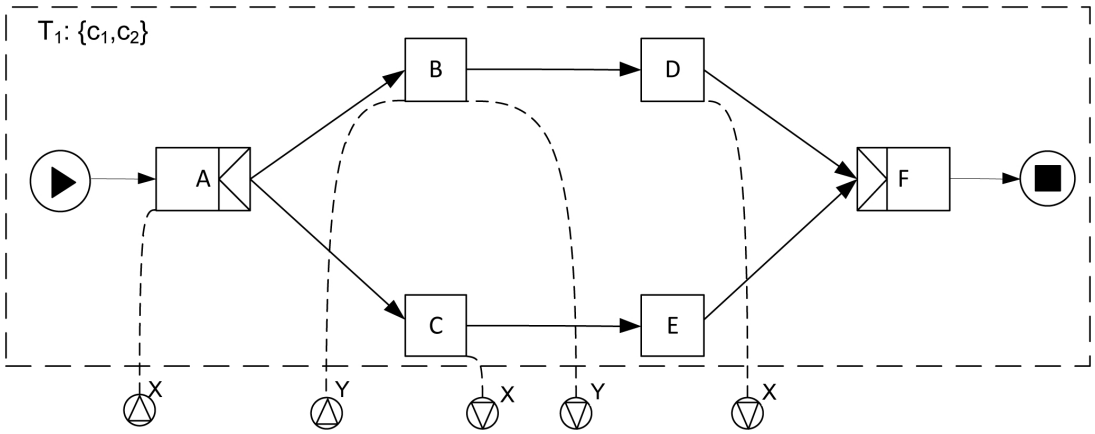

Abbildung 11: YAWL-Netz mit lesenden und schreibenden externen Parametern

Aufgrund dieser Probleme ist der Ablauf des Netzes nicht mehr deterministisch, da nicht bekannt ist welchen Wert eine Variable zum Abschluss des Netzes angenommen hat. Bei einem erneuten Ablauf des Netzes sind diese Ergebnis eventuell nicht wiederholbar. Eine Modellierung wie in Abbildung 11 sollte vermieden werden.

Da es derzeit noch keine Analysetools gibt, um den Modellierer vor diesem Fehler zu warnen und die *Snapshot Isolation* in tx+YAWL noch nicht einsatzfähig ist (Kapitel 4.3.5), wurde beschlossen die Pfadarchitektur einzuführen.

Lösungsansatz zu Risiko 1 Jeder Schreibzugriff auf eine Variable innerhalb eines Pfades erstellt eine neue Version der Variable, welche durch eine ID gekennzeichnet wird. Zusätzlich wird in einem LOG-File der letzte schreibende Zugriff auf diese Variable notiert. Nach dem Abschluss einer parallelen Abarbeitung wird die Version der Variable gewählt, auf die zuletzt schreibend zugegriffen wurde.

Lösungsansatz zu Risiko 2 Die Ergebnisse der anderen Pfade sind für die weiterführende Berechnung nicht interessant, werden aber im Speicher gehalten.

Lösungsansatz zu Risiko 3 *Pfad 0* stellt den Hauptpfad des jeweiligen Cases dar. Jeder Lesezugriff, der auf einen Parameter ausgeführt wird, wird in Pfad 0 gespeichert wenn in diesem Pfad noch kein Wert der Variable hinterlegt ist. Der initiale Lesezugriff auf einen Parameter wird demnach immer in Pfad 0 gespeichert.

Damit wird allen anderen Pfaden ein Zugriff auf die Grundversion ermöglicht. Die Anforderung des Readsets ist somit umgesetzt - keine Variable wird zweimal aus dem externen Speicher angefordert.

4.3.7 Lesezugriffe

Lesende Zugriffe durch Tasks in transaktionalen Sphären auf externe Variablen werden durch den Transaktionsmanager gesteuert, damit keine Variable zweimal vom externen Speichermedium gelesen werden muss. Damit sind auch schon die zwei Hauptarten der Lesezugriffe umrissen: *Zugriffe auf den externen Speicher* und *lesende Zugriffe auf einen Zwischenspeicher.*

Da ein Zugriff auf den externen Speicher immer den ersten Lesevorgang einer Variable darstellt, soll diese Art des Zugriffes hier als erstes untersucht werden.

Zugriff auf den externen Speicher Der Zugriff auf den externen Speicher erfolgt, wenn der Transaktionsmanager noch keinen Wert eines Parameters im Zwischenspeicher hält. Dies ist ausschließlich der Fall, wenn das Netz eine externe Variable erstmals liest. Der Zugriff auf diese Variable wird nicht im Transaktionsmanager bearbeitet, sondern muss das gesamte DAF durchlaufen. Das Resultat der Abarbeitung wird im Zwischenspeicher gespeichert und steht anschließend allen folgenden Tasks zur Verfügung.

Zugriffe auf den Zwischenspeicher Sollte der von YAWL angeforderte Parameter schon im Zwischenspeicher vorhanden sein, der Wert ist also im aktuell durchsuchten Pfad oder im

Hauptpfad hinterlegt, wird der Wert aus dem Zwischenspeicher gelesen. Die Bearbeitung der Chain wird dann im Transaktionsmanager gestoppt. In der PostProcess-Methode der Chain wird der Wert aus dem Zwischenspeicher gelesen und in das Kontextobjekt überführt

4.3.8 Schreibzugriffe

Wie im Kapitel Techniken beschrieben, wird der Transaktionsmanager das Modell der Offen geschachtelten Transaktionen verfolgen. Schreibzugriffe werden mit *Commit* der Subtransaktion *(=transaktionaler Sphäre)* an alle Subtransaktionen und der Vatertransaktion publiziert. Vor dem Commit ist jede (Sub-)Transaktion gegeneinander isoliert.

Während der Ausführung eines Cases darf eine Variable nur einmal in den persistenten Speicher geschrieben werden. Das bedeutet, dass die Schreibzugriffe durch den Transaktionsmanager ebenfalls verzögert werden. Bei einem Schreibzugriff innerhalb einer transaktionalen Sphäre eines Cases werden die Werte im Zwischenspeicher gespeichert und stehen den folgenden Tasks der Sphäre in der geänderten Fassung zur Verfügung.

Die geänderten Werte sollen mit dem *commit* (Kapitel 4.3.9) der transaktionalen Sphäre in den persistenten Speicher überführt werden. Die Ergebnisse werden mit dem `commit` anderen parallel ausgeführten transaktionalen Sphären zugänglich gemacht.

4.3.9 commit - Ende einer transaktionalen Sphäre

Das Ende einer transaktionalen Sphäre muss durch den Transaktionsmanager erkannt und behandelt werden. Da der Transaktionsmanager durch YAWL nur aufgerufen wird, wenn auf einen externen Parameter zugegriffen wird, muss eine Möglichkeit gefunden werden, den Transaktionsmanager zum Abschluss eines Workflows zu starten und das `commit` der Transaktion durchzuführen.

Beim `commit` der transaktionalen Sphäre werden die innerhalb der Sphäre veränderten Parameter in den persistenten Speicher geschrieben. Dazu könnte ein *commit*-Attribut im Kontextobjekt durch die Chain gereicht werden, welches dem PluginManager das persistente Speichern der veränderten Parameter erlaubt.

Sollte die Sphäre Teil einer anderen transaktionalen Sphäre sein, werden die Ergebnisse in die Vatersphäre publiziert und somit der Vatersphäre zugänglich gemacht.

4.3.10 abort - Verhalten im Fehlerfall

Die Ausführung der transaktionalen Sphäre kann aus verschiedenen Gründe vorzeitig beendet (*abort*) werden. Wenn ein Lesevorgang durch den Pluginmanager von tx+YAWL nicht erfolgreich durchgeführt werden konnte, muss die Ausführung der transaktionalen Sphäre abgebrochen werden. In diesem Fall sollte der Transaktionmanager eine Fehlernachricht generieren und an den Data Source Manager von tx+YAWL weiterreichen.

