Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Verteilte Systeme und Informationssysteme

Diplomarbeit

Prozessintegration in Middleware für mobile Systeme

Hamburg, 04. Oktober 2005

Sonja Zaplata

0zaplata@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr.: 5299733
Fachsemester 11

Erstgutachter: Prof. Dr. W. Lamersdorf
Zweitgutachter: Dr. G. Gryczan

Zusammenfassung

Die technische Entwicklung im Bereich mobiler Systeme schreitet schnell voran. Dennoch sind mobile Systeme im Vergleich zu stationären Arbeitsplatzrechnern und statischen Infrastrukturen inhärent leistungsschwach und fehleranfällig. Durch die Möglichkeit einer Zusammenarbeit mobiler Einheiten können viele dieser Defizite ausgeglichen und auch komplexere Aufgaben in Form von Prozessen bearbeitet und ausgeführt werden. Dazu wird eine Prozessbeschreibungssprache benötigt, die die verteilte Administration und eine flexible Ausführung von Prozessen unterstützt und dabei die besonderen Leistungsmerkmale der mobilen Systeme berücksichtigt.

Diese Arbeit untersucht bestehende Prozessbeschreibungssprachen darauf, ob sie die speziellen Anforderungen für den Einsatz innerhalb einer Middleware für mobile Systeme erfüllen können. Es wird zudem ein eigener Ansatz zur Definition einer Prozessbeschreibungsprache für mobile Systeme und die prototypische Implementierung einer geeigneten Ausführungsumgebung vorgestellt.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel wird zunächst motiviert, warum eine Integration von Prozessen in Middleware für mobile Systeme benötigt wird. Im Folgenden werden die genaue Zielsetzung der Arbeit vorgestellt und deren Gliederung erläutert.

1.1 Motivation

Nach einer aktuellen Studie des Marktforschungsinstituts IDC sind im Bereich des Mobile Computings immer noch hohe Wachstumsraten zu verzeichnen. So sind bis zum Jahr 2008 durchschnittlich jährlich 24,2% Wachstum für den Markt von mobiler Middleware für Mobiltelefone und PDAs prognostiziert [IDC04]. Mobile Anwendungen beschränken sich nicht mehr nur auf das einfache Telefonieren oder die Verwaltung von Terminen oder Adressdaten. Vielmehr werden sie zunehmend auch für die Ausführung von komplexeren Aufgaben und Diensten verwendet, die der Benutzer von seinem täglichen Leben im Umgang mit Arbeitsplatzrechnern gewohnt ist. E-Mail, Internet und E-Business sowie die Verwendung von speziellen mobilen Anwendungsprogrammen seien hier als Beispiele genannt.

Der erfolgreichen Bearbeitung von vielfältigen und komplexen Aufgaben stehen allerdings relativ beschränkte technische Möglichkeiten im Mobile Computing gegenüber. Die Leistungsfähigkeit mobiler Geräte verbessert sich zwar kontinuierlich, wird jedoch im Verhältnis zu dem Potential stationärer Rechner immer einen Schritt zurückbleiben [Sat96]. Die gleiche Situation zeigt sich bei der Betrachtung mobiler Kommunikation: Die Datenübertragung über verschiedene drahtlose Netze steht schon an vielen Orten zur Verfügung, ist aber im Vergleich zu kabelgebundenen Kommunikationsmedien eher leistungsschwach. Den wachsenden Anforderungen der Benutzer an die mobile Infrastruktur kann daher nur durch eine enge Zusammenarbeit der mobilen Geräte untereinander und eine Integration in be-

stehende stationäre Systeme Rechnung getragen werden. Mit steigendem Wachstum der Gerätezahlen im mobilen Bereich und damit einer zunehmenden Dichte an potentiellen Dienstleistern ergibt sich die Möglichkeit, auf dem eigenen Gerät nicht vorhandene Dienste aus der unmittelbaren Umgebung fremdzubeziehen. Einzelne einfache Aufgaben können an fremde Dienstanbieter ausgelagert werden, um damit die Beschränktheit des eigenen Geräts zu kompensieren oder dessen Ressourcen für andere Zwecke einzusetzen.

Es lassen sich jedoch nicht nur einzelne Funktionen, sondern auch komplexe Sequenzen von Aufgaben delegieren. Langfristiges Ziel soll es sein, auch höherwertige Prozesse auf mobilen Systemen automatisch ausführbar zu machen [Mat03]. Dieses erfordert zum einen eine plattformunabhängige Beschreibungssprache zur Definition der auszuführenden Aufgaben als auch eine für mobile Geräte geeignete Ausführungskomponente, die in der Lage ist, die Prozessbeschreibungen zu interpretieren und Aufgaben an passende Dienste zuzuweisen.

Das Einbeziehen völlig fremder, spontan, sporadisch und eventuell einmalig in der Umgebung auftretender Geräte muss jedoch auch dem Benutzer den Freiraum lassen, gewünschte Servicequalitäten der Dienstteilnehmer spezifizieren zu können. So können zum Beispiel Anforderungen zur Verbindungsqualität, zu anfallenden Kosten oder zu Sicherheitsaspekten über eine Zusammenarbeit mobiler Systeme entscheiden.

In der Telekommunikation existieren zur Zeit viele technisch ausgereifte Einzelprodukte, deren Wert für den Benutzer durch Integration und Vernetzung wesentlich erhöht werden kann [Wyb03]. Insbesondere bei der Ausführung von Geschäftsprozessen und speziell bei dem Einsatz von Außendienstmitarbeitern kann von einer derartigen Infrastruktur profitiert werden. So können Ressourcen für aufwändige Arbeitsschritte eingespart werden, Fehlerquellen entfallen und Kosten für manuelle Bearbeitungen gesenkt werden. Mobile Technologien können so bei einer gezielten Integration in die Arbeitsabläufe zur Produktivitäts- und Qualitätssteigerung von Unternehmen beitragen [SGSP04].

Das Vorhaben, schwache mobile Systeme durch die Möglichkeit der Zusammenarbeit zu einem insgesamt leistungsfähigen Netz zu verbinden, ist daher Thema dieser Arbeit. Es werden eine Prozessbeschreibungssprache und ein System zur Prozessverarbeitung benötigt, die gemeinsam den speziellen Anforderungen mobiler Systeme gerecht werden.

1.2 Zielsetzung

Diese Arbeit soll einen Beitrag zum Forschungsprojekt „Distributed Environment for Mobility Aware Computing“ (im folgenden kurz DEMAC) des Arbeitsbereichs VSIS am Fachbereich Informatik der Universität Hamburg leisten.

Zielsetzung ist es, ein für mobile Geräte adäquates Konzept zur Integration von Prozessen zu finden, das es erlaubt, Dienste verschiedener Systeme zielorientiert zu komponieren. Dabei müssen insbesondere die speziellen Eigenschaften mobiler Systeme berücksichtigt werden. Außerdem müssen wichtige von Benutzern geforderte nicht-funktionale Aspekte zur Ausführung der Prozesse konzeptuell einfließen.

Als Koordinationsmechanismus ist der Einsatz einer Beschreibungssprache zur Komposition verteilter Dienste vorgesehen. Dazu werden bereits existierende Standards und Sprachen für Prozessbeschreibungen untersucht und aus den Erkenntnissen ein passendes Schema zur Unterstützung kollaborativer Prozesse von mobilen Komponenten abgeleitet.

Die Umsetzung des Prozessmodells umfasst die Definition einer geeigneten Prozessbeschreibungssprache und die Implementierung einer entsprechenden lokalen Ausführungsumgebung zur Abarbeitung konkreter Ablaufpläne auf mobilen Systemen.

1.3 Vorgehensweise

Nach dieser Einleitung wird der Themenbereich der Diplomarbeit vorgestellt. Es werden die dem Vorhaben zugrunde liegende Servicearchitektur und das Konzept der Context Awareness präsentiert und das Thema in das DEMAC-Projekt eingeordnet. Ein Anwendungsbeispiel soll die beschriebenen Zusammenhänge verdeutlichen und Herausforderungen aufzeigen.

Im nächsten Kapitel wird der benötigte Hintergrund für eine Prozessintegration im Bereich mobiler Systeme dargelegt. Hierzu gehören die Beschreibung der spezifischen Eigenschaften mobiler Systeme und der damit einhergehenden Einschränkungen. Grundlagen des Prozessbegriffs und der Prozessbeschreibung sowie die Definition der in diesem Zusammenhang relevanten nicht-funktionalen Aspekte werden erläutert. Ziel ist die Identifikation der daraus resultierenden Anforderungen an die zu erstellende Beschreibungssprache und die Zusammenfassung und Gewichtung der wichtigsten Aspekte in einem Anforderungskatalog.

Im vierten Kapitel wird analysiert, inwieweit bestehende Konzepte zur Koordination heterogener Komponenten in einem mobilen System eingesetzt werden können. Verschiedene aktuelle Workflowbeschreibungssprachen werden daraufhin untersucht, ob sie als Ganzes oder in Auszügen zur Formulierung von Prozessen im beschriebenen Anwendungskontext geeignet sind. Dabei kommt es insbesondere darauf an, den erarbeiteten Anforderungen zu genügen.

Mit Hilfe der im vierten Kapitel gewonnenen Erkenntnisse wird im folgenden eine für das DEMAC-Projekt geeignete Prozessbeschreibungssprache definiert. Anforderungen, die vom gewählten Ansatz nicht erfüllt werden, werden durch ergänzende eigene Konzepte berücksichtigt. Neben der Spezifikation der Beschreibungssprache wird außerdem ein Prozessmanagementsystem zur Ausführung der generierten Ablaufpläne auf den mobilen Geräten vorgestellt. Die Ergebnisse werden in Auszügen am Anwendungsbeispiel präsentiert.

Zum Schluss der Arbeit werden die Erkenntnisse zusammengefasst und Einsatzmöglichkeiten sowie potentielle Erweiterungen diskutiert.

2 Einführung in den Themenbereich

Die Integration von Prozessen in Middleware für mobile Systeme wird im Umfeld des DEMAC-Projektes realisiert. In diesem Kapitel werden daher die für diesen Themenbereich relevanten Umstände und Konzepte erläutert.

2.1 Middleware für mobile Systeme

Middleware-Systeme für klassische stationäre verteilte Systeme sollen von den konkreten Gegebenheiten zugrunde liegender Betriebssysteme, Netzwerke und Protokolle abstrahieren, dort auftretende Fehler behandeln und gegebenenfalls sogar die Verteilung und die Heterogenität interagierender Subsysteme vor der Anwendungsebene und dem Benutzer verbergen. Sie sind daher auf einer logischen Ebene zwischen den einzelnen Anwendungskomponenten eines verteilten Systems und der Netzwerkschicht angesiedelt [Emm03]. Neben den unterstützten Protokollen ist eine Middleware durch ihre *Application Programming Interfaces* (*APIs*) definiert [Ber96]. Die Architektur einer Middleware besteht dazu aus einer Menge von Diensten, die aufsetzenden Anwendungsprogrammen Schnittstellen zur Verfügung stellen, um die zugrunde liegende Komplexität der Infrastruktur für den Benutzer oder den Anwendungsentwickler zu reduzieren [CDK02].

Im Gegensatz zu den stationären verteilten Systemen zeichnen sich mobile verteilte Systeme durch ihre Mobilität, durch nicht-permanente Netzwerkverbindungen und durch dynamische Kontexte aus [CEM02]. Im Fall mobiler Systeme tritt an die Stelle der genannten Verteilungstransparenz daher ein Bewusstsein über die Mobilität und eine sich ändernde Umgebung (vgl. 2.3). Die Middleware liefert hier die relevanten Umgebungsdaten und stellt sie den aufsetzenden Anwendungsprogrammen zur weiteren Verarbeitung zur Verfügung. In Zusammenhang mit Servicearchitekturen (vgl. 2.2) sind ins-

besondere Kenntnisse über Kosten, Sicherheitsmechanismen und Übertragungsgeschwindigkeiten bedeutend [Box04].

Hinzu kommt, dass eine Middleware für mobile Systeme in der Regel nicht die gleichen Qualitätsmerkmale zur Verfügung stellen kann, wie eine entsprechende Software für stationäre Systeme. Dieses ist insbesondere deshalb der Fall, weil mobile Systeme nur mit begrenzten Ressourcen und einer eingeschränkten Rechenleistung ausgestattet sind. Nicht-permanente Verbindungen machen zudem eine Konzentration auf asynchrone Kommunikation notwendig, um nicht von einer in vielen Fällen unrealisierbaren simultanen Verbindung von Sender und Empfänger abhängig zu sein [CEM02].

Abbildung 1 zeigt die wichtigsten Eigenschaften mobiler verteilter Systeme im Vergleich mit klassischen stationären Systemen im Überblick.

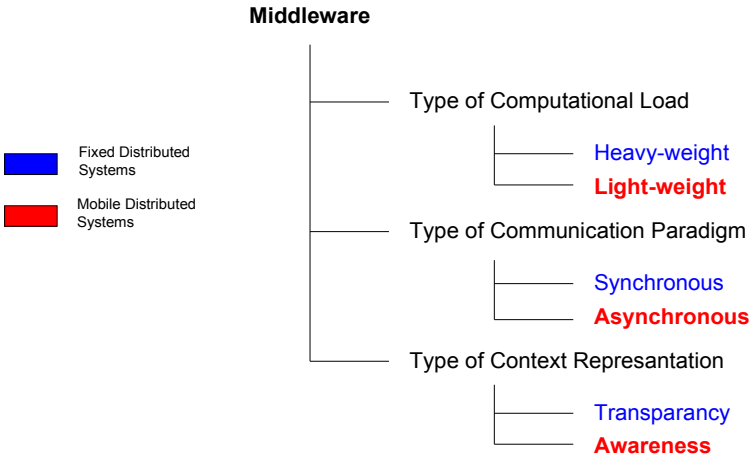


Abb. 1: Eigenschaften von verteilten Systemen (nach [CEM02])

Eine Middleware für mobile Systeme muss die Beschränktheit mobiler Geräte in Bezug auf ihre Leistungsfähigkeit, die Gefahr von Verbindungsabbrüchen und die Variabilität der Umgebung berücksichtigen. Dazu müssen Beschränkungen an der Performanz in Kauf genommen und insbesondere asynchrone Kommunikationsaktivitäten unterstützt werden. Die Auswertung und Weitergabe von Umgebungsinformationen an die Anwendungsschicht stellt eine weitere zentrale Aufgabe mobiler Middleware dar.

2.2 Service-Architekturen für verteilte Systeme

Der Begriff der *Service-Oriented-Architecture (SOA)* bezeichnet ein Architektur-Paradigma für verteilte Systeme, in der eine Menge von aufrufbaren Komponenten auf der Basis von Standards und Protokollen Funktionalitäten bereitstellt. Diese Komponenten werden durch wiederverwendbare und voneinander unabhängige Dienste (Services) realisiert, deren Schnittstellenbeschreibungen veröffentlicht

und abgerufen werden können [PaGe03]. Implementierungsdetails werden dabei gekapselt und bleiben dem Dienstaufrufer gemäß des Geheimnisprinzips verborgen. Ziel des Ansatzes ist eine lose Kopplung zwischen den einzelnen Komponenten [BHM+05].

Als besondere Ausprägung der *Service Oriented Architecture* sind *Web Services* bekannt. *Web Services* erlauben das maschinelle Auffinden und Nutzen von Services über Internetprotokolle. Bei den Diensten handelt es sich in der Regel um Softwaremodule, die eine spezielle Anwendungslogik implementieren und zur Integration und Komposition komplexerer Anwendungen genutzt werden können [PaGe03].

Für die Integration einer Anwendung können nach Eberhard und Fischer [EbFi03] drei Rollen definiert werden (Abb. 3):

Ein *Diensterbringer (Service Provider)* implementiert seinen Dienst in einer beliebigen Programmiersprache und versieht ihn mit einer XML-basierten Beschreibung, der sogenannten *Web Service Description (WSD)*. Als Beschreibungssprache für Dienste hat sich die *Web Service Description Language (WSDL)* als Defacto-Standard etabliert, die unter anderem den Dienstnamen, Nachrichten zur Dienstverwendung, Transportprotokolle und die Adresse, an denen der Dienst zur Verfügung steht, enthält [CCMW01].

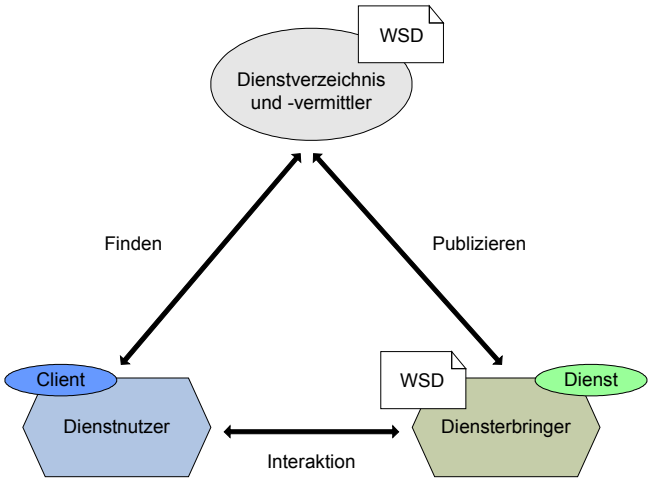


Abb. 2: Das Web Service Rollenmodell (aus [EbFi03])

Die zweite Rolle innerhalb der *Service-Oriented-Architecture* übernimmt der *Dienstnutzer (Service Requestor)*, der mit dem *Diensterbringer* interagiert, um eine bestimmte Funktionalität zu beziehen. Dabei kann es sich sowohl um einen direkten Request eines Internetnutzers über einen Browser als auch um eine Anfrage eines anderen Dienstes handeln, der die in einem *Web Service* implementierte Funktionalität weiterverarbeitet. Die Kommunikation bei der Suche und der Nutzung wird in der Regel über das SOAP-Protokoll abgewickelt. Zum Einbinden des fremden Services benötigt der potentielle *Dienstnutzer* die *WSD* des Dienstes. Diese erhält er im einfachsten Fall durch eine direkte Kommunikation mit dem Diensterbringer [EbFi03].

Ist dem *Dienstnutzer* der passende *Diensterbringer* nicht bekannt, so kann er eine Anfrage an ein *Dienstverzeichnis (Service Registry)* stellen. In einem *Dienstverzeichnis* können *Diensterbringer* In-

formationen über sich und ihre Dienste publizieren. Als globaler Verzeichnisdienst hat sich *UDDI (Universal Description, Discovery and Integration)* etabliert, welcher die Basis für die Suche und Klassifizierung von *Web Services* schafft. *UDDI* dient gleichzeitig auch der strukturierten Beschreibung von Unternehmen, deren Beziehungen untereinander, ihrer Dienste sowie weiterer Metadaten. Neben dem passiven *Dienstverzeichnis* kann auch ein aktiver *Vermittler (Broker)* zur Kontaktaufnahme zwischen *Dienstnutzer* und *Diensterbringer* genutzt werden [EbFi03].

Die Modularität und die wohldefinierte Schnittstelle zum Aufruf der bereitgestellten Funktionalität erlauben die Zusammensetzung einfacher Services zu erweiterten kombinierten Diensten. Man unterscheidet hierbei zwischen *Service-Aggregation* und *Service-Komposition* [EOH02]:

Service Aggregation ist die Kombination von verschiedenen Services, die zu einem bestimmten gemeinsamen Zweck zur Verfügung gestellt werden. Zum Beispiel stellt ein Telefonanbieter verwandte Dienste wie Rufumleitung, Anrufbeantworter oder Konferenzgespräche bereit. Diese Dienste stellen dabei Unterfunktionen des Dienstes „Telefonieren“ dar und sind ohne ihren Aggregator wertlos [EOH02].

Bei der *Service-Komposition* handelt es sich um die Integration von Sub-Services zu einem neuen Service, der gegenüber den einzelnen Services eine Leistung höheren Wertes erbringt. Die Unterfunktionen des neuen Services bleiben unabhängig und stellen weiterhin eigenständige Services dar. Beispiel für eine Service-Komposition ist ein Reisebüro, welches den Dienst „Reise buchen“ anbietet, der sich aus den selbstständigen Services „Flug buchen“, „Hotelzimmer buchen“ und „Mietwagen reservieren“ zusammensetzt [EOH02].

Ein alternatives, relativ neues Service-Konzept stellen die sogenannten *Location Based Services (LBS)* dar. *Location Based Services* sind ortsabhängige Dienste, die nur in einer bestimmten Region verfügbar sind oder deren Funktionalitäten speziell auf die Anforderungen eines bestimmten Ortes abgestimmt sind. Spiekermann [Spi04] schränkt diese Definition noch weiter ein, indem sie die Dienste auf eine Kombination der räumlichen Position eines mobilen Gerätes mit Informationen, die für den Benutzer nutzbringend sind, spezialisiert. Anwendungen für *Location Based Services* sind in erster Linie kontextangepasste Informationsdienste, zum Beispiel die Veröffentlichung des Wetterberichts für die jeweils relevante Region, in der sich der Empfänger der Informationen gerade befindet. Derartige Informationen werden als *Point-of-Interest Informationen* bezeichnet [StCo04].

Man differenziert bei *Location Based Services* zwischen zwei verschiedenen Service-Dimensionen: Zum einen können Dienste aktiv von einem Benutzer bezogen werden (*Pull Services*), zum Beispiel durch Auswahl des Services mittels eines Browsers oder durch die Benutzung einer Anwendung, die den Dienst vom Netzwerk bezieht. Das Auffinden von Services kann in diesem Fall wieder durch ein lokal zur Verfügung stehendes *Dienstverzeichnis* vereinfacht werden. Dort können zum Beispiel Anfragen wie „Wo befindet sich das nächste Hotel?“ gestellt werden. Zum anderen können Dienste dem Benutzer aber auch zugestellt werden, ohne dass er diese aktiv aufrufen muss (*Push Services*). Dies kann in Form eines gewollten Abonnements von Diensten oder auch als unaufgefordert zugesandte Werbung auftreten („Pizzeria Alberto ist nur zwei Straßen weiter!“) [Spie04].

Um Benutzer oder Geräte zu lokalisieren, ihre Eigenschaften abzufragen oder ortsbezogene Dienste auffinden zu können, bedarf es technischer Möglichkeiten der Wahrnehmung, die im nächsten Kapitel vorgestellt werden sollen.

2.3 Context Awareness

Von Softwarekomponenten angebotene Dienste können sich entweder statisch an einem eindeutig bestimmten Ort befinden oder sie existieren in einer wechselnden, dynamischen Umgebung. Um Dienste erkennen und nutzen zu können, benötigt der *Dienstaufrufer* die spezielle Fähigkeit, Umgebungsinformationen erlangen und verarbeiten zu können. Diese Fähigkeit wird in der Literatur als *Context Awareness* bezeichnet.

Der Begriff *Awareness* beinhaltet sowohl eine wahrnehmende Komponente, als auch ein Verstehen des Wahrgenommenen. Dabei wird speziell durch das Erkennen und Verstehen der Aktivitäten anderer Objekte der Kontext für eigene Aktivitäten gebildet [DoBe92]. *Awareness* ermöglicht also dem Individuum, aktuelle Informationen oder Ereignisse, die durch die Präsenz von anderen Geräten, Personen oder Diensten ausgelöst werden, wahrzunehmen und das eigene Handeln darauf abzustimmen. Als Ergebnis eines solchen Mechanismus kann dann beispielsweise die Anpassung eines Anwendungssystems resultieren.

Der *Kontext* stellt dabei die Abbildung der Menge aller verfügbarer Umgebungsdaten auf die für das Anwendungssystem relevanten Informationen und Bedingungen dar. Er umfasst alle Personen, Ortsdaten und Objekte, die Auswirkungen auf das Verhalten der Anwendung haben können, wobei auch die Anwendung selbst und ihre eigenen Aktivitäten Teile des *Kontextes* darstellen können [Dey01].

Informationen können sowohl kommunikativ als auch sensorisch gewonnen werden. Man unterscheidet im allgemeinen zwischen räumlichen und logischen Kontextdaten. Nach Müller-Wilkens [Mül02] gibt es vier verschiedene Ausprägungen von *Context Awareness*:

Location Awareness: *Location-Awareness* bezeichnet speziell die Erkennung und Verarbeitung physikalisch räumlicher Umgebungsbedingungen. Ortsbezogene Daten werden mit Hilfe von Sensormechanismen oder mittels satellitengestützter Positionsbestimmung (GPS) erhoben und geben Aufschluss über die absolute Position eines Individuums. Ein bekanntes Einsatzgebiet für diese Technik sind zum Beispiel Navigationsgeräte.

Situation Awareness: Bei der Betrachtung von logischen Kontextinformationen wird die exakte Positionsbestimmung vernachlässigt und stattdessen die relative Nähe zu anderen Geräten, Personen oder Objekten in den Mittelpunkt gerückt. Kommt zum Beispiel ein benötigter *Dienstanbieter* in den Aktionsbereich eines potentiellen *Dienstaufrufers*, so kann ein Event ausgelöst werden, welches auf den aktualisierten Kontext aufmerksam macht. Dieses Verhalten wird allgemein unter dem Begriff *Situation Awareness* zusammengefasst.

Network Awareness: Die Fähigkeit, Veränderungen der Kommunikationsverbindung wahrnehmen und beurteilen zu können, wird als *Network Awareness* bezeichnet. Im einfachsten Fall kann man sich darunter einen Indikator vorstellen, der anzeigt, ob ein bestimmtes Netz verfügbar ist oder nicht. Daneben auch qualitative Veränderungen des Netzes messen zu können, stellt einen weiteren Informationsgewinn für kontextsensitive Systeme dar. Hierzu gehören zum Beispiel die Ermittlung und Auswertung der Kommunikationsgeschwindigkeit oder die Feststellung der Fehlerrate bei einer Übertragung.

Energy Awareness: Informationen über den Zustand einer Energiequelle werden unter dem Begriff *Energy Awareness* zusammengefasst. Insbesondere Aussagen über den Energieverbrauch einzelner

Rechenvorgänge und Applikationen können in diesem Zusammenhang Entscheidungen über die Verwendung von Ressourcen beeinflussen.

Die Aufzählung dieser Awarenessaspekte ist nicht erschöpfend. Ein Bewusstsein über Zeit und Kosten oder gar über den Verlauf der eigenen Mobilität stellen weitere wichtige Punkte für die Integration von Services dar. Zu wissen, wann und wo sich das eigene System unter welchen Bedingungen befindet, kann unter Umständen entscheidende Informationen zur Auswahl von Diensten oder sogar über zukünftige Aufenthaltsorte oder Bedürfnisse liefern.

Kontextsensitive Architekturen eignen sich insbesondere für mobile Systeme, da sie in der Lage sind, die aus der Mobilität resultierenden dynamischen Umgebungsbedingungen zu erfassen. Anforderungen des Benutzers können automatisch aus der Anwendungssituation heraus erkannt und die Funktionalität des Systems entsprechend anpasst werden. Dies erlaubt nach der Vision des *Ubiquitous Computing* [Wei91] die größtenteils unbewusste Nutzung von mehreren miteinander vernetzten Geräten auch in Situationen, die keine exklusive Aufmerksamkeit des Benutzers für die Anwendungsinteraktion erlauben.

2.4 Vorstellung des DEMAC-Projekts

Das Projekt DEMAC[1] beschäftigt sich inhaltlich mit der Entwicklung einer kontextsensitiven Middleware zur Unterstützung komplexer Prozesse auf mobilen Systemen.

Ziel der DEMAC-Middleware ist es, eine Infrastruktur zu implementieren, die es erlaubt, Prozesse auf einem hohen Abstraktionsniveau zu definieren, auf mobilen Geräten auszuführen und bei Bedarf an Partner in der unmittelbaren Umgebung weiterzugeben. Konkrete Dienste und Prozesspartner werden umgebungsabhängig erst zur Laufzeit und somit erst unmittelbar vor der Bearbeitung der anstehenden Aufgabe ausgewählt. Benutzerdefinierte Anforderungen an potentielle Ausführungseinheiten und ihre Dienstleistungsqualitäten werden hierzu als besondere Rahmenbedingungen betrachtet. Innerhalb einer bestimmten Umgebung muss ein Gerät daher ohne zentrale Administration in der Lage sein, andere Komponenten über seiner Existenz zu informieren, seine Dienste anzubieten, die Dienste anderer Teilnehmer in Anspruch zu nehmen und mit anderen Geräten zu interagieren, um seine eigenen Aufgaben erfüllen zu können [Kun05].

Die grundlegende Architektur der DEMAC-Middleware besteht für diese Anforderungen aus vier Dienstkomponenten (Abb. 2):

Asynchronous Transport Service: Der *Asynchronous Transport Service* stellt eine proaktive Infrastruktur zur nachrichtenbasierten Kommunikation zur Verfügung. Andere Geräte können hier registriert werden, um Nachrichten zu senden und zu empfangen. Dazu verwaltet der *Transport Service* die konkreten Netzwerkeigenschaften und Protokolle der teilnehmenden Geräte über eine interne Adressierung, welche den anderen Komponenten der DEMAC-Architektur als Schnittstelle zur Kommunikation zur Verfügung steht. Die höheren Schichten der Architektur können daher Kommunika-

1 vgl. DEMAC unter http://vsis-www.informatik.uni-hamburg.de/projects/demac/

tionspartner über einen einheitlichen *Device Handle* erreichen, unabhängig davon, welche konkreten Technologien das Gerät zur Kommunikation verwendet.

Event Service: Nachrichten über Veränderungen des eigenen Zustandes oder der Umgebung werden vom *Event Service* an registrierte Komponenten weitergegeben. Dabei können sowohl einzelne Geräte für bestimmte Events registriert werden als auch architekturinterne Komponenten untereinander Ereignisse austauschen. Auftretende Ereignisse bezüglich Umgebungsveränderungen werden dem *Context Service* des mobilen Geräts übermittelt. Events können hierzu in verschiedene Sichtbarkeitskategorien eingeordnet werden, um Kontrolle über den Informationsfluss des Systems zu erlangen.

Context Service: Der *Context Service* sammelt Umgebungsdaten und filtert heraus, welche Informationen für das System relevant sind. Hierbei handelt es sich insbesondere um Daten zu verfügbaren Geräten und Diensten sowie deren Eigenschaften. Der *Context Service* hat Zugang zu den lokal zur Verfügung stehenden Umgebungsdaten und zu einer verteilten Registratur, welche ergänzende Informationen über die im Umfeld angebotenen Dienste bereitstellt. Neben Details zu fremden Diensten und Geräten hält der *Context Service* auch Informationen über das eigene Gerät, zum Beispiel Leistungsmerkmale oder benutzerdefinierte Präferenzen.

Process Service: Der *Process Service* stellt eine Ausführungsumgebung dar, in der Folgen komplexer Aufgaben verwaltet und verarbeitet werden können. Informationen über dafür verfügbare Dienste und Geräte erhält er vom *Context Service*. Desweiteren führt er über den *Asynchronous Transport Service* Dienstaufrufe aus und tauscht Prozessbeschreibungsdaten mit anderen Kommunikationspartnern aus [Kun05].

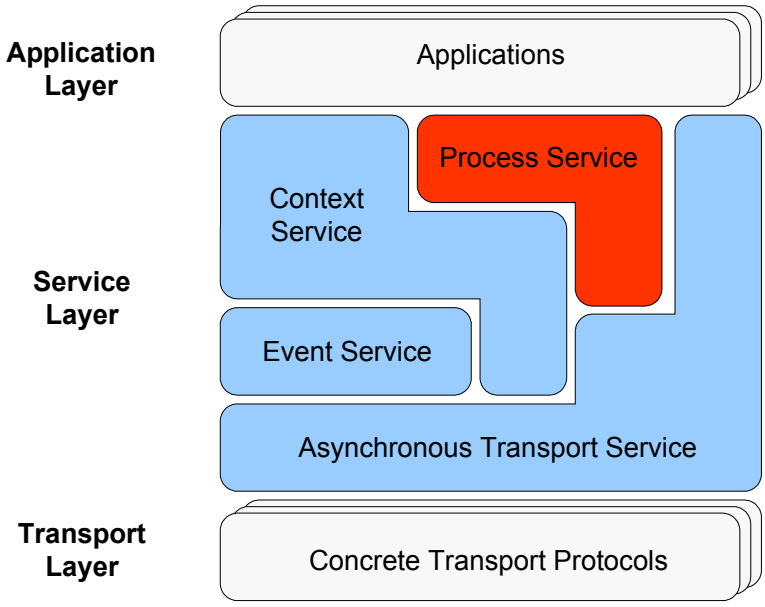


Abb. 3: Überblick über die Architektur der DEMAC Middleware (nach [Kun05])

Die Implementation dient als Middleware zwischen der Ebene der konkreten Kommunikationsprotokolle und der Anwendungsebene und stellt darauf aufsetzenden Anwendungsprogrammen Schnittstellen und Funktionalitäten zur Verfügung, um Dienste komponieren und ausführen zu können sowie um allgemeine Kontextinformationen zu beziehen [Kun05].

2.5 Anwendungsbeispiel: Versicherung

Das folgende Szenario beschreibt, wie der Prozess einer teilautomatisierten Schadensabwicklung im Versicherungsumfeld von einer kontextsensitiven und prozessorientierten Middleware für mobile Geräte unterstützt und verteilt bearbeitet werden kann.

Ein Kunde der ALLIANCE Versicherungs-AG erleidet auf einer Geschäftsreise im Ausland einen Verkehrsunfall mit seinem Fahrzeug. Er startet die Anwendungssoftware der Versicherung „ALLIANCE Schadenschnellhilfe“ auf seinem Mobiltelefon, welche ihm erlaubt, den Schadensfall zu melden und weitere Maßnahmen zur Schadensregulierung zu treffen (Abb. 4).

Abb. 4: Beispiel für eine Anwendungssoftware der Versicherung auf dem Mobiltelefon eines Versicherungsnehmers

Der Versicherungsnehmer identifiziert sich hierfür zunächst über seine Versichertennummer und entscheidet sich entsprechend der Schwere des Unfalls dafür, den Schadensfall der Versicherung zu melden und die Polizei anzufordern, um einen Unfallbericht anzufertigen. Da sein beschädigtes Fahrzeug nicht mehr straßenverkehrssicher ist, möchte er es abschleppen lassen und eine Werkstatt beauftragen, es so schnell wie möglich zu reparieren. Er bestellt außerdem einen Leihwagen, um seine Geschäftsreise fortsetzen zu können.

Der Kunde schließt seine persönlichen Eingaben ab und die Anwendungssoftware auf seinem Mobiltelefon ergänzt diese um weitere Details zur internen Abwicklung des Geschäftsfalls, zum Beispiel die Ermittlung von GPS-Daten des Unfallortes oder die Bestellung eines Gutachters zwecks Einschätzung des Schadens. Es resultiert ein gemeinsamer Ablaufplan aus Zielen des Versicherungs-

nehmers und der Versicherung. Da nicht alle benötigten Dienste vor Ort verfügbar sind, beziehungsweise nicht von der mobilen Anwendung des Versicherungsnehmers angesprochen und ausgeführt werden können, generiert die Anwendungssoftware aus dem Ablaufplan eine passende abstrakte Prozessbeschreibung, die es ermöglicht, einzelne Subprozesse oder den gesamten Prozess an andere mögliche Ausführungseinheiten weiterzugeben.

Die Prozessbeschreibung enthält eine Beschreibung der angeforderten Dienste und Informationen zur Ablaufsteuerung. Abbildung 5 zeigt die Abhängigkeiten der auszuführenden Aufgaben untereinander.

