Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Diplomarbeit

Parallele Ausführung von Prozessen auf mobilen Geräten

Kristof Hamann

kristof.hamann@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 5608752

Erstgutachter: Prof. Dr. Winfried Lamersdorf
Zweitgutachter: Prof. Dr. Wolfgang Menzel

Februar 2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die Informationstechnologie befindet sich in einem ständigen Wandel. Aufgrund des rasanten Fortschritts bei der Entwicklung von zunehmend kleineren und schnelleren Geräten ist es möglich geworden, den Computer immer besser an die Bedürfnisse des Menschen anzupassen. So ist es nicht ungewöhnlich, mit seinem Mobiltelefon unterwegs Fotos aufzunehmen und diese direkt in eine öffentliche Bildersammlung zu integrieren sowie von überall E-Mails abzurufen. Es ist für viele Menschen zur Selbstverständlichkeit geworden Computer ständig mit sich herumzutragen, um zu jeder Zeit und an jedem Ort den Zugriff auf verschiedene Dienste und die eigenen Daten zu erhalten. Die dabei verwendeten Geräte zeichnen sich durch ihre Portabilität und die spezielle Anpassung an die mobile Nutzung aus.

Gerade wegen dieser besonderen Eigenschaften mobiler Geräte müssen allerdings viele Defizite gegenüber stationären Geräten hingenommen werden. Mobile Geräte sind inhärent leistungsschwächer als stationäre Geräte und können bei der mobilen Nutzung in der Regel nur auf unzuverlässige drahtlose Kommunikationsnetze zugreifen [Sat96]. Diese Einschränkungen können zu einer Reihe von Problemen führen, die bei der Entwicklung von Anwendungen für mobile Systeme berücksichtigt werden müssen. Daher werden bisher in der Regel nur relativ einfache Aufgaben auf mobilen Geräten ausgeführt. Dabei ist es jedoch möglich, durch die Kooperation mit den Geräten in der jeweiligen Umgebung in mobilen Systemen sogar komplexe Aufgaben zu bearbeiten [Kun08].

Komplexe Aufgaben können durch Workflow-Prozesse ausgedrückt werden, einem generischen Modell zur Beschreibung von Aufgaben, die zu einem bestimmten Zweck miteinander verknüpft werden. Im Rahmen des Forschungsprojektes DEMAC (*Distributed Environment for Mobility-Aware Computing*) wurde eine Systemarchitektur entwickelt, die mobile Prozesse in eine adaptive Infrastruktur für das Mobile Computing integriert [Kun08]. Solche mobilen Prozesse können mit Hilfe einer (mobilen) Ausführungsumgebung vollkommen dezentral und auch ohne ständige Verbindung zu anderen Geräten ausgeführt werden [Zap05].

Um einen Prozess möglichst effizient auszuführen, können mehrere Geräte gleichzeitig in die Ausführung einbezogen werden. Die einzelnen Aufgaben werden dann parallel abgearbeitet. Dies ist gerade für ressourcenarme mobile Geräte vorteilhaft, da die

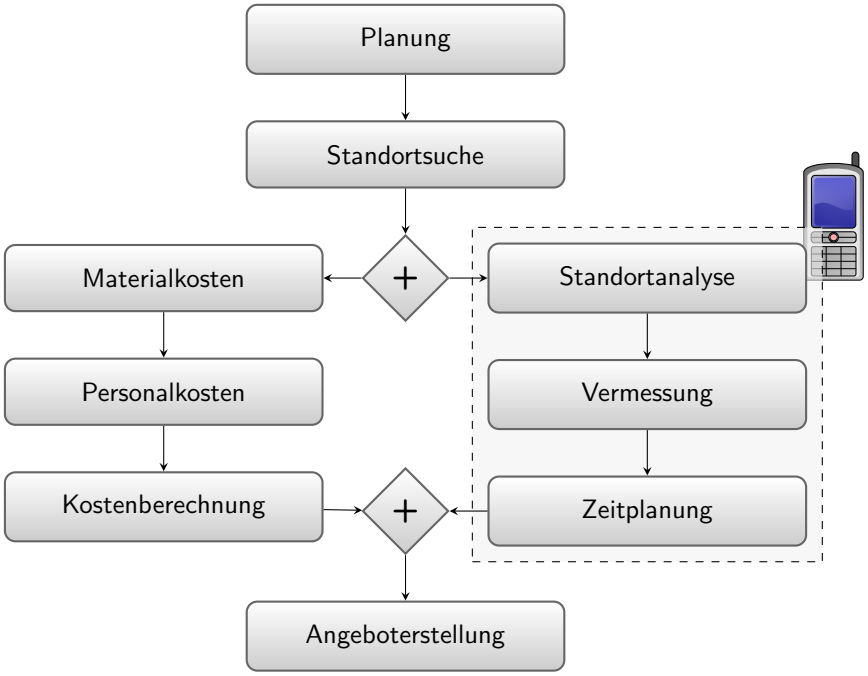


Abbildung 1.1: Prozess zur Planung von Unternehmensniederlassungen

Ausführung auf einem einzelnen Gerät in der Regel deutlich länger dauert. Aber auch ein einzelnes leistungsstärkeres mobiles Gerät kann durch die nebenläufige Ausführung eines Prozesses beispielsweise Wartezeiten bei entfernten Dienstaufrufen überbrücken und so die Effizienz steigern. Klassische Prozessbeschreibungssprachen sehen daher Konstrukte vor, die es der Ausführungsumgebung erlauben, einzelne Aufgaben nebenläufig oder echt parallel auszuführen. Da in der Literatur im Kontext von Workflow-Prozessen in der Regel nicht zwischen Nebenläufigkeit und Parallelität unterschieden wird [Wes07, AH02, LR00], verwendet auch diese Arbeit lediglich den Begriff der Parallelität.

Um die Vorteile der parallelen Ausführung zu verdeutlichen, wird nun beispielhaft ein Prozess zur Planung von Unternehmensniederlassungen betrachtet. Im Vorfelde der Gründung einer neuen Niederlassung müssen zunächst in einer Planungsphase allgemeine Randbedingungen manifestiert werden. Anschließend erfolgt die Suche eines geeigneten Standortes, an dem der Plan realisiert werden kann. Im weiteren Verlauf müssen auf der einen Seite verschiedene Berechnungen zu den voraussichtlichen Kosten des Projekts durchgeführt werden. Davon unabhängig muss auf der anderen Seite eine genauere Analyse des Standorts, die konkrete Vermessung auf dem Grundstück und die weitere Zeitplanung erfolgen. Dieser Teil des Prozesses muss vor Ort, d. h. mit der Unterstützung eines mobilen Geräts, ausgeführt werden, während die Kostenberechnung in der Zentrale mit Hilfe von stationären Geräten erfolgen kann. Der Prozess wird durch die Angebotserstellung abgeschlossen (vgl. Abbildung 1.1).

Aufgrund der logischen Unabhängigkeit zwischen der Kostenberechnung und den Arbeiten vor Ort, können diese beiden Teile des Prozesses parallel ausgeführt werden.

Neben der zeitlichen Entkopplung der Ausführung der Aktivitäten kann so das mobile Gerät weitgehend autark den entsprechenden Prozessabschnitt bearbeiten. Dies ist besonders vorteilhaft, wenn an dem Standort noch keine Kommunikationsinfrastruktur vorhanden ist.

Die parallele Ausführung von Prozessen auf mobilen Geräten birgt jedoch viele Probleme und Fragestellungen, die in klassischen verteilten Systemen mit stationären Geräten in dieser Form nicht auftreten. In mobilen Umgebungen sind beispielsweise Ausfälle von Geräten oder von Kommunikationsnetzen keine Ausnahmesituationen sondern die Regel. Derartige Eigenschaften mobiler Systeme müssen bereits auf konzeptioneller Ebene angemessen berücksichtigt werden. Dies hat unter anderem Auswirkungen auf die Auswahl geeigneter Teilnehmer eines Prozesses sowie auf das Verteilungsmodell der von einem Prozess benötigten Daten und auf die Art der Koordination paralleler Prozessabschnitte.

1.1 Zielsetzung

Die vorliegende Arbeit beschäftigt sich daher mit der parallelen Ausführung von Prozessen auf mobilen Geräten. Das Ziel der Arbeit ist es, zunächst die dabei auftretenden Probleme und Fragestellungen zu analysieren und bestehende Lösungsansätze auf ihre Eignung hin zu untersuchen. Auf dieser Grundlage soll ein Konzept zur Ausführung von parallelen Prozessabschnitten erarbeitet werden. Praktischer Teil ist die prototypische Implementierung im Rahmen der *DEMAC*-Middleware, um die Realisierbarkeit des Ansatzes zu zeigen.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst die für diese Arbeit wichtigen Grundlagen des Mobile Computings sowie der Dienste und Prozesse erarbeitet. Anschließend werden verschiedene Ansätze zur Ausführung von Prozessen auf mobilen Geräten diskutiert. Der weitere Verlauf der Arbeit orientiert sich an mobilen Prozessen als Vertreter der dezentralen Ausführung. Das Kapitel schließt mit der Analyse der Problematik bei der parallelen Ausführung von mobilen Prozessen.

Auf der Grundlage der Problemanalyse werden in Kapitel 3 bestehende Ansätze zur Synchronisation von nebenläufigen Vorgängen betrachtet. Dabei wird untersucht, ob und wie die Verfahren in mobilen Umgebungen angewendet werden können. Der Schwerpunkt der Untersuchung liegt in Verfahren zur Nebenläufigkeitskontrolle. Es wird also untersucht, wie parallele Zugriffe auf gemeinsame Daten koordiniert werden können.

Die Ergebnisse aus der Betrachtung bestehender Ansätze sowie der Problemanalyse führen zu dem erarbeiteten Konzept zur parallelen Ausführung von mobilen Prozessen

in Kapitel 4. Dabei wird zunächst eine Methode konzipiert, mit der Abhängigkeitskonflikte beim Zugriff auf gemeinsame Daten durch parallele Prozessabschnitte bereits vor der Ausführung erkannt werden können. Anschließend folgt ein Lösungsvorschlag zur effizienten Auflösung dieser Abhängigkeitskonflikte mit Hilfe von anwendungsabhängigen Kriterien. Darüber hinaus werden Vorschläge zur Beantwortung der in der Problemanalyse identifizierten Fragestellungen erarbeitet.

Kapitel 5 stellt die prototypische Implementierung des erarbeiteten Konzepts vor. Hierzu wird zuerst auf den Kontext der Realisierung – das Forschungsprojekt *DEMAC* – eingegangen, um anschließend die gewählte Architektur der Implementierung und ihre Integration in die bestehende Middleware vorzustellen.

Die Abhandlung schließt in Kapitel 6 mit einer Zusammenfassung der erreichten Ziele und einer Diskussion der Fragestellungen, die sich für weitere Arbeiten ergeben.

Kapitel 2

Prozessorientierung im Mobile Computing

Die vorliegende Arbeit basiert wesentlich auf den zwei grundlegenden Bausteinen *Mobile Computing* und *Workflow-Prozesse*. In diesem einführenden Kapitel werden daher zunächst die Besonderheiten und Möglichkeiten mobiler Umgebungen behandelt (Abschnitt 2.1). In Abschnitt 2.2 folgt die Betrachtung der für diese Arbeit notwendigen Grundlagen aus dem Gebiet der Dienste und Prozesse. Darauf aufbauend diskutiert Abschnitt 2.3 verschiedene Ansätze zur Ausführung von Prozessen auf mobilen Geräten und stellt somit die Verknüpfung der grundlegenden Bausteine her. Der Schwerpunkt liegt dabei in der Betrachtung mobiler Prozesse. Das Kapitel wird mit einer Problemanalyse abgeschlossen, bei der untersucht wird, welche Fragestellungen und Probleme bei der parallelen Ausführung von mobilen Prozessen zu berücksichtigen sind (Abschnitt 2.4).

2.1 Mobile Computing

Die steigende Verbreitung von mobilen Geräten macht die Erforschung und genaue Analyse dieser Systeme notwendig. Zwar können grundlegende Problemlösungen aus dem Bereich der verteilten Systeme übertragen werden. In mobilen Systemen treten jedoch auch völlig neue Eigenschaften und Probleme auf, die ein eigenes Forschungsgebiet rechtfertigen [Sat01]. *Mobile Computing* beschäftigt sich mit den Herausforderungen, die durch mobile Geräte und den Umgang mit ihnen entstehen. Dies betrifft explizit auch die Mobilkommunikation und Anwendungen auf mobilen Geräten [Rot05].

Drei Eigenschaften von mobilen Systemen sind essentiell für das *Mobile Computing*: Kommunikation, Portabilität und Mobilität [FZ94]. Die Bedeutung der Kommunikation hat nicht erst seit der Etablierung von Mobilfunkgeräten stark zugenommen. Ein Computer ohne Internetzugang ist heute so gut wie undenkbar und so soll ein Zugang zu diesen Ressourcen auch mit einem mobilen Gerät möglich sein, was wiederum zu einem steigenden Interesse an drahtlosen Netzen führt. Die Besonderheiten von drahtlosen Netzen im Speziellen und der Mobilkommunikation im Allgemeinen werden in Abschnitt 2.1.2 behandelt. Portabilität und Mobilität adressieren zwar beide Bewegung, es gibt jedoch wichtige Unterschiede. Portabilität ist eine Eigenschaft von Geräten, die besondere Anforderungen an die Gestaltung der Geräte stellt, wie beispielsweise Größe,

Gewicht und Energiebedarf. Mobilität hingegen leitet sich aus dem Benutzungsverhalten ab. Im folgenden Unterabschnitt werden diese beiden Begriffe genauer betrachtet.

2.1.1 Portabilität und Mobilität

Die technische Entwicklung der letzten Jahrzehnte hat dazu geführt, dass Computer immer kleiner wurden. Während früher ein Computer einen ganzen Raum belegen musste, existieren heute Geräte, die auf eine Handfläche passen und trotzdem die gleichen Berechnungen durchführen können wie ihre großen Verwandten. Aufgrund der Reduktion von Größe und Gewicht sind Computer erstmals tragbar, also portabel, geworden. Auf der anderen Seite befindet sich der Mensch, als Nutzer der Computer, im täglichen Leben an vielen verschiedenen Orten. Es ist daher naheliegend, dass der Wunsch aufgekommen ist, die eingesetzten technischen Geräte auch mobil verwenden zu können. Im *Mobile Computing* werden verschiedenen Ausprägungen des Begriffes Mobilität unterschieden:

Endgerätemobilität Wird ein Endgerät, beispielsweise ein Notebook oder Handy, physikalisch bewegt und bleibt dabei weiterhin mit dem Kommunikationsnetz verbunden, so wird es als mobil bezeichnet [WK01, JBK98]. Frühe Mobilfunknetze unterstützten beispielsweise nur beschränkt Endgerätemobilität, da sie den Wechsel der Sendestation beim Verlassen des Empfangsbereiches nicht während des Gespräches durchführen konnten [Rot05].

Benutzermobilität Nicht das für die Interaktion benutzte Gerät wird hier als Endpunkt betrachtet, sondern der Benutzer selbst. Dieser ist mobil, wenn er für die Ausführung einer Aufgabe verschiedene Geräte benutzen kann [JBK98]. Das ist beispielsweise in einem Rechnerverbund eines Rechenzentrums der Fall. Im Allgemeinen kann es sich dabei um Geräte unterschiedlichen Typs handeln.

Dienstemobilität Der Zugriff auf einen Dienst erfolgt dann mobil, wenn während der Nutzung des Dienstes das verwendete Gerät, und gegebenenfalls auch der Ort, gewechselt wird. Ist der Dienst bei einem solchen Wechsel weiterhin benutzbar, so ist er mobil. Ein Beispiel ist der Wechsel vom Festnetz-Telefon zum Handy während des Gespräches ohne dabei eine neue Verbindung aufbauen zu müssen. [WK01, JBK98]

Code-Mobilität Wird ein Programm von einem Gerät auf ein anderes Gerät übertragen, so spricht man von Code-Mobilität. Fugetta et al. unterscheiden dabei das Code-Segment, welches aus der statischen Beschreibung des Verhaltens des Programmes besteht, und den Ausführungszustand, welcher Position und Zustandsvariablen der aktuellen Ausführung beinhaltet, sowie Ressourcen, die von dem Programm verwendet werden. [FPV98]

Benutzermobilität und Dienstemobilität sind insbesondere bei der Interaktion mit Benutzern von Bedeutung. In dieser Arbeit stehen Endgerätemobilität sowie Code-Mobilität im Vordergrund. Ist im Folgenden die Art der Mobilität nicht angegeben oder aus dem Kontext ersichtlich so ist Endgerätemobilität gemeint.

Zunächst wird die Endgerätemobilität noch einmal genauer betrachtet. Es werden drei Stufen der Mobilität unterschieden [Per06]:

Stationär Das gleiche Gerät wird am gleichen Ort verwendet – es gibt keine Mobilität. Der klassische fest zugewiesene Arbeitsplatz mit Personalcomputer ist ein Beispiel für diese unterste Stufe.

Nomadisch Benutzer greifen von verschiedenen Orten mit verschiedenen Geräten auf das System zu. Während der Interaktion ändert sich aber nicht der Ort. Ein Beispiel ist der Geschäftsreisende, der sein Notebook beim Kunden mit dessen Firmennetz verbindet. Bereits in diesem einfachen Fall müssen Probleme gelöst werden, mit denen sich das Forschungsgebiet *Nomadic Computing* beschäftigt [Kle96].

Mobil Benutzer bewegen sich während ihrer Interaktion und stellen somit besondere Anforderungen an die Kommunikationsinfrastruktur. Werden beispielsweise während einer Reise mit einem mobilen Gerät verschiedene drahtlose Netze durchquert, sollte dies transparent für den Nutzer passieren.

Es ist also nicht notwendig, dass ein portables Gerät auch mobil eingesetzt wird. So kann ein Notebook am Arbeitsplatz einen Personalcomputer ersetzen ohne dabei mobil zu sein. Auch wird ein Gerät nicht unbedingt mobil genutzt, nur weil es mit einem drahtlosen Netz verbunden ist. Die höchste Stufe der Mobilität stellt den schwierigsten und somit interessantesten Fall dar. Das *Nomadic Computing* kann also dem *Mobile Computing* untergeordnet werden.

Probleme der Portabilität

Um Endgerätemobilität zu ermöglichen, müssen die verwendeten Geräte portabel sein. Dies hat zunächst zur Folge, dass Beschränkungen in Bezug auf Größe und Gewicht hingenommen werden müssen. Zusammen mit der Anforderung, vom Stromnetz autark zu sein, folgen jedoch eine Reihe weiterer Eigenschaften, die im Folgenden vorgestellt werden.

Trotz ständiger Verbesserungen der Akkuleistung nimmt die Energiequelle gerade bei kleinen Geräten wie Mobiltelefonen und GPS-Empfängern immer noch am meisten Platz und Gewicht in Anspruch. Wird Größe und Gewicht der Energiequelle auf Kosten ihrer Leistung reduziert, kann dies jedoch den Nutzen der Portabilität verringern, da das Gerät dann häufig wiederaufgeladen oder ausgeschaltet werden muss. [FZ94]

Um dies zu umgehen wird versucht, den Energiebedarf des Gerätes zu senken. Dazu wird spezialisierte, meist teurere Hardware eingesetzt, die Software für die effiziente

Nutzung der Ressourcen angepasst und die Leistung des Gerätes gedrosselt [FZ94]. Mobile Geräte werden zwar ständig verbessert, aufgrund des nötigen Kompromisses zwischen Größe, Gewicht, Akkulaufzeit, Leistung und Kosten werden sie jedoch immer leistungsschwächer als unbewegliche Geräte sein [Sat96]. Dies wirkt sich zum Einen auf die Prozessorleistung aus, zum Anderen aber auch auf die Größe des Festspeichers, welcher unter anderem aufgrund der Stoßempfindlichkeit von Magnetspeichern durch Flash-Speicher ersetzt wird.

Weiterhin sind nicht nur wegen der Platzbeschränkungen neue Methoden der Ein- und Ausgabe notwendig. Auch der Wunsch nach Interaktion während der Fortbewegung führt zu völlig anderen Anforderungen als sie am Arbeitsplatzcomputer herrschen, etwa die Bedienung mit nur einer Hand. In aktuellen mobilen Geräten sind bereits verschiedene alternative Techniken vertreten, wie die Eingabe mittels eines Stiftes auf einem druckempfindlichen Bildschirm, die Abbildung von Buchstaben auf eine Zehnertastatur oder die Erkennung von gesprochenen Befehlen. Die Ausgabe erfolgt primär weiterhin über einen Bildschirm, der jedoch deutlich kleiner sein muss. Dies führt dazu, dass klassische Fenstersysteme nicht benutzbar sind. In manchen Systemen, insbesondere bei Navigationsgeräten, wird darüber hinaus Sprachausgabe verwendet. [Per06]

Auch nichttechnische Probleme birgt die Portabilität. So kann ein getragenes Gerät viel leichter entwendet werden als ein stationäres. Defekte durch physisches Einwirken oder gar der komplette Verlust eines Gerätes sind ebenfalls deutlich wahrscheinlicher [Sat96]. Dies hat außerdem Konsequenzen für die auf dem Gerät gespeicherten Daten. Während der Verlust durch Sicherungen auf anderen Geräten verhindert werden kann, ist bei vertraulichen Daten der unauthorisierte Zugriff ein sehr ernstzunehmendes Problem. [Eck03]

Probleme der Mobilität

Die Fähigkeit, im laufenden Betrieb den Ort zu wechseln tritt erstmals bei mobilen Geräten auf. Klassische Systeme ändern nur selten ihren Ort und werden in einem solchen Fall manuell an die neue Umgebung angepasst. Im Falle von mobilen Systemen ist die manuelle Anpassung aber nicht wünschenswert, da hier mit einer hohen Flexibilität Orte gewechselt werden.

Um erfolgreich kommunizieren zu können, muss jedes Gerät eine eindeutige Adresse besitzen. In der Regel ist mit dieser Adresse verknüpft, auf welchem Weg das Gerät erreicht werden kann. Beim Abbruch der Verbindung und Anmelden in einem neuen Netz erhält das Gerät daher eine neue Adresse. Um weiterhin mit den bisherigen Partnern kommunizieren zu können, müssen deren Nachrichten auf geeignete Weise an die neue Adresse gelangen. [FZ94]

Neben der Adresse können sich noch weitere Umgebungsinformationen, die normalerweise statisch sind, plötzlich ändern [FZ94]. Dazu zählen beispielsweise andere Geräte in der Umgebung, erreichbare Dienste und deren Adressen. Um auf Änderungen rea-

Tabelle 2.1: Kombinationen von drahtlosen Netzen und mobilen Computern (aus [Tan03, S. 25])

drahtlos	mobil	Beispielanwendung
–	–	Arbeitsplatzrechner
–	✓	Im Hotelzimmer benutztes Notebook
✓	–	Netze in älteren Gebäuden ohne Verkabelung
✓	✓	PDA zur Erfassung des Lagerbestands

gieren zu können, müssen diese Umgebungsinformationen gesammelt und verarbeitet werden. Ein System, welches dazu in der Lage ist, wird als *kontextbewusst* (*context-aware*) bezeichnet [SAW95]. Abschnitt 2.1.3 befasst sich näher mit dieser Eigenschaft.

2.1.2 Mobilkommunikation und drahtlose Netze

Die Kommunikation zwischen mobilen Benutzern, auch *Mobilkommunikation* genannt, ist ein wichtiger Teilbereich vom *Mobile Computing* [Rot05]. Eine drahtlose Kommunikation ist zwar nicht zwingend notwendig[1] (vgl. Tabelle 2.1) aber aus praktischen Gründen die Regel. Im Folgenden stehen daher drahtlose Netze im Vordergrund.

Bevor auf die bei der Mobilkommunikation und insbesondere bei der drahtlosen Kommunikation auftretenden Probleme eingegangen wird, werden zunächst grundlegende Eigenschaften von Netzen behandelt, die für die Mobilkommunikation von Bedeutung sind.

Infrastruktur von Netzen

Jede Form der Kommunikation erfordert eine geeignete Art der Verbindung, über die Informationen ausgetauscht werden können. Sind mehrere Computer mit einer bestimmten Technologie miteinander verbunden, so spricht man von einem Rechnernetz [Tan03]. Ein solches Netz kann auf zwei Arten aufgebaut werden [Per06]:

Traditionelle Infrastruktur Traditionell wird die Infrastruktur eines Rechnernetzes hergestellt, bevor es benutzt wird. Dies hängt meist damit zusammen, dass Kabel verlegt und für die Infrastruktur notwendige Geräte, wie Router oder Zugangsknoten, konfiguriert werden müssen (vgl. Abbildung 2.1).

Ad Hoc Die Verbindung zwischen Teilnehmern wird spontan hergestellt sobald dies erforderlich ist. Das Netz benötigt keine oder wenig Konfiguration und ist meist nicht von langer Dauer. Die Verwaltung des Netzes erfolgt dezentral, da keine Router o. ä. zur Verfügung stehen (vgl. Abbildung 2.2). Die Teilnehmer sind somit gleichzeitig Kommunikationsendpunkte als auch Teil der Infrastruktur. Ein mobiles Ad-Hoc-Netz wird auch *MANET* genannt. [Rot05]

1 Ein mobiles Gerät, wie ein Notebook, kann beim Ortswechsel auch ein vorhandenes Ethernet durch Anschluss eines entsprechenden Kabels nutzen

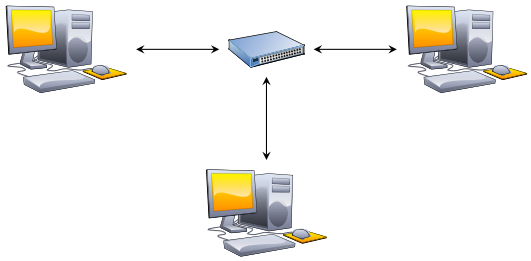


Abbildung 2.1: Traditionelle Netz-Infrastruktur

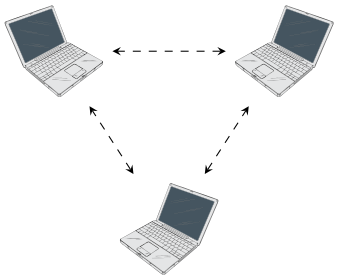


Abbildung 2.2: Ad-hoc-Netz

Gerade bei der *Mobilkommunikation* stellt die Ad-Hoc-Vernetzung ein wichtiges Verfahren dar, um kurzfristig mit Geräten in der Umgebung Daten auszutauschen [Rot05]. Schließlich dauert das Konfigurieren einer festen Kommunikationsinfrastruktur möglicherweise länger, als die Kommunikationspartner sich überhaupt in Reichweite befinden. Auf der anderen Seite kann ein Ad-Hoc-Netz praktisch nur über drahtlose Kommunikation realisiert werden, da eine Verkabelung ebenfalls einen zu großen Aufwand darstellt. Somit wird sich zwar im Bereich der Mobilkommunikation nicht auf drahtlose Netze beschränkt, dennoch spielen diese eine wichtige Rolle und müssen daher genauer betrachtet werden.

Beispiele verschiedener Klassen drahtloser Netze

Wie auch drahtgebundene Netze, können drahtlose Netze nach ihrem geographischen Verteilungsgrad klassifiziert werden. Im Folgenden werden ausgewählte Vertreter dieser Klassen kurz vorgestellt und in Tabelle 2.2 die wichtigsten Verfahren mit ihren für das *Mobile Computing* interessanten Eigenschaften zusammengefasst.

Wide Area Network (WAN) Bevor die Datenübertragung per Funk in das Interesse der Gesellschaft rückte haben sich bereits Mobilfunknetze zur Übertragung von Sprache etabliert. Die gewünschte Flächenabdeckung konnte nur mit dem Konzept zellulärer Netze realisiert werden. Dabei decken *Zellen*, in denen sich je eine *Basisstation* befindet, die zu versorgende Fläche ab. Dieses Konzept ist bis heute erhalten und stellt besondere Anforderungen an die Infrastruktur, wie beispielsweise die Gesprächsübergabe beim Eintritt in eine neue Zelle. [Rot05]

Mobilfunknetze bieten in der Regel für Datenübertragungen lediglich den Zugriff auf einen Zugangspunkt des Netzbetreibers, über den man sich dann beispielsweise mit dem Internet verbinden kann. Die Kommunikation mit Geräten in der Umgebung muss dann auch über diesen Knotenpunkt abgewickelt werden. Während GSM (*Global System for Mobile Communication*) nur Leitungsvermittlung unterstützt, ist die Paketvermittlung bereits im Systemdesign des GSM-Nachfolgers UMTS (*Universal Mobile Telecommunications System*) und dessen ergänzenden Standards HSDPA (*High Speed Downlink Packet Access*) verankert. Auch im GSM-Netz sind durch die Erweiterungen GPRS (*General*

Tabelle 2.2: Übersicht über aktuelle Standards für drahtlose Netze (vgl. [Rot05, Sau08, Mac02]). Die Zahlenwerte sind theoretische Maximalgrößen für Konfigurationen in mobilen Geräten. Die Datenrate kann mit zunehmender Entfernung und steigender Nutzerzahl stark abnehmen.

Standard	Sendeleistung	Reichweite	Datenrate	Vermittlungsart	Struktur
GSM	250 mW	35 km	14,4 kBit/s	Leitung	Infrastruktur
GPRS	2000 mW	35 km	170 kBit/s	Paket	Infrastruktur
EDGE	2000 mW	35 km	473 kBit/s	Paket	Infrastruktur
UMTS	250 mW	k. A.	2 MBit/s	Paket/Leitung	Infrastruktur
HSDPA	250 mW	k. A.	14 MBit/s	Paket	Infrastruktur
WLAN	100 mW	150 m	54 MBit/s	Paket	Infrastruktur/Ad hoc
Bluetooth	2,5 mW	50 m	2 MBit/s	Paket/Leitung	Ad hoc
IrDA	100 mW	1 m	4 MBit/s	Paket/Leitung	Ad hoc

Packet Radio Service) und EDGE (*Enhanced Data Rates for GSM Evolution*) nicht nur deutlich höhere Datenraten sondern auch Paketvermittlung möglich. Tabelle 2.2 zeigt eine Übersicht der Werte aktuell eingesetzter Techniken. [2]

Local Area Network (LAN) Gängigster Vertreter für die drahtlose Vernetzung von Gebäuden oder Universitäts- beziehungsweise Firmengeländen ist *Wireless-LAN* (WLAN) nach der Standard-Familie *IEEE 802.11*.[3] WLAN kann sowohl im Infrastrukturmodus mit einem sogenannten *Access-Point* verwendet werden als auch im *Ad-Hoc-Modus*, bei dem die Teilnehmer direkt miteinander kommunizieren.

Personal Area Network (PAN) Da die Übertragung per Infrarot wie sichtbares Licht durch Gegenstände blockiert sowie durch Sonnenlicht gestört werden kann, zudem gerichtet ist und nur eine geringe Reichweite aufweist, wird heute im Nahbereich kaum noch *IrDA* sondern statt dessen die Funktechnik *Bluetooth* verwendet [Rot05]. Bluetooth wurde unter den besonderen Anforderungen mobiler Geräte entwickelt und ist daher äußerst energiesparend. Es sind drei Geräteklassen spezifiziert, um einen Kompromiss zwischen Reichweite und Energieverbrauch, je nach Anwendungszweck, zu erlauben. Sogenannte Piconetze von bis zu acht aktiven Geräten erlauben außerdem die Ad-Hoc-Vernetzung von lokalen Geräten. [Sau08]

Probleme der drahtlosen Kommunikation

Wie bereits dargelegt besteht ein großes Interesse, auch unterwegs mit einem mobilen Gerät über ein Netz kommunizieren zu können, um beispielsweise E-Mails zu lesen oder Dienste des Firmennetzes nutzen zu können. Zwar ist es dabei nicht zwingend

2 Die maximale Sendeleistung bei GSM beträgt 2 W. Da aber Zeitmultiplexing angewendet wird, beläuft sich die durchschnittliche Sendeleistung nur auf ein Achtel, wie in der Tabelle angegeben.

3 IEEE (ursprünglich *Institute of Electrical and Electronics Engineers, Inc*) ist nach eigenen Angaben die größte Gesellschaft zur Weiterentwicklung von Technologie (vgl. `http://www.ieee.org/`)

notwendig, dass über drahtlose Netze kommuniziert wird. Aber schon aus praktischen Gründen ist es unvorteilhaft, ständig neue Kabelverbindungen herzustellen, um an einem anderen Ort weiterhin Zugang zu einem Netz zu haben.

Drahtlosen Netzen stehen gegenüber drahtgebundenen Netzen jedoch deutlich mehr Hürden gegenüber. Das Signal – derzeitige Technologien verwenden elektro-magnetische Wellen – wird durch die Umgebung beeinflusst; es entstehen Rauschen und Echos oder das Signal wird von Hindernissen blockiert. Dies führt bei der Datenübertragung zu Verzögerungen und zur Notwendigkeit, Nachrichten erneut zu übertragen, sowie zu erhöhtem Aufwand durch die Verarbeitung von Fehlerprotokollen und sogar zu Verbindungsabbrüchen. Es muss daher mit einem *unzuverlässigen Netz* gearbeitet werden. [FZ94]

Da nur ein gemeinsames Übertragungsmedium existiert, skaliert die drahtlose Übertragung nicht. Je mehr Benutzer gleichzeitig Daten übertragen, desto geringer ist die Datenrate, die dem Einzelnen zur Verfügung steht. Techniken, wie das parallele Nutzen verschiedener Frequenzen oder die Verringerung der Größe von Funkzellen, können zwar die Situation verbessern, sind aber nur im begrenzten Maße anwendbar [FZ94]. Auch Verbesserungen der Ausnutzung der verfügbaren Bandbreite durch ständige Forschung in diesem Bereich können nicht die flexible Bandbreitenerhöhung durch Verlegen eines neuen Kabels ersetzen, wie dies bei drahtgebundenen Netzen möglich ist. Somit wird die verfügbare Datenrate bei drahtlosen Netzen immer stark beschränkt sein.

Die Datenrate ist jedoch nicht nur beschränkt, sie kann auch während der Nutzung erheblich schwanken. Gerade wenn aufgrund eines Ortswechsels ein anderes Netz genutzt werden muss, kann sich die Datenrate bedeutend verändern [Sat96]. So ist ein Abfall um den Faktor Tausend realistisch, wenn ein Übergang von WLAN mit 54 MBit/s auf GPRS mit häufig nur nutzbaren 56 kBit/s statt findet (vgl. [Sau08]).

Neben der Datenrate verändert sich bei einem solchen Wechsel des Netzes in der Regel ebenso das Protokoll und die Netzarchitektur. Wie Tabelle 2.2 zeigt, existieren diverse Standards zur drahtlosen Datenübertragung. Aber auch die Verfügbarkeit von Netzen ist sehr heterogen. So können an manchen Orten mehrere Netze gleichzeitig verfügbar sein, während an anderen Orten – meist auf ländlichen Gebieten – gar keine Konnektivität zu Verfügung steht. [FZ94]

Unterbrechungen der Verbindung können auch noch andere Ursachen haben. Benutzer schalten ihr Telefon bewusst aus, um nicht erreichbar zu sein oder Energie zu sparen, die Verbindung wird getrennt, um kostenpflichtige Verbindungen möglichst kurz zu halten oder weil die Nutzung des Gerätes vor Ort nicht erlaubt ist.

Die Mobilität der Geräte macht es notwendig, dass die zur Datenübertragung verwendeten elektromagnetischen Wellen in alle Richtungen im Raum übertragen werden. So können jedoch auch nicht für den Empfang bestimmte Geräte die Wellen messen und die Informationen auslesen. Um die Privatsphäre zu wahren, müssen daher geeignete

Sicherheitsmaßnahmen getroffen werden, wie beispielsweise die Verschlüsselung der zu übertragenden Daten. [Eck03]

2.1.3 Kontextbewusstsein

Während sich bei stationären Geräten die Umgebung selten oder nie ändert, ist bei mobilen Geräten genau das Gegenteil der Fall. Aufgrund der Mobilität kann durch einen Ortswechsel die Situation, in der sich das Gerät und sein Benutzer befinden, plötzlich deutlich anders sein. Während der Autofahrt zum Geschäftstreffen ist beispielsweise eine akustische Ausgabe sinnvoll, damit der Fahrer sich auf das Lenken des Fahrzeuges konzentrieren kann. Befindet er sich dann im Konferenzraum, soll das Gerät jedoch keine Geräusche mehr produzieren, sondern ausschließlich visuelle Ausgaben erzeugen.

Als *Kontext* bezeichnet Rothermel et al. die Informationen, die zur Charakterisierung der Situation von Personen, Orten oder Objekten herangezogen werden können, wobei das Objekt selbst ebenfalls Teil des eigenen Kontexts ist. Eine kontextbezogene Anwendung ist dadurch gekennzeichnet, dass ihr Verhalten durch Kontextinformationen beeinflusst wird [RBB03]. Konkret heißt dies, dass ein kontextbewusstes System (*context-aware system*) sich an den Ort der Benutzung und an die sich in der Nähe befindendem Personen sowie Geräte automatisch anpasst und sich auch bei Änderungen mit der Zeit adaptiv verhält [SAW95].

Rothermel et al. unterscheiden den Primärkontext und Sekundärkontext. Der Primärkontext umfasst Ort, Identität und Zeit. Diese drei Eigenschaften, die jede Entität besitzt, werden als wohldefinierte Struktur verwendet, über die auf Informationen im Kontextmodell zugegriffen werden kann. Der Sekundärkontext umfasst beliebige weitere Eigenschaften, die spezifisch für jeden Entitätstyp sind. [RBB03]

Betrachtet man den Kontext aus der Perspektive des Anwenders, so kann man vier Dimensionen untersuchen [Per06]:

Raum Der eigene Ort, Objekte in der Umgebung und deren Positionen. In aktuellen kontextbasierten Anwendungen wird oft nur der Ort des Gerätes berücksichtigt, da sich diese Information relativ einfach mit einem GPS-Empfänger[4] auslesen lässt. Anwendungen dieser Art werden auch *Location Based Services* genannt. Dazu zählen Navigationssysteme und ortsbezogene Stadtführer.

Raumzeit Betrachtet man zusätzlich die Zeit, so lassen sich Eigenschaften wie Richtung, Geschwindigkeit und Streckenverlauf von Objekten untersuchen.

Umgebung Eigenschaften der Umgebung, denen keine Objekte zugeordnet werden, wie Temperatur, Licht, Lautstärke und Feuchtigkeit. Diese Daten können durch physikalische Sensoren ausgelesen werden.

---

4 Das *Global Positioning System* (*GPS*) besteht aus zahlreichen Satelliten, die ein Signal aussenden, mit dessen Hilfe Geräte ihre aktuelle Position berechnen können.

Persönlicher Kontext Der persönliche Kontext des Benutzers besteht aus den folgenden Bereichen: Der physiologische Kontext gibt Auskunft über den physiologischen Zustand der Person und beinhaltet Eigenschaften wie Puls, Blutdruck und Gewicht, gegebenenfalls auch seine Fähigkeiten und Vorlieben. Der mentale Kontext gibt Auskunft über Stress, Ärger und Stimmung. Zum sozialen Kontext zählen Freunde, Nachbarn, Kollegen und Verwandte. Außerdem bilden die Aktivitäten des Benutzers und seine dabei verfolgten Ziele einen Teil des persönlichen Kontexts.

Mit einer eher technischen Sichtweise lassen sich darüber hinaus folgende, für diese Arbeit besonders wichtigen Dimensionen unterscheiden, die sich teilweise aus den obigen ableiten lassen [DRD+00]:

Infrastruktur Wie bereits in Abschnitt 2.1.2 dargestellt, verändert sich die verfügbare Kommunikationsinfrastruktur ständig. Dies äußert sich in teilweise stark variierenden Eigenschaften wie Datenrate, Latenz und Dienstgüte.

System Komplexe Anwendungen werden in der Regel nicht autark auf einem mobilen Gerät ausgeführt, sondern sie sind Bestandteil eines größeren Systems. Der Systemkontext beinhaltet die Eigenschaften von zu berücksichtigenden Systemkomponenten.

Domäne Gerade auf mobilen Geräten kann die Anwendungsdomäne sehr unterschiedlich ausfallen. So hängen die dargestellten Informationen und die Art der Benutzerschnittstellen stark davon ab, in welchem Anwendungsbereich das Gerät genutzt wird.

Physikalischer Kontext Mobile Geräte sind oft in einem physikalischen Kontext eingebettet, der von Bedeutung ist. So muss ein Autoradio andere Funktionsmerkmale aufweisen als ein tragbares Radio.

Kontextbewusste Anwendungen können einige oder alle der dargestellten Kontexttypen berücksichtigen, um ihr Verhalten entsprechend anzupassen. Drei Merkmale weisen dabei auf Kontextbewusstsein hin [RBB03]: Bei der *kontextbezogenen Selektion* wird der Kontext bei der Auswahl von Diensten und Informationen einbezogen. So kann automatisch der nächste Drucker vorgewählt werden. Wird in Abhängigkeit des Kontextes die Darstellung einer Anwendung verändert, so handelt es sich um *kontextbezogene Präsentation*. Dies ist beispielsweise der Fall, wenn im Auto Sprachausgabe verwendet wird, während beim Geschäftstreffen auf visuelle Ausgabe zurückgegriffen wird. Im Falle von *kontextbezogenen Aktionen* trifft die Anwendung in Abhängigkeit des Kontextes selbstständig Entscheidungen und führt Aktionen aus, die über die Darstellung der Anwendung hinausgehen.
Besonders interessant ist es, neben dem aktuellen Kontext zusätzlich die Historie zu betrachten, um so eine Prognose über zukünftige Kontexte erzeugen zu können. [RBB03]

Auf einer höheren Ebene ist die *kontextbasierte Kooperation* angesiedelt. Hierbei werden die Nachteile mobiler Geräte abgemildert, indem mit im Kontext vorhandenen Geräten zusammengearbeitet wird und somit neue Potenziale nutzbar gemacht werden. [Kun08]

2.1.4 Ausblick: Ubiquitous Computing

Als eine Vision stellte Weiser im Jahr 1991 seine Ideen vom allgegenwärtigen Computer (*Ubiquitous Computing*, häufig auch *Pervasive Computing*) vor [Wei91]. Dabei soll der Computer, und insbesondere der Vorgang der Interaktion mit ihm, in den Hintergrund rücken. Ähnlich wie der Vorgang des Lesens von Schildern keine Aufmerksamkeit erfordert und somit in den Hintergrund gerückt ist, soll nicht die Arbeit mit dem Computer im Vordergrund stehen, sondern dass man angenehm und effektiv Dinge erledigen kann. So sollen hunderte Geräte in jedem Raum existieren, von denen beispielsweise einige wie Papier auf dem Schreibtisch angeordnet werden können und andere mit Gegenständen verbunden sind, um diese zu orten und Informationen auszutauschen. [Wei91]

Mobilität ist ein wesentlicher Bestandteil vom *Ubiquitous Computing*, da Bewegung und insbesondere Fortbewegung in unserem Leben eine große Rolle spielt. Aus diesem Grund sind Lösungen aus dem Bereich des *Mobile Computing* auch für das *Ubiquitous Computing* von elementarer Bedeutung. [Sat01]

Die (kostenlose) Verfügbarkeit von mobilen Infrastrukturen, die höhere Bandbreiten und permanente Netzverbindungen ermöglichen, wird die Art, wie Menschen Informationstechnologien nutzen, stark beeinflussen [Per06]. Durch die stärkere Einbindung von Computern in alltägliche Arbeitsabläufe im privaten und geschäftlichen Leben kann die Vision des *Ubiquitous Computing* eines Tages zur Realität werden. Ein vielversprechender Ansatz auf dem Weg dorthin sind eine diensteorientierte Sicht auf die Umgebung und eine prozessorientierte Zusammenführung der Dienste zur Unterstützung von Arbeitsabläufen. Im nächsten Abschnitt wird dieser Ansatz im Detail betrachtet.

2.2 Dienste und Prozesse

In den letzten Jahren haben sich zwei neue Sichtweisen auf die Infrastruktur und Architektur von Computersystemen etabliert. Bei der diensteorientierten Sichtweise werden die durch Software gelieferten Funktionalitäten als Dienste verteilt aufrufbar zur Verfügung gestellt. Eng damit verknüpft ist die prozessorientierte Sichtweise, bei der Software besser an den Arbeitsabläufen und an den Prozessen von Organisationen orientiert werden soll. Die diensteorientierte Sichtweise kann dabei die Realisierung einer Prozessorientierung unterstützen.

Im Folgenden wird zunächst die diensteorientierte Architektur genauer betrachtet. Der Abschnitt beschäftigt sich anschließend mit der Umsetzung einer Prozessorientierung, die stets im Kontext der diensteorientierten Architektur zu sehen ist.

2.2.1 Diensteorientierte Architektur

Software wird immer komplexer. Es mussten daher in der Vergangenheit ständig neue Entwicklungsmethoden geschaffen werden, um die Softwareentwicklung weiterhin beherrschen zu können. Begonnen hat dies mit dem Übergang von der Assemblerprogrammierung hin zu prozeduralen Programmiersprachen, mit denen einzelne Funktionalitäten gekapselt werden konnten. Inzwischen hat sich die objektorientierte Sichtweise durchgesetzt, die mit einer zusätzlichen gröberen Einteilung die inzwischen noch komplexer gewordene Software strukturiert. Programmteile können in Bibliotheken wiederverwendet werden. Eine noch bessere Wiederverwendbarkeit auch entfernt aufrufbarer Programmteile verspricht die komponentenbasierte Entwicklung. [M+08]

Diensteorientierte Architekturen (*Service Oriented Architecture*, kurz *SOA*) sollen mit einer weiteren Abstraktionsschicht das Programmieren im Großen erlauben, bei dem ganze Geschäftsprozesse modelliert werden müssen. Im Mittelpunkt steht dabei die Idee des Dienstes. Ein *Dienst* stellt eine Funktionalität bereit, die über ein Rechnernetz plattformunabhängig genutzt werden kann. Er sollte dabei grundlegende Merkmale aufweisen, die auch schon bei der objektorientierten Programmierung eine zentrale Rolle einnehmen. *Kapselung* bedeutet, dass der Zugriff über eine wohldefinierte Schnittstelle erfolgt und die konkrete Implementierung für den Nutzer des Dienstes verborgen bleibt. Der Begriff der *losen Kopplung* betrifft dabei nicht nur die Beziehung zwischen Diensten sondern auch die Nutzung von Diensten in Anwendungen. Einer Anwendung muss nicht schon zur Übersetzungszeit bekannt sein, welcher konkrete Dienst genutzt werden soll. Statt dessen kann eine Anwendung erst zur Laufzeit einen Dienst suchen und auf diesen dann zugreifen (*dynamisches Binden*). Zentraler Aspekt ist somit die Wiederverwendung von Diensten in einem anderen Kontext beziehungsweise einer anderen Anwendung. [M+08]

Im Wesentlichen sind drei Rollen an einer diensteorientierten Architektur beteiligt (vgl. Abbildung 2.3): Der *Dienstanbieter* stellt einen Dienst zur Verfügung und veröffentlicht die Beschreibung des Dienstes in einem *Verzeichnisdienst*. Der *Dienstnutzer* sucht im Verzeichnis einen geeigneten Dienst und bindet sich dann dynamisch an diesen um ihn zu nutzen. [Pap08]

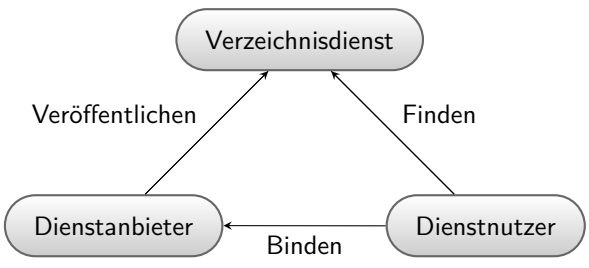


Abbildung 2.3: Struktur einer diensteorientierten Architektur (nach [Pap08, S. 24])

Anwendungen können einen solchen Verzeichnisdienst jedoch nur erfolgreich nutzen, wenn sie zum Einen formulieren können, welche Dienstart sie suchen. Dazu ist eine gemeinsame *Semantik* erforderlich, was derzeit noch ein großes Problem darstellt. Zum Anderen müssen sie maschinenlesbare Schnittstellenbeschreibungen verarbeiten, um den gesuchten Dienst, der zuvor nicht bekannt war, aufrufen zu können. Es müssen daher *offene Standards* eingesetzt werden, um die Nutzung verschiedenster Dienste in einer heterogenen Landschaft ermöglichen zu können. [M+08]

Mit diesen sehr einfachen Basistechnologien können nun komplexe Anwendungen realisiert werden, indem mehrere Dienste miteinander verknüpft werden. Ein auf höherer Ebene angesiedelter Dienst, der mehrere andere Dienste aufruft, kann so die unterliegende Komplexität verbergen. Dies erleichtert insbesondere die Umsetzung von Geschäftslogik, da von Details und Problemen, die auf einer unteren Ebene auftreten, abstrahiert werden kann. Man spricht dabei auch vom Komponieren, da die einzelnen Dienste so zusammengefügt werden, dass sie ein Gesamtwerk ergeben – wie die Noten beim Komponieren eines Musikstückes. [ACKM04]

In der Regel werden auf diese Weise Geschäftsprozesse (vgl. Abschnitt 2.2.2) realisiert. Da diese auch unternehmensübergreifend ausgeführt werden, sind offene Standards notwendig. Ein Beispiel für einen solchen offenen Standard und zugleich die derzeit gängigste Anwendungsform von SOA sind *Webservices*. Die verwendeten Protokolle basieren auf Internettechnologien und sind daher relativ einfach in bestehende Infrastrukturen zu integrieren, was zur großen Akzeptanz von Webservices geführt hat [PTDL07].

SOA hat viele Facetten. Insbesondere zählen dazu auch betriebswirtschaftliche und organisatorische Aspekte, die in dieser technisch orientierten Arbeit jedoch vernachlässigt werden. Von großer Bedeutung ist hingegen die bereits erwähnte Komposition von Diensten im Rahmen von Geschäftsprozessen.

2.2.2 Workflow-Prozesse

In Unternehmen und Organisationen bilden sich mit der Zeit eine Vielzahl von Strukturen im täglichen Arbeitsablauf. Dabei erfüllen verschiedene Aufgaben, die von Mitarbeitern oder Software ausgeführt werden, ein bestimmtes betriebliches Ziel und erzeugen damit einen Mehrwert für das Unternehmen. Eine Abfolge von solchen Aktivitäten wird *Geschäftsprozess* genannt [All05]. In der Regel müssen dafür Dokumente und Daten zwischen verschiedenen Abteilungen oder Rollen ausgetauscht werden. Im Interesse der Kostenersparnis und Fehlervermeidung haben Unternehmen den Wunsch, diese Vorgänge zu automatisieren. *Workflow-Management-Systeme* (WfMS) sollen hier unterstützen, indem sie den Informationsfluss zwischen den an einem Geschäftsprozess Beteiligten regeln [AH02]. Die dazu notwendige formale und ausführbare Definition eines Geschäftsprozesses wird *Workflow-Prozess*, oft auch nur *Workflow* oder *Prozess*, genannt [ACKM04].

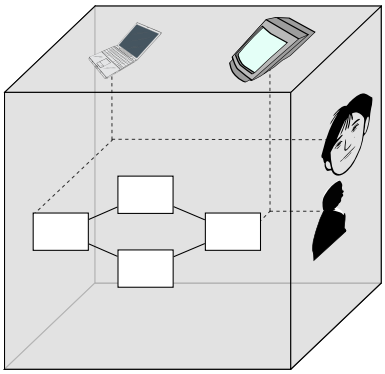


Abbildung 2.4: Die drei Dimensionen von Workflows (nach [LR00, S. 8]): Prozesslogik, Organisationsstrukturen und Infrastruktur

Neben dieser gängigen Definition, bei der ein Prozess ein Geschäftsziel mit einem Mehrwert für das Unternehmen erreichen soll, kann als Prozess auch allgemeiner ein Ablauf von zu einem beliebigen Zweck verknüpften Aufgaben bezeichnet werden (vgl. [AH02]). Ein Prozess muss also nicht notwendigerweise im Unternehmenskontext stehen, sondern er kann außerdem beispielsweise die notwendigen Schritte für das Eintragen eines unterwegs aufgenommenen Fotos mitsamt ortsbezogenen Informationen in die Bildersammlung auf der privaten Website beschreiben. Im Rahmen dieser Arbeit wird diese allgemeinere Definition verwendet.

Ein Prozess wird von drei Dimensionen charakterisiert (vgl. Abbildung 2.4): In der *Prozesslogik* ist festgelegt, welche Aufgaben ausgeführt werden und in welcher Reihenfolge dies passieren muss. Je nach konkreter Ausprägung des Prozesses wird möglicherweise nur ein Teil dieser Aufgaben bearbeitet – auch die Reihenfolge kann variieren. *Organisationsstrukturen* sind notwendig, um spezifizieren zu können, wer die Aufgaben ausführen soll. Dabei werden in der Regel Rollen statt konkreter Personen verwendet, um gewisse Fähigkeiten oder Befugnisse zu fordern und gleichzeitig dynamisch auf Änderungen der Personalstruktur reagieren zu können. Die dritte Dimension beschreibt die *Infrastruktur*, die für die Ausführung der Aufgaben nötig ist. Dies sind meistens Anwendungen, die eine Person bei der Erledigung einer Aufgabe unterstützen oder diese sogar autonom durchführen. Zusammenfassend werden Personen und Maschinen auch als *Ressourcen* bezeichnet, die zum Ausführen einer Aufgabe benötigt werden. [LR00]

Eine *Aufgabe* ist eine atomare logische Einheit der Arbeit, wie beispielsweise das Erstellen eines Konstenvoranschlages oder das Senden eines Faxes. Es können dabei drei Arten von Aufgaben unterschieden werden: *Manuelle Aufgaben* werden von einer Person ohne Unterstützung des Computers ausgeführt, *automatische Aufgaben* werden ohne Eingriff durch ein Programm ausgeführt und *semi-automatische Aufgaben* werden von einer Person mit Hilfe einer entsprechenden Anwendungssoftware erledigt. Ist eine Aufgabe einem Prozess zugeordnet, so wird sie *Aktivität* genannt. [AH02]

Prozesse lassen sich anhand verschiedener Eigenschaften klassifizieren, die Auskunft darüber geben, wie sinnvoll die Verwendung von Workflow-Management-Systemen ist

Tabelle 2.3: Klassifizierung von Prozessen (aus [Lie07, S. 86], gekürzt)

Klassifizierung	Prozessart	Beschreibung
Strukturgrad	strukturiert	vollständig vorherbestimmt und aufgrund fester Regelungen automatisierbar
	semi-strukturiert	enthält Elemente, die nicht im Voraus beschreibbar sind
	unstrukturiert	nicht formalisierbar, da strukturelle Entscheidungen im Vorweg nicht gefällt werden können
Auftretensart	zyklisch	regelmäßiges Auftreten zu genauen Zeitpunkten
	wiederholt	keine festgelegten Starttermine
	einmalig	einmaliger Vorgang, keine Wiederholung
Häufigkeit	häufig	häufig auftretende Vorgänge
	situativ	je nach Anfragesituation
	selten	selten oder einmaliges Auftreten

(vgl. Tabelle 2.3). Der Strukturgrad gibt an, ob eine Formalisierung und damit Automatisierung überhaupt möglich ist. Gewisse Prozesse enthalten kaum feste Strukturen und erfordern viel Kreativität. Diese *Ad-Hoc-Workflows* lassen sich daher im Gegensatz zu *Produktions-Workflows*, die eine sehr starre Struktur aufweisen, in der Regel nicht formalisieren [LR00]. Bei einmalig auftretenden Prozessen kann zwar eine Formalisierung theoretisch möglich sein, aber der Aufwand der Modellierung rechtfertigt normalerweise nicht den Nutzen. Häufig auftretende Vorgänge profitieren hingegen von der Modellierung, da eine Automatisierung den Vorgang meist beschleunigt und darüber hinaus eine etwaige Optimierung vereinfacht. [Lie07]

Unternehmen haben daher ein großes Interesse, ihre Geschäftsprozesse mit Hilfe von Workflow-Management zu automatisieren. Eine diensteorientierte Architektur kann dabei insofern unterstützend wirken, als dass die einzelnen Aktivitäten eines Prozesses von Diensten erbracht werden. Die im vorherigen Abschnitt vorgestellten Eigenschaften sollen dazu führen, dass eine so realisierte Infrastruktur auf Änderungen von Geschäftsprozessen flexibel reagieren kann. Diese sind für Unternehmen notwendig, um in der Marktwirtschaft dynamisch agieren zu können. Es ist daher von Vorteil, wenn Anwendungen nicht komplett neu geschrieben werden müssen, sondern bestehende Dienste in einer neuen Komposition wiederverwendet werden. [Lie07]

Workflow-Referenzmodell Verschiedene Unternehmen bieten eine Vielzahl von Produkten an, die im Workflow-Management auftreten und bei der Prozessorientierung unterstützen sollen. In dieser Arbeit wird sich jedoch nicht auf ein konkretes Produkt beschränkt, sondern es werden allgemeingültige Konzepte von Prozessen behandelt. Daher wird in diesem Abschnitt kurz auf die prinzipielle Struktur von Workflow-Management-Systemen anhand eines Referenzmodells der *Workflow Management Coalition* (WfMC) eingegangen. Die *WfMC* ist ein Zusammenschluss von im Workflow-Ma-

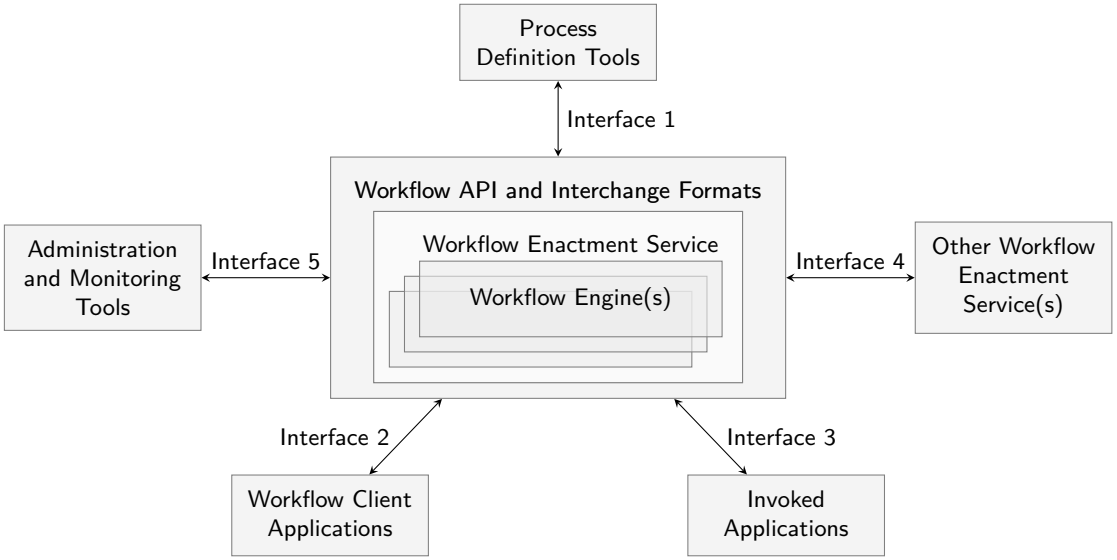


Abbildung 2.5: WfMC Workflow-Referenzmodell (aus [Hol95, S. 20])

nagement tätigen Unternehmen, der gegründet wurde, um Standards im Bereich des Workflow-Managements zu erarbeiten. Dies ist notwendig geworden, da Geschäftsprozesse nicht nur betriebsinterne Abläufe beschreiben sondern insbesondere auch unternehmensübergreifend spezifiziert werden sollen. Es wurde daher zunächst ein Architekturmodell für Workflow-Management-Systeme, das *Workflow-Referenzmodell*, entworfen [Hol95].

Im Mittelpunkt dieses Workflow-Referenzmodells (vgl. Abbildung 2.5) steht der *Workflow-Enactment-Service*, der aus mehreren *Workflow-Engines* – also Ausführungsumgebungen für Workflows – bestehen kann. Er verwaltet bestehende Workflows, erzeugt neue Ausführungsinstanzen und führt diese aus. Über eine Schnittstelle (Interface 1) wird das zuvor mit Modellierungswerkzeugen erstellte Prozessmodell übermittelt.

Zur Ausführungszeit müssen Benutzeranwendungen mit dem *Workflow-Enactment-Service* interagieren, um manuelle und semi-automatische Aktivitäten auszuführen (Interface 2). Für automatische Aktivitäten müssen die entsprechenden Anwendungen und Dienste aufgerufen werden (Interface 3). Falls ein Workflow unternehmensübergreifend agieren soll, muss es möglich sein, mit dem *Workflow-Enactment-Service* einer anderen Organisation zu kommunizieren (Interface 4).

Um Benutzer und Rollen des Systems zu verwalten, aber auch um die ordnungsgemäße Ausführung eines Prozesses zu überprüfen und gegebenenfalls Fehlerbehandlungen durchzuführen, sind Administrations- und Monitoring-Werkzeuge nötig (Interface 5).