Zusätzlich muss der Transaktionsmanager den für diesen Workcase angelegten Zwischenspeicher löschen. Da noch keine Daten in den persistenten Speicher geschrieben wurden, muss keine Aktion innerhalb des DAFs gestartet werden, um die Änderungen rückgängig zu machen.

4.4 Datenhaltung

Der Transaktionsmanager muss die eingelesenen und veränderten Parameter in einem Zwischenspeicher halten, um diese in späteren Tasks wiederverwenden zu können. Das DAF wird in jedem Lesevorgang als Webapplet aktiviert und ausgeführt. Nach dieser Ausführung wird das Applet wieder deaktiviert, das heisst sämtliche im Zwischenspeicher gespeicherte Daten würden verloren gehen. Der Zwischenspeicher muss persistent gestaltet sein, um die Daten Task- und Parameterübergreifend zur Verfügung stellen zu können. In dieser Sektion soll dargestellt werden, welche Speicherart für den Transaktionsmanager von tx+YAWL gewählt wurde.

4.4.1 Verschiedene Datenhaltungsmethoden

Zur Umsetzung des persistenten Speichers soll hier nun ein Vergleich zwischen einer Lösung mit MySQL, Java Concurrent Hashtree und dem Datenhaltungssystem Infinispan aufgeführt werden.

MySQL bietet ein robustes Datenbanksystem welches auch von YAWL genutzt und ohne zusätzlichen Aufwand konfiguriert werden kann. MySQL ist eine Open Source Datenbank, die mehrere Speicherengines unterstützt und über den JDBC Connector angesprochen wird.

Infinispan basiert auf JBoss Cache 3.x, ohne dass dadurch jedoch Abhängigkeiten entstehen, die eine Referenzierung und Implementierung von JBoss Cache 3.X erforderlich machen. Infinispan ist wie JBoss TS eine Entwicklung, die durch die JBoss Community durchgeführt wird und unter LGPL steht. Es kann als eine Datenhaltungsschicht angesehen werden. Infinispan hat eine reichhaltige Dokumentation in Form einer Online-Doku und eine sehr aktive Community.

Java Concurrent Hashtable ist in Java fest eingebaut und bietet eine ähnliche Funktionalität wie Infinispan, scheidet aber aufgrund eines entscheidenden Nachteils aus. Es gibt keine einge-

baute Funktion, um Daten auf die Festplatte zu übertragen und den Zwischenspeicher damit persistent zu gestalten.

4.4.2 Auswahl der Zwischenspeicherschicht

Nach dem Vergleich der drei Alternativen wurde entschieden, dass Infinispan die prototypische Zwischenspeicherschicht für den Transaktionsmanager darstellen wird. Infinispan bietet einige Vorteile, die es gegenüber MySQL für die prototypische Implementierung prädestiniert.

Speichern von Java Objekten Infinispan hat durch die Einbindung direkt in den JAVA Code die Möglichkeit, JAVA Objekte zu speichern. Dies bereitet dem Transaktionsmanager einen erhöhten Grad an Flexibilität, da nun jeder Datentyp im Zwischenspeicher gehalten werden kann. Diese wäre mit MySQL nicht möglich gewesen.

Skalierbarkeit Infinispan bietet die Möglichkeit eine hochskalierbare Datenhaltungsschicht zu nutzen, die auch mehrere Rechnerknoten umfassen kann.

MVCC Infinispan nutzt MVCC.

4.5 Recovery

Obgleich eine Recoverykomponente nicht Teil dieser prototypischen Implementierung ist, sollen an dieser Stelle einige Gedanken auf mögliche Strategien gerichtet werden. Die Recoverykomponente ist eine dem Transaktionsmanager nachgeschaltete Komponente. Sie kann im Fall eines Fehlers die Daten aus einem inkonsistenten oder sogar fehlerhaften Zustand wieder in einen konsistenten Zustand versetzen.

4.5.1 Rollback

Undo Eine mögliche Variante, um die Recovery zu realisieren, ist der sogenannte Rollback. Diese Variante ist aus dem Datenbankenbereich bekannt. Bei einem Rollback werden sämtliche geänderten Werte wieder auf ihren Ursprungswert zurückgesetzt. Nachdem die Werte zurückgesetzt sind, kann der Workcase erneut gestartet werden.

Dafür müssen Logdateien eingeführt werden, die den Status der Parameter eines Workcases vor (Before-Log) Veränderungen loggen. Bei jeder Aktion einer Transaktion (*schreiben, lesen, commit, abort*) muss ein Eintrag in die Logdatei erfolgen. Dieser Eintrag beinhaltet jeweils den

aktuellen **unveränderten** Wert und eine Identifikation, um die Transaktion im Log identifizieren zu können.

Diese Logdateien müssen im Recoverymanager geführt werden. In einem möglichen Fehlerfall kann der Recoverymanager die Schritte der Logdatei rückwärts durchlaufen, die Änderungen rückgängig machen und den ursprünglichen Wert der Parameter wiederherstellen. Nach Abschluss des Rollbacks können die betroffenen Workcases erneut gestartet werden.

Redo Eine andere aus dem Datenbankumfeld bekannte Rollbackmethode ist das *REDO*. Beim Redo wird die aktuelle Transaktion auf den Start zurückgesetzt und Stück für Stück erneut ausgeführt. Dafür ist es notwendig, dass die veränderten Werte eines jeden Tasks gespeichert werden. Die Logdatei muss den Status der Parameter des Workcases nach ihrer Veränderung (*After-Log*) loggen. Die Tasks des Workcases könnten dann vom ersten Task des Cases bis zum letzten ausgeführten Task mit den gespeicherten Werten ausgeführt werden.

Mit der Einführung einer globalen Zwischenspeicherebene im Recoverymanager könnte er Schreibaktionen auf die Parameter kontrollieren und zusätzlich eine workcaseübergreifende Datenbasis für die Wiederherstellung anbieten.

4.5.2 Fortsetzung der Aktion

Eine andere mögliche Variante zur Recovery wäre die Fortsetzung des Workcases an der Stelle, an welcher er abgebrochen wurde. Infinispan hat eine eingebaute Recoverykomponente und speichert die Daten des aktuellen Workcases persistent. Durch das ebenfalls gespeicherte Log können die bisher ausgeführten Arbeitsschritte erneut ausgeführt werden und nahtlos an den abgebrochenen Workcase anknüpfen.

5 prototypische Implementierung

Basierend auf dem im vorigen Kapitel entwickeltem konzeptionellen Transaktionsmanager soll in diesem Kapitel nun ein Prototyp implementiert werden. Da diese Arbeit Teil eines größeren Projektes ist, wird der entwickelte Transaktionsmanager zuerst ins tx+YAWL Framework eingefügt, bevor die einzelnen Komponenten des Transaktionsmanagers vorgestellt werden.

Der hier entwickelte Prototyp soll in das DAF eingebunden werden und Operationen wie das Lesen und Schreiben in einen Zwischenspeicher umsetzen. Im Verlauf des Kapitel soll die Eingliederung des Transaktionsmanagers in das DAF und die damit verbundenen Änderungen an der Datenstruktur des DAF beschrieben werden. Anschließend werden die Eingaben, die der

Transaktionsmanager benötigt, im Subkapitel 5.1.1 dargestellt. Im Kapitel 5.2 wird das umgesetzte Transaktionsmodul vorgestellt.

Die Hauptkomponenten des Transaktionsmanagers werden im Kapitel 5.3 dargestellt.

5.1 Einbindung in das DAF

Der entwickelte Transaktionsmanager fungiert als ein Bauteil des Data Access Frameworks. Er wird bei transaktionalen Sphären hinzugeschaltet. Dafür ist es nötig, dass der Transaktionsmanager in die Apache Chain integriert wird.