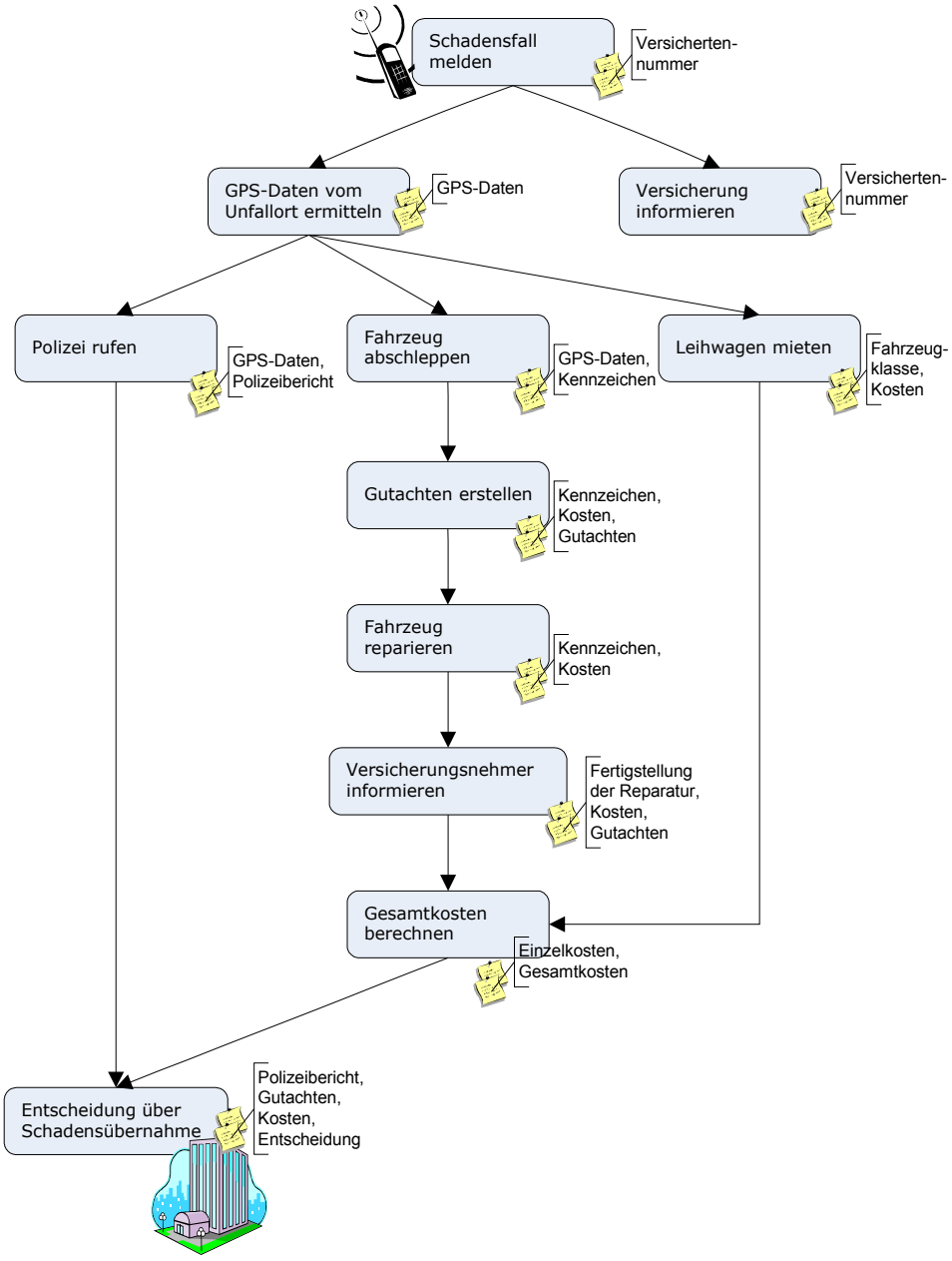


Abb. 5: Abstrakte Prozessbeschreibung zum Anwendungsfall “Schadensabwicklung” mit Angabe involvierter Daten

Außerdem definiert sie, welche Daten bei welcher Aufgabe benötigt und generiert werden und macht Angaben über spezielle Zeit-, Qualitäts- und Sicherheitsanforderungen. Der fertige Ablaufplan wird dann an die lokale Middleware auf dem Mobiltelefon des Kunden übergeben, um die notwendigen Ressourcen zur Ausführung erschließen und komponieren zu lassen. Dabei stehen zunächst nur die benutzerdefinierten Aktivitäten und Akteure fest, die konkreten ausführenden Dienste werden erst zur Bearbeitungszeit der jeweiligen Aufgabe abhängig von der spezifischen Umgebungssituation ermittelt.

Die Middleware interpretiert die Prozessbeschreibung und prüft zunächst, ob anstehende Aufgaben von Diensten auf dem mobilen Gerät des Versicherungsnehmers erfüllt werden können. Ist dies nicht der Fall, sucht die Middleware in ihrer Umgebung nach angebotenen Diensten, die den Anforderungen des Benutzers entsprechen. Kann eine aktuell zu bearbeitende Aufgabe nicht auf eine dieser beiden Weisen lokal erfüllt werden, wird der Prozess an andere geeignete Ausführungseinheiten weitergegeben, die ihrerseits nach passenden Komponenten zur Ausführung einzelner Aufgaben oder zur Weitergabe des Prozesses suchen. Nicht-funktionale Aspekte, wie zum Beispiel die Vertrauenswürdigkeit der beteiligten Partner, spielen hierbei eine große Rolle.

Im Beispielfall stehen Anwendungen für die Ausführung der beiden Aufgaben „GPS Daten vom Unfallort ermitteln“ und „Versicherung informieren“ direkt auf dem Mobiltelefon des Versicherungsnehmers bereit. Die beiden Punkte können parallel ausgeführt werden, da offensichtlich keine gegenseitigen Abhängigkeiten bestehen. Sobald die benötigten GPS-Daten verfügbar sind, wird nach ortsnahen Diensten für Polizeiruf und Abschleppen des Unfallfahrzeuges sowie nach einem Leihwagenverleih gesucht.

An dieser Stelle endet die Kompetenz des mobilen Geräts des Versicherungsnehmers, da kein lokaler Dienst zur Beauftragung eines Gutachters zur Verfügung steht. Der Prozess wird daher einem geeigneten verfügbaren Partner, in diesem Fall dem mobilen System des Abschleppdienstes, übertragen. Dieser übernimmt die Verantwortung für die weitere Prozesskette und überführt das beschädigte Fahrzeug zu einer entfernten Werkstatt. Dort findet er mehrere Gutachter vor und wählt nach den Vorgaben des Ablaufplans einen geeigneten Dienstleister aus. Hier hat er nun die Möglichkeit, entweder den Gesamtprozess an das mobile System des Gutachters zu übergeben, oder die Aufgabe lediglich lokal vom Gutachter erledigen zu lassen und den Prozess selbst weiterzuführen. Da jedoch Daten in Form eines Gutachtens vom Dienst zurückgegeben werden sollen und sich der Abschleppdienst in der Zwischenzeit außer Reichweite befinden könnte, entscheidet die kontextsensitive Ausführungsumgebung, den Prozess gänzlich durch den Gutachter weiter bearbeiten zu lassen.

Nach der Erstellung des Gutachtens kann mit der Bearbeitung der nächsten Aufgabe begonnen werden. Die für die Ausführung des Prozesses verantwortliche Systemkomponente des Gutachters überprüft, ob die Werkstatt, in welcher er sich gerade befindet, geeignet ist, um die Reparaturen am Fahrzeug durchzuführen oder ob es andere mögliche Anbieter gibt. Bei der Auswahl des konkreten Dienstes werden sowohl funktionale Kriterien, wie etwa die Präferenz einer Vertragswerkstatt für das spezielle Fahrzeugmodell, als auch nicht-funktionale Aspekte, wie zum Beispiel der Zeitbedarf für die Reparatur berücksichtigt (Abb. 6).

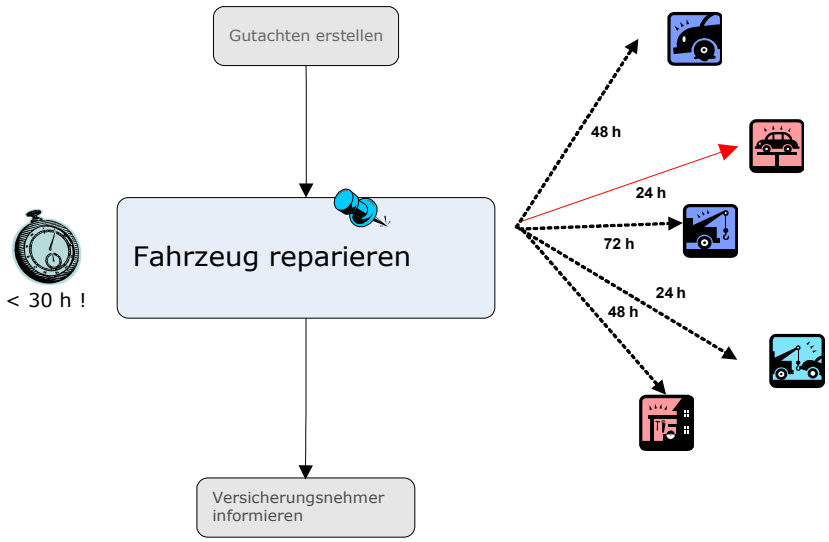


Abb. 6: Auswahl eines Dienstes zur Erfüllung der Aufgabe “Fahrzeug reparieren” anhand des nicht-funktionalen Kriteriums “Zeitbedarf für die Reparatur”

Das mobile System des Gutachters übergibt den Prozess an das stationäre System der gewählten Werkstatt. Nach der Instandsetzung des Schadens informiert die Werkstatt den Versicherungsnehmer, dass er sein Fahrzeug abholen kann und überlässt ihm bei der Übergabe den Restprozess mit zugehörigen Daten, wie Gutachten und Kostenaufstellung, zur weiteren Ausführung. Da der Akku seines Mobiltelefons bald erschöpft sein wird, übernimmt der Versicherungsnehmer den Prozess auf sein Notebook. Eine leistungsfähige Anwendung ist dort in der Lage, die nächste anstehende Aufgabe lokal auszuführen und berechnet aus Abschlepp-, Gutachten-, Reparatur- und Leihwagenkosten die Gesamtkosten des Unfalls. Sobald wieder eine zur Übertragung ausreichende Verbindung hergestellt werden kann, werden die gesammelten Daten an das stationäre System der Versicherung übermittelt. Diese kann nun über eine Übernahme des Schadens entscheiden und den Versicherungsfall mit einer Archivierung der relevanten Daten abschließen.

Abbildung 7 zeigt die während der Ausführung dynamisch erzeugten Verantwortlichkeiten für den Prozess. Die unterschiedlichen Farben illustrieren die Weitergabe des Prozesses an einen anderen Beteiligten.

Mit Ausnahme des Versicherungsnehmers als Initiator des Prozesses und der Versicherung als vorgegebenes Zielsystem sind vor Bearbeitung des Ablaufplans keine festen Prozessbeteiligten definiert. Die Ausführungseinheiten werden erst zur Laufzeit umgebungs- und funktionsabhängig ausgewählt und beauftragt.

Lokal auszuführende Aufgaben, die entweder durch geräteeigene Anwendungen oder lokal verfügbare externe Dienste realisiert werden können, fallen in den Verantwortungsbereich des prozessausführenden Systems. So ist zum Beispiel für die Aufgaben “Versicherung informieren” und “Polizei rufen” die Ausführungsumgebung auf dem Mobiltelefon des Versicherungsnehmers zuständig. Die Middleware dort entscheidet anhand der benutzerdefinierten Ziele und der vom Kontext gelieferten Informationen über die Auswahl möglicher weiterer Prozessbeteiligter. Für die Übernahme eines Prozesses kommt ein fremdes System nur dann in Frage, wenn es über den entsprechenden Interpreter für die Prozessbeschreibung verfügt und den angegebenen nicht-funktionalen Kriterien genügt.

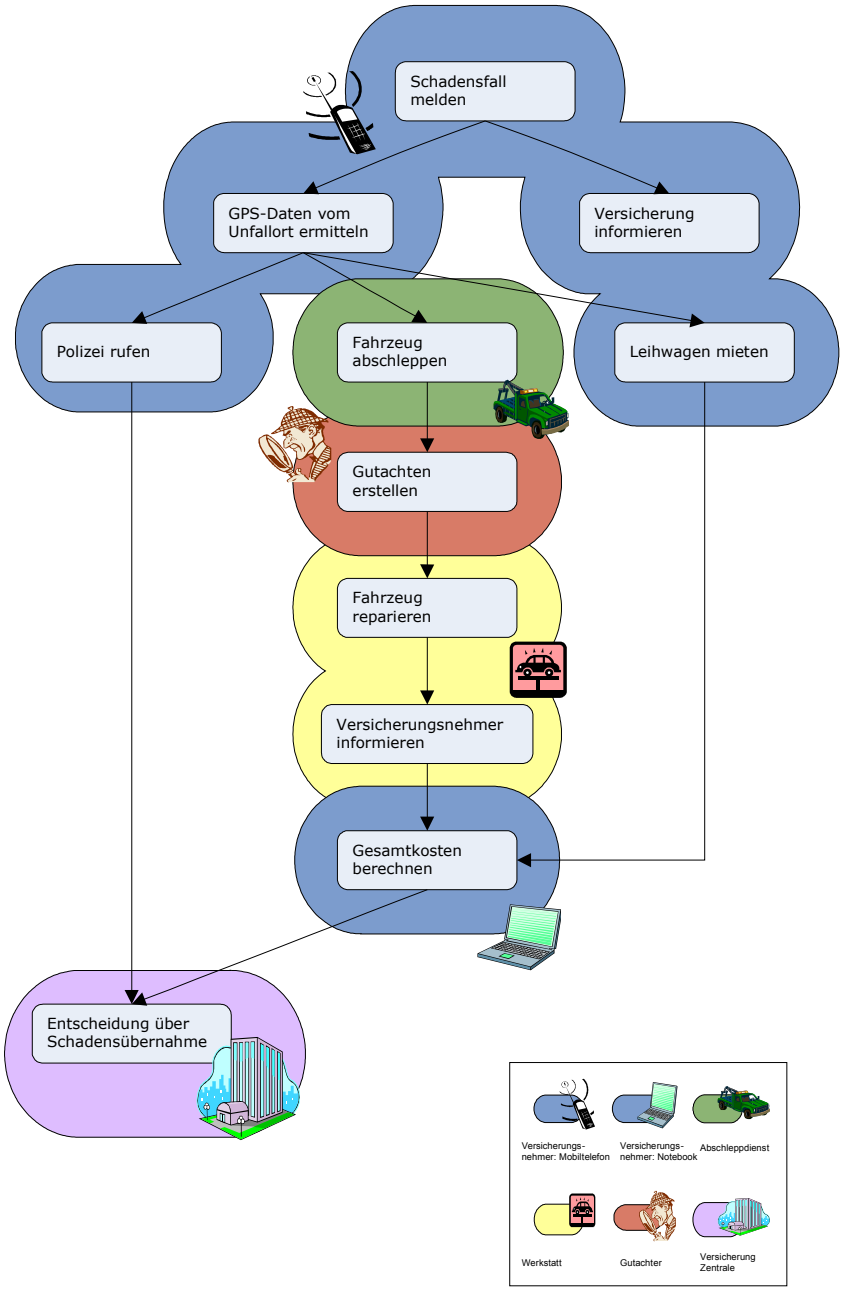


Abb. 7: Dynamisch ermittelte Systemverantwortlichkeiten für Ausführung und Weitergabe des Beispielprozesses

Mit der Übergabe der Prozessbeschreibung an ein anderes System wird auch die Verantwortung für deren weitere Ausführung übergeben. Im Beispiel wird aus Mangel an adäquaten verfügbaren Diensten der Prozess dem mobilen System des Abschleppdienstes übertragen. Abschleppdienst, Gutachter und Werkstatt erfüllen sequentiell die ihnen jeweils lokal möglichen Aufgaben und reichen den Prozess schließlich an den Versicherungsnehmer zurück. Dieser nimmt ihn jedoch nicht wieder mit seinem Mobiltelefon, sondern mit seinem Notebook auf.

Eine Synchronisation der Daten von beiden Geräten kann an dieser Stelle notwendig werden. Letztendlich wird der Prozess an die vorgegebene Instanz, nämlich das zentrale System der Versicherungsgesellschaft, weitergereicht, wo er ordnungsgemäß terminiert.

Das Beispiel zeigt zum einen, dass der Einsatz einer Prozessbeschreibungssprache für mobile Systeme durchaus zu anwendungsnahen Szenarien führen kann, die nicht nur für betriebliche Abläufe, sondern auch für private Nutzer in ihrem täglichen Leben relevant sein können. Zum anderen wird deutlich, dass im Zusammenhang mit mobilen Systemen Herausforderungen auftreten, die die Ausführung von Prozessen beschränken oder spezielle Vorkehrungen zum Prozessmanagement erfordern. Im nachfolgenden Kapitel wird diese Problematik vertieft.

2.6 Einordnung des Themas dieser Arbeit

Diese Diplomarbeit beschäftigt sich mit der Bereitstellung eines grundlegenden Teils des *Process Services* für das DEMAC-Projekt (vgl. Kapitel 2.4). Sie stellt ein Konzept für dessen Architektur sowie eine mögliche Umsetzung vor und beschreibt die erforderlichen Basiskomponenten. Dazu gehören die Definition einer für mobile Systeme geeigneten Prozessbeschreibungssprache und der Kern einer erweiterbaren Ausführungsumgebung. Die Infrastruktur zur Bereitstellung von Kontextinformationen und nachrichtenbasierter Kommunikation wird dabei vom DEMAC-Projekt zur Verfügung gestellt.

Der Fokus der Arbeit liegt bei der Komposition von möglichen Diensten in verteilt ausgeführten Prozessen und bei der Verwaltung des Kontrollflusses durch heterogene mobile Systeme. Für die eigentliche Ausführung von Diensten werden in dieser Arbeit nur Schnittstellendefinitionen angegeben, um eine prinzipielle Erweiterbarkeit auf beliebige Dienst- und Anwendungstypen vorzubereiten. Die Entwicklung einer übergeordneten Registrierungs- und Suchfunktion für Komponenten sowie die tatsächliche Einbindung von konkreten Anwendungen ist nicht Teil dieser Arbeit.

Weitere Einschränkungen werden gemacht, wo das Themengebiet der eigentlichen Arbeit verlassen wird und Anknüpfungspunkte im DEMAC-Projekt noch nicht vorhanden sind. Die Schnittstellen zum *Context Service* sind dabei als Vorschläge für eine spätere Implementierung zu sehen.

3 Prozessintegration in mobile Systeme

Das gezeigte Anwendungsbeispiel macht deutlich, dass der erfolgreichen Ausführung eines mobilen Prozesses eine ganze Reihe von Herausforderungen entgegenstehen. Das folgende Kapitel soll die Problematiken erarbeiten, die mit einer Prozessausführung auf mobilen Systemen in Zusammenhang stehen und die Anforderungen an eine für diesen Anwendungsbereich geeignete Prozessbeschreibungssprache konkretisieren.

3.1 Mobile Systeme

In diesem Abschnitt werden die grundlegenden Eigenschaften mobiler Systeme dargestellt. Dazu erfolgt zunächst in 3.1.1 eine Erläuterung des Begriffs *Mobile Computing*. In 3.1.2 werden die Eigenschaften mobiler Systeme betrachtet und ihre spezifischen Problemfelder und Beschränkungen gegenüber stationären Systemen aufgezeigt. Schließlich betrachtet Abschnitt 3.1.3 den aktuellen Stand mobiler Geräte und mobiler Kommunikation.

3.1.1 Mobile Computing

Unter dem Oberbegriff des *Mobile Computing* versteht man die Verwendung von Recheneinheiten an beliebigen Orten. Dabei steht die Kommunikation zwischen mobilen Benutzern, mobilen Geräten und den dazugehörigen Anwendungen im Vordergrund. Objekte werden allgemein als mobil bezeichnet, wenn sie geeignet sind, ohne großen Aufwand ihren Standort zu wechseln [Rot02].

3.1.1 Mobile Computing

Nach Pandya [Pan99] gibt es drei verschiedene Arten von Mobilität:

Endgerätemobilität: Endgeräte werden als mobil bezeichnet, wenn sie ihren Einsatzort wechseln können ohne ihre Netzanbindung dauerhaft zu verlieren. Dem Endgerät ist dabei ein fester Benutzer zugeordnet.

Benutzermobilität: Diese Art der Mobilität liegt vor, wenn ein Benutzer für eine bestimmte Aufgabe verschiedene Endgeräte von unterschiedlichen Orten aus benutzen kann. Es kommt hierbei also darauf an, den Benutzer unverwechselbar zu identifizieren und ihm ein beliebiges, nicht zwingend mobiles Endgerät als möglichen Zugangskanal zuzuordnen [Schi03].

Dienstmobilität: Unter Dienstmobilität versteht man die technische Möglichkeit, einen bestimmten Dienst von beliebigen Orten ausführen zu können, an denen ein entsprechender Zugangspunkt zu diesem Dienst besteht. Zum Beispiel kann der Dienst "E-Mail" von verschiedenen Orten weltweit in Anspruch genommen werden, wenn das benötigte Kommunikationsnetz dort zur Verfügung steht.

Neben der Mobilität stellt die Fähigkeit zur Kommunikation einen wichtigen Aspekt im *Mobile Computing* dar: Mobile Geräte können zwischen verschiedenen Netzwerken bewegt werden und so ihre Kommunikationsverbindung erneuern oder aufrecht erhalten. Die kurzfristige Vernetzung von Geräten ohne feste Kommunikationsinfrastruktur und ohne die Notwendigkeit, den Benutzer in eine aufwendige Konfiguration einzubeziehen, ermöglicht es, spontan Daten zwischen mobilen Geräten auszutauschen. Diese Art der Anbindung wird als *Ad-hoc-Vernetzung* bezeichnet. Die Verbindung mobiler Geräte mit einem stationären Netzwerk, um von verschiedenen Orten aus an bestehenden Kommunikations-Infrastrukturen, wie zum Beispiel dem Internet, teilhaben zu können, ist Thema des *Nomadic Computings* (vgl. [Klei97]).

Mobile Computing steht in engem Zusammenhang mit der Vision des *Ubiquitous Computing*, in der mobile Geräte nahtlos und im Idealfall unsichtbar in das tägliche Leben der Benutzer integriert sind. Die mobilen Geräte bilden dabei mit ihrer Kommunikationsverbindung untereinander ein System, das dem Benutzer überall und jederzeit zur Verfügung steht, um ihn zu unterstützen, ohne selbst besondere Aufmerksamkeit zu verlangen [Wei91].

Diehl und Held [DiHe95] haben anhand möglicher Einsatzgebiete drei Anwendungsschwerpunkte des *Mobile Computing* charakterisiert:

Erstens dient *Mobile Computing* Reisenden als Ersatz für einen stationären Arbeitsplatzrechner, zum Beispiel zur Nutzung von E-Mail, für die Informationsbeschaffung oder für die Durchführung von Transaktionen wie Online-Shopping. Zweitens können mobile Echtzeitsysteme Leit-, Überwachungs- und Steuerungsfunktionen für Verkehrssysteme bereitstellen, zum Beispiel für Navigationsgeräte. Drittens unterstützt *Mobile Computing* verteiltes Arbeiten und wird in bestehende Infrastrukturen zur Organisation, Verteilung und Ausführung der Arbeit integriert. Zum Beispiel können Mitarbeiter eines Unternehmens mit einem mobilen Eingabegerät aktuelle Lagerbestände aufnehmen und an die Einkaufsabteilung senden, um dort entsprechende Nachbestellungen tätigen zu können.

Insbesondere stehen beim *Mobile Computing* Anwendungen und Probleme im Vordergrund, die sich daraus ergeben, dass die Voraussetzungen der Stabilität fixer Infrastrukturen im mobilen Bereich nicht gegeben sind. Im nachfolgenden Abschnitt soll geklärt werden, welche Eigenschaften mobile Systeme aufweisen und welche Einschränkungen sich daraus ergeben.

3.1.2 Eigenschaften und Beschränkungen mobiler Systeme

Forman und Zahorjan [FoZa94] charakterisieren ein mobiles System anhand seiner Fähigkeiten, mobil und portabel zu sein sowie Möglichkeiten zur drahtlosen Kommunikation zu besitzen. Diese Eigenschaften bringen jeweils eine Reihe von Beschränkungen der Leistungsfähigkeit des mobilen Systems gegenüber stationären Geräten mit sich, die im Folgenden erläutert werden sollen. Die Beschränkungen sind jedoch keinesfalls nur an die aktuellen technischen Gegebenheiten geknüpft. Vielmehr sind sie spezifisch für mobile Systeme, die trotz kontinuierlicher Weiterentwicklung vergleichbaren statischen Systemen als dauerhaft leistungsschwach gegenüberstehen [Sat96].

Wireless Communication: Ein mobiles System verfügt über die Möglichkeit, mittels Funk- oder Infrarotschnittstellen drahtlos zu kommunizieren. Dadurch gewinnt es im Verfügbarkeitsbereich des drahtlosen Netzes an Ortsunabhängigkeit und kann flexibel eingesetzt werden [FoZa94]. Drahtlose Kommunikation ist jedoch im Vergleich zu Festnetztechnologien relativ leistungsarm:

Drahtlose Netzwerke verfügen nur über eine geringere Bandbreite, da die verfügbaren Ressourcen wie Frequenzen oder Übertragungszeiträume durch Multiplexverfahren zwischen den Kommunikationsteilnehmern einer Funkzelle aufgeteilt werden müssen. Es kann daher unter Umständen zu erheblichen Schwankungen der Bandbreite kommen, die von der Belastung und der Verfügbarkeit des Netzes abhängig ist [FoZa94].

Zusätzlich weisen drahtlose Netzwerke höhere Fehlerraten auf, da Funksignale nicht gegen Störungen abgeschirmt werden können. Durch viele Übertragungsfehler wächst zugleich der Aufwand an Kommunikation, denn Übertragungen müssen zur Fehlerkorrektur mit höherer Redundanz gesendet und gegebenenfalls wiederholt werden. Im Fehlerfall kommt es durch Timeouts noch zu weiteren Verzögerungen. Unerkannt gebliebene Fehler können die Übertragung verfälschen oder unbrauchbar machen [FoZa94].

In drahtlosen Netzen kommt es zudem häufig zu Verbindungsabbrüchen. Diese werden einerseits durch die Fehleranfälligkeit des Netzes verursacht, zum Beispiel wegen Überlastung des Netzes oder durch Störsignale aus der Umgebung. Zudem haben drahtlose Netze nur eine beschränkte Reichweite. Mobile Geräte, die den Verfügbarkeitsbereich des drahtlosen Netzes verlassen, werden in ihrer Verbindung getrennt [FoZa94]. Verbindungsabbrüche werden jedoch auch absichtlich vom Benutzer herbeigeführt [Mül02]: Mobile Geräte werden ausgeschaltet, um Energie zu sparen, die Kommunikation wird beendet, um Verbindungskosten zu reduzieren oder die Verwendung des mobilen Gerätes ist an bestimmten Orten untersagt, zum Beispiel im Flugzeug. Letztendlich könnte der Benutzer eine Verbindung sogar vorsätzlich unterdrücken, um “Unerreichbarkeit” vorzugeben.

Mobile Systeme müssen sich außerdem an heterogene Netzwerke anpassen. Diese verfügen eventuell nicht nur über unterschiedliche Bandbreiten und Quality-of-Service-Merkmale [Sat96], sondern auch über völlig andere Eigenschaften und Protokolle. So müssen zum Beispiel teilnehmende Geräte einer Infrarot-Verbindung aufeinander ausgerichtet werden, während sich Kommunikationspartner einer Funkverbindung unabhängig voneinander in verschiedenen Räumen befinden können.

Die Ausbreitung eines Signals durch die Luft kann kaum beschränkt werden und ist dadurch jedem empfangsbereiten Gerät zugänglich, um die Kommunikation anderer abzuhören oder zu manipulieren. Dies stellt ein erhebliches Sicherheitsrisiko dar, dem nur durch geeignete Verschlüsselungsverfahren entgegengetreten werden kann. Dem Bedürfnis nach Sicherheit und Datenschutz steht aber nur ein endlicher Vorrat an Energie und Rechenkapazität zur Verfügung. Eine Verschlüsselung von Daten direkt an der Quelle kann den Bedarf an diesen Ressourcen vervielfachen, was einige Anwendungen eventuell unmöglich macht [Mat03].

Schließlich ist die Benutzung von drahtlosen Netzen insbesondere im Bereich der Mobilkommunikation kostenpflichtig. Die Abrechnung erfolgt zumeist über Zeiteinheiten, die das mobile Gerät aktiv mit dem Netz verbunden ist, oder über das gesendete Datenvolumen. Eine permanente oder lang andauernde Kommunikation ist daher aus Kostengründen nicht vorteilhaft [SGSP04].

Mobility: Ein mobiles System verfügt über die Möglichkeit, seinen Aufenthaltsort zu ändern, ohne seine Funktionalität zu verlieren. Durch die Nutzung eines drahtlosen Netzes kann eine Kommunikation im Idealfall sogar während der Ortsänderung selbst aufrecht erhalten bleiben [FoZa94]. Die Mobilität des Systems stellt jedoch Herausforderungen an die konkrete Adressierung und an die Informationsbeschaffung:

Ein mobiles System hat keine ortsbezogene feste Adresse, unter der es permanent zu erreichen ist. Der Aufenthaltsort muss also nach einem Ortswechsel aufwändig ermittelt werden und kann unter Umständen vielleicht auch gar nicht mehr festgestellt werden [FoZa94]. Außerderm stellt sich die Frage, ob ein bestimmtes mobiles Gerät oder ein bestimmter mobiler Benutzer adressiert werden soll. Mobile Benutzer sind in der Regel durch mehrere mobile oder stationäre Infrastrukturen zu erreichen. Soll ein mobiler Benutzer ausfindig gemacht werden, so muss zugeordnet werden können, über welche mobilen Geräte eine Kommunikation möglich ist [Kun05]. Aus Sicherheitsgründen kann eine Authentifizierung von Benutzer oder Gerät notwendig werden.

Umgebungsinformationen und lokal verfügbare Dienste sind durch die Mobilität des Systems hochdynamisch. Nach einer Änderung des Aufenthaltsortes können ortsabhängig andere Informationen und Ressourcen zur Verfügung stehen als vor dem Ortswechsel. Eben noch verfügbare Dienste werden im Einzelfall sogar während der Dienstausführung unterbrochen oder verändern ihre Parameter. Mit den Diensten kann sich ebenfalls die zugrunde liegende Netzwerktopologie ändern und Ad-Hoc-Verbindungswechsel erfordern. Relevante Kontextdaten müssen daher zeitnah sensorisch oder kommunikativ erfasst und ausgewertet werden [Jac04].

Die Notwendigkeit, Kontextdaten aus der unmittelbaren Umgebung zu beziehen, steht jedoch dem Recht des Benutzers gegenüber, seine Privatsphäre zu wahren und die Weitergabe von vertraulichen Informationen zu verwehren. Insbesondere detaillierte Daten zum Aufenthaltsort eines mobilen Geräts oder eines Benutzers müssen vor Missbrauch geschützt werden [Eck03].

Portability: Mobile Systeme verfügen über eine geringe Größe und eine Beschaffenheit, die es dem Benutzer ermöglicht, sie ständig mit sich zu führen. So sind sie meistens handlich und stoßfest und verfügen über eine eigene stromnetzunabhängige Energiequelle [FoZa94]. Die Portabilität verleiht den mobilen Systeme erst ihre charakteristischen Eigenschaften und damit ihre praktische Einsatzfähigkeit. Hieraus ergeben sich aber auch große Einschränkungen in ihrem Leistungspotential:

Mobile Systeme besitzen eine endliche Energiequelle. Da auch die Größe des Akkus bzw. der Batterie durch das Design des mobilen Geräts begrenzt wird, ist hier ein Kompromiss zwischen der maximalen Kapazität und einem Mindestmaß an Portabilität gefragt. Ein zu großer Energiespeicher sichert zwar eine lang anhaltene Energieversorgung, ist jedoch schwer zu transportieren. Aber auch ein zu kleiner Akku mit zu geringen Energieressourcen würde die Portabilität einschränken, da der Benutzer dann gezwungen wäre, zu oft die Ladestation für das Gerät aufzusuchen. Begrenzte Energiequellen stellen in jedem Fall Ansprüche an einen möglichst effizienten Umgang mit den verfügbaren Ressourcen: Hardware und Software-Anwendungen müssen jeden unnötigen Energieverbrauch vermeiden und nicht benötigte Komponenten nach Gebrauch abschalten [Sat96].

Die Portabilität und die begrenzten Energieressourcen limitieren auch den möglichen Speicherplatz, der einem mobilen System zur Verfügung gestellt werden kann. Festplatten mit hoher Speicherkapazität sind aufgrund ihrer Empfindlichkeit, ihrer Größe und ihres hohen Stromverbrauchs in der Regel nicht für kleine mobile Geräte geeignet. Stattdessen verwendet man in erster Linie Speicherchips, deren Kapazität den optischen und elektromagnetischen Speichermedien um einiges nachsteht [Sat96] oder ähnlich leistungsfähige kleine Festplatten mit geringerem Energiebedarf.

Die Prozessorleistung von mobilen Geräten ist gegenüber stationären Arbeitsplatzrechnern als gering einzustufen. Prozessoren höherer Leistung verbrauchen zu viel Energie und stellen zudem unrealisierbare Ansprüche an Kühlung und Platzbedarf. Mobile Systeme müssen daher mit einer verminderten Rechengeschwindigkeit auskommen [FoZa94] .

Aufgrund ihrer geringen Größe können mobile Systeme nur mit sehr kleinen, leistungsarmen Benutzerinterfaces ausgestattet werden. Displays verfügen nur über eine sehr geringe Auflösung, sind oft nur monochrom und bieten wenig Platz, um umfangreiche Dokumente darzustellen oder deren Bearbeitung zu erlauben [FoZa94].

Letztendlich ist alles, was klein und problemlos portierbar ist, der Gefahr ausgesetzt, verloren, beschädigt oder gestohlen zu werden. Neben dem Verlust des Gerätes besteht das Risiko, dass darauf befindliche Daten nicht ersetzt werden können oder in falsche Hände geraten [Sat96].

Wie bereits festgestellt, unterliegt die Leistungsfähigkeit der drahtlosen Netze einer hohen Variabilität. Ebenso ist die Leistungsfähigkeit verschiedener mobiler Geräte stark unterschiedlich und vom konkreten Einsatzzweck abhängig. Der folgende Abschnitt wirft einen Blick darauf, welche Arten von mobilen Geräten und drahtlosen Netzen es zur Zeit gibt, um diesen Zusammenhang genauer zu betrachten.

3.1.3 Aktuelle Infrastruktur mobiler Systeme

Um eine konkrete Vorstellung von der Vielfalt und Leistungsfähigkeit aktueller mobiler Systeme zu bekommen, sollen im Folgenden typische Leistungsmerkmale gebräuchlicher Technologien vorgestellt werden. Zu beachten ist, dass insbesondere in diesem Technologiesegment der Fortschritt ungemein schnell voranschreitet. Deshalb kann dieser Überblick nur eine aktuelle Punktaufnahme in der

Entwicklung mobiler Systeme darstellen. Betrachtet werden mobile Geräte, drahtlose Netze sowie die Java 2 Micro Edition (J2ME) als Beispiel einer Entwicklungsplattform für mobile Systeme.