2.2.3 Modellierung von Prozessen

Damit ein Prozess von einem Workflow-Management-System ausgeführt werden kann, muss er zunächst auf geeignete Weise modelliert werden. Dazu ist es notwendig, dass die auszuführenden Aktivitäten definiert werden und die potentielle Ausführungsrei-

henfolge festgelegt wird. In Hinblick darauf werden *Bedingungen* definiert, die in Abhängigkeit von konkreten Datenwerten den Prozessablauf beeinflussen. Bei einem Bestellvorgang sind dies beispielsweise die Lieferanschrift sowie Menge und Artikelnummern der bestellten Produkte. Die Daten sind jedoch meistens nicht statisch, sondern können während der Ausführung des Prozesses von Aktivitäten verändert werden. Daher ist die tatsächliche Ausführungsreihenfolge erst zur Laufzeit aus der potentiellen Ausführungsreihenfolge unter Berücksichtigung der Bedingungen und der konkreten Datenwerte berechenbar. [LR00]

Die Ausführung eines Prozesses ist somit zustandsbehaftet. Zu diesem Zustand gehört neben der Information, welche Aktivität gerade ausgeführt wird, auch die für die Ausführung von Aktivitäten und zur Auswertung von den Bedingungen benötigten Daten. Das *Prozessmodell* stellt also die Vorlage für konkrete Ausführungsinstanzen dar, die sich in ihrem Zustand unterscheiden können.

Teile der in einem Prozess verarbeiteten Daten werden manchmal nur von wenigen Aktivitäten verwendet. Daher ist es sinnvoll, zwischen Kontrollfluss und Datenfluss zu unterscheiden. Der *Kontrollfluss* definiert die Ausführungsreihenfolge der Aktivitäten eines Prozesses. Im Gegensatz dazu beschreibt der *Datenfluss* den Verlauf der Daten während der Prozessausführung [LR00]. Da nicht für jede Aktivität alle Daten benötigt werden, können so gegebenenfalls größere Mengen von Daten ausgelagert werden, um die Verarbeitung zu beschleunigen. Im Folgenden werden weitere Aspekte von Daten- und Kontrollfluss genauer betrachtet.

Datenfluss

Aktivitäten benötigen für die Ausführung bestimmte Daten als Parameter. Meist werden nach der Ausführung auch Daten als Ergebnis zurück geliefert. Da die für die Ausführung benötigten Daten auch als Eingabe interpretiert werden können, bilden sie einen sogenannten *Input-Container*. Das Ergebnis einer Aktivität bildet entsprechend den *Output-Container*. Die in dieser Arbeit verwendete graphische Notation für Input- und Output-Container ist in Abbildung 2.6 dargestellt. Daten aus dem *Output-Container* werden später ausgeführten Aktivitäten zur Verfügung gestellt, die diese dann in ihrem *Input-Container* abrufen können. Möglicherweise sind dabei Transformationen notwendig, da ein bestimmter Datentyp verlangt wird. So kann beispielsweise eine Aktivität

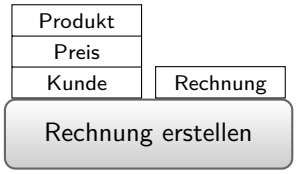


Abbildung 2.6: Input- und Output-Container (nach [Wes07]): Die Aktivität „*Rechnung erstellen*“ liest „*Produkt*“, „*Preis*“, und „*Kunde*“ (Input-Container), sie schreibt „*Rechnung*“ (Output-Container).

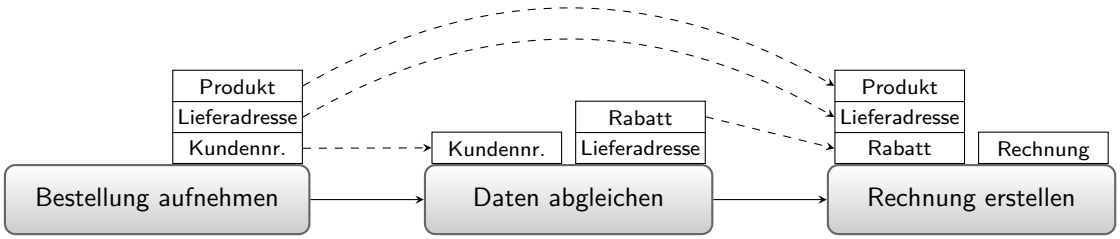


Abbildung 2.7: Explizite Datenflussmodellierung (nach [Wes07, S. 204]):
Die Aktivität „*Rechnung erstellen*“ liest die Variablen „*Produkt*“, „*Lieferadresse*“ sowie „*Rabatt*“ und schreibt „*Rechnung*“. Der verwendete Wert für „*Lieferadresse*“ wurde von der Aktivität „*Bestellung aufnehmen*“ geschrieben.

die aktuelle Temperatur in Grad Fahrenheit liefern, während eine andere einen Wert in Grad Celsius erwartet. Zusammenfassend werden die von einem Prozess benötigten Daten auch *Prozessdaten* genannt. [LR00]

Auf welche Art die Daten eines Prozesses zwischen den Containern verwaltet werden, regelt der Datenfluss. Es werden zwei Methoden zur Modellierung des Datenflusses unterschieden [ACKM04]. Der Ansatz des *Blackboards* (engl. für *Tafel*) ist besonders in traditionellen Programmiersprachen verbreitet. Das Blackboard besteht aus einer Menge von Variablen, denen ein veränderbarer Wert zugewiesen ist. Den Aktivitäten werden die benötigten Daten mit den Operationen Lesen und Schreiben zur Verfügung gestellt.

Die Daten können für verschiedene Sichtbarkeitsbereiche definiert sein. *Aktivitäts-Daten* sind nur innerhalb einer Aktivität sichtbar. Andere Aktivitäten – auch die desselben Prozesses – können nicht darauf zugreifen. Daten dieser Art werden entweder vom Workflow-Management-System nicht erfasst oder über die Container verfügbar gemacht. *Block-Daten* sind innerhalb eines Subprozesses sichtbar und *Workflow-Daten* innerhalb des gesamten Prozesses. Lediglich *Umgebungsdaten* können auch von anderen Prozessen verwendet werden. [RHEA04]

Im Gegensatz zum Blackboard wird bei einem *explizitem Datenfluss* genau definiert, welche Daten zwischen welchen Aktivitäten übertragen werden [ACKM04]. Bei dem in Abbildung 2.7 dargestellten sequentiellen Prozess ist beispielsweise definiert, dass zum Erstellen der Rechnung die Lieferadresse verwendet wird, die beim Aufnehmen der Bestellung angegeben wurde, und nicht diejenige, die in der Kundendatenbank hinterlegt ist und von der Aktivität „*Daten abgleichen*“ zurückgegeben wurde. Dateninteraktionen können aber auch zwischen einer Aktivität und einem Subprozess und sogar zwischen verschiedenen Prozessen erfolgen. Die Interaktion zwischen einem Prozess und dem Workflow-Management-System ist ebenso möglich. Dabei kann jeweils der konkrete Datenwert oder eine Referenz auf das Datum übermittelt werden. [Wes07]

Die explizite Modellierung des Datenflusses erlaubt zwar mehr Flexibilität, führt aber auch zu einer deutlich höheren Komplexität, wodurch die Modellierung unübersichtlich werden kann. Außerdem wird dabei implizit der Kontrollfluss geregelt, da aufgrund der

Datenabhängigkeiten eine Aktivität erst gestartet werden kann, wenn alle Aktivitäten, die im Datenfluss vorher auftreten, zuvor beendet sind. [ACKM04]

Datengetriebene Ansätze gehen einen Schritt weiter und verzichten auf die Modellierung des Kontrollflusses. Statt dessen werden lediglich die Datenabhängigkeiten zwischen den Aktivitäten festgelegt. Dieses Konzept kann insbesondere bei informationslastigen Prozessen eine effektivere Ausführung ermöglichen, da der starre Kontrollfluss gegebenenfalls aufgeweicht werden kann. So kann eine nachfolgende Aktivität möglicherweise schon vor Beendigung der vorherigen Aktivität gestartet werden, weil die benötigten Daten schon vor der Beendigung zur Verfügung gestellt wurden. [Wes07]

Steuerung des Kontrollflusses durch Bedingungen

Verschiedene Bedingungen können die Ausführung eines Prozesses beeinflussen. Dies ist notwendig, um auf bestimmte Eigenschaften eines konkreten Ablaufes reagieren zu können. So kann beispielsweise bei einem Reparaturprozess je nach Standort des defekten Gerätes und der verfügbaren Mitarbeiter eine Reparatur vor Ort oder eine Abholung des Gerätes durch einen Paketdienst initiiert werden. Es wird daher zwischen *potentiellem Kontrollfluss* und *tatsächlichem Kontrollfluss* unterschieden. Ersterer beschreibt, welche Aktivitäten zur Ausführung überhaupt in Frage kommen. Welche dann tatsächlich gestartet werden müssen, wird mit Hilfe sogenannter *Transitionsbedingungen* entschieden. Der tatsächliche Kontrollfluss hängt somit von der Auswertung der Transitionsbedingungen ab. [AH02]

Bevor eine Aktivität ausgeführt wird, muss eine *Aktivierungsbedingung* ausgewertet werden. Sie gibt an, wann die Aktivität gestartet werden darf. Dies ist beispielsweise sinnvoll, wenn eine Aktivität nur zu bestimmten Zeiten erfolgreich ausgeführt werden kann. Im Gegensatz dazu prüft die *Join-Bedingung*, ob die Aktivität überhaupt gestartet werden darf. Dies ist insbesondere zur Synchronisation von parallel ausgeführten Prozessteilen sinnvoll, um die erfolgreiche Abarbeitung aller parallelen Teile zu überprüfen. Mit Hilfe einer *Beendigungsbedingung* kann geprüft werden, ob die Aktivität vollständig bearbeitet oder lediglich unterbrochen wurde. [LR00]

2.2.4 Kontrollflussstrukturen

Der potentielle Kontrollfluss eines Prozesses wird modelliert, indem die möglichen Übergänge zwischen Aktivitäten definiert werden. Hierfür stehen je nach Modellierungssprache verschiedene Konstrukte zur Verfügung, von denen die elementaren im Folgenden vorgestellt werden. Die Konstrukte können rekursiv angewendet werden, so dass komplexe Strukturen entstehen. Zur Veranschaulichung wird auf eine einfache graphische Notation (vgl. [Wes07]) zurückgegriffen.

Sequentielle Ausführung

Die *sequentielle Ausführung* stellt die einfachste Verbindung von zwei Aktivitäten dar. Sie hat zur Folge, dass die Aktivitäten hintereinander ausgeführt werden [RHAM06]. Dies ist notwendig, wenn zwischen ihnen eine logische Abhängigkeit existiert. So muss in einem Prozess in der Buchhaltung beispielsweise zuerst die Aktivität „*Rechnung erstellen*“ ausgeführt werden, bevor „*Rechnung versenden*“ gestartet werden darf. Abbildung 2.8 zeigt die Sequenz schematisch.

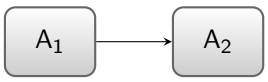


Abbildung 2.8: Sequentielle Ausführung

Parallele Ausführung

Der Kontrollfluss zwischen mehreren Aktivitäten kann auch als *parallele Ausführung* modelliert werden. Die tatsächliche Ausführungsreihenfolge der betroffenen Aktivitäten erfolgt hierbei nicht deterministisch: Sie können in einer beliebigen Reihenfolge hintereinander oder aber auf verschiedenen Ressourcen gleichzeitig ausgeführt werden [AH02]. Im Beispiel in Abbildung 2.9 wird also entweder C_1 nach B_1 oder B_1 nach C_1 ausgeführt oder es werden B_1 und C_1 gleichzeitig ausgeführt. Die tatsächliche Ausführungsreihenfolge kann auch von Datenabhängigkeiten zwischen den parallel ausgeführten Aktivitäten beeinflusst werden.

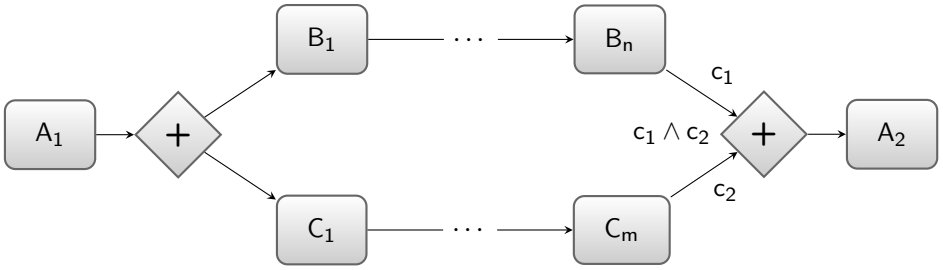


Abbildung 2.9: Parallele Ausführung

Die Verzweigung des Kontrollflusses in mehrere Pfade wird als *Split* oder *Fork* bezeichnet. Nur wenn tatsächlich mehrere Transitionsbedingungen zu „*wahr*“ ausgewertet werden, können mehrere Aktivitäten gleichzeitig gestartet werden und es kommt zur Parallelität. Dieser Fall wird häufig auch als *AND-Split* bezeichnet. Das Zusammenführen mehrerer Pfade wird *Join* genannt. Wurden die zusammengeführten Pfade parallel ausgeführt, wird der Begriff *AND-Join* verwendet. Dieses Konstrukt dient als Synchronisation des Kontrollflusses, da gewartet wird bis alle parallelen Ausführungspfade beendet sind. Mit einer *Join-Bedingung* wird nun geprüft, ob die Bearbeitung wie gewünscht erfolgt ist. Ist dies nicht der Fall, werden die nachfolgenden Aktivitäten nach

dem Prinzip *Dead-Path-Elimination* (vgl. Abschnitt 2.2.6) gesperrt, um eine erfolgreiche Terminierung des Prozesses sicherzustellen [LR00]. Im Beispiel muss also $c_1 \wedge c_2$ wahr sein, damit A_2 ausgeführt wird.

Zwar drückt die parallele Ausführung aus, dass Aktivitäten paralleler Ausführungspfade zeitlich unabhängig voneinander ausgeführt werden dürfen. Dennoch können Datenabhängigkeiten zwischen den Pfaden existieren. Es kann also sein, dass die Aktivität C_4 ein Datum benötigt, welches eine Aktivität eines anderen parallelen Pfades erzeugt, beispielsweise B_2. Die Aktivität C_4 kann also erst nach erfolgter Ausführung von B_2 gestartet werden. Solche Abhängigkeiten können jedoch auch explizit modelliert werden, falls ein Prozess aufgrund von Randbedingungen eine partielle Ordnung zwischen Aktivitäten vorgibt, die sich nicht aus den Datenabhängigkeiten ergibt.

Weitere Varianten des Joins sind denkbar. So wird beim *Discriminator* bereits nach Beendigung des ersten parallelen Ausführungspfades der dem Discriminator folgende Kontrollfluss ausgeführt. Sobald die anderen parallelen Ausführungspfade beendet sind, werden diese verworfen. Beim *Cancelling Discriminator* werden noch ausgeführte Pfade abgebrochen sobald der erste Ausführungspfad den Discriminator erreicht. Der *Partial Join* stellt die Verallgemeinerung des Discriminators dar. Hier kann beispielsweise festgelegt werden, dass zwei von drei parallelen Pfaden erfolgreich beendet werden müssen bis die Ausführung des Kontrollflusses fortgesetzt wird. [RHAM06]

Selektive Ausführung

Soll lediglich ein Pfad von mehreren im Kontrollfluss weiterverfolgt werden, wird die *selektive Ausführung* verwendet. Da es sich wie bei der parallelen Ausführung um eine Verzweigung handelt, wird dieses Konstrukt ebenfalls *Split*, zur Differenzierung jedoch genauer *XOR-Split* oder *exklusives OR-Split*, genannt [RHAM06]. Damit nur ein Pfad ausgewählt wird, müssen die Transitionsbedingungen so formuliert werden, dass immer genau eine zu „*wahr*“ ausgewertet wird. Das Konstrukt *XOR-Join* führt mehrere alternative Pfade wieder zusammen. Wurde die letzte Aktivität eines alternativ ausgeführten Pfades erfolgreich beendet, so wird sofort die Join-Aktivität ausgeführt. Konnte also die Transitionsbedingung c_1 im Beispiel von Abbildung 2.10 zu „*wahr*“ ausgewertet werden, so wird nach dem Beenden von B_n die Aktivität A_2 gestartet.

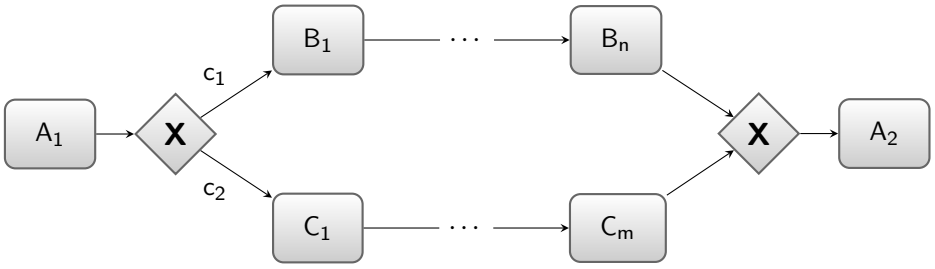


Abbildung 2.10: Selektive Ausführung

Wenn je nach aktuellen Datenwerten bei einem *Split* eine oder mehrere Transitionsbedingungen zu „*wahr*“ ausgewertet werden und sich dies somit wie ein *XOR-Split* und ein *AND-Split* verhält, wird es als *OR-Split* oder auch *Multi-Choice* bezeichnet. Hierbei erfordert der anschließende *Join* jedoch erhöhte Aufmerksamkeit. Da normalerweise nachfolgende Aktivitäten sofort ausgeführt werden, sobald der *XOR-Join* erreicht ist, würde dies mehrfach passieren, wenn mehrere Transitionsbedingungen eines *OR-Splits* zu „*wahr*“ ausgewertet wurden. Im Beispiel hieße dies, dass wenn c_1 und c_2 wahr sind, A_2 und alle nachfolgenden Aktivitäten zweimal ausgeführt werden. Wird statt dessen ein *AND-Join* verwendet, so kann es passieren, dass die Synchronisation nicht terminiert, da auf die Beendigung eines Pfades gewartet wird, der nie gestartet wurde. Um die genannten Probleme zu vermeiden, muss der Synchronisationspunkt erkennen, welche Entscheidung der *Split* getroffen hat. Dazu kann der *Join* explizit als *Synchronizing Join* modelliert werden [RHAM06]. Es ist jedoch auch möglich, einen *OR-Split* durch einen *AND-Split* und mehrere *XOR-Splits* zu simulieren [AHD05].

Iterative Ausführung

Zwar wird in der Regel bei der Ausführung eines Prozesses jede Aktivität nur einmal gestartet, es gibt jedoch Fälle, in denen die Wiederholung eines Prozessteils sinnvoll ist. So kann es beispielsweise notwendig sein, bestimmte Aktivitäten solange wiederholt zu starten, bis ein gewünschtes Ergebnis erreicht ist. Theoretisch lässt sich Iteration durch das bereits vorgestellte Mittel der selektiven Ausführung realisieren. Jedoch können hierbei ähnliche Probleme auftreten, wie sie im vorherigen Absatz diskutiert wurden. Daher wird manchmal gefordert, dass ein Prozessmodell zyklenfrei ist [LR00]. Iteration wird dann über ein eigenes Konstrukt realisiert, welches mehrere Aktivitäten zusammenfasst, die solange ausgeführt werden, bis eine Abbruch-Bedingung erfüllt ist. Dies entspricht dem aus klassischen Programmiersprachen bekannten Konzept `repeat ... until`. Je nach Modellierungssprache kann auch ein Konstrukt existieren, dass dem klassischen `while` entspricht (vgl. Abbildung 2.11).

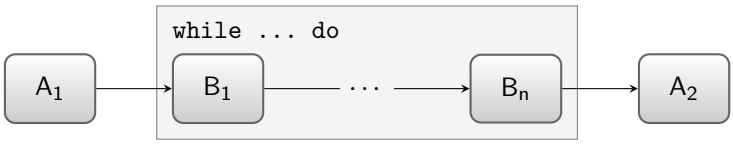


Abbildung 2.11: Iterative Ausführung

Mehrfache Ausführung von Aktivitäten

In manchen Fällen ist es notwendig, eine einzelne Aktivität mehrmals mit verschiedenen Parametern auszuführen. Dies ist der Fall, wenn dieselbe Aufgabe von mehreren Personen ausgeführt werden muss – beispielsweise weil zur Qualitätssicherung eine mehrfache Prüfung durch verschiedene Personen erforderlich ist – oder weil bei einem

Bestellprozess für Produkte verschiedener Kategorien eine separate Abfertigung notwendig ist [Wes07, AHD05]. Die Aktivitäten können in diesen Fällen parallel bearbeitet werden. Aalst et al. unterscheiden dabei vier Situationen [AHKB03]: (a) Es ist keine Synchronisation der Aktivitäten erforderlich und der Prozess kann sofort weitergeführt werden, (b) es ist bereits zur Entwicklungszeit bekannt, wieviele Aktivitäten ausgeführt werden, (c) zur Laufzeit kann vor der Ausführung der Aktivitäten ermittelt werden, wie viele Aktivitäten auszuführen sind oder (d) während der Ausführung der Aktivitäten können dynamisch weitere gestartet werden, so dass erst nach Beendigung aller Instanzen der Prozess fortgeführt werden darf. Abbildung 2.12 zeigt die mehrfache Ausführung von Aktivitäten schematisch.

Abbildung 2.12: Mehrfache Ausführung einer Aktivität

2.2.5 Subprozesse

Subprozesse sind Prozesse, die von einem übergeordneten Prozess gestartet werden. Dies ist beispielsweise nötig, wenn ein Prozess Unternehmensgrenzen überschreitet und somit die Workflow-Management-Umgebung verlässt. Dazu startet der Hauptprozess einen Subprozess in dem Workflow-Management-System des anderen Unternehmens. Ein solcher Subprozess wird *remote* genannt. Wird der Subprozess hingegen auf demselben Workflow-Management-System gestartet, so ist er *lokal* [LR00]. Dies ist unter anderem sinnvoll, um einen komplexen Prozess auf verschiedenen Abstraktionsebenen zu modellieren oder um wiederverwendbare Komponenten zu erzeugen.

Die *Workflow Management Coalition* hat sich zum Ziel gesetzt, heterogene Workflow-Management-Systeme interoperabel zu machen und verwendet Subprozesse daher, um systemübergreifende Prozesse zu realisieren. Dazu wurden vier Szenarien entwickelt, die im Folgenden vorgestellt werden. [Hol95]

Verbinden eigenständiger Prozesse

Bei diesem einfachen Fall werden zwei Aktivitäten von zwei unterschiedlichen Prozessen miteinander verbunden. Erreicht der Kontrollfluss eine solche Aktivität, so wird der andere Prozess als Subprozess mit der verknüpften Aktivität gestartet. Der Subprozess wird nun unabhängig von dem Mutterprozess ausgeführt und es erfolgt keine weitere Synchronisation. Im Beispiel von Abbildung 2.13 sind die Prozesse A und B miteinander verkettet, d. h. Prozess B wird ausgeführt, sobald Prozess A beendet wird. Dies passiert, weil die Aktivitäten A_n und B_1 miteinander verknüpft sind.

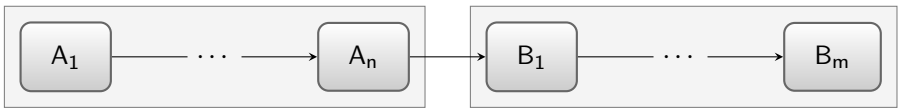


Abbildung 2.13: Zwei verkettete Prozesse (nach [Hol95, S. 38])

Prozesshierarchien

Das intuitive Verständnis von Subprozessen entspricht einem hierarchischen Modell. Hierbei wird ein Prozess in einer Aktivität gekapselt. Wird die Aktivität gestartet, so erzeugt sie eine neue Ausführungsinstanz des Subprozesses, die dann ausgeführt wird. Dies erfolgt synchron, d. h. die Aktivität ist erst beendet, sobald der Subprozess beendet ist. Auf diese Weise können Hierarchien mit mehreren Ebenen gebildet werden.

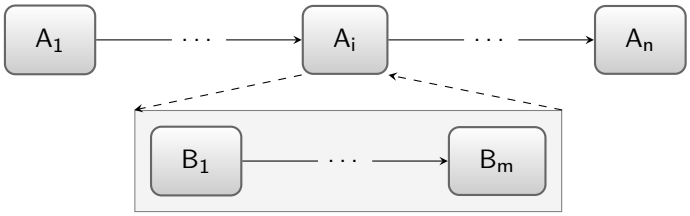


Abbildung 2.14: Hierarchische Schachtelung von Prozessen (nach [Hol95, S. 39])

Im Beispiel von Abbildung 2.14 wird bei Erreichen von Aktivität A_i der Subprozess B gestartet. Sobald dessen Aktivitäten B_1 bis B_m ausgeführt wurden, gilt die Aktivität A_i als beendet und der Prozess A wird weiter ausgeführt.

Vermaschte Prozesse

Im Gegensatz zum ersten Szenario sind hier die Prozesse im Kontrollfluss nicht klar getrennt. Es handelt sich um ein *Peer-to-Peer-Modell*, da nicht immer entschieden werden kann, welcher Prozess den anderen übergeordnet ist. Statt dessen ist jede Aktivität einem der beteiligten Prozesse zugeordnet. Dies hat zur Folge, dass der Gesamtprozess gegebenenfalls häufig zwischen mehreren Workflow-Engine wechseln muss. Damit verbunden ist auch die Notwendigkeit des Abgleichs der mit dem Prozess verknüpften Daten.

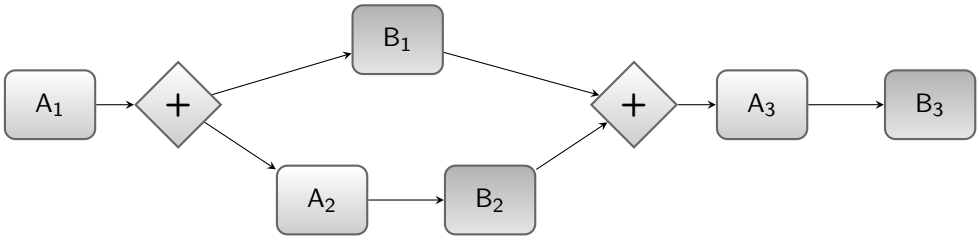


Abbildung 2.15: Zuordnung von Aktivitäten zu unterschiedlichen Subprozessen (aus [Hol95, S. 40])

Abbildung 2.15 zeigt die Prozesse A und B, die auf verschiedenen Workflow-Engines ausgeführt werden. Während der Ausführung müssen diese auf geeignete Weise kommunizieren, um über die Beendigung einer Aktivität zu informieren und dessen Ergebnisse bekannt zu geben.

Parallele synchronisierte Prozesse

In diesem Szenario werden zwei Prozesse im wesentlichen unabhängig voneinander parallel ausgeführt. Sie können jedoch über Synchronisationspunkte miteinander verbunden werden. Erreicht ein Prozess einen solchen Punkt, so muss er warten, bis der andere Prozess ebenfalls diesen Synchronisationspunkt erreicht. Beide Prozesse können dann beispielsweise Daten austauschen und werden anschließend wieder unabhängig voneinander ausgeführt, bis der nächste Synchronisationspunkt erreicht ist.

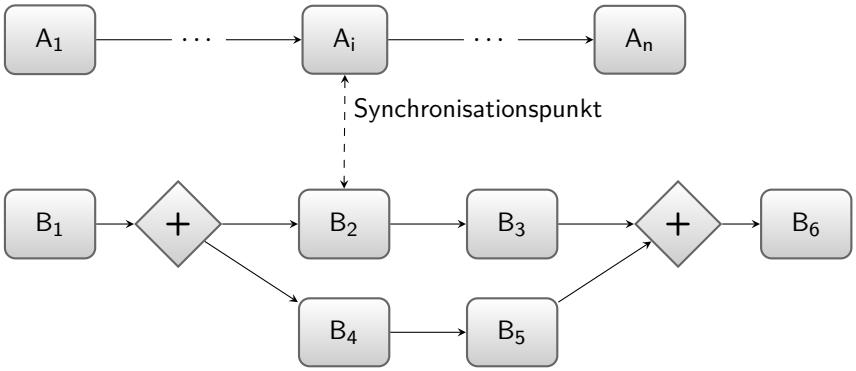


Abbildung 2.16: Synchronisation von parallel ausgeführten Prozessen (nach [Hol95, S. 41])

Die ansonsten unabhängigen Prozesse A und B von Abbildung 2.16 sind lediglich über einen Synchronisationspunkt verbunden. Erreicht Prozess A die Aktivität A_i, bevor Prozess B die Aktivität B_2 erreicht, so wird die Ausführung von A pausiert. Erst wenn Aktivität B_2 erreicht wird, werden A_i und B_2 gestartet. Man beachte, dass nur ein paralleler Pfad von Prozess B von der Synchronisation betroffen ist.

2.2.6 Konzepte zur Fehlerbehandlung

Die Ausführung von Prozessen kann nicht immer erfolgreich durchgeführt werden. Tritt ein Fehler auf, so muss angemessen reagiert werden, um den Schaden begrenzen zu können. Dazu sind geeignete Methoden der Fehlerbehandlung notwendig, die im Folgenden diskutiert werden. Zunächst werden allgemeine Strategien zum Verhalten eines Prozesses im Fehlerfall behandelt. Anschließend wird die Unterdrückung von toten Pfaden vorgestellt. Es handelt sich dabei um ein Konzept, das notwendig ist, um endloses Warten auf nicht erfolgreiche parallele Ausführungspfade bei der Synchronisation zu verhindern. Der Abschnitt schließt mit der Beschreibung von Transaktionen, die ebenfalls eine Form der Fehlerbehandlung darstellen, denn sie garantieren, dass sich auch im Fehlerfall das System in einem konsistenten Zustand befindet.

Fehlerbehandlung

Bei der Abwicklung von Prozessen können diverse Fehler auftreten. So kann es beispielsweise passieren, dass bei der Ausführung einer Aktivität ein Problem festgestellt und als Fehler an den Prozess gemeldet wird oder es kann eine Aktivität nicht ordnungsgemäß terminieren, so dass eine Antwort komplett ausbleibt [LR00]. Letzteres lässt sich nur mit Hilfe einer Vereinbarung, dass die Aktivität innerhalb eines bestimmten Zeitraumes antworten muss, feststellen. Möglicherweise kann eine Aktivität aber auch gar nicht gestartet werden, weil die entsprechende Implementierung nicht gefunden wurde oder der Benutzer nicht authorisiert ist, diese auszuführen [ACKM04].

Eine Vielzahl von Ursachen für die Fehler sind denkbar. Brambilla et al. teilen die möglichen Fehlerquellen daher in drei Kategorien ein [BCCT05]: falsches oder unangemessenes Benutzerverhalten, semantische Fehler der Prozesslogik, welche oft erst während der Ausführung des Prozesses sichtbar werden, und Systemfehler, die auf Probleme des Workflow-Management-Systems oder auf Übertragungsfehler im Netz zurückzuführen sind. In jedem Fall ist es sinnvoll, auf solche Ausnahmesituationen angemessen zu reagieren.

Es gibt verschiedene Ansätze, wie Fehlerbehandlungen umgesetzt werden können. Bei dem *flussbasierten Ansatz* wird die Abfrage, ob ein Fehler aufgetreten ist, direkt im Kontrollfluss modelliert. Dies hat jedoch den Nachteil, dass das Prozessmodell schnell unübersichtlich wird. Aus diesem Grund spezifizieren Strukturen nach dem Prinzip *Event-Condition-Action* (*ECA*) bei den *regelbasierten Ansätzen* das Verhalten im Fehlerfall. Wenn ein bestimmter Fehler auftritt (*Event*), entscheidet eine Vorbedingung (*Condition*), welche Aktion (*Action*) ausgeführt werden soll. [ACKM04]

Das u. a. von Java bekannte Konstrukt *try-catch* kann in ähnlicher Weise auch bei Workflow-Prozessen zum Einsatz kommen. Dazu werden Aktivitäten oder Gruppen von Aktivitäten mit einem Fehlerbehandlungsblock versehen, der im Falle von bestimmten Fehlertypen ausgeführt wird. Fehler, die so nicht behandelt werden, werden rekursiv an den umgebenen Bereich weitergereicht und dort behandelt. [M$^+$08]

Was innerhalb der so gestarteten Fehlerbehebung passiert, hängt von der konkreten Situation ab. Das klassische, von Datenbanken bekannte Zurücksetzen auf einen alten fehlerfreien Zustand ist bei Prozessen häufig nicht anwendbar, da viele Aufgaben Auswirkungen in der realen Welt haben und daher nicht ungeschehen gemacht werden können. Dies ist insbesondere der Fall, wenn eine manuelle Aktivität, wie das Versenden eines Briefes, ausgeführt wurde. Oft können aber durch das Starten von zusätzlichen Aktivitäten die Konsequenzen ausgeglichen werden – man spricht auch von *Kompensation*. Anschließend ist es möglich, die fehlgeschlagenen Aktivitäten erneut auszuführen. [ACKM04]

Kompensationen sind jedoch nicht bei jeder Aktivität möglich und erfordern eine anwendungsspezifische Modellierung. Dabei können sie auf unterschiedlichen Stufen der Granularität eingesetzt werden. Im einfachsten Fall wird jeder Aktivität eine kompen-

sierende Aktivität zugeordnet. Diese werden dann im Fehlerfall in umgekehrter Reihenfolge gestartet. Darüber hinaus können Kompensationen auch für mehrere Aktivitäten oder den gesamten Prozess definiert werden. [Ley95]

Treten weiterhin Fehler auf, so ist es denkbar, einen alternativen Pfad im Kontrollfluss zu wählen [LSKM00]. Auf diese Weise ist es möglich, eine Aufgabe mit anderen Mitteln zu bewältigen, die zwar mehr Aufwand erfordern, dafür aber trotzdem die erfolgreiche Ausführung des Prozesses erlauben. Schlagen alle Fehlerbehebungsmaßnahmen fehl, so bleibt noch der Weg, den Benutzer zur Fehlerbehebung heranzuziehen oder den Prozess als fehlgeschlagen zu beenden [LSKM00].

Unterdrückung von toten Pfaden

Aufgrund eines Fehlers bei der Ausführung von Aktivitäten oder der fehlenden Befriedigung von Vor- oder Nachbedingungen kann eine Transitionsbedingung zu „*falsch*“ ausgewertet werden. Dies hat zur Konsequenz, dass die mit der Transitionsbedingung verknüpfte Aktivität nicht gestartet werden kann. Insbesondere bei der Synchronisation von parallelen Ausführungspfaden muss jedoch verhindert werden, dass auf einen Pfad gewartet wird, der aufgrund einer negativen Transitionsbedingung nie beendet wird. Dies soll das Verfahren *Dead-Path-Elimination* (*Unterdrückung von toten Pfaden*) verhindern, welches im Folgenden vorgestellt wird.

Eine Aktivität wird als *tot* bezeichnet, wenn sie in einer Sequenz auftritt und die Transitionsbedingung zu „*falsch*“ auswertet oder wenn an der Aktivität ein Join durchgeführt wird, dessen Join-Bedingung zu „*falsch*“ auswertet. Wird nun festgestellt, dass eine Aktivität tot ist, so wird den Transitionsbedingungen aller Aktivitäten, die im Kontrollfluss direkt nach der toten Aktivität vorkommen, der Wahrheitswert „*falsch*“ zugeordnet. Dies hat zur Konsequenz, dass ab der toten Aktivität alle Aktivitäten bis zum nächsten Join oder dem Ende des Prozesses ebenfalls tot sind. [LR00]

Bei der Synchronisation paralleler Ausführungspfade wird nur auf nicht-tote Pfade gewartet. Die Auswertung der Join-Bedingung kann in manchen Fällen auch dann positiv sein, wenn unter den synchronisierten Pfaden einige tot sind. Dies ist beispielsweise der Fall, wenn die Ausführung eines Pfades optional ist. Durch die Unterdrückung von toten Pfaden wird somit sichergestellt, dass der Prozess nicht in eine dauerhafte Wartesituation gerät.

Transaktionen

Transaktionen dienen der sicheren und konsistenten Ausführung von Operationen trotz gleichzeitiger Zugriffe auf die Daten durch andere Benutzer und auftretender Fehler [HR01]. Klassisch ist eine Transaktion eine Menge von Operationen, welche die sogenannten *ACID-Eigenschaften Atomarität*, *Konsistenz*, *Isolation* und *Dauerhaftigkeit* erfüllt. Aufgrund der Atomarität wird eine Transaktion als unteilbare Aktion betrachtet. Es

müssen daher entweder alle oder gar keine Operationen ihre Wirkung entfalten. Außerdem muss sich das System immer in einem konsistenten Zustand befinden, insbesondere auch dann, wenn die Transaktion nicht erfolgreich war. Die Isolation garantiert, dass sich Transaktionen nicht gegenseitig beeinflussen, auch wenn mehrere parallel ausgeführt werden – in ihrer Sicht auf das System wird nur eine Transaktion zur Zeit ausgeführt. Wenn eine Transaktion erfolgreich beendet wird, sind alle Änderungen dauerhaft gespeichert und bleiben auch bei einem Systemausfall erhalten. [GR93]

Ein Transaktions-Manager kümmert sich um die Einhaltung dieser Eigenschaften. Anwendungen nehmen diesen Dienst in Anspruch und müssen somit die in vielen Fällen sehr aufwändige Fehlerbehandlung nicht selbst vornehmen, was zu einer erheblichen Erleichterung bei der Entwicklung von transaktionalen Anwendungen führt [HR01].

Transaktionen, welche die ACID-Eigenschaften erfüllen, werden auch als *flache Transaktionen* bezeichnet, da sich alle Operationen innerhalb der Transaktion auf der gleichen Ebene befinden [GR93]. Zudem müssen alle Operationen erfolgreich sein, damit die Transaktion festschreibt. Tritt ein Fehler auf, gehen auch die Ergebnisse von erfolgreich ausgeführten Operationen verloren und müssen wiederholt werden. In Datenbanksystemen, für die flache Transaktionen entwickelt wurden, ist dies von geringer Bedeutung, da die Transaktionen aus sehr feingranularen Lese- und Änderungsoperationen bestehen. Aktivitäten eines Workflow-Prozesses führen jedoch mitunter sehr komplexe Aktionen aus, die auch physikalische Auswirkungen haben können [ACKM04]. Außerdem dauert die Ausführung von Aktivitäten normalerweise deutlich länger als die Ausführung einzelner Operationen bei einer Datenbank, was somit zu lang andauernden Sperren von Ressourcen führen würde [LR00]. Flache Transaktionen können somit in der Regel in Prozessen nicht effizient eingesetzt werden.

Es wurden daher eine Reihe von Erweiterungen des klassischen Transaktionskonzeptes entwickelt. Da gerade die bei flachen Transaktionenen fehlende Binnenstruktur viele der oben genannten Probleme hervorruft, bestehen *geschachtelte Transaktionen* aus einer Menge von Sub-Transaktionen, die hierarchisch in Form einer Baumstruktur angeordnet sind. Im Fehlerfall können daher Sub-Transaktionen einzeln zurückgesetzt werden [Mos81]. Ein weiteres wesentliches Konzept ist das vorzeitige Festschreiben einzelner Sub-Transaktionen, wie dies bei der *offen geschachtelten Transaktion* im Gegensatz zur *geschlossen geschachtelten Transaktion* erfolgt. Muss nun nach dem Festschreiben jedoch die Transaktion abgebrochen werden, so ist es nötig, die erfolgten Änderungen mit Hilfe von Kompensationen wieder rückgängig zu machen – die Transaktion kann also keine Isolation gewährleisten [HR01].

Sagas verwenden ebenfalls diese Konzepte und sind dabei recht einfach zu implementieren sowie zu nutzen. In nur einer Hierarchieebene werden hier Sub-Transaktionen eingesetzt, die aus je einer klassischen Transaktion und je einer dazugehörigen kompensierenden Transaktion bestehen. Im Fehlerfall werden die kompensierenden Transaktio-

nen aller erfolgreich ausgeführten Transaktionen in umgekehrter Reihenfolge gestartet. [GMS87]

Speziell für Workflow-Prozesse existieren verschiedene Ansätze, bei denen die Grenzen der Transaktion in Form von Sphären definiert werden, welche mehrere Aktivitäten eines Prozesses beinhalten. Transaktionsgarantien beziehen sich dann immer auf die Menge von Aktivitäten innerhalb der Sphäre. Bei *atomaren Sphären* muss die Implementierung der Aktivitäten das Zurücksetzen unterstützen, da hier das Festschreiben erst erfolgt, wenn der Kontrollfluss die Sphäre verlassen hat [LR00]. *Kompensationssphären* verbinden das Konzept der Kompensation mit der flexiblen Transaktionsdefinition bei atomaren Sphären. Aktivitäten schreiben daher sofort fest und müssen somit keine transaktionale Untersützung besitzen. Ihnen muss jedoch eine Kompensations-Aktivität zugeordnet werden, die im Falle eines Abbruches der Kompensationssphäre wie bei den Sagas in umgekehrter Reihenfolge ausgeführt werden [Ley95].

2.3 Prozesse auf mobilen Geräten

Mobilität ist in heutigen globalen Unternehmen von großer Bedeutung. Dies betrifft nicht nur die Mitarbeiter im Außendienst, denn auch auf Geschäftsreisen existiert ein großer Bedarf an Unterstüzung der zu bewältigenden Aufgaben durch die Informationstechnologie. Gängige Büroanwendungen, wie Textverarbeitung und E-Mail, sind zwar bereits auf mobilen Geräten verfügbar; die Orientierung an Geschäftsprozessen und Einbindung in Workflows wird jedoch häufig nicht unterstützt, obwohl deren Bedeutung zunimmt. [CJK+08]

Das größte Problem bei der Teilnahme von mobilen Geräten an der Ausführung von Prozessen stellt die Mobilkommunikation dar. Klassische Workflow-Management-Systeme basieren auf einer Client-Server-Architektur, bei der die Verwaltung des Prozesses zentral erfolgt und die Clients ständig mit dem Server verbunden sein müssen [AGK+95]. Aufgrund der unzuverlässigen Netze, auf die mobile Geräte zurückgreifen müssen (vgl. Abschnitt 2.1.2), können diese Verbindungen bei mobiler Nutzung jedoch nicht gewährleistet werden.

Viele Ansätze zur Einbeziehung von mobilen Geräten in die Ausführung von Prozessen beschäftigen sich daher mit *disconnected clients*, also Szenarien, bei denen keine dauerhafte Netzverbindung besteht. Auf welche Art mobile Geräte hierbei an Prozessen teilnehmen können, lässt sich in drei Stufen einteilen. Auf der niedrigsten Stufe werden nur einzelne Aktivitäten von dem mobilen Gerät ausgeführt, der Prozess wird aber zentral verwaltet. Auf der nächsten Stufe kann das mobile Gerät autonom Teile eines Prozesses ausführen. Vollständige Selbstständigkeit und Unabhängigkeit von einer zentralen Verwaltung wird schließlich auf der dritten Stufe erreicht. Im Folgenden werden diese Ansätze ausführlicher diskutiert.

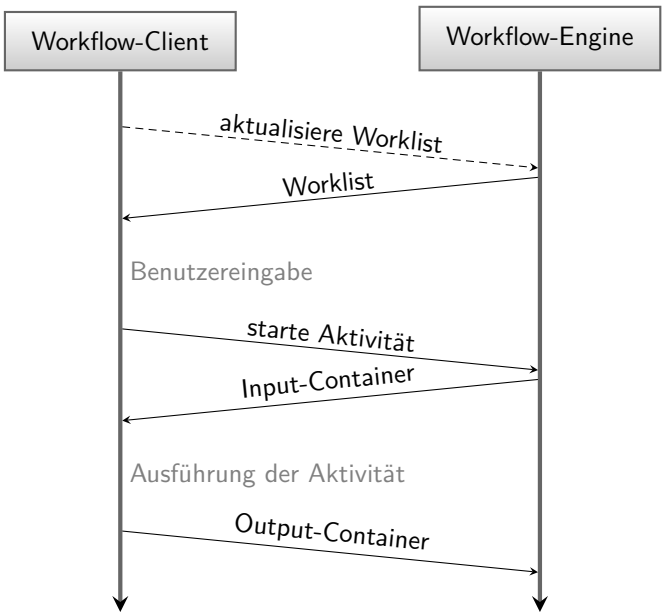


Abbildung 2.17: Kommunikation zwischen Workflow-Client und Workflow-Engine

2.3.1 Ausführung einzelner Aktivitäten

Bei der einfachsten Methode zur Integration von mobilen Geräten in die Ausführung von Prozessen ist eine Workflow-Client-Anwendung auf dem mobilen Gerät installiert. Die Anwendung kommuniziert mit einer zentralen Workflow-Engine, welche die Kontrolle über die auszuführenden Prozesse besitzt. Dies entspricht dem klassischen Workflow-Szenario mit zentralem Server (zur Lastverteilung gegebenenfalls auch mehrere Server), wie es auch in nicht-mobilen Umgebungen realisiert wird (vgl. [BD99]) und im Workflow-Referenzmodell vorgesehen ist (vgl. Abbildung 2.5).

Dem Benutzer des mobilen Gerätes oder dem mobilen Gerät selbst sind eine oder mehrere Rollen zugeordnet. Der Workflow-Client kann mit dieser Information bei der Workflow-Engine eine Liste von Aktivitäten abfragen, die gemäß dem Kontrollfluss der ausgeführten Prozesse gestartet werden sollen. Diese Liste wird dem Benutzer des mobilen Gerätes in einer *Worklist* präsentiert, von der eine Aktivität zur Ausführung ausgewählt werden kann. Die Auswahl wird der Workflow-Engine mitgeteilt. Diese muss nun die Aktivität sperren, damit sie nicht durch andere Benutzer ein zweites Mal ausgeführt wird. Die für die Ausführung der Aktivität benötigten Daten (Input-Container) werden nun an den Workflow-Client übertragen. Nach erfolgter Ausführung werden die Ergebnisse (Output-Container) an die Workflow-Engine übermittelt, welche nun die Prozessverarbeitung fortführt. [LR00]

Abbildung 2.17 zeigt die während dieses Ablaufes zu übertragenden Nachrichten. Die erste Nachricht ist optional, da die Worklist auch automatisch, also vom Server initiiert, aktualisiert werden könnte. Während der Benutzereingabe und der Ausführung der Aktivität auf dem mobilen Gerät ist keine Verbindung zum Workflow-Management-System

notwendig. Je nach Dauer der Aktivitäten und der Dauer des gesamten Prozesses sind diese einzelnen Phasen allerdings sehr kurz im Verhältnis zum gesamten Prozess. Dies hat zur Konsequenz, dass während der Ausführung eines Prozesses ständig Nachrichten zwischen dem mobilen Gerät und der Workflow-Engine ausgetauscht werden müssen. Ist die Verbindung für längere Zeit nicht verfügbar, so kann der Prozess nicht mit Hilfe des mobilen Gerätes weiter verarbeitet werden.

Neben der Häufigkeit der Datenübertragungen spielt auch die Menge der zu übertragenen Daten eine große Rolle, da bei der Mobilkommunikation häufig auf langsame Verbindungen zurückgegriffen werden muss. Die Menge der zu übertragenen Daten hängt hauptsächlich von der Größe der Daten-Container und somit vom Typ der Aktivität ab. Optimierungen sind dann möglich, wenn ein mobiles Gerät mehrere Aktivitäten eines Prozesses ausführt. Insbesondere bei sequentiell ausgeführten Aktivitäten besteht der Input-Container einer nachfolgenden Aktivität oft aus einem Großteil des Output-Containers von einer vorhergehenden Aktivität. Mit Hilfe von Cache-Mechanismen können hier unnötige Übertragungen zwischen mobilem Gerät und der Workflow-Engine vermieden werden. [Haa03]

In der Architektur von Alonso et al. kann eine längere Abwesenheit der Netzverbindung besser genutzt werden, falls diese vom Benutzer vorausgesehen wird – beispielsweise weil eine Flugreise ansteht. Der Benutzer kann dabei vor dem Trennen der Netzverbindung mehrere Aktivitäten aus der Worklist zur späteren Bearbeitung vormerken. Das System lädt nun alle erforderlichen Daten auf das mobile Gerät, damit diese später auch ohne Netzverbindung zur Ausführung der Aktivitäten verfügbar sind. Sobald die Netzverbindung wieder hergestellt wurde, werden die Ergebnisse der ausgeführten Aktivitäten an den Server übermittelt. [AGK+95]

Dazu muss eine weitere Komponente auf dem mobilen Gerät verfügbar sein, die sich in die Kommunikation zwischen den Client und den Server schaltet und somit transparent für den Server arbeitet. In Abbildung 2.18 ist der verallgemeinerte Nachrichtenaustausch zwischen den drei Komponenten dargestellt. Der lokale Worfklow-Dienst übersetzt die Anforderung einer Sperre in das Starten einer Aktivität. Dadurch werden die für die Ausführung der Aktivität benötigten Daten auf das mobile Gerät geladen. Möchte der Benutzer eine Aktivität starten, wenn keine Netzverbindung mehr besteht, so kann der lokale Workflow-Dienst die entsprechenden Daten bereitstellen. Im Gegensatz zum vorherigen Ansatz können so während der Zeit ohne Netzverbindung mehrere Aktivitäten hintereinander ausgeführt werden. [AGK+95]

Aber auch dieser verbesserte Ansatz ist noch stark beschränkt, da zum Einen ohne verfügbare Netzverbindung nur die Aktivitäten ausgeführt werden können, welche zuvor vom Benutzer zur späteren Ausführung markiert wurden. Zum Anderen kommen dafür nur solche Aktivitäten in Frage, die im Kontrollfluss des Prozesses zur Ausführung anstehen. Nur dann sind nämlich sämtliche zur Ausführung der Aktivitäten benötigten Daten verfügbar und es ist sichergestellt, dass die Aktivität wirklich ausgeführt

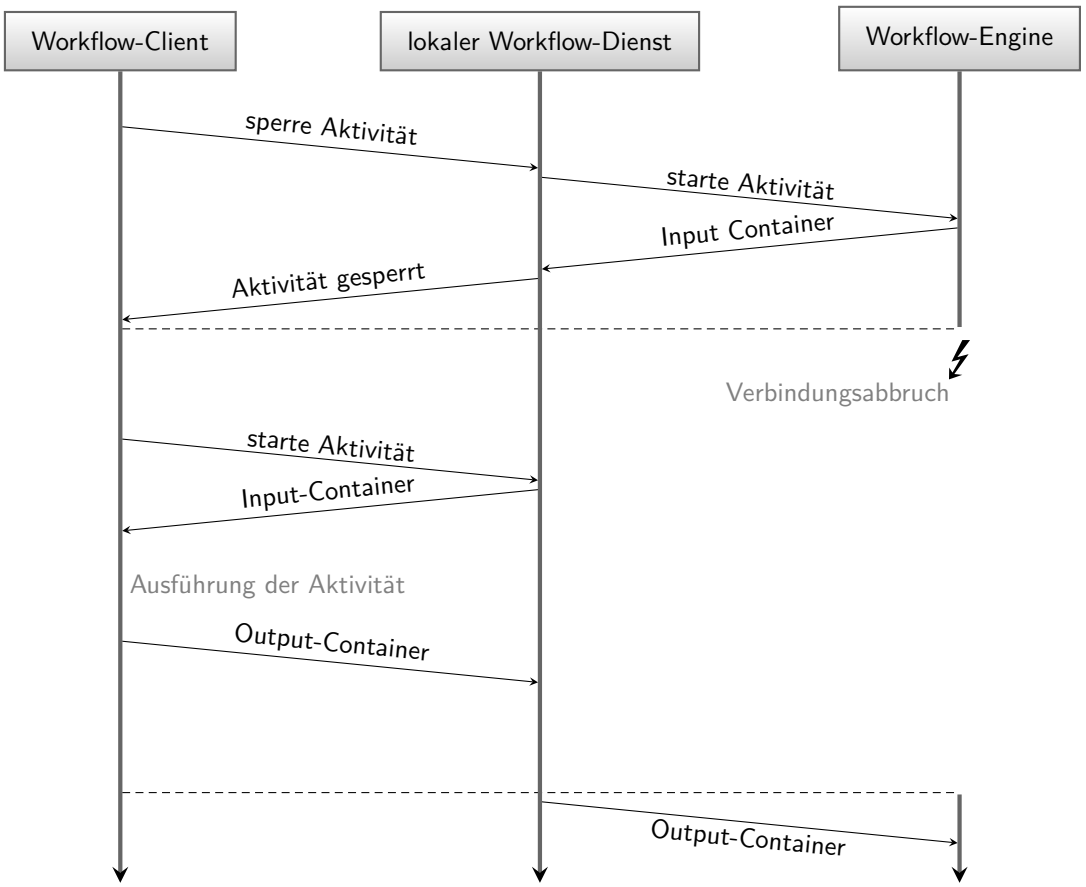


Abbildung 2.18: Kommunikation zwischen Workflow-Client und Workflow-Engine mit Hilfe einer zusätzlichen, auf dem mobilen Gerät installierten Komponente (vgl. [AGK+95])

werden darf [AGK+95]. Die Ausführung einzelner Aktivitäten auf einem mobilen Gerät kann also nicht im ausreichenden Maße der Mobilität und insbesondere der Mobilkommunikation und dessen Konsequenzen gerecht werden. Demzufolge ist es nötig, dass auf dem mobilen Gerät zumindest Teile eines Prozesses autonom ausgeführt werden können.

2.3.2 Teilautonome Ausführung von Prozessabschnitten

Bei der teilautonomen Ausführung von Prozessabschnitten kann ein mobiles Gerät eigenständig den Kontrollfluss eines Prozesses in bestimmten Abschnitten regeln. Die Kontrolle über den gesamten Prozess verbleibt jedoch weiterhin bei einem zentralen Koordinator. Vor dem Trennen der Netzverbindung kann ein mobiles Gerät nicht nur einzelne Aktivitäten zur späteren Ausführung reservieren sondern einen ganzen Prozessabschnitt. Dieser besteht aus mehreren Aktivitäten, die mittels Kontrollflussstrukturen verbunden sind. [AGK+95]

Damit ein Prozessabschnitt von einem mobilen Gerät ohne Netzverbindung ausgeführt werden kann, müssen einige Voraussetzungen erfüllt sein. So müssen alle Aktivitäten in diesem Abschnitt von derselben Person ausgeführt werden dürfen und al-

le für die Ausführung benötigten Programme auf dem mobilen Gerät installiert sein [AGK+95]. Darüber hinaus müssen alle Aktivitäten, die mit einer Kontrollflussstruktur in den Prozessabschnitt hineinführen, bereits erfolgreich beendet sein.

Auf dem mobilen Gerät muss eine Ausführungsumgebung für Prozesse vorhanden sein, damit in dem Prozessabschnitt navigiert werden kann. Diese übernimmt ausschließlich die Kontrolle für den Prozessabschnitt und verwaltet die benötigten Daten-Container. Sobald nach der Ausführung des Prozessabschnittes die Verbindung zum zentralen Server wieder hergestellt ist, werden die resultierenden Daten und der Status aller Aktivitäten an die Workflow-Engine übermittelt. [AGK+95]

Die teilautonome Ausführung von Prozessabschnitten erlaubt es dem mobilen Gerät, größere Abschnitte eines Prozesses ohne Netzverbindung selbstständig auszuführen. Dies ist jedoch nur möglich, wenn in dem Prozess Abschnitte vorhanden sind, die von derselben Person mit demselben Gerät ausgeführt werden können. Hinzu kommt, dass diese Abschnitte an besondere Bedingungen gekoppelt sind und entsprechend markiert sein müssen, damit sie in der Worklist als solche erscheinen können.

2.3.3 Dezentrale Ausführung am Beispiel von mobilen Prozessen

Wird auf einen zentralen Koordinator verzichtet, kann eine völlig verteilte Ausführung von Prozessen erreicht werden. Dazu muss jeder Teilnehmer eine vollständige Ausführungsumgebung für Prozesse besitzen. Bei der Ausführung einer Aktivität wird nun nicht der Input-Container übermittelt sondern der gesamte Prozess mit seinem aktuellen Zustand. [BD99]

Gerade mobile Umgebungen profitieren von dem Verzicht eines zentralen Koordinators, da eine Netzverbindung zu diesem nicht gewährleistet werden kann. Ein mobiles Gerät findet sich statt dessen in einer heterogenen Landschaft mit ständig wechselndem Kontext wieder. Durch eine kontextbasierte Kooperation können auch ohne Koordinator komplexe Prozesse ausgeführt werden. [Kun08]

Es existieren verschiedene Ansätze zur Realisierung einer dezentralen Ausführung von Prozessen, einige davon wurden sogar unter Berücksichtigung der Besonderheiten mobiler Geräte entwickelt [BD99, CJK+08]. Mobile Prozesse nach Kunze et al. eignen sich besonders für die Ausführung auf mobilen Geräten in heterogenen Umgebungen [KZL06] und werden daher im Folgenden näher betrachtet.

Mobile Prozesse

Aufgrund seiner Einschränkungen (vgl. Abschnitt 2.1) ist ein mobiles Gerät meist nicht in der Lage komplexe Prozesse vollständig selbstständig auszuführen. Die Mobilität und ihre Konsequenzen sind zwar offensichtlich ein großes Hindernis, zugleich kann aber durch das Bewusstsein über den daraus resultierenden dynamischen Kontext ein Nutzen entstehen, indem die Umgebung mit in die Ausführung komplexer Aufgaben

einbezogen wird. Komplexe Aufgaben setzen sich dabei aus elementaren Aufgaben zusammen und bieten einen Mehrwert für den Initiator der Aufgabe. Die auftretenden Zwischenergebnisse haben für den Nutzer keine Bedeutung. [Kun08]

Ein *mobiler Prozess* beschreibt komplexe Aufgaben durch eine Abfolge zusammengehöriger Dienste. Die Ausführung dieser Dienste dauert unter Umständen eine längere Zeitspanne und kann sich über mehrere Geräte erstrecken. Dies ist aufgrund der beschränkten Ressourcen mobiler Geräte von besonderer Bedeutung, da ein einzelnes Gerät den mobilen Prozess möglicherweise nicht erfolgreich beenden kann. Tritt ein solcher Fall ein, so wird der mobile Prozess auf ein Gerät in seiner Umgebung migriert. Auf diese Weise wird eine kontextbasierte Kooperation erreicht, die besonders von einer heterogenen Umgebung profitiert, wie sie im Umfeld mobiler Systeme häufig vorgefunden werden kann. [KZL06, Kun08]

Da die Zwischenergebnisse von komplexen Aufgaben für den Nutzer keine Bedeutung haben, sind auch für mobile Prozesse lediglich die Effekte von Bedeutung, die der Initiator von dem Prozess erwartet [KZL06]. Im Fokus stehen somit nicht notwendigerweise Berechnungen mit Daten, sondern die bei der Ausführung von Diensten entstehenden Auswirkungen. Der Zusammenhang zu Prozessen im Kontext von diensteorientierten Architekturen ist offensichtlich.