Mit dem Erstellen der `TransactionService`-JAVA Datei im `org.yawlfoundation.yawl.daf.chain` Package des DAF und dem Erstellen einer Referenz auf diese Datei in der Chain-Konfigurationsdatei `chain-config.xml` ist das Transaktionsmodul in die Chain eingefügt.

```
1 <catalog>
2   <chain name="readOnly">
3     <command id="CP"
4         className="org.yawlfoundation.yawl.daf.chain.ConnectionService"/>
5     <command id="TM"
6         className="org.yawlfoundation.yawl.daf.chain.TransactionService"/>
7     <command id="CS"
8         className="org.yawlfoundation.yawl.daf.chain.ConstraintService"/>
9     <command id="PI"
10        className="org.yawlfoundation.yawl.daf.chain.PlugInManger"/>
11  </chain>
12 </catalog>
```

Listing 1: Die Konfigurationsdatei mit dem Transaktionsservice

Nach dieser Änderung wird der Transaktionsservice beim Aufruf eines externen Parameters aufgerufen und ausgeführt.

5.1.1 Eingaben der Chain an den Transaktionsmanager

Die Apache Chain bietet ein Kontext-Objekt, welches durch die verschiedenen Instanzen der Chain gereicht wird. Aus dem Kontextobjekt können alle Kettenglieder Informationen beziehen und zurückschreiben. Es wird im *Data Source Manager* mit Informationen über den aktuellen Task und Workcase instanziiert.

Der Transaktionsmanager wird die benötigten Informationen aus dem Kontextobjekt beziehen und die von ihm generierten Resultate in ein Objekt innerhalb des Kontextobjektes speichern.

Der Transaktionsmanager benötigt verschiedene in der Urpsrungsversion von tx+YAWL nicht vorhandene Informationen. Zu diesen Informationen zählen die *WorkCase-ID*, der *Taskname* und der *Parametername*.

Der Zwischenspeicher des Cases wird über einer Kennung adressiert und persistent auf der Festplatte gespeichert. Diese Kennung setzt sich aus der *Workcase-ID* und dem Namen der aktuellen transaktionalen Sphäre zusammen. Die Nutzung der WorkCase-ID ist nötig, da verschiedene Instanzen eines Workcases sonst nicht differenzierbar wären.

Mit dem *Taskname* kann der Transaktionsmanager die Propertydatei durchsuchen und Informationen über diesen Task extrahieren. Zu diesen Informationen zählen beispielsweise der Name der transaktionalen Sphäre, in der sich der Task befindet und etwaige Tasks mit denen er synchronisiert werden muss.

In der derzeitigen Version von tx+YAWL wird die Chain pro Parameter eines Taskes separat durchlaufen. Der *Parametername*, kombiniert mit den Informationen des genutzten Plugins (Adressierung der Daten), ist nötig, um die Daten aus dem persistenten Speicher zu akquirieren.

Der Data Source Manager wurde um die Funktion `addTMtoCTX` erweitert, um die CaseID, den Parameternamen und den Tasknamen in das Kontextobjekt zu überführen.

```
1  private void addTMtoCTX(String taskName, String paramName,
2      int caseID, Context ctx) {
3    ctx.put("caseID", caseID);
4    ctx.put("paramName", paramName);
5    ctx.put("taskName", taskName);
6  }
```

Listing 2: Hinzufügen von Kontextinformationen

Andere benötigte Informationen wie die bereits angesprochene Plugin-Adressierung werden durch den Data Source Manager in das Kontextobjekt geschrieben und stehen jedem Glied der Chain zur Verfügung.

5.1.2 Veränderungen am DAF

Zur Bereitstellung der benötigten Informationen im DSM wurde der DataGateway und das Task-Objekt von YAWL angepasst. Die im Listing 2 aufgeführte Funktion wurde in den DataSourceManager hinzugefügt. Da YAWL standardmäßig keine Möglichkeit bot, um die CaseID auszulesen, musste für diese Arbeit eine Erweitung des Task-Objektes vorgenommen werden, die

die CaseID zurückgibt.

```
1  public class DAF_Task {
2    private YTask yTask;
3    public DAF_Task( YTask task)
4    {
5      yTask = task;
6    }
7    public String get_i ()
8    {
9      // In case that the net contains a composite task:
10     // split the number so that e.g. the number 38.3 gets to be CaseID 38
11     String[] parts = yTask._i.toString().split(".");
12     return parts[0];
13   }
14 }
```

Listing 3: Einführung des DAF Tasks

Im Listing 3 ist der eingeführte DAF-Task abgebildet. Er besteht aus einer Methode, die den ersten Teil der CaseID zurückliefert.

Bei der Umsetzung des Datentyps war allerdings ein Spezialfall zu beachten. Wenn ein *Netz 3* einen *Composite Task* enthält, wird dieser *Composite Task* durch YAWL als Subnetz des Netzes betrachtet. YAWL vergibt bei Auftreten eines Subnetzes eine eigene Identifikationsnummer. Der Composite Task und seine Zerlegung würden also die Identifikationsnummer *3.1* erhalten.

Der Zwischenspeichername setzt sich aus der WorkCaseID und dem Namen der transaktionalen Sphäre zusammen. Im Falle eines Composite Tasks würde ein neuer Zwischenspeicher angelegt werden. Dies würde dazu führen, dass n-fache Lese- und Schreibzugriffe auf die Parameter innerhalb einer transaktionalen Sphäre durchgeführt werden. Das Read- und Writeset von tx+YAWL würde verletzt werden. Die in Listing 3 gezeigte Funktion löst dieses Problem, indem sie nur die Zahl vor dem Punkt beachtet.

5.2 Transaktionsmodul

Das Transaktionsmodul ist neben dem Transaktionsmanager Hauptbestandteil dieser Arbeit. Es ist direkt im *Data Access Framework* integriert und bildet eine Brücke zu den Funktionalitäten des Transaktionsmanagers. Das Modul aktiviert den Zwischenspeicher und steuert die Konfiguration und Abarbeitung des Workcases.

5.2.1 Propertydatei

In Kapitel 4.3.2 wurde eine mögliche konzeptionelle Methode zur Speicherung der Strukturinformationen der Workcases vorgestellt. In dem hier vorgestellten Prototypen werden die Strukturinformationen in einer *JAVA-Propertydatei* auf dem Datenträger gespeichert und beim Aufruf des Transaktionsmoduls geladen.

Die Propertydatei muss nach einem nachfolgend genannten Schema aufgebaut sein, um durch die Konfiguration und Analyse richtig erfasst zu werden. Eine kommaseparierte Liste von Sphären (*spheres*) markiert den Anfang der Propertydatei. Anschließend werden die Attribute der in *spheres* aufgeführten Sphären aufgeführt. Darauffolgend werden im Punkt *Tasks* alle in dem Workcase vorkommenden Task kommasepariert aufgezählt, bevor den Tasks anschließend Attribute zugeordnert werden können.

5.2.2 Eingeführte Datentypen

Zur Speicherung der in der Propertydatei hinterlegten Strukturinformationen wurden die zwei Datentypen *Task* und *Sphere* eingeführt. Diese halten die in der Analyse des Workcases extrahierten Informationen und stellen sie den anderen Komponenten des Transaktionsmoduls zur Verfügung.

Sphere Dieses Objekt hält die Informationen über sämtliche in dem Workcase enthaltenen Sphären. Zu diesen Informationen gehören beispielsweise der Name der Sphäre, der Name einer etwaigen Vatersphäre, eine Auflistung der in der Sphäre vorhandenen Tasks und abschließend eine Auflistung der letzten Tasks, die in dieser Sphäre durch das DAF ausgeführt werden. Diese Auflistung ist nötig, da die Sphäre die Resultate ihrer Ausführung mit Beendigung an eine mögliche Vatersphäre oder an den persistenten Speicher publizieren soll. Da das DAF aber nur aktiviert wird, wenn ein externer Parameter gelesen wird, muss das Rückschreiben teilweise noch vor dem eigentlichen Abschluss der transaktionalen Sphäre erfolgen.

In Abbildung 12 ist der strukturelle Aufbau der Sphere-Klasse mit Attributen und den bereitgestellen Operationen abgebildet. Auf die Methode *deleteTaskFromFinalSet* wird in Kapitel 5.2.6 näher eingegangen. Diese steuert indirekt, wann eine Sphäre ihre Resultate in die Vatersphäre oder den persistenten Speicher zurückschreibt.