Mobile Geräte

Mobile Geräte unterstützen den Benutzer bei der Ausführung seiner Aufgaben und ermöglichen ihm ein flexibles Arbeiten an nahezu beliebigen Orten. Ihre Leistungsfähigkeit ist unterschiedlich und hängt in erster Linie von der Gerätegröße ab. In Anlehnung an Roth [Rot02] können mobile Geräte wie folgt kategorisiert werden:

Notebooks: Bei *Notebooks* handelt es sich um eine mobile Ausführung von Standardcomputern. Betriebssystem, Speicherkapazität und Rechenleistung sind mit denen von stationären Arbeitsplatzrechnern vergleichbar oder identisch. Daher sind für Standardcomputer entworfene Anwendungen in der Regel auch problemlos auf *Notebooks* einzusetzen. *Notebooks* haben jedoch ein deutlich kompakteres Erscheinungsbild, so dass sie ohne großen Aufwand an einen anderen Einsatzort getragen und dort benutzt werden können. Auf modernen Geräten können sowohl drahtlose als auch kabelgebundene Kommunikationsverbindungen verwendet werden. Eine Energieversorgung ist sowohl über ein Netzteil als dauerhafte Stromversorgung oder für den kurzfristigen mobilen Einsatz über einen eingebauten Akku gewährleistet [Rot02].

Personal Digital Assistants (PDAs): Bei *PDAs* handelt es sich um Universalgeräte, die bequem in der Hand gehalten und dabei bedient werden können [Rot02]. Sie verfügen zum Teil über ein individuelles Betriebssystem, für das nur eingeschränkt Anwendungen entwickelt werden können. In erster Linie handelt es sich hierbei um Organisations- und Verwaltungsfunktionen, wie zum Beispiel Termin- und Adressverwaltungsprogramme. Im Vergleich mit Standardcomputern ist die Leistungsfähigkeit von *PDAs* in Bezug auf Speicherkapazität und Rechengeschwindigkeit eher gering. Eine Anbindung an das Kommunikationsnetz erfolgt in der Regel drahtlos. Die Stromversorgung basiert auf der Verwendung von Akkus oder Batterien.

Javafähige Mobiltelefone: Bekanntester und zugleich einer der leistungsärmsten Vertreter der mobilen Geräte ist das *Mobiltelefon*. *Mobiltelefone* sind auf Sprachverbindungen spezialisierte Endgeräte und sind im Grunde nicht für eine erweiterte Programmierung geeignet. Ausnahme bilden *Mobiltelefone*, die über die speziell angepasste Java-Plattform J*ava 2 Micro Edition (J2ME)* verfügen, die es erlaubt, kleine Applikationen (sogenannte *Midlets*) auf dem *Mobiltelefon* auszuführen. *Mobiltelefone* sind über das Mobilfunknetz drahtlos an die Kommunikationsinfrastrukturen des Festnetzes angebunden und können über WAP bzw. TCP/IP auch für den Zugang zum Internet verwendet werden. Bei durchschnittlicher Nutzung ist die Energieversorgung über einen Akku oder über Batterien für einige Tage gewährleistet.

Smartphones: Programmierbare *Smartphones* verwirklichen einen hybriden Ansatz zwischen *PDA* und *Mobiltelefon*. Sie können Gespräche und Daten übertragen, sind aber bei gleicher Größe deutlich leistungsfähiger als ein normales *Mobiltelefon*. Ein großes Farbdisplay, das Potential, auf verschiedene Netzwerke zuzugreifen und die Möglichkeit Anwendungsprogramme ausführen zu können machen das *Smartphone* multifunktional einsetzbar. Oft beinhaltet das *Smartphone* auch noch eine Digitalkamera,

einen MP3-Player oder einen GPS-Empfänger. Diese zahlreichen Zusatzfunktionalitäten müssen jedoch auch die begrenzten Energieressourcen des *Smartphones* untereinander aufteilen. Insbesondere wenn Anwendungen gleichzeitig ausgeführt werden, wird viel Energie benötigt, was in einer verkürzten Akkulaufzeit resultiert.

Tabelle 1 stellt beispielhaft die typischen Eigenschaften mobiler Geräte auf dem heutigen Stand der Technik dar. Es wurde dazu jeweils ein aktueller Vertreter der genannten Kategorien *Notebook*, *PDA*, *Mobiltelefon* und *Smartphone* auf seine technischen Leistungsmerkmale untersucht. Ersichtlich wird, dass die Leistungsfähigkeit der betrachteten Geräte stark variiert und in der Regel mit abnehmender Gerätegröße sinkt.

	Acer TravelMate 4101LMi [1]	**HP iPAQ rx3715** [2]	**Motorola A1000** [3]	**Siemens S65** [4]
Art	Notebook	PDA	Smartphone	Mobiltelefon
Rechenleistung	1.60GHz	400 MHz	168 MHz	Java virtual processor speed: 33.4MHz [5]
Speicherkapazität	512MB DDR RAM, 60GB HDD	64 MB SDRAM, 128 MB ROM (erweiterbar mit bis zu 1 GB)	32 MB RAM, 22 MB Storage Memory (erweiterbar mit bis zu 256 MB)	10.31 MB interner Speicher, 32 MB externer Speicher (erweiterbar mit bis zu 512 MB)
Akkulaufzeit bei dauerhafter Benutzung	Max. 4 bis 5 h	Max. 3 h	Max. 3,75 h Gesprächszeit	Max. 5 h Gsprächszeit
Connectivity	LAN, WLAN, IrDa, Modem	WLAN, Bluetooth, IrDa	UMTS, GSM, GPRS, Bluetooth	GSM, GPRS, Bluetooth, IrDa
Betriebssystem	Windows XP	Windows Mobile 2003	Symbian 7.0 UIQ	proprietär
Programmier-barkeit	beliebig	J2ME Personal Profile, C++	J2ME MIDP 2.0	J2ME MIDP 2.0
Benutzerinterface	15.0" TFT Display, Auflösung 1024 x 768 Pixel, Tastatur, Touchpad	3,5" TFT Display, Auflösung 240x320 Pixel, Touch-Screen, 5-Wege-Navigations-Tasten	3" TFT Display, Auflösung 208x320 Pixel, Touch-Screen, Navigations-Tasten	2" TFT Display, Auflösung 132 x 176 Pixel, Nummern- Tastatur

Tabelle 1: Übersicht typischer Eigenschaften mobiler Geräte

Weitere für den mobilen Gebrauch entwickelte Geräte mit geringerer Systemleistung wie Pager oder elektronische Kalender sowie stark auf bestimmte Einsatzzwecke spezialisierte Systeme wie zum Beispiel Navigationsgeräte sollen in dieser Arbeit nicht betrachtet werden. Sie sind in der Regel nicht zur Installation alternativer Anwendungen geeignet und verfügen nicht über geeignete Programmierschnittstellen, um ihren Anwendungszweck zu ändern oder zu erweitern [Rot02].

1 Quelle: Datenblatt Acer TravelMate 4101LMi, http://www.acer.de, Markteinführung Sommer 2004
2 Quelle: Datenblatt HP iPAQ rx3715, http://www.hewlett-packard.de, Markteinführung Herbst 2004
3 Quelle: Produktbeschreibung Motorola A1000, http://my-symbian.com/uiq/a1000_review.php, Markteinführung Frühjahr 2005
4 Quelle: Datenblatt Siemens S65, http://www.t-mobile.de, Markteinführung Sommer 2004
5 Quelle: Testbericht http://www.club-java.com/TastePhone/J2ME/MIDP_Java_telephone.jsp?m=106&brand=Siemens& model=S65

Drahtlose Netze

Drahtlose Kommunikation beschreibt die Anbindung eines mobilen Endgeräts über eine Funk- oder Infrarotschnittstelle. Man unterscheidet Netze im allgemeinen nach ihrer räumlichen Ausdehnung. So können auch drahtlose Netzwerke durch ihre Verfügbarkeitsreichweite charakterisiert werden:

Mobilfunk: Mobilfunknetze sind für eine große Flächenabdeckung vorgesehen. Ihr primärer Einsatzzweck ist die Übertragung von Sprachsignalen für die Mobiltelefonie, während die Netze und dazugehörige Dienste heute mehr und mehr auch für die Datenübertragung optimiert werden.

- **Global System for Mobile Communication (GSM):** Der Mobilfunkstandard *GSM* ist nahezu weltweit anerkannt und stellt ein Mobilfunksystem der sogenannten zweiten Generation dar. Dieses ist hauptsächlich auf Sprachübertragung ausgelegt und arbeitet leitungsvermittelnd. Mit *GSM* wird nur eine relativ geringe Datenübertragungsrate von bis zu 9,6kbit/s erzielt [Schi03]. Eine Abrechnung der in Anspruch genommenen Leistungen erfolgt in der Regel nach Zeitvolumen.
- **General Packet Radio Service (GPRS):** *GPRS* ist für den Datentransfer entwickelt worden. Es ist paketvermittelnd und ermöglicht durch seine Architektur bereits Datenraten von bis zu 170kbit/s und eine direkte Anbindung mobiler Geräte an das Internet. Jedes *GPRS*-Gerät erhält eine IP-Adresse und kann unter Zuhilfenahme der Dienstknoten im Servicenetz IP-Datenpakete empfangen und versenden. Eine Abrechnung ist optional nach Zeit- oder Datenvolumen möglich [Schi03].
- **Universal Mobile Telecommunication System (UMTS):** *UMTS* stellt den ersten Vertreter einer dritten Generation von Mobilfunksystemen dar, die Datenraten von bis zu 2Mbit/s ermöglichen. Eine derart leistungsfähige *UMTS*-Funkzelle besitzt jedoch nur einen Radius von etwa 100m bis zu maximal 2km, was eine weitaus stärkere Überdeckung der Landschaft mit Funkmasten erfordert als zum Beispiel für *GSM*. Da dies aus Kostengründen nur in Ballungszentren wirtschaftlich erscheint, ist keine absolut flächendeckende Versorgung mit dieser Technik zu erwarten. *UMTS* ist wie *GPRS* paketvermittelnd [Eck03].

Lokale Funknetze: Lokale Funknetze haben nur eine räumlich eingeschränkte Verfügbarkeit. Ihr Einsatzzweck liegt in der Datenkommunikation über kurze Reichweiten durch überwiegend private Betreiber, Unternehmen und Organisationen. Zu den am meisten verwendeten lokalen Funknetzen gehören *WLAN*, *Infrarot* und *Bluetooth*:

- **Wireless Local Area Network (WLAN)**: *WLAN* ermöglicht die drahtlose Vernetzung von Gebäuden, die drahtlose Überbrückung von Entfernungen zwischen Gebäuden (zum Beispiel Campus einer Universität), oder ein drahtloses Angebot öffentlich nutzbarer Services in stark frequentierten Bereichen (zum Beispiel am Flughafen). Basierend auf dem IEEE 802.11 Standard sind zur Zeit Datenraten von bis zu 54Mbit/s und eine Signalreichweite bis zu 500m möglich. Ein Betrieb kann sowohl im *Infrastrukturmodus* (mittels Access Points) oder im *Ad-hoc-Modus* (spontan zwischen mehreren Rechnern innerhalb der Reichweite des Funksignals) erfolgen. Da die Funksignale auch feste Gegenstände und Mauern durchdringen, kann mit *WLAN* eine relativ große räumliche Ausdehnung abgedeckt werden [Schi03].
- **Personal Area Network (PAN)**: Ein *PAN* dient im Wesentlichen der Kopplung von Geräten des persönlichen Bedarfs. Notebooks, PDAs, Mobiltelefone sowie periphere Geräte wie Eingabegeräte oder Drucker nutzen gemeinsame Schnittstellen, um Daten auszutauschen oder zu synchronisieren. Die *Infrared Data Association (IrDA)* beschreibt dazu relativ simple Spezifikationen und Kom-

munikationsprotokolle zur optischen Datenübertragung mittels infrarotem Licht. Die Kommunikation funktioniert allerdings nur über sehr kurze Strecken mit Sichtverbindung. Die *IrDA*-Spezifikation sieht hier maximal 1m Reichweite vor. Zudem ist die Infrarot-Schnittstelle bei Tageslicht im Freien nicht zuverlässig zu verwenden [Schi03].
Die Infrarot-Übertragung wird nicht zuletzt wegen der genannten Nachteile immer weiter von der Funktechnologie *Bluetooth* verdrängt. Diese ist mit ihrem speziellen Fokus auf stromsparende Überbrückung kurzer Distanzen von bis zu 10m (Class 1) bei 1mW Verbrauch insbesondere für Mobiltelefone und periphere Anwendungen gut geeignet. Neuere Geräte können bis zu 50m bei 2mW (Class 2) abdecken und bis zu 100m bei 100mW (Class 3). Bis zu 8 *Bluetooth*-Geräte können sich außerdem spontan zu sogenannten *Piconetzen* zusammenfinden und mehrere solcher Piconetze können zu einem größeren Netz zusammengeschaltet werden [Eck03].

Netz	Reichweite	Datenrate	Sichtkontakt notwendig	Benutzungs-kosten
GSM	überregional	Bis zu 9,6 kbit/s	Nein	Ja
GPRS	überregional	Bis zu 170 kbit/s	Nein	Ja
UMTS	überregional	Bis zu 2 Mbit/s	Nein	Ja
WLAN	Bis zu 500 m	Bis zu 54 Mbit/s	Nein	i.d.R. Nicht
Bluetooth	Bis zu 100 m	Bis zu 1 Mbit/s	Nein	Nein
IrDa	Bis zu 1 m	Bis zu 4 Mbit/s	Ja	Nein

Tabelle 2: Überblick über Technologien zur Drahtlosvernetzung (nach [Schi03])

Tabelle 2 zeigt die Leistungsparameter der dargestellten Technologien in Übersichtsform. Es ist zu erkennen, dass mit Ausnahme von *WLAN* im Bereich der drahtlosen Netzwerke nur geringe Datenraten erzielt werden können. In vielen Netzen ist die Reichweite der Verfügbarkeit begrenzt oder der Zugang von einem Nutzungsentgelt abhängig. Die Verfügbarkeit und die Auswahl von geeigneten Netzen stellen also wichtige Aspekte zur erfolgreichen Kommunikation zwischen mobilen Geräten dar.

Entwicklungsplattform für mobile Systeme: Java 2 Micro Edition

Die *Java 2 Micro Edition (J2ME)* ist Teil der Java 2 Entwicklungsplattform und stellt eine spezielle Menge von Application Programming Interfaces (APIs) zur Programmierung von mobilen Geräten, Unterhaltungselektronik und eingebetteten Systemen zur Verfügung. Gegenüber der *Standard Edition (J2SE)* und der *Enterprise Edition (J2EE)* ist *J2ME* damit auf Geräte spezialisiert, die mit wenig Rechenleistung und einer geringen Speicherkapazität auskommen müssen. Gleichzeitig werden die besonderen Bedürfnisse dieser Gerätefamilien, wie die Anbindung an drahtlose Netzwerke und die Interaktion mit leistungsarmen Benutzerinterfaces, unterstützt [Whi01].

Sun Microsystems [Sun02] ordnet die *Micro Edition* logisch zwischen die *J2SE* und einer Plattform für die Programmierung von Chipkarten ein, so dass *J2ME* ein variabler, aber begrenzter Basisanteil benötigter APIs aus *J2SE* zur Verfügung steht. Diese Architektur wird durch ein mehrstufiges Schichtenmodell realisiert, dem eine geeignete virtuelle Maschine zugrunde liegt (Abb. 8).

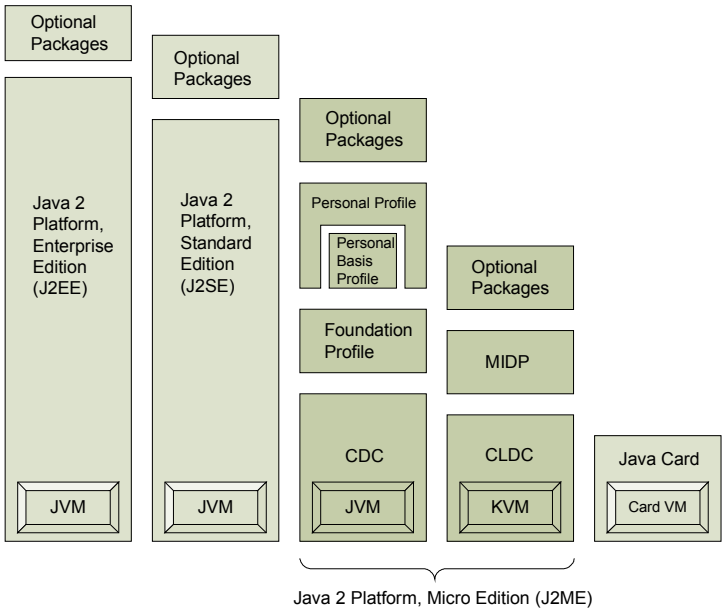


Abb. 8: Architektur und Einordnung von J2ME in die Java 2 Plattform (aus [Sun02])

Die komplette Laufzeitumgebung wird bei *J2ME* durch die Zusammensetzung sogenannter *Konfigurationen*, *Profile* und optionaler Packages nach Bedarf und nach den Leistungsmerkmalen der entsprechenden Hardware spezifiziert. Sun Microsystems [Sun02] beschreibt zwei Konfigurationen und eine Reihe von möglichen Profilen, die für verschiedene Gerätegruppen geeignet sind:

Konfigurationen: Eine *Konfiguration* stellt ein Paket aus einer virtuelle Maschine und einer minimalen Auswahl an Klassenbibliotheken zur Bereitstellung von Basisfunktionalitäten dar. Die Wahl einer *Konfiguration* hängt von der Leistungsfähigkeit des Gerätes ab:

1. Connected Device Configuration (CDC) ist für Geräte geeignet, die mindestens eine 32-bit CPU aufweisen können und über mehr als 2 MB Speicherplatz für die Java Plattform und entsprechende Anwendungsprogramme verfügen. Darunter fallen zum Beispiel Set-Top-Boxen, Verkehrstelematiksysteme oder High-End-PDAs. *CDC* enthält eine vollwertige Java Virtual Machine (JVM) und umfasst eine relativ große Bandbreite an Funktionalität [Whi01].

2. Connected Limited Device Configuration (CLDC) ist hingegen auf leistungsarme Geräte ausgerichtet, die nur über langsame Prozessoren und wenig Speicherkapazität verfügen. *CLDC* unterstützt auch 16- bis 32-bit CPUs und kann mit einem Minimum von 128 bis 512 KB Speicher auskommen. Im Gegensatz zu *CDC* basiert *CLDC* auf einer eingeschränkten virtuellen Maschine, der sogenannten „Kilobyte Virtual Machine“ (KVM) und enthält dementsprechend auch erheblich

weniger Funktionalität. Vor allem Mobiltelefone oder leistungsarme PDAs profitieren von den geringen technischen Ansprüchen dieser *Konfiguration* [Whi01].

Profile: *Profile* erweitern die *Konfigurationen*, indem sie die Gruppe der unterstützten Geräte genauer spezifizieren. Sie stellen APIs bereit, die unter anderem Schnittstellen zur Benutzerinteraktion, Persistenzmechanismen und gerätespezifische Eigenschaften definieren. Die von Sun Microsystems [Sun02] spezifizierten *Profile* sind jeweils an bestimmte *Konfigurationen* gekoppelt:

1. Mobile Information Device Profile (MIDP): Dieses mittlerweile in der zweiten Version vorliegende *Profil* wurde speziell für Mobiltelefone und PDAs entwickelt. Es ergänzt die *Konfiguration CLDC* und enthält detaillierte auf die Bedürfnisse dieser Gerätefamilie abgestimmte Funktionalitäten, unter anderem Unterstützung für deren Benutzerinterfaces, für die drahtlose Netzwerkanbindung, für die Datenspeicherung und für das Management von Anwendungsprogrammen [Sun02].

2. Foundation Profile (FP): *FP* ist ein Basis-*Profil* für die etwas anspruchsvollere Klasse der *Connected Device Configurations*. Hier können *Profile* nicht nur exklusiv, sondern auch auf einander aufbauend verwendet werden. *FP* stellt höheren Schichten von *Profilen* relativ geräteunabhängige Grunddienste zur Verfügung, wie zum Beispiel Netzwerkunterstützung. Eingebettete Systeme können dieses *Profil* nutzen, um ohne ein für sie überflüssiges Benutzerinterface auszukommen [Whi01].

3. Personal Basis Profile (PBP): Das *Personal Basis Profile* stellt Basisfunktionalitäten für die Graphikdarstellung bereit. Es ist für Geräte geeignet, die neben der Netzwerkfähigkeit einfache, standardisierte Benutzeroberflächen implementieren können [Whi01].

4. Personal Profile (PP): Das *Personal Profile* ist ein sowohl auf dem *Foundation Profile* als auch auf dem *Personal Basis Profile* aufbauendes *CDC-Profil*. Es richtet sich vor allem an Geräte, die ein komplexeres Benutzerinterface haben, so dass auch aufwändige GUIs angezeigt werden können. Zum Beispiel enthält *PP* das Abstract Window Toolkit (AWT). Zudem liefert es Unterstützung zur Ausführung web-basierter Applets. Leistungsfähige PDAs und Spielekonsolen gehören zu den Geräten, die das *PP* in Anspruch nehmen können [Whi01].

Optionale Packages: Die einzelnen Profile können durch modulare benutzerdefinierte Erweiterungen ergänzt werden. Optionale APIs sind zum Beispiel für die spezielle Benutzung von *Bluetooth*, *Web Services* oder für die Anbindung an Datenbanken verfügbar [Sun02].

Neben optionalen Packages besteht die Möglichkeit, eigene Profile zu definieren und damit besondere Geräteeigenschaften noch besser auszunutzen.

3.2 Grundlagen von Prozessbeschreibungen

An dieser Stelle sollen grundlegende Eigenschaften erläutert werden, die vielen Prozessbeschreibungssprachen gemeinsam sind. Zuerst erfolgt eine Erläuterung zu dem in dieser Arbeit verwendeten Verständnis von Prozessen und Prozessmodellen. Im Anschluss werden elementare Konstrukte von Prozessbeschreibungen genannt. Dazu werden die Grundprinzipien von Elementen zur Kontroll- und Datenflussregelung, zur Beschreibung von Prozessteilnehmern und Transaktionen sowie Konzepte zur Fehlerbehandlung vorgestellt. Abschließend wird die Interpretation der Prozessbeschreibung und die Ausführung von Prozessen betrachtet.

3.2.1 Prozessbegriff

Bei einem *Prozess* handelt es sich nach van der Aalst und van Hee [AaHe02] um eine Abfolge miteinander verknüpfter Aufgaben zur Erreichung eines übergeordneten Ziels. Ein Prozess enthält demnach die Menge der auszuführenden Aufgaben und eine Reihe von Bedingungen, die die Ausführungsreihenfolge dieser Aufgaben festlegen. Eine Aufgabe ist eine atomare logische Einheit, die einen Teilschritt des Prozesses darstellt. Jeder Teilschritt unterliegt der Verantwortung einer Ressource, die die Aufgabe bearbeitet oder die Bearbeitung administriert. Ressourcen stellen dabei generische Konstrukte für Personen oder Maschinen dar, die zur Ausführung von Aufgaben zur Verfügung stehen [AaHe02].

Prozesse erfordern in der Regel Eingaben in Form von Daten, die zur Bearbeitung von anstehenden Aufgaben notwendig sind und geben Ergebnisse aus, die dem nächsten Verarbeitungsschritt zur Verfügung gestellt werden oder das Endergebnis des Prozesses darstellen. Alternativ können Prozesse Zustandsänderungen ausführen, ohne Daten zurückzugeben [LeRo00].

In der Wirtschaft wird der Begriff des Prozesses oft synonym mit „Arbeitsablauf“ oder „Geschäftsvorgang“ verwendet. Diese Betrachtungsweise entspricht im Weiteren der Definition eines *Workflows*. Nach Leymann und Roller ist ein *Workflow* ein Prozess, der einen Geschäftsvorgang in einem Wirtschaftsunternehmen abbildet und mit Hilfe eines Computers ausgeführt wird. Ziel ist die Ausführung anliegender Aufgaben zur richtigen Zeit von der richtigen Instanz. Die Ausführung einer Aufgabe bzw. eines Teilschritts eines *Workflows* wird als *Aktivität* bezeichnet [LeRo00].

Workflows treten in unterschiedlichen Ausprägungen auf und lassen sich anhand ihrer Strukturierung und Planbarkeit kategorisieren. Dabei stellen sogenannte *Produktions-Workflows* relativ statische, stark strukturierte Prozesse dar, die sich häufig in gleicher oder ähnlicher Weise wiederholen. Sie repräsentieren insbesondere betriebliche Routine-Vorgänge und haben eine hohe Unternehmensrelevanz. *Ad-Hoc-Workflows* zeichnen sich hingegen durch eine hohe Dynamik aus. Sie sind im Gegensatz zu *Produktions-Workflows* wenig strukturiert und enthalten viele im Planungszeitraum des Prozesses unbekannte Komponenten. In der Regel handelt es sich um spontan auftretende Prozesse, die sich kaum in der gleichen Art und Weise wiederholen. *Semistrukturierte Workflows* vereinen einen hybriden Ansatz, der sowohl feste Strukturen als auch dynamische Komponenten enthält [BoSch98]. Tabelle 3 zeigt eine Übersicht dieser *Workflow*-Kategorien.

	Produktions-Workflow	Semi-Strukturierter Workflow	Ad-Hoc-Workflow
Vorgänge	Stark strukturiert	Teilweise vorstrukturiert	Teilweise vorstrukturiert, jedoch dynamisch änderbar
Typ der Aktivität	Im voraus beschreibbar	Teilweise im voraus beschreibbar	Nicht im voraus beschreibbar
Beziehungen zwischen Aktivitäten	Im voraus bekannt	Teilweise im voraus bekannt	Im voraus unbekannt
Informations-beziehungen zwischen Aktivitäten	Im voraus planbar	Teilweise im voraus planbar	Nur sehr unvollständig planbar
Zuständigkeiten und Verantwortung	Im voraus festlegbar	Nur teilweise festlegbar	Ergibt sich aus dem Ablauf

Tabelle 3: Vergleichende Übersicht von Workflow-Kategorien (aus [BoSch98])

Man unterscheidet weiterhin zwischen manuellen und automatisierten Aktivitäten eines Prozesses: Manuelle Aktivitäten erfordern Handlungen eines menschlichen Benutzers, während automatisierte Aktivitäten durch Softwareprogramme bearbeitet werden können. Die Ausführung eines Prozesses oder eines *Workflows* wird daher zusammenfassend von drei Dimensionen bestimmt: Der Prozesslogik, die beschreibt, welche Aktivitäten in welcher Reihenfolge ausgeführt werden sollen, der Organisationsstruktur der ausführenden Personen, zum Beispiel der Abteilungen in einem Unternehmen, und der am Prozess beteiligten Softwarekomponenten [LeRo00].

Bei einem Prozess im Sinne dieser Diplomarbeit handelt es sich um eine zeitlich-sachlogische Abfolge von Funktionen. Diese enthält eine detaillierte Festlegung der Ausführungsreihenfolge einzelner Teilschritte. Alle so festgelegten Aktivitäten verfolgen in der Regel gemeinsame Ziele, welche vom Initiator des Prozesses vorgegeben wurden. Vorgänge und Beziehungen zwischen Aktivitäten sollen im Normalfall nicht dynamisch veränderlich sein. Zuständigkeiten und Verantwortlichkeiten hingegen können jedoch erst zur Laufzeit geeigneten konkreten Instanzen zugewiesen werden. Dadurch ergibt sich eine Konzentration auf semistrukturierte Abläufe, die nicht zwingend repetitiv sein müssen.

Menschliche und technische Ausführungseinheiten sollen zusammenfassend als *Prozessteilnehmer* bezeichnet werden. In dieser Arbeit stellt ein Prozess insbesondere die Komposition voneinander unabhängiger Softwaredienste in den Vordergrund. Bei der Ausführung herrscht dabei ein hoher Verteilungsgrad, das heißt, es gibt keine zentrale Komponente zur Administration des Prozesses. Alle Teilnehmer müssen die Ausführung des Prozesses daher selbst untereinander regeln.

3.2.2 Prozessmodellierung

Die Prozessmodellierung beschäftigt sich mit der Identifikation, Analyse und Abbildung von realen Prozessketten auf ein Modell, welches zur Ausführung von Prozessen auf einem computergestützten

System geeignet ist. In der Regel handelt es sich bei diesen Prozessen um Geschäftsprozesse, die automatisiert oder teilautomatisiert bearbeitet werden können und dadurch eine schnellere, kostengünstigere und verlässlichere Durchführung versprechen [GeHo95] .

Ein *Prozessmodell* beschreibt allgemein die Struktur eines Prozesses, determiniert alle möglichen Pfade durch den Prozess hindurch und beinhaltet Regeln, die bestimmen, welcher Pfad verfolgt und welche Aktivitäten ausgeführt werden sollen. Diese Zusammenhänge werden entweder durch graphische oder durch sprachliche Ausdrücke beschrieben. Die so modellierten Beschreibungen stellen *Vorlagen (Templates)* für konkrete Prozesse dar, so dass das *Prozessmodell* im Anwendungsfall mit bestimmten Aufgaben, Daten und Objekten instantiiert und als konkreter Prozess ausgeführt werden kann [LeRo00]. Die Prozessbeschreibung wird dazu von einer Ausführungsumgebung interpretiert, die anstehende Aufgaben an geeignete Softwareprogramme oder Personen zuweist, die Abwicklung überwacht und die Ergebnisse weiterverarbeitet (vgl. 3.2.4).

Prozessmodelle können sowohl statische als auch dynamische Komponenten beinhalten. Die Prozesslogik mit der Festlegung der Aktivitäten und des Kontroll- und Datenflusses ist in der Regel unveränderbar festgelegt und damit eindeutig zu interpretieren. Hingegen können in der Organisationsstruktur unter den ausführenden Personen Änderungen auftreten: Mitarbeiter, die eine bestimmte Aufgabe bearbeiten sollen, können zum Beispiel im Urlaub sein oder die Abteilung gewechselt haben, so dass inzwischen andere Personen an ihre Stelle getreten sind [GeHo95]. Ebenso können Softwareprogramme, die eine Aufgabe erfüllen sollen, erst zur Laufzeit an den Prozess gebunden und zur Ausführung verpflichtet werden. Man unterscheidet in diesem Fall zwischen einer frühen Bindung, bei der das zuständige Programm bereits im Prozessmodell definiert wird, und einer späten Bindung, wo die Auswahl der Softwarekomponenten während des laufenden Prozesses und im Extremfall erst unmittelbar vor der Ausführung der betreffenden Aktivität festgelegt wird. Die späte Bindung hat sowohl im organisatorischen als auch im softwaretechnischen Bereich den Vorteil, dass Personen und Programme ausgetauscht oder geändert werden können, ohne dass eine Neubearbeitung des Prozessmodells notwendig wird [Emm03].

3.2.3 Prozessbeschreibungssprache

Die Beschreibung eines Prozesses muss genaue Angaben über die Art und Weise der Ausführung, der Weitergabe von Daten und der Definition von Prozessteilnehmern beinhalten. Dazu wird eine *Prozessbeschreibungsprache* benötigt, die ein Metamodell für die Beschreibung von Prozessmodellen darstellt. Sie enthält ein Repertoire von Konstrukten und Funktionen zur detaillierten Strukturierung eines Prozessmodells und zur Regelung des Kontroll- und Datenflusses einzelner Prozessinstanzen. Die Beschreibung, die festlegt, welche Teilschritte zur Ausführung benötigt werden und in welcher Reihenfolge diese ausgeführt werden sollen, wird als *Prozessdefinition* bezeichnet. Eine Prozessdefinition besteht in der Regel aus Aufgaben bzw. Aktivitäten und Bedingungskonstrukten, die die Ausführungsreihenfolge festlegen [AaHe]. Da einige Konstrukte für viele Prozessbeschreibungssprachen grundlegend sind, sollen diese im folgenden vorgestellt werden.

Kontrollflussregelung

Die Kontrollflussregelung beschreibt, wie die Ausführungsreihenfolge der zu erledigenden Aufgaben festgelegt ist. Sie setzt sich aus einer Menge von Konstrukten zusammen, die zu beliebigen Definitionen von Ablaufplänen kombiniert werden können.

Die Kontrollflusselemente der Prozessbeschreibung können entweder in einer *Blockstruktur* oder in einer *Graphstruktur* aufgebaut sein. In einer *Blockstruktur* werden Konstrukte geschachtelt und spezielle strukturierte Elemente stehen als Behälter zur Verfügung, um eine Ordnung zwischen den enthaltenen Aufgaben herzustellen. Die Ausführungsreihenfolge der graphstrukturierten Prozesse basiert hingegen auf Transitionen, die verbindende Übergänge zwischen den Aktivitäten darstellen. Dabei sind die Transitionen die Kanten und die Aktivitäten stellen die Knoten des Graphs dar [Sha01].

Aktivitäten: Eine *Aktivität* stellt in der Prozessbeschreibung eine bestimmte Aufgabe dar, die als Teilschritt des Prozesses bearbeitet werden soll. Dabei kann sowohl eine Person als auch eine technische Anlage wie eine Maschine oder ein Softwareprogramm an der Abarbeitung beteiligt sein. Einzelne *Aktivitäten* können auch selbst wieder aus mehreren Aufgaben bestehen – die *Aktivität* beinhaltet dann einen Block oder ist Platzhalter für einen Subprozess innerhalb des Prozesses [LeRo00].

Kontrollflusskonnektoren: In einer *Graphstruktur* werden die auszuführenden Aktivitäten untereinander durch *Kontrollflusskonnektoren* verknüpft, um eine potentielle Reihenfolge für deren Ausführung festzulegen. Man unterscheidet zwischen ausgehenden und eingehenden Konnektoren: Während die ausgehenden Konnektoren beschreiben, unter welchen Bedingungen eine Aufgabe als erledigt anzusehen ist, legen die eingehenden Konnektoren die als nächstes zu bearbeitende Aufgabe fest. *Aktivitäten* können durchaus mehrere eingehende und ausgehende *Kontrollflusskonnektoren* haben, die spezifizieren, ob Aufgaben sequentiell oder parallel ausgeführt werden sollen. Besitzt eine *Aktivität* keine eingehenden *Kontrollflusskonnektoren*, so ist sie als Startaktivität definiert. *Aktivitäten* ohne ausgehende Konnektoren signalisieren das Ende eines Prozesses [LeRo00].

Bedingungen: Der einfachste Anwendungsfall für die Abarbeitung eines Prozesses ist, dass eine unbedingte sequentielle Ausführungsreihenfolge zwischen zwei Aufgaben vorliegt [AaHe02]. Sind also zwei *Aktivitäten* A und B durch einen *Kontrollflusskonnektor* miteinander verbunden, so kann Aufgabe B erst ausgeführt werden, wenn Aufgabe A beendet worden ist (Abb. 9).

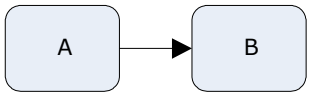


Abb. 9: Kontrollfluss: Sequenz

Wie bereits angedeutet können *Kontrollflusskonnektoren* jedoch zusätzlich mit *Bedingungen* verknüpft werden, um die Bearbeitungsfolge der Aufgaben weiter zu beeinflussen:

Zunächst einmal kann eine einfache sequentielle Ausführung von einer *Transitionsbedingung* abhängig gemacht werden. Diese beschreibt die Umstände, unter denen der Übergang von einer Aktivität zur nächsten erlaubt sein soll. Sind diese Umstände unzutreffend, wird die Bedingung zu „false“ ausgewertet und die nachstehende Aufgabe wird nicht ausgeführt (Abb. 10).

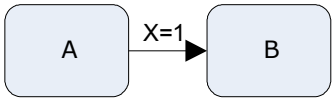


Abb. 10: Kontrollfluss: Transitionsbedingung

Weiterhin gibt es *Aktivierungsbedingungen*, die anhand bestimmter Vorgaben evaluieren, ob eine Aktivität gestartet werden soll (Abb. 11). Dies könnte zum Beispiel eine Zeitvorgabe sein: Beginne mit *Aktivität* A um 10:30 Uhr. *Aktivierungsbedingungen* werden erst nach etwaigen vorgelagerten *Join-Bedingungen* ausgewertet [LeRo00].