In mobilen Umgebungen verlangt die sich ständig verändernde Umgebung eine besondere Beachtung. Nicht nur die in der Umgebung befindlichen Dienste wechseln ständig, sondern auch die verfügbaren Technologien, mit denen auf die Dienste zugegriffen werden kann, variieren durch die Migration von mobilen Prozessen. Aus diesem Grund sollte die Auswahl eines konkreten Dienstes möglichst spät erfolgen, nämlich erst kurz vor dem Aufruf des Dienstes. Ein mobiler Prozess verwendet daher *abstrakte Dienstklassen* zur Beschreibung der Aktivitäten. Es handelt sich dabei um eine abstrakte und technologieunabhängige Beschreibung von Diensten, welche technologieabhängige abstrakte Beschreibungen – wie WSDL im Falle von Webservices oder IDL in Corba-Umgebungen – zusammenfasst und verallgemeinert. Erst zur Laufzeit werden aus abstrakten Dienstklassen, in Abhängigkeit der konkret verfügbaren Technologien, die in Frage kommenden technologiabhängigen Dienstbeschreibungen bestimmt. Anschließend wird ein in der Umgebung vorhandener konkreter Dienst anhand von nicht-funktionalen Aspekten ausgewählt und aufgerufen. [KZL07a]

Zur Ausführung einer Aktivität muss jedoch nicht notwendigerweise ein entfernter Dienstaufruf erfolgen. Stellt das mobile Gerät selbst eine entsprechende Funktionalität bereit, so wird die Aktivität lokal ausgeführt. Kann die Aktivität weder lokal noch durch einen entfernten Dienstaufruf ausgeführt werden, so wird der mobile Prozess auf ein anderes Gerät in der Umgebung migriert mit dem Ziel, dass dieses durch den Einsatz von anderen Technologien weitere Dienste erreichen kann und so der Prozess erfolgreich beendet wird. [KZL07b]

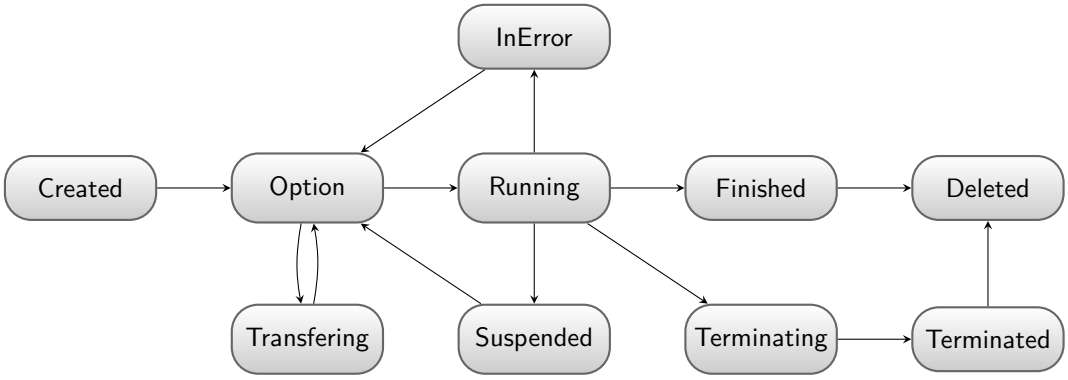


Abbildung 2.19: Lebenszyklus eines mobilen Prozesses (aus [Kun08])

Migrationen werden nur zwischen zwei Aktivitäten durchgeführt, niemals während der Ausführung einer Aktivität. Der Prozess befindet sich dann in einem wohldefinierten Zustand, der neben dem Prozessmodell auf das andere Gerät übertragen wird. Abbildung 2.19 zeigt den potentiellen Lebenszyklus eines mobiles Prozesses. Nach der Modellierung des Prozesses befindet sich dieser im Zustand *Created*. Er kann nun gestartet werden und gelangt so in den Zustand *Option*. Dies ist ein sicherer Zustand, in dem keine Aktivitäten ausgeführt werden und so keine Veränderungen der Daten vorgenommen werden – der Prozess befindet sich in einem konsistenten Ausführungszustand. Der Prozess kann nun entweder lokal ausgeführt (*Running*) oder aber auf ein anderes Gerät migriert werden (*Transfering*). [Kun08]

Tritt ein Fehler während der Ausführung auf, so wird in den Zustand *InError* gewechselt. Ist hingegen eine Aktivierungsbedingung nicht erfüllt, so wird der Prozess in den Wartezustand *Suspended* versetzt. In diesen beiden Zuständen wird der Prozess zunächst nicht weiter ausgeführt. Unter der optimistischen Annahme, dass ein anderes Gerät die Ausführung fortführen kann wird der Prozess anschließend wieder in den Zustand *Option* geschaltet, von dem aus auf ein anderes Gerät migriert werden kann. [LR00, Kun08]

Soll ein mobiler Prozess beendet werden, so wird er in den Zustand *Terminating* versetzt. Es wird nun gewartet, bis die derzeit ausgeführte Aktivität beendet ist und dann in den Zustand *Terminated* übergegangen. Das ordnungsgemäße Ende eines mobilen Prozesses wird durch den Zustand *Finished* erreicht. Wird der Prozess nicht mehr benötigt, so wird er gelöscht (*Deleted*). [LR00]

Um die Migration vollständig unterstützen zu können, muss über den Prozesslebenszyklus hinaus aber auch der Zustand der einzelnen Aktivitäten übertragen werden – nur so kann das andere Gerät an der richtigen Stelle fortsetzen. Auch Aktivitäten besitzen daher einen Lebenszyklus, den Abbildung 2.20 zeigt. Bei der Initialisierung des Prozesses – also wenn dieser den Zustand *Created* verlässt – befinden sich alle Aktivitäten im Zustand *inactive*. Ergibt sich aus dem Kontrollfluss, dass eine Aktivität nicht ausgeführt werden darf, beispielsweise weil bei einer selektiven Ausführung ein anderer Zweig

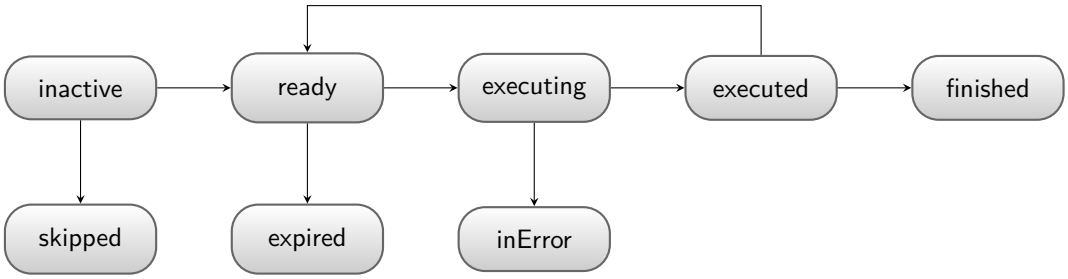


Abbildung 2.20: Lebenszyklus von Aktivitäten eines mobilen Prozesses (aus [KZL07b])

gewählt wurde, geht die Aktivität direkt in den Zustand *skipped* über und wird nicht weiter bearbeitet. Erreicht der Kontrollfluss hingegen die Aktivität, wird ihr Zustand auf *ready* gesetzt, d. h. sie kann nun ausgeführt werden. Läuft ihre Gültigkeit ab, bevor die Aktivität gestartet wird, wechselt sie in den Fehlerzustand *expired*. Wird die Aktivität hingegen rechtzeitig gestartet, befindet sie sich im Zustand *executing*. Der Prozess kann nun nicht mehr auf ein anderes Gerät migriert werden. Erst wenn die Ausführung erfolgreich beendet ist (*executed*), ist eine Migration wieder erlaubt. Falls eine Aktivität im Rahmen einer Schleife wiederholt werden soll, wird ihr Zustand anschließend wieder auf *ready* gesetzt, ansonsten auf *finished*. [Zap05, KZL07b]

2.4 Problem- und Anforderungsanalyse

Die parallele Ausführung von Prozessen auf mobilen Geräten erfordert eine besondere Beachtung, da hierbei Eigenschaften auftreten, die ohne explizite Behandlung eine korrekte Ausführung des Prozesses verhindern können. Während in klassischen verteilten Systemen in der Regel ein zentraler Koordinator den geregelten Ablauf des Prozesses ermöglicht, erschwert die Mobilität der Teilnehmer eine Übertragung dieses Ansatzes. Bevor auf die konkreten Probleme der parallelen Ausführung von Prozessen auf mobilen Geräten eingegangen wird, werden zunächst die Möglichkeiten der Modellierung von Parallelität diskutiert, um damit eine Basis für die anschließende Problemanalyse zu schaffen.

2.4.1 Modellierung von Parallelität

In Abschnitt 2.2 wurde gezeigt, dass sich Parallelität in Prozessen mit verschiedenen Mitteln modellieren lässt. Neben der traditionellen Form mit den Kontrollflussstrukturen *Split* und *Join* (vgl. Abschnitt 2.2.4) führt auch der asynchrone Aufruf eines Subprozesses (vgl. Abschnitt 2.2.5) zur parallelen Ausführung von Prozessen.

Parallelität tritt auch auf, wenn mehrere Ausführungsinstanzen von dem gleichen oder von unterschiedlichen Prozessmodellen gestartet werden. Auch wenn diese nicht in einer Subprozess-Beziehung miteinander stehen, können ebenfalls Konflikte beim Zugriff auf externe Daten oder Ressourcen – beispielsweise Datenbanken – auftreten

[Puu01]. Diese werden jedoch nicht von der Prozessausführungsumgebung verwaltet und sind daher nicht Betrachtungsgegenstand dieser Arbeit.

Im Folgenden werden daher ausschließlich Subprozesse sowie die Kontrollflussstrukturen *Split* und *Join* bei der Betrachtung von Parallelität in Prozessen herangezogen. Wie aus den in Abschnitt 2.2.5 vorgestellten Szenarien deutlich wurde, können Subprozesse in recht unterschiedlichen Ausprägungen auftreten. Im ersten Teil dieses Abschnitts werden daher Eigenschaften herausgearbeitet, mit denen sich Subprozesse charakterisieren lassen. Anschließend wird die Beziehung zwischen Subprozessen und den Kontrollflussstrukturen *Split* und *Join* anhand dieser Eigenschaften erörtert.

Eigenschaften von Subprozessen

Aus den in Abschnitt 2.2.5 vorgestellten Szenarien lassen sich elementare Eigenschaften von Subprozessen ableiten. Bei den folgenden Ausführungen dieser Eigenschaften wird ein besonderer Schwerpunkt auf Parallelität und eine mögliche Implementierung gelegt.

Zeitpunkt des Aufrufes Ein Subprozess kann prinzipiell zu jedem Zeitpunkt von einem anderen Prozess gestartet werden. Besonders bedeutsam ist es jedoch, wenn der Subprozess erst am Ende des Mutterprozesses gestartet wird, da dann eine Verkettung zweier Prozesse stattfindet, wie dies in Abbildung 2.13 dargestellt ist. Da hier keine Parallelität und auch keine Hierarchiebildung auftritt, ist dieser Fall deutlich einfacher zu implementieren, als wenn der Aufruf vor dem Ende erfolgt. So wäre selbst bei einer gemeinsamen Datenhaltung (s. u.) keine Synchronisation notwendig, da die Daten nur einseitig auf den Subprozess übertragen werden müssten. Erfolgt der Aufruf des Subprozesses vor dem Ende des aufrufenden Prozesses, so sind in Abhängigkeit der weiteren Eigenschaften des Subprozesses gegebenenfalls zusätzliche Maßnahmen notwendig.

Art des Aufrufes Der Aufruf eines Subprozesses kann entweder *synchron* oder *asynchron* erfolgen [Hol95]. Im synchronen Fall wird der aufrufende Prozess während der Ausführung des Subprozesses blockiert (vgl. Szenario *Prozesshierarchien*). Die Ausführung erfolgt somit nicht parallel und der Subprozess verhält sich aus der Sicht des Mutterprozesses wie eine Aktivität. Da keine parallele Ausführung auftritt, ist auch keine Synchronisation notwendig. Bei gemeinsamer Datenhaltung müssen die Daten zu Beginn auf den Subprozess und nach Beendigung wieder zurück zum Mutterprozess übertragen werden, analog der Input- und Output-Container bei Aktivitäten. Erfolgt der Aufruf hingegen asynchron, kann der Subprozess parallel zum Mutterprozess ausgeführt werden. Die dabei entstehenden Probleme werden in Abschnitt 2.4.2 analysiert.

Autonomie des Subprozesses Bei völliger Autonomie werden zwei Prozesse vollständig unabhängig voneinander ausgeführt. Beispielsweise würde die Termination eines

Prozesses den anderen nicht beeinflussen [LR00]. Es können jedoch auch Synchronisationspunkte definiert werden (vgl. Szenario *Parallele synchronisierte Prozesse*). Die Prozesse sind dann nicht autonom, sondern sie treffen sich an einem solchen Punkt und können dort Daten austauschen. Zusammenfassend betrifft Autonomie sowohl die fehlerbedingte als auch die ordnungsgemäße Termination sowie die Synchronisation zwischen zwei Prozessen. Wenn die Prozesse auf unterschiedlichen Geräten ausgeführt werden, muss ein geeigneter Mechanismus gefunden werden, um die Abhängigkeiten zwischen den Prozessen umzusetzen.

Datenhaltung Zwei unabhängige Prozesse verwalten in der Regel ihre eigenen Daten. Änderungen der Daten bleiben lokal und es können keine Konflikte auftreten. Da ein Subprozess jedoch meist im Kontext des Mutterprozesses gestartet wird, kann es notwendig sein, dass auf derselben Datenbasis gearbeitet wird [RHEA04]. Nach der Ausführung des Subprozesses müssen die Daten daher gegebenenfalls mit dem Mutterprozess synchronisiert werden. Erfolgt die Ausführung der Prozesse parallel, so kann eine Synchronisation aufgrund von Datenabhängigkeiten schon vorher erforderlich sein.

Ort der Ausführung Ein Subprozess kann entweder *lokal* auf demselben Workflow-Management-System wie der Mutterprozess ausgeführt werden oder *remote* auf einem anderen System [LR00]. Werden die Prozesse auf unterschiedlichen Workflow-Management-Systemen ausgeführt und sind die Prozesse nicht vollständig autonom voneinander, so müssen Nachrichten zwischen den Systemen ausgetauscht werden, damit die Abhängigkeiten zwischen den Prozessen umgesetzt werden können. Offene Schnittstellen sind dabei von großem Vorteil. In jedem Fall muss ein Prozess eindeutig identifizierbar sein, damit das entfernte Workflow-Management-System den gemeinten Prozess finden und starten kann. In einigen Fällen kann es sinnvoll oder sogar nötig sein, dass sich die Systeme gegenseitig authentifizieren.

Subprozesse im Vergleich zu den Kontrollflussstrukturen *Split* und *Join*

Neben den Kontrollflussstrukturen *Split* und *Join* können also auch Subprozesse als Mittel für Parallelität in Prozessen eingesetzt werden. Ein Subprozess wird parallel ausgeführt, wenn der Aufruf asynchron und nicht erst am Ende des aufrufenden Prozesses erfolgt. Im Gegensatz zu *Split* und *Join* erfolgt die Modellierung der parallelen Abschnitte jedoch zunächst unabhängig voneinander. Jeder Subprozess bildet somit eine eigenständige Einheit, die aufgrund ihrer Abgeschlossenheit mehrfach eingesetzt werden kann. Erst durch die Definition von Synchronisationspunkten oder durch die Verwendung von gemeinsam mit einem anderen Prozess genutzten Daten wird die Autonomie zumindest teilweise aufgegeben. Ein mit *Split* und *Join* modellierter paralleler Abschnitt weist immer diese – bei einem Subprozess optionale – Abhängigkeit auf.

In seiner Ausdrucksfähigkeit ist ein Subprozess somit mächtiger als die Konstrukte *Split* und *Join*. Offensichtlich existiert hier sogar eine echte Teilmengenrelation, denn ein Abschnitt eines Prozesses, der mit *Split* und *Join* modelliert wurde, lässt sich in Subprozesse umwandeln. Dazu wird aus jedem parallelen Pfad ein Subprozess erzeugt, der genau aus den Aktivitäten und Kontrollflussstrukturen besteht wie der ursprüngliche parallele Pfad. Dabei teilt der Subprozess seine Daten gemeinsam mit allen anderen so entstandenen Subprozessen und ist vollständig von dem Hauptprozess abhängig. An die Stelle des *Split* treten nun die asynchronen Aufrufe dieser Subprozesse. Der *Join* am Ende der parallelen Ausführung wird durch Synchronisationspunkte zwischen allen Subprozessen und dem Hauptprozess ersetzt.

Es wurde somit gezeigt, dass sich die Kontrollflussstrukturen *Split* und *Join* in Subprozesse umwandeln lassen. Im weiteren Verlauf der Arbeit wird gelegentlich von dieser Eigenschaft Gebrauch gemacht, um nicht mehrere Fälle der Parallelität unterscheiden zu müssen.

2.4.2 Problemanalyse

Wie in Abschnitt 2.3 gezeigt wurde, existieren verschiedene Möglichkeiten, wie mobile Geräte in die Ausführung von Prozessen eingebunden werden können. Da die dezentrale Ausführung der mobilen Nutzung besonders gerecht wird und zugleich, aufgrund des fehlenden zentralen Servers, die größten Schwierigkeiten bereitet, werden im Folgenden nur mobile Prozesse – als Vertreter der dezentralen Ausführung – betrachtet.

In diesem Abschnitt wird analysiert, welche Probleme bei der parallelen Ausführung von mobilen Prozessen auftreten können und welche Fragestellungen zu klären sind. Dazu werden nacheinander drei Phasen betrachtet: Der Beginn der parallelen Ausführung, welche durch einen *Split* oder einen asynchronen Subprozess-Aufruf ausgelöst wird, die parallele Ausführung selbst sowie das Ende der parallelen Ausführung.

Start der parallelen Ausführung

Der Beginn einer parallelen Ausführung ist entweder durch einen *Split* oder einen asynchronen Subprozessaufruf gekennzeichnet. Es müssen nun ein oder mehrere – je nachdem, wie viele Pfade parallel ausgeführt werden sollen – neue Ausführungspfade gestartet werden. Die dabei zu klärenden Fragestellungen werden im Folgenden diskutiert.

Bei der sequentiellen Ausführung eines mobilen Prozesses befindet sich dieser immer – mit Ausnahme des Zeitpunktes der Migration – auf genau einem Gerät [Zap05]. Wurde der Prozess hingegen mit Parallelität modelliert, gibt es mehrere Möglichkeiten für den Ausführungsort des Prozesses. So können zum Einen mehrere Pfade des Kontrollflusses nebenläufig auf demselben Gerät ausgeführt werden. Beinhaltet der Abschnitt manuelle Aktivitäten, so können diese bedingt durch die Benutzerinteraktion allerdings nicht echt parallel ausgeführt werden sondern müssen in eine sequentielle Bearbeitungsreihenfolge überführt werden. Auch automatische Aktivitäten können in der Regel nicht

echt parallel ausgeführt werden, da mobilen Geräten entsprechende Ressourcen fehlen. Statt dessen werden die Aktivitäten lediglich nebenläufig ausgeführt – auch dies ist jedoch nur dann möglich, wenn das Betriebssystem des mobilen Gerätes Nebenläufigkeit unterstützt. Ansonsten ist eine Sequentialisierung der Aktivitäten notwendig.

Anders verhält es sich, wenn die parallelen Pfade nicht auf demselben Gerät ausgeführt werden. Mehrere Geräte können einen Prozess parallel verarbeiten, so dass Aktivitäten tatsächlich gleichzeitig ausgeführt werden. Hierbei muss aber zunächst geklärt werden, welche Geräte für eine parallele Ausführung überhaupt in Frage kommen und wie diese gefunden werden können. Sobald die Auswahl der Geräte erfolgt ist, muss der Prozess auf geeignete Weise an die entsprechenden Geräte übertragen werden. Bei der Migration eines sequentiellen mobilen Prozesses auf ein anderes Gerät übernimmt dieses gleichzeitig die Verantwortung für die Fortführung des gesamten Prozesses, dadurch muss der gesamte Prozess übertragen werden. Geräte, die an der parallelen Ausführung teilnehmen, übernehmen jedoch lediglich die Verantwortung für einen Prozessabschnitt. Daher muss analysiert werden, welche konkreten Informationen zur Ausführung dieses Abschnitts notwendig sind und somit an das Gerät übertragen werden müssen. Dies betrifft insbesondere die Prozessdaten, welche an Aktivitäten übergeben werden und bei mobilen Prozessen in Form von globalen Variablen auftreten. Diese können entweder zentral verwaltet werden oder auf jedem teilnehmenden Gerät repliziert werden – was jedoch eine Synchronisation der Replikate notwendig macht.

Aufgrund des Verwaltungsaufwandes ist von der technischen Seite also die lokale Ausführung auf demselben Gerät möglicherweise vorteilhafter gegenüber der Ausführung auf mehreren Geräten. Ist das Gerät jedoch nicht in der Lage alle Aktivitäten auszuführen, muss ein anderes Gerät in die Ausführung einbezogen werden. Dies kann sogar als Anforderung im Prozess verankert sein, um sicherzustellen, dass die Aktivitäten tatsächlich parallel ausgeführt werden – beispielsweise um Ressourcen zu schonen.

Parallele Ausführung

Während der parallelen Ausführung müssen die für die Aktivitäten benötigten Daten (Input-Container) auf geeignete Weise zur Verfügung gestellt werden. Falls eine zentrale Verwaltung der Prozessdaten gewählt wurde, müssen die benötigten Daten vor dem Beginn einer jeden Aktivität abgerufen werden. Wurden die Prozessdaten hingegen auf jedes Gerät repliziert, so muss gegebenenfalls nur noch sichergestellt werden, dass die Daten den aktuellen Zustand widerspiegeln.

Werden durch die Aktivitäten Änderungen an Daten vorgenommen, die auch von anderen parallelen Pfaden verwendet werden, müssen sie diesen bekannt gemacht werden. Ansonsten kann es passieren, dass semantische Inkonsistenzen auftreten. Dies ist unter anderem der Fall, wenn eine Aktivität ein Datum liest und anschließend zurückschreibt, während der Änderung aber eine parallel ausgeführte Aktivität ebenfalls Schreibvorgänge auf diesem Datum durchführt. Es ist nun u. a. unklar, welche Version

des Datums künftig verwendet werden soll. Gerade in mobilen Umgebungen kann es schwierig sein zu prüfen, ob die verwendeten Daten aktuell sind. Diese Problematik tritt jedoch nur auf, wenn der Datenfluss mit dem Ansatz des *Blackboards* realisiert wird (vgl. Abschnitt 2.2.3). Ist der Datenfluss hingegen explizit modelliert, so geht aus dem Prozessmodell hervor, welche Version des Datums für welche Aktivität verwendet werden soll. Die Abhängigkeiten im Datenfluss implizieren dann Synchronisationspunkte im Kontrollfluss, welche weiter unten analysiert werden. Bei der späteren ausführlichen Diskussion des Problems der parallelen Änderung von Daten wird daher vorausgesetzt, dass ein Blackboard zur Realisierung des Datenflusses verwendet wird. Aufgrund der Komplexität der expliziten Modellierung des Datenflusses ist dies im Übrigen der verbreitetste Ansatz.

Derartige Konflikte können natürlich nicht nur bei den im Prozessmodell verankerten Daten auftreten. Aktivitäten können auch auf andere Daten – beispielsweise die einer externen Datenbank – zugreifen. Solche Daten werden jedoch nicht von einem Workflow-Management-System kontrolliert und daher im Weiteren nicht betrachtet.

Neben Datenabhängigkeiten spielen auch Abhängigkeiten im Kontrollfluss bei der parallelen Ausführung eine entscheidene Rolle. Bei der Modellierung des Prozesses kann beispielsweise eine Präzedenzrelation zwischen den Aktivitäten festgelegt worden sein. Auf diese Weise wird ausgedrückt, dass trotz der parallelen Ausführung zwischen den Aktivitäten verschiedener Pfade Abhängigkeiten existieren. Daher darf eine Aktivität gegebenenfalls erst nach Beendigung einer anderen Aktivität gestartet werden.

Zur Lösung dieser beiden Probleme müssen die parallelen Pfade miteinander oder mit einem Koordinator kommunizieren. Eine besondere Herausforderung dabei ist, dass mobile Prozesse zwischenzeitlich auf ein anderes Gerät migrieren können oder sich ein Gerät aus der Reichweite bewegt. Es kann dann zunächst keine Verbindung zwischen den parallelen Pfaden hergestellt werden, da eine direkte Adressierung nicht möglich ist.

Beendigung der parallelen Ausführung

Die Beendigung der parallelen Ausführung kann mehrere Ursachen haben. So kann es sein, dass in einem parallelen Ausführungspfad ein Fehler auftritt, der den sofortigen Abbruch des gesamten Prozesses auslöst. Aber auch der erfolgreiche Abschluss eines parallelen Pfades führt bei Verwendung des *Cancelling Discriminators* zum Abbruch der übrigen Pfade. Zwar sind die Ursachen grundverschieden, dennoch muss in beiden Fällen eine Nachricht an die anderen Ausführungspfade gesendet werden, die den Abbruch auslöst. Daher wird hier, wie auch während der parallelen Ausführung, eine Abstimmung der parallel ausgeführten Pfade benötigt.

Ist die Ausführung eines asynchronen Subprozesses erfolgreich beendet, so bleibt die Frage, was mit den Daten des Subprozesses passiert. Bei gemeinsamer Datenhaltung muss geprüft werden, welche Version der geänderten Daten weiterverwendet werden

soll. Dies entspricht dem Konsistenzproblem während der parallelen Ausführung und kann analog gelöst werden.

Ebenfalls analog zur parallelen Ausführung ist die Synchronisation des Kontrollflusses, wenn ein *Join* die parallelen Pfade zusammenführt. Bereits abgeschlossene Pfade müssen hier warten, bis alle restlichen Pfade ebenfalls beendet sind. Erst dann darf der Prozess fortgeführt werden. Wird der *Discriminator* beziehungsweise der *Partial Join* zur Zusammenführung der parallelen Pfade verwendet, muss ebenfalls festgestellt werden, wie viele Pfade bereits abgeschlossen wurden. Im Unterschied zum herkömmlichen *Join* kann der Prozess aber bereits fortgeführt werden, bevor alle Pfade beendet sind.

2.4.3 Zusammenfassung und Anforderungsdefinition

Anhand der oben angesprochenen Probleme werden nun konkrete Anforderungen formuliert, die für die parallele Ausführung von mobilen Prozessen notwendig sind, sowie die damit verbundenen Fragestellungen zusammengefasst. In den folgenden Kapiteln werden Lösungsansätze erarbeitet und diskutiert.

1. **Auswahl der Teilnehmer einer parallelen Ausführung** Welche Kriterien muss ein Gerät erfüllen, damit es an der parallelen Ausführung eines mobilen Prozesses teilnehmen kann?

2. **Distribution von Daten** Welche Daten benötigt ein an der parallelen Ausführung teilnehmendes Gerät? Wann sollten diese an das Gerät übertragen werden? Zentrale Datenhaltung oder Replikation der Prozessdaten? Wie bleiben Replikate synchron?

3. **Nebenläufigkeitskontrolle** Wie kann Konsistenz trotz gleichzeitiger Nutzung derselben Daten gewährleistet werden? Welche Anforderungen bezüglich der Datenkonsistenz sind in mobilen Prozessen angemessen?

4. **Synchronisation des Kontrollflusses** Wo findet die Synchronisation statt? Wie finden die parallelen Pfade diesen Ort? Wie werden die geänderten Daten zusammengeführt?

5. **Abbruch anderer paralleler Pfade** Wie kann der Abbruch der restlichen parallelen Pfade im Fehlerfall oder bei Verwendung des *Cancelling Discriminators* ausgelöst werden?

Im Zuge der Analyse und Entwicklung von Lösungsansätzen zu den vorgestellten Problemen sind bedingt durch die Mobilität der Teilnehmer (vgl. Abschnitt 2.1) u. a. folgende Aspekte zu berücksichtigen: Da mobile Geräte in der Regel nur über stark beschränkte Ressourcen verfügen, sollten die Teilnehmer einer parallelen Ausführung nur geringe Anforderungen an Speicherplatz und Rechenleistung erfüllen müssen, damit möglichst viele Teilnehmer für die Ausführung in Betracht kommen. Es ist außerdem

von Vorteil, wenn der Austausch von Nachrichten zwischen den Geräten auf ein Minimum reduziert ist, damit im Falle von Netzausfällen die Ausführung des Prozesses nicht zu lange verzögert wird. Die Größe der zu übertragenden Nachrichten selbst sollte dabei möglichst klein gehalten werden. Um den heterogenen Netztopologien gerecht zu werden, die in der Mobilkommunikation eingesetzt werden, ist ein dezentraler Koordinationsansatz vorzuziehen. Außerdem muss berücksichtigt werden, dass Prozesse auf andere Geräte migrieren können und dabei die Unsicherheit bezüglich der Konnektivität zwischen den parallelen Ausführungspfaden verstärkt wird.

Viele der vorgestellten Probleme lassen sich auf die Synchronisation des Kontrollflusses oder die Behandlung von Datenabhängigkeiten beziehungsweise -konflikten zurückführen. Kapitel 3 beschäftigt sich daher mit Maßnahmen zur Abstimmung nebenläufiger Vorgänge.

Kapitel 3

Synchronisation

Synchronisation beschreibt die Abstimmung von nebenläufigen Vorgängen aufeinander [Eng93]. Dies ist insbesondere immer dann notwendig, wenn mehrere Teilnehmer auf dieselbe Ressource zugreifen, gemeinsame Daten nutzen oder einen Konsens in einer bestimmten Fragestellung erzielen sollen [TS07]. Der Begriff Synchronisation umfasst daher viele Teilaspekte, die bei nebenläufigen Vorgängen von Bedeutung sind. In Kapitel 2 wurde bereits die Synchronisation des Kontrollflusses von Prozessen thematisiert. Sie wird durchgeführt, wenn das Prozessmodell die Kontrollflussstruktur *Join* verwendet oder Subprozesse miteinander verknüpft sind. In diesem Kapitel wird gezeigt, dass zur Sicherstellung der korrekten Ausführung von parallelen Prozessen eine Nebenläufigkeitskontrolle notwendig ist, welche ebenfalls die Synchronisation des Kontrollflusses erfordern kann. Synchronisation umfasst aber auch die in dieser Arbeit relevante Synchronisation von Daten. Dieser Aspekt des Begriffes bezeichnet den Abgleich der Kopien von Daten, auf welche nebenläufig zugegriffen wird. Bei der Verwendung des Begriffes Synchronisation sind die verschiedenen Bedeutungen wohl zu unterscheiden. Wenn sie nicht explizit angegeben wurde, so ist die konkrete Bedeutung aus dem Kontext ersichtlich.

Während ein sequentieller Prozess durch eine totale Ordnung seiner Aktivitäten gekennzeichnet ist, weist ein Prozess mit Parallelität nur eine partielle Ordnung auf. Auch bei der gleichen Eingabe existieren daher verschiedene Ausführungsreihenfolgen für diesen Prozess, die in unterschiedlichen Ergebnissen resultieren können. Abschnitt 3.1 konkretisiert dieses Problem zunächst anhand eines Beispiels. Anschließend wird ein Kriterium für die korrekte Ausführung von parallelen Prozessen erarbeitet und Methoden zur Nebenläufigkeitskontrolle vorgestellt. Abschnitt 3.2 beschäftigt sich mit Replikation – also der Verwendung von Datenkopien – sowie den dabei auftretenden Problemen. Dazu wird wie in Abschnitt 3.1 ein Korrektheitskriterium definiert und es werden Verfahren diskutiert, die eine Ausführung von Prozessen in diesem Sinne ermöglichen. In Abschnitt 3.3 werden Garantien der Replikate, Unterschiede in den verwalteten Datenbeständen nur im begrenzten Maße zuzulassen, betrachtet. Derartige Garantien werden auch Konsistenzmodelle genannt. Dabei wird von den zuvor behandelten Korrektheitsbegriffen Abstand genommen, da nicht jede Anwendung derart strikte Anforderungen aufweist.

3.1 Nebenläufigkeitskontrolle

Ein Prozessmodell lässt sich durch eine Präzedenzrelation ausdrücken. Es handelt sich dabei um eine irreflexive, transitive Relation. Es gilt $A_i < A_j$ genau dann, wenn A_i beendet sein muss, bevor A_j ausgeführt werden darf. Aufgrund der Transitivität ist jedoch die Angabe der direkten Präzedenzen sinnvoller. Es handelt sich dabei um die kleinste Relation, dessen transitiver Abschluss die Präzedenzrelation ergibt. Es gilt $A_i \lessdot A_j$ genau dann, wenn A_j direkt nach A_i ausgeführt wird. [Val05]

Zur Veranschaulichung wird der Prozess in Abbildung 3.1 betrachtet. Dieser besitzt folgende direkte Präzedenzen: $A_1 \lessdot A_2$, $A_1 \lessdot A_3$, $A_2 \lessdot A_4$ und $A_3 \lessdot A_4$. Die Aktivitäten des Prozesses führen einfache Berechnungen mit der globalen Prozessvariable x durch und implementieren die folgende Funktionalität:

$$
\begin{aligned}
< A_1 > &: x := 1 \\
< A_2 > &: x := x + 1 \\
< A_3 > &: x := 3 \cdot x \\
< A_4 > &: skip
\end{aligned}
$$

Bei diesem Prozess gibt es gemäß Kontrollfluss zwei mögliche Ausführungsfolgen: Die Folge $w_1 = A_1A_2A_3A_4$ terminiert mit dem Ergebnis $x = 6$, während die Folge $w_2 = A_1A_3A_2A_4$ das Ergebnis $x = 4$ berechnet. Da die Aktivitäten eines Prozesses im Allgemeinen komplex sind und die Ausführung lange dauern kann, ist es möglich, dass mehrere Aktivitäten gleichzeitig ausgeführt werden. Daher ist es notwendig, das Lesen der Variablen beim Start sowie das Schreiben der Variablen nach Beendigung der Aktivitäten gesondert zu betrachten. Dazu wird jede Aktivität A_i in zwei Operationen l_i (Lesen) und s_i (Schreiben) unterteilt [Val05]. Es gelten nun die Präzedenzen $s_i < l_j$, falls $A_i < A_j$, und es gilt $l_i < s_i$.

Das Prozessmodell aus Abbildung 3.1 ist in Abbildung 3.2 als Verfeinerung mit expliziten Lese- und Schreiboperationen dargestellt. In dieser genaueren Betrachtungsweise ist nun auch die Ausführungsfolge $w_3 = l_1s_1l_2l_3s_3s_2l_4s_4$ möglich, welche das Ergebnis $x = 2$ berechnet. Da hier die Aktivität A_2 die Berechnung von A_3 überschreibt ohne dessen Ergebnis gelesen zu haben, scheint es intuitiv zu einer fehlerhaften Gesamtbe-

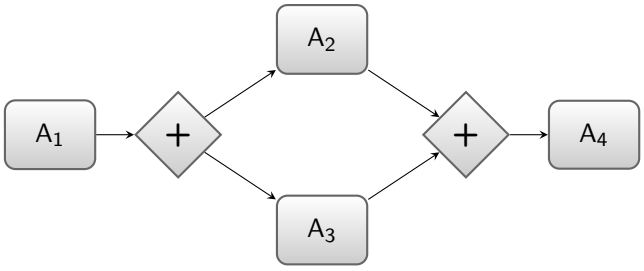


Abbildung 3.1: Einfaches Prozessmodell mit Parallelität

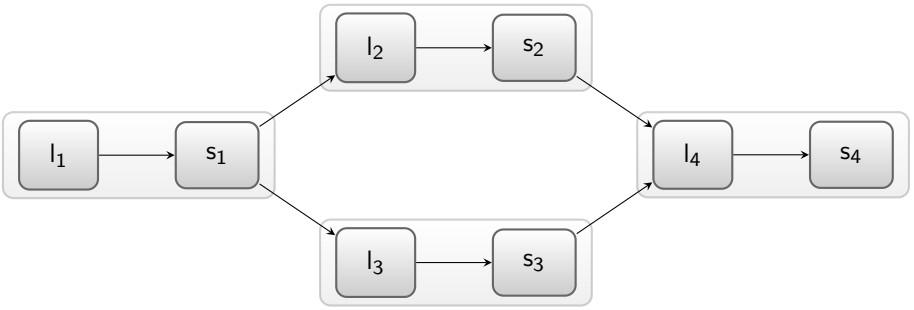


Abbildung 3.2: Verfeinerung des Prozessmodells aus Abbildung 3.1

rechnung gekommen zu sein. Es stellt sich daher die Frage, welche Ausführungsfolgen als korrekt bezeichnet werden dürfen. Dies ist selbstverständlich anwendungsabhängig und nur durch die Intention des Modelliers spezifiziert. Da sich diese jedoch nicht formal verifizieren lässt, wird eine Aktivität als Transaktion[1] betrachtet [Val05]. Es handelt sich dabei um eine sehr spezielle Transaktion, die immer aus genau zwei Operationen besteht. Die Erste ist eine Leseoperationen und die Zweite eine Schreiboperation von potentiell mehreren Daten (entsprechend dem Input- beziehungsweise Output-Container).

3.1.1 Serialisierbarkeit als Korrektheitskriterium

Für Transaktionen existiert das generische Korrektheitskriterium der *Serialisierbarkeit*: Eine Ausführungsfolge ist genau dann *serialisierbar*, wenn sie äquivalent zu einer Ausführungsfolge ist, bei der die Transaktionen seriell, d. h. ohne Überlappung, ausgeführt werden [EGLT76]. Die Äquivalenz liegt dann vor, wenn das gleiche Ergebnis berechnet wird und die gleichen Ausgaben erzeugt werden. Eine Ausführungsfolge wird daher genau dann als *korrekt* bezeichnet, wenn sie serialisierbar ist [EGLT76, HR01]. Dieser Ansatz stellt ein generisches Korrektheitskriterium dar, welches aufgrund seiner Einfachheit auch für mobile Prozesse geeignet zu sein scheint.

Die Ausführungsfolgen w_1 und w_2 des Beispiels von Abbildung 3.1 sind seriell und somit korrekt. In diesem Beispiel sind dies die einzigen seriellen Ausführungsfolgen. Da die Ausführungsfolge w_3 ein anderes Ergebnis als die seriellen Ausführungsfolgen berechnet, ist sie nicht serialisierbar und somit nicht korrekt. Das Phänomen, welches hier auftritt, wird auch *verlorengegangene Änderung* (*Lost Update*) genannt [HR01], da die von A_3 durchgeführte Änderung durch die Schreiboperation der Aktivität A_2 überschrieben wird, ohne die geänderten Daten durch eine Leseoperation wahrgenommen zu haben.

Bei Datenbank-Transaktionen spielen weitere Phänomene eine Rolle. Der *Zugriff auf schmutzige Daten* (*Dirty Read* beziehungsweise *Dirty Write*) bezeichnet das Lesen oder Schreiben von Daten, die durch eine andere, noch nicht abgeschlossene Transaktion er-

1 Man beachte, dass hier der Begriff der Transaktion zunächst sehr allgemein verwendet wird. Im Gegensatz zu Transaktionen auf Ebene des mobilen Prozesses (vgl. [Hol07]) ist an dieser Stelle die Isolation von Transaktionen relevant. Die Aggregation von mehreren Aktivitäten zu einer komplexen Transaktion hingegen wird in dieser Arbeit nicht erneut betrachtet.

zeugt wurden. Schlägt diese Transaktion fehl, so müssen die Änderungen rückgängig gemacht werden und alle bereits erfolgten Lesezugriffe auf die Daten sind ungültig. Falls zwischenzeitlich ein Schreibzugriff auf die Daten erfolgte ist unklar, auf welchen Zustand diese zurückgesetzt werden müssen. Wenn ein Datum innerhalb einer Transaktion mehrmals gelesen, zwischenzeitlich aber von einer anderen Transaktion geändert wird, so spricht man vom *nicht-wiederholbaren Lesen* (*Non-repeatable Read*). Diese Anomalie tritt insbesondere auch dann auf, wenn die andere Transaktion vor dem zweiten Lesevorgang festgeschrieben hat und die Daten somit nicht mehr schmutzig sind. Von einem *Phantom* spricht man dann, wenn sich das Ergebnis der Wiederholung einer mengenorientierten Anfrage aufgrund einer Einfügeoperation einer anderen Transaktion ändert. [HR01]

Bei Prozessen besitzt die durch eine Aktivität beschriebene Transaktion jedoch immer eine feste Struktur, bestehend aus einer Leseoperation zu Beginn und einer Schreiboperation als Abschluss. Das Lesen von mehreren Variablen erfolgt dabei als atomare Operation, da die Variablen zu einem Input-Container zusammengefasst sind. Ebenso erfolgt auch das Schreiben mehrerer Variablen als atomare Operation. In diesem Modell ist der Zugriff auf schmutzige Daten daher gar nicht möglich, da vor dem Abschluss der Transaktion keine Schreiboperationen stattfinden, die schmutzige Daten erzeugen könnten. Die Transaktion wird entweder erfolgreich mit der abschließenden Schreiboperation beendet und festgeschrieben oder aufgrund eines Fehlers vor der Schreiboperation abgebrochen. Ebenso kann das Phänomen des nicht-wiederholbaren Lesens hier nicht auftreten, da jede Transaktion aus nur einer Leseoperation besteht.

Das Beispiel hat jedoch zeigt, dass Mittel eingesetzt werden müssen, die verhindern, dass eine unkorrekte Folge ausgeführt wird und somit Änderungen verloren gehen können – man spricht daher auch von *Nebenläufigkeitskontrolle* [CDK02]. Zwar würde die strikte serielle Ausführung der parallelen Pfade eines Prozesses immer eine korrekte Ausführungsfolge ergeben. Dies wäre jedoch äußerst ineffizient, da so der Vorteil der parallelen Ausführung – nämlich die gleichzeitige Ausführung von Aktivitäten – zunichte gemacht werden würde. Die Nebenläufigkeitskontrolle sollte daher nur so restriktiv sein, wie es unbedingt notwendig ist.

Im Folgenden werden verschiedene Verfahren zur Nebenläufigkeitskontrolle betrachtet, die eine serialisierbare Ausführungsfolge garantieren sollen. Die zumeist aus klassischen Systemen bekannten Verfahren werden dazu in an die Situation von Workflow-Prozessen angepasster Weise diskutiert.

3.1.2 Sperrverfahren

Eine insbesondere bei Datenbanksystemen eingesetzte Methode zur Nebenläufigkeitskontrolle ist die Verwendung von Sperren für Objekte. Bei Prozessen wird dazu eine Variable gesperrt, bevor eine Aktivität auf diese zugreift. Parallel ausgeführte Aktivitäten können die Variable solange nicht verwenden, bis die Sperre aufgehoben ist. Dies

geschieht erst, nachdem die Aktivität ihren Zugriff beendet hat. Serialisierbarkeit ist dann garantiert, wenn die benötigten Sperren in einer Aufbauphase angefordert werden und in einer abschließenden Abbauphase freigegeben werden (*Zwei-Phasen-Sperrprotokoll*). [EGLT76]

Da andere Aktivitäten auf die Freigabe einer Sperre warten müssen, falls die angeforderte Ressource belegt ist, sollten Sperren so restriktiv wie möglich eingesetzt werden. Man unterscheidet daher zwischen *Lesesperren* und *Schreibsperren*. Wenn mehrere Aktivitäten die gleiche Variable lesen, entsteht kein Konflikt. Eine Lesesperre kann daher zur gleichen Zeit mehrfach vergeben sein. Sobald jedoch ein Schreibvorgang aktiv ist, darf keine andere Aktivität auf die betroffene Variable zugreifen. Lesesperren werden daher auch *gemeinsame Sperre* (*shared lock*) und Schreibsperren *exklusive Sperre* (*exclusive lock*) genannt. Tabelle 3.1 fasst die erlaubten Sperren zusammen. [CDK02]

Tabelle 3.1: Kompatibilitätsmatrix des RX-Sperrverfahrens (aus [HR01, S. 419])

Angeforderte Sperre	Bereits gesetzte Sperre		
	Keine Sperre	Lesesperre	Schreibsperre
Lesesperre	✓	✓	–
Schreibsperre	✓	–	–

Im Gegensatz zu Transaktionen in Datenbanken ist bei der Ausführung einer Aktivität als Transaktion bereits vor der Ausführung bekannt, auf welche Variablen mit welchen Operationen zugegriffen wird. Daher gibt es zwei Möglichkeiten, Sperren zu verwenden.

1. Es werden vor der Ausführung der Aktivität für alle Variablen des Input-Containers Lesesperren und für alle Variablen des Output-Containers Schreibsperren angefordert. Ist die Ausführung der Aktivität beendet, so wird die Variable entsprechend dem Output-Container geändert und die Sperren freigegeben.

2. Vor der Ausführung der Aktivität werden lediglich die Variablen des Input-Containers mit einer Lesesperre versehen. Erst wenn die Ausführung beendet ist, werden die Schreibsperren für die Variablen im Output-Container angefordert, die Werte der Variablen geändert und anschließend alle Sperren freigegeben.

Die erste Variante folgt nicht dem Prinzip der minimalen Sperren. Dies führt dazu, dass parallel ausgeführte Aktivitäten manchmal unnötig lange warten müssen. So werden die Aktivitäten des Beispiels aus Abbildung 3.3 aufgrund der Schreibsperre von x serialisiert ausgeführt, während sie bei der zweiten Variante korrekterweise parallel ausgeführt werden können – nur das Zurückschreiben der Output-Container wird serialisiert. Auf der anderen Seite birgt die zweite Variante die Gefahr, dass eine Verklemmung entsteht. Bei einer *Verklemmung* warten mehrere Aktivitäten darauf, dass bestimmte Sperren freigegeben werden, wobei eine zyklische Abhängigkeit zwischen den

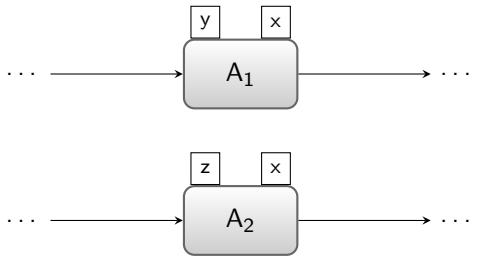


Abbildung 3.3: Zwei parallele Aktivitäten lesen die Variable y beziehungsweise z und schreiben jeweils x

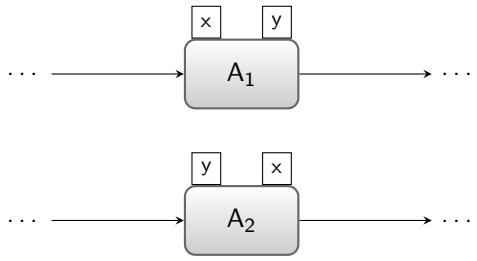


Abbildung 3.4: Zwei parallele Aktivitäten schreiben die jeweils von der anderen Aktivität gelesene Variable

Aktivitäten existiert [HR01]. Im Beispiel von Abbildung 3.4 entsteht eine Verklemmung bei der Ausführungsfolge $l_1 l_2 s_1 s_2$. Die Aktivität A_1 hat mit l_1 eine Lesesperre auf x und die Aktivität A_2 mit l_2 eine Lesesperre auf y erhalten. A_1 benötigt in s_1 eine Schreibsperre auf y und muss warten, da derzeit A_2 eine Sperre auf y besitzt. Umgekehrt muss A_2 darauf warten, bis A_1 die Lesesperre von x freigibt.

Das Verwenden von Sperren setzt voraus, dass zwischen den Teilnehmern Einigkeit über die vergebenen Sperren herrscht. Es wird daher eine Koordination zwischen allen Teilnehmern benötigt, die in der Regel von einem zentralen Koordinator übernommen wird. Dieser hat zwar den Vorteil, dass er gleichzeitig eine Erkennung und Auflösung von Verklemmungen – beispielsweise mittels Warte-Graphen – umsetzen kann [CDK02]. In einer verteilten, mobilen Umgebung muss jedoch zunächst ein für diese Rolle geeignetes Gerät gefunden werden, dass für alle Teilnehmer ständig verfügbar ist. Die an der parallelen Ausführung beteiligten Geräte müssen nämlich vor dem Beginn von jeder Aktivität und nach dessen Ausführung mit dem Koordinator kommunizieren. Kann keine Verbindung zu diesem hergestellt werden, so tritt eine Verzögerung des Prozesses ein, die sich auch auf andere parallele Pfade auswirken kann, falls eine nicht mehr benötigte Sperre aufgrund der unmöglichen Kommunikation nicht freigegeben werden kann.

Um die Anzahl der Kommunikationsvorgänge zu reduzieren, könnten Sperren zwar schon im Voraus auch für die folgenden Aktivitäten angefordert werden. Das für die Ausführung des entsprechenden Pfades zuständige Gerät könnte damit tatsächlich für längere Zeit ohne Netzverbindung zum Koordinator auskommen. Für die anderen parallelen Ausführungspfade steigt jedoch aufgrund der größeren Anzahl und späteren Freigabe der Sperren die Wahrscheinlichkeit deutlich, dass eine von ihnen benötigte Variable bereits gesperrt ist. Welche Strategie sinnvoller ist, hängt somit von der Anzahl der in verschiedenen parallelen Pfaden gemeinsam genutzten Variablen ab.

Das Verkürzen der Sperrdauer von Objekten, wie dies in Datenbanken mit niedrigeren Konsistenzstufen praktiziert wird, ist in mobilen Prozessen nicht sinnvoll, da das hier wichtigste Phänomen bei der parallelen Ausführung, der verloren gegangenen Änderung, weiterhin auftreten kann [HR01]. Weitere Variationen der Sperrverfahren, wie

hierarchische Sperren, Anwartschaftssperren und prädikatsbasierte Sperren, sind in der Regel für Datenbanken entwickelt worden [HR01] und aufgrund der speziellen Struktur von Aktivitäten in mobilen Prozessen nicht anwendbar oder die Verfahren bieten bei diesem Einsatz keine weiteren Vorteile.

Zusammenfassend lässt sich feststellen, dass Sperrverfahren aufgrund der hohen Ausfallwahrscheinlichkeit des Netzes und der beteiligten Geräte in mobilen Umgebungen ungünstig sind, da die Sperren teilweise sehr lang gehalten werden und die Ausführung des Prozesses entsprechend verzögert werden kann.

3.1.3 Optimistische Nebenläufigkeitskontrolle

Bei Sperrverfahren besteht das grundsätzliche Problem, dass durch einen erhöhten Verwaltungsaufwand und das Warten auf die Freigabe von Sperren die Nebenläufigkeit von parallelen Aktivitäten reduziert wird. Außerdem muss die daneben existierende Gefahr, dass Verklemmungen auftreten, mit geeigneten Mitteln behandelt werden. Der Ansatz von Kung und Robinson ist daher, Konflikte erst im Falle des Auftretens zu behandeln statt sie bereits im Voraus zu verhindern. Dabei wird angenommen, dass Konflikte relativ selten auftreten und präventive Sperren folglich einen unnötigen Aufwand darstellen. Es handelt sich somit um ein *optimistisches Verfahren*. [KR81]

Parallele Aktivitäten werden zunächst ohne Einschränkungen ausgeführt. Erst nach der Beendigung wird geprüft, ob Konflikte aufgetreten sind und die Transaktion deshalb rückgängig gemacht werden muss. Es werden dabei drei Phasen unterschieden [KR81]:

Lesephase Die eigentliche Ausführung der Transaktion erfolgt in der Lesephase. Dabei arbeitet die Transaktion jedoch nicht auf den tatsächlichen Daten sondern auf einer Kopie. Somit können zunächst keinerlei Konflikte mit anderen Transaktionen auftreten, da alle Zugriffe auf die Kopie exklusiv sind.

Validierungsphase Nachdem die Transaktion erfolgreich beendet wurde, wird geprüft, ob es zu Konflikten mit anderen Transaktionen gekommen ist, welche die Serialisierbarkeit und somit Korrektheit der Ausführung verhindern könnten. Ist die Validierung nicht erfolgreich, so muss eine Konfliktbehandlung ausgeführt werden, die aus dem Rücksetzen einer der konflikterzeugenden Transaktionen besteht.

Schreibphase Falls eine Transaktion Änderungen auf der lokalen Kopie vorgenommen hat, werden diese nun im permanenten Speicher festgeschrieben. Erst nach Abschluss dieser Phase ist eine Transaktion erfolgreich beendet.

Um die Serialisierbarkeit der Ausführung sicherzustellen, werden am Ende der Lesephase Transaktionsnummern in aufsteigender Reihenfolge vergeben. Somit wird die Serialisierungsreihenfolge festgelegt, zu der die Ausführung äquivalent sein soll. Ähnlich zu den Sperrkonfliktregeln, die im vorherigen Abschnitt vorgestellt wurden, werden nun die Lese- und Schreibzugriffe der Transaktionen verglichen. Dabei muss eine der

folgenden Bedingungen für alle Paare von Transaktionen T_i und T_j zutreffen, wobei die Transaktionsnummer von T_i kleiner ist als die von T_j: [KR81]

1. T_i hat die Schreibphase beendet, bevor T_j die Lesephase beginnt.
2. Keine von T_i geschriebene Variable wird von T_j gelesen und T_i beendet die Schreibphase bevor T_j die Schreibphase beginnt.
3. Keine von T_i geschriebene Variable wird von T_j gelesen oder geschrieben und T_i beendet die Lesephase, bevor T_j die Lesephase beendet.

Diese Bedingungen stellen sicher, dass die Transaktion keine Variablen verwendet hat, die von einer anderen, während der Lesephase beendeten Transaktion geändert wurden. Validierungen müssen serialisiert erfolgen, damit sich die Menge der beendeten Transaktionen während der Validierung nicht ändert [CDK02].

Für langdauernde Transaktionen, die mit vielen Variablen arbeiten, ist die Wahrscheinlichkeit, dass andere Transaktionen diese Variablen ebenfalls verwenden, relativ hoch. Entsprechend kann es passieren, dass die Validierung mehrmals fehlschlägt – vielleicht sogar immer wieder. Dieses als *Verhungern* bezeichnete Problem lässt sich nach dem Erkennen – beispielsweise mit Hilfe eines Zählers – nur mit der Abkehr von der optimistischen Strategie auf ein pessimistisches Verfahren, wie die vorrübergehende Sperrung der verwendeten Daten, lösen [HR01]. Bei Prozessen hingegen ist die Anzahl der parallelen Pfade in der Regel niedrig und die Anzahl der Aktivitäten in den Pfaden zumindest endlich. Ein Verhungern ist somit ausgeschlossen und es kann gegebenenfalls auf die Erkennung und Beseitigung verzichtet werden.

Das oben beschriebene Verfahren prüft, ob Konflikte mit bereits abgeschlossenen Transaktionen aufgetreten sind. Es wird daher auch als *rückwärtsorientiert* bezeichnet, während die ebenfalls mögliche *vorwärtsorientierte Auswertung* die aktuelle Transaktion mit allen noch laufenden Transaktionen vergleicht [HTKR+05]. Da in einer mobilen Umgebung jedoch meist nicht die derzeit ausgeführten Operationen auf anderen mobilen Geräten bekannt sind, ist diese Variante für mobile Prozesse nicht geeignet.

Die Validierungsphase sollte möglichst zeitnah im Anschluss an die Lesephase erfolgen, damit die Anwendung, welche die Transaktion initiiert hat, nicht unnötig lange blockiert ist. Bei der Anwendung in mobilen Prozessen müsste daher nach der Ausführung jeder Aktivität eine Koordination zur Validierung stattfinden. Wird hierbei ein Konflikt festgestellt, so muss die entsprechende Aktivität abgebrochen und wiederholt werden. Falls sich die Ausführung der Aktivität auch außerhalb der Prozessumgebung ausgewirkt hat, ist gegebenenfalls eine Kompensation nötig (vgl. Abschnitt 2.2.6). Im Gegensatz zu Sperrverfahren ist bei der optimistischen Nebenläufigkeitskontrolle nur noch am Ende einer Aktivität eine Koordination der parallelen Pfade notwendig. Zwar ist die Idee der optimistischen Ausführung gerade für mobile Umgebungen geeigneter als Sperrverfahren, dennoch ist die vorgestellte Lösung immer noch nicht befriedigend.

Dieser Ansatz wird daher in Abschnitt 3.2.4 im Kontext der Replikaten adaptiert und erneut betrachtet.

3.1.4 Zeitstempelkontrolle

Bei den Zeitstempelverfahren wird jeder Transaktion T zu Beginn ein eindeutiger Zeitstempel $ts(T)$ zugeordnet. Durch diesen Zeitstempel wird die Serialisierungsreihenfolge der Transaktionen bereits im Voraus festgelegt. Da die Zeitstempel in aufsteigender Reihenfolge vergeben werden müssen, ist es notwendig, dass dies entweder an einem zentralen Ort erfolgt oder alle beteiligten Geräte über synchronisierte physikalische beziehungsweise logische Uhren verfügen. [CDK02]

Bei dem grundlegenden Zeitstempelverfahren besitzt jedes Objekt x einen Lesezeitstempel $RTS(x)$ und einen Schreibzeitstempel $WTS(x)$, die angeben, welche Transaktion das Objekt zuletzt gelesen beziehungsweise geschrieben hat. Da eine Transaktion keine Daten lesen darf, die eine in der Serialisierungsfolge später positionierte Transaktion geschrieben hat, muss der Zeitstempel der lesenden Transaktion mindestens so groß wie der Schreibzeitstempel des zu lesenden Objekts sein: $ts(T) \geq WTS(x)$. Ein Schreibzugriff darf nur erfolgen, wenn keine jüngere Transaktion das Objekt bereits gelesen oder geschrieben hat: $ts(T) \geq \max(RTS(x), WTS(x))$. Sind die Bedingungen erfüllt, so wird die Operation durchgeführt und der Lese- beziehungsweise Schreibzeitstempel des Objektes auf den der Transaktion geändert. Sollte eine der Bedingungen aber nicht erfüllt sein, so ist ein Konflikt aufgetreten, der durch Abbrechen der Transaktion behoben wird. Die Transaktion kann nun erneut gestartet werden und erhält einen neuen Zeitstempel. [BG80]

Auch bei diesem Verfahren ist es möglich, dass Transaktionen verhungern. Wie bei der optimistischen Nebenläufigkeitskontrolle diskutiert wurde, kann dieses Problem bei mobilen Prozessen jedoch vernachlässigt werden, da aufgrund der endlichen Anzahl von nebenäufigen Transaktionen die Ausführung in endlicher Zeit erfolgen wird. Die Abbruchwahrscheinlichkeit kann allerdings verringert werden, indem in bestimmten Situationen das Lesen von Objekten verzögert wird [CDK02]. Denn insbesondere wenn viele ältere Transaktionen noch nicht festgeschrieben haben, besteht die Möglichkeit, dass diese das Objekt noch ändern werden. Hätte die neuere Transaktion das Objekt bereits gelesen, so müssten diese Transaktionen abgebrochen werden. Wird hingegen die Leseoperation der neueren Transaktion blockiert, können die Schreibzugriffe vorher erfolgen und die Transaktionen müssen nicht abgebrochen werden.

Transaktionsabbrüche durch Leseoperationen treten auf, weil eine gemäß der Zeitstempelreihenfolge später auftretende Schreiboperation bereits vor der Leseoperation ausgeführt wurde. Derartige Abbrüche können verhindert werden, indem mehrere Versionen zu jedem Objekt verwaltet werden. Jede Version besteht aus einem Schreibzeitstempel und dem Wert des Objekts. Bei einer Leseoperation wird nun der Wert derjenigen Version zurückgegeben, die den größten Schreibzeitstempel besitzt, der kleiner oder

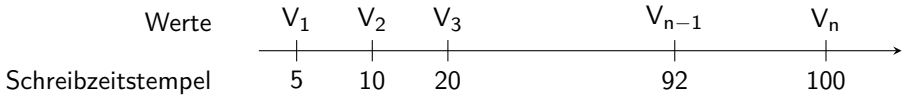


Abbildung 3.5: Mehrere Versionen eines Objektes mit Zeitstempeln (aus [BG80])

gleich dem Zeitstempel der anfragenden Transaktion ist. Wenn im Beispiel von Abbildung 3.5 die Transaktion T mit dem Zeitstempel $ts(T) = 95$ das Objekt lesen möchte, so erhält das Objekt den Wert von der Version V_{n-1}, welche den Schreibzeitstempel $WTS_{n-1} = 92$ besitzt. Leseoperationen können somit immer ausgeführt und die lesende Transaktion muss nicht abgebrochen werden. [BG80]

Schreiboperationen hingegen dürfen nur ausgeführt werden, wenn dies der Serialisierungsreihenfolge nicht widerspricht. Dazu muss protokolliert werden, welche Zeitstempel die lesenden Transaktionen besitzen. Ein Konflikt entsteht, wenn zwischen dem Zeitstempel der Schreiboperation und dem nächsthöheren Schreibzeitstempel des Objekt ein Lesezeitstempel liegt. Die hinter dem Lesezeitstempel stehende Transaktion hat dann eine Version des Objektes gelesen, die älter ist als die gerade geschriebene. Da dies nicht erlaubt ist, muss die schreibende Transaktion abgebrochen werden. Soll im Beispiel von Abbildung 3.6 die Transaktion U mit dem Zeitstempel $ts(U) = 93$ das Objekt schreiben, so wird dies verweigert und die Transaktion abgebrochen, da die Transaktion T (Zeitstempel 95) diesen Wert hätte lesen müssen, zuvor aber den Wert mit dem Zeitstempel $WTS_{n-1} = 92$ gelesen hat. Besteht hingegen kein Konflikt, so wird eine neue Version des Objektes mit dem zu schreibenden Wert und dem Zeitstempel der Transaktion erzeugt. [BG80]

Problematisch an dem Zeitstempelverfahren mit mehreren Versionen ist, dass im Laufe der Zeit sehr viele Versionen zu verwalten sind. Gerade auf leistungsschwachen mobilen Geräte kann der dafür erforderliche Speicherplatz meistens nicht bereitgestellt werden. Aus diesem Grund müssen ältere Versionen gelöscht werden. Im Idealfall werden nur Versionen und Lesezeitstempel gelöscht, die nicht mehr benötigt werden, d. h. alle Transaktionen mit einem kleineren Zeitstempel haben bereits festgeschrieben oder wurden abgebrochen. Um aufwändige Prüfungen zu vermeiden, kann der Versionierung alternativ auch ein fester Speicher zugeordnet werden. Ist dieser voll, so überschreibt eine neue Version die älteste. Wird später doch noch eine Operation mit einem älteren Zeitstempel gestartet, so muss die entsprechende Transaktion abgebrochen werden.

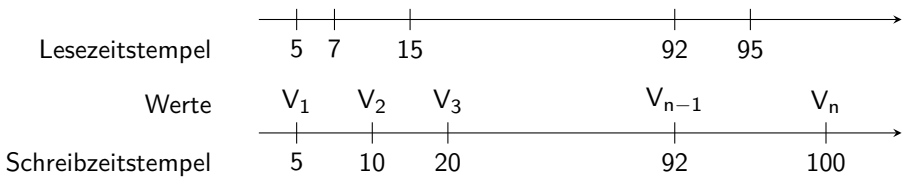


Abbildung 3.6: Schreibzeitstempel der Versionen sowie Lesezeitstempel des Objektes (aus [BG80])

Zeitstempelverfahren führen in mobilen Umgebungen aus verschiedenen Gründen zu Problemen, die nicht immer zufriedenstellend gelöst werden können. Zunächst ist für die Vergabe der Zeitstempel ein ständig erreichbarer zentraler Koordinator oder die Synchronisation der Uhren der beteiligten Geräte notwendig. Zur Durchführung der Nebenläufigkeitskontrolle ist außerdem bei jeder Operation eine Koordination zwischen allen Geräten respektive einem zentralen Koordinator notwendig. Darüber hinaus ist die Abbruchwahrscheinlichkeit bei der klassischen Variante sehr hoch, während bei dem Mehrversionenverfahren sehr viel Speicherplatz benötigt wird. Beides ist für mobile Prozesse unvorteilhaft.

3.1.5 Fazit

In diesem Abschnitt wurde gezeigt, dass Nebenläufigkeitskontrolle notwendig ist, damit die korrekte Ausführung von Prozessen garantiert werden kann. Eine Ausführung wird als korrekt bezeichnet, wenn sie zu einer seriellen Ausführungsfolge äquivalent ist. Alle betrachteten Verfahren zur Nebenläufigkeitskontrolle besitzen Eigenschaften, die in mobilen Umgebungen nicht wünschenswert sind. Die optimistische Nebenläufigkeitskontrolle bietet mit der optimistischen Betrachtungsweise dem ungeachtet einen interessanten Ansatz, der auch bei Netzausfällen greifen kann.

Gerade wegen der hohen Wahrscheinlichkeit von Netzausfällen ist es jedoch sinnvoll, dass die Teilnehmer von mobilen Prozessen über möglichst viele Daten lokal verfügen. Die bei den bisher betrachteten Ansätzen zur Nebenläufigkeitskontrolle implizit vorausgesetzte zentrale Datenhaltung ist daher äußerst ineffektiv. Im nächsten Abschnitt wird daher die Replikation von Daten betrachtet und die in diesem Abschnitt diskutierten Ansätze adaptiert.