Task Im Taskobjekt (siehe Abbildung 12) werden Informationen über einen Task gespeichert. Zu diesen Informationen zählen der Namen, der Pfad, das Writeset, ob er der Task ein *Combi-*

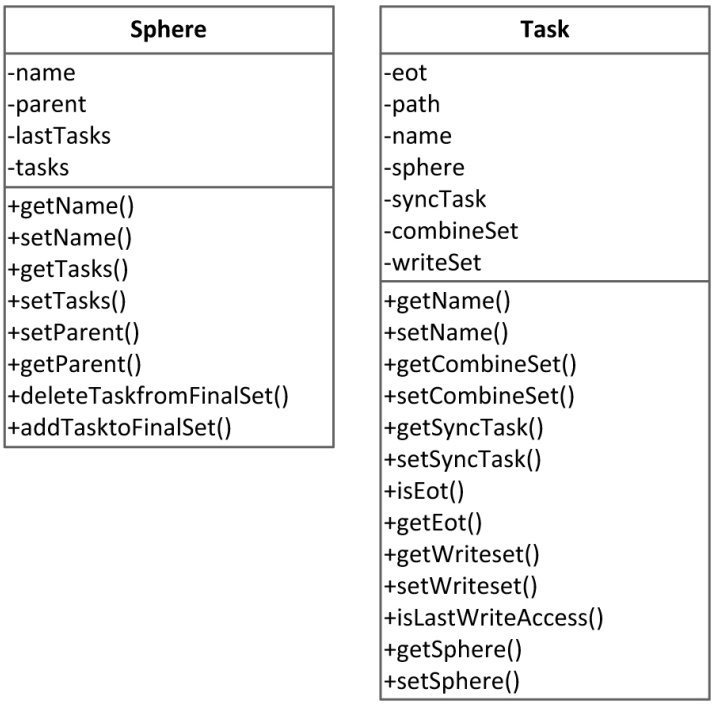

Abbildung 12: Die eingeführten Datentypen des Transaktionsservices

neSet enthält oder gegenüber anderen Tasks synchronisiert werden muss. Zusätzlich setzt das Taskobjekt ein Flag, wenn der Task der potentiell letzte Task einer Sphäre ist und den Namen der Sphäre, in welcher er enthalten ist.

Die Methode *islastWriteAccess* gibt `true` zurück wenn der Task den letzten Schreibzugriff auf einen Parameter durchführt. In diesem wird bei Beendigung des Tasks der Wert des Parameters in den persistenten Speicher zurückgeschrieben.

Die Einführung des *WriteSets* im Taskobjekt wurde nötig, weil tx+YAWL derzeit noch keine Massenschreibvorgänge durchführen kann. In der prototypischen Implementierung muss ein Task den Parameter in die Datenquelle schreiben, wenn er der Task ist, der den letzten schreibenden Zugriff auf diesen Parameter durchführt - ansonsten würden die benötigten Parameteridentifikationen verloren gehen.

5.2.3 Konfiguration des Transaktionsmanagers

Mit dem Laden der Propertydatei startet der Konfigurationsprozess des Transaktionsmanagers durch Ausführung von zwei Funktionen, die die transaktionalen Sphären und die Tasks des Workcases aus der Propertydatei extrahieren und in den oben genannten Datenstrukturen spei-

chern. Die im Workcase vorhandenen Sphären werden ausgelesen und in einer Map mit dem Key *Name* und Value *Sphereobjekt* gespeichert. Nach der Konfiguration der Sphären werden die in der Propertydatei vorhandenen Tasks durchlaufen und in einer Map gespeichert.

Ähnlich wie die Sphären wird auch hier der Name des Tasks als Key und ein Taskobjekt als Value verwendet. Sollte der Task ein potentiell letztes Element einer Sphäre darstellen, wird er in die *LastTasks*-Liste der betreffenden Sphäre hinzugefügt.

5.2.4 Zwischenspeicherinitialisierung

Bei der Zwischenspeicherinitialisierung wird ein Name aus der *CaseID* und dem *Sphärennamen* generiert. Mithilfe dieses Namens kann der Cachemanager von Infinispan den Cache von der Festplatte laden, anlegen und speichern. Das Anlegen oder Laden wird durch Infinispan implizit ausgeführt. Wenn bisher kein Cache unter dem Namen existierte, wird er angelegt. Ist der Cache bereits in einem vorhergehenden Durchlauf angelegt worden, wird er geladen. Anschließend wird überprüft, ob der Cache bereits Datensätze enthält.

Um die LastTasks-Liste auch über mehrere Ausführungen des DAFs persistent zu speichern, muss diese Liste im Infinispancache gespeichert werden.

Sollten noch keine Werte von Parametern im Cache gespeichert sein, wird davon ausgegangen, dass der Cache neu angelegt wurde. In diesem Fall wird die *LastTasks-Liste* aus der Analyse im Cache hinterlegt.

Sollten schon Werte im Cache gespeichert sein, wurde der Cache bei einem vorherigem Lese-/Schreibvorgang schon initialisert. Die *LastTasks-Liste* für die betroffene Sphäre wird dann aus dem Cache geladen und weiterverwendet.

5.2.5 Lese/Schreibzugriffe

Nach einer Abfrage, ob der aktuell zu bearbeitende Task in einer transaktionalen Sphäre liegt, wird bei einer positiven Antwort die weitere Verarbeitung innerhalb des Transaktionsmanagers angestoßen. Sollte der Task nicht in einer transaktionalen Sphäre liegen, wird kein transaktionales Verhalten für diesen Task angeboten. Der Zugriff erfolgt also ohne Beteiligung des Transaktionsmanagers.

Wenn der Task einen oder mehrere Einträge im Synchronisierungsattribut hat, wird in diesem prototypischen TM eine Nachricht ausgegeben. In einer zukünftigen Version könnte hier eine

Sequentialisierung der Abarbeitung vorgenommen werden.

Nach der etwaigen Sequentialisierung des Tasks wird überprüft, ob der Task ein *CombineSet* hat. Sollte ein CombineSet vorhanden sein, wird für jeden Parameter der in dieser Auflistung vorhanden ist, die Kombinierungsmethode der TransaktionsAPI aufgerufen und die Pfade zusammengeführt.

Als nächstes wird im Kontextobjekt überprüft, ob ein lesender oder schreibender Zugriff auf den Parameter erfolgen soll.

lesender Zugriff Bei einem lesenden Zugriff muss das *ReadSet* eingehalten werden. Dies bedeutet, dass ein Parameter nur einmal pro transaktionaler Sphäre aus der externen Quelle gelesen werden darf.

Diese Vorraussetzung wird durch den Transaktionsmanager implizit erfüllt: Bei einem lesenden Zugriff auf einen Parameter wird überprüft, ob der Parameter bereits im Zwischenspeicher vorhanden ist. Es wird versucht, den Wert unter Angabe des aktuellen Pfades mit der API-Funktion *getLatestValue* zu lesen. Sollte für den Parameter bereits ein Wert im Zwischenspeicher hinterlegt sein, wird dieser Wert zurückgegeben und die Variable *internalRead*[6] auf `true` gesetzt.

Wenn noch kein Wert im Zwischenspeicher hinterlegt ist, gibt die Funktion *getLatestValue null* zurück. Der Transaktionsmanager führt einen Lesezugriff auf diesen Parameter mit dem Standardpfad 0 durch. Sollte auch hier der Rückgabewert *null* sein, wird der aktuelle Pfad des Tasks auf *0* gesetzt und die Bearbeitung durch das DAF wird fortgesetzt.