Die sogenannte *Exit-Bedingung* kann als Nachbedingung der Aktivität angesehen werden. Sie spezifiziert, unter welchen Umständen eine Aufgabe nach dessen Ausführung als erledigt anzusehen ist (Abb. 12). Zum Beispiel könnte geprüft werden, ob alle Ergebnisdaten in vorgegebener Art und Weise vorliegen [AGK+95].

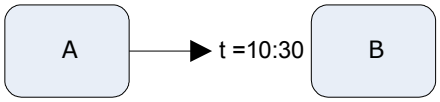


Abb. 11: Kontrollfluss: Aktivierungsbedingung

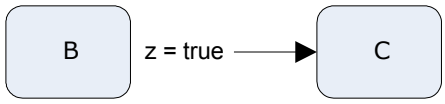


Abb. 12: Kontrollfluss: Exit-Bedingung

Wie bereits oben erwähnt, können voneinander unabhängige Aufgaben alternativ zu einer sequentiellen Ausführung auch parallel bearbeitet werden. Die *Split-Bedingung* (bei Leymann und Roller [LeRo00] auch als *Fork* bezeichnet) besagt, dass die nach der Abarbeitung der aktuellen *Aktivität* folgenden *Aktivitäten* parallel ausgeführt werden können. Dies bedeutet im Detail, dass die *Aktivitäten*, die einer *Split-Bedingung* folgen, gleichzeitig oder in einer beliebigen Reihenfolge ausgeführt werden können [AaHe02]. Die *Join-Bedingung* hingegen fungiert am eingehenden *Kontrollflusskonnektor* als Synchronisationspunkt parallel ausgeführter *Aktivitäten* und überwacht, ob alle spezifizierten Aufgaben so erfüllt worden sind, dass die nächste *Aktivität* gestartet werden kann (Abb. 13). *Split-* und *Join-Bedingungen* können zusätzlich mit booleschen Operatoren erweitert werden, die angeben, ob die aus- bzw. eingehenden Kontrollflusskonnektoren erfüllt werden müssen [AaHe02].

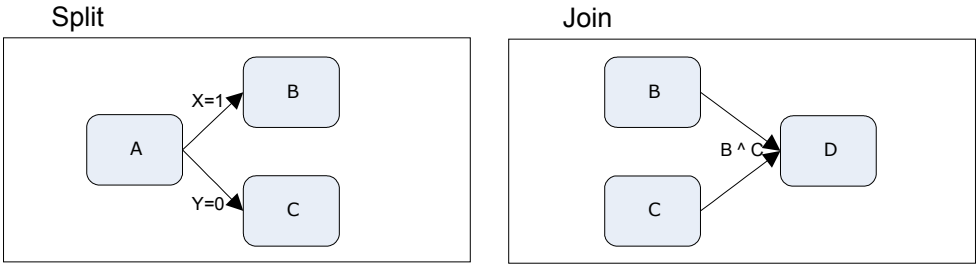


Abb. 13: Kontrollfluss: Parallele Ausführung durch Split und Join

Die vorgenannten Bedingungskonstrukte können in beliebiger Art kombiniert und zu komplexeren Formen zusammengesetzt werden. Im Prinzip können mit Hilfe der Bedingungskonstrukte alle möglichen Wahrheitswertkombinationen abgefragt werden.

Erweiterte Konstrukte: Um noch detaillierter Einfluss auf die Koordination und die Gestaltung von Ablaufplänen nehmen zu können, besteht die Möglichkeit, weitere Konstrukte auf dem Niveau von höheren Programmiersprachen zu definieren:

1. Iteration: Mit Hilfe der *Iteration* kann eine einzelne *Aktivität* oder eine Folge von *Aktivitäten* mehrmals durchlaufen werden (Abb. 14). Entweder kann eine definierte Anzahl an Durchläufen festgeschrieben werden oder die Schleife ist an eine bestimmte Bedingung geknüpft. Diese Art von Mehrfachausführung ist vergleichbar mit den Konstrukten „while ... do ...“ oder „repeat ... until ...“ aus imperativen Programmiersprachen, je nachdem ob die Abbruchbedingung am Anfang oder am Ende des Schleifenkörpers festgelegt ist [AaHe02].

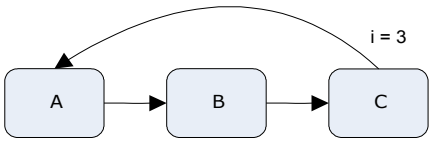


Abb. 14: Kontrollfluss: Beispiel für Iteration (hier: repeat ... until ...)

2. Subprozesse: *Subprozesse* entstehen immer dann, wenn eine aufgerufene *Aktivität* eine Komposition aus anderen Aufgaben benutzt, um ihre Arbeit zu verrichten. Dies ist insbesondere dann der Fall, wenn bereits bestehende autonome Prozesse in einem übergeordneten Prozess wiederverwendet werden. Der Grad der Unabhängigkeit zwischen aufrufendem Prozess und *Subprozess* kann jedoch stark variieren. Leymann [LeRo00] differenziert zwischen zwei Modellen von *Subprozessen*, dem *Connected Discrete Model* und dem *Hierarchical Model*. Bei *Subprozessen* nach dem *Connected Discrete Model* wird der *Subprozess* vollkommen autonom gestartet, während der Ursprungsprozess unverändert und ohne Rücksicht auf die Ergebnisse seines *Subprozesses* fortgeführt wird. Im Extremfall sind die beiden Prozesse einfach nur miteinander verknüpft, indem das Ergebnis von Prozess A die Ausführung von Prozess B anstößt. *Subprozess* und aufrufender Prozess stellen dadurch im eigentlichen Sinne zwei gleichwertige, verbundene Prozesse dar (Abb. 15).

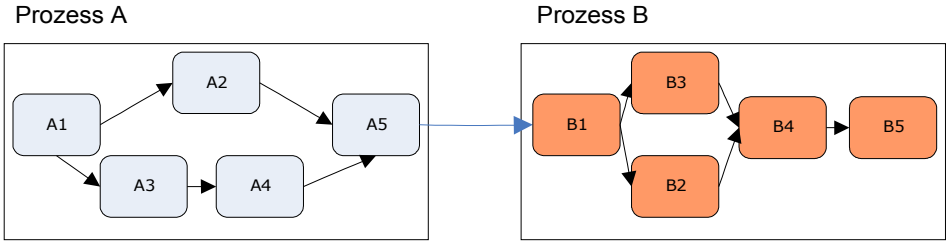


Abb. 15: Subprozess nach dem Connected Discrete Model (aus [LeRo00])

Im Fall des *Hierarchical Models* (Abb. 16) liefert der *Subprozess* ein Ergebnis an die aufrufende *Aktivität* zurück. Der Ursprungsprozess wird dadurch solange in seiner Ausführung blockiert, bis der *Subprozess* beendet ist und das Ergebnis seiner Ausführung übergibt. Daher handelt es sich hierbei um eine hierarchische Anordnung von Prozessen, die eine starke Abhängigkeit der Prozesse untereinander beschreibt [LeRo00].

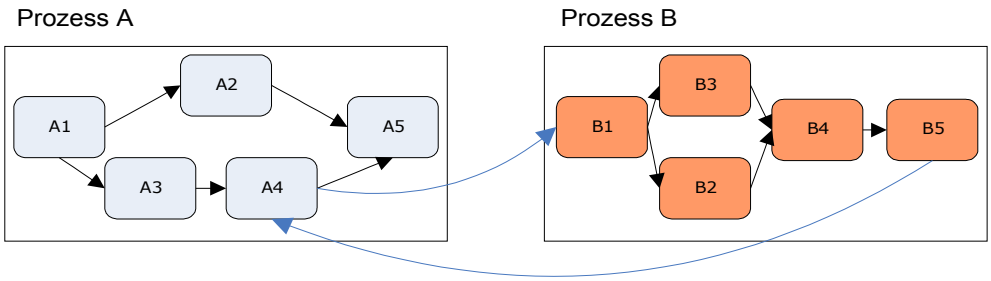


Abb. 16: Subprozesse: Hierarchical Model (aus [LeRo00])

Datenflussregelung

Werden zur Bearbeitung von Aktivitäten innerhalb eines Prozesses Daten oder Dokumente benötigt, so ist auch für diese eine entsprechende *Datenflussregelung* festzulegen. Daten, die für die Ausführung einer Aufgabe relevant sind und der Aktivität zugeführt werden müssen, werden als *Input-Daten* einer Aktivität bezeichnet. Sie werden in einem *Input-Container* der jeweiligen Aktivität gespeichert und bei Bedarf verarbeitet. Stellen die Daten hingegen ein Ergebnis einer Aktivität oder gar des Gesamtprozesses dar oder sollen sie von folgenden Aktivitäten weiterbearbeitet werden, so sind die Daten sogenannte *Output-Daten* der bearbeitenden Aktivität. Sie werden im *Output-Container* der Aktivität gespeichert, bis die Aktivität beendet ist oder die Daten von einer anderen Aktivität bezogen werden [LeRo00].

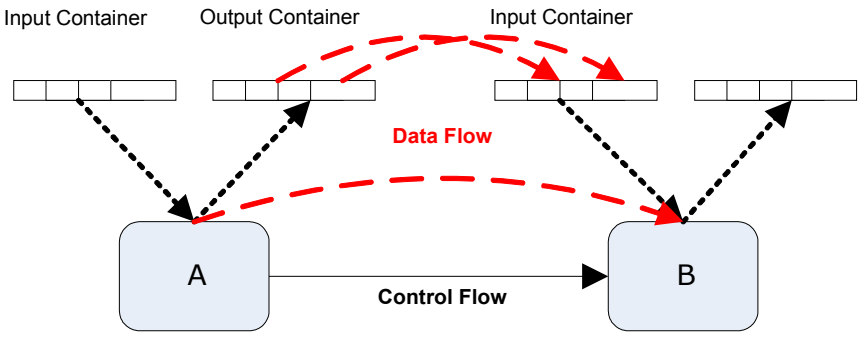


Abb. 17: Datenfluss zwischen zwei Aktivitäten (nach LeRo00)

Im einfachsten Fall ist die Kontrollstruktur des Datenflusses mit der des Kontrollflusses identisch (Abb. 17). Es kann jedoch Vorteile haben, den Datenfluss vom Kontrollfluss zu trennen, insbesondere dann, wenn bestimmte Daten nicht von allen Aktivitäten bearbeitet werden oder verschiedenen Prozessinstanzen gleichzeitig verfügbar gemacht werden sollen. Dies erhöht die Flexibilität der Ausführung, falls Daten erst an späterer Stelle des Prozesses wieder benötigt werden oder die Mitführung von komplexen Dokumenten über einen langen Zeitraum nicht realisierbar ist. Der Datenfluss bekommt in diesem Fall eigene von der Kontrollstruktur weitestgehend unabhängige Konnektoren, die sogenannten *Datenflusskonnektoren*, die die Zuteilung von Daten und Dokumenten an Aktivitäten des Prozesses beschreiben [LeRo00].

Beschreibung von Prozessteilnehmern

Ein weiterer wichtiger Bestandteil der Prozessbeschreibung ist die Definition und Beschreibung von Prozessteilnehmern. Prozessteilnehmer sind alle potentiellen Personen, Objekte und Programme, die an der Ausführung von Aufgaben oder an der Administration des Prozesses selbst beteiligt sein können. Eine Beschreibung von Prozessteilnehmern kann einerseits durch Angabe einer fixen Instanz (zum Beispiel „Herr Müller“) oder durch die Zuordnung von Instanzen zu Rollen und die Angabe einer Rolle (zum Beispiel „Mitarbeiter der Personalabteilung“) in der Prozessbeschreibung erfolgen. Wird die ausführende Instanz durch eine Rollenbezeichnung identifiziert, so kann flexibel zur Laufzeit festgelegt werden, welche tatsächliche Person oder welches konkrete Softwareprogramm zur Ausführung herangezogen werden soll [GeHo95]. Prozessteilnehmer können in dieser Arbeit unter anderem sein:

- konkrete Personen
- konkrete Softwareprogramme in Form von oben beschriebenen Diensten
- Rollen, die Personen eines bestimmten Nutzerkreises enthalten
- Beschreibungen von Diensten, die Softwarefunktionalität enthalten, wobei nur die Funktionalität, nicht aber der konkrete Dienstanbieter festgelegt ist.

Transaktionen

Um transaktionales Verhalten implementieren zu können, müssen sowohl die Schritte einer einzelnen Aktivität als auch im Sinne einer *Transaktion* zusammenhängende Aktivitäten als solche gekennzeichnet werden. Eine *Transaktion* ist allgemein eine Folge von Operationen, die zu einer logischen Einheit zusammengefasst sind und die über die sogenannten ACID-Eigenschaften verfügen [GrRe93]:

Atomicy (*Atomarität*, "Alles oder Nichts"): Eine *Transaktion* ist eine Folge von Verarbeitungsschritten, die nur gemeinsam oder gar nicht durchgeführt werden dürfen. Die Ausführung einer *Trans-*

aktion soll aus Sicht des Benutzers ununterbrechbar verlaufen. Wird die Bearbeitung aus irgendeinem Grund vor dem Erreichen des Transaktionsendes abgebrochen, so muss der Bearbeitungsstand auf den Zustand vor Beginn der *Transaktion* zurückgesetzt werden (*Rollback*) [BeNe97].

Consistency (*Konsistenz*): *Transaktionen* gehen von einem konsistenten Zustand aus und müssen ihrerseits einen konsistenten Zustand erzeugen. Zu beachten ist, dass die Konsistenz im Allgemeinen nur vor und nach der Ausführung einer *Transaktion* gewährleistet wird. Während einer *Transaktion* dagegen können temporäre Konsistenzverletzungen eintreten [BeNe97].

Isolation (*Isolation*): *Transaktionen* sind isoliert, wenn sie serialisierbar sind und damit keine unerwünschten Nebenwirkungen zwischen den Aufgaben eintreten. Die isolierte Ausführung einer *Transaktion* ist äquivalent zu einer serialisierten Ausführung von Aktionen und hinterlässt demzufolge einen konsistenten Zustand [BeNe97].

Durability (*Dauerhaftigkeit*): Nach Erreichen des Transaktionsendes werden die Daten durch Bestätigung (*Commit*) permanent gespeichert. Transaktionale Änderungen an Datenbeständen dürfen jedoch erst dauerhaft gespeichert werden, wenn die *Transaktion* vollständig durchlaufen ist. Dann allerdings sollen Änderungen dieser *Transaktion* auch alle künftigen Fehler überstehen, insbesondere auch Systemabstürze oder Speicherausfälle [BeNe97].

Die Anwendung von traditionellen Transaktionsmodellen wie zum Beispiel des *Zwei-Phasen-Commit-Protokolls (2PC)* (vgl. [BeNe97]) auf Prozesse gestaltet sich allerdings problematisch, da Fehler bei der Ausführung von Aufgaben eventuell erst lange nach Abschluss der *Transaktion* bekannt werden. Im Fall der mobilen Systeme kommt hinzu, dass kein zentraler Koordinator zur Durchführung der *Transaktion* zur Verfügung steht bzw. die Koordinatoren während der Ausführung einer *Transaktion* wechseln können. Das bedeutet, dass der Initiator einer *Transaktion* diese nicht zwingend auch koordiniert und die Kontrolle nach Abschluss der *Transaktion* nicht in jedem Fall an den Initiator zurückgegeben wird. Es kommt in diesem Fall zu Abhängigkeiten zwischen dem konkreten Kontrollfluss und der Ausführung der *Transaktion*.

Auch Sperrmechanismen (zum Beispiel das *Zwei-Phasen-Sperrprotokoll*) können nur begrenzt angewendet werden, da es sich bei Aktivitäten innerhalb eines komplexen Prozesses mit einem Großteil an Benutzerinteraktion meistens um zeitlich länger andauernde Aufgaben handelt und bearbeitete Daten eventuell freigegeben werden müssen, bevor die *Transaktion* abgeschlossen wurde [SGSP04]. *Transaktionen* müssen zudem in der Lage sein, Unterbrechungen durch Fehler oder durch beabsichtigte Vertagung einer bestimmten Aufgabe zu überstehen. Diese Anforderungen erfordern neue Transaktionsmodelle, die die ACID-Eigenschaften teilweise lockern können und daher besser zur Anwendung für Prozessmanagementsysteme geeignet sind. *Transaktionen* mit den genannten Eigenschaften und erweiterten Möglichkeiten zur Fehlerbehandlung werden als *Long Running (Business) Transactions* (*LRTs*) bezeichnet [ACD+03].

Leymann und Roller [LeRo00] schlagen für Workflow-Management-Systeme die Konzepte der *Atomic Sphere* und der *Compensation Sphere* sowie das Modell der *Open Nested Transactions* vor:

Atomic Sphere: Eine *Atomic Sphere* stellt eine Menge von Aktivitäten dar, die nur gemeinsam oder gar nicht durchgeführt werden dürfen. Dabei stellt jede beteiligte Aktivität selbst eine *Transaktion* dar. Ziel ist die Wiederverwendbarkeit von Einzeltransaktionen in einer globalen *Transaktion*. Der Kontrollfluss der Aktivitäten innerhalb einer *Atomic Sphere* darf daher keine Aktivitäten beinhalten, die nicht Teil der *Atomic Sphere* selbst sind. Zudem müssen alle Kontrollflusskonnektoren, die in die *Ato-*

mic Sphere hineinführen, die gleiche Aktivität zum Ursprung haben (Abb. 18). Die *Transaktion* wird gestartet, sobald der Kontrollfluss die *Atomic Sphere* erreicht. Verlässt der Kontrollfluss die *Atomic Sphere* wieder, so wird abgewartet, bis alle darin enthaltenen Aktivitäten erfolgreich durchgeführt worden sind, um die *Transaktion* abzuschließen. Muss eine Aktivität innerhalb der *Atomic Sphere* zurückgesetzt werden, so müssen auch alle zuvor bearbeiteten Aktivitäten der *Atomic Sphere* auf ihren ursprünglichen Zustand zurückgesetzt werden (*Rollback*). Für das atomare *Commitment* wird dabei zum Beispiel das *Zwei-Phasen-Commit-Protokoll* verwendet [LeRo00]. Da atomare *Transaktionen* Sperrmechanismen für die verwendeten Ressourcen verwenden, können *Atomic Spheres* nur für kurz andauernde Aktivitäten oder Aktivitäten mit geringen Interdepenzen untereinander angewendet werden.

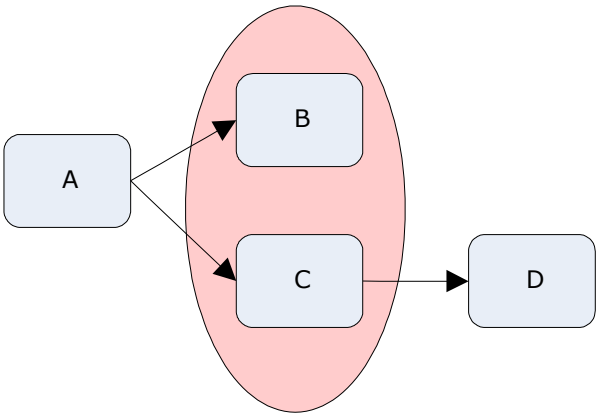


Abb. 18: Transaktionen: Atomic Sphere (nach [GrRe93])

Compensation Sphere: *Compensation Spheres* richten sich an *Transaktionen*, die bereits abgeschlossen wurden und deren Ursprungszustand im Fehlerfall nicht ohne weiteres wiederhergestellt werden kann. Insbesondere, wenn Fehler erst lange nach Abschluss der teilnehmenden Aktivitäten festgestellt werden, sind Daten über konkrete Zustände vor der Bearbeitung in der Regel nicht mehr verfügbar oder sie wurden inzwischen durch andere Aktivitäten verändert. Daher muss die fehlerhafte Ausführung durch Kompensation der durchgeführten Aktivitäten semantisch korrigiert werden. *Compensation Spheres* enthalten dazu eine Menge von Aktivitäten, die gemeinsam ausgeführt werden sollen, können aber im Gegensatz zu *Atomic Spheres* auch nicht-transaktionale Aufgaben innerhalb des Kontrollflusses beinhalten. Zu jeder innerhalb einer *Compensation Sphere* definierten Aktivität gibt es zwecks Kompensation eine Aktion, die diese Aktivität semantisch rückgängig machen kann. Diese Kompensations-Aktion kann entweder auch eine einzelne Aktivität oder sogar ein komplexer Prozess sein. Kann nun eine Aktivität innerhalb der *Compensation Sphere* nicht erfolgreich durchgeführt werden oder tritt ein Fehler auf, so werden alle anderen bereits ausgeführten Aktivitäten dieser *Compensation Sphere* durch Aufruf ihrer jeweiligen Kompensationsaktivität rückgängig gemacht (Abb. 19). Die passenden Kompensationsfunktionen müssen dazu in der Prozessbeschreibung deklariert werden [LeRo00]. Das Konzept der Kompensation von Aktivitäten einer fehlgeschlagenen Transaktion eignet sich zur Unterstützung von *Long Running Transactions*, da hier kaum Sperrmechanismen verwendet werden müssen.

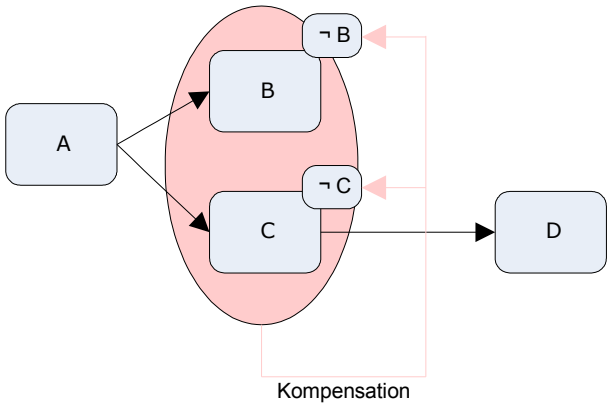


Abb. 19: Transaktionen: Compensation Sphere (nach [GrRe93])

Schließlich sollte noch erwähnt werden, dass es Aktivitäten gibt, deren Ausführung in der Realität nicht ohne weiteres rückgängig zu machen ist. Zum Beispiel ist die Stornierung einer Reisebuchung häufig mit Stornogebühren verbunden. Hier ist eine besondere Behandlungsweise erforderlich, zum Beispiel durch zusätzliche Benutzerinteraktion. Auch kann es insbesondere im Bereich der manuellen Ausführung vorkommen, dass geleistete Arbeit überhaupt nicht rückgängig zu machen ist, zum Beispiel die Bearbeitung eines Werkstücks oder die Ausführung einer Dienstleistung. Derartige Aktivitäten können in diesem Fall nicht Teil einer zu kompensierenden *Transaktion* sein.

Open Nested Transactions: *Open Nested Transactions* relativieren die ACID-Forderung nach "Isolation", indem Untermengen einer *Transaktion* auf verschiedenen Ebenen geschachtelt werden. So setzt sich eine *Open Nested Transaction* aus weiteren *Transaktionen* zusammen, die entweder atomar sein können oder wiederum eingebettete Transaktionen enthalten können (Abb. 20). Änderungen an abgeschlossenen Aktivitäten werden sofort persistent gemacht und für eine Aktion erforderliche Ressourcen nur für kurze Zeit gesperrt und nach Beendigung schnellstmöglichst wieder freigegeben. *Open Nested Transactions* eignen sich daher für *Transaktionen*, die nicht in einer für die Anwendung von Sperrmechanismen vertretbaren Zeitdauer abgeschlossen werden können. Da konsistente Zwischenzustände der *Transaktion* gespeichert werden, sind sie außerdem relativ resistent gegen Systemabstürze. Diese Art von Sicherung wird als *Savepoint* bezeichnet. Kommt es innerhalb einer *Open Nested Transaction* zu einem Fehler, so müssen Recovery-Maßnahmen eingeleitet werden, um sowohl die bereits abgeschlossenen eingebetteten *Transaktionen* zu kompensieren, als auch ein *Rollback* auf der noch nicht abgeschlossenen Elterntransaktion durchzuführen. Dazu werden zunächst die noch offenen Subtransaktionen zurückgesetzt und danach die bereits abgeschlossenen Aktivitäten in umgekehrter Ausführungsreihenfolge kompensiert [GrRe93].

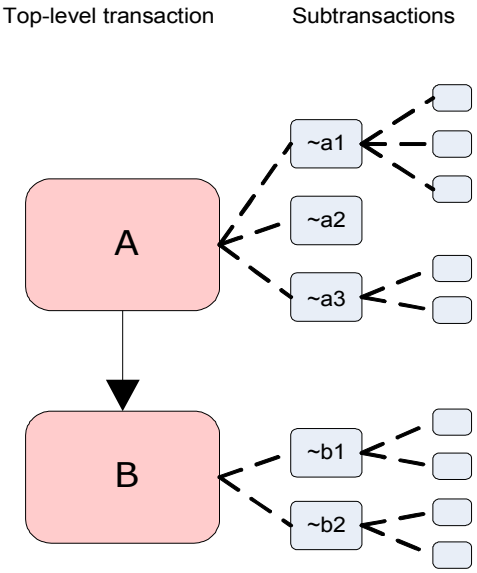


Abb. 20: Open Nested Transaction (nach [GrRe93])

Fehlerbehandlung

Da oft auf Ausnahmesituationen reagiert werden muss, sind Konstrukte zur Fehlerbehandlung nahezu unerlässlich. Fehler und Ausnahmen treten immer dann auf, wenn sich Situationen ereignen, die Abweichungen vom geplanten zum tatsächlichen Geschehenen darstellen oder die nicht in der Prozessbeschreibung modelliert sind. Werden Aktivitäten demzufolge nicht oder nur mangelhaft erfüllt, werden zunächst einmal Mechanismen zur Aufdeckung der fehlerhaften Ausführung benötigt. Hierzu zählt zum Beispiel die Definition von Deadlines und Timeouts für die Bearbeitung einer speziellen Aufgabe oder für die Durchführung eines gesamten Prozesses. Liegt innerhalb der vorgegebenen Zeit kein Ergebnis der Aktivität oder des Prozesses vor, so kann der Vorgang als gescheitert gekennzeichnet werden und es können weitere fehlerbehandelnde Maßnahmen getroffen werden [LSKM00].
Eine einfache Art der weiterführenden Ausnahmebehandlung stellt die Wiederholung gescheiterter Aktivitäten dar. Damit unerfüllbare Aufgaben erkannt werden, kann die Anzahl der möglichen Wiederholungen beschränkt werden. Stehen die Aktivitäten nicht in direktem Zusammenhang mit anderen nachfolgenden Aufgaben, so besteht außerdem die Möglichkeit, gescheiterte Aktivitäten abzubrechen oder zu überspringen, um zumindest die Ausführbarkeit des Restprozesses zu sichern [LeRo00].

Die Nicht-Vorhersehbarkeit von Ereignissen macht aber auch eine mögliche Interaktion mit Benutzern im Fehlerfall interessant. Hinweise auf Fehler im Prozess ermöglichen es, die Fehlerquelle aufzudecken, den Benutzer in Recovery-Maßnahmen einzubeziehen und können am Ende sogar helfen, Schwachstellen im Prozess aufzudecken und damit den Prozess selbst nachhaltig zu verbessern [LSKM00].

3.2.4 Ausführung von Prozessen

Um formulierte und instantiierte Prozesse computergestützt ausführen zu können ist eine entsprechende Ausführungsumgebung nötig, die Prozessbeschreibungen interpretiert und darin enthaltene Aufgaben an zuständige Personen oder Softwarekomponenten zuweist. Diese Ausführungsumgebung wird vor allem in der Wirtschaft als *Workflow-Engine* bezeichnet und ist dort Teil des sogenannten *Workflow-Management-Systems (WFMS)*, welches neben der Ausführungsumgebung die Prozessbeschreibungssprache, eine zentrale Datenbank und Tools zur Administration beinhaltet [AaHe02].

Das Abarbeitungsmodell des Prozesses wird als *Navigation* bezeichnet. Es beschreibt, wie der Prozess interpretiert wird und nach welchen Kriterien er ausgeführt werden soll. So wird festgelegt, dass Prozesse an einer oder mehreren Startaktivitäten begonnen werden können und dass nach Ausführung einer Aktivität dem ausgehenden Kontrollflusskonnektor zur nächsten Aktivität gefolgt werden soll [LeRo00].

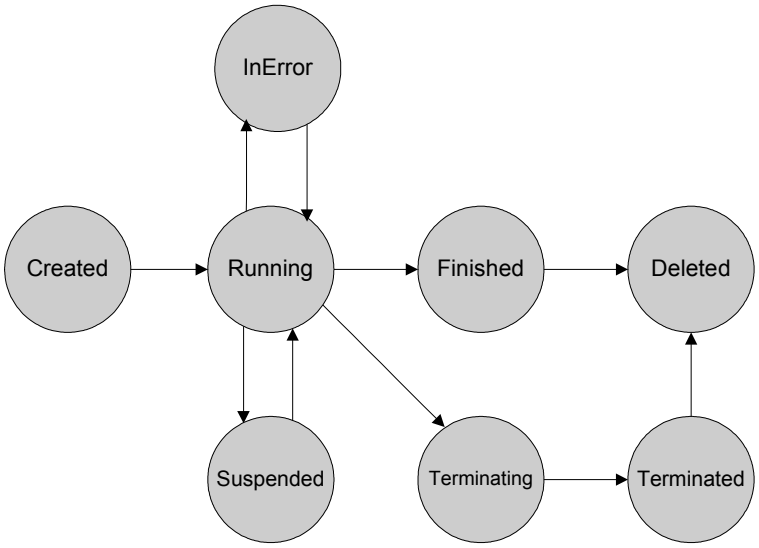


Abb. 21: Prozess-Lebenszyklus (aus [LeRo00])

Der Lebenszyklus eines Prozesses kann vereinfacht durch die Angabe seiner möglichen Zustände und der Zustandsübergänge beschrieben werden (Abb. 21):

- **Created**: Der *Created*-Zustand ist der Anfangszustand eines Prozesses nach dessen Erzeugung und Instantiierung.
- **Running**: In diesem Zustand erfolgt die eigentliche Ausführung des Prozesses. Er wird erreicht, indem der Prozess explizit durch den Request eines Benutzers gestartet oder als Subprozess von einem anderen Prozess aufgerufen wird.
- **Finished**: Alle beschriebenen Aktivitäten wurden bearbeitet und der Prozess hat seinen vordefinierten Endzustand erreicht. Gegebenenfalls wird an dieser Stelle ein Ergebnis ausgegeben.

- **Suspended**: Im *Suspended*-Zustand wird der Prozess vorübergehend angehalten. Er kann durch einen *Resume*-Request weitergeführt werden.
- **Terminating**: In diesem Zustand wird der Abbruch des Prozesses vorbereitet. Das bedeutet, dass gewartet wird, bis alle laufenden Aktivitäten abgeschlossen sind, aber keine neuen Aktivitäten mehr gestartet werden.
- **Terminated**: Der *Terminated*-Zustand wird erreicht, sobald alle Aktivitäten beendet wurden. Der Prozess wird nicht weiter ausgeführt.
- **InError**: Der Prozess wurde aufgrund eines Fehlers angehalten. Entsprechend definierte Fehlerbehandlungsmaßnahmen können des Prozess wieder in den *Running*-Zustand versetzen.
- **Deleted**: Der abgeschlossene oder abgebrochene Prozess wird aus dem System entfernt. Ressourcen, die vom Prozess verwendet wurden, werden wieder freigegeben [LeRo00].

Das Ausführungssystem ist neben der Abarbeitung des Prozesses auch für die Auswertung von Bedingungen zuständig. Sind an den Kontrollflusskonnektoren Bedingungen spezifiziert, so müssen diese zu „true“ oder „false“ evaluiert werden. Die Ausführung der folgenden Aufgaben wird genau dann gestartet, wenn der entsprechende Kontrollflusskonnektor zu „true“ ausgewertet worden ist. Im Navigationsmodell ist daher auch festgelegt, ob die Abarbeitung des Prozesses an einer *Join-Bedingung* anhält und wartet, bis alle eingehenden Kontrollflusskonnektoren evaluiert worden sind oder ob es reicht, eine bestimmte Anzahl von Evaluationen durchzuführen, um mit der Bearbeitung der nächsten Aktivität fortzufahren. Zum Beispiel könnte eine *Join-Bedingung*, die beinhaltet, dass zwei Drittel von vorgelagerten Aktivitäten erfüllt sein müssen, schon zu „true“ evaluieren, wenn zwei von drei Kontrollflusskonnektoren den Wahrheitswert „true“ aufweisen, auch wenn die Evaluation des dritten Kontrollflusskonnektors noch aussteht [LeRo00].

Die Ausführungsumgebung muss nicht nur die korrekte Abarbeitung der Prozesse sichern, sondern auch auf Fehlersituationen und mögliche Verklemmungssituationen reagieren können. Das Konzept der *Dead Path Elimination* beschäftigt sich daher mit der Vermeidung von unerwünschtem Stillstand der Abarbeitung eines Prozesses. Dieses in erster Linie von Bedingungen an Kontrollflusskonnektoren verursachte Verhalten kann bewirken, dass Prozesse unkontrolliert an beliebiger Stelle anhalten und nachfolgende Aktivitäten nicht mehr ausgeführt werden, ohne dass der Prozess explizit beendet wird. Wird zum Beispiel eine *Join-Bedingung* zu „false“ ausgewertet, so ist bisher nur spezifiziert worden, dass nachfolgende Aktivitäten auf diesem Pfad innerhalb des Prozesses nicht mehr ge-startet werden dürfen, was einer unzulänglichen impliziten Beendigung dieses Pfades oder sogar des ganzen Prozesses entspricht. Bei der *Dead Path Elimination* werden deshalb in so einem Fall die ausgehenden Kontrollflusskonnektoren der nachgelagerten Aktivitäten auf „false“ gesetzt, bis eine erneute Evaluation oder eine Endaktivität des Prozesses erreicht wird. Somit bleibt der Prozess nicht durch eine negative Evaluation stehen und es kann am Ende ein Ergebnis dargestellt werden, welches im schlimmsten Fall nur aus der Evaluation nach „false“ besteht [ADH03].

Insbesondere für Wirtschaftsunternehmen ist es zusätzlich von Vorteil, die Ausführung von Prozessen aufzuzeichnen und zu analysieren. Zum einen wird ein Nachweis über konkrete Verantwortlichkeiten für ausgeführte Aufgaben zur leistungsorientierten Entlohnung von Mitarbeitern benötigt, zum anderen können problematische Aktivitäten identifiziert und die Ausführung des Gesamtprozesses in Hinblick

auf Kosten und Geschwindigkeit optimiert werden [Pri03]. Aber auch aus Gesichtspunkten eines möglichen Backups im Fehlerfall oder der Rekonstruktion von ursprünglichen Zuständen zwecks Recovery-Maßnahmen ist eine Speicherung des Prozessverlaufs sinnvoll. Da Prozesse unterschiedliche Laufzeiten von einigen wenigen Sekunden bis im Extremfall zu mehreren Jahren aufweisen können, ist es notwendig, Prozesse jederzeit unterbrechen zu können, ohne die Kenntnis der bereits abgearbeiteten Teilschritte zu verlieren und zu beliebigen späteren Zeitpunkten wieder an den aktuellen Stand der Prozessbearbeitung anknüpfen zu können. Diese Art von Protokollierung relevanter Prozessereignisse wird als *Audit Trail* bezeichnet [LeRo00]. Wichtige Daten sind zum Beispiel Datum und Zeit der Ausführung sowie Identifikatoren von Aktivitäten, Prozessteilnehmern und Ereignissen [Pri03].