3.2 Replikation und Eine-Kopie-Serialisierbarkeit

Wie im Rahmen der Problemanalyse in Abschnitt 2.4 dargelegt wurde, muss eine geeignete Methode zur Distribution der Daten gefunden werden, die für die Ausführung von den Aktivitäten eines Prozesses benötigt werden. Es handelt sich dabei um den Ausschnitt der Prozessdaten, der durch die *Input-Container* der betroffenen Aktivitäten definiert ist. Diese Daten ausschließlich an einem zentralen Ort zu halten, ist in mobilen Umgebungen offensichtlich nicht sehr geeignet. Da ein Zugriff auf den Datenspeicher vor und nach der Ausführung einer jeden Aktivität notwendig ist, würde beispielsweise der Ausfall der Netzverbindung oder des zentralen Datenspeichers eine Fortführung des Prozesses auf unbestimmte Dauer verzögern. Aus diesem Grund kann es von Vorteil sein, Kopien der Daten auf den mobilen Geräten vorzuhalten. Die Verwaltung von derartigen Kopien wird auch als *Replikation* bezeichnet [CDK02]. Von elementarer Bedeutung ist dabei die *Konsistenz* der Replikate, da Änderungen nicht nur auf der lokalen

Kopie durchgeführt werden dürfen, sondern auch auf andere Kopien übertragen werden müssen [TS07].

Replikationsverfahren setzen Redundanz ein, um den Zugriff auf Daten durch eine höhere Verfügbarkeit oder schnellere Anfrageverarbeitung zu verbessern [HTKR+05]. Die mit der Redundanz einhergehenden Probleme sollten jedoch idealerweise vor den Anwendungen verborgen bleiben. Daher wird an Replikate der Anspruch erhoben, die gleiche Funktionalität zu besitzen wie ein zentraler Datenspeicher. Man spricht daher auch von *Replikationstransparenz*, da die Benutzer der Replikate nicht erkennen, dass mehrere Kopien existieren [CDK02]. Wenn ein System diese Forderung erfüllt, befinden sich die Replikate immer in einem konsistenten Zustand und Änderungen der Daten müssen nahezu ohne Zeitverlust an alle anderen Replikate übertragen werden – die Replikate sind *synchron* [TS07]. Diese auch als *strenge Konsistenz* oder *Linearisierbarkeit* bezeichnete Eigenschaft ist zwar für die meisten Anwendungen wünschenswert, aber nur mit erheblichem Aufwand realisierbar und für mobile Umgebungen aufgrund der unzuverlässigen Netzverbindungen äußerst ungeeignet. Bevor in Abschnitt 3.3 weniger strenge Konsistenzmodelle betrachtet werden, muss zunächst der Begriff der Serialisierbarkeit angepasst werden, um ein auch im Kontext von Replikation anwendbares Korrektheitskriterium zu erhalten.

In Abschnitt 3.1 wurde gezeigt, dass es sinnvoll ist, die Ausführung von Aktivitäten als Transaktion zu interpretieren. Das von Transaktionen bekannte Korrektheitskriterium der Serialisierbarkeit kann nun auf Prozesse übertragen werden, um ein Maß für die korrekte Ausführung von parallelen Prozessen zu erhalten. Erweitert man den Begriff der Serialisierbarkeit auf die Verwendung von Replikaten, so muss konsequenter Weise gefordert werden, dass die nebenläufige Ausführung von Transaktionen auf Replikaten äquivalent zu einer seriellen Ausführungsfolge auf einem zentralen Datenspeicher ist. Wird diese Forderung erfüllt, spricht man auch von *Eine-Kopie-Serialisierbarkeit* [BG83]. Diese Eigenschaft garantiert, dass sich der Prozess trotz der Verwendung von Replikation der Modellierung entsprechend verhält.

Im Folgenden werden Verfahren vorgestellt, welche die Eine-Kopie-Serialisierbarkeit garantieren. Einige Verfahren stammen aus dem Bereich der verteilten Datenbanken. Neben dem Lesen und Schreiben spielt dort auch das Einfügen und Löschen von Daten eine Rolle. Auf die Betrachtung dieser zusätzlichen Operationen wird in dieser Arbeit jedoch verzichtet, da aufgrund der starren Struktur der Input- und Output-Container in Prozessen lediglich das Lesen und Schreiben relevant ist. Zunächst werden zwei einfache Verfahren vorgestellt, die mit Netzausfällen gar nicht oder nur sehr schlecht umgehen können, dafür aber sehr anschaulich sind.

3.2.1 Read-One-Write-All

Bei dem einfachsten Verfahren, alle Replikate synchron zu halten, werden Änderungen gleichzeitig auf allen Kopien vorgenommen. Zum Lesen muss lediglich ein Replikat

kontaktiert werden. Aus diesem Grund wird das Verfahren *Read-One-Write-All* (einmal lesen, überall schreiben; *ROWA*) genannt. Eine-Kopie-Serialisierbarkeit wird durch das Setzen von Sperren (vgl. Abschnitt 3.1.2) erreicht. Bei Lese-Operationen setzt das angesprochene Replikat eine Lesesperre und bei Schreiboperationen setzt jedes Replikat eine Schreibsperre auf das entsprechende Datum. Ein Schreibvorgang wird somit verzögert, wenn auf irgendeinem Replikat eine Leseoperation durchgeführt wird, da dieses Replikat keine Schreibsperre zulässt, solange die Lesesperre nicht freigegeben ist. Transaktionen müssen sich auch hier dem Zwei-Phasen-Sperrprotokoll entsprechend verhalten. [Dad96]

Dieses Verfahren ist in dem betrachteten Kontext wenig praxistauglich, da gerade in mobilen Umgebungen nicht die ständige Verfügbarkeit aller Replikate garantiert werden kann, was aber für die zügige Ausführung von Schreiboperationen notwendig ist [MS04]. Dennoch zeigt dieses Verfahren, dass Eine-Kopie-Serialisierbarkeit mit einfachen Mitteln realisierbar ist und Sperrverfahren für Lese-Operationen in Replikaten durchaus sinnvoll eingesetzt werden können.

3.2.2 Primärkopie

Im Gegensatz zu *Read-One-Write-All* werden bei dem im Folgenden betrachteten Verfahren alle Schreiboperationen auf einer Primärkopie ausgeführt. Es handelt sich dabei um eine besonders ausgezeichnete Kopie und somit um ein zentralistisch ausgelegtes Verfahren [MS04]. Datenänderungen werden exklusiv von dieser Primärkopie vorgenommen und anschließend an die anderen Replikate weitergeleitet. Ein Schreibvorgang ist erst abgeschlossen, sobald alle Replikate die Änderung bestätigt haben. Lesezugriffe hingegen können lokal auf einem beliebigen Replikat erfolgen. Die Primärkopie kann die in Abschnitt 3.1 vorgestellten Verfahren zur Nebenläufigkeitskontrolle verwenden, um so die Eine-Kopie-Serialisierbarkeit zu ermöglichen [CDK02].

In dieser Form kann bei Ausfall der Primärkopie ein Replikat schnell einspringen und die Rolle der Primärkopie übernehmen. Bei Netzausfällen sinkt jedoch die Verfügbarkeit, wenn die Primärkopie nicht mehr erreicht werden kann. Sie sollte sich daher nicht dauerhaft auf einem mobilen Gerät befinden. Aber auch wenn ein stationäres Gerät verwendet wird, kann ein mobiles Gerät bei einem Netzausfall keine Änderungen an den Daten vornehmen. [MS04]

Um die Reaktionszeit bei Änderungen zu erhöhen, kann die Primärkopie die Aktualisierung der anderen Replikate asynchron ausführen. In diesem Fall müssen aber zusätzliche Maßnahmen durchgeführt werden, um weiterhin die Eine-Kopie-Serialisierbarkeit garantieren zu können. Dazu kann beispielsweise die optimistische Nebenläufigkeitskontrolle (vgl. Abschnitt 3.1.3) eingesetzt werden. Außerdem muss die Festlegung, welches Replikat die Rolle der Primärkopie übernimmt, nicht global für den gesamten Datenbestand festgelegt sein und kann sich sogar dynamisch verändern. Insbesondere wenn ein mobiles Gerät vorhersehen kann, dass in Kürze die Netzverbindung ausfällt,

kann es für anstehende Schreiboperationen die Rolle der Primärkopie der entsprechenden Prozessdaten übernehmen. Anschließend können Änderungen ohne Netzverbindung vorgenommen werden, während andere Geräte die verschobenen Daten zumindest aus ihren lokalen Kopien lesen und alle übrigen Daten sogar lesen und schreiben können. [TS07]

Aufgrund der Abhängigkeit von der Erreichbarkeit der Primärkopie ist dieses Verfahren insgesamt gesehen in mobilen Umgebungen allerdings eher ungeeignet.

3.2.3 Quorum-Konsens-Verfahren

Bei *Quorum-Konsens-Verfahren* findet eine Abstimmung zwischen den Replikaten statt, bevor eine Operation durchgeführt werden darf. In Abhängigkeit vom Typ der Operation muss ein bestimmtes Quorum (Q_L für eine Leseoperation und Q_S für eine Schreiboperation) erreicht werden, um insgesamt einen Konsens zwischen den n Replikaten über den aktuellen Zustand der replizierten Daten zu erreichen [Gif79]. Eine Operation erhält von einem Replikat keine Stimme, wenn dieses nicht erreichbar ist oder durch eine andere Operation bereits gesperrt ist. Um zu verhindern, dass Konflikte zwischen verschiedenen Operationen auftreten können, müssen die Quoren so gewählt werden, dass folgende Bedingungen erfüllt sind [Gif79]:

$$Q_L + Q_S > n \qquad \text{(verhindert Lese-Schreib-Konflikte)} \tag{3.1}$$

$$Q_{S_1} + Q_{S_2} > n \qquad \text{(verhindert Schreib-Schreib-Konflikte)} \tag{3.2}$$

Aus den Gleichungen folgt, dass die Werte für Q_S und Q_L mindestens so groß wie die folgenden Minimalwerte gesetzt werden müssen: $Q_{S,min} = \frac{n}{2} + 1$ falls n gerade beziehungsweise $Q_{S,min} = \frac{n+1}{2}$ falls n ungerade und $Q_{L,min} = n - Q_S + 1$. Bis zu dem Extremfall $Q_S = n$ und $Q_L = 1$, der dem Read-One-Write-All-Verfahren entspricht, sind alle Kombinationen möglich. Für eine Schreiboperation ist also immer die Zustimmung der Mehrheit der Replikate notwendig. [MS04]

Bei der Zustimmung zu einer Operation überträgt das Replikat auch die Versionsnummer oder den Zeitstempel des bei ihm gespeicherten Datums. Neben der Verhinderung von Lese-Schreib-Konflikten wird somit durch Gleichung 3.1 auch erreicht, dass immer das aktuellste Datum gelesen wird, selbst wenn die Schreiboperation noch nicht auf allen Replikaten durchgeführt wurde. Denn auch im ungünstigsten Fall wird immer von mindestens einem Replikat gelesen, auf dem die letzte Schreiboperation erfolgte. [Gif79]

Die Stimmen der Replikate können außerdem unterschiedlich stark gewichtet werden, um ein Replikat zu bevorzugen [Gif79]. Im Extremfall besitzt nur noch ein einziges Replikat ein Stimmgewicht und es entsteht ein Szenario wie bei der Primärkopie. Durch das Verändern der Gewichtung kann auch die Anzahl der zu übertragenen Nachrichten beeinflusst werden [Dad96].

Der Vorteil des Quorum-Konsens-Verfahren ist, dass im Falle einer Netzunterteilung weiterhin Schreibzugriffe innerhalb einer Netzpartition erfolgen können, auch wenn andere Partitionen nicht erreichbar sind [CDK02]. Dies gilt jedoch nur, wenn in der Partition ausreichend Replikate liegen, damit das Schreibquorum erreicht werden kann. Die Quoren können dem Einsatzszenario entsprechend entweder statisch beim Systemstart festgelegt werden oder sich dynamisch an verändernde Situationen anpassen. Dies ist sinnvoll, wenn sich die Anzahl der Replikate verändert, verursacht jedoch einen erhöhten Koordinationsaufwand [Dad96]. In mobilen Umgebungen kommt es hingegen häufig vor, dass ein einzelnes Gerät mit keinem anderen Replikat mehr kommunizieren kann. In der so entstehenden Netzpartition können dann keine Schreiboperationen ausgeführt werden, da lediglich die Stimme des lokalen Replikates gewonnen werden kann, was für ein Schreibquorum nicht ausreicht. Damit Leseoperationen erfolgen können, bietet es sich an $Q_L = 1$ zu setzen. Da dann aber $Q_S = n$ sein muss, wird effektiv ein Read-One-Write-All-Verfahren mit allen seinen Problemen eingesetzt. Darüber hinaus ist der bei dem Quorum-Konsens-Verfahren für jede Operation notwendige sehr hohe Kommunikationsaufwand nicht zu untertschätzen.

3.2.4 Optimistische Ausführung mit verzögerter Auswertung

Die in Abschnitt 3.1.3 vorgestellte optimistische Nebenläufigkeitskontrolle ist prädestiniert für den Einsatz in Replikaten, da hier jede Transaktion auf einer lokalen Kopie arbeitet. In der diskutierten Variante erfolgt das Einbringen der Änderungen in den zentralen Datenspeicher direkt nach jeder Transaktion. Verwendet man die Kopie jedoch nicht nur für eine einzelne Transaktion sondern auch für weitere auf demselben Gerät ausgeführte Transaktionen, entsteht ein zunächst unsynchronisiertes Replikat. Erst in einer gemeinsamen Validierungsphase werden die ausgeführten Operationen der Transaktionen auf Konflikte mit Transaktionen anderer Replikate überprüft und die Replikate synchronisiert.

Davidson verallgemeinert diesen Ansatz und beschreibt ein Szenario, in welchem bei Netzausfällen ein nahezu beliebiges Verfahren in jeder Netzpartition lokale Eine-Kopie-Serialisierbarkeit erzwingt. Sobald zwei Partitionen wieder miteinander kommunizieren können, wird ein Algorithmus zur Konflikterkennung gestartet, gegebenenfalls konfligierende Transaktionen kompensiert (vgl. Abschnitt 2.2.6) und anschließend die Daten synchronisiert. Dazu ist es notwendig, dass in jeder Partition alle Datenzugriffe aufgezeichnet werden und die Auswirkungen der Transaktionen auch nach dem Festschreiben mit Hilfe von Kompensationen überhaupt rückgängig gemacht werden können. [Dav84]

Die Konflikterkennung erfolgt mit Hilfe eines graphischen Verfahrens, bei dem jede Transaktion einen Knoten darstellt. Es existieren drei Arten von gerichteten Kanten zwischen zwei Knoten T_i und T_j, die eine notwendige Ordnung in einer äquivalenten Serialisierung darstellen: Wenn T_j ein in derselben Partition von T_i geschriebenes Da-

tum liest, findet sich diese Reihenfolge als Datenabhängigkeitskante im Graphen wieder. Eine Präzedenzkante existiert zwischen den Knoten, wenn T_i ein Datum liest, das später von T_j auf derselben Partition geändert wird. Eine Umkehrung dieser Reihenfolge ist nicht erlaubt, da T_i ansonsten den von T_j geschriebenen Wert hätte lesen müssen. Analog muss eine Interferenzkante eingefügt werden, wenn T_i ein Datum liest, das von irgendeiner Transaktion einer anderen Partition geändert wird. Da die Replikate zwischen unterschiedlichen Partitionen nicht synchronisiert und somit Änderungen von anderen Partitionen nicht gelesen werden können, muss in der Serialisierungsfolge T_i vor T_j vorkommen. Aus diesen Kanten lassen sich die möglichen Serialisierungen der Ausführungsfolge ablesen. Tritt ein Zyklus in diesem Graphen auf, so ist keine Serialisierung möglich. In diesem Fall muss daher eine Transaktion abgebrochen werden, um den Zyklus aufzulösen. [Dav84]

Diese konsequente Fortführung der optimistischen Ausführung führt allerdings dazu, dass im Konfliktfall durch das Abbrechen einer Transaktion etliche weitere Transaktionen ebenfalls abgebrochen und kompensiert werden müssen, falls sie Daten gelesen haben, die von der abgebrochenen Transaktion geschrieben wurden [Dav84]. Das Verfahren ist daher eher ungeeignet, wenn die parallelen Pfade eines Prozesses viele Daten gemeinsam nutzen, da dann mit häufigen Abbrüchen und somit einer großen Menge unnötig ausgeführter Arbeit zu rechnen ist. Dennoch kann eine solche optimistische Replikationsstrategie bei Netzausfällen eine deutlich höhere Verfügbarkeit ermöglichen als die bisher betrachteten pessimistischen Verfahren.

3.2.5 Data-Patches

Das Verfahren *Data-Patches* verzichtet zunächst gänzlich auf Maßnahmen, die Eine-Kopie-Serialisierbarkeit ermöglichen. Statt dessen werden alle Operationen ohne Synchronisation auf dem lokalen Replikat ausgeführt. Die Replikate können somit voneinander abweichen und werden inkonsistent. Sobald eine Netzverbindung zwischen den Replikaten besteht werden die verschiedenen Versionen der Daten zusammengeführt. Wurde eine Variable in mehreren Replikaten geändert, so tritt ein Konflikt auf. Mit Hilfe eines Kombinationsverfahrens wird nun der zu übernehmende Wert bestimmt. Hierbei sind verschiedene Verfahren denkbar, von denen bei der Modellierung eines Prozesses je eines für jede Variable festgelegt werden muss: [Dad96]

1. Die *zuletzt geänderte Version* wird verwendet, alle anderen Versionen werden verworfen. Bei dieser Lösung wird das Problem der verlorengegangenen Änderungen in Kauf genommen.

2. Die *Version eines bestimmten Replikates* wird verwendet. Auch hier gehen andere Änderungen verloren.

3. Aufgrund eines *artihmetischen Ausdruckes* wird ein neuer Wert berechnet, der künftig verwendet wird. Werden beispielsweise auf einem Konto lediglich Einzahlun-

gen und Abhebungen vorgenommen, so können zwei Replikate mittels der Formel $k_{neu} := k_1 + k_2 - k_{alt}$ wieder korrekt zusammengeführt werden. k_{alt} bezeichnet hierbei den konsistenten Kontostand vor der Netzpartitionierung und k_1 beziehungsweise k_2 die auf verschiedenen Replikaten berechneten Kontostände.

4. Es wird ein *externes Programm* ausgeführt, welches anhand der vorliegenden Variablenwerte einen neuen Wert berechnet.

5. Der Konflikt wird *manuell aufgelöst*, indem eine hierfür zuständige Person informiert wird.

Dieses Verfahren eignet sich prinzipiell sehr gut in mobilen Umgebungen, da sogar für längere Zeit keine Kommunikation zwischen den Replikaten erforderlich ist. Es wird jedoch vorausgesetzt, dass auf Eine-Kopie-Serialisierbarkeit verzichtet werden kann. Dies ist möglich, wenn lediglich die Prozessdaten von Bedeutung sind, nicht aber die sonstigen Effekte der Ausführung von Aktivitäten. Gerade bei mobilen Prozessen sind jedoch auch die Auswirkungen der Aktivitäten in der Realwelt interessant. Außerdem verwenden Prozesse teilweise komplexe Daten, wie beispielsweise ein Bilddokument. Kombinationsregeln sind daher oft nicht oder nur mit erheblichem Aufwand definierbar.

3.2.6 Semantische Konfliktauflösung in Bayou

In dem Projekt *Bayou* wird wie bei dem Data-Patch-Verfahren eine semantische Konfliktauflösung bei der Synchronisation von Replikaten verwendet. Der Anspruch, Eine-Kopie-Serialisierbarkeit zu ermöglichen, wurde von Anfang an fallengelassen, um statt dessen die höchste Verfügbarkeit der Daten zu gewährleisten. Motivation waren dabei Szenarien, in denen unzuverlässige Netze ein große Rolle spielen, wie beispielsweise das konsistente Arbeiten innerhalb einer vom restlichen System getrennten Teilgruppe oder das Arbeiten auf einem gänzlich isolierten Gerät. Elementar ist dabei die Feststellung, dass schwachen Netzverbindungen nur Rechnung getragen werden kann, indem schwache Konsistenzmodelle eingesetzt werden. [TTP+95]

Verschiedene Replikate müssen sich daher nicht in demselben Zustand befinden, aber trotzdem zu jeder Zeit anwendungsspezifische Integritätsbedingungen erfüllen. In einem Terminplaner wird beispielsweise sichergestellt, dass keine Termine doppelt belegt werden. Bei der Synchronisation zweier Replikate wird ein Verfahren mit dem Namen *Anti-Entropie* angewendet, das eine Einigung der Replikate auf eine gemeinsame Reihenfolge der Transaktionen erzwingt. Transaktionen werden daher zunächst nur probeweise ausgeführt, da sie später rückgängig gemacht und wiederholt werden können. Auf diese Weise können sich immer mehr Replikate über die globale Reihenfolge der Transaktionen einigen. Da aber ständig neue Transaktionen gestartet werden und sich die Netzpartitionierungen aufgrund neuer Netzausfälle laufend ändern können, ist es möglich, dass dennoch nie ein globaler konsistenter Zustand erreicht wird. Bayou ga-

rantiert lediglich, dass, wenn keine neuen Transaktionen gestartet werden, alle Replikate sich *irgendwann* in einem konsistenten Zustand befinden (*eventual consistency*). [PST+97]

Während bei den bisher betrachteten Verfahren sich der Initiator der Transaktion im Falle des erzwungenen Abbruches im Rahmen der Synchronisierung um das Neustarten kümmern muss, erfolgt die im Rahmen der Anti-Entropie-Sitzung durchgeführte Sortierung der Transaktionen ohne Eingriff des Initiators. Dies wäre auch gar nicht immer möglich, da zum Zeitpunkt der Änderung der Reihenfolge möglicherweise gar keine Verbindung zum Initiator besteht. Eine Transaktion muss in Bayou daher eine besondere Struktur aufweisen. Neben der Operation, welche die Daten ändert, besitzt eine Transaktion in Bayou eine Abhängigkeitsprüfung, die sicherstellt, dass sich das System in einem Zustand befindet, wie es der Initiator beim Starten der Transaktion vorgefunden hat. Dazu wird eine Anfrage an die Datenbank formuliert und das erwartete Ergebnis angegeben. Im Falle des Terminplaners kann so beispielsweise geprüft werden, ob im gewünschten Zeitraum immer noch kein anderer Termin eingefügt wurde. Schlägt die Abhängigkeitsprüfung fehl, so wird eine ebenfalls bereitgestellte Kombinationsprozedur aufgerufen, die nicht nur in Abhängigkeit vom Transaktionstyp sondern in Abhängigkeit von den konkreten Datenwerten der Transaktion eine Konfliktlösung erarbeitet. Bei dem Terminplaner kann eine Konfliktlösung im Verschieben des gewünschten Termins gefunden werden. Falls keine Konfliktlösung möglich ist, muss eine Fehlermeldung an den Benutzer gesendet werden. [TTP+95]

Der Nachteil des in Bayou gewählten Weges ist, dass Anwendungsentwickler für jede Transaktion Abhängigkeitsprüfungen und Kombinationsprozeduren bereitstellen müssen. Je nach Komplexität der Anwendung ist dies nicht immer möglich und kann sogar darauf hinauslaufen, dass in der Kombinationsprozedur die gesamte Funktionalität der Anwendung umgesetzt werden muss. Gerade in einer diensteorientierten Architektur kann aufgrund der verborgenen Implementierung eines Dienstes der Modellierer eines Prozesses eine solche Prozedur gar nicht bereitstellen.

Auch für den Anwender ergeben sich Nachteile, die seine Interaktion mit der Anwendung stark beeinflussen. So muss er sich bewusst sein, dass die gelesenen Daten noch Versuchsversionen sind und sich durch die Anti-Entropie-Sitzung noch ändern können. Viel bedeutsamer ist jedoch, dass sich nach Abschluss einer vom Benutzer initiierten Transaktion die eigenen Änderungen ebenfalls in einem Versuchszustand befinden. Welchen Wert die Daten endgültig annehmen, stellt sich erst später heraus. Bei Einfügen eines Termines in den Terminplaner bedeutet dies, dass der Benutzer nicht sofort weiß, wann der Termin tatsächlich stattfindet, da er solange noch verschoben werden kann, bis sich alle Replikate geeinigt haben.

Ist sich der Benutzer einer Bayou-Anwendung dieser Aufhebung gewohnter Transparenzen bewusst, kann das Verfahren die Arbeit in mobilen Netzen deutlich effizienter gestalten. Im Kontext von automatisierten Prozessen innerhalb einer diensteorientierten Architektur gestaltet sich der Einsatz jedoch schwierig oder ist gänzlich unpraktikabel.

3.2.7 Cache-Protokolle

Ein *Cache* ist eine spezielle Form von Replikat. Es handelt sich dabei um einen Zwischenspeicher, der häufig verwendete Daten zum schnelleren Abruf bereitstellt, beispielsweise in einem Prozessor [TS07]. Im Gegensatz zu einem Replikat wird ein Cache in einer Client-Server-Architektur eingesetzt und speichert nicht den gesamten Datenbestand [FCL97]. Auch in einer Prozessausführungsumgebung auf einem mobilen Gerät können Caches zum Einsatz kommen. Die Prozessdaten werden dabei an einem zentralen Ort gespeichert und die mobilen Geräte verfügen über einen lokalen Cache, der die tatsächlich verwendeten Prozessdaten zwischenspeichert. Auch hier ist es notwendig, dass mittels geeigneter Verfahren geänderte Daten zwischen lokalem Cache und zentralem Datenspeicher abgeglichen werden. Im Folgenden werden verschiedene Eigenschaften betrachtet, mit welchen sich derartige Verfahren charakterisieren lassen.

Franklin et al. teilen die Verfahren in Abhängigkeit von ihrer grundsätzlichen Methodik zum Verhindern des Zugriffs auf veraltete Daten in zwei Gruppen ein. Bei der ersten Gruppe werden zunächst Inkonsistenzen des Caches zugelassen. Spätestens beim Festschreiben einer Transaktion müssen die verwendeten Daten jedoch auf Aktualität geprüft und die Transaktion gegebenenfalls abgebrochen werden. Die zweite Gruppe führt Datenänderungen unmittelbar durch und erlaubt daher keine Inkonsistenzen. [FCL97]

Bei Verfahren, die Inkonsistenzen zulassen, wird weiterhin nach dem Zeitpunkt unterschieden, wann die Überprüfung, ob ein Zugriff auf veraltete Daten erfolgt, statt findet. Diese kann entweder unmittelbar bei dem Zugriff auf ein Datum oder bei Abschluss der Transaktion durchgeführt werden. Erfolgt die Überprüfung asynchron, so wird die Transaktion zunächst fortgeführt und, sobald eine negative Antwort empfangen wurde, abgebrochen. Der Vorteil der verzögerten Überprüfung am Ende der Transaktion ist, dass die Konsistenzprüfungen mehrerer Daten gebündelt werden können, um die Netznutzung zu reduzieren. Als Konsequenz ist allerdings die Wahrscheinlichkeit höher, dass eine Transaktion abgebrochen werden muss. [FCL97]

Werden keine Inkonsistenzen der Caches zugelassen, müssen Änderungen unmittelbar an den zentralen Datenspeicher und an die anderen Caches, die das betroffene Datum zwischenspeichern, übertragen werden. Dazu muss, ähnlich wie bei Sperrverfahren, eine Schreiberlaubnis angefordert werden. Dies kann direkt beim Schreibzugriff auf ein Datum oder erst bei Beendigung der Transaktion erfolgen. Die Vor- und Nachteile sind analog zu den oben genannten. [FCL97]

Da sich die konkreten Protokolle zur Realisierung der vorgestellten Eigenschaften nicht wesentlich von denen unterscheiden, die bei Replikaten allgemein verwendet werden [TS07], wird auf eine detailliertere Diskussion verzichtet. Die Verfahren der zweiten Gruppe verwenden einen Ansatz, der den in Abschnitt 3.2.3 vorgestellten Quorum-Konsens-Verfahren, beziehungsweise den Spezialfällen Primärkopie und Read-One-Write-All, entsprechen und daher für mobile Umgebungen wenig geeignet sind. Erfolgt die

Überprüfung der Konsistenz bei Verfahren der ersten Gruppe erst am Ende der Transaktion, so entspricht dies dem optimistischen Verfahren der Nebenläufigkeitskontrolle und führt zu den gleichen Konsequenzen (vgl. Abschnitt 3.2.4). Wird die Überprüfung hingegen unverzüglich vorgenommen, so steigt der Kommunikationsaufwand und verhindert die Ausführung des Prozesses, falls keine Netzverbindung zum zentralen Datenspeicher hergestellt werden kann.

3.2.8 Fazit

Replikation ermöglicht den Zugriff auf Daten ohne die Notwendigkeit eines ständig verfügbaren Datenspeichers. Dies ist gerade in mobilen Umgebungen notwendig, da die Geräte nicht dauerhaft eingeschaltet oder aufgrund unzuverlässiger Netze nicht erreichbar sind. Trotz der durch die Replikation entstehenden Redundanz kann mit Hilfe der vorgestellten Verfahren erreicht werden, dass Prozesse korrekt ausgeführt werden. Probleme treten dann auf, wenn Daten nicht nur gelesen sondern auch geändert werden. In diesem Fall muss die Änderung nicht nur auf die anderen Replikate übertragen werden, sondern es muss sichergestellt werden, dass nicht gleichzeitig auf einem anderen Replikat das gleiche Objekt ebenfalls geändert wird. Pessimistische Verfahren schränken daher den Zugriff auf die Replikate ein, indem Änderungen auf allen Replikaten gleichzeitig vorgenommen werden müssen (Read-One-Write-All), Änderungen ausschließlich auf einem bestimmten Replikat vorgenommen werden dürfen (Primärkopie) oder die Zustimmung einer bestimmten Anzahl von Replikaten gewonnen werden muss, damit eine Operation ausgeführt werden darf. Dies erfordert eine ständige Kommunikation zwischen mehreren Replikaten und führt bei Netzausfällen zu einer eingeschränkten Verfügbarkeit.

Optimistische Verfahren erlauben daher Inkonsistenzen zwischen den Replikaten, die erst später – beim Synchronisieren der Replikate – beseitigt werden. Die optimistische Ausführung mit verzögerter Auswertung führt dazu eine Konflikterkennung mit Hilfe eines graphischen Verfahrens durch. Konfligierende Transaktionen müssen abgebrochen und wiederholt werden, was sogar verschachtelt passieren kann. Semantische Verfahren (Data-Patches und Bayou) versuchen Konflikte mit anwendungsspezifischen Regeln zu beseitigen. Dies erfordert jedoch spezielle Kombinierungsmethoden, die gerade bei automatisierten Prozessen innerhalb einer diensteorientierten Architektur oft nicht oder nur mit erheblichem Aufwand realisierbar sind.

Zusammenfassend lässt sich feststellen, dass in mobilen Umgebungen optimistische Verfahren deutlich besser geeignet sind als die verfügbarkeitsreduzierenden pessimistischen Verfahren. Um die aufgrund der erlaubten Inkonsistenzen nicht zu verhindernden Konflikte dennoch möglichst gering zu halten, können Konsistenzmodelle eingesetzt werden. Diese begrenzen die Unterschiede zwischen Replikaten und werden im folgenden Abschnitt diskutiert.

3.3 Konsistenzmodelle

Wie in den ersten beiden Abschnitten dieses Kapitels deutlich wurde, lassen sich Serialisierbarkeit ermöglichende Verfahren – sei es zur Nebenläufigkeitskontrolle oder zur Synchronisation von Replikaten – in der Regel entweder der Gruppe der pessimistischen oder der optimistischen Verfahren zuordnen. Pessimistische Verfahren erzwingen strenge Konsistenz, sie erlauben also keine Inkonsistenzen zwischen Replikaten. Optimistische Verfahren erlauben hingegen meist beliebige Inkonsistenzen, die erst bei einer nachträglichen Prüfung beseitigt werden. Zwischen den durch diese beiden Gruppen repräsentierten Extrema liegt ein Kontinuum an Konsistenzmodellen, die sich in den erlaubten Abweichungen zwischen den Replikaten unterscheiden [YV02]. Bei einem *Konsistenzmodell* handelt es sich um eine Garantie bezüglich der Konsistenz der replizierten Daten, die ein Replikat einzuhalten verspricht [TS07]. Man beachte, dass mit Konsistenzmodellen die Korrektheit von Replikaten ausgedrückt wird, nicht jedoch die Korrektheit von auf ihnen arbeitenden Prozessen. Es werden daher keine Transaktionen sondern lediglich einzelne Lese- und Schreiboperationen betrachtet.

Die Konsistenz eines Replikates ist gegenläufig zur Verfügbarkeit und damit auch zur Performanz eines Systems, das Replikate einsetzt [YV02]. Strenge Konsistenz erlaubt keine Abweichung der Replikate und erfordert daher eine ständige Datensynchronisation, wodurch bei Netz- oder Knotenausfällen die Verfügbarkeit sinkt und die Performanz des Systems abnimmt. Bei schwacher Konsistenz hingegen ist eine Datensynchronisation seltener notwendig, wodurch das System auch bei Ausfällen verfügbar bleibt. Dafür muss die Anwendung mit veralteten Daten und Schreibkonflikten zurechtkommen. Je nach Situation kann so ein für die Anwendung geeignetes Konsistenzmodell gewählt werden.

Dieser Abschnitt beschäftigt sich mit grundlegenden Konsistenzmodellen. Zunächst wird das bereits erwähnte Kontinuum der Konsistenzmodelle konkretisiert. Anschließend wird das Modell der strengen Konsistenz in zwei Stufen gelockert. Dabei steht weiterhin die Reihenfolge im Vordergrund, in der Operationen auf einem Replikat ausgeführt werden. Bei schwächeren Konsistenzmodellen können aufgrund der erlaubten Unterschiede zwischen den Replikaten eine Reihe von Anomalien auftreten, wenn eine Anwendung mehrere Replikate nutzt. Diese Anomalien widersprechen der natürlichen Erwartung an ein System und können durch gezielte Garantien verhindert werden. Dieses Problem wird am Ende dieses Abschnitts genauer betrachtet.

3.3.1 Stufenlose Konsistenz

Zwischen den beiden Extrema – strenge Konsistenz auf der einen und optimistische Modelle auf der anderen Seite – liegt ein Kontinuum an möglichen Konsistenzmodellen. Dieses wird parametrisiert durch die erlaubte Abweichung eines Replikates zu dem gemeinsamen finalen konsistenten Zustand, der entsteht, wenn alle Schreiboperationen

in einer global eindeutigen Reihenfolge ausgeführt wurden. YU und VAHDAT quantifizieren die Abweichungen zwischen Replikaten durch die drei unabhängigen Parameter numerische Abweichung, Alter und Abweichung der Operationenreihenfolge [YV02]:

Numerische Abweichung Unterschiede zwischen den konkreten Werten der replizierten Objekte werden hier numerisch ausgedrückt. Der Unterschied zwischen zwei numerischen Werten kann als die arithmetische Differenz dieser Werte definiert werden. Nicht-numerische kontinuierliche Werte können möglicherweise in numerische Werte überführt werden, um die arithmetische Differenz anzuwenden. Für diskrete Werte muss jedoch ein alternatives Maß gefunden werden. Hier kann statt dessen die Anzahl der erfolgten Schreiboperationen verwendet werden. Die numerische Abweichung einzelner Objekte kann außerdem gewichtet werden, um die Bedeutsamkeit bestimmter Änderungen auszudrücken oder die genutzten Intervalle des Wertebereiches zu normalisieren. Die erlaubte Abweichung eines Replikates kann entweder in absoluten oder relativen Beträgen definiert werden. Mit Hilfe der numerischen Abweichung kann beispielsweise von einem Replikat gefordert werden, dass die monetären Beträge um maximal 0,02 € (absolut) oder 0,5 % (relativ) abweichen dürfen.

Alter Der Zeitraum von der letzten Änderung, die noch nicht auf dem lokalen Replikat ausgeführt wurde, bis zum aktuellen Zeitpunkt gibt an, wie veraltet der lokale Wert des Objekts ist. Bei bestimmten Anwendungen kann das Verwenden veralteter Daten akzeptiert werden, solange die Daten nicht zu alt sind. Dies kann beispielsweise bei Daten der Wettervorhersage der Fall sein [TS07].

Abweichungen der Operationenreihenfolge Die auf verschiedenen Replikaten ausgeführten Operationen müssen für ein konsistentes Abbild in eine globale Reihenfolge gebracht werden. Die Abweichung von Replikaten kann auch über die Abweichung der Reihenfolge der lokal ausgeführten Operationen zu der globalen Reihenfolge definiert werden. Dazu werden alle ausgeführten Operationen betrachtet, die noch nicht endgültig festgeschrieben sind, (vgl. Abschnitt 3.2.6). Wird die Anzahl dieser Operationen begrenzt, muss vor dem Überschreiten der Grenze eine Synchronisation mit anderen Replikaten erfolgen.

Die von einem Replikat zu garantierende Konsistenz kann über eine beliebige Festlegung der Grenzen für die drei Parameter spezifiziert werden. Zunächst können Operationen auf jedem Replikat ohne Synchronisation ausgeführt werden. Sobald jedoch die Grenze eines Parameters erreicht wird, werden weitere Operationen blockiert, bis eine Synchronisation der Replikate erfolgt. Durch das Festlegen der Grenzen kann so die Konsistenz der Replikate an die vorliegende Situation angepasst werden. Bei niedrigen Parameterwerten werden aktuellere Daten gelesen und es entstehen weniger Konflikte. Dafür muss jedoch bei höheren Parameterwerten seltener auf die Synchronisation gewartet werden, was gerade in unzuverlässigen Netzen wichtig ist. [YV02]

3.3.2 Konsistente Ordnung von Operationen

Um nebenläufige Operationen auf einem Replikat anwenden zu können, müssen die Operationen in eine Reihenfolge gebracht werden, in der sie ausgeführt werden. Wie die auf einem Replikat verwendete Reihenfolge bestimmt wird und ob diese Reihenfolge global eindeutig ist, kann durch das Konsistenzmodell definiert werden. Zu Beginn von Abschnitt 3.2 wurde bereits die strenge Konsistenz vorgestellt. Bei diesem Konsistenzmodell muss eine global eindeutige Ordnung der Operationen verwendet werden, die sich aus der Reihenfolge des tatsächlichen Auftretens der Operationen ergibt. Im Folgenden werden zwei Konsistenzmodelle vorgestellt, deren Anforderungen niedriger und in der Praxis leichter zu implementieren sind.

Sequentielle Konsistenz

Strenge Konsistenz erfordert, dass sich die Wirkungen der Operationen auf allen Replikaten in der Reihenfolge entfalten, wie sie tatsächlich – also in Echtzeit – erfolgten [CDK02]. Aufgrund ungenauer Uhren ist die Feststellung der tatsächlichen Reihenfolge jedoch in vielen Fällen nicht möglich oder mit viel Aufwand verbunden. Lamport fordert daher für das Korrektheitskriterium der *sequentiellen Konsistenz*, dass für jedes Ergebnis einer Ausführung von Operationen auf den Replikaten eine sequentielle Ordnung existiert, deren Ausführung auf dem Datenspeicher das gleiche Ergebnis erzeugt, wobei die Reihenfolge der Operationen innerhalb eines Prozesses in der globalen Sequenz erhalten bleibt [Lam79]. Die Wirkung von Operationen verschiedener Prozesse kann also von der tatsächlichen Reihenfolge des Auftretens der Operationen abweichen, solange alle Replikate in der gewählten Reihenfolge übereinstimmen.

P_1	s[x] = a																		
P_2				s[x] = b															
P_3							l[x] = b										l[x] = a		
P_4												l[x] = a		l[x] = b					

Abbildung 3.7: Zugriffe auf einen nicht sequentiell konsistenten Datenspeicher (aus [TS07, S. 283]) Ohne die Zugriffe von P_4 wäre sequentielle Konsistenz jedoch erfüllt.

Ein Beispiel für Zugriffe auf Replikate, die dieser Forderung nicht entsprechen, zeigt Abbildung 3.7, in der die Operationen von vier Prozessen, die auf unterschiedliche Replikate zugreifen, dargestellt sind. Die Prozesse P_1 und P_2 schreiben nacheinander Werte für x. Betrachtet man die globale Zeit, so findet die Schreiboperation von P_1 vor der von P_2 statt. Die Ergebnisse des Zugriffs von P_3 auf den Datenspeicher entsprechen nicht der strengen Konsistenz, da hier zuerst das Ergebnis der Schreiboperation von P_2 und erst später der von P_1 geschriebene Wert gelesen wird. Bei Vernachlässigung der globalen Zeit sind die Ergebnisse von P_3 jedoch akzeptabel, da sie der erlaubten sequentiellen Ordnung $s_2 l_3 s_1 l_3$ entsprechen. Da keine weiteren Schreiboperationen erfolgen, würden

beliebig verzahnte Leseoperationen von weiteren Prozessen eine Zeit lang b und später nur noch a lesen. Die Ergebnisse der Leseoperationen von P_4 widersprechen jedoch dieser Schlussfolgerung und somit ist der Datenspeicher nicht sequentiell konsistent. [TS07]

Kausale Konsistenz

Bei der *kausalen Konsistenz* muss, im Gegensatz zur sequentiellen Konsistenz, nur für kausal voneinander abhängige Operationen eine globale Reihenfolge gefunden werden. Zwei Operationen sind kausal voneinander abhängig, wenn sie sich gegenseitig beeinflussen. Jede von demselben Prozess ausgeführte Operation ist potentiell kausal abhängig von den zuvor von diesem Prozess ausgeführten Operationen. Außerdem ist eine Leseoperation, die den von einem anderen Prozess geschriebenen Wert liest, von dieser Schreiboperation kausal abhängig. Liest ein Prozess P_2 beispielsweise zunächst das Objekt x, welches zuvor von einem anderen Prozess P_1 geschrieben wurde, und schreibt anschließend y, so ist die Schreiboperation von P_2 kausal von der Schreiboperation von P_1 abhängig, da die Berechnung von y auf dem gelesenen Wert von x basieren könnte. Wenn aber zwei Prozesse ohne zuvor ein Objekt gelesen zu haben ein Objekt verändern, so sind diese Operationen nicht kausal voneinander abhängig, sondern konkurrieren. Konkurrierende Operationen dürfen bei kausaler Konsistenz auf verschiedenen Replikaten in unterschiedlichen Reihenfolgen verarbeitet werden. [HA90]

Es wurde dargelegt, dass das in Abbildung 3.7 gezeigte Beispiel nicht sequentiell konsistent ist, weil P_3 die Schreiboperationen von P_1 und P_2 in einer anderen Reihenfolge wahrnimmt, als P_4. Da die Schreiboperationen von P_1 und P_2 jedoch nicht kausal voneinander abhängig sind, sind diese Zugriffe auf einen kausal konsistenten Datenspeicher erlaubt.

3.3.3 Schwache Konsistenzmodelle

Schwache Konsistenzmodelle besitzen wenige oder gar keine Garantien bezüglich der Konsistenz der Replikate. Aus diesem Grund muss nur selten eine Synchronisation zwischen den Replikaten ausgeführt werden, wodurch in mobilen Umgebungen die Replikate selbst bei Netzausfällen vollständig verfügbar bleiben. Anwendungen, die auf die Replikate zugreifen, müssen sich aber bewusst sein, dass sie möglicherweise inkonsistente Daten lesen. Für die Benutzer der Anwendungen kann es jedoch verwirrend sein, wenn beispielsweise ein älterer Wert eines Datums gelesen wird, nachdem zuvor bereits ein neuerer Wert gelesen wurde [TDP+94]. Dies kann passieren, wenn auf verschiedene Replikate zugegriffen wird, die nicht synchronisiert sind. Gerade in mobilen Umgebungen ist dieses Szenario wahrscheinlich, da sich die Verfügbarkeit der Replikate ständig ändert und somit während einer Sitzung die Operationen nicht nur auf einem einzigen Replikat ausgeführt werden können. Dies trifft auch für mobile Prozesse zu, wenn diese auf andere Geräte migrieren und dort ein anderes Replikat nutzen müssen.

Im Rahmen des in Abschnitt 3.2.6 vorgestellten Projekts *Bayou* wurden die bei schwachen Konsistenzmodellen auftretenden Anomalien untersucht [TDP+94]. Es wurden vier Garantien entwickelt, die das Auftreten bestimmter Anomalien verhindern sollen. Die Garantien sind dabei nur für eine einzelne Sitzung gültig, die durch eine Folge von Lese- und Schreiboperationen beschrieben wird. Bei mobilen Prozessen entspricht eine Sitzung der serialisierten Folge von Aktivitäten, die keine solche Anomalien gestattet.

Lesen eigener Schreiboperationen

Führt ein Benutzer eine Schreiboperation aus, so wird in der Regel erwartet, dass beim anschließenden Lesen die Effekte der Schreiboperation sichtbar sind. Dies ist nicht notwendigerweise der Fall, wenn die Leseoperation auf einem anderen Replikat als die Schreiboperation ausgeführt wird. *Lesen eigener Schreiboperationen* (*Read Your Writes*) garantiert: Wird eine Leseoperation L auf dem Replikat R in derselben Sitzung nach einer (gegebenenfalls auf einem anderen Replikat gestarteten) Schreiboperation S abgesetzt, so muss S auf R beendet sein, bevor L ausgeführt wird. [TDP+94]

Das Lesen eigener Schreiboperationen ist beispielsweise beim Ändern eines Kennwortes wichtig. Bei einem verteilten Authentifizierungssystem nimmt das Übertragen der Änderung an alle Replikate einige Zeit in Anspruch. Der Benutzer bleibt im Unklaren, wann sich seine Änderung ausgewirkt hat und wundert sich über eine fehlgeschlagene Authentifizierung, bei der das neue – in den Augen des Benutzers korrekte – Kennwort verwendet wurde.

Monotones Lesen

Von einer Datenbank wird meistens erwartet, dass die Aktualität ihres Bestandes zunimmt. Es würde daher den Erwartungen widersprechen, wenn beim erneuten Lesen eines Datums ein älterer Wert, als der zuvor gelesene, zurückgegeben wird. *Monotones Lesen* (*Monotonic Reads*) garantiert: Erfolgt die Leseoperation L_1 auf dem Replikat R_1 in einer Sitzung vor der Leseoperation L_2 auf dem Replikat R_2, so müssen alle relevanten Schreiboperation, die den von L_1 gelesenen Wert ergeben haben, auch auf R_2 ausgeführt worden sein, bevor L_2 ausgeführt wird. [TDP+94]

Betrachtet man beispielsweise eine replizierte E-Mail-Datenbank, so kann ein Benutzer von einem Replikat eine Liste aller E-Mails auslesen. Während der Anzeige dieser Liste muss aufgrund der Mobilität des Anwenders das Replikat gewechselt werden. Wählt der Benutzer nun eine E-Mail zur Anzeige aus, so garantiert das monotone Lesen, dass die E-Mail tatsächlich auch auf dem neuen Replikat bereits empfangen wurde und der Benutzer nicht statt dessen eine Fehlermeldung erhält. [TDP+94]

Schreiben nach dem Lesen

In Schreiboperationen geschriebene Daten werden meistens aus zuvor gelesenen Daten berechnet. Daher ist es manchmal wünschenswert, dass auf allen Replikaten, die diese Schreiboperation ausführen, zuvor die Daten verfügbar sind, die zu der Schreiboperation geführt haben. *Schreiben nach dem Lesen* (*Write Follow Reads*) garantiert: Alle relevanten Schreiboperationen, die den von einer Leseoperation L gelesenen Wert ergeben haben, müssen vor der Schreiboperation S ausgeführt werden, wenn S in einer Sitzung nach L ausgeführt wird. [TDP+94]

Wird beispielsweise in einer Datenbank eine Änderung eines Eintrages vorgenommen, die auf dem aktuellen Zustand des Eintrages basiert, ist es wichtig, dass auf anderen Replikaten die Änderung auf demselben Zustand ausgeführt wird. Wird in einem E-Mail-Verteiler beispielsweise eine Antwort auf eine E-Mail versendet, so muss in einem Replikat zunächst die ursprüngliche Nachricht empfangen werden, bevor die Antwort angezeigt wird. [TDP+94]

Monotones Schreiben

Werden in einer Sitzung mehrere Schreiboperationen ausgeführt, so erwartet man, dass diese auf allen Replikaten in der von dem Prozess vorgegebenen Reihenfolge ausgeführt werden. Ist dies nicht der Fall, so würden spätere Änderungen möglicherweise von älteren Schreibvorgängen überschrieben werden. *Monotones Schreiben* (*Monotonic Writes*) garantiert: Wenn eine Schreiboperation S_1 in einer Sitzung vor einer Schreiboperation S_2 ausgeführt wird, so muss auf jedem Replikat S_1 vor S_2 ausgeführt werden. [TDP+94]

Arbeitet ein Benutzer beispielsweise mit einer Textverarbeitung und sichert dort regelmäßig das bearbeitete Dokument, so garantiert das monotone Schreiben, dass auf jedem Replikat keine neuere Sicherung durch eine ältere überschrieben wird. In diesem Fall könnte das monotone Schreiben aber auch durch *Schreiben nach dem Lesen* realisiert werden, wenn die Textverarbeitung vor dem Erzeugen einer neuen Version das Dokument explizit liest. [TDP+94]

3.4 Ergebnis der Untersuchung

Zu Beginn dieses Kapitels wurde gezeigt, dass die parallele Ausführung von Aktivitäten zu unterschiedlichen Ergebnissen führen kann. Es ist daher notwendig, gewünschte und ungewünschte Ausführungen eines Prozesses zu unterscheiden. Serialisierbarkeit ist ein generisches Korrektheitskriterium, welches eine anwendungsunabhängige Definition der Korrektheit der Ausführung eines Prozesses erlaubt und im Bereich der Datenbanken bereits genau untersucht wurde. Die Überprüfung, ob eine Ausführungsfolge serialisierbar ist, erfordert keine zusätzliche Kenntnis über die Intention des Modellierers eines Prozesses. Dennoch sind serialisierbare Ausführungsfolgen fast immer auch

im Sinne des Modellierers korrekt, da hier eine sequentielle Ausführung existiert, die zu dem gleichen Ergebnis führt.

Parallele Aktivitäten werden im Allgemeinen auf unterschiedlichen Geräten ausgeführt. Dabei wird auf verschiedene Daten – wie beispielsweise auf das Prozessmodell und auf die Prozessvariablen – zugegriffen. Da in mobilen Umgebungen die ständige Verfügbarkeit eines zentralen Datenspeichers nicht garantiert werden kann, wurde in Abschnitt 3.2 die Replikation von Daten im Allgemeinen diskutiert. Bei der Replikation treten in Verbindung mit parallelen Zugriffen jedoch zusätzliche Probleme auf, wie die Sicherstellung der Konsistenz von Replikaten. An die Stelle der Serialisierbarkeit tritt nun die auf Replikation angepasste Eine-Kopie-Serialisierbarkeit. Da die Replikation der für die Ausführung eines parallelen Pfades benötigten Daten die Ausführung unabhängiger von Netzverbindungen zu anderen Geräten macht, handelt es sich dennoch um einen vielversprechenden Weg, den Besonderheiten mobiler Umgebungen gerecht zu werden.

Um sicherzustellen, dass die Ausführung eines parallelen Prozesses korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist, wurden verschiedene Verfahren entwickelt, die den Zugriff auf die Replikate regeln. Die meisten der vorgestellten Ansätze senken jedoch die Verfügbarkeit der Replikate deutlich, wenn die Kommunikation zu anderen Replikaten eingeschränkt ist. Optimistische Verfahren erlauben hingegen das Fortsetzen der Ausführung. Sie prüfen mögliche Konflikte, sobald die Netzverbindungen wiederhergestellt sind. Wird nun ein Fehler festgestellt, müssen jedoch Aktivitäten abgebrochen und wiederholt werden. Dies kann sogar verschachtelt passieren und zu einer großen Menge unnötig durchgeführter Arbeit führen.

Zusammenfassend ist festzustellen, dass mit den in diesem Kapitel diskutierten Konzepten die korrekte Ausführung von parallelen Prozessen auf mobilen Geräten zwar prinzipiell möglich ist. Da in mobilen Umgebungen die Wahrscheinlichkeit des Ausfalls der Geräte und der Kommunikation zwischen den Geräten besonders hoch ist, führt die Anwendung dieser Methoden jedoch entweder zu langen Verzögerungen während der Ausführung oder zur Wiederholung von möglicherweise sehr aufwändigen Aktivitäten. In Kapitel 4 wird daher ein Ansatz vorgestellt, bei dem durch verschiedene Maßnahmen Konflikte zwischen parallelen Ausführungspfaden reduziert werden. Weiterhin wird die als elementar erachtete Replikation von Prozessdaten zusammen mit der optimistischen Ausführung eingesetzt, um verbleibende Konflikte unter Beibehaltung einer hohen Verfügbarkeit zu behandeln.

Kapitel 4

Parallelität in mobilen Prozessen

Wie aus der Problemanalyse in Abschnitt 2.4 deutlich wurde, müssen bei der parallelen Ausführung von mobilen Prozessen mehrere Teilprobleme gelöst werden. In diesem Kapitel wird ein Konzept zur parallelen Ausführung von mobilen Prozessen erarbeitet, wobei eine Orientierung an den in der Problemanalyse aufgezeigten Teilproblemen erfolgt.

In Kapitel 3 wurde gezeigt, dass die Nebenläufigkeitskontrolle bei der parallelen Ausführung eines Prozesses elementar ist, um Konflikte bei dem Zugriff auf gemeinsame Daten zu behandeln. Da jedoch nicht bei jedem Prozess Konflikte auftreten, kann in manchen Fällen auf die Nebenläufigkeitskontrolle verzichtet werden, falls dies vorher bekannt ist. Aber auch die Art der Nebenläufigkeitskontrolle kann in Abhängigkeit von den tatsächlich vorhandenen Konflikten gewählt werden. Abschnitt 4.2 beschäftigt sich daher mit der Erkennung von problematischen Konflikten aus dem Prozessmodell.

Eine zentrale Idee des in diesem Kapitel vorgestellten Konzeptes ist die Definition von Datenklassen. Zwar wurde in Kapitel 3 Serialisierbarkeit als sinnvolles Korrektheitskriterium vorgestellt, dies wird aber nicht von jedem Prozess für seine korrekte Ausführung benötigt. Werden beispielsweise Daten der Wettervorhersage verarbeitet, so ist nicht immer Serialisierbarkeit notwendig, statt dessen kann in einigen Fällen auch mit veralteten Daten produktiv gearbeitet werden. In Abschnitt 4.3 wird dieses Problem genauer betrachtet und es werden exemplarisch drei mögliche Datenklassen diskutiert.

In Abschnitt 4.4 wird die Distribution des Kontrollflusses betrachtet. Daher werden Kriterien erarbeitet, die ein Gerät erfüllen muss, damit es an der parallelen Ausführung teilnehmen kann. Außerdem wird analysiert, welche Daten von den Teilnehmern benötigt werden und wie diese auf das entsprechende Gerät übertragen werden.

Die Problemanalyse hat zudem gezeigt, dass eine Koordination der parallelen Pfade aus verschiedenen Gründen notwendig ist: Synchronisation des Kontrollflusses, Nebenläufigkeitskontrolle, Abbruch von parallelen Ausführungspfaden (Fehler oder Cancelling Discriminator). Abschnitt 4.5 beschäftigt sich daher mit der Koordination und Kommunikation zwischen parallelen Pfaden. Da die optimistische Ausführung gerade in Netzen mit häufigen Verbindungsabbrüchen vorteilhaft sein kann, wird in Abschnitt 4.6 ein Konzept zur optimistischen Konfliktauflösung vorgestellt.

Das Kapitel wird in Abschnitt 4.7 mit der Diskussion der Korrektheit des vorgestellten Konzepts abgeschlossen.

4.1 Grundkonzept

Bevor die einzelnen Teilprobleme im Detail behandelt werden, soll in diesem Abschnitt zunächst der Zusammenhang zwischen den Komponenten dargestellt werden. Das in diesem Kapitel vorgestellte Konzept betrachtet nicht nur die Ausführung von Prozessen sondern setzt bereits in der Modellierungsphase an (vgl. Abbildung 4.1). Ziel ist es dabei, die Notwendigkeit der Kommunikation zwischen parallelen Ausführungspfaden zu reduzieren, damit auch im Falle von Netzausfällen keine unnötigen Verzögerungen bei der Ausführung entstehen. Greifen in einem Prozess jedoch parallele Aktivitäten in bestimmter Weise auf dieselben Variablen zu, so ist eine Koordination zwischen den parallelen Pfaden zwingend notwendig, damit die Ausführung korrekt im Sinne der Serialisierbarkeit erfolgen kann. Derartige Situationen werden in dieser Arbeit als *Abhängigkeitskonflikt* bezeichnet und können bereits bei der Modellierung eines Prozesses erkannt und somit gegebenenfalls vermieden werden. Eine Erkennung der Abhängigkeitskonflikte zur Modellierungszeit räumt dem Modellierer also die Möglichkeit ein seinen Prozess zu modifizieren, um die Anzahl der notwendigen Synchronisierungen während der parallelen Ausführung zu reduzieren.

Im nächsten Schritt kann der Modellierer die nach der Modifikation immer noch existierenden Abhängigkeitskonflikte analysieren und die dort verwendeten Variablen auf die Notwendigkeit einer serialisierbaren Ausführung untersuchen. Serialisierbarkeit ist zwar ein intuitives und insbesondere generisches Korrektheitskriterium, in vielen Fällen schränkt es die Menge der als korrekt zu interpretierenden Ausführungsfolgen jedoch zu stark ein. Wird ein flexibleres Korrektheitskriterium verwendet, kann die Anzahl der notwendigen Synchronisierungen während der parallelen Ausführung weiter reduziert werden. Wie stark von der Serialisierbarkeit abgewichen werden darf ist dabei anwendungsspezifisch. Durch die Verwendung von Datenklassen können die benötigten Garantien bezüglich der Konsistenz der von einem Prozess verwendeten Daten formuliert werden. Der Modellierer ergänzt dazu das Prozessmodell um eine Zuordnung von Datenklassen.

Zur Laufzeit lassen sich die in Abbildung 4.2 dargestellten Komponenten unterscheiden. Treten im Kontrollfluss ein *AND-Split* oder ein asynchroner Subprozessaufruf auf, müssen Teilnehmer für die nun parallel auszuführenden Pfade gefunden und der Prozess entsprechend verteilt werden (*Distribution*). Während der parallelen Ausführung ist in vielen Fällen eine *Koordination* notwendig. Dies betrifft zum Einen die Nebenläu-

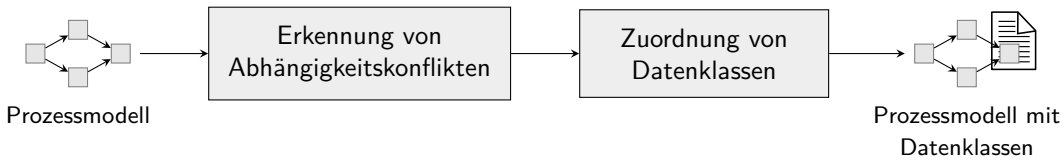


Abbildung 4.1: Komponenten zur Modellierungszeit

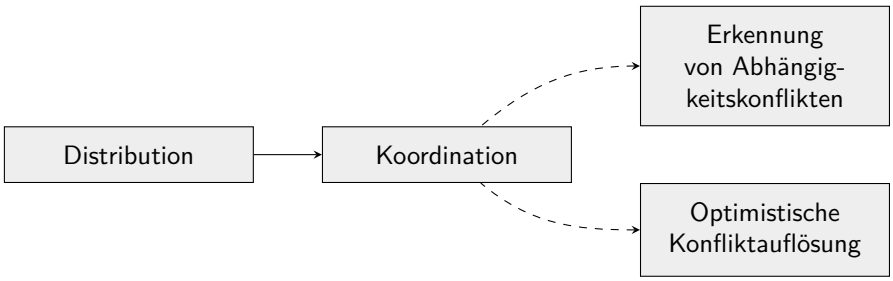


Abbildung 4.2: Komponenten zur Laufzeit

figkeitskontrolle, die in Abhängigkeit von den auftretenden Abhängigkeitskonflikten durch die *optimistische Konfliktauflösung* realisiert wird. Zum Anderen betrifft dies die Synchronisation des Kontrollflusses, die wegen eines *AND-Joins* am Ende der parallelen Ausführung oder aufgrund einer expliziten Synchronisation auch während der parallelen Ausführung durchgeführt werden muss. Bei gemeinsam genutzten Daten müssen dann Änderungen an diesen ebenfalls zusammengeführt werden.

4.1.1 Replikation von Prozessdaten

Da die in dieser Arbeit betrachteten Prozesse in mobilen Umgebungen ausgeführt werden und somit nicht die ständige Verfügbarkeit der Kommunikation zwischen den parallelen Pfaden oder einem Koordinator gegeben ist, wurde in Kapitel 3 vorgeschlagen, die zur Ausführung der parallelen Pfade benötigten Daten zu replizieren. Das in diesem Kapitel vorgestellte Konzept sieht daher vor, dass jeder parallele Ausführungspfad über ein vollständiges Replikat der von ihm zur Ausführung benötigten Daten verfügt. Dies betrifft insbesondere gemeinsam genutzte Variablen. Zu Beginn der parallelen Ausführung muss daher der bisherige Zustand des Prozesses kopiert und den künftigen Teilnehmern als Replikate zur Verfügung gestellt werden. Während der parallelen Ausführung werden Änderungen an den Daten zunächst nur auf dem lokalen Replikat vorgenommen. Erst bei der Synchronisation des Kontrollflusses werden auch die Replikate wieder zusammengeführt. Während der Ausführung eines parallelen Prozesses existieren daher mehrere Replikate, die über unterschiedliche Daten verfügen können. Im Folgenden soll daher $\mathcal{W}_{\mathfrak{R}}(x)$ den Wert der Variable x auf dem Replikat $\mathfrak{R}$ bezeichnen.