Der lesenden Zugriff wird auf den persistenten Speicher ausgeführt und in der PostProcess-Methode (Kapitel 5.2.6) werden die Resultate der tieferliegenden Ebene in den Zwischenspeicher geschrieben. In Abbildung 13 ist der Lesevorgang grafisch aufbereitet.

schreibender Zugriff Ein schreibender Zugriff auf einen Parameter wird durch den Transaktionsservice verarbeitet und im Zwischenspeicher gespeichert, wenn der Parameter nicht im *writeSet* des aktuellen Tasks aufgeführt ist. Ist der Parameter im *writeSet* des Tasks enthalten ist wird der Wert des Parameters durch die Chain weitergereicht und in den persistenten Speicher geschrieben.

[6] Die Variable *internalRead* identifiziert Lesezugriffe die durch den Zwischenspeicher durchgeführt werden.

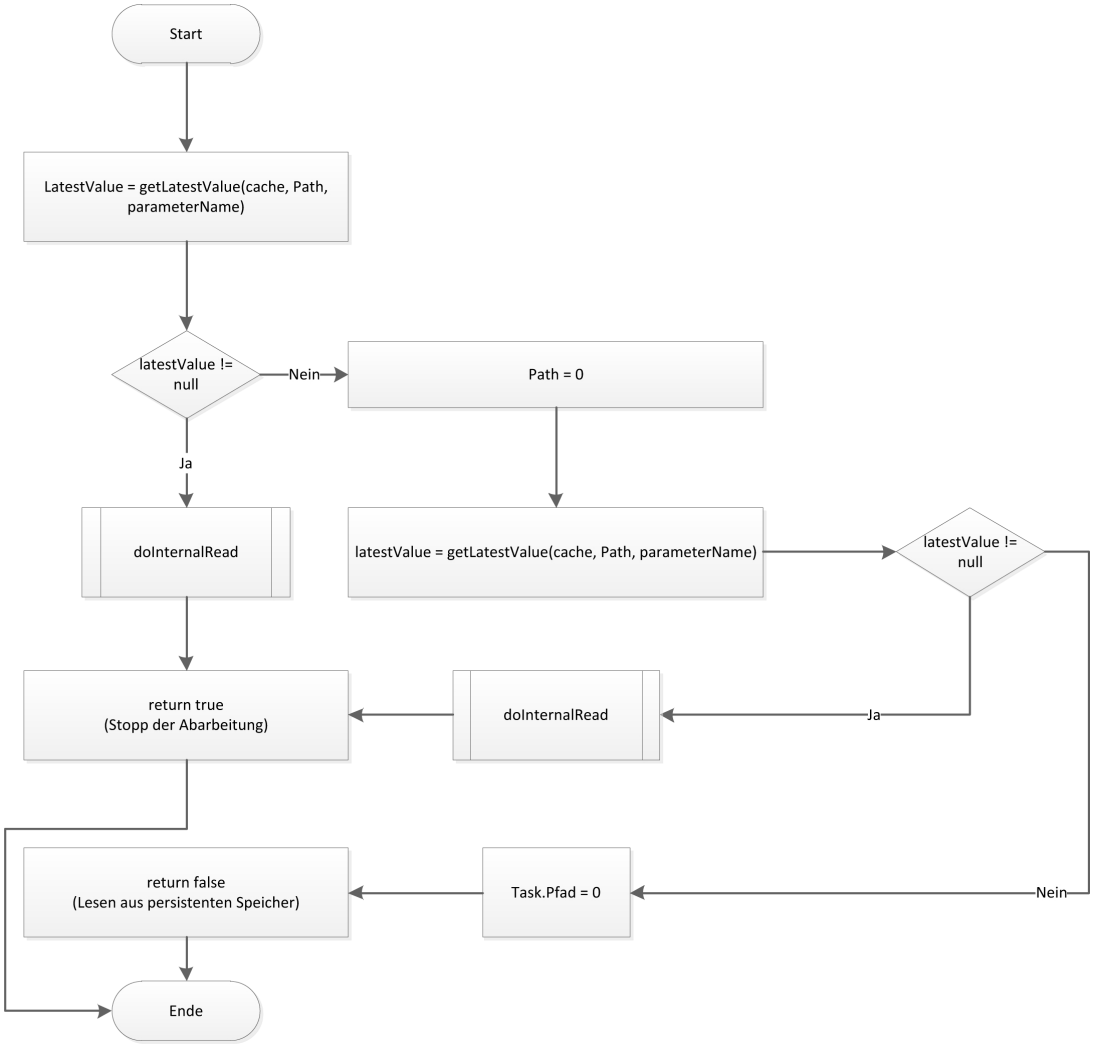

Abbildung 13: Lesevorgang im Transactionsservice

Wenn der Parameter in den Zwischenspeicher geschrieben werden soll, schreibt der Transaktionsmanager den aktuell übergebenen Parameterwert gekapselt in ein *Result*-Objekt, in das Kontextobjekt. Dieser wird in der PostProcess Methode (Kapitel 5.2.6) in den Zwischenspeicher geschrieben.

Wenn der Wert in den persistenten Speicher geschrieben wurde, wird er in der PostProcess Methode aus dem Kontext geladen und nochmals in den Zwischenspeicher geschrieben, um ihn für etwaige spätere lesende Zugriffe aus dem Cache laden zu können.

5.2.6 PostProcess Methode

Die PostProcess Methode wird aufgerufen, nachdem ein Wert erfolgreich durch die Chain verarbeitet wurde. In dieser Methode werden die Resultate der Verarbeitung aufgearbeitet. Wenn sich der aktuelle Task in einer transaktionalen Sphäre befindet, wird mit der `internalRead`-Variable überprüft, ob der Lesevorgang durch den Transaktionsservice bearbeitet wurde.

Sollte der Wert nicht durch den Transaktionsservice aus dem Zwischenspeicher gelesen worden sein, wird der Wert des Parameters aus dem Kontext gelesen und in den Zwischenspeicher unter Angabe seines Pfades und dem Parameternamen geschrieben. Dafür wird die API Funktion `addNewValue` genutzt.

Vor Abschluss der Methode muss überprüft werden, ob dieser Task der letzte Task einer transaktionalen Sphäre ist. Hierfür wird untersucht, ob das *EOT* Attribut des Tasks gesetzt ist und die Anzahl der Tasks in der *lastTasks* Liste gleich *1* ist.

Sollte die Anzahl der Tasks gleich *1* sein, ist der letzte Task der transaktionalen Sphäre erreicht. Wenn die aktuelle transaktionale Sphäre eine Vatersphäre besitzt, werden die Resultate dieser Sphäre in die Vatersphäre geschrieben (*finalizeSubCache*).

Wenn das *EOT*-Attribut des Tasks nicht gesetzt ist oder die Anzahl der Tasks in der LastTasks-Liste nicht gleich *1* ist, wird der Name des Tasks aus der *LastTasks*-Liste gestrichen und die Anzahl der Tasks damit dekrementiert. Die aktualisierte LastTasks-Liste wird im Zwischenspeicher persistent abgespeichert.

5.3 Datenzugriff

Dieses Kapitel wird die tiefste Ebene des entwickelten Transaktionsmanagers behandeln. Der Transaktionsmanager entscheidet in dieser Ebene, wo der Wert eines Parameters gespeichert wird und führt ein Log über Zugriffe auf den persistenten Speicher. Gesteuert wird der Transaktionsmanger durch API-Aufrufe des Transaktionsservices.

Intern werden die Aufrufe durch eine modifizierte dreiteilige Decoratorstruktur behandelt, die aus den Komponenten API, Log und Ressourcemanager besteht. Die Decorators werden im Kapitel 5.3.2 detaillierter dargestellt. Sie werden gegen ein gemeinsames Interface implementiert, welches im nächsten Kapitel 5.3.1 offengelegt wird.

5.3.1 Transaktions-API

Ein Ziel bei der Entwicklung des Transaktionsmanager war, dass er skalierbar, leicht erweiter- und einsetzbar ist. Diesem Ziel wurde Rechnung getragen, indem die Funktionalität des Transaktionsmanagers durch eine API gekapselt wird. Durch die Kapselung konnte der Transaktionsmanager leicht in die vorhandene Architektur integriert werden.

getLatestValue Diese Funktion soll den aktuellsten Wert eines Parameters eines Pfades zurückgeben. Als Eingabeparameter werden der *Infinispan Cache*, die *Pfadnummer* und der *Parametername* übertragen.