3.3 Nicht-funktionale Aspekte

Nicht-funktionale Aspekte sind qualitative Eigenschaften eines Systems oder eines Dienstes. Sie charakterisieren ein Objekt über seine eigentliche Funktion hinaus und geben Aufschluss über dessen Benutzbarkeit in Bezug auf spezielle Anforderungen. Entsprechen diese nicht-funktionalen Anforderungen eines bestimmten Leistungsnehmers nicht den vorliegenden Merkmalen, so wird für diesen die Funktion des Systems oder des Gerätes eingeschränkt oder sogar nutzlos. Edmond, Hofstede und O`Sullivan [OSEd03, EOH02] definieren klassische nicht-funktionale Aspekte für die Benutzung von Diensten, wie zum von Beispiel von Web Services:

Verfügbarkeit: Ein entscheidendes Kriterium eines Dienstes stellt dessen Verfügbarkeit dar. Dienste, deren Funktionen nicht erreichbar sind, können nicht in Anspruch genommen werden. Edmond und O`Sullivan [OSEd03] differenzieren zusätzlich zwischen zeitlicher und räumlicher Verfügbarkeit. Der mögliche Zeitpunkt und die Dauer der Dienstausführung sowie aller damit zusammenhängender Vorgänge, wie Verhandlung, Bezahlung, Unterbrechung oder endgültiger Abbruch des Dienstes, werden unter zeitlicher Verfügbarkeit zusammengefasst. Unter räumlicher Verfügbarkeit versteht man Informationen über Orte, an denen der Dienst angefragt, aufgerufen, ausgeführt und bezahlt werden kann.

Der Verfügbarkeitsaspekt ist gerade bei mobilen Systemen sehr relevant. So können mobile Geräte oder Kommunikationsmedien entweder nur an einem bestimmten Ort zur Verfügung stehen oder sie sind ausschließlich zu einer angegebenen Zeit erreichbar. In vielen Fällen sind räumliche und zeitliche Aspekte hier sogar gekoppelt: Zum Beispiel ist ein mobiles Gerät mit temporärer Anbindung an ein lokales Netz nur regional und nur für eine begrenzte Zeitdauer verfügbar.

Zugangskanäle (Channels): Anbieter und Nachfrager eines Dienstes einigen sich auf bestimmte Zugangskanäle, über die der Dienst aufgerufen werden kann. Zum Beispiel kann bei einem Bankunternehmen der Kontostand am Schalter, telefonisch oder über das Internet abgefragt werden [EOH02].

Bei mobilen Systemen kann unter anderem die Art der Netzwerkverbindung als Zugangskanal zu mobilen Geräten verstanden werden.

Sicherheit und Vertrauen: Beim Sicherheitsaspekt steht der Schutz von Identität, Privatsphäre und sensiblen Daten durch sichere Übertragungskanäle und Verschlüsselungsmechanismen im Vordergrund. Sicherheitskritische Anwendungen müssen zudem vor Fälschungen, Änderungen und unerlaubtem Zugriff bewahrt werden. Vertrauen hingegen wird in erster Linie durch das abzuschätzende Risiko einer Transaktion definiert und begründet sich auf Erfahrungen des Benutzers und die Transparenz einer Dienstleistung. Dabei spielen insbesondere die Identitäten der Serviceanbieter und die bereitgestellten Sicherheitsmechanismen eine große Rolle [EOH02].

Gerade im sicherheitsgefährdeten mobilen Bereich ist die Beachtung von Sicherheitskriterien und die Bereitstellung entsprechender Maßnahmen zur Gewährleistung einer sicheren Kommunikation maßgeblich. Verschlüsselungs- und Authentifikationsmechanismen können einige der genannten Mängel der mobilen Systeme kompensieren (vgl. Kapitel 3.1.2). Die Bildung von Vertrauen hängt hier insbesondere von der Identität der mobilen Benutzer ab. Zum Beispiel werden sich Gruppen, deren Mitglieder untereinander bekannt sind, in der Regel ein höheres Maß an Vertrauen entgegenbringen als völlig fremde Personen.

Rechte: Für den Benutzer einer Leistung kann es unter Umständen relevant sein, welche Rechte er bei der Ausführung eines Dienstes besitzt. Nicht-funktionale Eigenschaften sind die Möglichkeiten des Rücktritts von der Inanspruchnahme, des vorzeitigen Abbruchs, der Aussetzung des Dienstes und dessen Wiederaufnahme [EOH02].

Art der Vertragsbindung: Der Service-Vertrag regelt generell alle gegenseitigen Verpflichtungen zwischen Service-Provider und Konsument. Die Art der Vertragsbindung gibt dabei zum Beispiel an, ob es sich um eine einmalige Leistung oder um ein Abonnement handelt [EOH02].

Abrechnung und Bezahlung: Insbesondere die konkreten Kosten eines Dienstes sind ein wichtiger Aspekt bei der Auswahl von Leistungen. Aber auch Abrechnungsverfahren, wie zum Beispiel die Abrechnung nach Zeit- oder Mengeneinheiten und mögliche Zahlungsarten sind beeinflussende Faktoren [EOH02]. Bei mobilen Systemen sind zusätzlich zu den direkten Kosten des Dienstes auch die anfallenden Nebenkosten in Form von Nutzungsgebühren für Netzanbindung oder Mobilfunk ein wichtiger Aspekt.

Qualität: Die Qualität stellt allgemein das Verhältnis zwischen einer vom Kunden erwarteten und der tatsächlich vom Provider gelieferten Leistung dar. Sie umfasst alle funktionalen und nicht-funktionalen Eigenschaften, begründet sich jedoch auf eine subjektive, nicht messbare Wahrnehmung. Um diesem Problem entgegenzutreten, können sich Dienstanbieter in Rahmen von *Service-Level-Agreements* verpflichten, ein verabredetes Maß an Qualität zu liefern und dafür eine Garantie auf die Dienstleistung gewähren. Die Gewährung von Garantien stellt dabei auch selbst wieder ein Qualitätsmerkmal dar [EOH02].

Diese Auflistung ist keinesfalls erschöpfend und kann anwendungsspezifisch ergänzt werden. In dieser Arbeit werden insbesondere die für mobile Systeme relevanten nicht-funktionalen Aspekte berücksichtigt, während die eher dienstbezogenen Kriterien aus Komplexitätsgründen nicht weiter verfolgt werden. Nicht-funktionale Kriterien im Sinne dieser Arbeit sind demnach:

- Verfügbarkeit
- Zugangskanäle

- Sicherheitsmechanismen
- Kosten

Zusätzlich sollen neben den genannten Aspekten auch weitere Gesichtspunkte, die neben der eigentlichen Funktion über die Benutzbarkeit eines Systems entscheiden, in die Reihe der nicht-funktionalen Aspekte einbezogen werden. Im Zusammenhang mit mobilen Systemen wären das alle diejenigen Leistungsmerkmale eines mobilen Systems, die Qualitätsmerkmale für den Benutzer darstellen. Ergänzende nicht-funktionale Aspekte im weiteren Sinne sind zum Beispiel:

- Rechengeschwindigkeit
- Speicherkapazität
- Verfügbare Energiereserven
- Verbindungsqualität
- Zugehörigkeit zu Gruppen oder Rollen
- Spezielle Eigenschaften und Fähigkeiten, die zur Teilnahme am Prozess benötigt werden, zum Beispiel Transaktionsfähigkeit oder ein graphisches Benutzerinterface.

3.4 Anforderungen an eine Prozessbeschreibungssprache für mobile Systeme

In diesem Abschnitt werden die Anforderungen an eine Prozessbeschreibungssprache für mobile Systeme festgelegt. Zunächst einmal werden die Anforderungen an die Funktion der zu definierenden Prozessbeschreibungssprache durch ihr potentielles Einsatzgebiet als Teil des *Process Services* in der DEMAC-Architektur determiniert. Die dort gemachten Vorgaben werden dann im weiteren durch die allgemeinen Anforderungen an Prozessbeschreibungssprachen und die speziellen Rahmenbedingungen mobiler Systeme ergänzt.

3.4.1 Anforderungen aus dem DEMAC Projekt

Die folgenden Anforderungen entstehen durch die Aufgabenstellung des DEMAC-Projektes und die Eingrenzung dieser Arbeit. Sie stellen grundlegende Rahmenbedingungen dar.

D1 **Beschreibung und Ausführung komponierter automatisierter Services:** Es soll eine Prozessbeschreibungssprache für die Komposition automatisierter Services entwickelt werden, die für mobile Systeme geeignet ist. Der *Process Service* soll in der Lage sein, diese Prozessbeschreibungen zu interpretieren und so eine Ausführung komponierter automatisierter Services auf mobilen Systemen zu ermöglichen [Kun05].

D2 **Weitergabe von Prozessen an andere Ausführungseinheiten:** Die Ausführungsumgebung der Prozessbeschreibung soll in der Lage sein, Prozesse an andere Ausführungseinheiten weiterzugeben, falls in der Umgebung keine geeigneten Dienste zur Ausführung einer Aufgabe existieren oder die begrenzten Ressourcen des mobilen Geräts eine Ausführung der Aufgabe und damit eine Weiterbearbeitung des Prozesses nicht zulassen [Kun05].

D3 **Auswahl von Prozessteilnehmern durch benutzerdefinierte nicht-funktionale Kriterien:** Die Auswahl potentieller Dienste und Prozesspartner soll durch benutzerdefinierte nicht-funktionale Kriterien beeinflusst werden, die bestimmen, welche qualitativen Eigenschaften potentielle Dienstanbieter und Prozessteilnehmer aufweisen müssen. Die Beschreibungssprache benötigt also Konstrukte, die die Einbindung von nicht-funktionalen Aspekten in die Prozessbeschreibung ermöglichen [Kun05]. Dabei soll die Formulierung von geforderten Eigenschaften sowohl für einzelne Dienste getrennt als auch global für den ganzen Prozess festgelegt werden können. Zum Beispiel können sich Ansprüche an Sicherheitsmaßnahmen auf einen einzelnen Dienst, wie eine Kreditkartentransaktion beschränken oder die gesamte Struktur des Prozesses umfassen, um dessen Ablauf vor ungewollter Einsichtnahme oder Manipulation zu schützen. Die nicht-funktionalen Kriterien beziehen sich auf die in Kapitel 3.3 vorgegebenen Aspekte.

Als weitere Verfeinerung dieser Vorgehensweise sollte eine mögliche Priorisierung und Relativierung von Anforderungen angestrebt werden, um bei der Beschreibung geforderter Qualitäten eine möglichst hohe Flexibilität erreichen zu können. Dieses umfasst mögliche Konstrukte wie

- Vergleichsoperationen: Zum Beispiel soll je nach Situation das optimale und kostengünstigste Zugangsnetz genutzt werden: GSM/GPRS/UMTS von unterwegs, ADSL verbunden mit Funkzugangstechnologien wie Bluetooth/WLAN von stationären Einrichtungen.
- Höchst-/Mindestanforderungen: Zum Beispiel muss einem Dienst oder einem Prozess mindestens eine Datenrate von 144 kbit/s zur Kommunikation zur Verfügung stehen oder ein Dienst darf nicht mehr als 10 Geldeinheiten kosten.
- Alternativen: Ein Dienst soll entweder Bedingung A oder Bedingung B erfüllen.
- Priorisierung: Eine Verschlüsselung sensibler Daten könnte zum Beispiel wichtiger sein, als eine schnelle Übertragung.
- Relativierung: Konstrukte, die aussagen, ob die nicht-funktionale Eigenschaft schwerer wiegt als als die erfolgreiche Ausführung des Prozesses oder ob die Anforderung aufgegeben werden soll, wenn damit der Fortgang der Ausführung gesichert werden kann.

D4 **Kompatibilität zu bestehenden Ansätzen:** Die Prozessbeschreibungssprache soll möglichst zu anderen bestehenden Ansätzen kompatibel sein oder im Idealfall eine Untermenge einer etablierten Sprache bilden [Kun05].

3.4.2 Allgemeine Anforderungen an Prozessbeschreibungssprachen

In diesem Abschnitt soll ein kurzer Einblick darüber vermittelt werden, welche allgemeinen Anforderungen an Sprachen zur Prozessbeschreibung bestehen. Dabei ist die Prozessbeschreibung auch von Anforderungen und Eigenschaften ihrer Ausführungsumgebung abhängig, da dort implementierte Konzepte notwendigerweise an bestimmte Beschreibungen gekoppelt sind. Leymann und Roller [LeRo00] nennen Erfordernisse an ein System zum Prozessmanagement, die jedoch nur auszugsweise für die Anwendung auf einem mobilen System zutreffend sind.

P1 **Kontrollflusskonstrukte:** Die Prozessbeschreibungssprache muss ein Mindestmaß an Kontrollflusskonstrukten zur Verfügung stellen, die es erlauben, Reihenfolge und Bedingungen auszuführender Aktivitäten zu definieren. Es muss also mindestens ein Konstrukt zur einfachen sequentiellen Ausführung und zur Definition von Bedingungen existieren, so dass mehrere Aufgaben in einen sinnvollen Zusammenhang gebracht werden können.

P2 **Datenflussbeschreibung**: Es muss beschrieben werden können, wie sich der Datenfluss in Abhängigkeit vom Kontrollfluss verhalten soll. Eingangs- und Ergebnisdaten einer Aktivität müssen gekennzeichnet werden können. Eine mögliche Trennung von Kontroll- und Datenfluss kann die Flexibilität der Ausführung zusätzlich erhöhen.

P3 **Spezifikation von Prozessteilnehmern:** Konkrete oder variable Prozessteilnehmer müssen durch Angabe ihres Namens oder ihrer Rolle spezifiziert werden, damit Aufgaben an passende Personen oder Softwarekomponenten zugewiesen werden können. Prozessteilnehmer dienen hierbei entweder zur Ausführung einer bestimmten Aufgabe oder zur Weiterbearbeitung des Prozesses selbst.

P4 **Beschreibung von Fehlerbehandlungsmaßnahmen:** Es sind Maßnahmen zur Fehlerbehandlung notwendig, um auf Ausnahmesituationen reagieren zu können. Die Prozessbeschreibungssprache muss zusammen mit ihrer Ausführungsumgebung garantieren, dass der Prozess, so wie er definiert wurde, auch ausgeführt werden kann. Aktivitäten müssen genau so oft ausgeführt werden, wie es ursprünglich geplant war. Im Fehlerfall muss sichergestellt sein, dass keine korrekt ausgeführte Aktion rückgängig gemacht oder wiederholt wird und dass alle Aktionen, deren Ausführung wegen des Fehlers nicht vollendet werden konnte, kompensiert und neu gestartet werden. Diese Anforderung ist auch für die Ausführung von Prozessen auf den inhärent fehleranfälligen mobilen Systemen wichtig, insbesondere für die Behandlung von Verbindungsabbrüchen.

P5 **Dauerhafte Verfügbarkeit des Prozess-Systems:** Das Prozessmanagementsystem soll permanent erreichbar sein und so eine Verfügbarkeit und Bearbeitung der anliegenden Aufgaben rund um die Uhr ermöglichen. Diese Anforderung ist bei einem verteilten Prozess-System auf mobilen Systemen ohne zentrale Administrationsinstanz in der Regel nicht realisierbar, da es ja gerade in der Eigenschaft *ad-hoc vernetzter* mobiler Geräte liegt, keine feste Infrastruktur zu bilden und damit das System nicht dauerhaft zu unterstützen. Im Bereich des *Nomadic Computings* bedarf es spezieller Konventionen, um eine dauerhafte Verfügbarkeit aller teilnehmenden Prozessteilnehmer zu gewährleisten. In beiden Fällen können mobile Prozessteilnehmer theoretisch beliebig lange voneinander getrennt sein und müssen danach trotzdem in der Lage sein, die Ausführung des Prozesses fortzusetzen.

P6 **Skalierbarkeit**: Das System soll für eine unbeschränkt hohe Nutzer- und Teilnehmerzahl ausgelegt sein und die Ausführung einer Vielzahl von Prozessen ermöglichen. Diese Art der Skalierbarkeit kann von mobilen Geräten mit begrenzten Ressourcen nur beschränkt gefordert werden. In der Regel ist jedes mobile Gerät nur einem einzigen Benutzer zugeordnet, der hierdurch adressiert werden kann. Die Unterstützung einer großen Menge von Aufgaben und Dokumenten kann von den speicherarmen Geräten nicht gewährleistet werden. Es wäre sicherlich utopisch anzunehmen, dass Tausende von Prozessen gleichzeitig von einem leistungsarmen Gerät wie einem Mobiltelefon bearbeitet werden können. Andererseits ist die Forderung nach Skalierbarkeit jedoch auch so auszulegen, dass das Ausführen von Prozessen auch bei einer steigenden Zahl an mobilen Geräten, die am Prozess teilnehmen, möglich sein muss. Dieses ist jedoch gleichzeitig mit einer steigenden Zahl an Diensterbringern und Ausführungseinheiten verbunden, die die erfolgreiche Ausführung von Prozessen eher erleichtern als erschweren.

P7 **Erweiterbarkeit**: Prozessbeschreibungen und Ausführungsumgebung müssen erweiterbar sein. Zusätzlich erforderliche Ressourcen oder Features dürfen nicht eine Neukonzeption des Prozessmanagementsystems erforderlich machen, sondern müssen ergänzend oder modifizierend eingearbeitet werden können.

Trotz aller vorherrschenden technischen und funktionalen Beschränkungen muss auch eine Prozessbeschreibungssprache für mobile Systeme für Neuerungen erweiterbar bleiben. Nach dem Gesetz von Moore verdoppelt sich die Leistungsfähigkeit von Computern etwa alle 18 Monate, und nach Mattern [Mat03] gelte dies auch für Effizienzsteigerungen von anderen Technologieparameter wie Speicherkapazität oder Kommunikationsbandbreite. So ist auch der Fortschritt im Bereich der Mobiltechnologie immer noch als überproportional anzusehen. Sogar in der Batterietechnik ist eine moderate Steigerung der Kapazität von 5% pro Jahr möglich [Mat03]. Steigende Nutzerzahlen, zusätzliche Erweiterungen sowie das Auftauchen neuer Bedürfnisse und die Anpassbarkeit an weitere Beschränkungen können die Erweiterbarkeit des Prozess-Systems daher zu einem der wichtigsten Benutzbarkeitskriterien machen [DuGa03].

P8 **Sicherheit**: Die Prozessmodelle müssen gegen versehentliche oder absichtliche Änderungen geschützt werden. Zugangskontrollen sollen sicherstellen, dass der Inhalt und die Struktur insbesondere von sensiblen Geschäftsprozessen nicht unbefugten Benutzern offengelegt und zugänglich gemacht wird und auch nicht durch bösartige Absichten modifiziert werden kann.

P9 **Unterstützung von Transaktionen:** Die Prozessbeschreibungssprache muss sowohl transaktionales als auch nicht-transaktionales Verhalten unterstützen. Transaktionen müssen also solche gekennzeichnet werden können und durch spezielle Transaktionsmodelle unterstützt werden.

Die Behandlung von Transaktionen muss zudem für eine Anwendung im Bereich mobiler Systeme an die besonderen Umstände wechselnder Umgebungen angepasst werden. Prozessorientierte Transaktionsmodelle wie das Konzept der Kompensation können in Umgebungen mit häufigen Verbindungsabbrüchen nicht ohne Überarbeitung angewendet werden, da eventuell Einheiten, die Aktivitäten kompensieren müssen, zum Zeitpunkt der Rücksetzung nicht mehr zu erreichen sind. Transaktionen müssen außerdem daraufhin ausgelegt sein, dass einzelne Vorgänge sehr lange Zeiträume in Anspruch nehmen können, besonders, wenn menschliche Benutzer integriert sind.

P10 **Plattformunabhängigkeit:** Die Prozessbeschreibungssprache muss verschiedene Plattformen unterstützen und unabhängig von Betriebssystemen und Netzwerkprotokollen sein. Verschiedene Prozessmanagementsysteme sollten untereinander kompatibel sein, um unternehmens- und her-

stellerübergreifende Workflows zu ermöglichen. Diese Anforderung gilt auch insbesondere für eine Prozessbeschreibungssprache für mobile Systeme, wo viele verschiedenartige und vor allem proprietäre Betriebssysteme und Anwendungen im Einsatz sind (vgl. 3.1.3).

P11 **Zentrales System zur Administration:** Zu Verwaltungszwecken soll ein zentrales System zur Verfügung stehen, um Prozesse und Ausführungseinheiten zu administrieren. In einem Szenario mit hohem Verteilungsgrad wie in der Anwendung mobiler Prozessausführung existiert keine unabhängige zentrale Komponente. Verantwortlich für die Verwaltungs- und Koordinationsaufgaben ist jeweils das mobile System, welches momentan die Kontrolle über den Prozess hat. Die Administration zwischen diesen Prozessteilnehmern muss verteilt durchgeführt werden und kann folglich nicht die gleichen Leistungsmerkmale bieten wie ein zentralisiertes System.

P12 **Audit Trail:** Die Ausführungsumgebung sollte die Fähigkeit besitzen, Informationen zur Ausführung des Prozesses inklusive aller Teilschritte aufzuzeichnen. Diese sollen persistent gespeichert werden und so dauerhaft für eine Überwachung oder Analyse des Prozesses zur Verfügung stehen. Um eine unnötige Belastung mit Daten zu vermeiden, ist eine Option vorzusehen, mit deren Hilfe für jedes Prozessmodell spezifiziert werden kann, welche Information im Audit Trail gesichert werden sollen.

Das Speichern von kompletten Prozessinformationen in Form eines Protokolls auf mobilen Geräten scheint in der beschriebenen Form unrealisierbar zu sein. Schon alleine wegen der Verteilung des Prozesses auf verschiedene Geräte würde es zu einer Zerteilung des Protokolls kommen, welches eine sinnvolle Analyse der Daten so gut wie unmöglich macht. Zudem stellen mobile Geräte nicht genug Speicher bereit, um auch noch mit Informationen aus vergangenen Prozessen belastet werden zu können. Eine temporäre Speicherung von Zuständen und Ergebnissen des Prozesses bis zum Prozessende ist hingegen empfehlenswert, um in Fehlerfällen die Wiederaufnahme des Prozesses ermöglichen zu können.

3.4.3 Anforderungen aus den Eigenschaften mobiler Systeme

Für eine Prozessbeschreibungssprache, die primär für die Komposition von Diensten auf mobilen Systemen eingesetzt werden soll, lassen sich ergänzend zu den genannten allgemeinen Anforderungen an Prozessbeschreibungssprachen weitere wesentliche Anforderungen aus den speziellen Eigenschaften und Einschränkungen mobiler Systeme ableiten. Die folgenden Anforderungen resultieren aus den in Kapitel 3.1.2 genannten Aspekten.

M1 **Geringer Speicherverbrauch:** Die Prozessbeschreibung darf nur einen geringen Speicherverbrauch aufweisen. Diese Anforderung resultiert zum einen aus der beschränkte Speicherkapazität von mobilen Geräten, zum anderen aus deren relativ leistungsarmen drahtlosen Verbindungen, über die die Prozessbeschreibung übertragen werden muss. Zusätzlich muss berücksichtigt werden, dass die Ausführungsumgebung den verfügbaren Speicherplatz des Geräts nicht exklusiv

in Anspruch nehmen darf, da natürlich auch die eigentlichen Anwendungsprogramme noch auf dem Gerät Platz finden müssen. Daher sollte die Prozessbeschreibung möglichst auf elementare Elemente beschränkt sein sowie eventuell die Möglichkeit aufweisen, bereits abgearbeitete Prozesse aus der Beschreibung zu entfernen, um Speicherplatz zu sparen.

M2 **Geringer Anspruch an Rechenleistung:** Die mobile Ausführungsumgebung, die die Prozessbeschreibung interpretiert und verarbeitet, muss weitestgehend von komplexen Rechenvorgängen entlastet werden. Außerdem muss noch genügend Rechenleistung verbleiben, um die eigentlichen Funktionen und Dienste des mobilen Systems aufrecht erhalten zu können. Zum Beispiel darf der Benutzer eines Mobiltelefons nicht durch eine zu starke Belegung des Prozessors an der Möglichkeit, Telefongespräche zu führen, gehindert werden.

M3 **Verantwortlicher Umgang mit Energieressourcen:** Der Prozessinterpreter muss in der Lage sein, möglichst energiesparend zu arbeiten und bei Bedarf die Ausführung eines Prozesses zu einem späteren Zeitpunkt fortzusetzen, falls das mobile Gerät abgeschaltet werden muss. Während Leerlaufzeiten sollte der durch die Prozessverarbeitung resultierende Energieverbrauch minimal sein, um die beschränkten Ressourcen des mobilen Geräts zu schonen. Im Idealfall sollten mit dem Stromnetz verbundene Geräte in unmittelbarer Nähe genutzt werden, um Rechenleistung zu übernehmen oder Übertragungen weiterzuleiten [Mat03].

M4 **Lokale Datenhaltung:** Für die Ausführung des Prozesses benötigte Daten sollten lokal auf dem mobilen Gerät gespeichert werden können, um Problemen in der Verfügbarkeit vorzubeugen.

M5 **Geringer Kommunikationsaufwand:** Netzwerke für drahtlose Kommunikation verfügen über eine geringe Bandbreite und zudem ist ihre Nutzung meist kostenpflichtig. Der Kommunikationsaufwand, der durch die übertragenen Prozessbeschreibungen und durch den Austausch von Nachrichten zur Kooperation verursacht wird, muss also möglichst gering gehalten werden.

M6 **Behandlung von Verbindungsabbrüchen:** Besondere Berücksichtigung bedarf die Behandlung von möglichen Verbindungsabbrüchen. Da mobile Systeme im Regelfall nicht permanent miteinander verbunden sind, müssen Konzepte zur Synchronisation und zum Datenaustausch unter unsicheren Bedingungen entwickelt werden. Durch Verbindungsabbrüche und eine spätere Wiederaufnahme des Prozesses kann sich dessen Ausführung zudem zeitlich erheblich verzögern. Kontextdaten sind gegebenenfalls nach der Wiederherstellung der Verbindung geändert oder nicht mehr vorhanden [Jac04]. Voraussetzung für die Bearbeitung von Prozessen mit unzuverlässigen oder nicht verfügbaren Verbindungen ist eine erweiterte Fehlerbehandlung, die spezielle Konstrukte für den Fall eines plötzlichen Verbindungsabbruchs bereitstellt.

M7 **Synchronisation**: Eine weitere Anforderung stellt in dem genannten Zusammenhang die Notwendigkeit der Synchronisation mobiler Systeme dar. So müssen nicht nur Daten, die für die Bearbeitung von Aufgaben benötigt werden, während das mobile System nicht mit dem Netzwerk verbunden ist, synchronisiert werden, sondern auch die Zusammenarbeit mit anderen Prozessteilnehmern koordiniert werden. Parallel ausgeführte Aktivitäten oder hierarchische Subprozesse müssen zusammengeführt und Ergebnisse zu einer bestimmten Zeit an einem bestimmten Ort übergeben werden. [SGSP04] empfiehlt zum Beispiel die Verwendung von Zeitstempeln, um Konflikte zwischen verschiedenen Versionen von Daten und Zuständen zu vermeiden.

M8 **Priorisierung**: Forman [For98] hat sich damit beschäftigt, bei unbekannten Ressourcenverfügbarkeiten die Antwortzeiten für die Kommunikation von mobilen Systemen zu optimieren und da-

mit Schwankungen von Verbindungsqualitäten auszugleichen. Er schlägt vor, Aufgaben zu priorisieren und die Allokation von unwichtigeren oder erst später benötigte Daten zugunsten von relevanten Daten zu verzögern. Veraltete oder fehlgeschlagene Vorgänge sollen abgebrochen werden können, um Ressourcen für neuere Aktionen freizugeben.

M9 **Ergänzende Sicherheitsmechanismen**: Aufgrund der höheren Sicherheitserfordernisse von drahtlosen Netzen sollte es möglich sein, Prozessbeschreibungen und mitgeführte sensible Daten durch Verschlüsselung zu schützen. Eine Überprüfung der Identität eines Nutzers oder eines Kommunikationspartners kann zudem gewährleisten, dass Prozesse nicht von unautorisierten Individuen verfälscht und verlorene oder gestohlene Geräte nicht missbraucht werden können. Insbesondere die Nicht-Abstreitbarkeit von durchgeführten Aktivitäten und Transaktionen stellt eine wichtige Bedingung für den Einsatz in betrieblichen Prozess-Systemen dar. Die Urheberschaft des Zugriffs bzw. der Aktion sollte auch im Nachhinein noch eindeutig der ausführenden Einheit zurechenbar sein [Eck03]. Ferner muss eine unerwünschte Speicherung von Profilen und nutzerbezogenen Daten über Zugriffs-, Kommunikations und Bewegungsverhalten vermieden werden, um auch den Bedürfnissen der Benutzer nach Privatsphäre nachzukommen [Eck03]. Jacobsen [Jac04] schlägt vor, benutzerbezogene Orts- und Profildaten anwendungsindividuell aktivieren oder abschalten zu können.

M10 **Auswahl von variablen Diensten und Prozessteilnehmern zur Laufzeit:** Da die Mobilität eine dynamische Veränderung der Systemumgebung mit sich bringt, müssen variable Prozessparameter flexibel anpassbar sein. Die Ausführungseinheit muss über geeignete Mittel verfügen, über Veränderungen in ihrer Umgebung informiert zu werden, um angemessen darauf reagieren zu können. Dienste und Personen, die am Prozess und an der Erfüllung von Aufgaben beteiligt sind, können eventuell erst zur Laufzeit unmittelbar vor der Ausführung einer Aufgabe festgelegt werden, wenn konkrete Informationen über deren Verfügbarkeit vorliegen. Während der Ausführung der Prozessbeschreibung kann daher nur die jeweils aktuelle Aufgabe betrachtet werden, es besteht kein Wissen über spätere Aktivitäten oder Anforderungen. Integrierte Dienste müssen daher in der Prozessbeschreibung abstrakt beschrieben werden, ohne eine Kenntnis darüber zu verlangen, wie sie intern operieren.

M11 **Benutzerintegration**: Da mobile Systeme den primären Zweck erfüllen, ihren Benutzer an beliebigen Orten zu unterstützen, ist es notwendig, Schnittstellen für eine mögliche Benutzerintegration zu schaffen. Es muss davon ausgegangen werden, dass insbesondere Prozesse, die sich an mobile Benutzer richten, Aktivitäten beinhalten, welche manuell ausgeführt werden müssen oder zumindest Eingaben des Benutzers in einem Dialog erfordern. Dazu muss die Möglichkeit bestehen, neben Diensten und Geräten auch bestimmte Benutzer explizit zu adressieren. Außerdem verzögern sich Prozesse durch Interaktionen mit dem Benutzer unkontrolliert, was bei einer Fehlerbehandlung berücksichtigt werden muss.

M12 **Verteilte Administration des Prozesses:** Die Prozessbeschreibung muss Konzepte zur verteilten Administration des Prozesses beinhalten. Sie muss berücksichtigen, dass Zustände nicht bei einer bestimmten Instanz global gespeichert und beliebig abgerufen werden können. Insbesondere muss sie festhalten, welche Aktivitäten bereits ausgeführt worden sind und welche Ergebnisse die Ausführung hatte [SGSP04]. Ergebnisse und andere Prozessdaten müssen solange für andere Prozessteilnehmer vorgehalten werden, bis der Prozess beendet wird oder gescheitert ist.

M13 **Mächtigkeit der Prozessbeschreibungssprache:** Aufgrund der unterschiedlichen Beschränkungen der mobilen Systeme muss in Bezug auf die Mächtigkeit der Prozessbeschreibungssprache ausreichend Flexibilität zur Verfügung stehen, um anwendungsspezifische Anforderungen erfüllen zu können. Der Verschiedenheit von mobilen Systemen und möglichen Anwendungen kann prinzipiell nur durch eine angemessen große Anzahl an möglichen Strategien zur Durchführung entgegengetreten werden.

Beispiel:

Anwendung A benötigt möglichst aktuelle Daten, da es sich um eine Echtzeitanwendung handelt. Bei einem lang andauernden Gesamtprozess macht es für diesen Fall keinen Sinn, die benötigten Daten zu Beginn des Prozesses mitzuführen, da sie mit großer Wahrscheinlichkeit bereits veraltet sind, wenn sie ihr Ziel erreichen. Vorteilhafter ist es, die aktuellen Daten möglichst nah zum Zeitpunkt der Verarbeitung zu laden.

Anwendung B ist eine zeitkritische Anwendung. Für eine erfolgreiche Ausführung benötigt sie Daten, die zu einem bestimmten Zeitpunkt vorliegen müssen. Hier macht es Sinn, verfügbare Daten sofort zu laden, damit die Datenverfügbarkeit später bei Verbindungsproblemen keinen Engpass für die Anwendung darstellt.

Anwendung C arbeitet mit besonders großen Datenmengen. Allerdings hängt der tatsächliche Bedarf von der konkreten Umgebungssituation ab, die zur Entwicklungszeit noch nicht ausgemacht werden kann. Für den Fall, dass die Anwendung die Daten nicht benötigt, wären die mobilen Systeme unnötig mit der Last der Datenmengen belastet oder vielleicht sogar überlastet worden. Es ist also sinnvoll, die nur eventuell benötigten Daten erst bei tatsächlichem Bedarf zu laden.

Am Beispiel wird deutlich, dass die Wahl der richtigen Strategie zur Ausführung nur durch die Anwendung oder durch Benutzervorgaben getroffen werden kann. Würde nur ein Default-Mechanismus zur Datenakquisitation zur Verfügung stehen, würden die dargestellten Prozesse mit großer Wahrscheinlichkeit scheitern. Nur ein ausreichendes Maß an Flexibilität kann daher die erfolgreiche Ausführung von Prozessen auf mobilen Systemen sicherstellen. Dabei ist jedoch zu beachten, dass jeder Gewinn an Flexibilität auch wieder ein Aufwand an zusätzlichen Beschreibungen gegenübersteht. Hier muss ein angemessener Kompromiss zwischen der benötigten Mächtigkeit der Prozessbeschreibungssprache und der zur Verfügung stehenden Leistungsfähigkeit der mobilen Systeme getroffen werden.

3.4.4 Gewichteter Anforderungskatalog

Die erarbeiteten Anforderungen an eine Prozessbeschreibungssprache für mobile Systeme werden im Folgenden zu einem Anforderungskatalog zusammengefasst. Dabei soll zusätzlich eine Gewichtung der Ziele vorgenommen werden, die Aufschluss darüber gibt, wie wichtig die einzelnen Aspekte für

den vorgesehenen Einsatzzweck sind. Unterschieden wird zwischen starken Anforderungen, optionalen Anforderungen und nicht realisierbaren Anforderungen.

Starke Anforderungen stellen die minimalen funktionalen Bedingungen dar, ohne die Prozesse auf mobilen Systemen nicht ausgeführt werden können. Diese müssen also zwingend von der zu spezifizierenden Prozessbeschreibungssprache und gegebenenfalls ihrer Ausführungsumgebung erfüllt werden. Können Konzepte, die sich auf einer starken Anforderung begründen, aus wichtigen Gründen nicht in die Prozessbeschreibung integriert werden, so kann das geplante Vorhaben dieser Arbeit nicht unter den erläuterten Bedingungen umgesetzt werden. Starke Anforderungen enthalten zudem die vorgegebenen Rahmenbedingungen aus dem DEMAC-Projekt. Zur Unterscheidung sind diese zusätzlich gekennzeichnet (*).

Optionale Anforderungen sind Anforderungen, die von mobilen Systemen höherer Leistungsfähigkeit realisiert werden können. Sie stellen in diesem Sinne ergänzende Funktionalitäten dar, die die Prozessbeschreibung erweitern und ihre Funktionalität und Benutzbarkeit verbessern. Mit zunehmender Erfüllung optionaler Anforderungen wächst das Potential der Beschreibungssprache und stellt Benutzern und Anwendungsprogrammen mehr Konstrukte und Konzepte zur Verfügung, um die Flexibilität und damit die Ausführbarkeit von Prozessen zu steigern. Hierzu zählt zum Beispiel die Bereitstellung erweiterter Kontrollflussmöglichkeiten.