Im Prozess von Abbildung 4.3 werden beispielsweise nach der Ausführung von der Aktivität A_1 zwei Replikate aus dem in $\mathfrak{R}_0$ manifestierten aktuellen Prozesszustand erzeugt. Auf dem Replikat $\mathfrak{R}_1$ werden die Aktivitäten A_2 und A_3 beziehungsweise A_2 und A_4 ausgeführt, während auf dem Replikat $\mathfrak{R}_2$ die Aktivität A_5 ausgeführt wird. Vor der Ausführung von der Aktivität A_6 werden die beiden Replikate synchronisiert um die parallele Ausführung zu beenden.

In bestimmten Situationen müssen bereits während der parallelen Ausführung Variablenwerte von anderen Replikaten gelesen werden, damit die Ausführung korrekt ist.

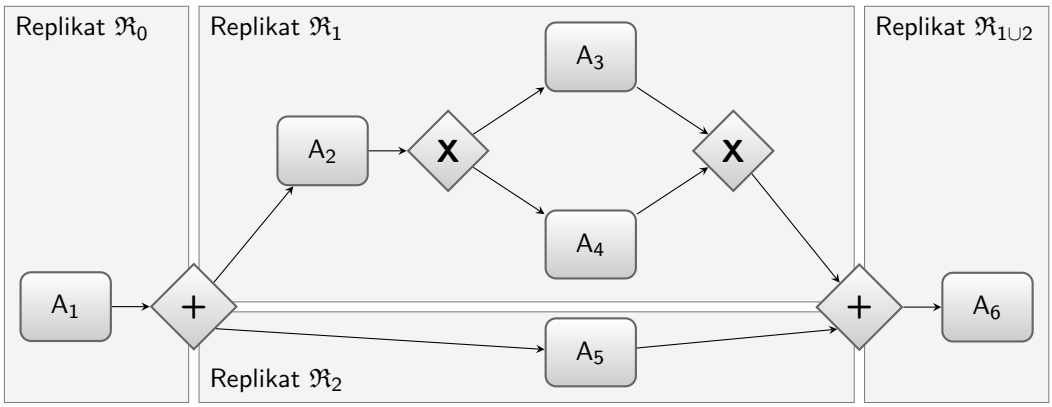


Abbildung 4.3: Jeder parallele Ausführungspfad verfügt über ein eigenes Replikat

Aufgrund der Replikation tritt als Korrektheitskriterium nun die Eine-Kopie-Serialisierbarkeit an die Stelle der Serialisierbarkeit. Das Auftreten solcher Abhängigkeitskonflikte kann bereits zur Modellierungszeit erkannt werden. Dazu ist die Analyse des Prozessmodells notwendig. Da es diverse Techniken zur Modellierung eines Prozesses gibt, wird im Folgenden ein abtraktes Prozess-Metamodell eingeführt.

4.1.2 Abstraktes Prozess-Metamodell

Parallelität liegt in Prozessen immer dann vor, wenn gleichzeitig mehrere Ausführungspfade bearbeitet werden. In Abschnitt 2.4.1 wurden verschiedene Mittel diskutiert, mit denen Parallelität modelliert werden kann. Es wurde gezeigt, dass dabei asynchron aufgerufene Subprozesse das allgemeinste Werkzeug darstellen und die Kontrollflussstrukturen *Split* und *Join* simulieren können. Mit diesen Mitteln können sehr komplexe Prozesse modelliert werden, die viele Verzweigungen aufweisen. Das in diesem Kapitel vorgestellte Konzept soll jedoch nicht abhängig von den für die Modellierung gewählten Mitteln sein. Im Folgenden werden daher einige Relationen definiert, welche die Beziehungen zwischen Aktivitäten beschreiben und die sich aus dem konkreten Prozessmodell ableiten lassen. Dazu wird auch die bereits in Kapitel 3 beschriebene Präzedenzrelation explizit definiert.

Definition 4.1 (Präzedenzen) *Seien A_1 und A_2 Aktivitäten eines mobilen Prozesses. Es gilt $A_1 \lessdot A_2$ genau dann, wenn A_1 durch genau eine Kontrollflussstruktur (vgl. Abschnitt 2.2.4) mit A_2 verbunden ist, so dass A_1 vor A_2 ausgeführt werden muss (direkte Präzedenz). In $\lessdot$ existieren keine Zyklen[1]. Die Präzedenzrelation $<$ sei der transitive Abschluss: $< := \lessdot^+$*

Definition 4.2 (Alternativität) *Seien A_1 und A_2 Aktivitäten eines mobilen Prozesses. Es gilt $A_1 \otimes A_2$ genau dann, wenn eine selektive Verzweigung (XOR-Split) existiert, so dass entweder A_1 oder A_2 ausgeführt wird.*

1 Iterative Ausführung kann als Blockkonstrukt realisiert werden, das mehrere Aktivitäten zusammenfasst (vgl. Abschnitt 2.2.4).

Definition 4.3 (Parallelität) *Seien A_1 und A_2 Aktivitäten eines mobilen Prozesses. Es gilt $A_1 \parallel A_2$ genau dann, wenn nicht $A_1 = A_2 \vee A_1 < A_2 \vee A_2 < A_1 \vee A_1 \otimes A_2$ gilt. Die Aktivitäten A_1 und A_2 heißen dann parallel zueinander.*

Zwei Aktivitäten werden also auch dann als parallel betrachtet, wenn diese wie beim OR-Split in manchen Fällen alternativ ausgeführt werden. Dies ist notwendig, da die Auswertung der Bedingungen eines OR-Splits erst erfolgen kann, wenn der Kontrollfluss während der Ausführung des Prozesses an dem OR-Split angelangt ist. Des Weiteren werden Steuerungselemente, die nur der Verzweigung und Zusammenführung des Kontrollflusses dienen, in diesem Metamodell ebenfalls als Aktivität interpretiert.

Man beachte, dass die Relationen $\otimes$ und $\parallel$ jeweils symmetrisch sind. Außerdem gilt für beliebige voneinander verschiedene Aktivitäten A_1 und A_2, dass immer genau eine der folgenden Aussagen wahr ist: $A_1 < A_2$, $A_2 < A_1$, $A_1 \otimes A_2$ oder $A_1 \parallel A_2$. Im Beispiel von Abbildung 4.3 gelten folgende Relationen:

$A_1 < A_2$				
$A_1 < A_3$	$A_2 < A_3$			
$A_1 < A_4$	$A_2 < A_4$	$A_3 \otimes A_4$		
$A_1 < A_5$	$A_2 \parallel A_5$	$A_3 \parallel A_5$	$A_4 \parallel A_5$	
$A_1 < A_6$	$A_2 < A_6$	$A_3 < A_6$	$A_4 < A_6$	$A_5 < A_6$

Von parallelen Aktivitäten durchgeführte Änderungen wirken sich also zunächst nur auf dem lokalen Replikat aus. Damit die Ausführung des Prozesses auf Replikaten korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist, muss sie äquivalent zu einer seriellen Ausführungsfolge auf einer einzigen Kopie sein. Im nächsten Abschnitt wird erarbeitet, wie diese äquivalente Ausführungsfolge gefunden werden kann und wann zusätzliche Maßnahmen notwendig sind, damit sie überhaupt existiert.

4.2 Erkennung von Abhängigkeitskonflikten

Im Gegensatz zu Datenbanksystemen, bei denen Transaktionen ohne absehbare Reihenfolge ankommen und die dabei verwendeten Daten nicht vorhersehbar sind, erfolgt der Datenzugriff in mobilen Prozessen vordefiniert gemäß dem ihm zu Grunde liegenden Prozessmodell. Dies ermöglicht die präventive Analyse eines Prozesses zur Erkennung von potentiellen Konflikten bei dem Zugriff auf Daten sowie zur Einleitung von Maßnahmen, die das tatsächliche Auftreten der Abhängigkeitskonflikte verhindern und somit Eine-Kopie-Serialisierbarkeit gewährleisten. Konflikte können zwischen parallelen Aktivitäten auftreten, wenn sie die gleichen Variablen verwenden. Das Ziel der Analyse ist es, serielle Ausführungsfolgen zu finden, in denen potentiell konfligierende Aktivitäten in einer Reihenfolge ausgeführt werden, in der möglichst wenig zusätzliche Maßnahmen zur Nebenläufigkeitskontrolle notwendig sind. Dazu werden Methoden

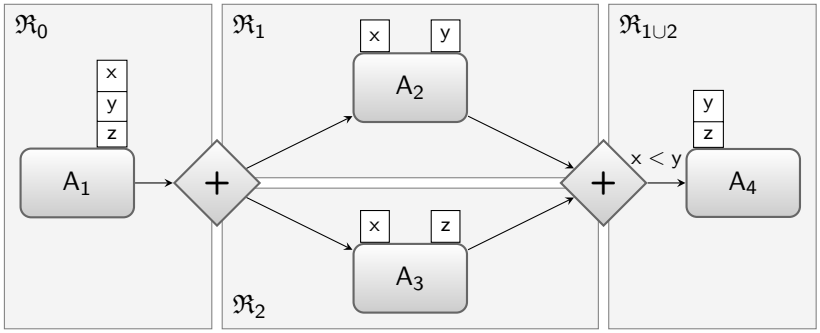


Abbildung 4.4: Die parallelen Aktivitäten A_2 und A_3 lesen und schreiben die Variablen jeweils nur auf dem entsprechenden Replikat. In diesem Beispiel werden keine von parallelen Aktivitäten verwendeten Variablen geändert.

eingesetzt, die den bei der optimistischen Ausführung eingesetzten Verfahren (vgl. Abschnitt 3.2.4) sehr ähneln. Im Unterschied zu den optimistischen Verfahren wird die Analyse hier jedoch nicht zur nachträglichen Prüfung der nebenläufigen Ausführung von Transaktionen sondern vor der Ausführung der Transaktionen beziehungsweise Aktivitäten durchgeführt.

In Abschnitt 3.1 wurde gezeigt, dass die Äquivalenz einer seriellen Ausführungsfolge zur tatsächlichen Ausführung von der gewählten Reihenfolge der Lese- und Schreiboperationen abhängt. Die Analyse des Prozessmodells betrachtet daher, welche Variablen von den Aktivitäten während der parallelen Ausführung gelesen und geschrieben werden. Analog zu [Dav84] werden diese Mengen zur leichteren Identifizierung explizit definiert:

Definition 4.4 (Lese- und Schreibmengen von Aktivitäten) *Es sei $\mathcal{L}(A_i)$ definiert als die Menge der von der Aktivität A_i gelesenen Variablen und $\mathcal{S}(A_i)$ die Menge der von der Aktivität A_i geschriebenen Variablen. $\mathcal{V}(A_i)$ ist die Summe der von der Aktivität A_i verwendeten Variablen: $\mathcal{V}(A_i) := \mathcal{L}(A_i) \cup \mathcal{S}(A_i)$*

Den Kontrollfluss steuernde Bedingungen (vgl. Abschnitt 2.2.3) werden hierbei einer Aktivität zugeordnet; dadurch sind neben dem Input-Container der Aktivität auch die in den Bedingungen verwendeten Variablen in der Menge $\mathcal{L}(A_i)$ enthalten. Im Prozess aus Abbildung 4.4 beinhaltet daher $\mathcal{L}(A_4)$ auch die Variable x, da diese in der zur Aktivität A_4 gehörenden Aktivierungsbedingung verwendet wird. Insgesamt ergeben sich somit die Lese- und Schreibmengen wie folgt:

$$\begin{array}{lll}
\mathcal{L}(A_1) = \emptyset & \mathcal{S}(A_1) = \{x,y,z\} & \mathcal{V}(A_1) = \{x,y,z\} \\
\mathcal{L}(A_2) = \{x\} & \mathcal{S}(A_2) = \{y\} & \mathcal{V}(A_2) = \{x,y\} \\
\mathcal{L}(A_3) = \{x\} & \mathcal{S}(A_3) = \{z\} & \mathcal{V}(A_3) = \{x,z\} \\
\mathcal{L}(A_4) = \{x,y,z\} & \mathcal{S}(A_4) = \emptyset & \mathcal{V}(A_4) = \{x,y,z\}
\end{array}$$

Werden wie in Abbildung 4.4 in allen Aktivitäten nur Variablen verändert, die von keiner anderen parallelen Aktivität verwendet werden, ist jede mögliche Ausführungsfolge serialisierbar. Es ist somit keine Nebenläufigkeitskontrolle notwendig und die Ausführung des Beispiels ist äquivalent zu den prinzipiell möglichen seriellen Ausführungsfolgen $w_1 = A_1A_2A_3A_4$ und $w_2 = A_1A_3A_2A_4$. Bei der Synchronisation des Kontrollflusses müssen nun auch die Replikate synchronisiert werden, da diese nicht notwendigerweise über identische Werte verfügen. Das synchronisierte Replikat verwendet dazu für alle in einem Ausführungspfad veränderten Variablen den Wert des entsprechenden Replikates. Nicht veränderte Variablen können aus einem beliebigen Replikat übernommen werden, da sich diese in den Replikaten nicht unterscheiden. Der konkrete Ablauf einer Synchronisation wird in Abschnitt 4.5 beschrieben.

4.2.1 Lese-Schreib-Abhängigkeiten

Anders verhält es sich, wenn von einer Aktivität Variablen verändert werden, die auch von weiteren parallelen Aktivitäten verwendet werden, d. h. wenn $\mathcal{S}(A_i) \cap \mathcal{V}(A_j) \neq \emptyset$ für $A_i \parallel A_j$. Dabei muss unterschieden werden, ob eine Variable von der anderen Aktivität gelesen oder geschrieben wird. Ändern mehrere Aktivitäten eine Variable ohne dass ein erneutes Lesen dieser Variable auftritt, so ist ebenfalls keine Nebenläufigkeitskontrolle notwendig. Bei der Synchronisation der Replikate wird der Wert übernommen, der gemäß einer äquivalenten Serialisierung zuletzt berechnet wurde. Bevor dies anhand eines Beispiels in Abschnitt 4.2.3 verdeutlicht wird, werden Lese-Schreib-Abhängigkeiten betrachtet, da diese die Menge der in Frage kommenden Serialisierungen einschränken. Wichtig ist dabei die Unterscheidung zwischen tatsächlicher Ausführung der Aktivitäten auf lokalen Replikaten und der Menge der Serialisierungen, die äquivalent zu dieser Ausführung sind und nur ein Hilfsmittel zur Überprüfung der Korrektheit darstellen. Die Reihenfolge paralleler Aktivitäten bei der tatsächlichen Ausführung kann sich zu der Reihenfolge in einer Serialisierung unterscheiden, obwohl beide äquivalent sind. In der folgenden Definition werden Lese-Schreib-Abhängigkeiten durch die Relation $\prec$ formalisiert und anschließend dessen Bedeutung diskutiert.

Definition 4.5 (Lese-Schreib-Abhängigkeit) *Seien A_L und A_S zwei Aktivitäten eines Prozesses. Es gilt $A_L \prec A_S$ genau dann, wenn $A_L \parallel A_S$ und $\mathcal{L}(A_L) \cap \mathcal{S}(A_S) \neq \emptyset$.*

Eine *Lese-Schreib-Abhängigkeit* tritt also auf, wenn von einer Aktivität eine Variable gelesen wird, die von einer parallelen Aktivität geschrieben wird. Da die Änderung der Variable lediglich auf dem lokalen Replikat erfolgt, ist eine Ausführung im Allgemeinen nur dann serialisierbar, wenn in der serialisierten Ausführungsfolge die Leseoperation vor der Schreiboperation erfolgt. Wird diese Reihenfolge nicht eingehalten, müsste die Leseoperation den von der Schreiboperation geänderten Wert lesen. Dafür wäre jedoch eine Kommunikation zwischen den Geräten, auf denen die Aktivitäten ausgeführt werden, notwendig. Da es dies zu vermeiden gilt, wird durch die Lese-Schreib-Abhängig-

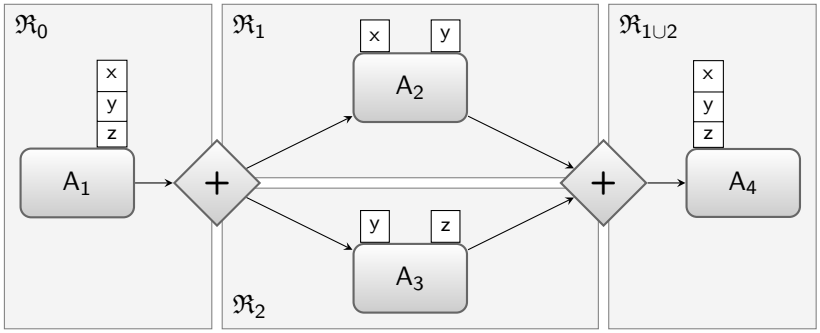


Abbildung 4.5: Prozess mit einer Lese-Schreib-Abhängigkeit: A_3 liest die Variable y, welche von der parallelen Aktivität A_2 auf einem anderen Replikat geändert wird

keit eine Ordnung in der Serialisierung vorgegeben, so dass die Serialisierung auch zur unsynchronisierten Ausführung äquivalent ist.

Um dies zu verdeutlichen, wird der Prozess in Abbildung 4.5 betrachtet. Die Aktivität A_1 initialisiert alle Variablen mit dem Wert 1. Während der parallelen Ausführung existieren die Replikate $\mathfrak{R}_1$, auf dem A_2 ausgeführt wird, und $\mathfrak{R}_2$, auf dem A_3 ausgeführt wird. Die Aktivität A_2 implementiert die Funktionialität $y := x + 2$ und A_3 berechnet $z := y \cdot 2$. Da die Änderungen der Variablen durch die Aktivitäten lediglich lokal erfolgen, führt die unsynchronisierte Ausführung der Aktivitäten A_2 und A_3 zu zwei sich unterscheidenden Replikaten mit den folgenden Werten: $\mathcal{W}_{\mathfrak{R}_1}(x, y, z) = (1, 3, 1)$ und $\mathcal{W}_{\mathfrak{R}_2}(x, y, z) = (1, 1, 2)$. Das Zusammenführen der in den Replikaten veränderten Variablen ergibt $\mathcal{W}(x, y, z) = (1, 3, 2)$. Damit diese Ausführung korrekt ist, muss eine äquivalente serielle Ausführungsfolge existieren. Für diesen Prozess gibt es zwei serielle Ausführungsfolgen: Die Folge $w_1 = A_1A_2A_3A_4$ ergibt $\mathcal{W}(x, y, z) = (1, 3, 6)$, während die Folge $w_2 = A_1A_3A_2A_4$ die Werte $\mathcal{W}(x, y, z) = (1, 3, 2)$ berechnet. Wie erwartet ist die unsynchronisierte Ausführung der parallelen Pfade auf lokalen Replikaten lediglich äquivalent zu der Serialisierung, bei der die Lese-Schreib-Abhängigkeit der Variable y berücksichtigt ist.

Lese-Schreib-Abhängigkeiten beeinflussen also die Menge der äquivalenten Serialisierungen, indem die Reihenfolge von den betroffenen Aktivitäten innerhalb einer Serialisierung vorgegeben wird. Dies beeinflusst aber nicht zwangsläufig die tatsächliche Ausführung der Aktivitäten, da Änderungen nur auf den lokalen Replikaten vorgenommen werden. Lese-Schreib-Abhängigkeiten führen somit nicht notwendigerweise zu einem Problem – dieser Fall wurde in dem Prozess von Abbildung 4.5 beobachtet. In der Serialisierung muss zwar A_3 vor A_2 vorkommen, tatsächlich darf aber A_2 auch vor A_3 ausgeführt werden, da A_3 die Änderungen von A_2 nicht sieht.

Neben der Präzedenzrelation $<$ (beziehungsweise $\lessdot$ für die direkten Präzedenzen) gibt also ebenso die Relation $\prec$ aufgrund von Lese-Schreib-Abhängigkeiten Beziehungen zwischen Aktivitäten vor, die sich in der Reihenfolge der Aktivitäten in einer Serialisierung widerspiegeln müssen. Dies ist jedoch nicht immer möglich: Lese-Schreib-Abhängigkeiten können in Verbindung mit Präzedenzen zu Widersprüchen führen. Ist

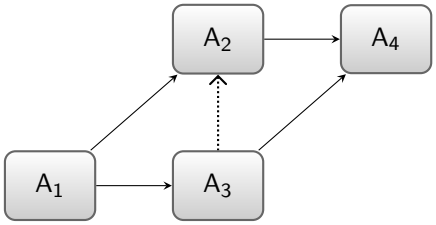


Abbildung 4.6: Zyklenfreier Abhängigkeitsgraph des Prozesses aus Abbildung 4.5

dies der Fall, so ist eine Nebenläufigkeitskontrolle notwendig, die durch eine vorzeitige Synchronisation Lese-Schreib-Abhängigkeiten auflöst. Mit einem graphischen Verfahren kann erkannt werden, wann dies erforderlich ist. Dazu wird ein Graph definiert, der die beiden Typen von Abhängigkeiten zusammenfasst.

Definition 4.6 (Abhängigkeitsgraph) *Ein Abhängigkeitsgraph ist ein gerichteter Graph* $G = (V, E)$*, dessen Knotenmenge* V *die Menge der bereits beendeten, derzeit ausgeführten oder noch auszuführenden Aktivitäten eines Prozesses ist und dessen Kanten die Abhängigkeiten zwischen diesen Aktivitäten repräsentieren:* $E = \prec \cup \lessdot$*. Zur Unterscheidung der unterschiedlichen Typen von Kanten werden in der graphischen Darstellung direkte Präzedenzen durch* $\longrightarrow$ *und Lese-Schreib-Abhängigkeiten durch* $\cdots\!\!\rightarrow$ *gekennzeichnet.*

Der Abhängigkeitsgraph betrachtet also nur für die Ausführung relevante Aktivitäten. Unberücksichtigt bleiben Aktivitäten, die aufgrund negativer Transitionsbedingungen, abgelaufener Aktivierungsbedingungen oder Fehler während der Ausführung in den Zuständen „*skipped*“, „*expired*“ beziehungsweise „*inError*“ verbleiben und damit nie zur Ausführung kommen.

Ist der Abhängigkeitsgraph zyklenfrei, kann aus ihm mindestens eine Serialisierung der Aktivitäten abgeleitet werden, indem diese topologisch sortiert werden. Die Ausführung des Prozesses ist dann auch ohne Nebenläufigkeitskontrolle zu den aus dem Abhängigkeitsgraphen abgeleiteten Serialisierungen äquivalent, somit serialisierbar und deshalb korrekt. Betrachtet man den in Abbildung 4.6 dargestellten Abhängigkeitsgraph von dem Prozess aus Abbildung 4.5, stellt man fest, dass dieser zyklenfrei ist. Wie erwartet ist die einzige aus dem Graph ableitbare Serialisierung die Ausführungsfolge $w = A_1 A_3 A_2 A_4$.

4.2.2 Abhängigkeitskonflikte

Aus einem Abhängigkeitsgraph lässt sich in manchen Fällen jedoch keine einzige Serialisierung ableiten. Dies ist immer dann der Fall, wenn in dem Graph ein Zyklus existiert. Ein Zyklus entsteht, wenn Lese-Schreib-Abhängigkeiten alleine oder zusammen mit den Präzedenzen zu einer zirkulären Abhängigkeit führen. Zirkuläre Abhängigkeiten können in der Ausführung des Prozesses nicht umgesetzt werden, da sie ein Widerspruch in der Reihenfolge der Aktivitäten sind. Dieser Fall wird daher auch als Abhängigkeitskonflikt bezeichnet.

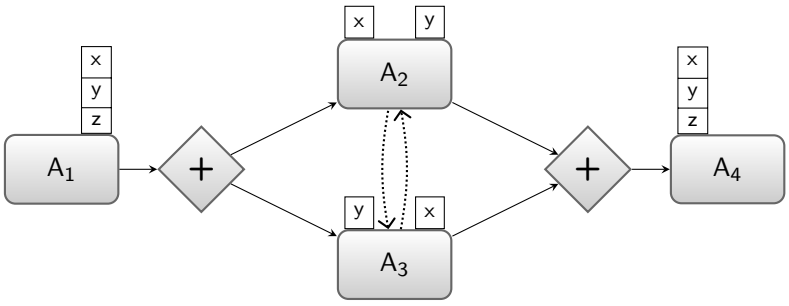


Abbildung 4.7: Lese-Schreib-Abhängigkeiten zwischen A_2 und A_3 ergeben einen Abhängigkeitskonflikt

Definition 4.7 (Abhängigkeitskonflikt) *Ein Zyklus in einem Abhängigkeitsgraph heißt Abhängigkeitskonflikt. Die Menge $\mathfrak{K}$ beschreibt die an diesem Zyklus beteiligten Abhängigkeiten.*

Der Prozess in Abbildung 4.7 besitzt einen Abhängigkeitskonflikt. Da die Aktivität A_2 die Variable x liest, welche von der parallelen Aktivität A_3 geschrieben wird, sollte A_2 in der Serialisierung vor A_3 ausgeführt werden, um eine Synchronisation zu vermeiden. Analog sollte jedoch auch A_3 vor A_2 ausgeführt werden, da A_3 die Variable y liest, welche von A_2 geschrieben wird. Die in diesem Prozess auftretenden Lese-Schreib-Abhängigkeiten können durch die Wahl einer geeigneten Serialisierung nicht aufgelöst werden. Aus diesem Grund ist für die Beseitigung von Abhängigkeitskonflikten eine Synchronisation der parallelen Aktivitäten zwingend notwendig. In dem Beispiel handelt es sich um einen wechselseitigen Ausschluss, da aufgrund der Lese-Schreib-Abhängigkeiten entweder A_2 vor A_3 ausgeführt werden muss, wobei A_3 den von A_2 geschriebenen Wert von y liest, oder es muss A_3 vor A_2 ausgeführt werden, wobei dann A_2 den von A_3 geschriebenen Wert von x lesen muss. Es gilt $\mathfrak{K} = \{A_2 \prec A_3, A_3 \prec A_2\}$.

Ein Abhängigkeitskonflikt muss nicht ausschließlich aufgrund von Lese-Schreib-Abhängigkeiten entstanden sein. Im Prozess von Abbildung 4.8 bestehen die Lese-Schreib-Abhängigkeiten $A_3 \prec A_4$ und $A_4 \prec A_2$. Nur weil das Prozessmodell eine sequentielle Ausführung der Aktivitäten A_2 und A_3 vorsieht, besteht die Präzedenz $A_2 \lessdot A_3$ und es kommt zu einem Zyklus. In diesem Szenario besteht kein wechselseitiger Ausschluss zwischen zwei Aktivitäten sondern eine zirkuläre Abhängigkeit zwischen drei Aktivitäten, die durch die Synchronisation der Replikate aufgebrochen werden muss.

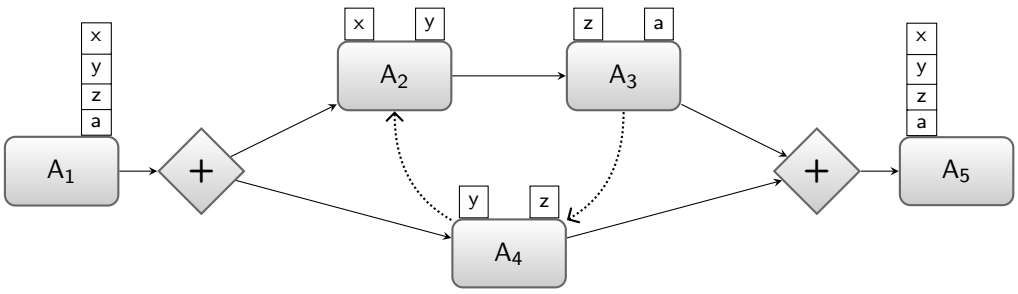


Abbildung 4.8: Prozess mit zyklischen Lese-Schreib-Abhängigkeiten

4.2.3 Paralleles Schreiben

Wie bereits erwähnt, hängt das berechnete Ergebnis bei von mehreren parallelen Aktivitäten geschriebenen Variablen von der sich ergebenden Serialisierung ab. Dies hängt damit zusammen, dass in dem vorgestellten Ansatz geschriebene Variablen unabhängig von gelesenen Variablen betrachtet werden. In der Literatur wird häufig angenommen, dass geschriebene Variablen zuvor gelesen werden und somit $\mathcal{S}(A_i) \subseteq \mathcal{L}(A_i)$ gilt (vgl. [Dav84] sowie beispielsweise die Nebenläufigkeitskontrolle mit Sperrverfahren in Abschnitt 3.1.2). Bei mobilen Prozessen sind jedoch die Input- und Output-Container klar voneinander getrennt. Diese Differenzierung fortzuführen lohnt sich gerade bei der Ausführung von Prozessen in mobilen Umgebungen, da so unnötige Synchronisationen vermieden werden können.

Zur Veranschaulichung der Idee wird ein kurzes Beispiel betrachtet [BSR80]: Zwei Aktivitäten werden parallel ausgeführt, wobei eine die Variable „*Name*“ liest und daraufhin die Variable „*Telefonnummer*“ schreibt. Die andere Aktivität liest „*Personen-ID*“ desselben Datensatzes und ändert in dessen Abhängigkeit ebenfalls „*Telefonnummer*“. Da sich die geschriebenen Werte für die Telefonnummer unterscheiden können, hängt der resultierende Wert bei einer sequentiellen Ausführung von der Reihenfolge ab, in der die Aktivitäten ausgeführt werden. Die parallele Ausführung der Aktivitäten ist somit in jedem Fall serialisierbar, es muss nur eine der möglichen Serialisierungen gewählt werden. Bei der Synchronisation der Replikate wird nun der Wert der gemäß der gewählten Serialisierung auf der Variable „*Telefonnummer*“ zuletzt durchgeführten Schreiboperation verwendet. Dieses Szenario erfordert daher keine Synchronisation, obwohl bei der Verwendung von Sperren beispielsweise ein wechselseitiger Ausschluss entstanden wäre.

Im Zusammenhang mit Lese-Schreib-Abhängigkeiten wird die Menge der möglichen Serialisierungen durch die von $\prec$ entstehenden zusätzlichen Abhängigkeiten jedoch eingeschränkt. Bei der Synchronisation von Replikaten kann nur aus den Serialisierungen gewählt werden, die aus dem Abhängigkeitsgraph ableitbar sind. Im Prozess von Abbildung 4.9 wird beispielsweise die Variable x von den Aktivitäten A_3 und A_4 parallel geschrieben. Da jedoch die Lese-Schreib-Abhängigkeit $A_4 \prec A_2$ existiert, ist nur die Serialisierung $w = A_4A_2A_3$ möglich. In dieser Serialisierung wird x zuletzt durch die

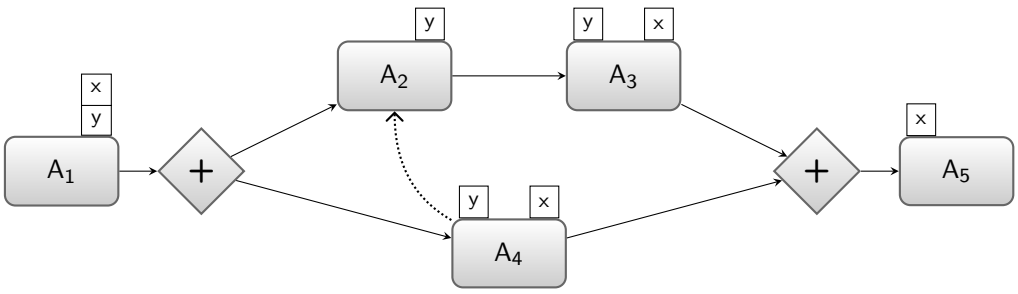


Abbildung 4.9: A_3 und A_4 schreiben die gleiche Variable

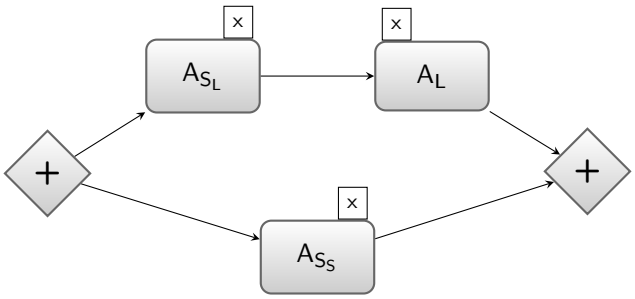


Abbildung 4.10: Die Aktivität A_L liest eine Variable, die auf demselben und einem anderen Replikat parallel geschrieben wird.

Aktivität A_3 geändert. Somit muss der von A_3 geschriebene Wert bei der Synchronisation der Replikate verwendet werden, um schließlich von A_5 gelesen werden zu können.

In einem bestimmten Zusammenspiel von Lese-Schreib-Abhängigkeiten und parallelem Schreiben kommt es dazu, dass die aus der Lese-Schreib-Abhängigkeit folgende Ordnung die Menge der äquivalenten Serialisierungen unnötig einschränkt. Dies ist immer dann der Fall, wenn zwei Aktivitäten A_L und A_{S_S} eine Lese-Schreib-Abhängigkeit aufweisen, es aber eine weitere Aktivität A_{S_L} gibt, welche die betroffene Variable auf dem Replikat von A_L vor dessen Ausführung ändert (vgl. Abbildung 4.10):

$$A_{S_L} < A_L \ \wedge \ A_{S_S} \parallel A_L \ \wedge \ A_{S_S} \parallel A_{S_L} \ \wedge \ \mathcal{L}(A_L) \cap \mathcal{S}(A_{S_L}) \cap \mathcal{S}(A_{S_S}) \neq \emptyset$$

Damit die Ausführung ohne Synchronisation auskommt, muss A_L den von A_{S_L} auf demselben Replikat geschriebenen Wert lesen. Aus diesem Grund darf A_{S_S} in einer Serialisierung nur entweder nach A_L oder vor A_{S_L} vorkommen. Während in Definition 4.5 nur der erste Fall berücksichtigt wurde, gibt es also theoretisch zwei Möglichkeiten zur Behandlung des Problems, von denen in manchen Situationen jedoch nur eine zu einem Abhängigkeitskonflikt führt. Um die konfliktfreie Variante wählen zu können, müssten daher zwei Abhängigkeitsgraphen erzeugt und analysiert werden. Tritt diese Situation mehrmals in einem Prozess auf, sind alle Kombinationen – pro Auftreten je zwei Varianten – zu untersuchen. Bei n Lese-Schreib-Abhängigkeiten müssten daher bis zu 2^n Abhängigkeitsgraphen erzeugt und auf Zyklen untersucht werden. Aufgrund des exponentiellen Wachstums, das gerade für leistungsschwache mobile Geräte ein Problem darstellen kann, verzichtet dieses Konzept auf die Untersuchung aller Kombinationen. Statt dessen wird nur die Reihenfolge A_L vor A_{S_S} in der Serialisierung berücksichtigt, die sich gemäß Definition 4.5 in der Lese-Schreib-Abhängigkeit $A_L \prec A_{S_S}$ wiederfindet. Dadurch kann auf eine Ausnahmebehandlung verzichtet werden, und komplexe Berechnungen werden unnötig. Bei manchen Prozessen werden deshalb möglicherweise zu viele Abhängigkeitskonflikte erkannt; somit müssen teilweise zusätzliche – theoretisch unnötige – Synchronisationen durchgeführt werden. Dies ist jedoch vertretbar, da ansonsten der Prozess aufgrund der möglichen fehlenden Verfügbarkeit von leistungsstarken Geräten eventuell gar nicht zur Ausführung gekommen wäre.

4.2.4 Potentieller Kontrollfluss

Die Konflikterkennung mit Hilfe des Abhängigkeitsgraphen erfolgt auf der Grundlage des potentiellen Kontrollflusses, da nur dieser durch das Prozessmodell gegeben ist. Es können daher auch nur potentielle Abhängigkeitskonflikte erkannt werden. Ob die Konflikte während der Ausführung tatsächlich auftreten, hängt von der Auswertung der Bedingungen im Kontrollfluss zur Laufzeit ab. Dies betrifft insbesondere die selektive Ausführung, da hier erst zur Laufzeit ermittelt werden kann welcher Ausführungspfad gewählt werden muss. So ist beispielsweise vor der Ausführung des Prozesses aus Abbildung 4.11 nicht bekannt, ob A_4 oder A_5 ausgeführt werden muss, da die Selektionsbedingung von dem von A_3 geschriebenen Wert für b abhängt.

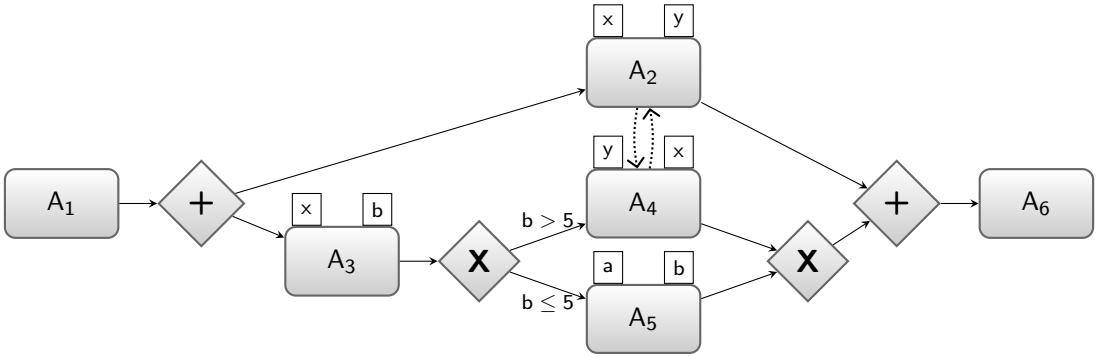


Abbildung 4.11: Da die Aktivitäten A_4 und A_5 nur alternativ ausgeführt werden, ist der Zyklus zwischen A_2 und A_4 nur in manchen Fällen relevant.

Konkret bedeutet dies, dass in manchen Fällen während der Ausführung eines Prozesses bereits Maßnahmen zur Synchronisation paralleler Aktivitäten getroffen worden sein können – wie etwa das Verzögern der Ausführung einer Aktivität – und bei der Ausführung eines OR-Splits auf einem parallelem Pfad festgestellt wird, dass an dieser Stelle gar kein Abhängigkeitskonflikt auftritt, da die Selektion andere Aktivitäten zur Ausführung ausgewählt hat. Daher muss in diesem Beispiel zwischen den Aktivitäten A_2 und A_4 eine Synchronisation erfolgen, falls $b > 5$ ist. Dies kann jedoch erst festgestellt werden, wenn A_3 ausgeführt wurde und somit der Wert für b feststeht. Erschwerend kommt hinzu, dass A_2 potentiell auf einem anderen Gerät als A_3 ausgeführt wird und dadurch die Entscheidung über den auszuführenden Pfad zunächst nicht bekannt ist.

Dieses Problem lässt sich prinzipbedingt nicht vor der Ausführung eines Prozesses lösen. Erst zur Laufzeit kann die Problematik abgeschwächt werden, indem beispielsweise nach jedem Split die getroffene Entscheidung an alle Geräte, die potentiell konfligierende Aktivitäten ausführen sollen, übertragen wird. Empfängt ein Gerät eine derartige Information, könnte es in der lokal gespeicherten Prozessbeschreibung den Zustand der betroffenen Aktivitäten auf „*skipped*“ setzen oder diese gänzlich aus der Beschreibung entfernen. Wird nun der Abhängigkeitsgraph unter Verwendung dieser Daten aktualisiert, führen bei der Konflikterkennung keine falschen Abhängigkeiten zu nie auftreten-

den Abhängigkeitskonflikten. Trotz alledem können aber keine Abhängigkeitskonflikte auftreten, die von der Konflikterkennung nicht registriert wurden.

Neben der selektiven Ausführung betrifft die dargestellte Problematik abenso die iterative Ausführung sowie die mehrfache Ausführung von Aktivitäten. Hier kann ebenfalls nicht im Voraus bestimmt werden wie oft die Aktivitäten auszuführen sind. Solche Kontrollflussstrukturen können daher bei der Erkennung von Abhängigkeitskonflikten nur als zusammengefasste Aktivität mit gemeinsamen Input- und Output-Containern betrachtet werden, die in einer Serialisierung unteilbar auftritt.

4.2.5 Ablauf der Erkennung von Abhängigkeitskonflikten

Es wurde gezeigt, dass die Ausführung eines Prozesses, bei der jeder parallele Ausführungspfad über ein eigenes Replikat verfügt, unter bestimmten Voraussetzungen keine zusätzliche Synchronisation durch die Nebenläufigkeitskontrolle erfordert und dennoch korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist. Sind diese Voraussetzungen nicht gegeben, tritt ein Abhängigkeitskonflikt auf, der mit Hilfe eines Abhängigkeitsgraphen erkannt werden kann. Dazu wird ein mathematisches Modell des Prozesses benötigt. Damit die Erkennung von Abhängigkeitskonflikten nicht auf bestimmte Prozessbeschreibungssprachen beschränkt ist, verwendet das Metamodell nur generische Komponenten (vgl. Abschnitt 4.1.2). Konkrete Prozessbeschreibungssprachen können über Adapter integriert werden, welche die in einer bestimmten Sprache formulierte Prozessbeschreibung in das mathematische Modell überführen.

In Abbildung 4.12 ist der Ablauf der Erkennung von Abhängigkeitskonflikten dargestellt. Nachdem der entsprechende Adapter das mathematische Modell des Prozesses erzeugt hat, werden die Lese-Schreib-Abhängigkeiten gemäß Definition 4.5 bestimmt und der Abhängigkeitsgraph erzeugt. In diesem wird nun eine Zyklensuche durchgeführt. Das Ergebnis ist eine Menge von Abhängigkeitskonflikten, die jeweils durch die Menge der an dem Abhängigkeitskonflikt beteiligten Abhängigkeiten repräsentiert werden (vgl. Definition 4.7).

Die Erkennung von Abhängigkeitskonflikten wird sowohl zur Modellierungszeit als auch zur Laufzeit angewendet. Zur Modellierungszeit dient sie dem Modellierer als Hilfsmittel bei der Optimierung eines Prozesses. Er ist dadurch in der Lage, auf die in seinem Prozess auftretenden Abhängigkeitskonflikte angemessen zu reagieren. So könnte er beispielsweise versuchen, Abhängigkeitskonflikte durch das Umstrukturieren des Prozesses zu verhindern oder durch das Hinzufügen von zusätzlichen Präzedenzen

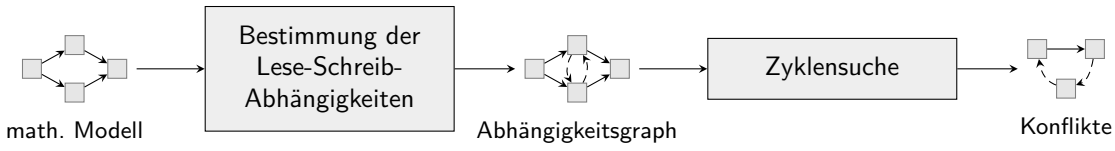


Abbildung 4.12: Ablauf der Erkennung von Abhängigkeitskonflikten

explizit aufzulösen. Er wird aber auch davor bewahrt, ungünstige Rollenbeziehungen anzugeben. So ist zur Auflösung eines Abhängigkeitskonfliktes die Kommunikation zwischen den Geräten, auf denen konfligierende Aktivitäten ausgeführt werden, notwendig. Wird dies bereits bei der Modellierung berücksichtigt, kann die Ausführung des Prozesses gerade in Umgebungen mit schlechten Kommunikationsverbindungen möglicherweise deutlich schneller erfolgen.

Zur Laufzeit muss die Ausführungsumgebung eines Prozesses die Erkennung von Abhängigkeitskonflikten ausführen um zu überprüfen, ob zwischen parallelen Ausführungspfaden eine Nebenläufigkeitskontrolle erforderlich ist. Diese muss dann die gefundenen Abhängigkeitskonflikte unter Berücksichtigung der in Abschnitt 4.3 vorgestellten Datenklassen auflösen.

4.3 Definition von Datenklassen

In Kapitel 3 wurde Serialisierbarkeit als generisches Korrektheitskriterium für die Ausführung von Prozessen eingeführt. Bei der Verwendung von replizierten Daten tritt Eine-Kopie-Serialisierbarkeit an dessen Stelle. Dabei wird die Ausführung eines Prozesses auf verschiedenen Replikaten als korrekt bezeichnet, wenn sie äquivalent zu einer seriellen Ausführungsfolge auf einer einzelnen Kopie ist. Es wurde allerdings auch klar gestellt, dass Serialisierbarkeit beziehungsweise Eine-Kopie-Serialisierbarkeit lediglich ein mögliches Korrektheitskriterium darstellt. Es ist zwar aufgrund seiner Generalität für beliebige Prozesse anwendbar und liefert sinnvolle Ergebnisse. In einigen Fällen ist es jedoch unnötig, ein derart restriktives Kriterium zu benutzen, da die Menge der erlaubten Ausführungsfolgen sehr stark beschränkt wird. Diese Einschränkung führt zum Auftreten von Abhängigkeitskonflikten (vgl. Abschnitt 4.2), die wiederum die Synchronisation zwischen parallelen Ausführungspfaden notwendig machen. Prozessspezifische Korrektheitskriterien können hingegen in vielen Fällen die Anzahl der notwendigen Synchronisationen reduzieren und dadurch die Ausführung eines Prozesses in mobilen Umgebungen deutlich beschleunigen.

Der in diesem Kapitel vorgestellte Ansatz ist daher, anwendungsspezifisch Garantien bezüglich der Konsistenz der von einem Prozess verwendeten Daten zu definieren und die Variablen eines Prozesses entsprechend auszuzeichnen, um während der Ausführung des Prozesses den Synchronisationsaufwand reduzieren zu können. Der Modellierer kann dabei aus einer Menge von Datenklassen wählen und die Prozessbeschreibung um eine Zuordnung der verwendeten Variablen zu je einer Datenklasse ergänzen. Die Einbeziehung des Modellierers ist deshalb notwendig, weil in dem ursprünglichen Prozessmodell keinerlei semantische Informationen über den Zweck des Prozesses enthalten sind und somit nicht festgestellt werden kann, wann eine Ausführung des Prozesses korrekt ist. Standardmäßig wird daher angenommen, dass die Ausführung des Prozesses Eine-Kopie-Serialisierbarkeit erfordert. Durch die explizite Auszeichnung der

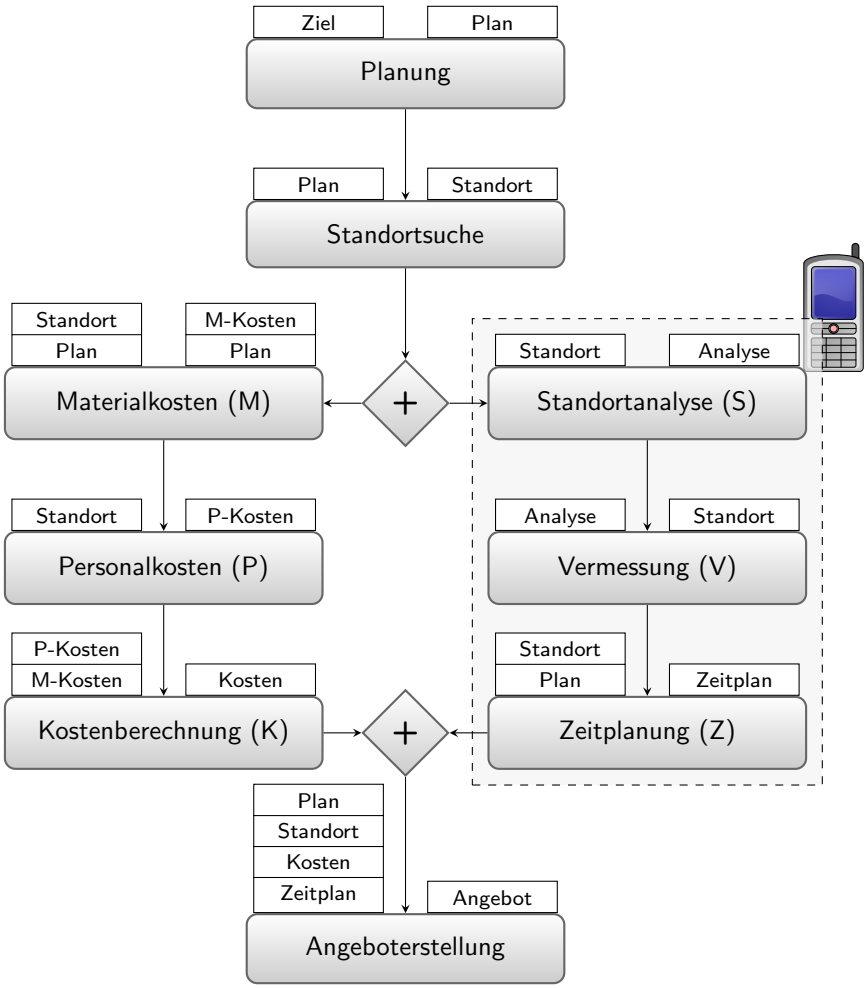


Abbildung 4.13: Prozess zur Planung von Unternehmensniederlassungen

Variablen des Prozesses ist es hingegen möglich festzustellen, ob und wann auf eine serialisierbare Ausführung verzichtet werden darf.

Die Idee wird nun anhand des in Kapitel 1 vorgestellten Prozesses zur Planung von Unternehmensniederlassungen veranschaulicht. In Abbildung 4.13 ist der gesamte Prozess inklusive der Input- und Output-Container aller Aktivitäten dargestellt. Mit dem in Abschnitt 4.2 vorgestellten Verfahren kann nun erkannt werden, dass in diesem Prozess mehrere Abhängigkeitskonflikte auftreten, an denen immer die Variable „*Standort*“ beteiligt ist (vgl. den Abhängigkeitsgraph in Abbildung 4.14). Um eine serialisierbare Ausführung von diesem Prozess zu erreichen, muss daher eine Synchronisation zwischen den parallelen Ausführungspfaden erfolgen. Bei näherer Betrachtung des Prozesses zeigt sich jedoch, dass Eine-Kopie-Serialisierbarkeit für die korrekte Ausführung des Prozesses nicht zwingend notwendig ist. In der Aktivität „*Standortsuche*“ wird der Standort zunächst durch die Wahl eines Grundstückes bestimmt. Während der parallelen Ausführung wird der geplante Standort des Gebäudes durch exakte Vermessung auf dem Grundstück festgelegt. Für die Berechnung der Material- und Personalkosten ist es jedoch unerheblich, welchen Ort auf dem Grundstück die Standortangabe bezeichnet,

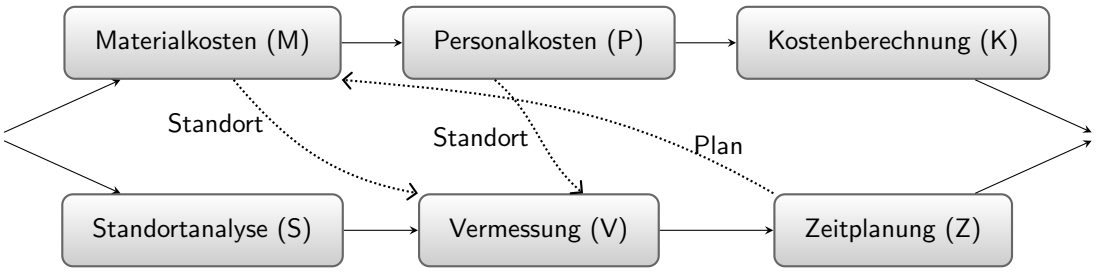


Abbildung 4.14: Abhängigkeitsgraph der parallelen Aktivitäten des Prozesses aus Abbildung 4.13

da sich die für die Kostenberechnung relevante Fahrtweglänge dadurch nicht signifikant ändert. Aus diesem Grund ist die Ausführung des Prozesses auch dann korrekt, wenn während der parallelen Ausführung ein veralteter Wert von „*Standort*“ gelesen wird. *Veraltet* bedeutet hier, dass ein Wert gelesen wird, der in einer seriellen Ausführungsfolge bereits überschrieben worden wäre. Ist dieser Umstand während der Ausführung des Prozesses bekannt, so kann auf die zusätzliche Synchronisation der parallelen Aktivitäten verzichtet werden, da die Lese-Schreib-Abhängigkeiten $M \prec V$ und $P \prec V$ wegfallen und der Abhängigkeitsgraph nun zyklenfrei ist.

Beliebige Ausprägungen der anwendungsspezifischen Garantien bezüglich der Konsistenz der von einem Prozess verwendeten Daten sind konstruierbar. Es können dabei insbesondere die von Konsistenzmodellen bekannten Ansätze (vgl. Abschnitt 3.3) verwendet werden. So ist beispielsweise denkbar, dass ein Datum ein bestimmtes Alter nicht überschreiten oder die Abweichung der Werte zwischen den Replikaten der parallelen Pfade nicht zu groß werden darf. Damit jedoch die anwendungsspezifische Auszeichnung einer Variable überhaupt berücksichtigt werden kann, ist es notwendig, dass die Ausführungsumgebung mit der geforderten Garantie umgehen und diese sicherstellen kann. Daher werden in diesem Kapitel sogenannte Datenklassen eingeführt. Eine *Datenklasse* beschreibt eine Garantie bezüglich der Konsistenz von Variablen und ist dabei so allgemein formuliert, dass sie in verschiedenen Prozessen angewendet werden kann. Datenklassen stellen also den gemeinsamen Nenner dar, mit dem der Modellierer des Prozesses einer Ausführungsumgebung mitteilt, wie mit den in seinem Prozess verwendeten Variablen umzugehen ist. Denn ist eine Variable eines Prozesses mit einer Datenklasse ausgezeichnet, welche der Ausführungsumgebung bekannt ist, so kann eine der Datenklasse entsprechende Lockerung der Eine-Kopie-Serialisierbarkeit während der Ausführung des Prozesses vorgenommen werden.

Die Reduktion auf wenige Datenklassen ermöglicht eine höhere Kompatibilität zwischen Prozessen und Ausführungsumgebungen, da letztere aufgrund der Beschränktheit mobiler Geräte nicht beliebig viele Datenklassen implementieren können. Auf der anderen Seite tritt ein Verlust von Optimierungspotenzial auf, wenn eine für den vorliegenden Prozess ideale Datenklasse nicht existiert. Aber auch bei der Wahl der nächstbesten Alternative kann bereits viel Synchronisationsaufwand gespart werden. In dem

obigen Beispielprozess kann die Variable „*Standort*“ sogar gänzlich von der Synchronisation ausgenommen werden.

Zur Definition einer Datenklasse gehört nicht nur die für den Benutzer interessante Garantie über die Konsistenz einer Variable sondern auch die Realisierung der Konfliktauflösung in einer Implementierung der Ausführungsumgebung. De facto unterscheiden sich Datenklassen also nur in der Art und Weise der Behandlung von Abhängigkeitskonflikten eines Prozesses. Exemplarisch soll dies im Folgenden anhand von drei Datenklassen gezeigt werden. Da eine Ausführungsumgebung in jedem Fall eine serialisierbare Ausführung von Prozessen ermöglichen muss, wird diese Datenklasse zuerst diskutiert. Anschließend wird eine Datenklasse betrachtet, bei der auch bei Abhängigkeitskonflikten keine Synchronisation zwischen parallelen Aktivitäten notwendig ist. Diese beiden Datenklassen stellen die zwei Extrema bezüglich der möglichen Garantien dar. Ein Beispiel für eine dazwischen liegende Datenklasse rundet die Diskussion ab.

4.3.1 Datenklasse: Serialisierbar

Die Standard-Datenklasse für Variablen eines Prozesses, bei dem keine Auszeichnung vorgenommen wurde oder sie für die Ausführungsumgebung nicht interpretierbar ist, muss Eine-Kopie-Serialisierbarkeit garantieren. Nur so kann auch im Fehlerfall – insbesondere aber wenn der Modellierer seine Intention nicht durch die Auszeichnung der Variablen mitgeteilt hat – die korrekte Ausführung des Prozesses sichergestellt werden. Die Datenklasse „*Serialisierbar*“ garantiert, dass die Ausführung des Prozesses in Bezug auf die damit ausgezeichneten Variablen äquivalent zu einer seriellen Ausführungsfolge ist. Existieren Prozessvariablen, die mit einer anderen Datenklasse ausgezeichnet sind, wird die Äquivalenz zu einer seriellen Ausführungsfolge ohne Berücksichtigung dieser Variablen gemessen.

In dieser Datenklasse müssen somit alle Abhängigkeitskonflikte berücksichtigt werden, die durch als serialisierbar ausgezeichnete Variablen entstehen. Jeder Abhängigkeitskonflikt $\aleph$ muss durch eine zusätzliche Synchronisation aufgelöst werden, damit garantiert ist, dass die Ausführung des Prozesses serialisierbar ist. Dazu wird zunächst eine an dem Abhängigkeitskonflikt beteiligte Lese-Schreib-Abhängigkeit $A_L \prec A_S \in \aleph$ ausgewählt. Anstatt diese Abhängigkeit zu befriedigen, wird die zusätzliche Präzedenz $A_S \preccurlyeq A_L$ in das Prozessmodell eingefügt. Dies führt dazu, dass vor der Ausführung der Aktivität A_L eine zusätzliche Synchronisation erfolgt und somit A_L die von A_S geschriebenen Werte lesen kann. Nach der Änderung des Prozessmodells müssen die Lese-Schreib-Abhängigkeiten und der Abhängigkeitsgraph neu berechnet werden, da nun nicht mehr $A_S \parallel A_L$ gilt sondern $A_S < A_L$ – der Zyklus ist gebrochen.

Das Einfügen von zusätzlichen Präzedenzen zur Synchronisation paralleler Aktivitäten verändert die Semantik eines Prozesses nur insofern als die Parallelität bei der Ausführung eingeschränkt wird. Dies ist bei pessimistischer Nebenläufigkeitskontrolle jedoch immer der Fall (vgl. Kapitel 3) und wird in diesem Ansatz im Unterschied zu an-

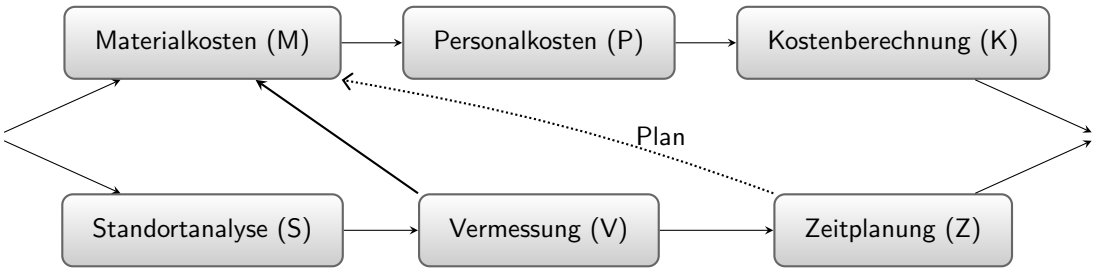


Abbildung 4.15: Dieser Abhängigkeitsgraph zeigt, wie der Abhängigkeitskonflikt des Prozesses aus Abbildung 4.13 durch das Einfügen einer Präzedenz aufgelöst wurde.

deren Verfahren explizit im Prozessmodell verankert. Durch die neue Präzedenz kann kein Zyklus im Kontrollfluss entstehen, da ansonsten $A_L < A_S$ hätte gelten müssen. Dann würde jedoch nicht $A_L \prec A_S$ existieren und somit keine Konfliktauflösung. Des Weiteren kann im Abhängigkeitsgraph kein neuer Zyklus entstehen, da die eingefügte Präzedenz bereits indirekt durch den aufzulösenden Zyklus vorhanden war.

Abbildung 4.14 zeigt, dass der Abhängigkeitsgraph des Prozesses aus Abbildung 4.13 bei fehlender Auszeichnung der Variablen mit Datenklassen zwei Zyklen aufweist. Zur Sicherstellung der Eine-Kopie-Serialisierbarkeit muss daher eine zusätzliche Synchronisation erfolgen. Die erste an einem Zyklus beteiligte Lese-Schreib-Abhängigkeit ist $M \prec V$. Wird das obige Verfahren mit dieser Abhängigkeit ausgeführt, so entsteht das in Abbildung 4.15 dargestellte Prozessmodell. Die Lese-Schreib-Abhängigkeit $P \prec V$ tritt nun nicht mehr auf, da die beiden Aktivitäten nicht mehr parallel ausgeführt werden. $Z \prec M$ muss hingegen weiterhin bei der Wahl der äquivalenten Serialisierung beachtet werden; dies führt jedoch zu keinem Abhängigkeitskonflikt mehr. Die korrekte Ausführung ist nun sichergestellt und eine äquivalente Serialisierung kann aus dem neuen Abhängigkeitsgraph abgegelesen werden.

In dem Beispiel konnten zwei Abhängigkeitskonflikte durch eine einzige Synchronisation aufgelöst werden. Wäre hingegen die Lese-Schreib-Abhängigkeit $P \prec V$ verwendet worden, würde nach wie vor ein Abhängigkeitskonflikt existieren, der durch eine weitere Synchronisation aufgelöst werden müsste. Für eine effiziente Ausführung der parallelen Aktivitäten sollte daher geprüft werden, wie mit möglichst wenig Synchronisierungen alle Abhängigkeitskonflikte aufgelöst werden können.