Die Funktion gibt den Wert des Parameters zurück, wenn bereits ein Wert für diesen Pfad angelegt wurde. Sollte kein Wert für diesen Pfad gespeichert sein, wird `null` zurückgegeben. Das Rückgabeobjekt der Funktion ist ein *Object*[7] um dem Umstand Rechnung zu tragen, dass der Wert einen jeden Datentyp annehmen kann.

addNewValue Fügt dem Zwischenspeicher einen neuen Wert eines Parameters hinzu und legt eine neue Version des Parameters an. Diese Funktion hat vier Eingabeparameter:

Infinispan Cache Eine Referenz zum aktuell genutzten Zwischenspeicher

Pfad Der Pfad in welchem der aktuelle Task liegt

Parametername Der Name des zu schreibenden Parameters

Wert Der aktuelle Wert des Parameters im Typ *Object*

Sollte der Zwischenspeicher des Cases leer sein oder der Zwischenspeicher noch keinen Wert für den Parameter bereithalten wird der Parameter in den Zwischenspeicher geschrieben. Die Parameterversion wird mit 0 in der Parameterhistorie initialisiert. Andernfalls wird die aktuelle Version um 1 inkrementiert und der zu schreibende Pfad wird in der Parameterhistorie hinterlegt.

resetVariable Wenn ein Task abbricht, aber schon schreibend auf eine Variable zugegriffen hat, muss der geschriebene Wert zurückgesetzt werden, um eine konsistente Datensicht zu ermöglichen. Die Funktion benötigt hierbei folgende Eingaben:

Infinispan Cache Eine Referenz zum aktuell genutzten Zwischenspeicher

Pfad Der Pfad in welchem der aktuelle Task liegt

Parametername Der Name des zu schreibenden Parameters

[7] JAVA Object

Die Arbeitsweise der Funktion ist ähnlich gelagert wie das Hinzufügen eines neuen Wertes. Sollte der Cache leer sein oder noch kein Wert in dem Pfad stehen, wird `null` zurückgegeben, ansonsten wird der gelöschte Wert zurückgegeben.

combinePaths Wenn parallelverlaufende Pfade vereinigt werden, muss sichergestellt sein, dass sämtliche lokale Parameter der Pfade auf den Hauptpfad vereinigt und dem Rest des Cases zur Verfügung gestellt werden. Begleitend müssen in dieser Funktion auch Parameter vereinigt werden, auf die in mehreren Pfaden schreibend zugegriffen wurde. Die Funktion benötigt als Eingabeparameter:

Infinispan Cache Eine Referenz zum aktuell genutzten Zwischenspeicher

Parametername Der Name des zu schreibenden Parameters

Die aktuellste Version des Parameters in allen Pfaden wird in der Parameterhistorie angefordert und deren Wert in den Hauptpfad zurückgeschrieben.

finalizeSubCache Wenn eine transaktionale Sphäre beendet wird, sollen ihre Ergebnisse in eine etwaig vorhandene Vatersphäre übertragen werden. Die Eingabeparameter sind die folgenden:

Vaterzwischenspeicher Der Vaterzwischenspeicher und somit empfangende Speicher

KindZwischenspeicher Der Kindzwischenspeicher, der die Daten liefert

Sämtliche Parameter die innerhalb der Kindsphäre verändert wurden, werden aus dem Kindzwischenspeicher in den Vaterspeicher übertragen und dort im Hauptpfad gespeichert.

5.3.2 Decorators

Um den hier entwickelten Prototyp einfach wart- und erweiterbar zu gestalten, wurden die Hauptfunktionen des Transaktionsmanger in drei Decorators gekapselt. Durch die Nutzung des Decorator-Entwurfsmuster[8] können die Bestandteile des Transaktionsmanager leicht in der Reihenfolge getauscht oder gänzlich herausgenommen werden. Die Decorators werden gegen das in Kapitel 5.3.1 vorgestellte Interface implementiert.

In diesem Prototypen werden 3 Decorators verwendet: TransactionAPI, LoggingDecorator und der Ressourcemanager.

[8] Wikipedialink: http://de.wikipedia.org/wiki/Decorator

TransactionAPI Die oberste Ebene der Decorators dieses Prototypens stellt die TransactionAPI dar. Dieser Decorator fungiert als administrative Ebene und kontrolliert beispielsweise die Kombination von Pfaden und das Rückschreiben eines Kindzwischenspeichers in seinen Vaterspeicher. Dieser Decorator führt die Operationen *combinePaths* und *finalizeSubCache* aus.

combinePaths sucht die neueste Version des Parameters in allen Pfaden des Zwischenspeichers und fügt den Rückgabewert in den Standardpfad 0 ein. *finalizeSubCache* liest alle geänderten Parameter aus dem Subcache und fügt diese in den Vatercache mit der Operation *addNewValue* im Pfad 0 ein. Da ein Subcache pro transaktionaler Sphäre existiert und die Werte des Subcaches beim letzten Schreibzugriff schon auf den persistenten Speicher geschrieben wurden, werden die Werte im Hauptcache überschrieben.

Zugriffe auf die anderen Funktionen der API leitet die TransactionAPI auf den nächsten Decorator weiter.

LoggingDecorator Der Logging Decorator wird bei jedem schreibenden oder lesenden Zugriff auf den Ressourcemanager aufgerufen. Bei lesenden Zugriffen stellt er auf der Konsole unter Nutzung von *Log4j* den gelesenen Wert und Pfad dar. Bei schreibenden Zugriff loggt er den zu schreibenden Wert in einer LogArchitektur (Abbildung 14). Diese muss ebenfalls im Zwischenspeicher gespeichert werden.

Im LogDecorator existieren drei Logarten: Path-,Variablen- und Cachelog.

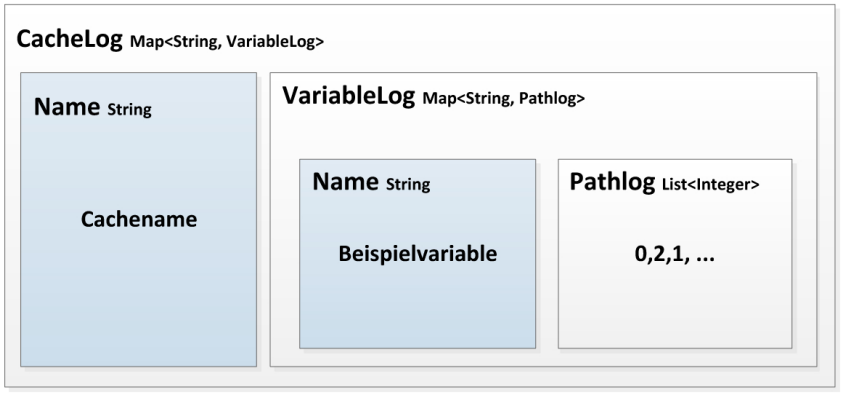

Abbildung 14: Aufbau des Cachelogs

Der Aufbau der Logs ist in Abbildung 14 dargestellt. Jeder Cache besitzt ein Cachelog. Das Cachelog ist eine Map und wird im Zwischenspeicher gespeichert. Es hat als Schlüssel den Namen

des in die Decorator übergebenen Caches und als Value das Variablenlog. Das Variablenlog ist eine Map, die als Schlüssel den Variablennamen und als Wert das Pathlog speichert. Im Pathlog stehen in einer numerischen Liste die letzten Schreibzugriffe durch die Pfade. Wenn ein Task einen Schreibzugriff auf einen Parameter durchführt, speichert das Pathlog in historischer Reihenfolge der Zugriffe, die Pfade der Tasks.

Ohne die Speicherung im Infinispancache würde der LoggingDecorator den Fortschritt in der Abarbeitung der transaktionalen Sphäre nicht mitverfolgen können. Die Zugriffe würden immer auf die erste Version des Parameters im Speicher erfolgen. Der Transaktionsmanager wäre nicht einsetzbar.