Nicht realisierbare Anforderungen enthalten alle allgemein geforderten Aspekte einer Prozessbeschreibung, die von einer Prozessbeschreibungssprache für mobile Systeme aufgrund ihrer speziellen Eigenschaften nicht oder nur teilweise erfüllt werden können. Ihre Unrealisierbarkeit ist jedoch nur zweitrangig, da es sich um Nebenbedingungen handelt, die zum Einsatz der Sprache nicht zwingend notwendig sind oder durch andere Konzepte ersetzt werden können.

Tabelle 4 zeigt die gewichteten Anforderungen in Übersichtsform.

Die Daten der Tabelle beschreiben zusammenfassend die für eine Prozessbeschreibungssprache im Bereich der mobilen Systeme relevanten Konstrukte und Mechanismen. Neben den Anforderungen aus dem DEMAC-Projekt sind im wesentlichen Elemente zur Beschreibung von Kontroll- und Datenfluss und zur Spezifikation von Prozessteilnehmern erforderlich. Die entsprechenden Definitionen müssen plattformunabhängig und auf die begrenzte Speicherkapazität und Rechenleistung der mobilen Geräte zugeschnitten sein. Geringe Bandbreiten bei der Übertragung sowie die Gefahr von Verbindungsabbrüchen müssen berücksichtigt werden.

Besonders wichtige optionale Anforderungen sind die Möglichkeiten zur Fehlerbehandlung und die Bereitstellung von Sicherheitsmaßnahmen sowie von Synchronisationsmechanismen und die Fähigkeit zur Benutzerinteraktion.

3.4.4 Gewichteter Anforderungskatalog

	Anforderung	Stark	Optional	Nicht realisierbar
D1	Beschreibung und Ausführung komponierter automatisierter Services	X(*)		
D2	Weitergabe von Prozessen an andere Ausführungseinheiten	X(*)		
D3	Auswahl von Prozessteilnehmern durch benutzerdefinierte nicht-funktionale Kriterien	X(*)		
D4	Kompatibilität zu bestehenden Ansätzen		X(*)	
P1	Kontrollflusskonstrukte	X		
P2	Datenflussbeschreibung	X		
P3	Spezifikation von Prozessteilnehmern	X		
P4	Beschreibung von Fehlerbehandlungsmaßnahmen		X	
P5	Dauerhafte Verfügbarkeit des Prozess-Systems			X
P6	Skalierbarkeit		X	
P7	Erweiterbarkeit		X	
P8	Sicherheit		X	
P9	Unterstützung von Transaktionen		X	
P10	Plattformunabhängigkeit	X		
P11	Zentrales System zur Administration			X
P12	Audit Trail		Begrenzt	
M1	Geringer Speicherverbrauch	X		
M2	Geringer Anspruch an Rechenleistung	X		
M3	Verantwortlicher Umgang mit Energieressourcen		X	
M4	Lokale Datenhaltung		X	
M5	Geringer Kommunikationsaufwand	X		
M6	Behandlung von Verbindungsabbrüchen	X		
M7	Synchronisation		X	
M8	Priorisierung		X	
M9	Ergänzende Sicherheitsmechanismen		X	
M10	Auswahl von variablen Diensten und Prozessteilnehmern erfolgt zur Laufzeit	X (*)		
M11	Benutzerintegration		X	
M12	Verteilte Administration des Prozesses	X		
M13	Mächtigkeit der Prozessbeschreibungssprache		X	

X Forderung

X(*) Forderung aus dem DEMAC-Projekt

Tabelle 4: Anforderungskatalog mit Gewichtungen

4 Untersuchung von vorhandenen Ansätzen zur Prozessbeschreibung

In diesem Kapitel wird untersucht, inwieweit bestehende Ansätze für die Integration von Prozessen in eine Middleware für mobile Systemen geeignet sind. Zunächst wird in einer kurzen Übersicht erläutert, welche Ansätze zur Prozessbeschreibung aktuell existieren und motiviert, welche Auswahl davon in dieser Arbeit näher betrachtet wird. Aufgrund der dargestellten Relevanz der Komposition automatisierter Services werden dann im Zentrum dieses Kapitels aktuelle Sprachen zur Definition von Workflows und zur Beschreibung von automatisierten Kollaborationen analysiert. Zur Betrachtung eines konträren Ansatzes, welcher verstärkt eine benutzerintensive Kontrolle von Ablaufplänen verfolgt, wird im Anschluss das Paradigma des *Prozessmusters* betrachtet. Im letzten Abschnitt werden die gesammelten Erkenntnisse zusammengefasst.

4.1 Übersicht bestehender Beschreibungssprachen für Prozesse

Zur Zeit bestehen viele Ansätze von Prozessbeschreibungssprachen, die die Definition von Kontroll- und Datenfluss sowie die Spezifikation von Zuständen und Prozessteilnehmern unterstützen. Die meisten sich im Einsatz befindlichen Sprachen zur Prozessbeschreibung definieren Workflows und Kooperationen von Wirtschaftsunternehmen. Hinzu kommen *Architekturbeschreibungssprachen (ADLs)* und *Grid Service Composition Languages*. *Architekturbeschreibungssprachen* sind jedoch eher auf einem technischen als auf einem abstrakten Niveau angesiedelt und daher für den beabsichtigten Einsatzzweck nicht ausreichend geeignet. Beschreibungssprachen zur Komposition von *Grid Services* sind noch nicht ausgereift genug, da sie sich noch in einer zu frühen Phase der Entwicklung befinden. Ins-

besondere das Fehlen allgemeingültiger Spezifikationen erschwert die Analyse und den Einsatz dieser Beschreibungssprachen. Die vorliegende Arbeit konzentriert sich daher auf bereits etablierte Prozessbeschreibungsprachen zur Definition von Geschäftsprozessen und allgemeinen Abläufen.

Um verschiedene Einsatzbereiche dieser Prozessbeschreibungssprachen gegeneinander abzugrenzen, ist eine Kategorisierung bestehender Ansätze vorteilhaft. Da es jedoch keine allgemeingültige Definition der verwendeten Konzepte gibt, soll im folgenden versucht werden, einige generelle Eigenschaften der sich aufzeigenden Gruppierungen zu betrachten.

Klassische *Workflow-Beschreibungssprachen* befassen sich überwiegend mit der Zuweisung von Aufgaben an Personen, mit dem Ziel, menschliche Arbeitskräfte möglichst effizient zu koordinieren [LeRo00]. Zur Verwaltung der Prozesse werden daher meistens Computersysteme eingesetzt, die aber eher selten auch in automatisierter Form an den Aktivitäten des Prozesses beteiligt sind.

Das *Business Process Management (BPM)* stellt eine Verknüpfung von Workflow mit Unternehmensfunktionen und der Erbringung von konkreten Leistungen dar. Ein Business-Prozess spezifiziert dazu den Ablauf von menschlichem Arbeitshandeln, wobei Teilaufgaben auch durch den gezielten Einsatz von Softwarekomponenten unterstützt oder automatisiert ausgeführt werden können. Ziel ist es, menschliches Arbeitshandeln und Softwareprogramme möglichst effizient zu koordinieren. Business Prozesse können dabei sowohl unternehmensübergreifende Beziehungen modellieren (*B2B Integration*) oder interne Arbeitsabläufe vergegenständlichen (*Enterprise Application Integration*). Derart öffentliche und private Prozesse haben zwar gemeinsame Wurzeln in Bezug auf Kontrollflusskonstrukte, müssen jedoch auch individuelle Mechanismen zur Unterstützung ihrer Anwendungsdomäne aufweisen, zum Beispiel Sicherheitskonzepte für die unternehmensübergreifende Kommunikation oder die Formulierung von Ausführungsdetails für eine unternehmensinterne Verarbeitung [Rio02].

Beschreibungssprachen zur automatisierten koordinierten Kollaboration zweier oder mehrerer Partner werden unter den Begriffen *Orchestration* oder *Choreography* zusammengefasst. Die *Orchestration* von Prozessteilnehmern konzentriert sich dabei in erster Linie auf die Definition einer logisch und zeitlich geordneten Reihenfolge von Aktivitäten, wobei der Prozess selbst immer von Seiten eines Partners dargestellt und kontrolliert wird. Hingegen steht bei einer *Choreography* die Kommunikation und die Definition von Verhaltensweisen der Parteien im Mittelpunkt. Jede am Prozess beteiligte Partei beschreibt dazu ihre eigene Rolle an der Kollaboration und tauscht mit anderen Parteien Nachrichten und Dokumente zur Interaktion aus [Pel03].

Bei Prozessbeschreibungssprachen, die rein zur Definition von interagierenden *Web Services* entwickelt werden, spricht man von *Web Service Orchestration* bzw. von *Web Service Choreography* (Abb. 22). Bestehende *Web Services* können so durch Komposition zu neuen Web *Services* integriert werden. Sprachen zur Koordination von *Web Services* beinhalten neben den Primitiven zur Strukturierung des Kontrollflusses auch bestimmte Operationen, um die Kommunikation mit *Web Services* zu unterstützen, also insbesondere diese aufzurufen oder den Aufruf eines *Web Services* entgegenzunehmen. Ein Beispiel für eine Sprache zur *Web Service Orchestration* ist *BPEL4WS*, die im nächsten Abschnitt vorgestellt wird. Typische Beispiele für Sprachen, die sich bei der Kollaboration nicht explizit auf *Web Services* beschränken, sind *ebBPSS* von ebXML oder *WSCI* [Pel03].

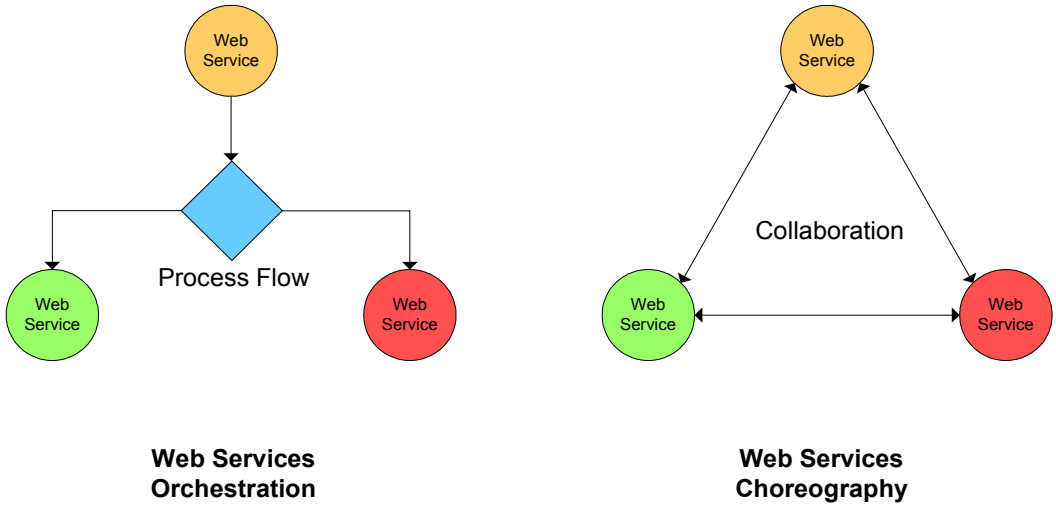


Abb. 22: Orchestration und Choreography (aus [Pel03])

Eine erste Kategorisierung von bestehenden Ansätzen kann in Hinblick auf den Schwerpunkt der manuellen bzw. der automatisierten Ausführung von Prozessen erfolgen. Während *klassische Workflows* fast ausschließlich menschliches Arbeitshandeln durch den Einsatz eines Computersystems koordinieren, nimmt der Grad der Automatisierung im *Business Process Management* zu. Bei der Orchestrierung und Choreographie vollautomatisierter Services werden in der Regel keine Benutzer mehr in die definierten Arbeits- und Kommunikationsabläufe integriert (Abb. 23). Die verwendeten Workflowbeschreibungssprachen weisen dazu je nach Einsatzzweck verschiedene Schwerpunkte auf, zum Beispiel die Definition von Rollenbeziehungen oder Geschäftstransaktionen.

Aufgrund der im vorherigen Kapitel spezifizierten Anforderungen sind geeignete Prozessbeschreibungssprachen für die Definition von Prozessen auf mobilen Systemen vor allem in den Bereichen zu suchen, in denen sich die erläuterten Einsatzgebiete überlappen. Benutzerinteraktionen sind sowohl wegen der inhärenten Eigenschaft mobiler Geräte als Begleiter von Benutzern (vgl. [Wei91]) als auch zur Behandlung von Fehlern und Ausnahmezuständen relevant. Für die Koordination von unabhängigen verteilten Diensten einer serviceorientierten Architektur bieten sich vor allem Beschreibungssprachen für Prozesse an, die für die Komposition vom *Web Services* entwickelt wurden oder diese zumindest integrieren. Das Fehlen einer zentralen Komponente im Szenario kollaborierender mobiler Systeme macht zudem Beschreibungssprachen zur Definition von verteilten Interaktionen interessant. Da XML-basierte Ansätze Plattformunabhängigkeit bieten, soll sich diese Analyse auf XML-basierte Sprachen beschränken.

Im Zentrum der folgenden Betrachtungen stehen daher die Prozessbeschreibungssprachen *XPDL* und *BPEL4WS*. *XPDL* erlaubt sowohl manuelle Workflows als auch die Integration von Business-Anwendungen. *BPEL4WS* ist eine reine Sprache zur Komposition von *Web Services*. Das Schema *ebBPSS* wird erläutert, da es nicht-funktionale Aspekte wie Sicherheitsanforderungen und Vertragseigenschaften in seine Beschreibungen integriert. Als Beschreibungssprache zur koordinierten Kollaboration zwischen verschiedenen Unternehmen soll *WSCI* vorgestellt werden. Ein sehr schlanker, alternativer Ansatz zur Prozessbeschreibung wird durch die Betrachtung von *jPdl* vermittelt.

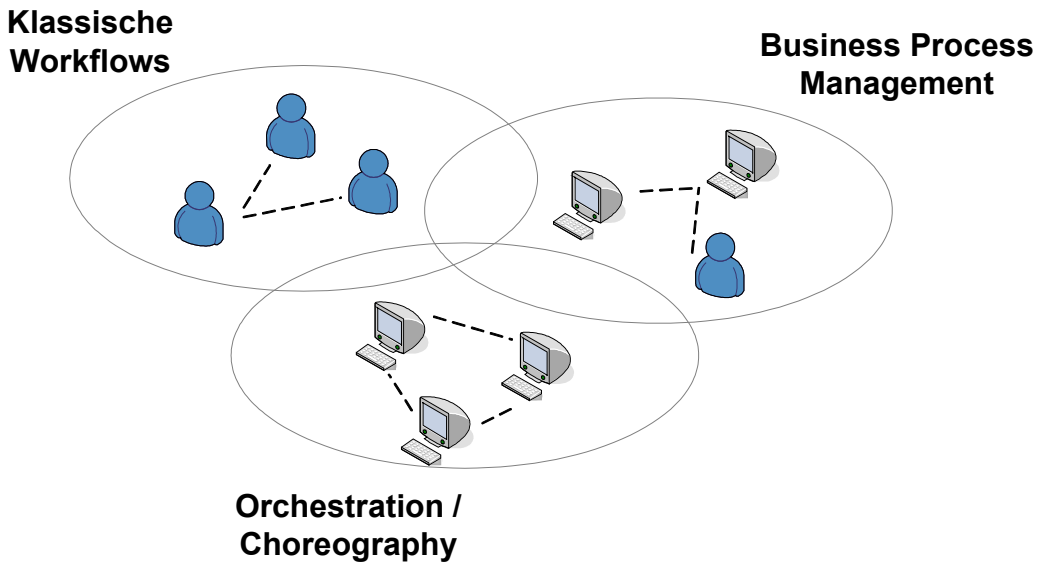


Abb. 23: Einsatzgebiete von Prozessbeschreibungssprachen (nach [jBo05])

Die vorgestellten Sprachen sind dabei nicht als “echte” Programmiersprachen zu verstehen. Vielmehr handelt es sich um XML-Schemata für die Spezifikation von Prozessen, die mit konkreten Werten instantiiert und von einer Ausführungsumgebung interpretiert und ausgeführt werden können.

Standard-Nachrichten-Formate wie zum Beispiel *RosettaNet*, *OBI* oder *cXML* betrachten nur konkrete Anwendungsfälle, zum Beispiel das Supply Chain Management, und sind damit zu eingeschränkt, um ein allgemeines Schema für mobile Systeme bereitzustellen. Außerdem sind sie zu statisch ausgelegt, um auch das dynamische Auffinden von Services zur Laufzeit zu ermöglichen und sollen daher nicht in die Analyse einbezogen werden.

4.2 Analyse ausgewählter Sprachen zur Prozessbeschreibung

Im folgenden werden die Beschreibungssprachen *XPDL*, *BPEL4WS*, *ebBPSS*, *WSCL* und *jPdl* betrachtet und bewertet, ob sie als Ganzes oder in Auszügen für eine Prozessbeschreibungssprache für mobile Systeme geeignet sind. Dazu werden ihre Eigenschaften dargestellt und mit den Anforderungen aus Abschnitt 3.4 verglichen.

Die Bewertung der Sprachen bezieht sich lediglich auf deren native Eigenschaften. Eine prinzipielle Erweiterbarkeit um Konzepte, um die genannten Anforderungen zu erfüllen, ist theoretisch bei allen Sprachen möglich.

4.2.1 XPDL

Die *XML Process Definition Language* (*XPDL*) ist Teil des *Workflow Reference Models* der Workflow Management Coalition (WfMC). Diese bemüht sich seit 1993 um einen Standard für den Austausch von Prozessinformationen, um unternehmensübergreifende Workflows zwischen heterogenen Workflow-Management-Systemen zu ermöglichen.

Das *Workflow Reference Model* spezifiziert fünf Schnittstellen [Hol95], die zusammen ein vollständiges Framework zur Begründung von Interoperabilität zwischen Workflow-Management-Systemen darstellen sollen (Abb. 24). Im Zentrum des Models steht eine Ausführungsumgebung für die Interpretation und Bearbeitung von Prozessen, der sogenannte *Workflow Enactment Service*, der aus mehreren Workflow-Engines bestehen kann.

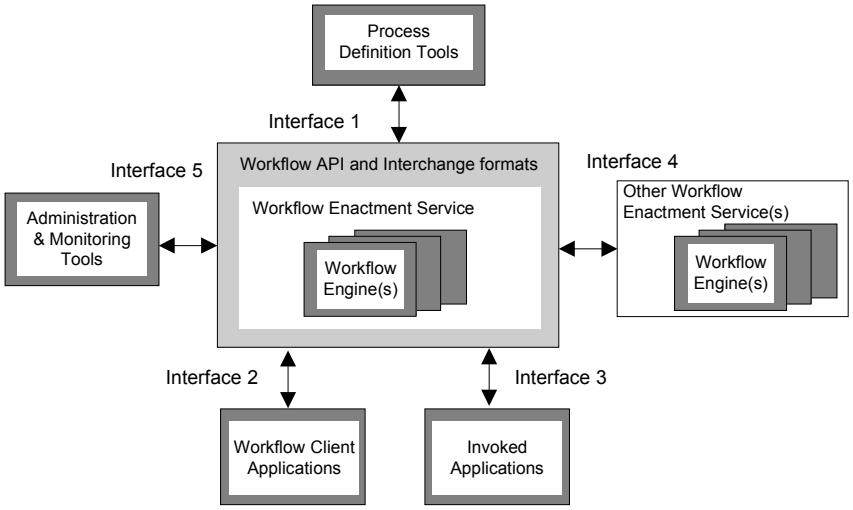


Abb. 24: Workflow Management Coalition Reference Model (aus [Hol95])

Process Definition (Interface 1): Die erste Schnittstelle im Referenzmodell definiert ein allgemeingültiges Format zur Prozessbeschreibung. Proprietäre Prozessmodelle können in eine XML-basierte Sprache übersetzt werden und zum Austausch mit anderen Prozesspartnern verwendet werden. Das Interface enthält hierzu sowohl ein Meta-Modell, welches abgrenzbare elementare Konstrukte eines Prozesses definiert, als auch ein konkretes XML-Schema zum Austausch von Prozessdefinitionen.

Workflow Client Application (Interface 2): Diese Schnittstelle steht zur Verfügung, um Prozesse und Aktivitäten interaktiv durch menschliche Benutzer zu initiieren oder Aufgaben mit manuellem Anteil ausführen zu lassen.

Invoked Applications (Interface 3): Interface 3 erlaubt die automatisierte Ausführung von Aktivitäten durch einen direkten Programmaufruf.

Workflow Interoperability (Interface 4): Mit Hilfe der *Workflow Interoperability* Schnittstelle können Prozesse und Zustände transformiert werden, um Abläufe auf anderen Workflow-Systemen zu

ermöglichen. Zum Beispiel sind Mappings und Synchronisationsmaßnahmen nötig, wenn Subprozesse auf einem heterogenen Workflow-Management-System ausgeführt werden sollen.

Administration and Monitoring Tools (Interface 5): Hinter dieser Schnittstelle verbergen sich Verwaltungs- und Überwachungsfunktionen zum Management von Benutzern, Ressourcen und Prozessen.

XPDL unterstützt die Realisierung des *Process Definition Interface* (Schnittstelle 1) des beschriebenen Referenzmodells. Die Definition einer implementierungsunabhängigen Beschreibungssprache verhilft dem Modell zur Trennung zwischen der Definition von Prozessen und deren Ausführung auf beliebigen Workflow-Engines und stellt somit ein geeignetes Format zum Austausch von Prozessbeschreibungen zwischen verschiedenen Workflow-Management-Systemen dar.

Zur Zeit der Erstellung dieser Arbeit liegt neben der hier verwendeten Version 1.0 der Sprache bereits ein neuer Entwurf (XPDL-Version 1.13) vor, der insbesondere Konzepte der *Business Process Modelling Notation (BPMN)* in die ursprüngliche Ausarbeitung einbezieht, um hierfür ein einheitliches Dateiformat bereitzustellen. Die *BPMN* stellt eine graphische Notation dar, die zur Modellierung und Diskussion von Geschäftsprozessen zwischen Entwicklern und Anwendern dient [WfMC05]. Da sich durch die Integration der beiden Darstellungsformen keine wesentlichen neuen Aspekte für die Prozessbeschreibung als solche eröffnen und zusätzlich die Sprache an Komplexität gewinnt, soll in dieser Arbeit von der bisherigen XPDL-Version 1.0 ausgegangen werden.

Meta-Modell

Das Meta-Modell von WfMC beschreibt die elementaren Konstrukte, die einer Prozessbeschreibung in *XPDL* zugrunde liegen und daher für die Modellierung von Prozessen von Bedeutung sind [WfMC02]. Das Meta-Modell kann in zwei Komponenten gegliedert werden: Es besteht zum einen aus dem *Prozess-Meta-Modell*, welches grundlegende Entitäten zur Konstruktion von Prozessen enthält, und einem Meta-Modell für die Definition von sogenannten *Packages*, durch die bestimmte Entitäten und Prozesse gruppiert und wiederverwendbar gemacht werden können.

Der Aufbau des *Prozess-Meta-Modells* ist in Abbildung 25 illustriert. Die *Workflow Process Definition* stellt dabei einen Container für den Prozess dar und enthält administrative und operative Parameter zu dessen Ausführung, zum Beispiel Erzeugungsdatum oder Zeitbeschränkungen.

Eine Prozessdefinition kann eine Menge von Aktivitäten enthalten, die durch Transitionen logisch miteinander verbunden sind. Diese *Workflow Process Activities* sind zum einen elementare, in sich geschlossene Arbeitsschritte (*Atomic Activity*), können aber auch wiederum mehrere Aktivitäten als einen Block enthalten (*Block Activity*) und so als zusammengehörige Menge von Aktivitäten ausgeführt werden (*Activity Set*). Alternativ kann eine *Activity* einen ganzen Subprozess (*Subprocess Definition*) darstellen, der anstelle dieser einzelnen Aktivität ausgeführt wird.

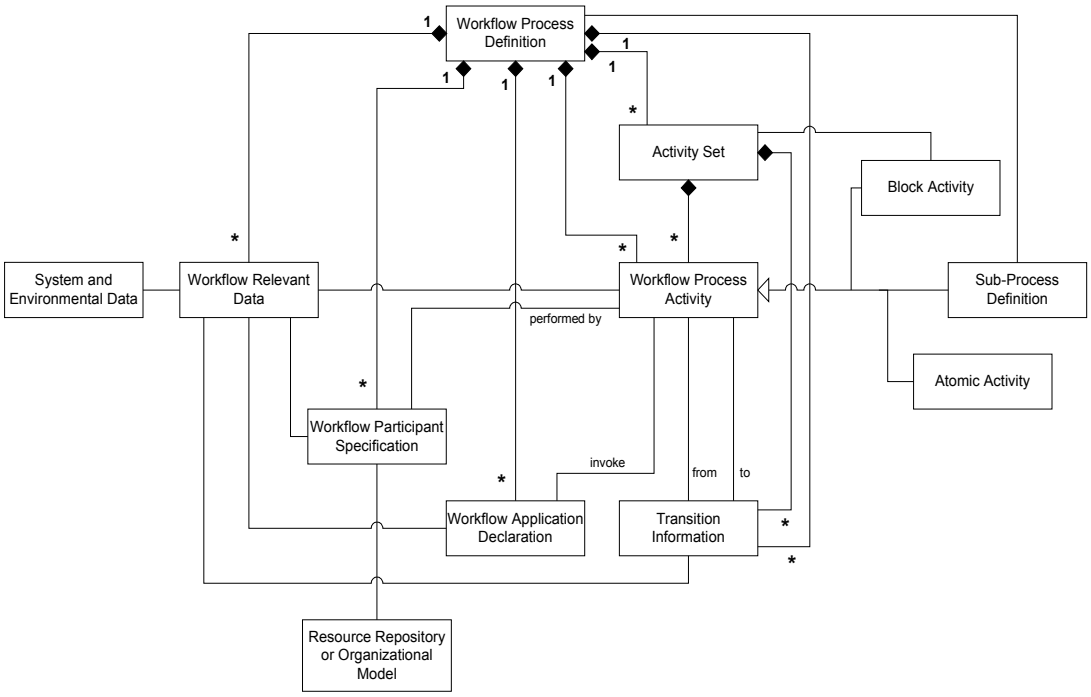


Abb. 25: Workflow Process Definition Meta Model (aus [WfMC02])

Die *Workflow Participant Specification* gibt an, welche Ressourcen an der Ausführung von Aktivitäten beteiligt sind. Dabei müssen keine konkreten Personen oder Geräte benannt werden, sondern Prozessbeteiligte können auch indirekt über ihre Fähigkeiten oder Verantwortlichkeiten identifiziert werden. Informationen über Personen, Softwarekomponenten oder Maschinen, die am Workflow teilnehmen können, können dazu in einem *Resource Repository* oder in einem *Organizational Model* gehalten werden. Während sich die *Workflow Participant Specification* hauptsächlich an menschliche Benutzer richtet, werden in der *Workflow Application Declaration* Softwarekomponenten definiert, die an der Ausführung des Prozesses beteiligt sein können.

Die *Workflow Relevant Data* stellt die Menge der während der Prozessausführung benötigten Daten dar. Es handelt sich hierbei um Daten oder Dokumente, die zur Verarbeitung einzelner Aktivitäten benötigt, von diesen als Ergebnis erzeugt oder zur Evaluation einer Transitionsbedingung verwendet werden. Für die *Workflow Relevant Data* können verschiedene Datentypen zugrunde liegen: *XPDL* unterstützt sowohl elementare als auch komplexe Datentypen. Die *System and Environmental Data* hingegen sind interne Daten zum Management der Ausführungsumgebung und haben nur bedingt Einfluss auf den Ablauf eines Prozesses.

Zur semantischen Trennung wiederverwendbarer Beschreibungskomplexe wird das Meta-Modell der *Packages* eingeführt (Abb. 26). Prozessteilnehmer, Softwareprogramme und Workflow-Daten können für den Namensraum eines *Packages* global definiert werden, um eine Referenzierung von mehreren Prozessdefinitionen aus diesem *Package* zu ermöglichen. Das *Package* wirkt dabei wie ein Container, der zusammengehörige Daten und Prozesse aufnimmt. Alle potentiell beteiligten Ressourcen können vereinfacht bearbeitet und zugegriffen werden, indem sie in einem *Resource Repository* persistent gemacht werden.

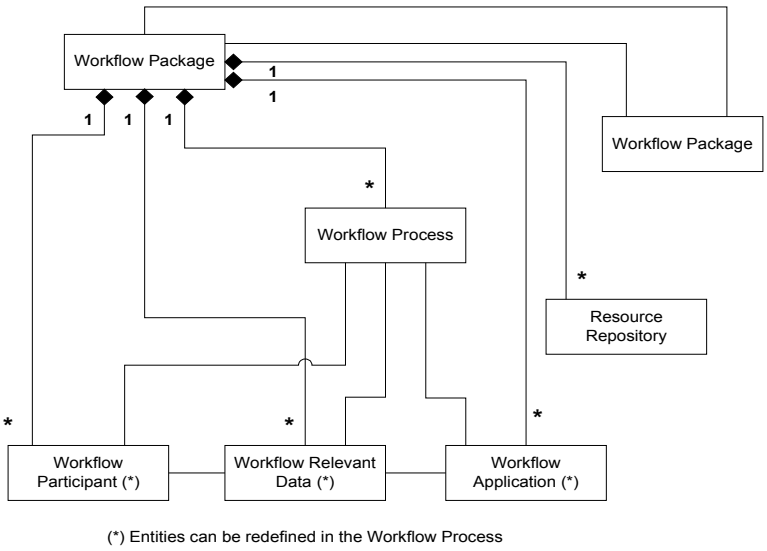


Abb. 26: Package Definition Meta Model (aus [WfMC02])

Aufbau

XPDL ist eine graph-strukturierte Beschreibungssprache, deren sprachlicher Aufbau sich aus dem Meta-Modell ableiten lässt. Globaler Container für alle Elemente ist das *Package*. Dieses kann eine beliebige Anzahl von Prozessen enthalten und ergänzende Informationen speichern. So können hier unter anderem in mehreren Prozessen verwendete Prozessteilnehmer, Applikationen und Datentypen im Vorwege deklariert werden. Das Package enthält ebenfalls Hinweise auf die zugrunde liegende *XPDL*-Spezifikation und die entsprechenden XML-Schemata (Codebeispiel 1, Beispieldaten aus [WfMC02]).

```
<Package xmlns="http://www.wfmc.org/2002/XPDL1.0"
         xmlns:xpdl="http://www.wfmc.org/2002/XPDL1.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xyz="http://www.xyzeorder.com/workflow"
         xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
         http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd"
         Id="0"   Name="sample workflow process">

  <PackageHeader> ... </PackageHeader>
  <TypeDeclarations> ... </TypeDeclarations>
  <Participants> ... </Participants>
  <Applications> ... </Applications>
  <DataFields> ... </DataFields>
  <WorkflowProcesses> ... </WorkflowProcesses>

</Package>
```

Codebeispiel 1: XPDL Workflow Package

Ein Package kann beliebig viele Prozesse beschreiben. Im einfachsten denkbaren Fall enthält die Prozess-Definition genau einen Prozess (Codebeispiel 2). Obligatorisch ist für einen *XPDL*-Prozess nur das Attribut *Id*.

```
<WorkflowProcesses>
   <WorkflowProcess Id="1" Name="EOrder" AccessLevel="PUBLIC">
         ...
   </WorkflowProcess>
</WorkflowProcesses>
```

Codebeispiel 2: XPDL Workflow Processes

Innerhalb eines einzelnen *XPDL*-Prozesses können wiederum bestimmte Elemente für eine lokale Verwendung innerhalb dieses Gültigkeitsbereiches definiert werden. Daneben setzen sich die Prozesse aus verschiedenen Aktivitäten zusammen, deren Ausführungsreihenfolge durch die Angabe ihrer Transitionen spezifiziert wird. Codebeispiel 3 zeigt einen exemplarischen Prozess mit den genannten Elementen.

```
<WorkflowProcess Id="1" Name="EOrder" AccessLevel="PUBLIC">

   <ProcessHeader> ... </ProcessHeader>
   <FormalParameters> ... </Formal Parameters>
   <DataFields> ... </DataFields>
   <Participants> ... </Participants>
   <Applications> ... </Applications>
   <ActivitySets> ... </ActivitySets>

   <Activities>
      <Activity Id="1" Name="Check Data"> ... </Activity>
      <Activity Id="2" Name="E-Mail Confirmation"> ... </Activity>
      <Activity Id="3" Name="Check Credit"> ... </Activity>
                  ...
   </Activities>

   <Transitions>
      <Transition Id="1" From="1" To="2" />
      <Transition Id="2" From="1" To="3" />
      <Transition Id="3" From="2" To="3" />
      ...
   </Transitions>

</WorkflowProcess>
```

Codebeispiel 3: XPDL Workflow Process Definition

Aktivitäten können durch die Referenzierung global definierter Beschreibungen auf Prozessteilnehmer, Softwarekomponenten oder Daten Bezug nehmen. Optional können weitere Bedingungsattribute zur Steuerung der Ausführung dieser Aktivität angegeben werden. Benutzerdefinierte Erwei-

terungen von Aktivitätsbeschreibungen dienen der Anpassung von *XPDL* auf unternehmensspezifische Bedürfnisse.

Im folgenden Codebeispiel (Codebeispiel 4 [WfMC02]) wird eine zuvor global definierte Applikation „checkData“ mit der Ausführung der anliegenden Aktivität beauftragt. Entsprechende Parameter in Form von *Workflow Relevant Data* werden dazu beim Aufruf übergeben. Weiterhin wird mit Hilfe einer *Transition Restriction* festgelegt, dass nach Bearbeitung der Aktivität nur eine der beiden ausgehenden Transitionen verfolgt werden soll (XOR-Bedingung). Zusätzlich liegen benutzerdefinierte Attribute in Form von Koordinaten vor.

```
<Activity Id="1" Name="Check Data">

  <Implementation>
    <Tool Id="checkData" Type="APPLICATION">
      <ActualParameters>
        <ActualParameter>orderInfo</ActualParameter>
        <ActualParameter>status</ActualParameter>
      </ActualParameters>
    </Tool>
  </Implementation>

  <TransitionRestrictions>
    <TransitionRestriction>
      <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="2" />
            <TransitionRef Id="3" />
          </TransitionRefs>
      </Split>
    </TransitionRestriction>
  </TransitionRestrictions>

  <ExtendedAttributes>
    <ExtendedAttribute Name="Coordinates">
      <xyz:Coordinates xpos="183" ypos="389" />
    </ExtendedAttribute>
  </ExtendedAttributes>

</Activity>
```

Codebeispiel 4: XPDL Workflow Activity

Sprachliche Konstrukte

Tabelle 5 zeigt eine Übersicht der im Meta-Modell beschriebenen XPDL-Sprachelemente mit ihren definierten Attributen.