Daneben ist aber auch der Zeitpunkt der Entscheidung, welche Lese-Schreib-Abhängigkeit durch eine Synchronisation ersetzt wird, für die Performanz relevant. Zur Veranschaulichung wird der wechselseitige Ausschluss aus Abbildung 4.16 betrachtet. Aufgrund der Symmetrie geht aus dem Prozessmodell nicht hervor, welche der beiden Lese-Schreib-Abhängigkeiten zur Konfliktauflösung verwendet werden sollte. Es wird daher angenommen, dass sich für $A_4 \prec A_2$ entschieden und somit die Präzedenz $A_2 \lessdot A_4$ eingefügt wird. Während der Ausführung des Prozesses kann es nun jedoch zu der Situation kommen, dass die Bearbeitung von A_1 sehr lange dauert, während A_3 schnell beendet ist. Aufgrund der Präzedenzen kann nun weder A_2 und damit auch nicht A_4

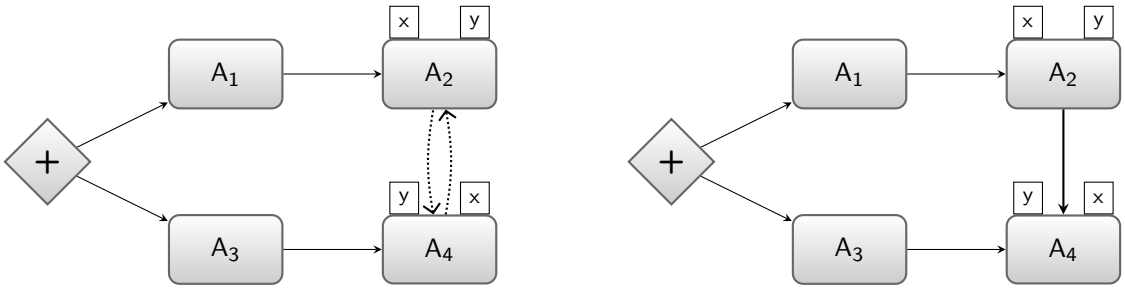


Abbildung 4.16: Der Abhängigkeitskonflikt zwischen den Aktivitäten A_2 und A_4 (links) wird durch das Einfügen der Präzedenz $A_2 \lessdot A_4$ (rechts) aufgelöst

gestartet werden. Bei der Wahl der Lese-Schreib-Abhängigkeit $A_2 \prec A_4$ zur Konfliktauflösung könnte hingegen A_4 bereits ausgeführt werden.

Das Beispiel macht deutlich, dass es vorteilhaft sein kann, die Entscheidung, welche Lese-Schreib-Abhängigkeit zur Konfliktauflösung gewählt wird, erst direkt vor dem Auftreten des Abhängigkeitskonfliktes zu treffen. Problematisch ist es allerdings, wenn die mobilen Geräte, welche die parallelen Pfade ausführen, zu diesem Zeitpunkt nicht miteinander kommunizieren können. Dies ist jedoch zwingend notwendig, damit die Geräte sich über die konkrete Wahl der Konfliktauflösung einig werden. Bei pessimistischer Ausführung kann nun die Wartezeit, bis die Verbindung wiederhergestellt ist, länger sein als die Wartezeit auf lang laufende Aktivitäten – wie bei der Aktivität A_1 im obigen Beispiel. Im Voraus kann jedoch nicht festgestellt werden, welche Methode das bessere Ergebnis erzielt.

Um dennoch den Vorteil der späten Konfliktauflösung nutzen zu können, wird in Abschnitt 4.6 die optimistische Konfliktauflösung vorgestellt. Ähnlich wie bei der in Abschnitt 3.2.4 diskutierten optimistischen Ausführung mit verzögerter Auswertung, wird hier im Falle der fehlerhaften Kommunikation optimistisch lokal eine Konfliktauflösung vorgenommen. Sobald die Netzverbindung wiederhergestellt ist, erfolgt eine Einigung mit allen an dem Abhängigkeitskonflikt beteiligten Pfaden bezüglich der Konfliktauflösung. Die angesprochene Minimierung der Anzahl zusätzlicher Präzedenzen in Abhängigkeit von der gewählten Konfliktauflösung wird daher aufgrund des hohen Aufwands für die mobilen Geräte nicht weiter betrachtet.

4.3.2 Datenklasse: Unsynchronisiert

In manchen Fällen können Abhängigkeitskonflikte gänzlich ignoriert werden. Dies ist beispielsweise der Fall, wenn die Änderung der Daten in so geringem Ausmaße stattfindet, dass dies für die lesenden Aktivitäten irrelevant ist. In dem eingangs vorgestellten Prozess zur Planung von Niederlassungen ist dies beispielsweise für die Variable „*Standort*“ der Fall. Lese-Schreib-Abhängigkeiten, die durch eine so ausgezeichnete Variable entstehen, werden bei der Konflikterkennung nicht berücksichtigt. Je nach Prozess

treten daher deutlich weniger oder gar keine Abhängigkeitskonflikte mehr auf, was sich entsprechend positiv in der Anzahl der notwendigen Synchronisierungen niederschlägt.

Da die Daten der verschiedenen Replikate bei der Synchronisation des Kontrollflusses – beispielsweise durch einen *Join* – gemäß einer aus dem Abhängigkeitsgraph ableitbaren Serialisierung zusammengeführt werden, kann bei der Verwendung dieser Datenklasse das in Kapitel 3 beschriebene Problem der verlorengegangen Änderungen auftreten. Diese Gefahr besteht jedoch immer, wenn Serialisierbarkeit gelockert wird und ist der Preis für die effizientere Ausführung des Prozesses in mobilen Umgebungen. Aus diesem Grund muss für jeden Prozess abgewägt werden, welche Datenklasse in Frage kommt und ob die damit verbundenen Probleme vertretbar sind.

4.3.3 Datenklasse: Maximales Alter

In manchen Fällen kann zwar auf Serialisierbarkeit verzichtet werden, aber die vollständige Vernachlässigung der Synchronisation führt wiederum zu einem falschen Ergebnis. Als Beispiel für ein solches Szenario wird im Folgenden eine Datenklasse betrachtet, bei der zwar die verwendeten Werte nicht immer auf dem neuesten Stand sein müssen, eine gewisse Aktualität jedoch garantiert sein muss. Dies ist bei Daten der Wettervorhersage der Fall. In der Regel weist die Aktualisierung einer Vorhersage nur kleine Abweichungen auf. Ein Prozess, der diese Daten verwendet, kann also ein gewisses Alter der Daten akzeptieren.

Lese-Schreib-Abhängigkeiten, die durch eine Variable dieser Datenklasse entstehen, werden bei der Bildung des Abhängigkeitsgraphen nicht berücksichtigt. Sobald in einem parallelen Ausführungspfad auf die Variable zugegriffen wird, muss daher untersucht werden, ob es parallele Aktivitäten gibt, welche diese Variable verändern. Nur wenn dies der Fall ist, könnte der lokale Wert der Variable veraltet sein und die Aktualität muss daher überprüft werden. Dazu muss die Ausführungsumgebung den Zeitpunkt der letzten Aktualisierung dieser Variable mitführen. Liegt dieser zu weit in der Vergangenheit, muss zunächst eine Synchronisation zur Aktualisierung der Variable erfolgen. Diese wird dann explizit im Prozessmodell durch das Einfügen einer zusätzlichen Präzedenz dokumentiert.

4.3.4 Auszeichnung von Prozessen mit Datenklassen

Im Anschluss an die Analyse des Prozesses mit Hilfe der Konflikterkennung kann der Modellierer die für seinen Prozess geeigneten Datenklassen wählen. Damit die Ausführungsumgebung die gewählten Datenklassen berücksichtigen kann, muss nun eine eindeutige Auszeichnung erfolgen. Dabei sind prinzipiell zwei Vorgehensweisen möglich. Bei der internen Auszeichnung erfolgt die Zuordnung direkt in der Prozessbeschreibung. Da gängige Prozessbeschreibungssprachen jedoch keine Verwendung von Datenklassen vorsehen, ist eine Erweiterung der Sprache um entsprechende Konstrukte

notwendig. Dies kann allerdings nur dann realisiert werden, wenn die Sprache erweiterbar ist, was nicht auf jede in der Praxis eingesetzte Sprache zutrifft.

Bei der externen Auszeichnung bleibt die ursprüngliche Prozessbeschreibung unverändert und die Zuordnung der Datenklassen erfolgt in einem zusätzlichen Datencontainer, welcher der Ausführungsumgebung neben der Prozessbeschreibung bereitgestellt werden muss. Es kann sich dabei um eine Datei mit einfacher Notation handeln.

Beide Vorgehensweisen besitzen Vor- und Nachteile. Während bei der internen Auszeichnung möglicherweise ein sehr hoher Aufwand für die Anpassung konkreter Prozessbeschreibungssprachen zu bewältigen ist, muss bei der externen Auszeichnung mit mehreren Dateien umgegangen werden. Aufgrund der höheren Flexibilität wird in dieser Arbeit die externe Auszeichnung bevorzugt und in Abschnitt 5.1.1 ein konkretes Datenmodell diskutiert.

4.4 Distribution des Kontrollflusses

Der Modellierer eines Prozesses hat durch die Erkennung von Abhängigkeitskonflikten und die Definition von Datenklassen die Möglichkeit, Synchronisationen zwischen parallelen Ausführungspfaden zu minimieren. Diese Optimierungen finden zur Modellierungszeit statt. In den folgenden Abschnitten wird die parallele Ausführung zur Laufzeit behandelt. Dieser Abschnitt beginnt daher mit der Distribution des Kontrollflusses, also der Verzweigung eines sequentiellen Ausführungspfades in mehrere parallel ausgeführte Pfade. Modelliert werden kann die Verteilung durch einen AND-Split beziehungsweise OR-Split, bei dem mehrere Transitionsbedingungen zu „*wahr*“ auswerten, oder durch einen asynchronen Aufruf eines Subprozesses. Hierbei sind zwei Fragestellungen zu behandeln. Zunächst muss geklärt werden, welche Anforderungen mobile Geräte erfüllen müssen, damit sie an der Ausführung von parallelen Pfaden teilnehmen können. Bei der Distribution des Kontrollflusses sollten nur Geräte gewählt werden, die diesen Anforderungen entsprechen. Des Weiteren muss geklärt werden, welche Daten während der parallelen Ausführung benötigt werden und daher bei der Distribution repliziert und an die teilnehmenden Geräte übertragen werden müssen.

4.4.1 Auswahl der Teilnehmer

Bei der Auswahl der Geräte, die an der parallelen Ausführung eines Prozesses teilnehmen, spielen sowohl allgemeine Überlegungen, welche Geräte einen Prozess ausführen sollten, als auch spezielle Anforderungen der Parallelität eine Rolle. Bevor die sich aus der Parallelität ergebenden Anforderungen behandelt werden, sollen zunächst Aspekte betrachtet werden, die allgemein bei jeder Migration eines Prozesses zu beachten sind.

Migrationen finden in sequentiellen Prozessen immer dann statt, wenn ein Gerät nicht in der Lage ist, die Ausführung des Prozesses fortzuführen. Um eine Verbesserung der Situation zu erreichen, sind bei der Migration daher solche Geräte vorzuziehen, welche

die nachfolgenden Aktivitäten ausführen können. Neben dieser funktionalen Anforderung ist es darüber hinaus sinnvoll, auch nicht-funktionale Aspekte zu berücksichtigen. Im Allgemeinen gehören beispielsweise die durch die Migration oder Ausführung auf dem anderen Gerät entstehenden Kosten sowie der Ladezustand der Energiequelle als auch die Leistung des Gerätes dazu. Im Interesse einer gesicherten, schnellen und günstigen Ausführung sollte der erste Wert möglicht niedrig und die anderen beiden Werte möglichst hoch sein. Ferner kann gefordert werden, dass Aktivitäten von bestimmten Geräten oder Personen beziehungsweise Rollen ausgeführt werden [Zap05]. Funktionale wie auch nicht-funktionale Aspekte können somit anwendungs- oder domänenabhängig sein und müssen daher dem Prozess explizit zugeordnet werden.

Die bisher betrachteten Kriterien sind sowohl bei der sequentiellen als auch bei der parallelen Ausführung von Prozessen zu berücksichtigen. Da an der parallelen Ausführung eines Prozesses gleichzeitig mehrere Geräte beteiligt sind, kommen hier neue Anforderungen hinzu, welche die Beziehung zwischen den Geräten beschreiben oder aber auf die Tatsache zurückzuführen sind, dass die parallelen Ausführungspfade in vielen Fällen untereinander koordiniert werden müssen.

Neben der direkten Zuordnung von Aktivitäten zu Rollen, Personen oder Geräten kann bei der parallelen Ausführung gefordert werden, dass Aktivitäten auf Geräten von derselben Person beziehungsweise Rolle oder gar auf demselben Gerät ausgeführt werden müssen, ohne sich dabei jedoch auf ein bestimmtes Gerät festlegen zu müssen. Dies ist beispielsweise sinnvoll, wenn die parallelen Ausführungspfade eng miteinander verknüpft sind und thematisch verwandte Aufgaben realisieren, die eine gewisse Lokalität aufweisen, aber dennoch parallel ausgeführt werden können. Auf der anderen Seite kann auch genau das Gegenteil gefordert werden, nämlich dass die parallele Ausführung auf unterschiedlichen Geräten erfolgen muss. Dies bewirkt, dass Aktivitäten echt-parallel ausgeführt werden, was in einigen Situationen für die korrekte Ausführung des Prozesses notwendig sein könnte. Wurden keine widersprechenden Anforderungen definiert, kann diese Strategie darüber hinaus insbesondere die Ausführung von rechenintensiven Aktivitäten beschleunigen, da im Gegensatz zur nebenläufigen Ausführung auf einem Gerät hier die Rechenlast zwischen mehreren Geräten aufgeteilt wird.

Weist ein auszuführender paralleler Pfad Abhängigkeitskonflikte auf, müssen die ausführenden Geräte mit diesen Konflikten umgehen können. Sie müssen daher die verwendeten Datenklassen – mindestens jedoch Serialisierbarkeit – kennen und die entsprechende Methode zur Auflösung von Abhängigkeitskonflikten bereitstellen. Da im Rahmen von Konfliktauflösungen die Kommunikation zwischen parallelen Ausführungpfaden notwendig ist, sollten die beteiligten Geräte ein gewisses Maß an Erreichbarkeit nicht unterschreiten. Dazu könnte beispielsweise verboten werden, parallele Pfade auf Geräte zu migrieren, deren Bewegungsvektoren in unterschiedliche Richtungen zeigen.

Ähnliches gilt ebenso für die explizite Synchronisation des Kontrollflusses, beispielsweise durch einen Join. In Abschnitt 4.5.2 wird eine zentrale und eine verteilte Synchro-

nisation vorgestellt. Während bei der zentralen Synchronisation die parallelen Ausführungspfade mit dem zentralen Koordinator kommunizieren können müssen, kann eine Einschränkung der Mobilität, wie sie oben vorgeschlagen wurde, die verteilte Synchronisation beschleunigen.

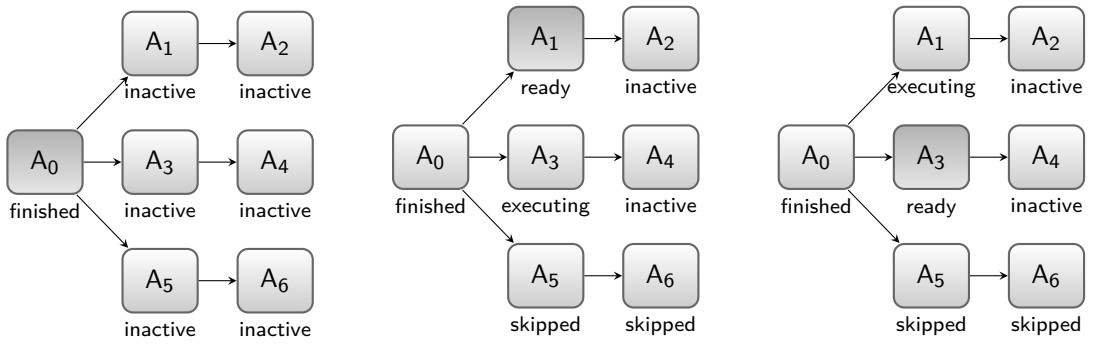
Auch während der parallelen Ausführung könnte im Hintergrund ständig Position und Erreichbarkeit der anderen parallelen Pfade geprüft werden, um beim Unterschreiten einer Grenze die Migration zurück in die Nähe der anderen Pfade zu erzwingen. Dabei würden erneut die diskutierten Anforderungen die Auswahl eines passenden Gerätes bestimmen.

4.4.2 Verteilungsmodell

Die an der parallelen Ausführung teilnehmenden Geräte müssen Zugriff auf verschiedene Prozessdaten haben. Da in mobilen Umgebungen keine dauerhaften Netzverbindungen garantiert werden können, wurde in diesem Konzept eine zentrale Datenhaltung ausgeschlossen. Parallele Ausführungspfade verfügen daher über ein vollständiges Replikat der von ihnen zur Ausführung benötigten Daten. In diesem Abschnitt wird geklärt, um welche Daten es sich dabei konkret handelt und wie diese den teilnehmenden Geräten bereitgestellt werden können.

Zunächst muss jedes einen parallelen Pfad bearbeitende Gerät das auszuführende Prozessmodell kennen. Es können dabei zwei grundlegende Strategien zur Verteilung unterschieden werden. Entweder speichert das Gerät das vollständige Prozessmodell oder es besitzt lediglich einen Ausschnitt, in dem der auszuführende parallele Pfad und alle konfligierenden Aktivitäten enthalten sind. Es ist offensichtlich, dass zur Verwaltung eines Ausschnittes weniger Speicherplatz erforderlich ist als für die Verwaltung des gesamten Prozesses. Allerdings muss berücksichtigt werden, dass bei einer Synchronisation des Kontrollflusses wieder das gesamte Prozessmodell benötigt wird, um die Ausführung fortzuführen. Da bei der verteilten Synchronisation a priori nicht bekannt ist welches Gerät die Synchronisation durchführt (vgl. Abschnitt 4.5.2), muss das gesamte Prozessmodell von mindestens einem an der parallelen Ausführung beteiligten Gerät gesichert und bei der Synchronisation zur Verfügung gestellt werden. Die Synchronisation kann aber erst dann beendet werden, nachdem diese Daten an das synchronisierende Gerät übertragen wurden. Insbesondere bei der Verwendung des *Cancelling Discriminators* können so Verzögerungen bei der Fortführung des Prozesses entstehen, da es hier nicht notwendig wäre, dass alle parallelen Ausführungspfade beendet sind.

Das Konzept sieht daher vor, dass jedes einen parallelen Pfad ausführende Gerät auf das gesamte Prozessmodell lokal zugreifen kann und daher eine vollständige Kopie besitzt. Dies hat neben dem Verzicht auf einen zentralen Verwalter den Vorteil, dass eine komplexe Analyse des Prozessmodells zur Bildung eines geeigneten Ausschnittes nicht erforderlich ist.



(a) Vor der Distribution (b) Neues Replikat $\mathfrak{R}_1$ (c) Neues Replikat $\mathfrak{R}_2$

Abbildung 4.17: Bei dieser Distribution des Kontrollflusses werden zwei Replikate erzeugt, da die Transitionsbedingung zum dritten parallelen Pfad negativ ist

Neben dem statischen Prozessmodell muss der dynamische Zustand des Prozesses repliziert werden. Da sich der Zustand während der Ausführung ändert, ist zu prüfen, ob die lokal ausgeführten Änderungen zu Widersprüchen führen können. Der Gesamtzustand eines Prozesses setzt sich aus dem Zustand der Aktivitäten in ihrem Lebenszyklus, den Werten der Prozessvariablen sowie der durch Konfliktauflösungen eingefügten Präzedenzen zusammen. Widersprüche durch lokale Änderungen an diesen Daten können nur bei Prozessvariablen auftreten, da der Zustand einer Aktivität nur im Rahmen der Ausführung dieser Aktivität geändert und die Widerspruchsfreiheit der zusätzlichen Präzedenzen bereits bei der Konfliktauflösung überprüft wird. Letztendlich verhindert eben diese Nebenläufigkeitskontrolle auch das Auftreten von Widersprüchen bei Prozessvariablen. Trotz des lokalen Zugriffs auf den replizierten Zustand kann somit die korrekte Ausführung des Prozesses ermöglicht werden.

Jedes Gerät, das einen parallelen Pfad ausführt, besitzt also nur eine lokale Sicht auf den Gesamtzustand des Prozesses. Dies kann genutzt werden um sicherzustellen, dass derselbe parallele Pfad nicht von mehreren Geräten ausgeführt wird. Dazu wird bei der Distribution des Kontrollflusses genau eine dem Split beziehungsweise Subprozessaufruf folgende Aktivität in den Zustand „*ready*“ versetzt, während die anderen Aktivitäten den Zustand „*executing*“ erhalten [Zap05]. Das Gerät unterlässt daher den Zugriff auf die anderen Aktivitäten und beginnt mit der Ausführung der für ihn bestimmten Aktivität (vgl. Abbildung 4.17). Für jede der Distribution direkt folgenden Aktivität, deren Transitionsbedingung zu „*wahr*“ auswertet, wird genau ein Replikat erzeugt, bei dem sich nur diese Aktivität im Zustand „*ready*“ befindet. Somit wird sichergestellt, dass alle parallelen Pfade ausgeführt werden und dabei aber keiner mehrfach bearbeitet wird.

In Prozessbeschreibungssprachen, welche die explizite Deklaration einer Start-Aktivität vorsehen, kann auf die eben vorgestellte indirekte Zuordnung des auszuführenden Pfades über die Zustandsinformation verzichtet werden. Eine Start-Aktivität gibt an, bei welcher Aktivität eine Ausführungsumgebung die Ausführung eines Prozesses fortführen muss, wenn diese unterbrochen wurde. Durch das Setzen der Start-Aktivität auf die erste Aktivität des entsprechenden parallelen Ausführungspfades kann nun ebenfalls

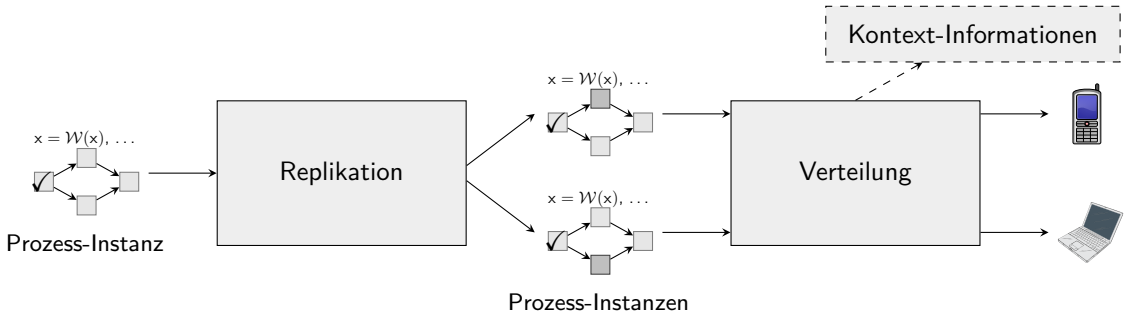


Abbildung 4.18: Ablauf der Distribution des Kontrollflusses

sichergestellt werden, dass keine Aktivitäten mehrfach bearbeitet werden. Unterstützt die Prozessbeschreibungssprache die Deklaration von Start-Aktivitäten, ist dessen Nutzung der Modifikation der Zustände vorzuziehen, da bei Letzterem eine falsche Sicht auf den Zustand des Prozesses vermittelt werden kann.

Zur Ausführung der Aktivitäten müssen die konkreten Werte der Prozessvariablen in jedem parallelen Ausführungspfad verfügbar sein. Hier muss nun unterschieden werden, ob die Distribution des Kontrollflusses durch einen *Split* oder einen asynchronen Subprozessaufruf hervorgerufen wurde. Wird ein Subprozess ohne gemeinsame Datenhaltung aufgerufen, so verfügt dieser nur über lokale Variablen, die folglich nicht repliziert werden müssen. Bei gemeinsamer Datenhaltung – insbesondere bei einem klassischen Split – müssen die Prozessvariablen jedoch repliziert werden. Da sich die Variablen durch die Ausführung von Aktivitäten verändern, kann bei der Distribution des Kontrollflusses lediglich der aktuelle Zustand repliziert werden. Die Nebenläufigkeitskontrolle (vgl. Abschnitt 4.3) verhindert, dass während der parallelen Ausführung lokal durchgeführte Änderungen zu Konflikten führen können.

Zusammenfassend müssen bei der Distribution des Kontrollflusses vier Elemente repliziert werden: das Prozessmodell, der Zustand im Lebenszyklus der Aktivitäten und des Prozesses, die Prozessvariablen sowie die zusätzlichen Präzedenzen, die im Rahmen der Auflösung von Abhängigkeitskonflikten im Prozessmodell gespeichert werden. Für jeden parallelen Pfad, dessen Transitionsbedingung zu „*wahr*“ auswertet, muss genau ein Replikat erzeugt werden, das nun auf ein anderes Gerät migriert werden kann. Nach der Erzeugung aller Replikate wird das ursprüngliche Replikat verworfen, da es nicht mehr benötigt wird.

Der gesamte Vorgang ist in Abbildung 4.18 dargestellt. Die Auswahl der Teilnehmer bei der Verteilung der Replikate kann mit Hilfe von Kontext-Informationen erfolgen, die aus gesammelten Daten über die Geräte in der Umgebung bestehen. Prinzipiell ist denkbar, dass die Distribution des Kontrollflusses als optionales Zusatzmodul einer Workflow-Engine realisiert wird. So können auch leistungsschwache Geräte die sequentiellen Abschnitte von parallelen Prozessen ausführen. Die Distribution des Kontrollflusses für parallele Abschnitte muss auf Geräten erfolgen, die über ausreichend dimensionierte Ressourcen und das entsprechende Zusatzmodul verfügen.

4.5 Koordination paralleler Pfade

Die Ausführung paralleler Pfade muss in bestimmten Situationen koordiniert werden. Dies ist offensichtlich der Fall, wenn im Kontrollfluss eine explizite Synchronisation – beispielsweise durch einen Join – modelliert wurde. Eine ähnliche Situation tritt auf, wenn implizit durch die Nebenläufigkeitskontrolle eine Datensynchronisation durchgeführt werden muss. In Abschnitt 4.3 wurde vorgeschlagen, dass implizite Synchronisationen explizit im Prozessmodell verankert werden. Aus diesem Grund können beide Fälle der Synchronisation gleich behandelt werden. Koordination ist aber auch notwendig, wenn parallele Pfade abgebrochen werden müssen. Dies kann passieren, wenn auf einem Ausführungspfad ein schwerwiegender Fehler auftritt oder die parallele Ausführung durch einen Cancelling Discriminator beendet wird.

Sowohl für die Synchronisation als auch für den Abbruch paralleler Ausführungspfade ist es notwendig, dass sie zum Zwecke der Koordination kommunikativ erreicht werden können. In Abschnitt 4.5.1 wird diese Problematik detaillierter behandelt und ein Lösungsansatz diskutiert. Weiterhin kann die Synchronisation des Kontrollflusses in zwei Teilprobleme unterteilt werden. Zunächst ist es notwendig, dass die auf unterschiedlichen Geräten ausgeführten parallelen Pfade auf ein Gerät zusammengeführt werden. In Abschnitt 4.5.2 werden dazu ein zentraler und ein verteilter Ansatz diskutiert. Anschließend müssen die Replikate der parallelen Ausführungspfade synchronisiert werden. Die dabei erforderliche Vorgehensweise wird in Abschnitt 4.5.3 erarbeitet.

Der Abbruch paralleler Ausführungspfade wird im Folgenden nicht weiter explizit diskutiert, da hierbei lediglich eine entsprechende Nachricht an das Gerät, das den abzubrechenden Pfad ausführt, zu übermitteln ist. Dies kann im Rahmen der allgemeinen Kommunikation zwischen parallelen Aktivitäten realisiert werden, die im folgenden Abschnitt diskutiert wird.

4.5.1 Kommunikation zwischen parallelen Aktivitäten

Da parallele Aktivitäten auf unterschiedlichen Geräten ausgeführt werden können, muss zur Koordination der Parallelität die Kommunikation zwischen den beteiligten Geräten ermöglicht werden. Gerade bei der Mobilkommunikation existiert jedoch aufgrund der Mobilität der Geräte und der potentiellen Ad-Hoc-Vernetzung eine sehr hohe Dynamik, die in Verbindung mit heterogenen Netzen die direkte Kommunikation zwischen mobilen Geräten häufig nicht ermöglicht (vgl. Abschnitt 2.1.2). Sogenannte *Overlay-Netze* kapseln konkrete Technologien und verbergen eine Vielzahl durch die Mobilität entstehende Probleme [DO03]. Im Folgenden wird vorausgesetzt, dass ein solches Netz verfügbar ist und damit die Erreichbarkeit von mobilen Geräten deutlich gesteigert wird. Dennoch kann auch der Einsatz eines Overlay-Netzes nicht verhindern, dass mobile Geräte zeitweise nicht erreichbar sind oder gänzlich aus dem Kommunikationsbereich verschwinden.

Bei der parallelen Ausführung von mobilen Prozessen tritt außerdem ein weiteres Problem auf. Aufgrund der Migrationsfähigkeit von mobilen Prozessen kann während der Ausführung eines parallelen Pfades das ausführende Gerät gewechselt werden. Andere parallele Pfade wissen in einem solchen Fall zunächst nicht, an welches Gerät sie nun ihre Nachrichten adressieren sollen. Dieses Problem, das sich in einer Abstraktionsstufe über den Overlay-Netzen befindet, wird in diesem Abschnitt diskutiert.

Migrationen können lediglich zwischen zwei Aktivitäten eines mobilen Prozesses durchgeführt werden und niemals während der Ausführung einer Aktivität (vgl. Abschnitt 2.3.3). Daher existiert eine eindeutige Zuordnung von Aktivitäten zu mobilen Geräten. Diese Zuordnung ist jedoch nur bei Aktivitäten bekannt, die bereits ausgeführt wurden oder zur Ausführung ausgewählt sind. Da der Prozess vor der Ausführung der Aktivität aber noch auf ein anderes Gerät migrieren kann oder bereits ausgeführte Aktivitäten im Rahmen der Fehlerbehandlung kompensiert und gegebenenfalls auf einem anderen Gerät erneut ausgeführt werden, kann sich die Zuordnung der Aktivitäten zu den Geräten im Laufe der Zeit ändern. Für Aktivitäten, die erst später zur Ausführung anstehen, existiert jedoch noch keine Zuordnung. An dieser Stelle kann lediglich die Annahme getroffen werden, dass eine Aktivität ohne Zuordnung von demselben Gerät ausgeführt wird wie die direkt vor ihr auszuführende Aktivtät.

Muss nun zwischen Aktivitäten eine Koordination erfolgen, können die ausführenden Geräte auf diese Zuordnung zurückgreifen, um Nachrichten an den richtigen Empfänger zu adressieren. Wie bereits angesprochen, ist diese Zuordnung jedoch dynamisch und wird von jedem beteiligten Gerät aufgrund von Migrationen angepasst. Konflikte wie bei den Variablen eines Prozesses können zwar nicht auftreten, da die Zuordnung einer Aktivität immer nur von maximal einem Gerät – nämlich jenes, welches die Aktivität ausführt – vorgenommen wird. Änderungen müssen aber trotzdem zeitnah für die anderen Geräte verfügbar sein, damit diese für eine Koordination die notwendigen Informationen nutzen können.

Eine solche Zuordnung von Aktivitäten zu Geräten muss daher im Rahmen des Prozessmanagements bereitgestellt werden. Existiert während der Ausführung eines parallelen Prozesses eine geeignete begleitende Infrastruktur, die genutzt werden kann, um den Ausführungsort einer Aktivität zu bestimmen, so müssen keine zusätzlichen Maßnahmen getroffen werden, um die Kommunikation zwischen parallelen Ausführungspfaden zu ermöglichen.

Kann auf eine derartige Infrastruktur nicht zugegriffen werden, könnte ein Gerät alternativ per Rundruf (*Broadcast*) eine Anfrage nach dem Ausführungsort einer bestimmten Aktivität versenden. Das Gerät, das diese Aktivität ausführt, antwortet daraufhin mit seiner Adresse. Für den Fall, dass das entsprechende Gerät temporär nicht erreichbar ist, muss die Anfrage regelmäßig wiederholt und die Ausführung des Prozesses gegebenenfalls blockiert werden. Diese Problematik wird in dieser Arbeit jedoch nicht weiter behandelt. Statt dessen wird vorausgesetzt, dass die Adressierung von parallelen Akti-

vitäten auf geeignete Weise umgesetzt ist und zur Kommunikation zwischen parallelen Ausführungspfaden verwendet werden kann.

4.5.2 Zusammenführen der parallelen Ausführungspfade

Bei der Synchronisation des Kontrollflusses werden mehrere parallele Ausführungspfade zusammengeführt, von denen nur einer fortgeführt wird. Dazu müssen die auf den einzelnen Replikaten durchgeführten Änderungen in ein gemeinsames Replikat integriert und gegebenenfalls Join-Bedingungen überprüft werden. Damit mehrere Replikate synchronisiert werden können, müssen sie auf einem Gerät verfügbar sein, das in der Lage ist die Synchronisation durchzuführen. Ein solches Gerät muss daher über ein mehrfaches an Speicherplatz verfügen, als für die Verwaltung eines einzelnen parallelen Ausführungspfades notwendig wäre. Um den tatsächlichen Bedarf zu senken, kann die Synchronisation von mehr als zwei Pfaden schrittweise erfolgen. Sobald zwei zu synchronisierende Replikate auf einem Gerät vorhanden sind, werden diese umgehend zu einem zusammengeführt und so sukzessive die Gesamtzahl der zu synchronisierenden Replikate reduziert.

Vor der eigentlichen Synchronisation muss jedoch gewartet werden bis die Ausführung aller beteiligten parallelen Pfade beendet ist. Dabei existieren verschiedene Möglichkeiten für das Verhalten eines Gerätes, auf dem die Ausführung eines Prozesses an einem Synchronisationspunkt angelangt ist. Da es das Ziel ist, die zu synchronisierenden parallelen Ausführungspfade auf einem Gerät zusammenzuführen, das zur Durchführung der Synchronisation in der Lage ist, werden bereits beendete parallele Ausführungspfade auf dieses Gerät migriert. Zur Auswahl des synchronisierenden Gerätes sind zwei Strategien denkbar. Bei der zentralen Synchronisation ist das synchronisierende Gerät bereits a priori festgelegt, während bei der verteilten Synchronisation prinzipiell alle dazu fähigen Geräte in Frage kommen, die Auswahl jedoch ad hoc anhand der tatsächlichen Verfügbarkeit der Geräte erfolgt und sich sogar während der Synchronisation ändern kann. Beide Strategien besitzen Vor- und Nachteile und werden im Folgenden genauer betrachtet.

Bei der zentralen Synchronisation erfolgt die Migration auf ein idealerweise stationäres Gerät, welches zuvor für diese Synchronisation ausgewählt wurde [KZL06]. Dies ist sinnvoll, wenn die parallelen Pfade auf Geräten ausgeführt werden, bei denen es ungewiss ist, ob sie aufgrund ihrer Mobilität jemals wieder in Kontakt treten können. Das Gerät sollte daher besonders zuverlässig und dabei möglichst immer erreichbar sein sowie ausreichend Ressourcen besitzen, um die Synchronisation durchführen zu können. Durch das Festlegen einer eindeutigen Adresse – beispielsweise durch Angabe eines *URI*[2] – kann zur Modellierungszeit, beim Starten eines Prozesses oder bei Beginn

2 Bei einem URI (*Uniform Ressource Identifier*) handelt es sich um eine standardisierte Zeichenfolge zur Identifizierung von abstrakten oder physikalischen Ressourcen [BLFM05]

der parallelen Ausführung innerhalb des Prozesses bestimmt werden, dass die zentrale Synchronisation angewendet und wo diese durchgeführt werden soll.

Unterbleibt hingegen die Festlegung eines solchen Ortes, wird die verteilte Synchronisation durchgeführt. Hierbei ist im Voraus noch nicht bekannt, welches Gerät die Synchronisation durchführt. Entscheidend ist daher, dass eine Strategie gewählt wird, die das Zusammenführen der parallelen Ausführungspfade ermöglicht. Dabei wird auf die in Abschnitt 4.5.1 beschriebene Zuordnung von Aktivitäten zu Geräten zurückgegriffen, um die zu synchronisierenden Geräte zu finden. Ist ein paralleler Ausführungspfad auf einem Gerät beendet, bestimmt seine Fähigkeit zur Synchronisation die nächste Aktion. In der folgenden Aufzählung sind die durchzuführenden Aktionen nach Prioritäten sortiert.

A) Das aktuelle Gerät ist in der Lage eine Synchronisation durchzuführen:

   1. Migration des Prozesses auf ein Gerät, das einen zu synchronisierenden parallelen Ausführungspfad dieses Prozesses beendet hat und auf die Migration der anderen Pfade wartet.
   2. Migration des Prozesses auf ein Gerät, das ebenfalls einen zu synchronisierenden parallelen Ausführungspfad dieses Prozesses ausführt und in der Lage ist, eine Synchronisation durchzuführen. Sobald der andere Pfad beendet ist, wird die Synchronisation durchgeführt.
   3. Warten bis andere zu synchronisierende parallele Pfade beendet wurden und auf das lokale Gerät migrieren. Die Synchronisation wird dann von dem lokalen Gerät selbst durchgeführt.

B) Das aktuelle Gerät ist nicht in der Lage eine Synchronisation durchzuführen:

   1. Migration des Prozesses auf ein Gerät, das einen zu synchronisierenden parallelen Ausführungspfad dieses Prozesses beendet hat und auf die Migration der anderen Pfade wartet.
   2. Migration des Prozesses auf ein Gerät, das ebenfalls einen zu synchronisierenden parallelen Ausführungspfad dieses Prozesses ausführt und in der Lage ist, eine Synchronisation durchzuführen. Sobald der andere Pfad beendet ist, wird die Synchronisation durchgeführt.
   3. Migration des Prozesses auf ein Gerät, das in der Lage ist, eine Synchronisation durchzuführen, aber bisher an der Ausführung des Prozesses nicht beteiligt ist. Das Gerät wechselt daraufhin in den Wartezustand, bis weitere beendete parallele Pfade erreichbar sind.
   4. Migration des Prozesses auf ein Gerät, das einen zu synchronisierenden parallelen Ausführungspfad dieses Prozesses ausführt und ebenfalls nicht in der Lage ist eine Synchronisation durchzuführen. Sobald ein Gerät erreichbar ist, das eine Synchronisation durchführen kann, werden beide Replikate auf dieses Gerät migriert und dort synchronisiert.

Jede dieser Aktionen führt einen Schritt in Richtung des Ziels durch, bei dem sich alle zu synchronisierenden parallelen Ausführungspfade auf einem einzigen zur Synchronisation fähigen Gerät befinden. Damit nicht mehrere Geräte darauf warten, dass andere Pfade zu ihnen migrieren (Punkt A3) und dabei eine Verklemmungssituation entsteht, muss ein wartendes Gerät die anderen an der Synchronisation beteiligten Geräte über seinen Wartezustand informieren. Ein Gerät beginnt nur dann zu warten, wenn noch kein anderes an dieser Synchronisation beteiligtes Gerät ebenfalls wartet. Da sich aufgrund der in unzuverlässigen Netzen unbestimmbaren Nachrichtenlaufzeit die Ereignisse überschneiden können, muss das Warten gegebenenfalls abgebrochen werden, falls eine Nachricht über den Beginn des Wartens eines anderen Gerätes empfangen wird.

Durch die Festlegung der Prioritäten der Aktionen gemäß der obigen Reihenfolge wird erreicht, dass zunächst versucht wird, zwei beendete parallele Ausführungspfade auf einem zur Synchronisation fähigem Gerät zusammenzuführen. Ist derzeit kein entsprechendes Gerät verfügbar, wird dennoch die Situation verbessert, indem der Prozess auf ein Gerät migriert wird, das besser geeignet ist als das derzeitige Gerät. Unter der Voraussetzung, dass Netzpartitionen und Geräteausfälle nicht dauerhaft bestehen, also im Laufe der Zeit – wenn auch nicht gleichzeitig – die Kommunikation zwischen den an der Synchronisation beteiligten Geräten möglich ist, wird durch diese Strategie die erfolgreiche Synchronisation garantiert. Befinden sich jedoch zwei an der parallelen Ausführung eines Prozesses beteiligten Geräte in unterschiedlichen Netzpartitionen, die dauerhaft getrennt bleiben, oder verliert ein Gerät fortwährend jeglichen Kontakt – etwa weil es ausgeschaltet ist – so kann die Synchronisation nicht ermöglicht werden. Eine derartige Situation kann nur durch eine Zeitüberschreitung (*Timeout*) erkannt werden. Bei der Betrachtung von Zeitüberschreitungen kann jedoch nicht unterschieden werden, ob ein Prozess nicht erfolgreich beendet werden kann oder ob ein Gerät lediglich sehr lange nicht verfügbar ist. Dies wäre jedoch auch bei klassischen verteilten Systemen der Fall und wird daher im Weiteren nicht problematisiert.

4.5.3 Synchronisation der Replikate

Sind auf einem Gerät mindestens zwei zu synchronisierende parallele Ausführungspfade vorhanden und beendet, kann die eigentliche Synchronisation der Replikate durchgeführt werden. Dabei müssen alle auf den parallelen Ausführungspfaden veränderten Werte in ein gemeinsames neues Replikat zusammengeführt werden. Die veränderbaren Werte können in die drei Kategorien Prozessvariablen, Zustände von Aktivitäten sowie im Rahmen der Nebenläufigkeitskontrolle hinzugefügte Präzedenzen eingeteilt werden. Im Folgenden wird beschrieben, wie die Datensynchronisation in jeder dieser Kategorien erfolgt. Dabei wird ohne Beschränkung der Allgemeinheit angenommen, dass genau zwei parallele Ausführungspfade synchronisiert werden.

Zusammenführen der Präzedenzen

Während der Ausführung der parallelen Pfade kann auf diesen eine Auflösung von Abhängigkeitskonflikten mit anderen Pfaden durchgeführt worden sein. Bei der Konfliktauflösung entstehen neue Präzedenzen, die zur Berechnung von möglichen Serialisierungen berücksichtigt werden müssen. Da Abhängigkeitskonflikte mit verschiedenen Pfaden auftreten können, ist die Menge der aufgrund der Konfliktauflösung zunächst lokal eingefügten zusätzlichen Präzendenzen bei den zu synchronisierenden Pfaden nicht notwendigerweise gleich. Es werden daher die beiden aus den zu synchronisierenden Pfaden stammenden Mengen von Präzedenzen unter Vernachlässigung der Duplikate vereinigt, um so den globalen Abhängigkeitsgraphen unter Berücksichtigung der lokalen Konfliktauflösungen konstruieren zu können.

In Abschnitt 4.3.1 wurde gezeigt, dass durch das Einfügen dieser Präzedenzen keine zusätzlichen Zyklen im Abhängigkeitsgraphen entstehen. Da während der parallelen Ausführung jeder lokale Zyklus im Abhängigkeitsgraph durch die Konfliktauflösung beseitigt wurde oder aufgrund der verwendeten Datenklassen generell nicht berücksichtigt wird, weist die mit dieser Synchronisation beendete parallele Ausführung keinen Zyklus im Abhängigkeitsgraph auf. Aus ihm kann daher eine serielle Ausführungsfolge abgelesen werden, die für die Synchronisation der Prozessvariablen notwendig ist.

Aktualisierung der Zustände der Aktivitäten

Um die Verwaltung von mobilen Prozessen zu erleichtern, befinden sich die Aktivitäten immer in einem wohldefinierten Zustand (vgl. Abschnitt 2.3.3). Dieser wird gegebenenfalls bei der Unterdrückung von toten Pfaden und im Rahmen der Ausführung der Aktivität angepasst. Zu synchronisierende Pfade weisen daher nur Änderungen an den von ihnen verarbeiteten Aktivitäten auf. Die Zustände können somit ohne Widersprüche in das neue Replikat übernommen werden. In der resultierenden Prozessbeschreibung befinden sich die Aktivitäten von mehreren parallelen Ausführungspfaden im Zustand „*finished*“ beziehungsweise „*skipped*“ oder „*expired*“. Der tatsächliche Zustand der Aktivitäten von noch nicht synchronisierten Pfaden ist jedoch weiterhin unbekannt und verbleibt in der lokalen Sichtweise in „*inactive*“.

Abbildung 4.19 zeigt die Synchronisation von zwei Replikaten. Links wird die in dem Replikat $\mathfrak{R}_1$ materialisierte Sichtweise des Gerätes gezeigt, das die Aktivitäten A_5 und A_6 ausgeführt hat. A_5 und A_6 befinden sich daher im Zustand „*finished*“, während alle restlichen Aktivitäten im Zustand „*inactive*“ verweilen. Dieses Replikat wird mit dem in der Mitte dargestellten Replikat $\mathfrak{R}_2$ synchronisiert. Hier ist bereits eine Synchronisation zwischen zwei parallelen Pfaden durchgeführt worden und somit befinden sich die Aktivitäten A_1 bis A_4 in den Zuständen „*finished*“ beziehungsweise „*skipped*“. In dieser Sicht besitzen die Aktivitäten A_5 und A_6 den Zustand „*inactive*“, da noch nicht bekannt ist, dass diese Aktivitäten auf dem Replikat $\mathfrak{R}_1$ bereits erfolgreich ausgeführt wurden.

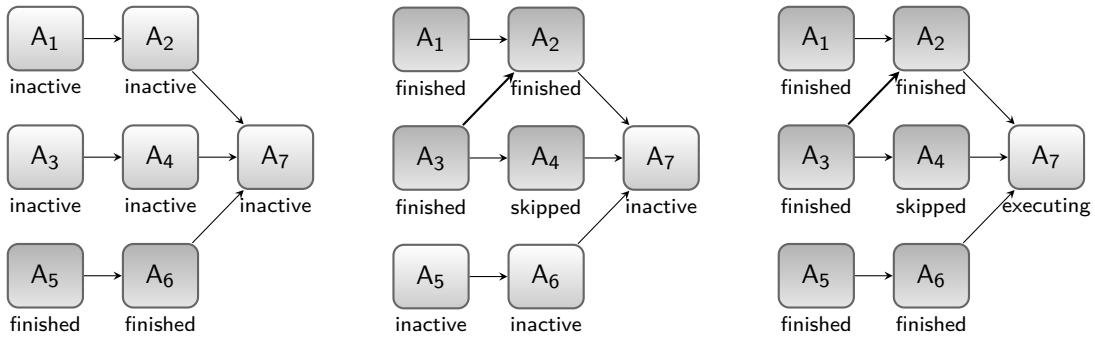


Abbildung 4.19: Die zwei von unterschiedlichen Geräten stammenden Replikate $\mathfrak{R}_1$ und $\mathfrak{R}_2$ werden zu einem Replikat $\mathfrak{R}_{1\cup 2}$ zusammengeführt

Weiterhin wurde im Rahmen der Nebenläufigkeitskontrolle in Replikat $\mathfrak{R}_2$ die zusätzliche Präzedenz $A_3 \lessdot A_2$ eingefügt. Bei der Synchronisation entsteht das rechts dargestellte Replikat $\mathfrak{R}_{1\cup 2}$. Hier sind alle Zustände zusammengefasst und auch die zusätzlichen Präzedenzen wurden übernommen. Da alle an der Synchronisation teilnehmenden parallelen Ausführungspfade beendet sind, kann nun die Aktivität A_7 ausgeführt werden.

Synchronisation der Prozessvariablen

Bei der Synchronisation der Prozessvariablen muss unterschieden werden, zu welcher Datenklasse die konkrete Variable gehört. Während bei Variablen, die Serialisierbarkeit erfordern, zunächst eine äquivalente serielle Ausführungsfolge gebildet werden muss, ist dies bei anderen Datenklassen nicht erforderlich. Nach dem Zusammenführen der Präzedenzen wird daher ein neuer Abhängigkeitsgraph berechnet, der lediglich Lese-Schreib-Abhängigkeiten berücksichtigt, die durch Zugriffe auf Variablen entstehen, die Serialisierbarkeit erfordern. Aus dem Abhängigkeitsgraph lassen sich mögliche Serialisierungen für die zu synchronisierenden parallelen Ausführungspfade ableiten. Die Synchronisation der Prozessvariablen muss in Abhängigkeit von einer gültigen Serialisierung erfolgen, damit die Ausführung des Prozesses serialisierbar ist. Dazu werden alle Aktivitäten betrachtet, die in den zu synchronisierenden Pfaden beendet wurden und sich somit in dem Zustand *„finished“* befinden. Aktivitäten, die sich nicht in diesem Zustand befinden, wurden entweder ausgelassen (aufgrund von Fehlern oder negativen Transitionsbedingungen) oder sie werden von anderen parallelen Pfaden beziehungsweise erst nach der Synchronisation ausgeführt. In jedem Fall ist ihre Position in einer äquivalenten Serialisierung für diese Synchronisation der Variablen unbedeutend.

Lassen sich aus dem Abhängigkeitsgraph mehrere Serialisierungen ableiten, darf unter ihnen eine ausgewählt werden. Es kann nun vermutet werden, dass die Wahl einer Serialisierung spätere Synchronisationen negativ beeinflusst. Dies ist jedoch nicht der Fall, da lediglich aus den Serialisierungen gewählt werden darf, die aus dem Abhängigkeitsgraphen ableitbar sind. Äquivalente Serialisierungen werden ausschließlich

von den Präzedenzen des Prozessmodells sowie von Lese-Schreib-Abhängigkeiten bestimmt (vgl. Abschnitt 4.2). Da der Abhängigkeitsgraph alle potentiell auftretenden Lese-Schreib-Abhängigkeiten berücksichtigt und – im Rahmen der Konfliktauflösung – nur Präzedenzen hinzukommen, die zuvor bereits indirekt durch Lese-Schreib-Abhängigkeiten gegeben waren, können nach der Wahl einer Serialisierung keine ihr widersprechenden Abhängigkeiten hinzukommen. Wenn durch die Konfliktauflösung oder durch selektive Pfade Lese-Schreib-Abhängigkeiten entfallen, bleibt eine gewählte Serialisierung weiterhin gültig, da der Wegfall von Abhängigkeiten lediglich zusätzlichen Freiheitsgraden entspricht.

Wurde eine Serialisierung gewählt, müssen die zu verwendenden Variablenwerte bestimmt werden. Dazu muss für jede Variable die Aktivität gefunden werden, welche die Variable zuletzt geändert hat. Für diese Variable wird nun der Wert des Replikates verwendet, in dem die gefundene Aktivität ausgeführt wurde. Falls die Variable bisher nicht verändert wurde, ist ihr Wert in allen Replikaten gleich; er entspricht dem nun zu verwendenden Initialisierungwert.

Abschluss der Synchronisation

Die Synchronisierung der Replikate erfolgt paarweise bis alle zu synchronisierenden parallelen Ausführungspfade berücksichtigt wurden. Es existiert nun lediglich ein Replikat auf einem einzigen Gerät. Dieses muss nun gegebenenfalls die Join-Bedingung prüfen. Wertet diese zu „*falsch*“ aus, muss der Zustand der synchronisierenden Aktivität auf „*skipped*“ gesetzt werden, und die Unterdrückung von toten Pfaden (vgl. Abschnitt 2.2.6) wird gestartet. Ist die Auswertung der Join-Bedingung erfolgreich oder ist keine Join-Bedingung angegeben, kann die auf die Synchronisation folgende Aktivität gestartet werden oder der synchronisierte Prozess auf ein anderes Gerät migrieren.

4.6 Optimistische Konfliktauflösung

Erfolgt die Konfliktauflösung zur Sicherstellung der Eine-Kopie-Serialisierbarkeit ad hoc zur Laufzeit – wie in Abschnitt 4.3.1 vorgeschlagen –, müssen sich die an dem Abhängigkeitskonflikt beteiligten Geräte einig sein, welche zusätzliche Präzedenz $A_S \ll A_L$ zur Auflösung des Zyklus eingefügt wird. Ist zu diesem Zeitpunkt keine Kommunikation zwischen den Geräten möglich, kann die Koordination nicht umgehend erfolgen; es muss gewartet werden, bis die Netzverbindungen wiederhergestellt sind. Das bereits in Abschnitt 4.3.1 problematisierte Szenario kann durch die in diesem Abschnitt vorgestellte optimistische Ausführung verbessert werden. Anstatt die notwendige Koordination sofort durchzuführen, wird – optimistisch – angenommen, dass der Kontrollfluss auf den anderen Geräten noch nicht den Abhängigkeitskonflikt erreicht hat. Weiterhin wird angenommen, dass rechtzeitig die Kommunikation mit den anderen Geräten möglich ist, bevor diese den Konflikt erreichen.

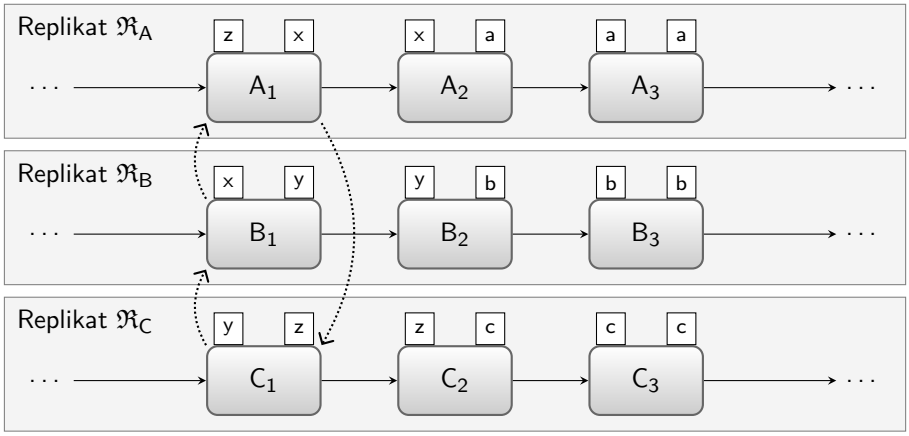
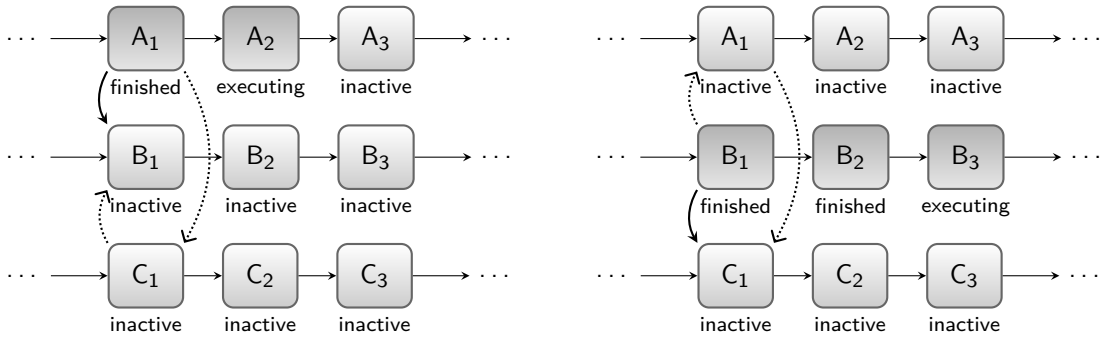


Abbildung 4.20: Drei parallele Ausführungspfade weisen einen Abhängigkeitskonflikt auf

Unter der Bedingung, dass diese Voraussetzungen gegeben sind, kann lokal eine Konfliktlösung gewählt werden, bei der die durchzuführende Synchronisation auf einem anderen parallelen Ausführungspfad erfolgt. Sobald eine an einem Abhängigkeitskonflikt $\mathfrak{K}$ beteiligte Aktivität A_S ausgeführt werden soll, wird daher die Konfliktauflösung mit der Lese-Schreib-Abhängigkeit $A_L \prec A_S \in \mathfrak{K}$ durchgeführt. Gemäß der Konfliktauflösung (vgl. Abschnitt 4.3.1) muss nun die Präzedenz $A_S \lessdot A_L$ eingefügt werden. Das Gerät kann so mit der Ausführung der nächsten Aktivität beginnen, während es gleichzeitig die restlichen an dem Abhängigkeitskonflikt beteiligten Geräte über die Konfliktauflösung informiert. Aufgrund der fehlerhaften Netzverbindung kann sich der Empfang der Nachricht jedoch verzögern. Da die anderen Geräte gemäß der Annahme beim Wiederherstellen der Netzverbindung den Abhängigkeitskonflikt noch nicht erreicht haben, können sie bei Erhalt der Nachricht problemlos zustimmen und die zusätzliche Präzedenz in ihrem lokalen Replikat vermerken. Nach der Ausführung der Aktivität A_S müssen die Prozessdaten wie bei einem Split repliziert werden, damit sie bei der nun neu auftretenden Synchronisation an der Aktivität A_L verfügbar sind.

Im Allgemeinen können die Annahmen jedoch nicht garantiert werden. Es kann daher passieren, dass mehrere Geräte gleichzeitig mit der optimistischen Konfliktauflösung beginnen und erst später die entsprechende Nachricht eines anderen Gerätes erhalten. Zu diesem Zeitpunkt könnten bereits mehrere Aktivitäten ausgeführt worden sein, die allerdings gemäß der von dem anderen Gerät eingefügten Präzedenzen gar nicht hätten gestartet werden dürfen.

So kann im Beispiel von Abbildung 4.20 auf dem Replikat $\mathfrak{R}_A$ die Aktivierung von Aktivität A_1 anstehen. Da derzeit keine Kommunikation zu $\mathfrak{R}_B$ und $\mathfrak{R}_C$ möglich ist, wird A_1 optimistisch ausgeführt und anschließend die Ausführung von A_2 gestartet. In der Zwischenzeit hat der Kontrollfluss auf dem Replikat $\mathfrak{R}_B$ die Aktivität B_1 erreicht. Da immer noch keine Kommunikation möglich ist, startet auch auf $\mathfrak{R}_B$ die optimistische Phase und es beginnt die Ausführung von B_1. Als $\mathfrak{R}_B$ die Nachricht über die auf $\mathfrak{R}_A$ gewählte Konfliktauflösung erreicht, wurden bereits die Aktivitäten B_1 und B_2 beendet



(a) Replikat $\mathfrak{R}_A$ nach lokaler Konfliktauflösung (b) Replikat $\mathfrak{R}_B$ nach lokaler Konfliktauflösung

Abbildung 4.21: Lokale Zustände des Prozesses von Abbildung 4.20, der aufgrund eines Netzausfalls optimistisch ausgeführt wird

sowie die Aktivität B_3 gestartet: Es entsteht das in Abbildung 4.21 dargestellte Szenario. Die gewählten Strategien zur Konfliktauflösung lassen sich hier nicht vereinen, da die entstandenen Präzedenzen $A_1 \lessdot B_1 \lessdot C_1$ im Widerspruch zu der bereits getätigten Ausführung von B_1 stehen, die ohne Berücksichtigung der von A_1 geschriebenen Werte erfolgte.

In einem solchen Fall müssen die durchgeführten Änderungen rückgängig gemacht werden, um unter Berücksichtigung des modifizierten Prozessmodells die Aktivitäten erneut zu starten. Da die Ausführung von Aktivitäten auch Auswirkungen außerhalb des Prozesses haben kann, reicht es nicht aus, den Prozess auf den Zustand vor der optimistischen Ausführung zurückzusetzen. Diese Problematik wurde bereits im Rahmen der Fehlerbehandlung in Abschnitt 2.2.6 diskutiert und kann durch die Ausführung von zusätzlichen kompensierenden Aktivitäten aufgelöst werden. Die optimistische Ausführung ist daher nur möglich, wenn die auszuführenden Aktivitäten kompensierbar und wiederholbar sind.

Vor dem Start der optimistischen Ausführung muss daher zunächst geprüft werden, ob die folgende Aktivität kompensierbar ist. Weiterhin muss der aktuelle Zustand des Replikates gesichert und auch bei der Migration auf ein anderes Gerät weitergegeben werden. Der Kontrollfluss kann nun solange optimistisch ausgeführt werden wie die auszuführenden Aktivitäten kompensierbar sind. Trifft der Kontrollfluss auf eine Aktivität, die nicht kompensierbar ist, muss die Ausführung angehalten werden bis eine Bestätigung oder eine Ablehnung über die lokale Wahl der Konfliktauflösung von allen an dem Abhängigkeitskonflikt beteiligten Geräte empfangen wird.

Trifft der Kontrollfluss während der optimistischen Ausführung auf einen weiteren Abhängigkeitskonflikt, so ist prinzipiell denkbar, dass eine verschachtelte optimistische Ausführung erfolgt. Dieser Fall wird in dieser Arbeit jedoch nicht weiter betrachtet, da er die Situation deutlich verkompliziert. So müssten beispielsweise die Geräte in der Lage sein, mehrere Zustände zu verwalten, auf die sie im Falle eines Abbruchs zurücksetzen können. Außerdem müsste bei der Mitteilung der lokalen Konfliktauflösung an die anderen Geräte vermerkt werden, dass diese Entscheidung auf der Grundlage von

optimistischen Annahmen getroffen wurde. Wenn die erste Konfliktauflösung abgebrochen werden muss, hat auch jede später getroffene Entscheidung zu einer Konfliktauflösung keinen Bestand mehr und die anderen von dem späteren Abhängigkeitskonflikt betroffenen parallelen Ausführungspfade müssen hierüber in Kenntniss gesetzt werden. Dies kann wiederum dazu führen, dass diese ebenfalls einen Abbruch durchführen und bereits ausgeführte Aktivitäten kompensieren müssen.

Wird also während einer optimistischen Ausführung ein weiterer Abhängigkeitskonflikt, eine Verzweigung des Kontrollflusses oder eine Synchronisation erreicht, wird die Ausführung dieses Pfades pausiert bis entschieden ist, ob die Annahmen der optimistischen Ausführung sich bestätigt haben oder ein Abbruch erfolgen muss.