Ein Schreibvorgang auf einen Parameter im Cache ist im Listing 4 dargestellt. Wenn das CacheLog nicht leer ist und bereits einen Eintrag für den übergebenen Cache enthält, wird das Log für diesen Cache geladen. Sollte bereits ein Eintrag für diesen Parameter vorhanden sein, wird der übergebene Pfad hinten angehängt. Andernfalls wird ein neues *pathLog* mit dem aktuellen Pfad angelegt und im *CacheLog* gespeichert.

```
1  public SortedMap<Integer, Object> addNewValue(Cache cache, Integer path,
2      String variableName, Object value) {
3
4    SortedMap<String, List<Integer>> log =  new TreeMap<String, List<Integer>>();
5    _PathLog = new ArrayList();
6
7    // Create a log for write accesses on the caches if there is none to date
8    // otherwise just work with the current log
9    initializeCacheLog(cache);
10   initializeVariableLog(cache);
11
12   // check if there is already a log for the current cache
13   // if there's a log for the current cache -> write into it
14   // otherwise create a new Log for the cache
15   if(_CacheLog.isEmpty() == false)
16   {
17     //  get the current log for the write accesses on the variables and their
           corresponding paths
18     if(_CacheLog.containsKey(cache.getName()))
19     {
20       log = _CacheLog.get(cache.getName());
21
22       // if there's already a list for the history of written paths .. just append,
             otherwise add a new
23       if(log.containsKey(variableName))
24         log.get(variableName).add(path);
25       else
26       {
27         _PathLog.add(path);
28         log.put(variableName, _PathLog);
```

```
29          }
30        }
31        else
32        {
33          _PathLog.add(path);
34          log.put(variableName,_PathLog);
35          _CacheLog.put(cache.getName(), log );
36        }
37      }
38      else
39      {
40        // create a path in the pathlog
41        _PathLog.add(path);
42        // add the pathlog to the variableLog
43        _VariableLog.put(variableName,_PathLog);
44        // put everything in the cache
45        _CacheLog.put(cache.getName(), _VariableLog );
46      }
47      // save the logfiles into the cache
48      cache.put("_CacheLog", _CacheLog);
49      cache.put("_VariableLog", _VariableLog);
50      _log.debug("w "+ cache.getName() +" | Variable: "+ variableName +" | Path: "+ path
            + " | Value: "+ value);
51      return _service.addNewValue(cache, path, variableName, value);
52    }
```

Listing 4: addNewValue im LogDecorator

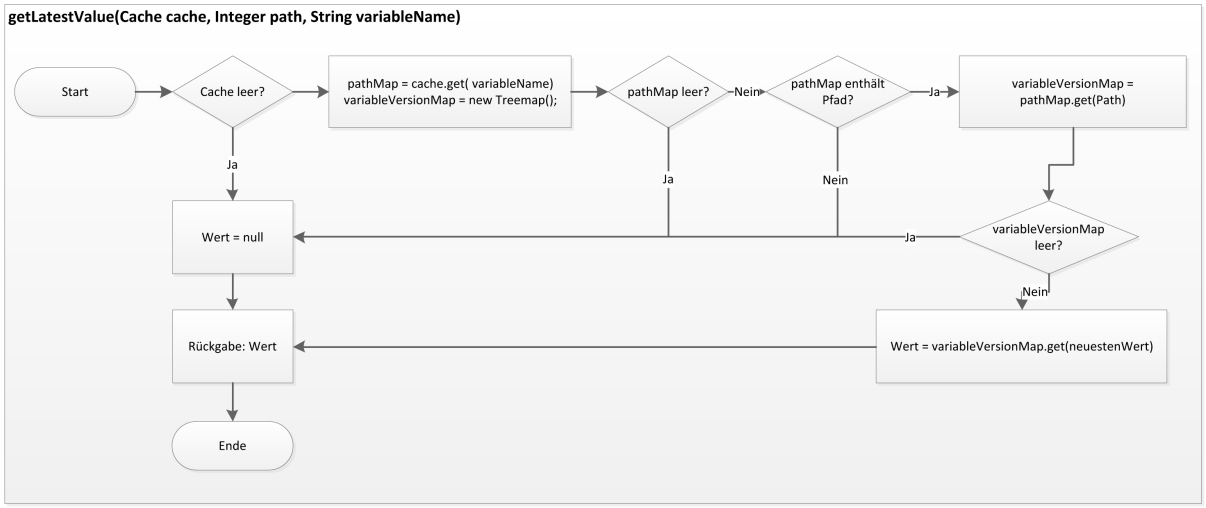

Abbildung 15: Lesen des letzten Wertes

Ressourcemanager Der Ressourcenmanager ist die tiefste Decoratorebene des entwickelten Prototyps. In dieser Ebene finden die Lese- und Schreibzugriffe auf den Zwischenspeicher statt.

Zum Ausführen der Lese- und Schreibvorgänge gibt es zwei Maps, die **pathMap**- und die **variableVersionMap**.

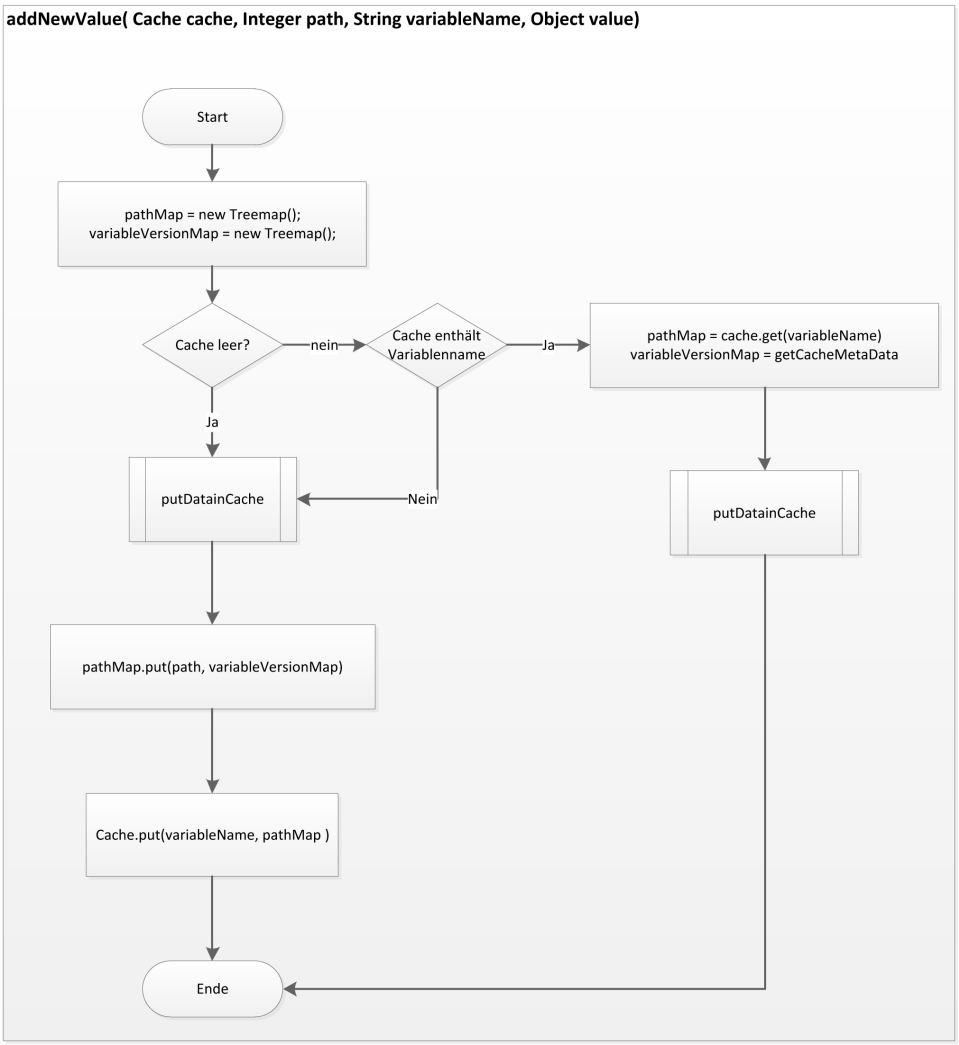

Abbildung 16: Einfügen eines neuen Wertes

Bei jedem durch das DAF ausgeführte Lese- oder Schreibzugriff werden die gelesenen oder geschrieben Werte in ihrer aktuellen Version neben den dazugehörigen Pfaden in der *pathMap* und *variableVersionMap* gespeichert.