	Package	Workflow Process	Activity	Transition	Application	Data Field (Workflow Relevant Data)	Participant
1	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes	- ID - Name - Description - Extended Attributes
2	- XPDL Version - Source Vendor ID - Creation Date - Version - Author - Codepage - Country Key - Publication Status - Conformance Class - Priority Unit	- Creation Date - Version - Author - Codepage - Country Key - Publication Status - Priority - Limit - Valid From Date - Valid To Date	-Automation Mode - Split - Join - Priority - Limit - Start Mode - Finish Mode - Deadline			- Data Type	- Participant Type
3	- Responsible - External Package	- Parameters - Responsible	- Performer - Tool - Subflow - Activity Set - Actual Parameter	- Condition - From - To	- Parameters	- Initial Value	
4	- Documen-tation - Icon	- Documen-tation - Icon	- Documen-tation - Icon	- Documen-tation - Icon	- Documen-tation - Icon	- Documen-tation - Icon	- Documen-tation - Icon
5	- Cost Unit	- Duration Unit - Duration - Waiting Time - Working Time	- Cost - Duration - Waiting Time - Working Time				

Tabelle 5: Übersicht der XPDL-Sprachelemente (aus [WfMC02])

Die erste Gruppierung von Parametern zeigt hierbei Eigenschaften, die allen Entitäten gemeinsam sind. So haben alle Elemente einen eindeutigen Identifikator (*ID*), einen Namen und optional eine kurze Beschreibung. Zudem besteht zu jedem XPDL-Element die Möglichkeit, weitere benutzerspezifische Attribute zu definieren. Diese sogenannten *Extended Attributes* sind durch XPDL nur durch die Angabe ihres Namens und eines Wertes definiert und können beliebige Inhalte aufweisen. Komplexe Konstrukte können zudem durch erweiterte XML-Schemata angegeben werden. Da mit der Definition

von eigenen Elementen der Gültigkeitsbereich des Standards verlassen wird, müssen bei deren Verwendung aber gegenseitige Absprachen zwischen den Betreibern der beteiligten Workflow-Engines getroffen werden [WfMC02].

In der zweiten Zeile der Tabelle sind entitätsspezifische Attribute aufgeführt. Darunter sind zum einen verwaltungsbezogene Informationen enthalten, wie das Erzeugungsdatum, die Version, der Autor oder das Länderkennzeichen, zum anderen jedoch auch Eigenschaften, die während der operativen Ausführung von Prozessen eine Rolle spielen. So können für Prozesse und Aktivitäten Prioritäten und Zeitbeschränkungen angegeben und explizite Kontrollflussstrukturen spezifiziert werden. Abbildung 27 zeigt die Konstrukte, mit denen eine bedingte Ausführungsreihenfolge festgelegt werden kann.

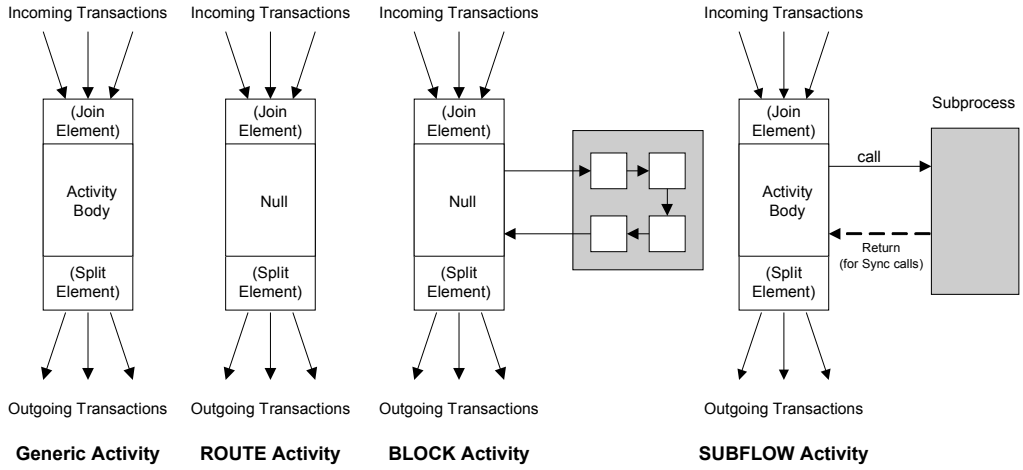


Abb. 27: Aktivitäten und Transitionsbedingungen (nach [WfMC02])

In der Abbildung können wieder die im Meta-Modell angegebenen Elemente identifiziert werden: Atomare Aktivitäten, *Block*-Aktivitäten und *Subflows* können durch *Join*- und *Split*-Elemente mehrere ein- und ausgehende Transitionen verarbeiten. *XPDL* unterstützt hierfür den *AND-Join/-Split* sowie den *XOR-Join/-Split*. Für komplexere Bedingungskonstrukte dienen die sogenannten *Route-Activities*. Diese haben keinen eigentlichen Inhalt und wirken als Dummy-Aktivitäten, um auch komplexe bedingte Verzweigungen darstellen zu können.

Ebenfalls entitätsspezifische Attribute sind die Parameter *Start Mode* und *Finish Mode,* die zusätzliche Angaben darüber machen, unter welchen Umständen die Aktivität gestartet wird bzw. wie sich das Workflow-Management-System verhält, nachdem die Aufgabe bearbeitet worden ist. Um das gewünschte Verhalten festzulegen, kann durch die Spezifikation des *Automation Modes* beschrieben werden, ob die Kontrolle manuell erfolgen oder ob eine automatische Steuerung durch das System vorgenommen werden soll [WfMC02].

Mit Hilfe der *Conformance Class Declaration* kann angegeben werden, inwieweit die Modellierung von Prozessen strukturell beschränkt werden soll, um die Kompatibilität zu anderen Systemen zu erhöhen. Die Optionen hierfür sind:

- FULL-BLOCKED: Es gelten Beschränkungen für die Einbettung von *Split/Join* und Iterationen.

- LOOP-BLOCKED: Es gelten Beschränkungen für die Einbettung von Iterationen.
- NON-BLOCKED: Es gelten keine Einschränkungen (default).

Die dritte Gruppe von Eigenschaften in Tabelle 4 gibt an, welche Referenzen zu anderen Elementen bestehen und mit welchen Sprachfunktionen darauf zugegriffen werden kann. Das Attribut *Responsibles* beschreibt, welcher Prozessteilnehmer verantwortlich ist und die Ausführung beaufsichtigt. *Parameters* stellen die benötigten Parameter zum Aufruf von Anwendungsprogrammen dar oder bilden den Rückgabewert von entsprechenden Funktionen.

Jede Transition, die Aktivitäten zueinander in Beziehung setzt, hat drei elementare Parameter: Die *From-Activity* beschreibt die Ausgangsaktivität, die *To-Activity* spezifiziert die Zielaktivität und das Bedingungsargument *Condition* gibt an, unter welchen Umständen ein Zustandsübergang erfolgt [WfMC02].

Definitionen von externen Entitäten können zusätzlich durch das Attribut *External Reference* angegeben werden. Dieses kann auf Datentypen, Prozessteilnehmer und Anwendungsprogramme angewendet werden und muss eine URI, den Namensraum sowie die ID des Elements im externen Dokument spezifizieren. Über eine *External Reference* können zum Beispiel *Web Services* in einen Prozess eingebunden werden (Codebeispiel 5).

```
<Application Id="placeOrder">
   <ExternalReference
         location="http://abc.com/PO/services/poService.wsdl"
         xref="PlaceOrder" namespace= "http://abc.com/services/
            poService.wsdl/definitions/portType"/>
</Application>
```

Codebeispiel 5: XPDL Einbindung von Web Services

Die in der vierten Gruppierung der Tabelle aufgeführten Attribute für *Documentation* und *Icon* stehen dem Präsentationsmechanismus der Workflow-Engine zur Verfügung.

Die letzte Zeile enthält erforderliche Attribute für Simulation, Analyse und Optimierung von Prozessen und dient in erster Linie dem *Business Process Reengineering (BPR)*. Da es jedoch Aufgabe dieser Arbeit ist, eine möglichst elementare Prozessbeschreibungssprache zu identifizieren, soll hier zunächst davon abgesehen werden, Aspekte der Prozessoptimierung und Simulation zu vertiefen.

Bewertung

Der Ansatz der WfMC, zum Austausch von Workflowsprachen eine standardisierte Beschreibungssprache bereitzustellen, die den kleinsten gemeinsamen Nenner an möglichen Prozesskonstrukten vereint, ist für eine Übernahme in den Anwendungsbereich mobiler Systeme interessant. Mit Ausnahme

der Simulations- und Analyseattribute ist *XPDL* auf sehr grundlegende Elemente beschränkt und kann ohne großen Overhead auskommen. Zudem ist die Sprache nicht ausschließlich auf einen wirtschaftlichen Einsatz fixiert, was auch allgemeine Prozessbeschreibungen abseits von Geschäftsvorgängen ermöglicht.

Die Einbindung und Komposition von automatisierten Diensten wie zum Beispiel von *Web Services* ist über die beschriebenen externen Referenzen möglich. Manuelle Tätigkeiten werden unterstützt und können von automatisierten Aufgaben unterschieden werden. Durch den in *XPDL* integrierten Ansatz der Erweiterbarkeit sind außerdem beliebige zusätzliche Konstrukte definierbar, die einer individuellen Benutzung oder der allgemeinen Skalierbarkeit zuträglich sind.

Die Wiederverwendbarkeit von Prozessdefinitionen ist ein sehr effizienter Ansatz, um Informationen über Prozessteilnehmer, Programme und Abläufe optimal zu verwalten. Der Aufwand im Falle einer Änderung an den entsprechenden Beschreibungen und die Gefahr, dass es zu Inkonsistenzen kommt, wird dadurch deutlich vermindert. Zudem fördert eine Beschreibung außerhalb des eigentlichen Prozesses die dynamische Einbindung von Prozessteilnehmern zur Laufzeit. Dennoch sind diese Vorteile eng mit den Eigenschaften eines zentralen Verwaltungssystems verknüpft. In einem verteilten System mit vielen unterschiedlichen mobilen Komponenten ist es hingegen relevant, dass jeder Teilnehmer die benötigten Daten selbst für die Ausführung der ihm zugewiesenen Aufgaben zur Verfügung hat. Im Falle eines Verbindungsabbruchs soll das mobile System möglichst in der Lage sein, autark seine Arbeit fortzusetzen, ohne weitere Steuerinformationen zum auszuführenden Prozess nachfragen zu müssen. Das Versenden und Bereithalten ganzer *Packages* hingegen belastet die geringe Speicherplatzkapazität und die leistungsarmen Übertragungsmöglichkeiten der mobilen Geräte.

In einer Analyse der Ausdrucksmächtigkeit [Aal03], in der das potentielle Kontrollflussverhalten von XPDL untersucht wurde, wurde bemängelt, dass viele gängige Kontrollstrukturen nicht explizit unterstützt werden und die Semantik an vielen Stellen unklar sei. So werden zum Beispiel nur zwei Typen von *Joins* explizit von *XPDL* definiert: *AND* und *XOR*. Was passieren soll, wenn im Fall von *XOR* mehr als eine eingehende Transition zu “true” evaluiert (also ein nicht ausschließendes *OR*), wird nicht spezifiziert. Eine Kompensation dieser fehlenden Ausdrucksmächtigkeit ist zwar durch verschiedene Workarounds mit Hilfe der erwähnten *Route-Activities* möglich, jedoch haben diese behelfsmäßigen Lösungen den Nachteil, dass die Prozessbeschreibungen dadurch sehr komplex, lang und unübersichtlich werden. Für die Evaluierung dieser zusätzlichen verschachtelten Bedingungen fällt zusätzlicher Aufwand an Beschreibungen und an Speicherbedarf sowie eine zusätzliche Rechenlast an. Andererseits stellt sich natürlich die Frage, inwieweit wirklich komplexe Kontrollstrukturen im Umfeld mobiler Systeme überhaupt Anwendung finden.

Auffällig ist weiterhin, dass es keine vordefinierten Konstrukte gibt, um zusammengehörige Aktivitäten als Transaktion zu kennzeichnen. Zwar wird davon ausgegangen, dass eine einzelne Aktivität atomaren Charakter hat [WfMC02], doch wie zwischen mehreren Aktivitäten zum Beispiel innerhalb eines *Activity Sets* transaktionales Verhalten implementiert werden kann, bleibt durch die Spezifikation offen. Transaktionen können zwar alternativ durch eigene Formulierungen beschrieben werden, zum Beispiel durch eine Konstellation von *AND*-Beziehungen und Transitionsbedingungen. Dies resultiert jedoch zumeist in langen und komplexen Umschreibungen, die die knappen Ressourcen mobiler Systeme unnötig in Anspruch nehmen. Besser wäre in diesem Umfeld eine einfache Kennzeichnung der zur Transaktion gehörigen Aktivitäten, um die Prozessbeschreibung kurz und übersichtlich zu halten.

Dubray [Dub02] kritisiert außerdem, dass keine expliziten Fehlerbehandlungsmaßnahmen durch *XPDL* definiert werden. Dem Modellierer steht es jedoch frei, diese wie im Fall der Transaktionen durch die Verknüpfung von Aktivitäten mit bedingten Alternativen durch eigene Konstrukte zu ersetzen. Die dort genannten Kritikpunkte gelten hier entsprechend.

4.2.2 BPEL4WS

BPEL4WS (Business Process Execution Language for Web Services) ist eine XML-basierte Beschreibungssprache zur Spezifikation von Geschäftsprozessen. Sie wurde 2002 von dem Firmenkonsortium aus Bea, IBM, Microsoft, SAP und Siebel entwickelt und ist das Produkt des Zusammenwachsens zwischen Microsofts Beschreibungssprache *XLANG* und IBMs *Web Services Flow Language (WSFL)*.

Bei *WSFL* handelt es sich um eine Sprache zur Komposition von *Web Services*. Diese ist kompatibel zu SOAP, UDDI und WSDL und kann sowohl ausführbare Geschäftsprozesse beschreiben als auch die Zusammenarbeit zwischen verschiedenen Unternehmen in einer Definition festhalten. *WSFL* ist eine graph-strukturierte Beschreibungssprache (vgl. [Ley01]).

XLANG ist Teil der *Enterprise-Application-Integration-Plattform BizTalk* und dient der Modellierung von Geschäftsvorgängen. Der Kontrollfluss zwischen den Aktivitäten eines Prozesses wird bei *XLANG* durch eine Blockstruktur realisiert, in der Aktivitäten und Kontrollflusskonstrukte ineinander verschachtelt werden (vgl. [Tha01]).

BPEL4WS vereint und erweitert die Eigenschaften ihrer beiden Ursprungssprachen. Wie der Name schon impliziert, werden in erster Linie Kontrollflusskonstrukte bereitstellt, um die geordnete Ausführung von *Web Services* zu koordinieren. Bei der Modellierung von Prozessen werden zwei verschiedene Ausprägungen unterschieden:

- Modellierung ausführbarer Geschäftsprozesse: Tatsächlich ausführbare Geschäftsprozesse stellen unternehmensinterne Vorgänge dar. Sie enthalten eine konkrete Implementierung, die nach außen hin sichtbar und veränderbar ist [ACD+03]. Der Prozess wird dabei immer aus der Perspektive einer der involvierten Parteien koordiniert.
- Modellierung abstrakter Prozesse: Hierbei handelt es sich um die Definition sogenannter Geschäftsprotokolle, die für eine Interaktion mit Kunden oder anderen Unternehmen zur Verfügung stehen. Die Koordination ist daher von gemeinschaftlicher Natur, in der jede involvierte Partei beschreibt, welche Rolle sie innerhalb des Prozesses spielt. Abstrakte Prozesse bilden lediglich Schnittstellen zu konkreten ausführbaren Geschäftsprozessen und verbergen so ihr internes Verhalten. Dies ist zum einen aus Vertraulichkeitsaspekten gegenüber unbekannten Prozesspartnern sinnvoll, zum anderen kann die dahinter stehende Implementierung beliebig ausgetauscht oder verändert werden, ohne dass externe Prozessteilnehmer sich in komplexe Prozessabläufe einarbeiten und anpassen müssen [ACD+03].

BPEL4WS ist durch die Marktmacht von Microsoft und IBM aktuell eine der populärsten Workflow-beschreibungssprachen in Hinblick auf die Integration von *Web Services* [Dub04]. Es ist anzunehmen, dass die Spezifikation zukünftig unter dem Namen *WS-BPEL* von OASIS als Standard bestätigt werden wird. Da sich die Entwicklung von *WS-BPEL* jedoch augenscheinlich noch stärker in den B2B-Bereich verlagern wird [Dub02], soll jedoch für diese Arbeit von der *BPEL4WS*-Spezifikation 1.1 ausgegangen werden (vgl. [ACD+03]).

Beziehung zu WSDL

Die Beschreibung und Einbindung von externen Ressourcen und Prozessteilnehmern erfolgt bei *BPEL4WS* über die Definition von WSDL-Schnittstellen. Dabei haben sowohl die einzelnen innerhalb eines Prozesses aufgerufenen *Web Services* als auch der gesamte *BPEL4WS*-Prozess eine eigenständige WSDL-Definition. Der BPEL4WS-Prozess wird dadurch als Komposition von Diensten selbst zum aufrufbaren *Web Service* und kann auch innerhalb eines anderen *BPEL4WS*-Prozesses als Subprozess eingebunden werden [LRT03] .

Ein WSDL-Dokument beschreibt die Schnittstelle eines *Web Services*. Es basiert auf XML und ist aus Definitionen zusammengesetzt, die spezifizieren, wo ein bestimmter *Web Service* zur Verfügung steht und welche Operationen er anbietet [CCMW01]. Dabei sind die abstrakten Definitionen von Endpunkten und Nachrichten von ihrer konkreten Implementation und somit vom Deployment getrennt, um eine Wiederverwendbarkeit abstrakter Beschreibungen zu erlauben. Die wesentlichen Elemente von WSDL sind im folgenden kurz skizziert [CCMW01]:

Port Type: *Port Types* enthalten eine abstrakte Beschreibung der von einem *Web Service* angebotenen Operationen. Ein *Port Type* legt jedoch noch nicht fest mit welchem Protokoll (zum Beispiel SOAP) die Übertragung der Nachrichten erfolgt oder wie die Parameter des Protokolls zu wählen sind. Diese Festlegung kann für jeden *Port Type* mit einem *Binding* geschehen.

Message: *Messages* stellen eine abstrakte Definition der übertragenden Nachrichten dar. Nachrichten können aus verschiedenen logischen *Parts* bestehen.

Part: Ein *Part* stellt einen Parameter als Bestandteil einer WSDL-Nachricht dar. Jedes *Part*-Element besitzt einen Namen und einen Datentyp.

Operation: Eine abstrakte Beschreibung der vom *Web Service* unterstützten Aktionen wird als *Operation* bezeichnet. Je *Operation* werden mehrere *Messages* zu Eingabe-, Ausgabe- und Fehlernachrichten zusammengesetzt und so Operationssignaturen beschrieben.

Types: Im *Types*-Element werden Datentypen für die Beschreibung definiert.

Binding: Ein *Binding* stellt die Definition konkreter Protokolle und Datenformate für einen bestimmten *Port Type* dar.

Port: Ein *Port* ist ein einzelner Kommunikationsendpunkt und wird durch die Kombination eines *Bindings* mit einer konkreten Netzwerkadresse beschrieben.

Service: Ein *Service* bezeichnet eine Menge zusammengehöriger Kommunikationsendpunkte.

WSDL unterstützt unidirektionale und bidirektionale *Port Types*. Für den Nachrichtenaustausch stehen in WSDL also vier verschiedene Muster zur Verfügung: Einfache eingehende Nachrichten (*one-way*), einfache ausgehende Nachrichten (*notification*), eingehende Nachrichten gefolgt von einer ausgehenden Antwortnachricht (*request-response*) und ausgehende Nachrichten gefolgt von einer eingehenden Antwortnachricht (*solicit-response*). Fehlermeldungen können daher nur durch bidirektionale *Port Types* ausgedrückt werden.

Neben der WSDL-Beschreibung der *Web Services* benötigt BPEL4WS noch die dazugehörigen XML-Schema-Definitionen. Abbildung 28 zeigt deren grundsätzliche Abhängigkeiten, die im nächsten Abschnitt genauer erläutert werden.

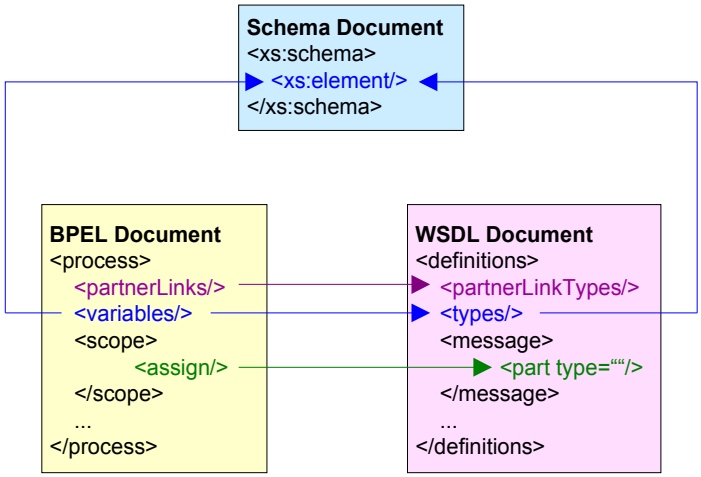


Abb. 28: Abhängigkeiten zwischen BPEL, WSDL und XML-Schema (aus [Fan05])

BPEL4WS erweitert WSDL um die Möglichkeit, einen abstrakten Ablaufplan zur koordinierten Nutzung und Ausführung von *Web Services* zu definieren. Durch die Angabe der abstrakten WSDL-Elemente in der *BPEL4WS*-Beschreibung können Prozesse zu ihrer Designzeit abstrakt spezifiziert werden und erst zur Laufzeit mit den tatsächlichen physikalischen Bindungen und Adressinformationen in Beziehung gebracht werden [Fan05].

Meta-Modell

Das Meta-Modell von *BPEL4WS* [Dub04] beschreibt die elementaren Konstrukte, die durch diese Sprache zur Modellierung von Prozessen verwendet werden können (Abb. 29).

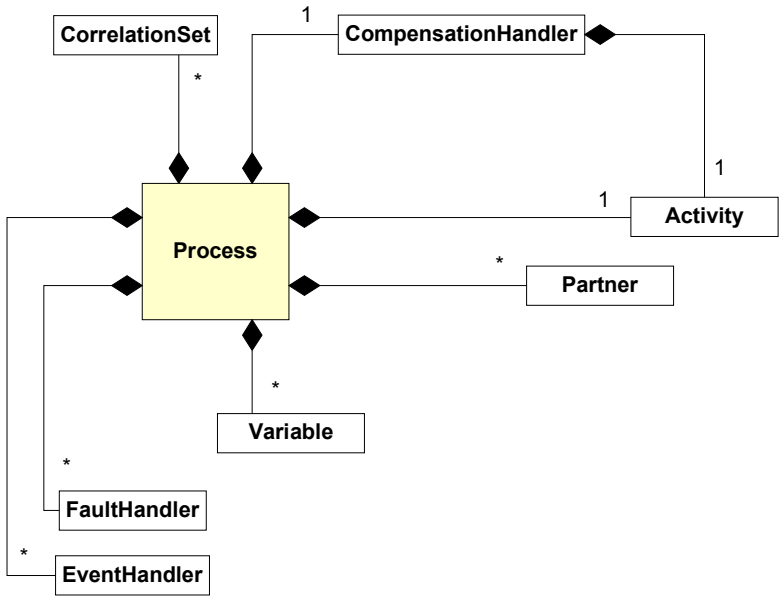


Abb. 29: BPEL4WS Meta-Modell (vgl. [Dub04] für BPEL4WS 1.0)

Die Blockstrukturierung stellt bei *BPEL4WS* ein zentrales Konzept dar. So besteht ein Prozess auf oberster Strukturebene aus genau einer Aktivität. Diese kann in sich weiter verschachtelt sein und so einen Behälter für den eigentlichen Kontrollfluss innerhalb des Prozesses darstellen. In einer gedachten Baumstruktur stellen die komplexen Aktivitäten die Knoten des Baumes dar, die über dessen weitere Verzweigung bestimmen. Die atomaren Aktivitäten können hingegen als Blätter des Baumes verstanden werden. Sie enthalten die eigentliche Implementation des Vorganges. Zur besseren Unterscheidung wird eine atomare Aktivität als *Basic Activity* und eine komplexe Aktivität als *Structured Activity* bezeichnet [ACD+03].

Neben den Aktivitäten, die im Normalfall ausgeführt werden, kann jedem Prozess ein optionaler *Compensation Handler* zugeordnet werden. Dieser kann aktiviert werden, wenn an späterer Stelle des Prozesses ein Fehler auftritt, der die Kompensation der bisher abgearbeiteten Aktivitäten notwendig macht. Er enthält eine anwendungsspezifische, gegebenenfalls weiter strukturierte Aktivität, die die Wirkung des Prozesses semantisch rückgängig macht. Zudem ist es auch möglich, *Compensation Handler* auf der Ebene der Aktivitäten zu definieren, so dass auch einzelne Aktionen gezielt kompensiert werden können.

Die *Web Services,* mit denen der Prozess interagiert, werden bei BPEL4WS als *Partner* bezeichnet. In der Definition eines *Partners* werden neben der Angabe konkreter Schnittstellen auch die Beziehungen zwischen den am Prozess teilnehmenden Parteien und deren Rollen spezifiziert. So kann zum Beispiel detailliert festgelegt werden, wer der Anbieter und wer der Nutzer eines Dienstes ist und wie die Kommunikation zwischen ihnen zu erfolgen hat. Im Vordergrund der Partnerdefinition steht also eine Verzahnung von geschäftsübergreifenden Prozessen: Man geht in der Regel davon aus, dass auch der angesprochene Geschäftspartner wiederum einen *BPEL4WS*-Prozess besitzt, um die durch den entfernten Prozess angestoßenen Aktivitäten automatisiert ausführen zu können. Dazu können drei *Partner*-Konstellationen festgelegt werden [ACD+03]:

• Ein *Partner* wird vom Prozess aufgerufen (*invoked partner*)

• Ein *Partner* ruft den Prozess auf (*client partner*)

• Ein *Partner* ist ein Service, welcher vom Prozess aufgerufen wird und welcher wiederum den Prozess aufruft oder umgekehrt.

Partner stellen also die Prozessteilnehmer eines *BPEL4WS*-Prozesses dar. Es ist jedoch zu beachten, dass Aktivitäten eines *BPEL4WS*-Prozesses ausschließlich durch *Web Services* erfüllt werden können. Eine manuelle Ausführung oder der Aufruf von alternativen Softwareprogrammen, die keine WSDL-Schnittstelle besitzen, wird nicht unterstützt.

Prozesse können auch bei der exklusiven Nutzung von Web Services lang andauernd sein und müssen daher Eigenschaften zustandsbehafteter Sitzungen aufweisen. Dieses wird bei BPEL4WS durch die Konzepte der Variablen und der Korrelationsmengen realisiert. Eine *Variable* ist ein Container, der WSDL-Nachrichten oder XML-Schema-Typen speichern kann [Web03]. Die Zuordnung von Nachrichten an einzelne Prozessinstanzen wird durch die Angabe spezieller Eigenschaften realisiert, die durch ein *Correlation Set* referenziert werden. Die Beziehung von Nachrichten zu der zugehörigen Instanz wird dabei als Korrelation bezeichnet. Es können entweder einzelne Attribute, wie zum Beispiel eine ID zur Korrelation verwendet oder verschiedene Eigenschaften zu einem zusammengesetzten *Correlation Key* kombiniert werden [ACD+03].

BPEL4WS unterstützt außerdem explizite Konstrukte zur Fehlerbehandlung und zum Eventmanagement. Mit Hilfe beliebig vieler *Fault*- bzw. *Event-Handler* kann auf Ausnahmesituationen und eingehende Nachrichten oder Timeouts reagiert werden [ACD+03].

Aufbau

Auf Prozessebene ist BPEL4WS blockorientiert aufgebaut. Der Prozess enthält somit nur eine einzige Aktivität, welche in sich weiter strukturiert sein kann und auch graphorientierte Strukturen aufweisen kann. Neben der Aktivität besitzt der Prozess die weiteren im Meta-Modell vorgestellten Komponenten, die als optionale Bestandteile des Prozesses globale Definitionen für den gesamten Ablauf bereitstellen können [ACD+03].

Auf oberster Ebene werden der Name des Prozesses, verwendete XML-Namensräume sowie die benutzerdefinierte *Query*- und *Expression-Language* spezifiziert. Als Defaultwert für letztere wird *XPath 1.0* verwendet (vgl. [ClDe99]). Zusätzlich kann angegeben werden, ob zu „false“ evaluierte *Join*-Ausdrücke unterdrückt werden sollen (*surpressJoinFailure*), um eine *Death Path Elimination* durchführen zu können. *EnableInstanceCompensation* gibt an, ob ganze Prozessinstanzen rückgängig gemacht werden können sollen und *AbstractProcess* spezifiziert, ob es sich um einen ausführbaren Prozess oder ein Geschäftsprotokoll handelt (Codebeispiel 6).

```
<process name="ncname" targetNamespace="uri"
   queryLanguage="anyURI"
   expressionLanguage="anyURI"
   suppressJoinFailure="yes|no"
   enableInstanceCompensation="yes|no"
   abstractProcess="yes|no"
   xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

   <partnerLinks>
         <partnerLink name="ncname" partnerLinkType="qname"
                  myRole="ncname" partnerRole="ncname">
         </partnerLink>
   </partnerLinks>

   <partners>
      <partner name="ncname">
            <partnerLink name="ncname"/>
      </partner>
   </partners>

   <variables>
         <variable name="ncname" messageType="qname"
                              type="qname" element="qname"/>
   </variables>

   <correlationSets>
         <correlationSet name="ncname" properties="qname-list"/>
   </correlationSets>

   <faultHandlers>
         <catch faultName="qname" faultVariable="ncname">
                  ... some activity ...
         </catch>
         <catchAll>
                  ... some activity ...
         </catchAll>
   </faultHandlers>

   <compensationHandler>
                  ... some activity ...
   </compensationHandler>

   <eventHandlers>
         <onMessage partnerLink="ncname" portType="qname"
                     operation="ncname" variable="ncname">
            <correlations>
                  <correlation set="ncname" initiate="yes|no">
            <correlations>
            ... some activity ...
         </onMessage>
         <onAlarm for="duration-expr" until="deadline-expr">
            ... some activity ...
         </onAlarm>
   </eventHandlers>

... some activity ...

</process>
```

Codebeispiel 6: BPEL4WS Process Definition

Innerhalb des Elementes *PartnerLink* werden die konkreten *Web Services*, mit denen der Prozess interagieren soll, definiert. Jeder solcher *Partner Link* hat einen eindeutigen Namen und wird durch einen *Partner Link Type* charakterisiert. Dieser spezifiziert, welche Rollen die einzelnen Services einnehmen und definieren, welche *Port Types* zur Kommunikation bereitgestellt sind. *Partner Link Types* können durch den Erweiterungsmechanismus von WSDL auch direkt in die WSDL-Beschreibung integriert werden. Das folgende Beispiel (Abb. 30) zeigt einen *Partner Link* zwischen einem Käufer (Buyer) und einem Verkäufer (Seller) (Codebeispiel 7, vgl. [ACD+03]):

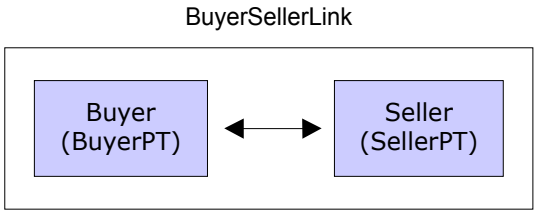


Abb. 30: BPEL4WS Partner Link Type zwischen zwei Web Services (nach [Dje04])

```
<partnerLinkType name="BuyerSellerLink">
   <role name="Buyer">
      <portType name="BuyerPT"/>
   </role>
   <role name="Seller">
      <portType name="SellerPT"/>
   </role>
</partnerLinkType>
```

Codebeispiel 7: BPEL4WS PartnerLinkType

Der *Partner Link Type* wird im Beispiel als „BuyerSellerLink“ benannt und enthält die Rollen der beiden beteiligten *Web Services* sowie die Angabe ihrer *Port Types*. Dieser *Partner Link Type* wird nun zur Definition des *Partner Links* innerhalb des *BPEL4WS*-Prozesses verwendet (Abb. 31, Codebeispiel 8):

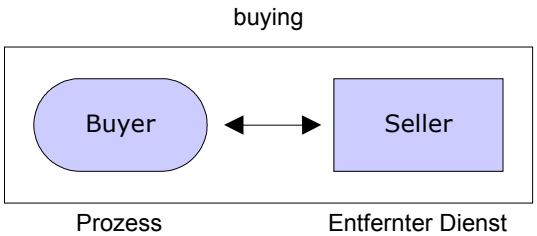


Abb. 31: BPEL4WS Partner Link (nach [Dje04])

```
<partnerLinks>
  <partnerLink
        name="buying"
        partnerLinkType="BuyerSellerLink"
        myRole="Buyer"
        partnerRole="Seller"/>
</partnerLinks>
```

Codebeispiel 8: BPEL4WS Partner Links

Im Beispielprozess tritt der *BPEL4WS*-Prozess als Nachfrager eines entfernten Dienstes auf. Die im *Partner Link Type* definierten Rollen werden den teilnehmenden Parteien fest zugeordnet. Der erstellte *Partner Link* kann nun unter dem Namen „buying“ in einer beliebigen Aktivität referenziert werden.

Optional können mehrere solcher *Partner Links* zur Definition einer *Partner*- Beziehung zusammengezogen werden. Dies ermöglicht die Kennzeichnung der Funktionalitäten, über die ein Geschäftspartners mindestens verfügen muss und erleichtert die Interaktion mit mehreren Beziehungen gleichzeitig. Ein *Partner Link* darf in höchstens einer Partnerdefinition vorkommen, kann jedoch auch ohne Zuordnung existieren [ACD+03].

Im Gegensatz zu Partnerkonstrukten können Variablen, *Correlation Sets*, *Fault Handler*, *Compensation Handler* und *Event Handler* sowohl auf globaler Prozessebene als auch in speziell definierten Gültigkeitsbereichen definiert werden. *BPEL4WS* unterstützt dazu das Konzept von *Scopes*. Ein *Scope* ist ein lokaler Gültigkeitsraum, der das Verhalten der in ihm enthaltenen Objekte bestimmt. Äußerster *Scope* ist der Prozess selbst. Jeder *Scope* besitzt zumindest eine Hauptaktivität, die das normale Verhalten innerhalb dieses Gültigkeitbereichs definiert [ACD+03]. Codebeispiel 9 zeigt die möglichen Elemente eines Scopes.

```
<scope>
  <variables> ... </variables>
  <correlationSets> ... </correlationSets>
  <faultHandlers> ... </faultHandlers>
  <compensationHandler> ... </compensationHandler>
  <eventHandlers> ... </eventHandlers>
  ... some activity...
</scope>
```

Codebeispiel 9: BPEL4WS Scope

Fault Handler behandeln Fehler, indem sie Abläufe für bestimmte Fehlertypen definieren. Dies geschieht durch ein *Catch*-Konstrukt, welches auch aus anderen Programmiersprachen bekannt ist. Alle

Fehler, die in einer Aktivität auftreten, werden an den *Fault Handler* des umschließenden *Scopes* weitergegeben. Wurden keine *Fault Handler* für den betreffenden *Scope* definiert, so wird der Fehler bis auf die oberste Prozessebene durchgereicht. Steht für einen bestimmten Fehler keine spezielle Ausnahmebehandlung zur Verfügung, so kann der Fehler dennoch durch ein optionales *Catch-All*-Element aufgefangen werden [ACD+03].

Die Schritte, die zu einer Kompensation von Aktivitäten notwendig sind, werden in *Compensation Handlern* zusammengefasst. Diese können auf *Scope*-Level definiert werden, es darf aber höchstens ein *Compensation Handler* pro *Scope* zugewiesen werden. *Compensation Handler* können durch die Aktivität *compensate* aufgerufen werden. Im Falle eines Fehlers können nun die Aktivitäten in den dafür vorgesehenen *Scopes* zurückgesetzt werden. Abbildung 32 zeigt ein Beispiel, in dem ein Fehler auf Prozessebene im äußersten Scope durch einen *Fault Handler* aufgefangen wird. Das *Catch*-Konstrukt ruft den *Compensation Handler* des mittleren *Scopes* auf. Dieser wiederum ruft den *Compensation Handler* des innersten *Scopes* auf. So werden gezielt alle Aktivitäten innerhalb des Prozesses rückgängig gemacht [BrSz03].