Sind zwei parallel gestartete optimistische Ausführungen nicht miteinander vereinbar, brauchen jedoch nicht alle abgebrochen und kompensiert werden. Um die serialisierbare Ausführung zu ermöglichen, müssen zwei Eigenschaften erfüllt sein. Zum Einen darf durch die eingefügten Präzedenzen kein Zyklus auftreten. Ein Zyklus entsteht, wenn jedes an dem Abhängigkeitskonflikt beteiligte Gerät eine Konfliktauflösung nach dem oben vorgestellten Prinzip durchführt. Im Prozess von Abbildung 4.20 würden dann statt der Lese-Schreib-Abhängigkeiten $C_1 \prec B_1 \prec A_1 \prec C_1$ die Präzedenzen $A_1 \preccurlyeq B_1 \preccurlyeq C_1 \preccurlyeq A_1$ auftreten und einen Zyklus bilden. Zum Anderen dürfen die eingefügten Präzedenzen nicht im Widerspruch zu dem bei der optimistischen Ausführung entstandenen Datenfluss stehen. Daher müssen durch die neuen Präzedenzen entstandene Synchronisationen zwingend eingehalten werden, damit nicht eine Situation wie in Abbildung 4.21 entsteht.

Ist eine der beiden Eigenschaften nicht erfüllt, müssen optimistisch begonnene Ausführungen abgebrochen und kompensiert werden bis beide Eigenschaften erfüllt sind. Da es verschiedene Möglichkeiten zur Auswahl der abzubrechenden Pfade geben kann, wäre es vorteilhaft, wenn dabei nicht nur die Anzahl der abzubrechenden Aktivitäten minimiert wird sondern auch der für die Ausführung der Aktivitäten benötigte Aufwand berücksichtigt wird. Davidson hat jedoch gezeigt, dass alleine schon der Aufwand für die Minimierung der Anzahl der abzubrechenden Transaktionen im Allgemeinen NP-vollständig ist [Dav84]. Da in diesem Konzept jedoch keine verschachtelte optimistische Ausführung stattfinden kann, wird die Komplexität deutlich eingeschränkt. Im Falle des notwendigen Abbruchs von optimistischen Ausführungen kann somit mit relativ geringem Aufwand die Ausführung mit der geringsten Komplexität ausgewählt werden.

4.6.1 Algorithmus zur Auflösung von Abhängigkeitskonflikten

Der in diesem Abschnitt vorgestellte Algorithmus realisiert die optimistische Konfliktauflösung mit den zuvor gestellten Anforderungen. Zur Vereinfachung werden drei Mengen unterschieden, welche die Präzedenzen eines Prozesses beinhalten.

Definition 4.8 (Mengen zur Verwaltung von Präzedenzen) *$\mathcal{P}$ sei die Menge der aus dem Prozessmodell hervorgehenden Präzedenzen. Diese Menge bleibt während der Ausführung des Prozesses konstant. Bei einer Konfliktauflösung hinzugefügte Präzedenzen werden in die Menge $\mathcal{P}_Z$ aufgenommen, sobald die Auflösung endgültig ist. Beginnt ein paralleler Ausführungspfad eine optimistische Ausführung, wird in $\mathcal{O}$ die dabei vorläufig hinzugefügte Präzedenz gespeichert. Die Menge $\mathcal{P}_O$ beinhaltet hingegen die von anderen parallelen Ausführungspfaden optimistisch hinzugefügten Präzedenzen. Im Falle des Abbruchs einer optimistischen Ausführung werden hier Präzedenzen wieder entfernt.*

Der Kontrollfluss folgt lediglich den Präzedenzen aus $\mathcal{P}$ sowie aus $\mathcal{P}_Z$ und während der optimistischen Ausführung ebenso der Präzendenz in $\mathcal{O}$. Die Konflikterkennung mit dem Abhängigkeitsgraph wird gleichfalls nur aus diesen Präzedenzen berechnet. Erreicht der Kontrollfluss während der optimistischen Ausführung einen Split, Join oder Subprozess-Aufruf, so wird der Prozess pausiert bis die optimistische Ausführung beendet ist. Dies macht komplexe Speicherstrukturen unnötig, die gebraucht würden, um mehrere bereits synchronisierte Ausführungspfade zurückzusetzen. Ebenfalls pausiert werden muss die optimistische Ausführung, wenn eine Aktivität erreicht wird, die nicht kompensierbar und wiederholbar ist.

Die optimistische Konfliktauflösung wird mit Hilfe verschiedener Nachrichten koordiniert. Beim Eintritt in die optimistische Ausführung wird die Nachricht $\textsc{request}(A_S \lessdot A_L)$ an alle an dem aufzulösenden Abhängigkeitskonflikt beteiligten Aktivitäten gesendet. Diese Nachricht stellt eine Anfrage dar, ob die gewählte Konfliktauflösung in Übereinstimmung mit den optimistischen Ausführungen ist, die von den parallelen Ausführungspfaden möglicherweise ebenso begonnen wurden. Diese antworten entweder $\textsc{ack}(A_S \lessdot A_L)$ als Bestätigung oder mit $\textsc{nack}_\omega(A_S \lessdot A_L, B_S \lessdot B_L)$ als Widerspruch. Im zweiten Fall wird mit Hilfe eines Gewichts ω, das die Komplexität der bereits getätigten optimistischen Ausführung angibt, bestimmt, welche von den im Parameter angegebenen sich widersprechenden optimistischen Ausführungen abgebrochen werden soll. Diese Strategie minimiert den für die Wiederholung der Ausführung notwendigen Aufwand. Im Falle des Abbruchs wird die Nachricht $\textsc{cancel}(A_S \lessdot A_L)$ an alle an dem Abhängigkeitskonflikt beteiligten Aktivitäten gesendet. Nur wenn alle Bestätigungen eintreffen, wird mit der Nachricht $\textsc{apply}(A_S \lessdot A_L)$ die gewählte Präzedenz endgültig hinzugefügt (vgl. Ablauf in Abbildung 4.22).

Als Voraussetzung für die Übertragung der Nachrichten werden dabei verschiedene Annahmen über das unterliegende Netz getroffen. Aufgrund der unzuverlässigen Verbindungen kann zwar nicht vorausgesetzt werden, dass die Nachrichten innerhalb eines bestimmten Zeitraumes ankommen. Dennoch ist es notwendig, dass alle gesendeten Nachrichten irgendwann ankommen und somit nicht verloren gehen. Darüber hinaus müssen die von einem Gerät gesendeten Nachrichten in derselben Reihenfolge ankommen, wie sie gesendet wurden (*single-source FIFO*). Während die erste Anforderung sehr leicht durch Empfangsbestätigungen sichergestellt werden kann, ist für die

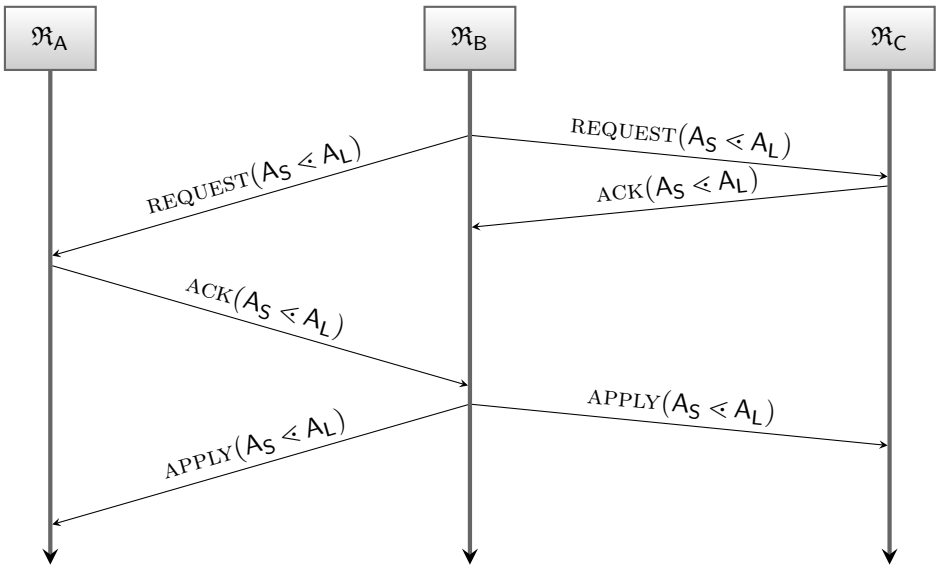


Abbildung 4.22: Übertragene Nachrichten bei einem Abhängigkeitskonflikt zwischen drei Replikaten. $\mathfrak{R}_B$ leitet die optimistische Konfliktauflösung ein, welche von allen Teilnehmern bestätigt wird (zwischen $\mathfrak{R}_A$ und $\mathfrak{R}_B$ besteht eine langsame Verbindung).

zweite Anforderung die Verwendung von Sequenznummern in den Nachrichten notwendig. Die Anforderungen sind somit relativ einfach zu realisieren und können damit als gegeben vorausgesetzt werden.

Adressiert werden die Nachrichten – wie auch bei der Koordination paralleler Pfade in Abschnitt 4.5.1 – an Aktivitäten innerhalb eines bestimmten Prozesses. Eine entsprechende Infrastruktur muss daher auch hier die Geräte finden, die diese Aktivitäten ausführen, voraussichtlich ausführen werden oder im Rahmen der optimistischen Ausführung bereits ausgeführt haben.

In Algorithmus 4.1 ist dargestellt wie sich die Ausführungsumgebung verhält, sobald gemäß dem Kontrollfluss eine an einem Abhängigkeitskonflikt $\mathfrak{K}$ beteiligte Aktivität aktiviert werden soll. In einem solchen Fall existiert immer eine Aktivität A_L, die gemäß einer Lese-Schreib-Abhängigkeit in einer Serialisierung vor dieser Aktivität ausgeführt werden soll. Würde es eine solche Lese-Schreib-Abhängigkeit nicht geben, so müsste eine Aktivität A_i existieren, für die $A_L \prec A_i$ und $A_i < A_S$ gilt, wobei A_S die aktuelle Aktivität bezeichnet. Dann wäre der Kontrollfluss allerdings bereits bei der Aktivierung der Aktivität A_i auf diesen Abhängigkeitskonflikt gestoßen und hätte ihn auflösen müssen. Die aktuelle Aktivität wird daher mit A_S bezeichnet und die Lese-Schreib-Abhängigkeit lautet ohne Beschränkung der Allgemeinheit $A_L \prec A_S$.

Die optimistische Konfliktauflösung wird nur gestartet, wenn noch keine optimistische Ausführung aktiv ist (keine verschachtelte Ausführung) und wenn kein anderer paralleler Ausführungspfad mit der optimistischen Ausführung begonnen hat, bei der mit hoher Wahrscheinlichkeit bereits die Aktivität ausgeführt wurde, die bei der zu startenden Konfliktauflösung eine Synchronisation durchführen müsste (A_L). Letzteres ver-

Algorithmus 4.1 Aktivierung einer an einem Konflikt $\mathfrak{K}$ beteiligten Aktivität A_S

```
1: A_L ≺ A_S ← LSA(𝔎, A_S)                    ▷ Lese-Schreib-Abhängigkeit (s. o.)
2: if optimistische Ausführung aktiv then
3:     suspend                                  ▷ Verschachtelung verhindern
4: else if ∃A_i : A_L ⋖ A_i ∈ P_O then
5:     suspend                                  ▷ Abbruch vermeiden
6: else
7:     Beginne optimistische Ausführung
8:     O ← A_S ⋖ A_L
9:     A ← konfliktPfade(𝔎, A_S)
10:    for all A_i ∈ A do
11:        send REQUEST(A_S ⋖ A_L) to A_i
12:    end for
13: end if
```

meidet den fast sicheren Abbruch der optimistischen Ausführung, der notwendig wird, wenn die optimistische Ausführung des parallelen Pfades erfolgreich beendet wird.

Bei dem Start der optimistischen Ausführung muss eine Kopie des aktuellen Zustands des ausführenden Replikats erzeugt und gesichert werden, um im Falle des Abbruchs den alten Zustand wiederherstellen zu können. Anschließend werden alle parallelen Ausführungspfade mit konfligierenden Aktivitäten über die gewählte Konfliktauflösung informiert. Aktivitäten konfligieren, wenn sie an demselben Abhängigkeitskonflikt beteiligt sind. Da bedingt durch direkte Präzedenzen auch mehrere Aktivitäten eines parallelen Ausführungspfades betroffen sein können, reicht es aus, die Nachricht an die jeweils erste Aktivität zu senden.

Ist die Aktivität A_S nach dem Start der optimistischen Konfliktauflösung an keinem weiteren Abhängigkeitskonflikt beteiligt, kann sie ausgeführt werden, falls sie kompensierbar und wiederholbar ist. Nach ihrer Ausführung muss gemäß der zusätzlichen Präzedenz $A_S \lessdot A_L$ eine Distribution des Kontrollflusses erfolgen. Dabei wird, wie in Abschnitt 4.4.2 beschrieben, ein Replikat erzeugt. Dieses wird jedoch während der optimistischen Ausführung nicht gestartet oder auf ein anderes Gerät migriert. Erst wenn die optimistische Konfliktauflösung endgültig ist, wird das Replikat reaktiviert und kann zur Synchronisation auf das entsprechende Gerät migrieren. Im Falle eines Abbruchs wird es jedoch verworfen.

Algorithmus 4.2 zeigt das Verhalten bei dem Empfang der Nachricht REQUEST($A_S \lessdot A_L$), die beim Start der optimistischen Ausführung einer parallelen Aktivität versendet wurde. Hier wird zunächst unterschieden, ob sich das Gerät ebenfalls in einer optimistischen Ausführung befindet oder nicht. Ist dies der Fall, so wird geprüft, ob ein Zyklus entsteht, wenn die lokale sowie die in der Nachricht angegebene optimistisch hinzugefügten Präzedenzen verwendet würden. Trifft dies zu, so muss eine der beiden optimistischen Ausführungen abgebrochen werden. Daher wird zunächst der Aufwand ω berechnet, der bei der bisherigen optimistischen Ausführung notwendig war und im

Falle des Abbruches voraussichtlich in ähnlicher Höhe notwendig sein wird. Prinzipiell sind verschiedene Maße denkbar, an dieser Stelle wird jedoch die Summe des Zeitaufwandes der Ausführung der Aktivitäten verwendet. Da sich dieses Maß während der Ausführung ändert, muss vermerkt werden, bei welcher optimistischen Konfliktauflösung welcher Wert angegeben wurde (Ω). Nur so kann später der lokale Aufwand mit dem des anderen parallelen Pfades verglichen sowie eindeutig bestimmt werden, welche optimistische Ausführung abgebrochen werden muss.

Algorithmus 4.2 Empfang der Nachricht REQUEST$(A_S \lessdot A_L)$ von A_S

```
1: if optimistische Ausführung aktiv then
2:     if besitztZyklus(P ∪ P_Z ∪ O ∪ A_S ≪ A_L) then
3:         Berechne Aufwand ω
4:         Ω(A_S ≪ A_L) ← ω
5:         send NACK_ω(A_S ≪ A_L, O) to A_S
6:     else if A_L ≠ inactive then
7:         Berechne Aufwand ω
8:         Ω(A_S ≪ A_L) ← ω
9:         send NACK_ω(A_S ≪ A_L, O) to A_S
10:    else
11:        send ACK(A_S ≪ A_L) to A_S
12:        P_O ← P_O ∪ {A_S ≪ A_L}
13:    end if
14: else                                  ▷ Abhängigkeitskonflikt noch nicht erreicht
15:    send ACK(A_S ≪ A_L) to A_S
16:    P_O ← P_O ∪ {A_S ≪ A_L}
17: end if
```

Weiterhin müsste durch das Hinzufügen der Präzedenz $A_S \lessdot A_L$ vor der Ausführung der Aktivität A_L eine zusätzliche Synchronisation erfolgen. Wurde A_L jedoch bereits im Rahmen der optimistischen Ausführung gestartet, so kann die Synchronisation nicht mehr erfolgen. Auch hier muss daher eine der beiden optimistischen Ausführungen abgebrochen werden. Analog wird daher der Aufwand berechnet und der Sender der Nachricht informiert.

Tritt keiner der beiden genannten Problemfälle auf, kann die optimistisch erzeugte Präzedenz bestätigt und lokal in der Menge $\mathcal{P}_O$ vermerkt werden. Wenn hingegen derzeit keine optimistische Ausführung aktiv ist, hat der Kontrollfluss den betroffenen Abhängigkeitskonflikt noch nicht erreicht und die neue Präzedenz kann ebenfalls bestätigt werden, da sie keine Widersprüche auslösen kann.

Der Empfang einer Bestätigung ist in Algorithmus 4.3 dargestellt. Hier wird zunächst geprüft, ob es sich um eine Bestätigung für die aktuelle optimistische Ausführung handelt. Ist dies nicht der Fall, muss die Bestätigung für eine frühere optimistische Ausführung sein, die inzwischen jedoch abgebrochen wurde. Die Nachricht wird daher ignoriert. Wenn alle Bestätigungen empfangen wurden, wird die Nachricht APPLY$(A_S \lessdot A_L)$

versendet, die neue Präzedenz endgültig gespeichert und die optimistische Ausführung beendet, indem der zuvor gesicherte Zustand verworfen wird.

Algorithmus 4.3 Empfang der Nachricht $\textsc{ack}(A_S \lessdot A_L)$

```
1: if O = A_S ≪ A_L then
2:     if ACK(A_S ≪ A_L) von allen A_i ∈ A empfangen then
3:         for all A_i ∈ A do
4:             send APPLY(A_S ≪ A_L) to A_i
5:         end for
6:         P_Z ← P_Z ∪ {A_S ≪ A_L}
7:         Beende optimistische Ausführung
8:     end if
9: end if
```

Durch den in Algorithmus 4.4 dargestellten Empfang der Nachricht $\textsc{apply}(A_S \lessdot A_L)$ wird eine optimistische Konfliktauflösung auch bei den anderen an dem Abhängigkeitskonflikt beteiligten parallelen Ausführungspfaden endgültig festgeschrieben. Es werden daher zunächst die Mengen $\mathcal{P}_O$ und $\mathcal{P}_Z$ entsprechend angepasst. Falls sich der Empfänger dieser Nachricht in der optimistischen Ausführung befindet, wird anschließend geprüft, ob durch die neue Präzedenz der die optimistische Ausführung auslösende Abhängigkeitskonflikt aufgelöst wurde. In diesem Fall kann auch die lokale optimistische Ausführung erfolgreich beendet werden, ohne jedoch die dabei hinzugefügte Präzedenz zu verwenden. Dies wäre überflüssig und würde eine zusätzliche Synchronisation hervorrufen. Das dazu eventuell erzeugte Replikat wird daher ebenfalls verworfen. Zur Signalisierung des Abbruches wird die Nachricht $\textsc{cancel}(\mathcal{O})$ versendet.

Algorithmus 4.4 Empfang der Nachricht $\textsc{apply}(A_S \lessdot A_L)$

```
1: P_O ← P_O \ {A_S ≪ A_L}
2: P_Z ← P_Z ∪ {A_S ≪ A_L}
3: if optimistische Ausführung aktiv then
4:     if ℜ tritt nicht mehr auf then
5:         Beende optimistische Ausführung
6:         for all A_i ∈ A do
7:             send CANCEL(O) to A_i
8:         end for
9:     end if
10: end if
```

Falls eine optimistische Konfliktauflösung nicht von allen beteiligten Geräten bestätigt wird sondern mindestens ein Gerät mit der Nachricht $\textsc{nack}_\omega(A_S \lessdot A_L, B_S \lessdot B_L)$ antwortet, wird Algorithmus 4.5 ausgeführt. $A_S \lessdot A_L$ ist hierbei die bei der lokalen Konfliktauflösung und $B_S \lessdot B_L$ die von dem widersprechenden Gerät optimistisch hinzugefügte Präzedenz. Im Anschluss an die Überprüfung, ob die derzeitige optimistische Ausführung gemeint ist, werden zwei Fälle unterschieden. In dem Fall, dass die von dem widersprechenden Gerät optimistisch hinzugefügte Präzedenz in der Menge

$\mathcal{P}_O$ enthalten ist, wurde von dem lokalen Gerät bereits die Nachricht $\textsc{ack}(A_S \lessdot A_L)$ zur Bestätigung versendet. Das lokale Gerät muss daher die optimistische Ausführung abbrechen, um den Widerspruch aufzulösen, da der parallele Pfad seine Konfliktauflösung bereits als endgültig festgeschrieben haben könnte. Ist die Präzedenz nicht in $\mathcal{P}_O$ enthalten, so wurde auf $\textsc{request}(A_S \lessdot A_L)$ mit $\textsc{nack}_\psi(B_S \lessdot B_L, A_S \lessdot A_L)$ geantwortet. Anhand des in $\Omega(B_S \lessdot B_L)$ hinterlegten Gewichts ψ wird nun geprüft, ob die lokale oder die entfernte Ausführung abgebrochen werden muss. Wenn die Gewichte der beiden beteiligten Ausführungen gleich sind, wird der Abbruch über eine eindeutig definierte Ordnungsrelation der Aktivitäten des Prozesses bestimmt.

Im Falle eines Abbruchs wird der Zustand im Lebenszyklus der Aktivitäten und des Prozesses sowie die Werte aller Variablen wiederhergestellt. Die Mengen $\mathcal{P}_O$ und $\mathcal{P}_Z$, die bereits bestätigte Präzedenzen beinhalten, müssen jedoch dabei zur Wahrung der Konsistenz beibehalten werden.

Algorithmus 4.5 Empfang der Nachricht $\textsc{nack}_\omega(A_S \lessdot A_L, B_S \lessdot B_L)$

```
1: if O = A_S ≪ A_L then
2:     if B_S ≪ B_L ∈ P_O then                ▷ B_S ≪ B_L wurde bereits bestätigt
3:         Breche optimistische Ausführung ab
4:         for all A_i ∈ A do
5:             send cancel(A_S ≪ A_L) to A_i
6:         end for
7:     else                                   ▷ B_S ≪ B_L wurde nicht bestätigt
8:         if Ω(B_S ≪ B_L) < ω or (Ω(B_S ≪ B_L) = ω and ord(A_S) < ord(B_S)) then
9:             Breche optimistische Ausführung ab
10:            for all A_i ∈ A do
11:                send cancel(A_S ≪ A_L) to A_i
12:            end for
13:        end if
14:    end if
15: end if
```

Mit der Nachricht $\textsc{cancel}(A_S \lessdot A_L)$ wird mitgeteilt, dass sich eine optimistische Annahme nicht erfüllt hat und daher die vorgenommene Konfliktauflösung nicht durchgeführt werden darf. Beim Empfang der Nachricht muss somit die Präzedenz aus der Menge $\mathcal{P}_O$ entfernt werden (Algorithmus 4.6).

Algorithmus 4.6 Empfang der Nachricht $\textsc{cancel}(A_S \lessdot A_L)$

```
1: P_O ← P_O \ {A_S ≪ A_L}
```

4.6.2 Fazit

Die in diesem Abschnitt vorgestellte optimistische Konfliktauflösung ermöglicht es den Teilnehmern einer parallelen Ausführung, die auftretenden Abhängigkeitskonflikte zur Laufzeit ohne zentralen Koordinator aufzulösen, selbst wenn mobile Geräte beteiligt

sind. Da die Auflösung eines Abhängigkeitskonfliktes erst so spät wie möglich erfolgt, kann der aktuelle Ausführungszustand des Prozesses bei der Wahl einer geeigneten, den Abhängigkeitskonflikt auflösenden Präzedenz berücksichtigt werden. Mit dieser Strategie wird vermieden, dass durch eine Konfliktauflösung Synchronisationen entstehen, die zum Warten auf langandauernde Aktivitäten führen (vgl. Beispiel aus Abbildung 4.16).

4.7 Diskussion der Korrektheit

In Rahmen der Herleitung zur Erkennung von Abhängigkeitskonflikten wurde zwar bereits diskutiert, dass Abhängigkeitskonflikte aufgelöst werden müssen, um die korrekte Ausführung eines Prozesses auf Replikaten zu gewährleisten. Dies genügt jedoch nicht formalen Ansprüchen an einen Korrektheitsbeweis. In diesem Abschnitt erfolgt daher die formale Betrachtung des vorgestellten Konzepts. Es ist dabei zu zeigen, dass jede Ausführung beliebiger Prozesse nach der vorgestellten Methode korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist. Die Korrektheit betrifft jedoch nur Variablen, die mit der Datenklasse „*Serialisierbar*“ ausgezeichnet sind. Variablen anderer Datenklassen weisen eigene Korrektheitskriterien auf und werden daher im Folgenden nicht berücksichtigt. Bevor der eigentliche Beweis durchgeführt werden kann, müssen zunächst einige Begriffe eingeführt und die Rahmenbedingungen geklärt werden.

4.7.1 Begriffe und Rahmenbedingungen

Um zu überprüfen, ob eine Ausführung gemäß dem Konzept korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist, muss diese zunächst abgeschlossen sein. Es wurden daher alle existierenden Abhängigkeitskonflikte durch das Hinzufügen von weiteren Präzedenzen aufgelöst, d. h. der Abhängigkeitsgraph ist zyklenfrei. Dies wird durch die optimistische Konfliktauflösung sichergestellt. Ebenso existiert keine selektive Ausführung mehr, da die nicht gewählten Aktivitäten in den Zustand „*skipped*“ versetzt wurden und somit in einer Serialisierung nicht zu berücksichtigen sind. Somit gilt für zwei Aktivitäten A und B, dass entweder $A = B$, $A < B$, $B < A$ oder $A \parallel B$ ist.

Die Handhabung der Replikate und das in dem Konzept verankerte Replizieren der Prozessdaten bei einer Verzweigung des Kontrollflusses sowie das Zusammenführen der Replikate bei einer Synchronisation des Kontrollflusses müssen für den Beweis formal ausgedrückt werden. Gemäß der in Abschnitt 4.1.1 beschriebenen Replikation existiert neben dem initial verwendeten Replikat für jede aktivierte von einer Verzweigung oder Synchronisation ausgehende Transition ein eigenes Replikat. Alle Aktivitäten, die zwischen zwei Verzweigungen, zwei Synchronisationen oder einer Verzweigung und einer Synchronisation liegen, greifen ausschließlich auf das entsprechende Replikat zu. Formal gesehen muss daher bei einer Verzweigung des Kontrollflusses eine separate Transaktion alle Variablen des bisherigen Replikats lesen und anschließend in die neu-

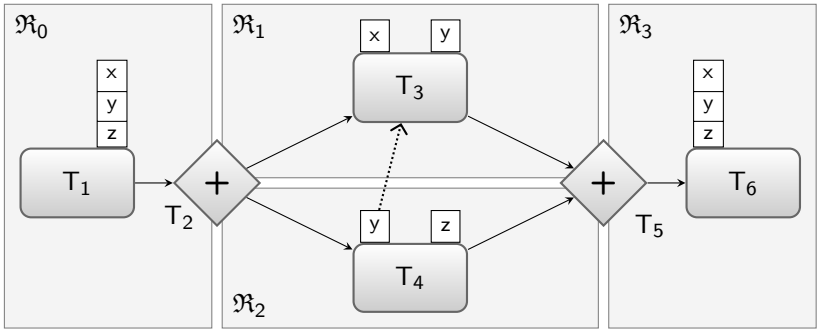


Abbildung 4.23: Aktivitäten, Verzweigungen und Synchronisationen werden als Transaktionen interpretiert

en Replikate schreiben. Ebenso muss bei der Synchronisation des Kontrollflusses eine Transaktion jede Variable von einem der bisherigen Replikate lesen und in das neue Replikat schreiben. Für eine Variable x wird dabei das Replikat zum Lesen gewählt, auf welchem diejenige Aktivität ausgeführt wurde, die x in der topologischen Sortierung des Abhängigkeitsgraphen zuletzt geschrieben hat.

Eine-Kopie-Serialisierbarkeit wurde im Kontext von Datenbanktransaktionen definiert. Die formale Grundlage bilden daher sogenannte *Datenbank-Logs* (kurz *Log*). Ein *Log* ist eine partiell geordnete Menge von Operationen auf einer Datenbank [BG83]. Jede Operation gehört zu einer Transaktion T_i und liest beziehungsweise schreibt eine Variable x auf einem Replikat $\mathfrak{R}_a$. Im Folgenden wird eine Lese- beziehungsweise Schreiboperation durch $r_i[x_a]$ beziehungsweise $w_i[x_a]$ ausgedrückt. Die partielle Ordnung spiegelt dabei die in den beteiligten Transaktionen existierende Ordnung wieder. Des Weiteren muss jeder Leseoperation auf einem Datum eine Schreiboperation auf diesem Datum desselben Replikats vorangegangen sein und konfligierende Operationen müssen geordnet sein. Zwei Operationen konfligieren, wenn sie auf dasselbe Datum auf demselben Replikat zugreifen sowie eine der Operationen eine Schreiboperation ist. Ein *One-Copy-Log* ist ein spezielles Log, bei dem es für jede Variable nur genau ein Replikat gibt. Zur besseren Unterscheidung wird ein Log mit mehreren Replikaten auch *Replicated-Database-Log* (*RD-Log*) genannt.

Die Ausführung eines Prozesses gemäß dem vorgestellten Konzept muss daher zunächst in ein RD-Log überführt werden. Aktivitäten entsprechen dabei Transaktionen, die Leseoperationen, gefolgt von Schreiboperationen, ausführen und somit eine Ordnung der Lese- und Schreiboperationen vorgeben. Diese Ordnung wird erweitert um die Ordnung aus den Präzedenzen gemäß des Prozessmodells und bildet so das RD-Log. Ohne Beschränkung der Allgemeinheit wird weiterhin angenommen, dass die erste Transaktion eines Prozesses alle Variablen mit einem Standardwert initialisiert. Auf jedem Replikat erfolgt somit für jede Variable zunächst eine Schreiboperation bevor von den Variablen gelesen wird. Da der Zugriff von Aktivitäten auf ein bestimmtes Replikat immer sequentiell erfolgt, sind alle konfligierenden Operationen geordnet. Es handelt sich also um ein gültiges Log (siehe Beispiel in den Abbildungen 4.23 bis 4.25).

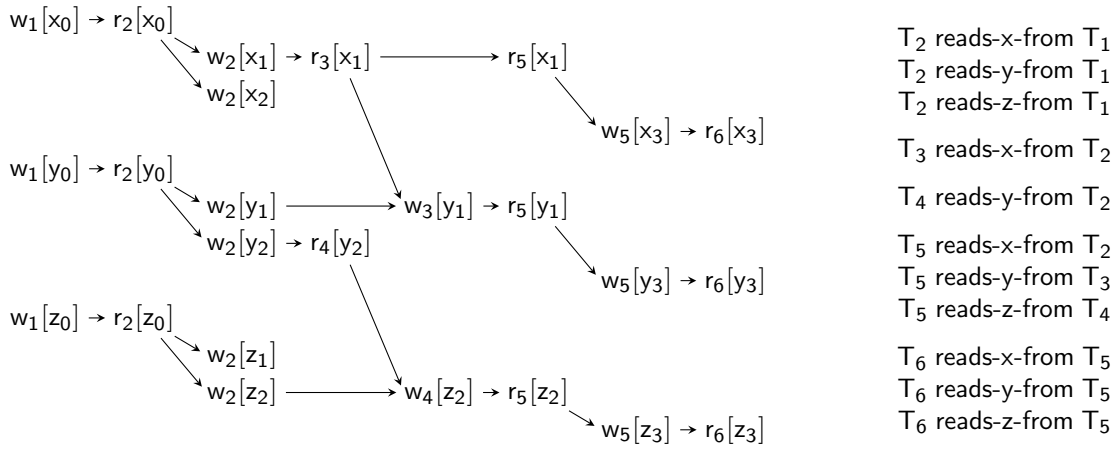


Abbildung 4.24: Vereinfachtes RD-Log der Ausführung des Prozesses aus Abbildung 4.23. Zur besseren Übersicht sind nur die charakteristischen Kanten dargestellt. Rechts ist die *reads-from*-Relation vollständig abgebildet.

Der Beweis stützt sich auf die Tatsache, dass eine Ausführung korrekt im Sinne der Eine-Kopie-Serialisierbarkeit ist, wenn sie äquivalent zu einer seriellen Ausführungsfolge auf einer einzigen Kopie ist. Nach Bernstein und Goodman liegt eine Äquivalenz genau dann vor, wenn zwei Ausführungen die gleiche *reads-from*-Relation besitzen [BG83]. Es muss also gezeigt werden, dass zu jeder gemäß dem Konzept möglichen Ausführung eine serielle Ausführungsfolge existiert, die bei der Betrachtung einer einzigen Kopie dieselbe *reads-from*-Relation besitzt wie die ursprüngliche Ausführung.

Definition 4.9 (reads-from) *Zwei Transaktionen T_i und T_j weisen in einem Log L die Beziehung T_j* reads-x-from *T_i genau dann auf, wenn $w_i[x_a]$ und $r_j[x_a]$ Operationen in L sind, welche die Ordnung $w_i[x_a] < r_j[x_a]$ aufweisen und kein $w_k[x_a]$ zwischen die beiden Operationen fällt. [BG83]*

Die *reads-from*-Relation stellt somit eine eindeutige Abbildung $f\colon D \rightarrow T$ dar, wobei $D \subseteq T \times V$, T die Menge der Transaktionen und V die Menge der Variablen ist. Die Definitionsmenge ist dabei für alle Logs, die dieselben Transaktionen beinhalten, stets gleich. Um die Gleichheit der *reads-from*-Relationen f_1 und f_2 zweier Logs mit denselben Transaktionen zu zeigen, muss also nur bewiesen werden, dass für jedes Tupel $(T, V) \in \mathrm{Def}(f_1)$ gilt: $f_1(T, V) = f_2(T, V)$. Es muss somit gezeigt werden, dass jedes T_j reads-x-from T_i, das in dem einen Log existiert, auch in dem anderen Log existiert.

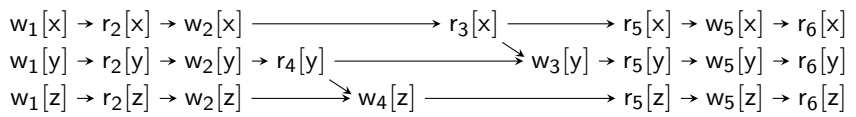


Abbildung 4.25: Vereinfachtes One-Copy-Log der seriellen Ausführung des Prozesses aus Abbildung 4.23. Es ist zu erkennen, dass dieses Log die gleiche *reads-from*-Relation besitzt, wie das RD-Log aus Abbildung 4.24.

4.7.2 Beweis

Satz 4.10 *Sei L ein RD-Log gemäß dem vorgestellten Konzept und sei L_S das One-Copy-Log der seriellen Ausführung, die bei den Synchronisationen in der ursprünglichen Ausführung verwendet wurden. Dann existieren alle T_j* reads-x-from *T_i aus L auch in L_S.*

Beweis: Sei T_j reads-x-from T_i in L. Gemäß Definition 4.9 müssen drei Eigenschaften zutreffen, damit T_j reads-x-from T_i in L_S gilt:

1. Es gibt die Operationen $w_i[x]$ und $r_j[x]$ in L_S.
   Beweis: Da T_j reads-x-from T_i in L gilt, gibt es $w_i[x_a]$ und $r_j[x_a]$ in L und daher auch die entsprechenden Operationen $w_i[x]$ und $r_j[x]$ in L_S.

2. Es gilt $w_i[x] < r_j[x]$ in L_S.
   Beweis: Da T_j reads-x-from T_i in L, gilt $w_i[x_a] < r_j[x_a]$ in L. Es muss daher im Prozessmodell eine Präzedenz $T_i < T_j$ geben, die zu dieser Ordnung geführt hat. Wegen der Ableitung von L_S aus dem Abhängigkeitsgraphen gilt diese Ordnung auch für die entsprechenden Operationen $w_i[x]$ und $r_j[x]$ in L_S.

3. Kein $w_k[x]$ liegt zwischen $w_i[x]$ und $r_j[x]$ in L_S.
   Beweis: Sei $w_k[x]$ eine beliebige Schreiboperation auf x in L_S. Falls $k = i$ oder $k = j$, so ist die Behauptung trivialerweise erfüllt. Im Folgenden wird daher Ungleichheit vorausgesetzt. Es werden zwei Fälle unterschieden:

   a) T_k wird in L parallel zu T_j und T_i ausgeführt.
      Es existiert dann eine Lese-Schreib-Abhängigkeit $T_j \prec T_k$.
      Da L_S aus dem Abhängigkeitsgraphen abgeleitet wurde, liegt $r_j[x]$ vor $w_k[x]$ in L_S. Da $w_i[x] < r_j[x]$ und $r_j[x] < w_k[x]$, liegt $w_k[x]$ somit nicht zwischen $w_i[x]$ und $r_j[x]$ in L_S.

   b) T_k wird in L nicht parallel zu T_j und T_i ausgeführt.
      T_k wird somit in L in einer seriellen Sequenz mit T_j und T_i ausgeführt.
      Da T_j reads-x-from T_i in L, gilt somit $T_k < T_i$ oder $T_j < T_k$ in L. Diese Ordnung muss sich auch im Abhängigkeitsgraphen widerspiegeln.
      Da L_S aus dem Abhängigkeitsgraphen abgeleitet wurde, liegt $w_k[x]$ vor $w_i[x]$ oder nach $r_j[x]$ in L_S und somit nicht zwischen den beiden Operationen. □

Satz 4.11 *Jede Ausführung gemäß dem vorgestellten Konzept ist korrekt im Sinne der Eine-Kopie-Serialisierbarkeit.*

Beweis: Sei L das RD-Log einer beliebigen Ausführung gemäß dem vorgestellten Konzept. Nach Satz 4.10 existiert dann ein serielles One-Copy-Log L_S, das dieselbe *reads-from*-Relation besitzt. Nach [BG83] sind L und L_S dann äquivalent. L ist somit korrekt im Sinne der Eine-Kopie-Serialisierbarkeit. □

Kapitel 5

Prototypische Implementierung

Neben der konzeptionellen Lösung der identifizierten Probleme ist das Ziel dieser Arbeit, das entwickelte Konzept in einer prototypischen Implementierung auf seine Realisierbarkeit zu überprüfen. In diesem Kapitel werden die Ergebnisse der praktischen Umsetzung vorgestellt. Die im Konzept verankerte Abstraktion von konkreten Prozessbeschreibungssprachen und Workflow-Engines setzt sich auch in der prototypischen Implementierung fort soweit dies möglich und sinnvoll ist. Die Implementierung besteht daher aus zwei Teilen: Den ersten Teil bildet die generischen Komponente *PAEX* (*PArallel Execution eXtension*, Erweiterung für die parallele Ausführung), die das entwickelte mathematische Metamodell verwendet, um Abhängigkeitskonflikte zu erkennen und aufzulösen. Darüber hinaus werden Methoden bereitgestellt, um die Replikate paralleler Ausführungspfade zu synchronisieren. Der zweite Teil der Implementierung stellt die Verbindung zwischen einer konkreten Workflow-Engine, die sequentielle Prozesse ausführen kann, und der Erweiterung für die parallele Ausführung her. Hierfür muss ein Prozess, der in der konkreten Prozessbeschreibungssprache formuliert ist, verarbeitet und in das mathematische Modell überführt werden. Außerdem müssen Schnittstellen zu bestimmten Bereichen der Workflow-Engine hergestellt werden. Softwaretechnisch handelt es sich dabei also um einen Adapter [GHJV03] (vgl. Abbildung 5.1).

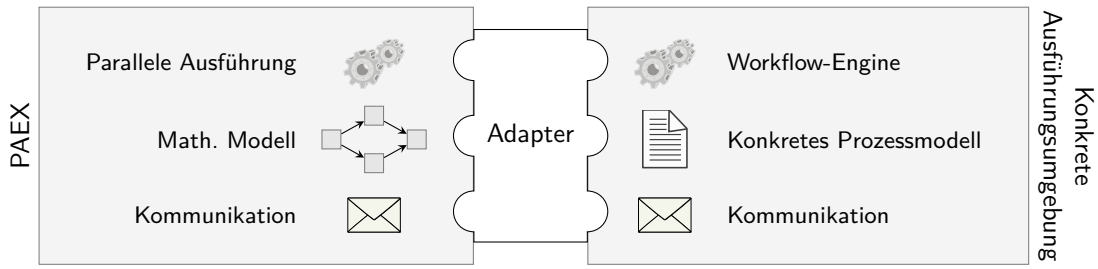


Abbildung 5.1: Ein Adapter verbindet die Komponente zur parallelen Ausführung mit bestehenden Workflow-Engines

In diesem Kapitel wird zunächst auf die von konkreten Workflow-Engines unabhängige Komponente eingegangen. Abschnitt 5.1 beginnt daher mit der Beschreibung einer Software-Architektur wie sie bei der prototypischen Implementierung dieser Komponente verwendet wurde. Anschließend wird die Schnittstelle zu konkreten Workflow-

Engines diskutiert. In Abschnitt 5.2 wird schließlich die Realisierung dieser Schnittstelle in Form eines Adapters für die konkrete Prozessbeschreibungssprache und Ausführungsumgebung im Projekt *DEMAC* vorgestellt. Das Kapitel schließt in Abschnitt 5.3 mit einer Bewertung der prototypischen Implementierung.

5.1 PAEX-Architektur

Die Grundlage für die Implementierung ist die objektorientierte Repräsentation des mathematischen Metamodells wie es in Kapitel 4 vorgestellt wurde. Mit Hilfe von Adaptern können konkrete Prozessbeschreibungssprachen angebunden werden. Ein Adapter erzeugt dazu aus dem Prozessmodell beziehungsweise der Prozess-Instanz einer konkreten Prozessbeschreibungssprache das für die parallele Ausführung verwendete mathematische Modell. Gleichzeitig ermöglicht der Adapter zur Ausführungszeit des Prozesses die Rückkopplung von Operationen auf dem mathematischen Modell, deren Wirkungen auch auf die ursprüngliche Prozess-Instanz übertragen werden müssen. Dies betrifft beispielsweise das Hinzufügen von Präzedenzen im Rahmen der optimistischen Konfliktauflösung. In diesem Abschnitt wird zunächst die objektorientierte Repräsentation des mathematischen Metamodells dargestellt und anschließend auf die Architektur der Erweiterung für die parallele Ausführung von mobilen Prozessen eingegangen.

5.1.1 Objektorientierte Abbildung des mathematischen Metamodells

Das in dieser Arbeit entwickelte mathematische Metamodell verwendet mehrfach Graphen zum Ausdruck verschiedener Beziehungen. Es wurde daher ein Paket zur Darstellung von mathematischen Graphen und zur Durchführung von Operationen auf diesen implementiert wie beispielsweise die Zyklenerkennung und die topologische Sortierung zur Berechnung von möglichen Serialisierungen (vgl. Abbildung 5.2).

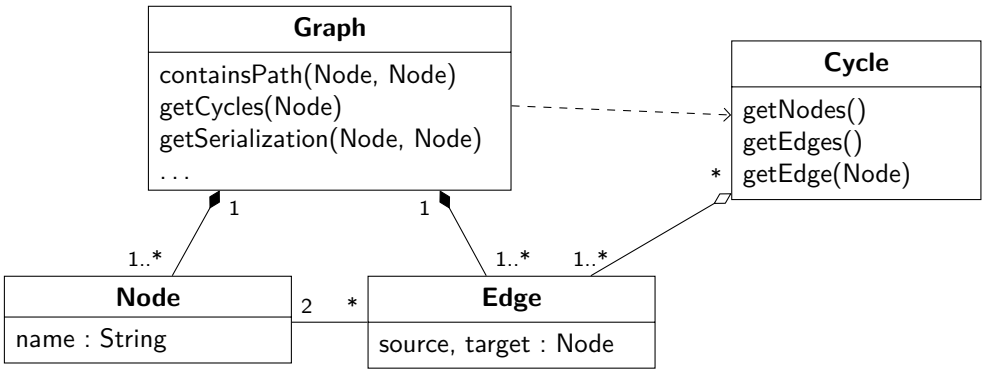


Abbildung 5.2: Struktur des Java-Pakets `de.korelstar.math.graph`

Die Klasse `Process` verwendet einen solchen Graphen aus Aktivitäten und Präzedenzen zur Verwaltung des Kontrollflusses. Abbildung 5.3 zeigt, wie der generische Graph durch das Paket `model` um konkrete Knoten (`Activity`) und konkrete Kanten

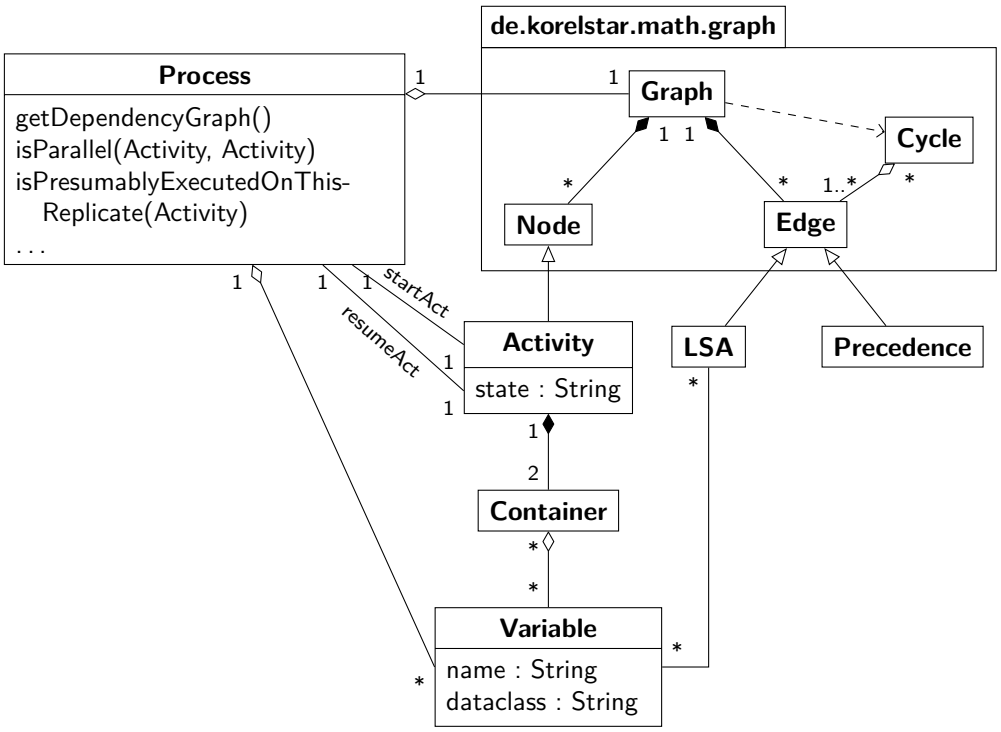


Abbildung 5.3: Struktur des Java-Pakets `de.korelstar.mobile.process.parallel.model`

(`Precedence`) erweitert wird. Zusätzlich bedient sich `Process` einer Menge von Beziehungen zwischen Aktivitäten, um die selektive Ausführung auszudrücken. Aus diesen Daten wird automatisch abgeleitet welche Aktivitäten parallel ausgeführt werden, um anschließend den Abhängigkeitsgraphen eines Prozesses berechnen zu können. Dieser wird ebenfalls durch die Klasse `Graph` repräsentiert, wobei einzelne Kanten nun statt Präzedenzen auch Lese-Schreib-Abhängigkeiten (`LSA`) sein können. Auf diese Weise werden alle in dem mathematischen Metamodell benötigten Operationen auch in der objektorientierten Repräsentation realisiert; dazu zählt insbesondere die Erkennung von Abhängigkeitskonflikten.

Um die Konflikterkennung unter Berücksichtigung der von dem Modellierer gewählten Datenklassen durchführen zu können, wird eine Möglichkeit zur Zuordnung von Variablen zu Datenklassen benötigt. Die gewählte Architektur sieht daher eine zusätzliche Datei vor, die mit der Prozessbeschreibung verknüpft wird. Bei der Zuordnung von Datenklassen handelt es sich um eine nicht notwendigerweise totale Abbildung von der Menge der Variablen eines Prozesses in die Menge der Datenklassen. Da sowohl Varia-

```
Zuordnungsdatei = { Eintrag } ;
Eintrag = Variablenname "=" Datenklassenname "\n" ;
Variablenname = Bezeichner ;
Datenklassenname = Bezeichner ;
```

Abbildung 5.4: EBNF-Syntax einer Zuordnungsdatei zur Zuweisung von Datenklassen

blen als auch Datenklassen über ihren Namen eindeutig identifiziert werden können, erfolgt in der Datei eine einfache Verknüpfung von Zeichenketten. In Abbildung 5.4 ist die eingesetzte Syntax unter Verwendung der erweiterten Backus-Naur-Form (EBNF) angegeben.

5.1.2 Ausführung von parallelen mobilen Prozessen

Die Steuerung des Kontroll- und Datenflusses, die Ausführung einzelner Aktivitäten sowie weitere Aufgaben, die bei der Ausführung von sequentiellen Prozessen notwendig sind, bleiben auch bei parallelen Prozessen in der Hand der Workflow-Engine. Die entwickelte Erweiterung zur Ausführung paralleler mobiler Prozesse greift nur in bestimmten Situationen in den existierenden Ablauf ein. Dies betrifft im Wesentlichen drei Aufgaben: die Distribution des Kontrollflusses, die Synchronisation des Kontrollflusses sowie das Überprüfen und Auflösen von Abhängigkeitskonflikten während der parallelen Ausführung.

Im Zentrum der Realisierung dieser Aufgaben steht die Klasse `ParallelExecutor` (vgl. Abbildung 5.5). Sie wird an den entsprechenden Stellen des Kontrollflusses von der Workflow-Engine aufgerufen. Die Referenz auf den jeweiligen Prozess bildet die abstrakte Klasse `ProcessAdapter`. Sie setzt das Entwurfsmuster *Adapter* [GHJV03] um, indem ihre Schnittstelle den Zugriff auf das mathematische Prozessmodell (`Process`) und die Rückkopplung zu der konkreten Prozessbeschreibung der jeweiligen Workflow-Engine ermöglicht. Die entsprechenden Methoden, wie beispielsweise das Hinzufügen von neuen Präzedenzen, müssen von einer konkreten Adapterklasse implementiert werden, die von `ProcessAdapter` erbt.

Da zur Ausführung der Synchronisation die Replikate anderer paralleler Pfade eines Prozesses gefunden werden müssen, benötigt `ParallelExecutor` den Zugriff auf die von der Workflow-Engine derzeit verwalteten Prozess-Instanzen. Dieser Zugriff erfolgt über die Schnittstelle `AdapterFactory`, die eine Fabrikmethode vorgibt und von der Workflow-Engine beziehungsweise einem entsprechendem Modul implementiert werden muss.

Eine Konfliktauflösung erfolgt immer in Abhängigkeit der zur Verfügung stehenden Implementierungen von Datenklassen. Diese stellen jeweils die Schnittstelle `DataClass` bereit und werden von `ParallelExecutor` verwaltet. Soll ein Abhängigkeitskonflikt aufgelöst werden, prüft `ParallelExecutor`, welche Datenklassen an diesem Konflikt beteiligt sind. Zur eigentlichen Durchführung der Konfliktauflösung wird die Methode `solveConflict(...)` von dem Objekt der entsprechenden Datenklasse aufgerufen.

Während die Datenklasse `Unsynchronized` zur Konfliktauflösung keine weiteren Operationen ausführt, wird in der Datenklasse `Serializable` die optimistische Konfliktauflösung initiiert. Dazu müssen Nachrichten mit anderen Geräten ausgetauscht werden. Auch hier muss daher ein Adapter eingesetzt werden, um die Funktionalität zur Übertragung von Nachrichten der jeweiligen Ausführungsumgebung nutzen zu können.

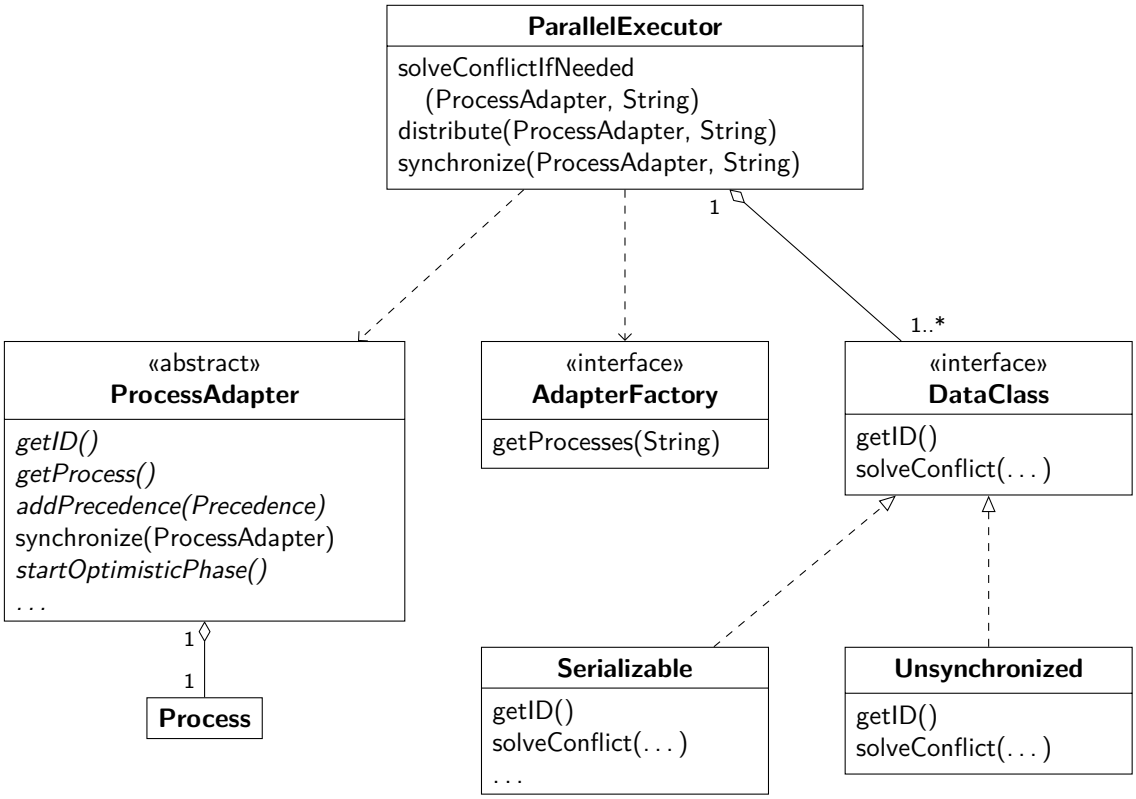


Abbildung 5.5: Realisierung der notwendigen Aufgaben bei paralleler Ausführung

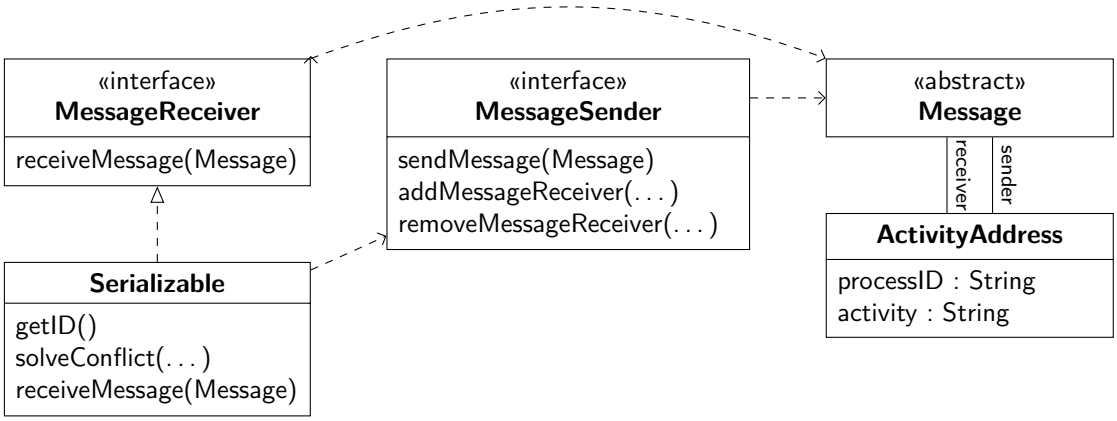


Abbildung 5.6: Schnittstellen zur Übertragung von Nachrichten für die Konfliktauflösung

Abbildung 5.6 zeigt die zur Nachrichtenübertragung verwendeten Komponenten. Zum Versenden von Nachrichten wird dabei die Schnittstelle `MessageSender` genutzt. Um Antworten auf die gesendeten Nachrichten empfangen zu können, kann über dieselbe Schnittstelle ein `MessageReceiver` angemeldet werden. `Serializable` implementiert daher die entsprechende Methode. Auf diese Weise kann eine Konfliktauflösung mit den am Abhängigkeitskonflikt beteiligten Aktivitäten koordiniert werden. Die dabei übermittelten Nachrichten verwenden zur Adressierung die Klasse `ActivityAddress`, die eine Zuordnung zu einer bestimmten Aktivität einer Prozess-Instanz realisiert.

5.1.3 Schnittstellen zur Workflow-Engine

Abschließend wird kurz zusammengefasst, welche Schritte zur Anbindung von PAEX an eine konkrete Workflow-Engine notwendig sind. Voraussetzung ist dabei stets, dass die Workflow-Engine bereits sequentielle Prozesse ausführen kann.

Implementierung von Schnittstellen Um den Zugriff auf das Prozessmodell und den Prozesszustand zu ermöglichen, müssen die Schnittstellen `AdapterFactory` und `ProcessAdapter` implementiert werden. `AdapterFactory` erlaubt den Zugriff auf die derzeit ausgeführten Prozesse über eine Fabrikmethode. `ProcessAdapter` ermöglicht das Durchführen von Änderungen des Zustands eines bestimmten Prozesses, so dass sich diese nicht nur im mathematischen Modell auswirken sondern auch von der Workflow-Engine wahrgenommen werden. Zur Realisierung von Konfliktauflösungen muss darüber hinaus die Schnittstelle `MessageSender` implementiert werden, da hierbei Nachrichten zu anderen parallelen Ausführungspfaden versendet werden müssen.

Verwendung von ParallelExecutor In ihrer Verarbeitung des Kontrollflusses kann die Workflow-Engine auf die Funktionalität vom `ParallelExecutor` zurückgreifen. Konkret kann immer, wenn gemäß des Kontrollflusses eine Distribution oder Synchronisation erfolgt – beispielsweise durch *Split* und *Join*– die entsprechende Methode vom `ParallelExecutor` aufgerufen werden. Um die korrekte Ausführung im Sinne der Eine-Kopie-Serialisierbarkeit zu gewährleisten, muss vor der Ausführung einer Aktivität auf einem parallelen Ausführungpfad die Methode `solveConflictIfNeeded(...)` aufgerufen werden, um eine Konfliktauflösung vorzunehmen, falls dies erforderlich ist.

5.2 Integration in das Projekt DEMAC

In diesem Abschnitt wird die Integration von PAEX in eine konkrete Workflow-Engine für die Ausführung von mobilen Prozessen gezeigt. Dies erfolgt am Beispiel der DEMAC-Middleware, die in Abschnitt 5.2.1 vorgestellt wird. Im Zuge der Integration der parallelen Ausführung in DEMAC waren einige grundlegende Modifikationen

notwendig, um gewisse auftretende Probleme lösen zu können. Diese werden in Abschnitt 5.2.2 diskutiert. In Abschnitt 5.2.3 wird die Architektur des Adapters präsentiert, der die Verknüpfung zwischen PAEX und DEMAC herstellt.