Die *PathMap* ist eine *Map<Integer, variableVersionMap>*. Als Schlüssel fungieren die verschiedenen Pfade in denen auf den Parameter schreibend oder lesend zugegriffen wurde. Der Wert ist die *VariableVersionMap<Integer,Object>*. Die *variableVersionMap* hält für jede Version der

Variable ein eigenes Wertobjekt.

Abbildung 15 beschreibt die Durchführung eines Lesevorganges auf einen Parameter. Wenn der Cache nicht leer ist (das heißt, es wurde mindestens ein Schreib-/Lesevorgang ausgeführt) wird überprüft, ob es diesen Pfad für diesen Cache bereits gibt und ob für den übergebenen Pfad bereits eine Version angelegt wurde. Sollte dies erfüllt sein, wird der aus dem Cache gelesene Wert zurückgegeben. Wenn das nicht möglich ist, wird *null* zurückgegeben.

Ein Schreibvorgang wird durch den Ressourcemanager wie in Abbildung 16 durchgeführt. Wenn der Cache nicht leer ist, wird in der ***putDataInCache*** Methode eine neue Version der Variable in der *variableVersionMap* angelegt. Diese wird in der *PathMap* gespeichert, welche daraufhin im Infinispancache gespeichert wird. Die Funktion ***getCacheMetaData*** liefert als Rückgabewert die (neuangelegte) *variableVersionMap*, die in der ***putDataInCache*** Methode mit dem aktuellen Wert befüllt wird.

6 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung eines prototypischen Transaktionsmanagers und die Eingliederung des selbigen in die Apache Chain des DAF. Zur Erreichung dieses Ziels sollten bestehende Transaktionsmanager verglichen und nach Möglichkeit in das System integriert werden. der Größe und des Umfanges der bestehenden Transaktionsmanager und des daraus folgenden Aufwandes der Anpassungsarbeiten wurde entschieden, einen eigenen Transaktionsmanager zu entwickeln.

Als Hauptbestandteile dieser Arbeit wurden die TransaktionsAPI, der Zwischenspeicher und die Konfiguration sowie Ausführung des Transaktionsservice identifiziert. Als Zwischenspeicherlösung wurde mit Infinispan eine bereits fertig implementierte Lösung genutzt.

Die Funktionalität des Transaktionsmanagers wird dem *Data Access Framework* über eine API zur Verfügung gestellt. Durch die Nutzung des Decoratorpatterns in der API kann die Funktionalität leicht erweitert und angepasst werden.

Der entwickelte Prototyp besitzt drei Decorators: Transactionservice, Logdecorator und Ressourcemanager. Der *Ressourcemanager* führt die durch den *Logdecorator* geloggten, Lese- und Schreibzugriffe auf den Zwischenspeicher durch, während der *Transactionservice* Pfade und die Inhalte transaktionaler Sphären zusammenführt.

Der entwickelte Transaktionsmanager kann durch die Strukturinformationsdatei transaktionale Sphären erkennen und Tasks Sphären zuordnen. Er kontrolliert den Zugriff auf externe Parameter einer transaktionalen Sphäre unter Einhaltung des *Read-* und *WriteSets*. Dies war eine der Grundvorraussetzungen, die in der Aufgabenstellung gefordert wurde. Zur Erfüllung dieser Voraussetzung werden die Werte der Parameter in dem eingeführten Zwischenspeicher hinterlegt und bei Bedarf erneut aus diesem gelesen. Wenn auf einen Parameter das letzte Mal schreibend innerhalb einer transaktionalen Sphäre zugegriffen wird, erkennt das der Transaktionsmanager und schreibt den Wert des Parameters unter Nutzung des DAFs in den korrespondierenden persistenten Speicher.

Des Weiteren erfolgen die Zugriffe auf die externen Parameter innerhalb einer transaktionalen Sphäre aufgrund von transaktionalen Grundprinzipien. Die Zugriffe werden gegeneinander isoliert.

Zusätzlich zu dieser Grundfunktionalität setzt der Transaktionamanager auch andere fortgeschrittene Funktionen um. So kann er bei Beendigung einer transaktionalen Sphäre die Inhalte, sprich Resultate, der Subsphäre in die Hauptsphäre übertragen und stellt damit Datenkonsistenz zwischen den verschiedenen Sphären her. Diese Funktion ist im derzeitigen Entwicklungsstand von tx+YAWL jedoch derzeit nicht ausführbar, da die YAWL Modellierungstools noch keine transaktionale Sphäre modellieren können.

Werden innerhalb eines Workflows parallele Abarbeitungen mit Zugriffen auf verschiedene Parameter durchgeführt, kann der Transaktionsmanager diese Pfade kombinieren und den aktuellsten Wert in den korrespondierenden Hauptpfad des Workflows schreiben.

Zusammenfassend kann festgehalten werden, dass in dieser Arbeit ein prototypische Transaktionsmanager entwickelt wurde, der die Anforderungen der Zielsetzung umsetzt.

7 Zukünftige Arbeiten

Im Rahmen dieser Ausarbeitung war es leider nicht möglich, das Prinzip der *Offen geschachtelten Transaktionen* in den Transaktionsmanager zu integrieren. tx+YAWL bietet derzeit noch keine Möglichkeit, verschiedene Parameter gleichzeitig in die Datenquellen zu schreiben. Die verschiedenen Parameter hätten in einem kombiniertem Schreibvorgang nicht identifiziert werden können. Stattdessen findet der Schreibvorgang in dem Prototypen statt, wenn das letzte Mal schreibend auf einen Parameter zugegriffen wird.

Die Einführung des in [CRF06] erwähnten Prinzips der *serializable Snapshot Isolation* konnte im Rahmen dieser Arbeit nicht in den Transaktionsmanager integriert werden, da bisher noch keine zufriedenstellende Methode gefunden wurde, um die Abarbeitung eines Tasks oder Pfades durch YAWL so zu pausieren, dass andere parallele Tasks weiterlaufen können. Dies ist allerdings eine Prämisse, um einen *write/write Konflikt* sequentialisieren zu können. Diese Funktionalität könnte im Rahmen einer zukünftigen Ausarbeitung im Bereich von tx+YAWL umgesetzt werden.

8 Literaturverzeichnis

Literatur

[Ato11a] Atomikos. Transactionessential, December 2011.

[Ato11b] Atomikos. Transactionessential homepage, December 2011.

[CRF06] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. (1), 2006.

[HAAR09] A.H.M. Hofstede, W.M.P. Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and Its Support Environment*. Springer, 2009.

[Hol95] David Hollingsworth. Workflow management coalition: The workflow reference model. *Management*, (1):1–55, 1995.

[JBo11a] JBossTS. Jboss ts features, December 2011.

[JBo11b] JBossTS. Jboss ts plattform homepage, December 2011.

[SH99] G. Saake and A. Heuer. *Datenbanken. Implementierungstechniken.* MITP-Verl., 1999.

[SHA11] S.Schick, H.Meyer, and A.Heuer. Enhancing workflow data interaction patterns by a transaction model. 1-14, 2011.

[vdAtH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. *null*, 2005.

[Wei88] G. Weikum. *Transaktionen in Datenbanksystemen: fehlertolerante Steuerung paralleler Abl*
äufe. Addison-Wesley, 1988.

[WV02] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2002.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 12. April 2012