5.2.1 Die DEMAC-Middleware

Um das im Forschungsprojekt DEMAC[1] (*Distributed Environment for Mobility-Aware Computing*) entwickelte Konzept der *kontextbasierten Kooperation* (vgl. Abschnitt 2.1.3) und dessen Realisierung in Form von mobilen Prozessen (vgl. Abschnitt 2.3.3) zu evaluieren, wurde eine Systemplattform erarbeitet, die mobile Prozesse ohne zentrale Koordinierungsinstanz ausführen kann. Umgesetzt ist diese Systemplattform als Middleware, die Anwendungen eine Schnittstelle mit hohem Abstraktionsgrad zur Verfügung stellt. Die Verwendung von Konzepten und Methoden der diensteorientierten Architektur hat zu einem modular aufgebauten System geführt, bei dem einzelne Elemente beliebig austauschbar sind (vgl. Abbildung 5.7). So abstrahiert eine einheitliche Kommunikationsplattform zunächst von konkreten Transportprotokollen – wie Bluetooth oder WLAN – und ermöglicht den asynchronen Versand von Nachrichten (*asynchroner Transportdienst*). Falls dies gewünscht ist, kann eine Sicherheitskomponente die Nachrichten transparent signieren oder verschlüsseln [Win07]. Ein *Ereignisdienst* ermöglicht die Mitteilung von Zustandsänderungen in Form von Ereignissen. Da gerade in mobilen Umgebungen ein sehr dynamischer Kontext vorliegt, verwaltet ein *Kontextdienst* diese Informationen und stellt sie über seine Schnittstelle weiteren Anwendungen bereit. Die eigentliche Ausführung von komplexen Aufgaben in Form von mobilen Prozessen wird durch einen *Prozessdienst* realisiert. [Kun08]

Mobile Prozesse werden im Rahmen von DEMAC in der Prozessbeschreibungssprache *DPDL* (*DEMAC Process Description Language*) formuliert. DPDL basiert auf der Meta-Prozessbeschreibungssprache *XPDL* und ist speziell an die Bedürfnisse mobiler Geräte angepasst. So speichert DPDL in einer einzigen XML-Datei sowohl das Prozessmodell als auch den Zustand einer konkreten Ausführung des Prozesses. Der gesamte Prozess kann daher sehr einfach auf ein anderes Gerät migriert werden, wenn das aktuelle Gerät nicht in der Lage ist, die Ausführung fortzuführen. [Zap05]

Um leistungsschwache mobile Geräte zu entlasten, werden bestimmte abgeleitete Daten über den Zustand des Prozesses explizit in der Prozessbeschreibung verankert. Beispielsweise wird die aktuell zu bearbeitende Aktivität explizit angegeben, so dass nicht ständig der komplette Kontrollfluss verarbeitet werden muss. Darüber hinaus existiert eine explizite Fehlerbehandlung sowie die Möglichkeit nicht-funktionale Kriterien angeben zu können. Parallelität kann in DPDL sowohl durch die Konstrukte *Split* und *Join* realisiert werden als auch durch asynchrone Subprozesse. [Zap05]

---

1 `http://vsis-www.informatik.uni-hamburg.de/projects/demac/`

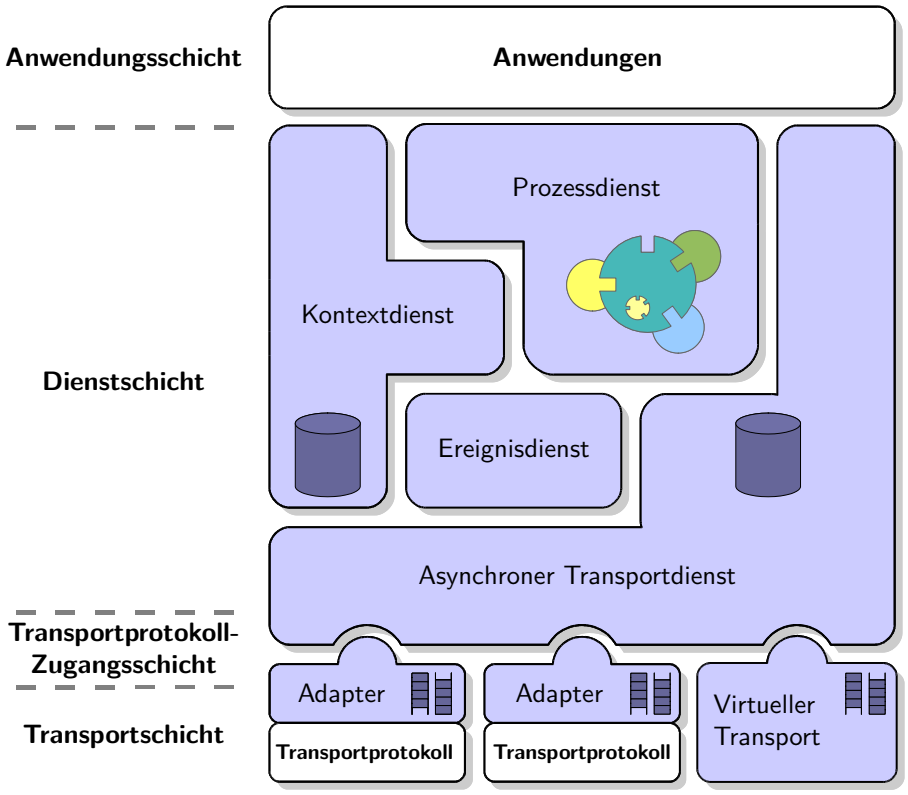


Abbildung 5.7: Architektur der DEMAC-Middleware (aus [KZL07b])

Der Prozessdienst hat innerhalb der DEMAC-Middleware die Aufgabe, die in DPDL formulierten mobilen Prozesse zu verwalten und auszuführen. Er ist in zwei grundlegende Module aufgeteilt: Das *Kernmodul* kann Prozessbeschreibungen empfangen, in einem persistenten Speicher ablegen und zu einem anderen Gerät migrieren, während das *Basismodul* die eigentliche Ausführung von Prozessen durchführt und daher die Workflow-Engine beinhaltet. Durch diese Trennung können auch leistungsschwache Geräte, die nur über das Kernmodul verfügen, die Ausführung von Prozessen initiieren, indem sie diese auf Geräte übertragen, welche die Ausführung fortführen können. Darüber hinaus ergänzen verschiedene Erweiterungsmodule den Prozessdienst um zusätzliche Funktionalitäten wie beispielsweise um die Transaktionsverarbeitung oder um eine Interaktionskomponente. [Zap05]

5.2.2 Vorbereitende Modifikationen des Prozessdienstes

Der Prozessdienst in DEMAC war ursprünglich nur für die sequentielle Bearbeitung von mobilen Prozessen ausgelegt. Während der Entwicklung der Erweiterung für die parallele Ausführung wurde festgestellt, dass die Architektur des Prozessdienstes in einigen Fällen Probleme bereitet. Es waren daher mehrere Änderungen des Prozessdienstes notwendig, von denen im Folgenden die beiden bedeutendsten vorgestellt werden.

Verknüpfung von zusätzlichen Daten mit einer Prozessbeschreibung

In Abschnitt 4.3.4 wurde dargelegt, dass es sinnvoll sein kann, die Zuordnung von Datenklassen in einer separaten Datei vorzunehmen. Insbesondere führt jedoch die Verwendung dieses Ansatzes innerhalb der DEMAC-Middleware die konsequente Modularisierung des Prozessdienstes fort. Sequentielle Abschnitte können somit auch von Geräten ausgeführt werden, die diese zusätzlichen Daten nicht interpretieren können, weil ihnen das Modul zur parallelen Ausführung fehlt. Es wurde daher eine flexible Erweiterung für den DEMAC-Prozessdienst entwickelt, die beliebige Daten mit einem Prozess verknüpft, diese im persistenten Speicher ablegt und bei der Migration eines Prozesses auf das neue Gerät überträgt. Die Interpretation und Verwendung der Daten erfolgt hingegen durch optionale Zusatzmodule.

Realisiert wird dies durch die Java-Klasse `Hashtable`. Hier können beliebige Objekte einem Schlüssel zugeordnet werden, der in diesem Fall eine Zeichenkette ist. Die Datenklassen eines Prozesses können beispielsweise über den Schlüssel `dataclasses` abgefragt werden. Der im Kernmodul des Prozessdienstes existierende Persistenzdienst wurde erweitert, so dass die serialisierte Hashtable im XML-Format in einer zusätzlichen Datei gespeichert und der Zugriff darauf über eine entsprechende Schnittstelle ermöglicht wird. Objekte, die als Erweiterung in der Hashtable abgelegt werden, müssen daher serialisierbar (`Serializable`) sein.

Neben dem Persistenzdienst wurde auch der Migrationsvorgang angepasst. Nachdem der Empfänger eines Prozesses zugestimmt hat, wurde hier bisher lediglich die Prozessbeschreibung im XML-Format übertragen. Nach der Modifikation wird nun eine XML-Datei weitergegeben, die aus zwei Elementen besteht: Die Prozessbeschreibung im XML-Tag „*dpdl*“ sowie die Erweiterungen im XML-Tag „*extensions*“.

Mit Hilfe der durchgeführten Modifikationen ist es nun also möglich, Datenklassen einem Prozess zuzuordnen und diese auch bei der Migration zu übermitteln. Aufgrund des generischen Ansatzes können ebenso beliebige Daten von anderen Zusatzmodulen des Prozessdienstes mit einem Prozess verknüpft werden.

Nebenläufiger Zugriff auf Prozesse

Der bereits erwähnte Persistenzdienst im Kernmodul des Prozessdienstes erlaubt das Auslesen von Prozessbeschreibungen aus dem persistenten Datenspeicher. Zur weiteren Verarbeitung existiert im Basismodul ein Interpreter, der aus dem XML-Format die objektorientierte Repräsentation des Prozesses erzeugt. Sollten Modifikationen eines Prozesses vorgenommen werden, so wurde bisher jedesmal mit Hilfe des Persistenzdienstes die XML-Beschreibung geladen und von dem Interpreter in Objekte überführt. Nach Abschluss der Änderungen wurde aus den Objekten wieder eine XML-Beschreibung generiert, die von dem Persistenzdienst festzuschreiben war.

Dieses Vorgehen birgt bei nebenläufigen Zugriffen auf denselben Prozess jedoch Probleme. Wird beispielsweise gerade von dem Prozessdienst ein Prozess P ausgeführt, so verwendet er eine objektorientierte Repräsentation von P. Währenddessen kann aber auf einem anderen Gerät, das einen parallelen Pfad von P ausführt, eine Konfliktauflösung erfolgen. Das aktuelle Gerät empfängt dann die Nachricht $\textsc{apply}(A_S \lessdot A_L)$ und muss diese Präzedenz festschreiben (vgl. Abschnitt 4.6.1). Wird nun wie oben beschrieben der Persistenzdienst und der Interpreter verwendet, um die Änderung durchzuführen, kann ein Datenverlust auftreten. Die Ursache dafür ist, dass gleichzeitig verschiedene Objekte desselben Prozesses existieren. Werden diese nun mit dem Persistenzdienst gespeichert, überschreiben sie gegenseitig ihre Änderungen.

Im Rahmen dieser Arbeit wurde daher eine neue Abstraktionsschicht über dem bestehenden Persistenzdienst eingeführt. Der `ProcessPersistenceService` kapselt im Basismodul die Zugriffe auf den Persistenzdienst des Kernmoduls und auf den Interpreter. Wird ein Prozess über den `ProcessPersistenceService` geladen, werden Persistenzdienst und Interpreter aufgerufen, um die objektorientierte Repräsentation des Prozesses zu erzeugen. Der `ProcessPersistenceService` speichert zusätzlich eine Referenz auf das erzeugte Objekt, um spätere Anfragen an denselben Prozess mit demselben Objekt beantworten zu können. Somit ist sichergestellt, dass gleichzeitig immer nur ein Objekt eines Prozesses existieren kann.

Damit nicht mehr benötigte Prozesse automatisch aus dem Arbeitsspeicher entfernt werden (*garbage collection*), verwendet der `ProcessPersistenceService` schwache Referenzen (*weak references*). Im Gegensatz zu starken Referenzen werden Objekte auch aus dem Speicher entfernt, wenn nur schwache Referenzen auf das Objekt verweisen [Sun06]. So ist sichergestellt, dass der begrenzte Arbeitsspeicher mobiler Geräte nicht unnötig belastet wird.

5.2.3 Adapter zwischen PAEX und DEMAC

Damit eine Workflow-Engine die in Abschnitt 5.1 vorgestellte Erweiterung zur Ausführung von parallelen Prozessen verwenden kann, müssen die Schnittstellen `AdapterFactory`, `ProcessAdapter` und `MessageSender` implementiert werden. Darüber hinaus muss die Workflow-Engine den `ParallelExecutor` an den entsprechenden Stellen im Kontrollfluss aufrufen. Da bestehende Workflow-Engines in der Regel diese Schnittstellen nicht zur Verfügung stellen, muss ein Adapter realisiert werden, der die Schnittstellen der Workflow-Engine auf die oben genannten Schnittstellen abbildet. Weiterhin sieht das Konzept zur parallelen Ausführung von mobilen Prozessen vor, den Entwickler bereits zur Modellierungszeit mit einem Werkzeug zur Erkennung von Abhängigkeitskonflikten zu unterstützen. Im Folgenden wird daher zunächst auf die Realisierung dieses Werkzeugs eingegangen und anschließend die Komponenten zur Ausführungszeit diskutiert.

Werkzeug zur Erkennung von Abhängigkeitskonflikten

Zur Modellierungszeit soll ein Werkzeug die Abhängigkeitskonflikte anzeigen können, die in einem konkreten Prozessmodell existieren. DEMAC verfügt über eine graphische Benutzeroberfläche, mit der unter anderem neue Prozesse zur Ausführung vorbereitet werden können. Diese Oberfläche wurde daher durch ein zusätzliches Eingabefeld für die Zuweisung von Datenklassen ergänzt. Über ein Interaktionselement kann die Erkennung von Abhängigkeitskonflikten unter Berücksichtigung der gewählten Datenklassen initiiert werden. Es wird nun aus der DPDL-Prozessbeschreibung mit Hilfe der Klasse `DPDLAdapter` (siehe weiter unten) die objektorientierte Repräsentation des mathematischen Prozessmodells (Klasse `Process`, vgl. Abschnitt 5.1.1) erzeugt und die dort angebotene Konflikterkennung aufgerufen. Der Modellierer erhält somit die Möglichkeit, den Prozess anhand der Ausgabe der Konflikterkennung weiter zu untersuchen und entsprechend zu modifizieren.

Komponenten zur Ausführungszeit

Der DEMAC-Prozessdienst erlaubt relativ einfach die Integration von zusätzlichen Modulen, die bestimmte Aufgaben bei der Ausführung eines Prozesses übernehmen. Der zu implementierende Adapter zwischen DEMAC und PAEX wird daher als Erweiterungsmodul `ConcurrencyService` realisiert. Er erbt dazu von der abstrakten Klasse `ExtensionModule`, die wiederum einen allgemeinen DEMAC-Dienst (`Service`) implementiert (vgl. Abbildung 5.8).

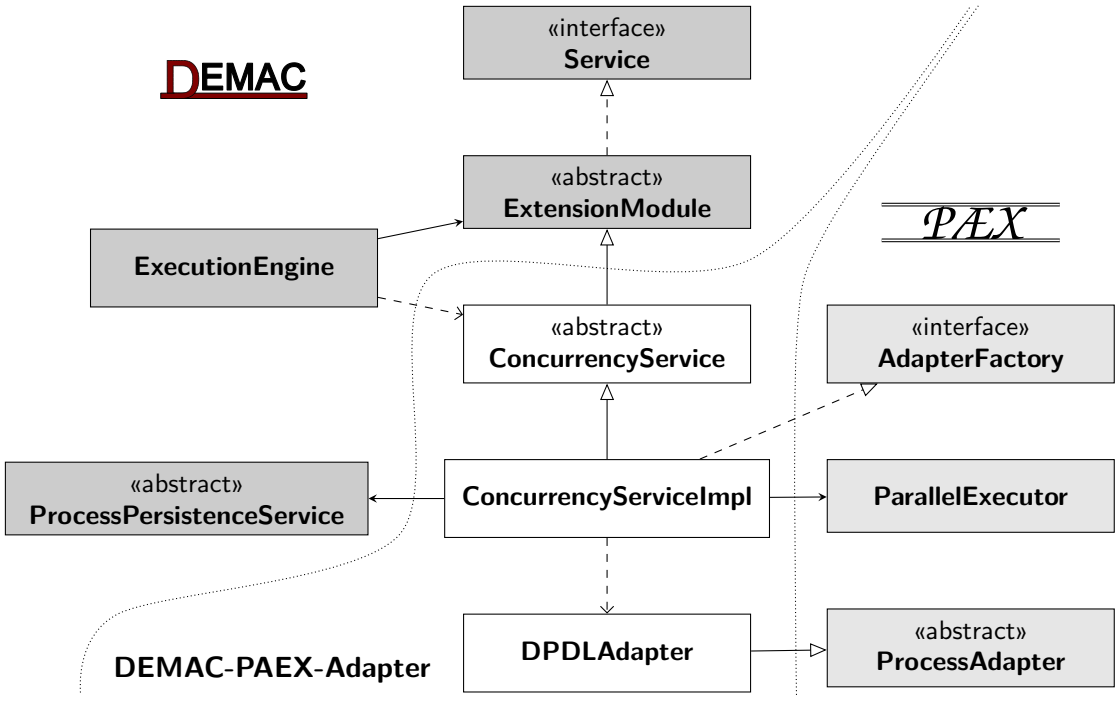


Abbildung 5.8: Realisierung des Adapters als DEMAC-Dienst

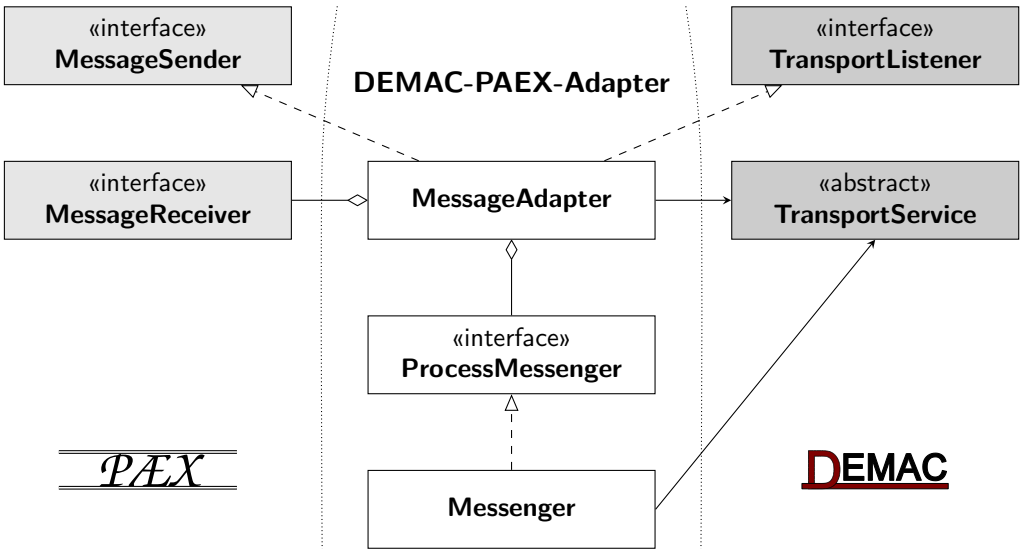


Abbildung 5.9: Realisierung des Adapters zum Nachrichtenaustausch

`ConcurrencyService` bietet Methoden zur Konfliktauflösung sowie zur Distribution und Synchronisation des Kontrollflusses an, die von dem Prozessdienst – konkret die `ExecutionEngine` – aufgerufen werden. Da die DEMAC-Architektur eine dynamische Bindung von Erweiterungsmodulen vorsieht, müssen Schnittstelle und Implementierung strikt getrennt werden. Die `ExecutionEngine` greift nur über die Schnittstelle `ConcurrencyService` auf das Erweiterungsmodul zu. Die eigentliche Implementierung liegt hingegen in der Klasse `ConcurrencyServiceImpl`, sofern sie auf dem konkreten Gerät verfügbar ist. Prinzipiell kann daher auch eine Implementierung verwendet werden, die nicht nach dem in dieser Arbeit vorgestelltem Konzept vorgeht.

Aufrufe von Methoden der Schnittstelle `ConcurrencyService` werden von `ConcurrencyServiceImpl` an den `ParallelExecutor` weitergegeben. Vorher muss jedoch die DPDL-Prozessbeschreibung in das mathematische Modell überführt werden. Dies realisiert die Klasse `DPDLAdapter`, welche durch die Implementierung der Schnittstelle `ProcessAdapter` auch die Rückkopplung zur DPDL-Prozessbeschreibung herstellt.

Zusätzlich übernimmt `ConcurrencyServiceImpl` gegenüber dem `ParallelExecutor` die Rolle der `AdapterFactory`, indem Prozesse vom `ProcessPersistenceService` geladen und mit Hilfe von `DPDLAdapter` als `ProcessAdapter` zur Verfügung gestellt werden.

Senden von Nachrichten Zur Konfliktauflösung muss die PAEX-Klasse `Serializable` mit anderen Geräten Nachrichten austauschen. Diese Nachrichten sind jedoch nicht an bestimmte Geräte adressiert sondern an Aktivitäten des auszuführenden Prozesses. Um das gesuchte Gerät zu finden, wurde in Abschnitt 4.5.1 vorgeschlagen, eine Anfrage per Rundruf auszuführen. Für den Fall, dass das gesuchte Gerät derzeit nicht erreichbar ist, wird das Senden der Anfrage entsprechend wiederholt. Da jedoch auch andere Strategien denkbar sind, wurde das konkrete Übertragungsverfahren hinter der

zusätzlichen Schnittstelle `ProcessMessenger` verborgen. Die Klasse `Messenger` realisiert diese Schnittstelle unter Verwendung der Rundruf-Strategie und des DEMAC-Transportdienstes (`TransportService`).

Um PAEX-Nachrichten zu versenden, wird die Schnittstelle `MessageSender` verwendet. Es wurde daher die Klasse `MessageAdapter` realisiert, die diese Schnittstelle implementiert und somit das Versenden von Nachrichten erlaubt (vgl. Abbildung 5.9). Eine konkrete PAEX-Nachricht wird dazu in ein generisches Format überführt und an einen `ProcessMessenger` weitergegeben. Der Empfang von Nachrichten wird durch die Implementierung der Schnittstelle `TransportListener` realisiert. `MessageAdapter` prüft anschließend, an welchen Prozess eine empfangene Nachricht adressiert ist, und ruft die Methode `receiveMessage` von dem entsprechenden `MessageReceiver` auf.

An dieser Stelle kann auf die Sicherheitskomponente des DEMAC-Transportdienstes zugegriffen werden, um die Authentizität der Nachrichten zu überprüfen. Damit ist sichergetellt, dass die Nachrichten von Angreifern nicht gefälscht wurden und so der korrekte Ablauf eines Prozesses nicht beeinflusst wird.

5.3 Realisierung und Bewertung

In diesem Abschnitt wird zunächst kurz auf technische Aspekte der prototypischen Implementierung eingegangen. Anschließend wird die Testumgebung beschrieben, die zum Überprüfen der korrekten Ausführung von Prozessen verwendet wurde. Das Kapitel schließt mit einer Bewertung der Ergebnisse der prototypischen Implementierung. Der Quellcode zum praktischen Teil dieser Arbeit befindet sich auf der beiliegenden CD-ROM.

5.3.1 Technische Aspekte der Implementierung

Als Rahmenbedingung für die praktische Umsetzung dieser Arbeit wurde festgelegt, dass die Implementierung im Rahmen des Projekts DEMAC erfolgt. Diese Vorgabe implizierte zugleich die Festlegung auf die Laufzeitumgebung Java ME.

Java ME (*Java Platform, Micro Edition*) ist eine Plattform zur Ausführung von in der Programmiersprache *Java* geschriebenen Programmen, die speziell an die Bedürfnisse von mobilen Geräten und eingebetteten Systemen (*embedded systems*), wie beispielsweise Mobiltelefone, Set-Top-Boxen oder Unterhaltungsgeräte, angepasst ist. Java-Programme werden nicht direkt in eine Maschinensprache übersetzt sondern in einen Bytecode, der von der entsprechenden Java-Plattform interpretiert wird und somit unabhängig von konkreten Hardware-Plattformen ist. Dies ist gerade im mobilen Bereich wichtig, da hier eine sehr hohe Heterogenität der verwendeten Hardware-Architekturen vorliegt. [Sun09]

Zwei sogenannte *Konfigurationen* stellen jeweils unterschiedliche Anforderungen an die Leistungsfähigkeit der Geräte. Die *Connected Limited Device Configuration* (*CLDC*) ist

für sehr leistungsschwache Geräte gedacht und stellt daher gegenüber der *Java Standard Edition* (*Java SE*) eine sehr stark eingeschränkte Laufzeitumgebung zur Verfügung. Demgegenüber besitzt die *Connected Device Configuration* (*CDC*) deutlich mehr Funktionalität, benötigt aber auch mehr Speicher und Prozessorleistung. Darüber hinaus ergänzen sogenannte *Profile* die Laufzeitumgebung um wichtige Anwendungsschnittstellen (*API*).

Aufgrund der ständigen Leistungssteigerung mobiler Geräte verliert CLDC an Bedeutung. Die *DEMAC*-Middleware verwendet daher CDC mit dem *Personal Profile* [Kun08]. Aus diesem Grund wurde auch für die prototypische Implementierung der parallelen Ausführung diese Plattform als Grundlage gewählt.

5.3.2 Testumgebung

Um zu verifizieren, dass die Ausführung paralleler Prozesse wie gefordert arbeitet, wurden im Rahmen dieser Arbeit verschiedene Testprozesse erzeugt und ausgeführt. Damit ein Test als funktional erfolgreich gewertet werden kann, müssen mehrere Eigenschaften erfüllt sein, die sich auf zwei wesentliche Aspekte zurückführen lassen:

Terminierung Ein Replikat terminiert, wenn sich alle Aktivitäten in einem der Zustände *finished*, *skipped* oder *expired* befinden. Dieser Zustand wird in endlicher Zeit nach dem Starten eines Prozesses erreicht. Wenn ein Replikat terminiert, existieren auf jedem Gerät höchstens zum Zwecke der Archivierung weitere Replikate dieser Prozessinstanz.

Korrektheit der Ausführung Die Ausführung jedes einzelnen parallelen Ausführungspfades erfolgt korrekt im Sinne der sequentiellen Ausführung, d. h. es werden beispielsweise zeitliche Beschränkungen und Transitionsbedingungen berücksichtigt. Weiterhin existiert eine serielle Ausführung des parallelen Prozesses, dessen berechnetes Ergebnis mit dem berechneten Ergebnis bei der Ausführung des Testprozesses identisch ist.

Der Aspekt der korrekten Terminierung ergibt sich erst aus der Verwendung von Replikaten für parallele Ausführungspfade. Hier muss sichergestellt werden, dass der Prozess nicht auf einem Replikat terminiert, obwohl andere Replikate beispielsweise auf die Durchführung einer Synchronisation warten.

Die Aktivitäten der Testprozesse führen mathematische Operationen aus, da hier die Korrektheit der Berechnungen sehr einfach überprüft werden kann. Außerdem werden Aktivitäten eingesetzt, die durch Ausgaben den aktuellen Zustand sichtbar machen. Ein Beispiel für einen einfachen Testprozess ist in Abbildung 5.10 dargestellt.[2] Im Folgenden wird der konkrete Ablauf der Ausführung dieses Prozesses näher betrachtet. Da der Prozess mit $x = 1$ initialisiert wird, werden korrekte Ausführungsfolgen im Sinne

2 Die vollständige in DPDL formulierte Prozessbeschreibung befindet sich auf der beiliegenden CD in der Datei `ParallelMath.xml`.

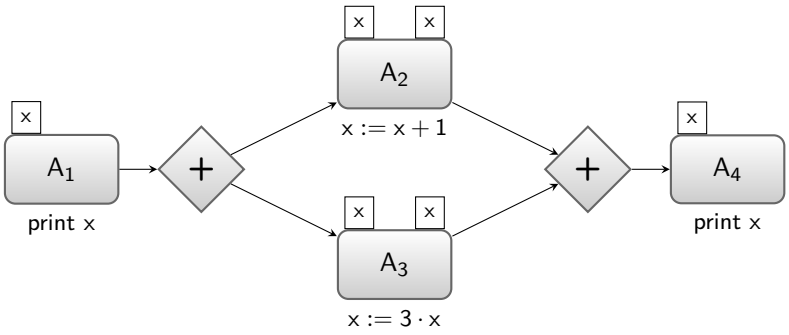


Abbildung 5.10: Beispiel eines Testprozesses

der Eine-Kopie-Serialisierbarkeit durch die Ergebnisse $x = 4$ und $x = 6$ sichtbar. Ausführungen unter vollständiger Vernachlässigung der Serialisierbarkeit führen zu den Ergebnissen $x = 2$ oder $x = 3$.

Mit Hilfe der graphischen Benutzeroberfläche von DEMAC kann der Prozess geladen und die Datenklassen angegeben werden. Das Ausführen der Konflikterkennung zeigt, dass die Variable x einen Abhängigkeitskonflikt zwischen den Aktivitäten A_2 und A_3 erzeugt. Durch die Angabe von `x=unsynchronized` wird der Konflikt während der Ausführung jedoch nicht berücksichtigt. Nach dem Start des Prozesses kann die Aktivität A_1 zunächst ohne die Erweiterung zur parallelen Ausführung bearbeitet werden. Erst bei dem folgenden *AND-Split* wird der `ConcurrencyService` zur Ausführung dieser Aufgabe aufgerufen. `ConcurrencyService` erzeugt nun ein zusätzliches Replikat, setzt dort A_3 als Startaktivität und übergibt das Replikat als neue Prozessbeschreibung dem DEMAC-Prozessdienst. Dieser kann nun unter Berücksichtigung von nicht-funktionalen Aspekten in der Umgebung ein geeignetes Gerät zur Ausführung des parallelen Pfades suchen. Auf dem ursprünglichen Replikat wird A_2 als Startaktivität gewählt.

Bevor A_2 jedoch in den Zustand *ready* versetzt werden kann wird sichergestellt, dass an dieser Stelle keine Abhängigkeitskonflikte die korrekte Ausführung behindern. Dazu prüft die `ExecutionEngine` zunächst, ob es gemäß Prozessmodell parallele Aktivitäten gibt. Nur wenn dies der Fall ist, wird `ConcurrencyService` beauftragt zu überprüfen, ob ein Abhängigkeitskonflikt bei der aktuellen Aktivität vorliegt. `ConcurrencyService` gibt die Anfrage an `ParallelExecutor` weiter, der mit Hilfe des mathematischen Modells die Konflikterkennung durchführt. An dem gefundenen Abhängigkeitskonflikt sind nur Variablen der Datenklasse „*unsynchronized*“ beteiligt, insofern wird die Methode `solveConflicts(...)` der Klasse `Unsynchronized` aufgerufen. Diese führt keine weiteren Operationen durch, und somit kann A_2 gestartet werden. Analog erfolgt die Aktivierung von A_3 auf dem anderen Replikat.

Sobald einer der beiden parallelen Ausführungspfade beendet ist, wird die Synchronisation initiiert. Die `ExecutionEngine` ruft daher über den `ConcurrencyService` den `ParallelExecutor` auf, der zunächst prüft, ob noch parallele Pfade zu synchronisieren sind. Da dies der Fall ist, werden alle lokal vorhandenen Prozessinstanzen auf zu syn-

chronisierende Pfade überprüft. Wenn das zu synchronisierende Replikat auf ein anderes Gerät migriert wurde, schlägt die Suche fehl und auch das lokale Replikat wird dem Prozessdienst zur Migration übergeben. Existieren hingegen auf einem Gerät zwei Replikate mit beendeten zu synchronisierenden Ausführungspfaden, führt `ProcessAdapter` die Synchronisation dieser Replikate durch. Das nun überflüssige zweite Replikat wird anschließend gelöscht, und es folgt die Ausführung von A_4 auf dem synchronisierten Replikat. Dies kann wieder auf einem Gerät erfolgen, auf dem PAEX nicht verfügbar ist.

Werden bei dem Start dieses Testprozesses keine Datenklassen angegeben, muss eine zusäzliche Synchronisation während der parallelen Ausführung stattfinden. `ParallelExecutor` ruft daher nicht die Klasse `Unsynchronized` sondern `Serializable` auf. Hier wird gemäß der Algorithmen aus Abschnitt 4.6.1 vorgegangen, um die Nachrichten zur Konfliktauflösung auszutauschen. Welche Präzedenz endgültig hinzugefügt wird, hängt davon ab, auf welchem Replikat die Konfliktauflösung zuerst gestartet wurde. Es ist somit nicht vorhersehbar, ob das Ergebnis $x = 4$ oder $x = 6$ berechnet wird.

Die Testprozesse wurden sowohl auf einem einzelnen Gerät als auch unter Verwendung mehrerer Geräte in einem lokalen Netz ausgeführt. Kurzzeitige Netzausfälle konnten dabei überbrückt werden. Bei längeren Netzausfällen zeigen sich jedoch die Schwächen des gewählten Ansatzes zur Zuordnung von Aktivitäten eines Prozesses zu Geräten. Überschreitet die Dauer des Netzausfalls eine bestimmte Grenze, wird das Senden der Nachricht abgebrochen und muss manuell wiederholt werden. Eine andere Strategie ist hier möglicherweise effektiver. Insgesamt konnten die Testausführungen jedoch als funktional erfolgreich im Sinne der anfangs gestellten Anforderungen gewertet werden.

5.3.3 Bewertung

Abschließend werden in diesem Abschnitt die erreichten Ergebnisse diskutiert. Dazu wird zunächst überprüft, ob die in Abschnitt 2.4.3 gestellten Anforderungen erfüllt wurden und in einer anschließenden Zusammenfassung die Flexibilität der gewählten Implementierungsform erörtert.

Überprüfung der gestellten Anforderungen

Im Rahmen der Problemanalyse wurden in Abschnitt 2.4.3 verschiedene Anforderungen und damit verbundene Fragestellungen formuliert, die in dieser Arbeit zu behandeln waren. Im Folgenden wird überprüft wie diese Aspekte in der Umsetzung berücksichtigt wurden.

Die offenen Fragestellungen zur *Auswahl der Teilnehmer einer parallelen Ausführung* wurden in dieser Arbeit konzeptionell diskutiert. Dazu wurden in Abschnitt 4.4.1 Kriterien erarbeitet, die von Geräten zu erfüllen sind, um an der parallelen Ausführung teilzunehmen. In DPDL können nicht-funktionale Aspekte als *Strategie* definiert werden. Diese müssen von dem Prozessdienst oder einer entsprechenden Erweiterung verarbeitet werden, um sie mit Hilfe des Kontextdienstes umzusetzen.

Bei der *Distribution von Daten* für die parallele Ausführung auf unterschiedlichen Geräten hat sich die vollständige Replikation eines Prozesses als geeignet erwiesen. Dass sich dabei die Replikate in unterschiedlichen Zuständen befinden können, wurde durch den Beweis in Abschnitt 4.7 gezeigt und durch die prototypische Implementierung bestätigt. Die Geräte benötigen somit keine dauerhafte Netzverbindung sondern können autark den entsprechenden parallelen Ausführungspfad bearbeiten. Durch die Erkennung und Auflösung von Abhängigkeitskonflikten mit Hilfe von PAEX wird dabei die notwendige *Nebenläufigkeitskontrolle* umgesetzt. Anforderungen bezüglich der Datenkonsistenz können durch die Zuordnung von Datenklassen realisiert werden. In der prototypischen Implementierung sind die Datenklassen *Unsynchronisiert* und *Serialisierbar* verfügbar.

Die *Synchronisation des Kontrollflusses* kann vollständig verteilt durch die Migration der beendeten Ausführungspfade erfolgen. Dabei werden die Daten der einzelnen Replikate durch den `ProcessAdapter` zusammengeführt. Die korrekte Vorgehensweise der prototypischen Implementierung wurde mit Hilfe von Testprozessen überprüft. Darüber hinaus kann der *Abbruch anderer paralleler Pfade* durch das Senden einer entsprechenden Nachricht realisiert werden.

Neben diesen elementaren Anforderungen waren auch die sich aus der Mobilität der Teilnehmer ergebenden Aspekte zu berücksichtigen. Dabei lässt sich zunächst feststellen, dass die verwendete dezentrale Koordination Migrationen angemessen einbezieht und wegen des optimistischen Ansatzes mit Verbindungsabbrüchen umgehen kann. Obwohl bei der Umsetzung komplexe Berechnungen gemieden wurden, besteht dennoch Optimierungspotenzial bei der Verwaltung mehrerer paralleler Ausführungspfade desselben Prozesses auf einem einzigen Gerät. Hier ist möglicherweise das Verwalten mehrerer Replikate nicht notwendig, wodurch Ressourcen gespart werden können.

Bewertende Zusammenfassung

Die prototypische Implementierung hat gezeigt, dass das entwickelte Konzept realisierbar ist und den gestellten Anforderungen genügt. Die Implementierung besteht dabei aus der generischen Komponente PAEX sowie einer Komponente, die konkrete Workflow-Engines mit PAEX verbindet. Ein solcher Adapter wurde für die DEMAC-Middleware realisiert.

Mit Hilfe der durchgeführten Implementierung können nun mobile Prozesse mit parallelen Pfaden innerhalb der DEMAC-Middleware ausgeführt werden. Die Distribution des Kontrollflusses wurde dabei innerhalb des DEMAC-PAEX-Adapters realisiert, da die Operationen direkt auf dem DPDL-Prozess arbeiten müssen. Darüber hinaus findet in PAEX keine Berücksichtigung von Kontextinformationen statt, weil die Workflow-Engine Migrationen durchführt und auf einen Kontextdienst zugreift, um dabei funktionale wie auch nicht-funktionale Anforderungen zu berücksichtigen. Während der

parallelen Ausführung werden Abhängigkeitskonflikte erkannt und unter Berücksichtigung der verwendeten Datenklassen aufgelöst.

Trotz Aufteilung der Implementierung in die zwei Module PAEX und PAEX-Adapter waren einige Änderungen in der existierenden Workflow-Engine notwendig. Die Ursache dafür ist, dass in der Architektur der Workflow-Engine gewisse Anwendungsfälle keine Berücksichtigung fanden, da sie zur Entwurfszeit keine Rolle spielten. Konkret betraf dies unter anderem den parallelen Zugriff auf die ausgeführten Prozesse sowie die fehlende Möglichkeit zur Verknüpfung eines Prozesses mit zusätzlichen Informationen. Obwohl sich die Erfahrungen im Projekt DEMAC nicht verallgemeinern lassen, ist es dennoch zulässig zu vermuten, dass die entdeckten Probleme auch in anderen Workflow-Engines zu lösen sind – es sei denn, Parallelität wurde dort bereits zur Entwurfszeit angemessen berücksichtigt.

Umfangreichere Anpassungen der Workflow-Engine sind in der Regel auch bei der Verarbeitung des Kontrollflusses notwendig. Hier müssen im Prozessmodell Verzweigung und Zusammenführung des Kontrollflusses zum Zwecke der parallelen Ausführung verarbeitet und die entsprechenden Methoden der Erweiterung PAEX aufgerufen werden. Dass an dieser Stelle gegebenenfalls tief in den Ablauf der Workflow-Engine eingegriffen werden muss, ist jedoch prinzipieller Natur, da die parallele Ausführung ein elementarer Bestandteil von Prozessen ist.

Bei der Implementierung wurde so viel Funktionalität wie möglich und sinnvoll in der generischen Komponente PAEX realisiert. Zur Integration der parallelen Ausführung in weitere Workflow-Engines müssen lediglich drei Schnittstellen implementiert sowie die Klasse zur Ausführungsunterstützung eingebunden werden. Die prototypische Implementierung hat daher gezeigt, dass das Konzept zur parallelen Ausführung von mobilen Prozessen in einer Workflow-Engine tatsächlich verwendbar ist.

Kapitel 6

Zusammenfassung und Ausblick

Parallelität wird in Workflow-Prozessen eingesetzt, um diese möglichst effizient auszuführen. Im Idealfall bearbeiten dabei mehrere Geräte gleichzeitig unterschiedliche parallele Ausführungspfade. Dadurch können die zur Verfügung stehenden Ressourcen besser genutzt und die Ausführungsdauer eines Prozesses verkürzt werden. Die Ausführung paralleler Prozesse ist daher insbesondere für mobile Geräte von großem Interesse, da diese inhärent leistungsschwächer als stationäre Geräte sind.

In dieser Arbeit wurde gezeigt, dass auch in mobilen Umgebungen parallele Prozesse auf mehreren mobilen Geräten ohne zentrale Koordinierungsinstanz effektiv ausgeführt werden können. Dazu wurde zunächst analysiert, welche Probleme und Fragestellungen auftreten, wenn zugleich die Besonderheiten mobiler Umgebungen als auch die Herausforderungen der Parallelität im Gegensatz zur sequentiellen Ausführung berücksichtigt werden müssen. Als größte Problemstellung wurde dabei die Synchronisation paralleler Ausführungspfade identifiziert.

Synchronisation beschreibt die Abstimmung von nebenläufigen Vorgängen aufeinander. Hierbei muss sichergestellt werden, dass die Ausführung eines Prozesses korrekt im Sinne der Serialisierbarkeit ist. In mobilen Umgebungen ist darüber hinaus der Einsatz von Replikation zur Erhöhung der Verfügbarkeit empfehlenswert. Bestehende Ansätze aus dem Bereich der Nebenläufigkeitskontrolle sind für mobile Umgebungen jedoch eher ungeeignet, da sie in der Regel nicht angemessen mit Verbindungsabbrüchen umgehen können. Es wurde daher eine Methode entwickelt, um Abhängigkeitskonflikte schon zur Entwicklungszeit erkennen zu können. Somit kann bereits im Voraus geprüft werden, an welchen Stellen im Prozessablauf die Synchronisation zwischen parallelen Ausführungspfaden notwendig ist.

Da nicht jeder Prozess serialisierbar ausgeführt werden muss, wurden Datenklassen eingeführt, mit denen anwendungsabhängig die zu erfüllenden Garantien bezüglich der Aktualität der verwendeten Daten festgelegt werden kann. Mit Hilfe dieser zusätzlichen Informationen kann auf einige Synchronisationen während der Ausführung verzichtet werden. Weiterhin existierende Abhängigkeitskonflikte müssen durch eine Nebenläufigkeitskontrolle aufgelöst werden. In dieser Arbeit wurde vorgeschlagen, die Konfliktauflösung durch die explizite Modifikation eines Prozesses durchzuführen. Dazu werden dem Prozess zusätzliche Präzedenzen hinzugefügt. Mit Hilfe eines optimistischen An-

satzes kann der Prozess trotz aufzulösender Abhängigkeitskonflikte auch bei Geräte- oder Netzausfällen fortgeführt werden.

Es wurde somit ein Verfahren zur Ausführung paralleler Prozesse entwickelt, das die Besonderheiten mobiler Geräte angemessen berücksichtigt. In einer prototypischen Implementierung wurde gezeigt, dass dieses Verfahren praktisch umsetzbar ist und sich in bestehende Workflow-Engines für mobile Prozesse integrieren lässt.

Zukünftige Arbeiten könnten die Verbesserung der parallelen Ausführung von iterativen Abschnitten aufgreifen. Da das für die parallele Ausführung verwendete mathematische Metamodell keine Zyklen erlaubt, müssen Schleifen als Blockkonstrukte realisiert werden. Lese-Schreib-Abhängigkeiten betreffen dann allerdings den ganzen Block. Dies führt zu einer sehr grob-granularen Betrachtungsweise, die in einigen Fällen eine ineffiziente Ausführung des Prozesses ergibt. Als möglicher Lösungsansatz könnten Schleifen daher schrittweise abgewickelt werden. Dazu müsste immer dann, wenn der Schleifenkörper betreten werden soll, eine Kopie von diesem vor dem Schleifenkonstrukt eingefügt werden. Soll die Schleife abgebrochen werden, würde das Schleifenkonstrukt entfernt. Auf diese Weise werden die iterativen Durchläufe einer Schleife explizit im Prozess verankert, ähnlich wie bei der Auflösung von Abhängigkeitskonflikten. Bei der Abwicklung einer Schleife sind jedoch deutlich umfangreichere Modifikationen des Prozesses notwendig. Die Korrektheit einer solchen Änderung muss daher gesondert überprüft werden.

Raum für weitere Arbeiten ergibt sich außerdem, wenn Transaktionen nicht nur auf der Ebene von Aktivitäten betrachtet werden. Im Rahmen der Herleitung der Korrektheitsbegriffe wurden Aktivitäten als Transaktionen aufgefasst. Auf einer höheren Ebene können Transaktionen jedoch auch mehrere Aktivitäten umfassen. Die parallele Ausführung dieser Transaktionen ist nur dann korrekt im Sinne der Eine-Kopie-Serialisierbarkeit, wenn es eine Serialisierung gibt, in der zwischen den einzelnen Aktivitäten einer Transaktion keine Aktivitäten paralleler Ausführungspfade vorkommen. Auch bei diesem Problem müssen daher die Lese- und Schreiboperationen der beteiligten Transaktionen untersucht werden. Möglicherweise kann auch hier mit Hilfe eines Abhängigkeitsgraphen eine geeignete Serialisierung gefunden oder der Bedarf zur Auflösung von Abhängigkeitskonflikten festgestellt werden. Somit ließe sich das Konzept zur parallelen Ausführung auch im Kontext von komplexeren Transaktionen anwenden.

Der Einsatz des entwickelten Konzepts ist prinzipiell auch in stationären verteilten Systemen denkbar. Dies ist insbesondere dann interessant, wenn auf einen zentralen Koordinator verzichtet werden soll. Problematisch ist der Einsatz jedoch in Umgebungen, in denen die Geräte eine sehr hohe Mobilität aufweisen und nach ihrem Verschwinden gar nicht oder erst sehr viel später wieder in Kommunikationsreichweite zu den übrigen Geräten kommen. Hier kann es passieren, dass parallele Pfade erst nach sehr langer Zeit oder gar nicht wieder zusammengeführt werden können. In solchen Umgebungen muss daher sichergestellt werden, dass der Prozess nur auf Geräte migriert, die regel-

mäßig miteinander in Verbindung stehen. Dazu wäre es nützlich, wenn Aussagen über den zukünftigen Kontext der Geräte getroffen werden können. Möglicherweise kann dies realisiert werden, indem die Kontexthistorie – also zu welchem Zeitpunkt welcher Kontext vorgelegen hat – analysiert und regelmäßige Ereignisse erkannt werden. Dafür müssen jedoch noch umfangreiche Forschungen durchgeführt werden.

Im Bereich des Mobile Computing im Allgemeinen aber auch bei der Ausführung von Prozessen auf mobilen Geräten im Speziellen sind also noch viele Problemstellungen ungelöst. Ungeachtet dessen nimmt die Bedeutung der mobilen Nutzung von Computern in Unternehmen und der gesamten Gesellschaft weiterhin zu. Mobilität muss also in der Informatik ein wichtiger Bestandteil der Forschung sein, da klassische Algorithmen und Vorgehensweisen oft nicht adäquat in mobilen Umgebungen anwendbar sind. Es werden statt dessen völlig neue Methoden benötigt, die eine schrittweise Näherung an die Vision des Ubiquitous Computing erlauben. Dabei werden voraussichtlich parallele prozessorientierte Abläufe auf mobilen Geräten eine große Rolle spielen. In der vorliegenden Arbeit konnte somit ein wichtiger Aspekt zur Realisierung dieser Vision beleuchtet werden.

Literaturverzeichnis

[ACKM04] Alonso, Gustavo ; Casati, Fabio ; Kuno, Harumi ; Machiraju, Vijay: *Web Services: Concepts, Architecture and Applications*. Berlin [u. a.] : Springer, 2004

[AGK+95] Alonso, Gustavo ; Günthör, Roger ; Kamath, Mohan ; Agrawal, Divyakant ; Abbadi, Amr E. ; Mohan, C.: Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. In: *Conference on Cooperative Information Systems*, 1995, 99–110

[AH02] Aalst, Wil M. P. d. ; Hee, Kees van: *Workflow Management: Models, Methods, and Systems*. Cambridge : MIT Press, 2002 (Cooperative Information Systems)

[AHD05] Aalst, Wil M. P. d. ; Hofstede, Arthur H. M. ; Dumas, Marlon: Patterns of Process Modeling. In: Dumas, Marlon (Hrsg.) ; Aalst, Wil M. P. d. (Hrsg.) ; Hofstede, Arthur H. M. (Hrsg.): *Process-Aware Information Systems*. Hoboken, New Jersey : Wiley-Interscience, 2005, S. 179–203

[AHKB03] Aalst, Wil M. P. d. ; Hofstede, Arthur H. M. ; Kiepuszewski, B. ; Barros, A. P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Juli, Nr. 1, S. 5–51

[All05] Allweyer, Thomas: *Geschäftsprozessmanagement: Strategie, Entwurf, Implementierung, Controlling*. Herdecke, Bochum : W3L, 2005

[BCCT05] Brambilla, Marco ; Ceri, Stefano ; Comai, Sara ; Tziviskou, Christina: Exception handling in workflow-driven Web applications. In: *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA : ACM, 2005, S. 170–179

[BD99] Bauer, Thomas ; Dadam, Peter: Verteilungsmodelle für Workflow-Management-Systeme – Klassifikation und Simulation. In: *Informatik – Forschung und Entwicklung* 14 (1999), Nr. 4, S. 203–217

[BG80] Bernstein, Philip A. ; Goodman, Nathan: Timestamp-based algorithms for concurrency control in distributed database systems. In: *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, VLDB Endowment, 1980, S. 285–300

[BG83] Bernstein, Philip A. ; Goodman, Nathan: The failure and recovery problem for replicated databases. In: *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM, 1983, S. 114–122

[BLFM05] Berners-Lee, Tim ; Fielding, Roy T. ; Masinter, Larry: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard). `http://www.ietf.org/rfc/rfc3986.txt`. Version: Januar 2005 (Request for Comments)

[BSR80] Bernstein, Philip A. ; Shipman, David W. ; Rothnie, James B. Jr.: Concurrency control in a system for distributed databases (SDD-1). In: *ACM Transactions on Database Systems* 5 (1980), März, Nr. 1, S. 18–51

[CDK02] Coulouris, George ; Dollimore, Jean ; Kindberg, Tim: *Verteilte Systeme: Konzepte und Design*. 3., überarbeitete Auflage. Pearson Studium, 2002

[CJK+08] Carbon, Ralf ; Johann, Gregor ; Keuler, Thorsten ; Muthig, Dirk ; Naab, Matthias ; Zilch, Stefan: Mobility in the virtual office: a document-centric workflow approach. In: *SAM '08: Proceedings of the 1st international workshop on Software architectures and mobility*. New York, NY, USA : ACM, 2008, S. 21–26

[Dad96] Dadam, Peter: *Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte und Realisierungsformen*. Berlin [u. a.] : Springer, 1996

[Dav84] Davidson, Susan B.: Optimism and consistency in partitioned distributed database systems. In: *ACM Transactions on Database Systems* 9 (1984), September, Nr. 3, S. 456–481

[DO03] Doval, Diego ; O'Mahony, Donal: Overlay Networks: A Scalable Alternative for P2P. In: *IEEE Internet Computing* 7 (2003), Nr. 4, S. 79–82

[DRD+00] Dix, Alan ; Rodden, Tom ; Davies, Nigel ; Trevor, Jonathan ; Friday, Adrian ; Palfreyman, Kevin: Exploiting space and location as a design framework for interactive mobile systems. In: *ACM Transactions on Computer-Human Interaction* 7 (2000), September, Nr. 3, S. 285–321

[Eck03] Eckert, Claudia: Mobil, aber sicher! In: [Mat03], S. 85–121

[EGLT76] Eswaran, Kapali P. ; Gray, Jim N. ; Lorie, Raymond A. ; Traiger, Irving L.: The notions of consistency and predicate locks in a database system. In: *Communications of the ACM* 19 (1976), November, Nr. 11, S. 624–633

[Eng93] Engesser, Herrmann (Hrsg.): *Duden Informatik*. 2., vollständig überarbeitete und erweiterte Auflage. Mannheim : Dudenverlag, 1993

[FCL97] Franklin, Michael J. ; Carey, Michael J. ; Livny, Miron: Transactional client-server cache consistency: alternatives and performance. In: *ACM Transactions on Database Systems* 22 (1997), September, Nr. 3, S. 315–363

[FPV98] Fuggetta, Alfonso ; Picco, Gian P. ; Vigna, Giovanni: Understanding Code Mobility. In: *IEEE Transactions on Software Engineering* 24 (1998), Mai, Nr. 5, S. 342–361

[FZ94] Forman, George H. ; Zahorjan, John: The Challenges of Mobile Computing. In: *IEEE Computer* 27 (1994), April, Nr. 4, S. 38–47

[GHJV03] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 5., korrigierter Nachdruck. München [u. a.] : AddisonWesley, 2003

[Gif79] Gifford, David K.: Weighted voting for replicated data. In: *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 1979, S. 150–162

[GMS87] Garcia-Molina, Hector ; Salem, Kenneth: Sagas. In: *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 1987, S. 249–259

[GR93] Gray, Jim ; Reuter, Andreas: *Transaction Processing: Concepts and Techniques*. San Mateo, California : Morgan Kaufmann Publishers, 1993

[HA90] Hutto, Phillip W. ; Ahamad, Mustaque: Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In: *Proceedings 10th International Conference on Distributed Computing Systems*, 1990, S. 302–309

[Haa03] Haase, Christoph: *Konzeption und Realisierung einer verteilten Workflow-Steuerung für mobile Endgeräte*, Universität Dortmund, Diplomarbeit, 2003. `http://ebus.informatik.uni-leipzig.de/www/media/lehre/diplomarbeiten/da-haase-pdf.pdf`, Abruf: 19. 08. 2008

[Hol95] Hollingsworth, David: *The Workflow Reference Model*. Version: 1995. `http://www.wfmc.org/standards/docs/tc003v11.pdf`, Abruf: 22. 06. 2008

[Hol07] Holbreich, Alexander: *Transaktionsunterstützung für verteilt ausgeführte mobile Prozesse*, Universität Hamburg, Diplomarbeit, 2007. `http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/435/AlexanderHolbreichDA.pdf`, Abruf: 09. 04. 2008

[HR01] Härder, Theo ; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Zweite, überarbeitete Auflage. Berlin [u. a.] : Springer, 2001

[HTKR+05] Höpfner, Hagen ; Türker, Can ; König-Ries, Birgitta u. a.: *Mobile Datenbanken und Informationssysteme: Konzepte und Techniken*. Heidelberg : dpunkt, 2005

[JBK98] Joseph, Anthony D. ; Badrinath, B. R. ; Katz, Randy H.: The case for services over cascaded networks. In: *WOWMOM '98: Proceedings of the 1st ACM international workshop on Wireless mobile multimedia*. New York, NY, USA : ACM, 1998, S. 2–10

[Kle96] Kleinrock, Leonard: Nomadicity: anytime, anywhere in a disconnected world. In: *Mobile Networks and Applications* 1 (1996), Dezember, Nr. 4, S. 351–357

[KR81] Kung, H. T. ; Robinson, John T.: On optimistic methods for concurrency control. In: *ACM Transactions on Database Systems* 6 (1981), Juni, Nr. 2, S. 213–226. – ISSN 0362–5915

[Kun08] Kunze, Christian P.: *Kontextbasierte Kooperation: Unterstützung verteilter Prozesse im Mobile Computing*, Universität Hamburg, Diss., April 2008

[KZL06] Kunze, Christian P. ; Zaplata, Sonja ; Lamersdorf, Winfried: Mobile Process Description and Execution. In: Eliassen, Frank (Hrsg.) ; Montresor, Alberto (Hrsg.): *Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, Springer, 2006, S. 32–47

[KZL07a] Kunze, Christian P. ; Zaplata, Sonja ; Lamersdorf, Winfried: Abstrakte Dienstklassen zur Realisierung mobiler Prozesse. In: Braun, Torsten (Hrsg.) ; Carle, Georg (Hrsg.) ; Stiller, Burkhard (Hrsg.) ; Gesellschaft für Informatik (Veranst.): *Konferenzband zur KiVS 2007 für Industrie-, Kurz- und Workshopbeiträge* Gesellschaft für Informatik, VDE Verlag, Februar 2007, S. 123–128

[KZL07b] Kunze, Christian P. ; Zaplata, Sonja ; Lamersdorf, Winfried: Mobile Processes: Enhancing Cooperation in Distributed Mobile Environments. In: *Journal of Computers* 2 (2007), Februar, Nr. 1, S. 1–11

[Lam79] Lamport, Leslie: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Progranm. In: *IEEE Transactions on Computers* 28 (1979), September, Nr. 9, S. 690–691

[Ley95] Leymann, Frank: Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In: *Datenbanksysteme in Büro, Technik und Wirtschaft: GI-Fachtagung, Dresden, 22.–24. März 1995*. Berlin [u. a.] : Springer, 1995 (Informatik aktuell), S. 51–70

[Ley06] Leymann, Frank: Workflow-Based Coordination and Cooperation in a Service World. In: *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE* Bd. 4275/2006. Berlin [u. a.] : Springer, 2006, S. 2–16

[Lie07] Liebhart, Daniel: *SOA goes real*. München, Wien : Hanser Fachbuchverlag, 2007

[LR00] Leymann, Frank ; Roller, Dieter: *Production workflow: concepts and techniques*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2000

[LSKM00] Luo, Zongwei ; Sheth, Amit ; Kochut, Krys ; Miller, John: Exception Handling in Workflow Systems. In: *Applied Intelligence* 13 (2000), Nr. 2, S. 125–147

[M+08] Melzer, Ingo u. a.: *Service-orientierte Architekturen mit Web Services: Konzepte, Standards, Praxis*. 3. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2008

[Mac02] Mackensen, Elke: Drahtlose Datenkommunikation für intelligente, autarke Mikrosysteme. In: *2. GMM-Workshop Energieautarke Sensorik*, 2002

[Mat03] Mattern, Friedemann (Hrsg.): *Total vernetzt: Szenarien einer informatisierten Welt*. Berlin [u. a.] : Springer, 2003 (Xpert.press)

[Mos81] Moss, J. Eliot B.: *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA, USA, Massachusetts Institute of Technology, Diss., 1981

[MS04] Mutschler, Bela ; Specht, Günther: *Mobile Datenbanksysteme: Architektur, Implementierung, Konzepte*. Berlin [u. a.] : Springer, 2004 (Xpert.press)

[Pap08] Papazoglou, Michael P.: *Web Services: Principles and Technology*. Harlow, England [u. a.] : Pearson Education Limited, Prentice Hall, 2008

[Per06] Pernici, Barbara: Basic Concepts. In: Pernici, Barbara (Hrsg.): *Mobile Information Systems: Infrastructure and Design for Adaptivity and Flexibility*. Berlin [u. a.] : Springer, 2006, S. 3–23

[PST+97] Petersen, Karin ; Spreitzer, Mike J. ; Terry, Douglas B. ; Theimer, Marvin M. ; Demers, Alan J.: Flexible update propagation for weakly consistent replication. In: *ACM SIGOPS Operating Systems Review* 31 (1997), Nr. 5, S. 288–301

[PTDL07] Papazoglou, Michael P. ; Traverso, Paolo ; Dustdar, Schahram ; Leymann, Frank: Service-Oriented Computing: State of the Art and Research Challenges. In: *IEEE Computer* 40 (2007), November, Nr. 11, S. 38–45

[Puu01] Puustjärvi, Juha: Workflow Concurrency Control. In: *The Computer Journal* 44 (2001), Nr. 1, S. 42–53

[RBB03] Rothermel, Kurt ; Bauer, Martin ; Becker, Christian: Digitale Weltmodelle – Grundlagen kontextbezogener Systeme. In: [Mat03], S. 123–141

[RHAM06] Russell, Nick ; Hofstede, Arthur H. M. ; Aalst, Wil M. P. d. ; Mulyar, Nataliya: Workflow Control-Flow Patterns: A Revised View / BPMcenter.org. 2006 (BPM-06-22). – BPM Center Report

[RHEA04] Russell, Nick ; Hofstede, Arthur H. M. ; Edmond, David ; Aalst, Wil M. P. d.: Workflow Data Patterns / Queensland University of Technology. Brisbane, 2004 (FIT-TR-2004-01). – QUT Technical report

[Rot05] Roth, Jörg: *Mobile Computing: Grundlagen, Technik, Konzepte*. 2., aktualisierte Auflage. Heidelberg : dpunkt, 2005

[Sat96] Satyanarayanan, Mahadev: Fundamental challenges in mobile computing. In: *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM, 1996, S. 1–7

[Sat01] Satyanarayanan, Mahadev: Pervasive computing: vision and challenges. In: *IEEE Personal Communications* 8 (2001), Nr. 4, S. 10–17

[Sau08] Sauter, Martin: *Grundkurs mobile Kommunikationssysteme: von UMTS, GSM und GPRS zu Wireless LAN und Bluetooth-Piconetzen*. 3., erweiterte Auflage. Wiesbaden : Vieweg, 2008

[SAW95] Schilit, Bill ; Adams, Norman ; Want, Roy: Context-Aware Computing Applications. In: *IEEE Workshop on Mobile Computing Systems and Applications*. Los Alamitos, Calif. [u. a.] : IEEE Computer Society Press, 1995

[SO04] Schulz, Karsten A. ; Orlowska, Maria E.: Facilitating Cross-Organisational Workflows with a Workflow View Approach. In: *Data & Knowledge Engineering* 51 (2004), Oktober, Nr. 1, S. 109–147

[Sun06] *Personal Profile 1.1.2 (JSR 216)*. Version: 2006. `http://java.sun.com/javame/reference/apis/jsr216/`, Abruf: 17. 02. 2009. Sun Microsystems, Inc.

[Sun09] *Java ME Platform Overview*. Version: 2009. `http://java.sun.com/javame/technology/index.jsp`, Abruf: 06. 02. 2009. Sun Microsystems, Inc.

[Tan03] Tanenbaum, Andrew S.: *Computernetzwerke*. 4., überarbeitete Auflage. München : Pearson Studium, 2003

[TDP+94] Terry, Douglas B. ; Demers, Alan J. ; Petersen, Karin ; Spreitzer, Mike ; Theimer, Marvin ; Welch, Brent W.: Session Guarantees for Weakly Consistent Replicated Data. In: *PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. Washington, DC, USA : IEEE Computer Society, 1994, S. 140–149

[TS07] Tanenbaum, Andrew S. ; Steen, Maarten van: *Distributed Systems: Principles and Paradigms*. Second Edition. Upper Saddle River, NJ, USA : Prentice Hall, 2007

[TTP+95] Terry, Douglas B. ; Theimer, Marvin M. ; Petersen, Karin ; Demers, Alan J. ; Spreitzer, Mike J. ; Hauser, Carl H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 1995, S. 172–182

[Val05] Valk, Rüdiger: Parallelität und Nebenläufigkeit / Universität Hamburg. Hamburg, 2005. – Prüfungsunterlagen zur Vorlesung F4

[Wei91] Weiser, Mark: The Computer for the 21st Century. In: *Scientific American* 265 (1991), September, S. 66–75

[Wes07] Weske, Mathias: *Business Process Management: Concepts, Languages, Architectures*. Berlin [u. a.] : Springer, 2007

[Win07] Winnicki, Alice: *Erweiterung einer Middleware für das Mobile Computing um nicht-funktionale Sicherheits- und Vertrauensaspekte*, Universität Hamburg, Diplomarbeit, 2007. `http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/439/DA_AliceWinnicki.pdf`, Abruf: 17. 02. 2009

[WK01] Wang, Helen J. ; Katz, Randy H.: Mobility support in unified communication networks. In: *WOWMOM '01: Proceedings of the 4th ACM international workshop on Wireless mobile multimedia*. New York, NY, USA : ACM, 2001, S. 95–102

[YV02] Yu, Haifeng ; Vahdat, Amin: Design and evaluation of a conit-based continuous consistency model for replicated services. In: *ACM Transactions on Computer Systems* 20 (2002), August, Nr. 3, S. 239–282

[Zap05] Zaplata, Sonja: *Prozessintegration in Middleware für mobile Systeme*, Universität Hamburg, Diplomarbeit, 2005. `http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/305/Diplomarbeit_ZA_041005.pdf`, Abruf: 09. 04. 2008

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, den 27. Februar 2009

Kristof Hamann