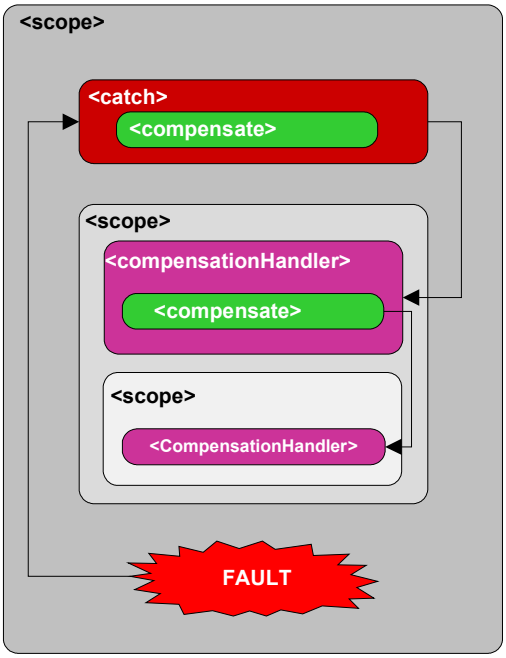


Abb. 32: BPEL4WS Compensation (aus [BrSz03])

Neben den Möglichkeiten zur Fehlerbehandlung und zur Kompensation kann jeder *Scope* ein *EventHandler*-Element besitzen. *EventHandler* können sowohl auf eingehende Nachrichten reagieren (*onMessage*) als auch sogenannte *Alarme* (*onAlarm*) verarbeiten. Letztere können für eine bestimmte Dauer (*duration*) oder einen konkreten Zeitpunkt (*deadline*) ausgedrückt werden. Das Besondere an *EventHandlern* ist, dass die durch das Event ausgelösten Aktivitäten innerhalb eines *Scopes* parallel ausgeführt werden können [Web03].

Aktivitäten

Die Definition von Aktivitäten bilden den eigentlichen Kern einer *BPEL4WS*-Definition. Wie bereits oben erläutert, kann man die hierfür zur Verfügung stehenden Konstrukte nach elementaren und strukturierten Aktivitäten differenzieren. Die zentralen *Basic Activities* von *BPEL4WS* beschreiben die Verhaltensweise bei der Kommunikation zwischen einzelnen *Web Services.* Tabelle 6 zeigt eine Übersicht dieser Sprachelemente.

Aktivität	Beschreibung
<invoke>	Aufruf eines Web Services
<receive>	Warten auf eine Nachricht bzw. einen Aufruf
<reply>	Antwort auf eine Nachricht

Tabelle 6: BPEL4WS Basic Activities zur Kommunikation mit Web Services

Die Aktivität *invoke* erlaubt den ausgehenden Aufruf einer asynchronen *one-way*- oder einer synchronen *request-response*-Operation am *Port Type* eines zuvor definierten *Partners*. Dazu werden als Attribute der global definierte *Partner Link*, der verwendete *Port Types* und die aufzurufende Operation des *Web Services* angegeben. Außerdem werden im Rahmen der Kommunikation zu sendende (*inputVariable*) und zu empfangende (*outputVariable*) Daten spezifiziert. Im Falle eines asynchronen Aufrufs entfällt die Definition einer Output-Variable, da hier keine Antwort als Bestandteil der Operation erwartet wird [ACD+03].

Im fortgesetzten Beispiel (Abb. 33, Codebeispiel 10) bietet ein entfernter *Web Service* („seller“) eine Operation „Buy“ an, die unter Angabe einer Artikelnummer („itemId“) vom *BPEL4WS*-Prozess aufgerufen werden kann. Im Falle einer *request-response*-Kommunikation sendet der *Web Service* eine Antwort an den aufrufenden Prozess zurück. Dabei kann es sich zum Beispiel um eine Fehlernachricht handeln.

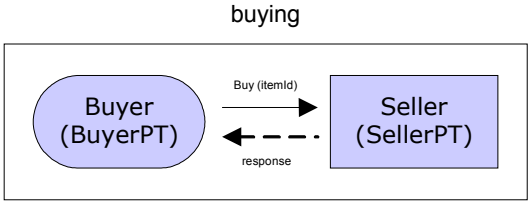


Abb. 33: BPEL4WS Invoke-Aktivität (nach [Dje04])

```
<invoke
   partnerLink="buying"
   portType="SellerPT"
   operation="buy"
   inputVariable="itemid"
   outputVariable="response"
</invoke>
```

Codebeispiel 10: BPEL4WS <invoke>

Neben der Inanspruchnahme anderer *Web Services* kann ein *BPEL4WS*-Prozess auch eigene Dienste bereitstellen, die durch die Aktivität *receive* aufgerufen werden können. Wie im umgekehrten Fall wird der entsprechende *Partner Link*, von dem der Aufruf erwartet wird, der *Port Type* und die gewünschte Operation spezifiziert. Optional kann eine Variable angegeben werden, die die eingehenden Nachrichtendaten festhält. Die Aktivität *receive* wirkt für den *BPEL4WS*-Prozess blockierend, das heißt, der Prozess wartet solange, bis der eingehende Aufruf durch einen *Partner* erfolgt ist [ACD+03].

Wenn es sich um eine synchrone Interaktion handelt, kann eine *receive*-Aktivität durch eine *reply*-Aktivität beantwortet werden. Eine Antwort im Fall einer asynchronen Kommunikation würde hingegen mittels einer *invoke*-Aktivität durchgeführt werden, die sich wieder an die korrespondierende *one-way*-Operation des Partners richtet. Die Kombination der beiden Aktivitäten *receive* und *reply* ermöglicht also eine *solicit-response*-Operation für einen WSDL-*Port-Type* des Prozesses [ACD+03].

Zum Beispiel könnte der *BPEL4WS* Prozess darauf warten, dass der entfernte *Web Service* mittels *receive* Daten zum Kaufvorgang nachfragt. Der *Partner Link* weist dann die entsprechende umgekehrte Beziehung auf. Im synchronen Fall antwortet der *BPEL4WS*-Prozess mit einem *reply* und den gewünschten Daten (Abb. 34).

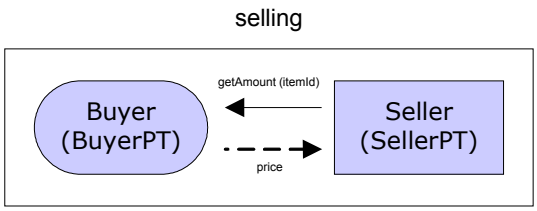


Abb. 34: BPEL4WS Receive- und Reply Aktivitäten (nach [Dje04])

Zusätzlich können durch ein *receive* neue Prozess-Instanzen erzeugt werden, falls die eingehende Nachricht mittels Korrelation keiner bestehenden Instanz zugeordnet werden kann und die „createInstance“ Option von *receive* gesetzt worden ist. Die folgenden Codebeispiele zeigen den Einsatz von *receive* und das Versenden der Rückgabewerte mittels *reply*.

```
<receive
   partnerLink="selling”
   portType="SellerPT"
   operation="getamount"
   variable="itemid"
   createInstance="yes"
</receive>
```

Codebeispiel 11: BPEL4WS <receive>

```
<reply
   partnerLink="selling"
   portType="SellerPT"
   operation="buy"
   variable="price"
</reply>
```

Codebeispiel 12: BPEL4WS <reply>

Die Entscheidung darüber, ob synchrone oder asynchrone Kommunikation zu modellieren ist, hängt von den zur Verfügung stehenden Interfaces der Service-Anbieter ab.

Neben diesen mit tatsächlichen fachlichen Aufgaben belegten Aktivitäten gibt es eine Reihe weiterer *Basic Activities* zur Unterstützung des Kontrollflusses (Tabelle 7):

Aktivität	Beschreibung
<empty>	Leere Aktivität. Diese kann zum Beispiel verwendet werden, um Fehler abzufangen oder Aktivitäten zu synchronisieren.
<wait>	Der Prozess wartet eine vorgegebene Zeitdauer oder bis zu einem vorgebenen Zeitpunkt.
<terminate>	Explizites Beenden eines Prozesses.

Tabelle 7: BPEL4WS Basic Activities für den Kontrollfluss

Der Datenfluss wird durch die Aktivität *assign* festgelegt. Hierbei werden Daten an Variablen zugewiesen, die zur Zustandsspeicherung und zur weiteren Verarbeitung genutzt werden können (Tabelle 8):

Aktivität	Beschreibung
<assign>	Zuweisung von Daten an eine Variable, zum Beispiel Nachrichten oder XML-Schema-Elemente.

Tabelle 8: BPEL4WS Basic Activities für den Datenfluss

Weitere Konstrukte dienen zur Bildung strukturierter Aktivitäten und somit zur Definition des Kontrollflusses. Es kann zwischen blockstrukturierten und graphorientierten Elementen sowie Konstrukten zur Bereitstellung von Nicht-Determinismus unterschieden werden. Tabelle 9 zeigt die möglichen Sprachelemente zur Bildung von Blöcken.

Dabei stellen die Elemente *sequence*, *switch* und *while* Aktivitäten zur Festlegung des Kontrollflusses dar, die bei der normalen, fehlerfreien Abarbeitung von Prozessen zum Einsatz kommen. *Throw, catch* und *compensate* kontrollieren die Abwicklung im Fehlerfall. *Scope* beschreibt wie schon oben genannt einen Gültigkeitsraum [ACD+03].

Aktivität	Beschreibung
<sequence>	Sequentielle Ausführung aufeinander folgender Aktivitäten
<switch> <case> <otherwise>	Modellierung von bedingten Verzweigungen, die beliebig viele Bedingungen (<case condition="boolscher Ausdruck">) zur Auswertung und optional einen alternativen Zweig (<otherwise>) enthalten kann.
<while>	Eine Schleife zur wiederholten Ausführung von Aktivitäten, bis der dazugehörige boolsche Ausdruck nicht mehr erfüllt wird.
<scope>	Ermöglicht die Einbettung von Variablen, Fehlerbehandlungs- und Kompensationsbehandlungmaßnahmen in einem lokalen Gültigkeitsraum.
<throw> <catch>	Definition von Maßnahmen zur Behandlung Interner Fehler eines Scopes oder Prozesses.
<compensate>	Aufruf einer vorher festgelegten Kompensationsbehandlungsmaßnahme. Ein Aufruf ist nur aus einem Fault-Handler oder einem CompensationHandler aus einem anderen Scope möglich.

Tabelle 9: BPEL4WS Structured Activities für den blockorientierten Kontrollfluss

Um auch Abhängigkeiten zwischen Blockdefinitionen ausdrücken zu können, verwendet *BPEL4WS* eine Aktivität, in der graphorientierte Strukturen definiert werden können. Das *<flow>*-Element stellt dazu einen Container für weitere Aktivitäten dar, die innerhalb der definierten Sektion durch Transitionen verbunden sind und somit auch parallel ausgeführt werden können (Tabelle 10). Das Konstrukt *link* spezifiziert dabei eine Transition von einer Aktivität zur nächsten [ACD+03].

Aktivität	Beschreibung
<flow>	Ermöglicht die parallele Verarbeitung von einer oder mehreren Aktivitäten. Mittels dem <link> Konstrukt können direkt oder indirekt eingebettete Aktivitäten synchronisiert werden.

Tabelle 10: BPEL4WS Structured Activities für den graphorientierten Kontrollfluss

Zur Verknüpfung der Transitionen können alle Aktivitäten mit den Attributen *source* und *target* um die Angabe ihrer Quell- und Zielaktivitäten erweitert werden. Eine einfache parallele Verzweigung mit anschließendem *Join* kann mit *flow* zum Beispiel wie folgt modelliert werden (Codebeispiel 13):

```
<flow>

  <links>
    <link name="A-to-B"/>
    <link name ="A-to-C"/>
    <link name ="C-to-D"/>
    <link name ="B-to-D"/>
  </links>

  <invoke name="A"> ...
    <source link="A-to-B"/>
    <source link="A-to-C"/>
  </invoke>
```

```
  <receive name="B"> ...
     <source link="B-to-D"/>
     <target link="A-to-B"/>
  </receive>

  <receive name="C"> ...
     <source link="C-to-D"/>
     <target link="A-to-C"/>
  </receive>

  <receive name="D"> ...
     <target link="B-to-D"/>
     <target link="C-to-D"/>
  </receive>

</flow>
```

Codebeispiel 13: BPEL4WS <flow>

Jede Aktivität, die Ziel eines Links ist, besitzt ein implizites Attribut *JoinCondition*, welches evaluiert, ob wenigstens ein eingehender Link den Wert „true" aufweist. Diese default *JoinCondition* kann durch benutzerdefinierte boolsche Ausdrücke erweitert und damit explizit gemacht werden. Zusätzlich können Transitionsbedingungen gesetzt werden, die evaluieren, ob ein Transitionsübergang erlaubt ist [ACD+03].

Schließlich sind mit der Aktivität *pick* auch nicht-deterministische Konstrukte möglich (Tabelle 11):

Aktivität	Beschreibung
<pick>	Wartet auf das Eintreten eines Events und führt danach die dafür definierte Aktivität aus

Tabelle 11: BPEL4WS Nicht-Determinismus

Die *pick*-Aktivität wartet auf das Eintreten eines bestimmten Ereignisses aus einer Menge von Ereignissen und führt dann die dazu definierten Aktionen aus. Die Events können entweder aus einer eingehenden Nachricht bestehen (*onMessage*), welche über Korrelation der betreffenden Instanz zugeordnet worden ist, oder an ein zeitliche Ereignis gebunden sein (*onAlarm*) [ACD+03].

Bewertung

BPEL4WS zielt mit einem starken Fokus auf die Komposition von *Web Services* auf die Definition von Prozessen im B2B-Bereich ab. Die Verzahnung von Prozessen steht dabei gegenüber einer schlichten funktionsorientierten Definition von Arbeitsschritten im Vordergrund [Dub02]. Dabei wird eine Kooperation zwischen zwei Prozessteilnehmern immer aus der Sicht eines der Partner beschrieben. Eine Verzahnung zwischen verschiedenen Prozessen auf mobilen Geräten ist zwar möglich,

wird aber im Rahmen dieser Arbeit mit dem Ziel einer möglichst ressourcenschonenden Komposition nicht angestrebt. Die Beschreibung der Interaktion aus der Sicht eines bestimmten Prozessteilnehmers ist bei der Weitergabe der Prozessbeschreibung an andere mitwirkende Parteien hinderlich, da es hierbei zu Kontextänderungen kommen kann.

Da es keine zentrale Komponente in Szenario untereinander interagierender mobiler Systeme gibt, stellt es keinen echten Vorteil dar, vollständige Prozesse wiederum als *Web Services* anbieten zu können. Allerdings kann dieses Konzept für die Formulierung von Subprozessen Anwendung finden.

Die exklusive Komposition von *Web Services* lässt zudem wenig Platz für andere Aktivitäten. So gibt es unter anderem keine Möglichkeit direkte Interaktionen mit Benutzern zu definieren. Aktivitäten können nicht manuell ausgeführt werden oder Benutzer und Administratoren zur Verwaltung oder zur Behebung von auftretenden Fehlern herangezogen werden [Dub02]. Es wäre sicher eine unvorteilhafte alternative Möglichkeit, einen *Web Service* zu einer Anwendung zur Verfügung zu stellen, die die anstehende Aufgabe einem menschlichen Benutzer präsentieren muss, nur um manuelle Aktivitäten in den Prozess einzubinden.

Die Erweiterbarkeit von *BPEL4WS* ist dabei als eher dürftig anzusehen: Die Sprache erlaubt benutzerdefinierte Erweiterungen durch zusätzliche Konstrukte aus anderen XML-Namensräumen solange hierbei die Semantik von *BPEL4WS*-Elementen oder Attributen nicht verändert wird. Spezielle Konstrukte zur Einführung neuer Elemente bietet *BPEL4WS* jedoch nicht an.

Im Gegensatz zu graphstrukturierten Sprachen wie zum Beispiel *XPDL* wird der Kontrollfluss nicht durch Konstrukte zwischen Aktivitäten, sondern durch die Aktivitäten selbst festgelegt. Dadurch kommt es zu einer Zunahme an strukturierten Aktivitäten, die keine fachliche Aufgabe enthalten und nur zu Routingzwecken existieren. Im Gegenzug dazu fallen jedoch zusätzliche Transitionsinformationen weg. Die Mächtigkeit der Sprache, die sich nicht zuletzt aus der Kombination von blockorientierter und graphorientierter Struktur ergibt, stellt dennoch hohe Anforderungen an die Rechenleistung der Ausführungsumgebung, was die Bearbeitung von Prozessen insbesondere auf schwächeren mobilen Systemen erschwert. Die Blockstrukturierung hat zudem den Nachteil, dass bereits abgearbeitete Prozessteile nicht ohne weiteres aus der Prozessbeschreibung entfernt werden können, um Ressourcen für Übertragung und Bearbeitung des Prozesses einzusparen.

Sehr vorteilhaft ist hingegen, dass *BPEL4WS* sowohl synchrones als auch asynchrones Kommunikationsverhalten unterstützt und daher gut für die Modellierung asynchroner Beziehungen im Umfeld mobiler Systeme geeignet ist. Die Unterstützung von Fehlerbehandlungsmaßnahmen ist für die fehleranfällige Mobilkommunikation ebenfalls ein sehr wichtiger Aspekt. Zudem werden Transaktionen zumindest durch die Möglichkeit der Definition von Kompensationsaktivitäten unterstützt.

Die Auswahl geeigneter Dienste zur Laufzeit wird durch *BPEL4WS* ohne Umwege ermöglicht, da sich die Sprache auf die Einbindung der abstrakten Bestandteile der WSDL-Definition beschränkt. So kann ein Service zur Designzeit des Prozesses definiert und erst kurz vor der Ausführung an einen konkreten Dienst gebunden werden. Zudem stellt ein gesamter *BPEL4WS*-Prozess wieder einen *Web Service* dar, was die Einbindung von Subprozessen vereinfacht. Diese können wieder als atomare *Web Services* betrachtet und integriert werden.

4.2.3 ebBPSS

ebBPSS ist ein Schema für die Spezifikation von Geschäftsprozessen (*Business Process Specification Schema*). Dieses ist Teil des semantischen Rahmens von *ebXML (electronic business XML)*, einem Projekt zur Bereitstellung einer offenen XML-basierten Infrastruktur zum weltweiten Austausch elektronischer Geschäftsdaten. Das Projekt wurde 1999 von OASIS und UN/CEFACT begonnen und hat mittlerweile viele weitere Mitglieder, die sich an der Entwicklung beteiligen, unter anderem Sun, OAG und IBM sowie zahlreiche Standardisierungsorganisationen [WHB01].

Ziel des Schemas *ebBPSS* ist es, die Modellierung von Geschäftsprozessen mit der Spezifikation von eBusiness-Softwarekomponenten zu verknüpfen, so dass Unternehmen ihre Geschäftsprozesse auf eine einheitliche und konsistente Art und Weise definieren können und eine unternehmensübergreifende Zusammenarbeit möglich wird. Dazu arbeitet *ebBPSS* mit den *ebXML*-Dokumenttypen *CPP (Collaboration Protocol Profile*) und *CPA (Collaboration Protocol Agreement*) zusammen, die Eigenschaften von *ebXML*-Teilnehmern und ihre konkreten Kooperationsvereinbarungen beschreiben (vgl. [Nau03]). Eine unternehmensinterne Repräsentation von Abläufen wird nicht unterstützt.

Eine *ebBPSS*-Definition enthält die Festlegung und die Choreographie von Geschäftstransaktionen. Diese Prozessdefinitionen können als XML-Schema-Beschreibung und als Beschreibung in der *Unified Modeling Language (UML)* ausgedrückt werden, wobei letztere in erster Linie der (graphischen) Geschäftsprozessmodellierung dient [Rie01]. In dieser Betrachtung soll nur auf die Beschreibung im XML-Format eingegangen werden.

Metamodell

ebBPSS basiert auf dem Metamodell der *UN/CEFACT-Modellierungsmethodologie (UMM)* (vgl. [Oas02]). Abbildung 35 zeigt eine vereinfachte Übersicht der *ebBPSS*-Kernkomponenten.

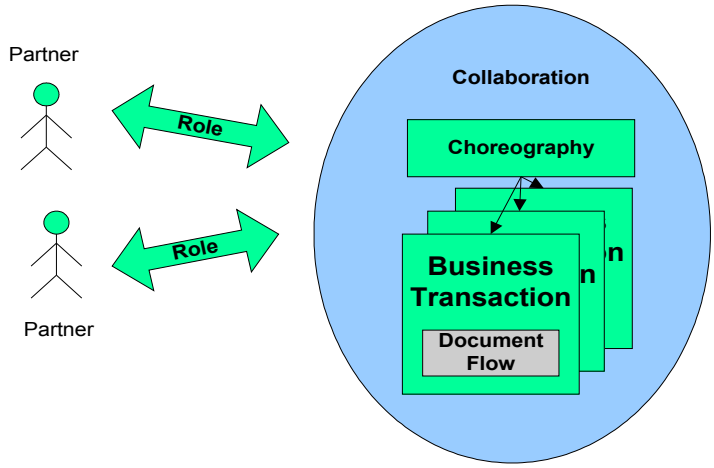


Abb. 35: ebBPSS-Kernkomponenten (aus [Rie01])

Ein Geschäftsprozess wird in *ebXML* durch eine Choreographie beschrieben. Dabei versteht man unter einer Choreographie den geordneten Ablauf der Aktivitäten von teilnehmenden Geschäftspartnern.

Die Geschäftpartner interagieren miteinander über abstrakte Rollen, die die *Service Interfaces* der Kooperation darstellen. In *ebBPSS* sind dafür zwei Geschäftsarten vorgesehen, die das Zusammenspiel von Prozessteilnehmern (*Business Collaboration*) beschreiben. Im einfachsten Fall nehmen zwei Geschäftspartner an der Zusammenarbeit teil (*Binary Collaboration)*. Hierbei wird festgelegt, wer der Initiator des Vorgangs ist (*Initiating Role*) und wer auf die Anfrage reagieren soll (*Responding Role*). Eine *Multi Party Collaboration* beschreibt die Zusammenarbeit von mehr als zwei Parteien innerhalb eines Geschäftsprozesses. Diese wird aus mehreren *Binary Collaborations* zusammengesetzt, so dass die Prozessteilnehmer auch gleichzeitig unterschiedliche Rollen zueinander einnehmen können [Rie01].

Business Collaborations bestehen generell aus einer Reihe von *Geschäftstransaktionen (Business Transactions*). Dabei bezeichnet der Begriff der Transaktion den Austausch von Informationen oder Geschäftsdokumenten zwischen zwei Unternehmen (*Business Documents*). Eine *Business Transaction* ist dabei die kleinste atomare Einheit in einem *ebBPSS*-Prozess und stellt eine Transaktion im Sinne des ACID-Paradigmas dar (vgl. 3.2.4). Je nach beabsichtigter Transaktionssemantik kann eine *Business Transaction* aus einer *Requesting Business Activity* bestehen, die ein *Business Document* versendet, ohne auf eine Antwort zu warten oder zusätzlich eine optionale *Responding Business Activity* beinhalten, die eine Antwort auf die *Requesting Business Activity* darstellt [Rie01].

Eine Choreographie enthält eine Menge von Zuständen und eine bestimmte Sequenz von Transitionen. Zwischen dem spezifizierten Anfangszustand (*start state*) und einem Endzustand (*completion state*) kann ein Geschäftsprozess beliebig viele Zustände aufweisen. Die Übergänge zwischen den Transitionen (*transitions*) werden durch Handlungen eines Unternehmens bestimmt oder durch die Auswertung von Bedingungen (*Condition Guards*) beeinflusst. Parallele Ausführungen werden durch spezielle *Fork*- und *Join-Zustände* ermöglicht [Rie01].

Die erläuterten Beziehungen des Schemas sind noch einmal als UML-Klassendiagramm im Anhang graphisch dargestellt (Anhang 1).

Aufbau

Wie bereits erwähnt handelt es sich bei *ebBPSS* um eine graphstrukturierte Definitionssprache. Die Wurzel eines *ebBPSS*-Dokumentes ist das *ProcessSpecification*-Element. Hier werden global verwendete *Business Documents*, *Binary* und *Multi Party Collaborations* sowie *Business Transactions* deklariert. Eine erweiterte Strukturierung der Elemente erfolgt durch die Bildung von *Packages*. *Packages* definieren einen Namensraum und stellen Behälter für zusammengehörige Beschreibungen und wiederverwendbare Elemente wie *Collaborations* und *Transactions* dar. Mit zusätzlichen *Include*-Anweisungen können zudem weitere externe Beschreibungen in die Definition eingebunden werden. Codebeispiel 14 zeigt die vereinfachte Grundstruktur einer *ebBPSS*-Prozessbeschreibung.

```
<ProcessSpecification name="Example" version="1.1" uuid="[1234-5678- 3394901234]"
   <BusinessDocument name="Catalog Request"/>
   <BusinessDocument name="Catalog"/>
     ...
   <Package name="Ordering">
      <MultiPartyCollaboration name="DropShip"> ... </MultiPartyCollaboration>
      <BinaryCollaboration name="FirmOrder"> ... </BinaryCollaboration>
      <BusinessTransaction name="Catalog Request"> ... </BusinessTransaction>
      <BusinessTransaction name="Create Order"> ... </BusinessTransaction>
       ...
   </Package>
</ProcessSpecification>
```

Codebeispiel 14: ebBPSS Process Specification (nach [Rie01])

Eine *Binary Collaboration* enthält zwei definierte Rollen, die festlegen, welcher Akteur eine Zusammenarbeit initiieren darf (*InitiatingRole*) und wer zur Erteilung einer Antwort festgelegt ist (*RespondingRole*). Diese können als *AuthorizedRoles* in einer oder mehreren *Business Transaction Activities* referenziert werden und so das Verhalten der Prozessteilnehmer während der Ausführung bestimmen. Eine *Business Transaction Activity* stellt dabei die Ausführung einer *Business Transaction* dar, die ihrerseits über die Angabe ihres Namens referenziert wird (Codebeispiel 15).

```
<BinaryCollaboration name="Firm Order" timeToPerform="P2D">
   <Documentation> ... </Documentation>
   <InitiatingRole name="buyer"/>
   <RespondingRole name="seller"/>
   <BusinessTransactionActivity name="Create Order"
         businessTransaction="Create Order"
         fromAuthorizedRole="buyer"
         toAuthorizedRole="seller"/>
</BinaryCollaboration>
```

Codebeispiel 15: ebBPSS BinaryCollaboration (nach [Rie01])

Codebeispiel 16 zeigt den Aufbau einer einfachen Business Transaction mit nur einem Dokument. Der Datenfluss wird in *ebBPSS* nicht explizit formuliert, sondern wird durch das Senden und Empfangen der Dokumente durch die *Business Partner* festgelegt [Rie01]. Der Anfragevorgang wird mit dem Element *Requesting Business Activity* abgebildet. Dieses definiert genau einen *Document Envelope*, in dem das Geschäftsdokument sowie optionale Zusatzdokumente versandt werden. Die *Responding Business Activity* kann als Antwort hingegen mehrere *Document Envelopes* definieren, zum Beispiel für die Bestätigung oder die Ablehnung eines Angebots.

```
<BusinessTransaction name="Notify of advanceshipment">
   <RequestingBusinessActivity name="">
      <DocumentEnvelope BusinessDocument name="ASN"/>
   </RequestingBusinessActivity>
   <RespondingBusinessActivity name=""
   </RespondingBusinessActivity>
</BusinessTransaction>
```

Codebeispiel 16: ebBPSS Business Transaction (aus [Rie01])

Die Choreographie von Geschäftsvorgängen wird durch die Definition einer Reihenfolge unter den *Business Activities* festgelegt. Verknüpft werden können *Business Activities* innerhalb von *Binary Collaborations* oder zwischen mehreren *Binary Collaborations*, falls diese Teil einer *Multi Party Collaboration* sind. Die Definition des Kontrollflusses ist also der Definition der Geschäftspartner untergeordnet. Zur Festlegung des Kontrollflusses stehen Transitionen zur Verbindung einzelner Zustände sowie *Fork*- und *Join*-Zustände zur Verfügung. Daneben können Start- und Endzustände einer Zusammenarbeit gekennzeichnet werden. Durch *Condition Guards* können zusätzlich der Status des *Document Envelopes*, der Dokumenttyp oder der Inhalt des Dokuments ausgewertet und entsprechend darauf reagiert werden [Rie01]. Im Codebeispiel 17 wird festgestellt, ob die Transaktion mit dem Namen „Notify Shipment“ erfolgreich ausgeführt wurde und demnach das Ergebnis der Zusammenarbeit bewertet. Diese Art, über „Success“ oder „Failure“ zu entscheiden, stellt gleichzeitig das *Exception Handling* von *ebBPSS* dar [Rio02].

```
<BinaryCollaboration name="Product Fulfillment"
    ...
   <BusinessTransactionActivity name="Create Order" .... />
   <BusinessTransactionActivity name="Notify shipment" .../>

   <Start toBusinessState="Create Order"/>
   <Transition
       fromBusinessState="Create Order"
       toBusinessState="Notify shipment"/>

   <Success fromBusinessState="Notify shipment"
       conditionGuard="Success"/>
   <Failure fromBusinessState="Notify shipment"
       conditionGuard="BusinessFailure"/>

</BinaryCollaboration>
```

Codebeispiel 17: ebBPSS Choreographie (aus [Rie01])

Business Transactions

Die Durchführung einer Transaktion wird durch die in der *Business Collaboration* spezifizierten Rollenverteilung bestimmt. Dabei beginnt der initiierende Teilnehmer immer mit einem Request und übergibt die Kontrolle dann dem angesprochenem Geschäftspartner. Dieser führt seine entsprechende Aktivität aus und schickt gegebenenfalls eine Antwort. Der Abschluss einer Transaktion kann dabei bei beiden Partnern erfolgreich sein oder bei einem von beiden misslingen. Fehler treten unter anderem bei der Überschreitung von Timeouts oder beim Eintreten von Ausnahmen auf, zum Beispiel bei nicht parameterkonformen Antworten oder Übertragungsproblemen. In diesem Fall müssen die Transaktionen auf beiden Seiten zurückgesetzt werden. Transaktionen können nicht geschachtelt werden und es existieren keine Konstrukte, um Kompensationsaktivitäten zu definieren [Rio02].

Das Transaktionsmodell von ebBPSS ist auf lang andauernde Geschäftstransaktionen ausgerichtet und stellt hierfür zusätzlich eine Reihe von Quality-of-Service-Attributen bereit, die neben der Choreographie von Aktivitäten die Rahmenbedingungen der geschäftlichen Zusammenarbeit behandeln.

Zum einen kann eine Transaktion mit dem Attribut *isLegallyBinding* gekennzeichnet werden, um zu spezifizieren, ob durch die Zusammenarbeit zweier Parteien ein rechtlich bindender Vertrag zustande kommt. Sicherheitsaspekte in Bezug auf Benutzer werden durch die Konstrukte *isNonRepudiationRequired* und *isAuthorizationRequired* berücksichtigt. Mit *isNonRepudiationRequired* werden Partner angewiesen, die ausgetauschten Dokumente für eine spätere Einsichtnahme als *Audit Trail* zu speichern, um das nachträgliche Abstreiten von erfolgten Geschäftshandlungen zu erschweren. Durch *isAuthorizationRequired* wird die Identität von Geschäftspartnern überprüft. Ähnliche Sicherheitsmaßnahmen bestehen auch für die versendeten Dokumente: Mittels *isConfidential* wird eine Verschlüsselung des Dokumentes angefordert, *isTamperProof* vermeidet versehentliche oder bösartige Manipulationen am Dokument, etwa durch Bereitstellung einer Checksumme, und mit *isAuthenticated* weist der Absender auf eine beigefügte digitale Signatur zu seiner Identifikation als Urheber des Dokuments hin. Ein weiterer wichtiger Punkt ist die Verlässlichkeit der Kommunikation. Der Parameter *isGuaranteedDeliveryRequired* gibt an, dass die Zustellung von Dokumenten in jedem Fall garantiert sein muss, zum Beispiel dadurch, dass Übertragungskanäle von einer dritten Partei überwacht werden [Rie01].

Collaboration Protocol Profile und Collaboration Protocol Agreement

Neben der Spezifikation einer Beschreibungssprache für die Zusammenarbeit verschiedener Unternehmen definiert das *ebXML*-Framework zwei weitere Dokumenttypen, die aufgrund ihrer Relevanz zur Kommunikation nicht-funktionaler Aspekte an dieser Stelle kurz betrachtet werden sollen. Es handelt sich hierbei um die bereits angesprochenen XML-Dokumente *Collaboration Protocol Profile (CPP)* und *Collaboration Protocol Agreement(CPA)*.

Das *CPP* beschreibt die nicht-funktionalen Eigenschaften eines Unternehmens, die benötigt werden, um an einer *Business Collaboration* teilzunehmen. Ihre Eignung hierzu wird einerseits durch technische Begebenheiten, wie unterstützte Transportprotokolle oder Sicherheitsmechanismen be-

schrieben und andererseits durch betriebliche Eigenschaften, wie involvierte Geschäftsfelder oder Kontaktinformationen charakterisiert. Durch die Möglichkeit, im *CPP* eine digitale Signatur bereitzustellen, können Dokumente einer teilnehmenden Partei im weiteren Verlauf der Kommunikation eindeutig gekennzeichnet werden [Nau03].

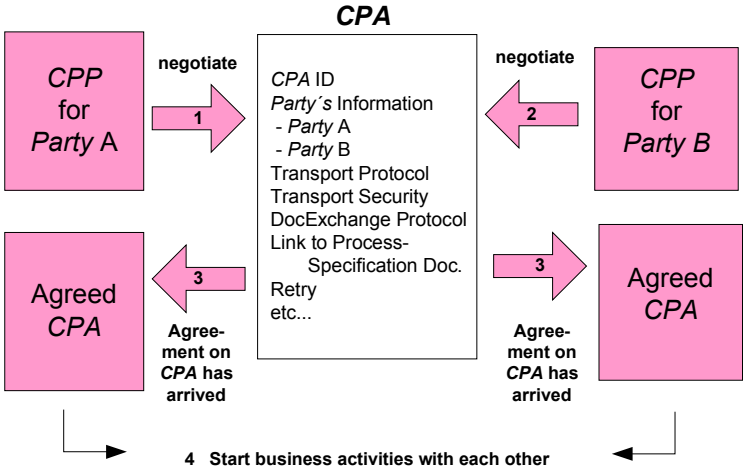


Abb. 36: Überblick über Collaboration Protocol Profile (CPP) und Collaboration Protocol Agreement (CPA) (aus [Nau03])

Verhandeln zwei *ebXML*-Business-Partner miteinander, können die entsprechende *CPP*-Dokumente ausgetauscht werden, um eine Einigung auf gemeinsame Konditionen herbeizuführen. Die so verhandelten Eigenschaften werden in einem *Collaboration Protocol Agreement* festgehalten (Abb. 36). Neben den Attributen des *CPP* enthält das *CPA* weitere Elemente, die zum Beispiel den Status oder das Datum der Vereinbarung enthalten [Nau03].

Bewertung

ebBPSS konzentriert sich sehr stark auf eine Definition von Geschäftsbeziehungen. Dabei steht nicht etwa die Spezifikation des Kontrollflusses zwischen einzelnen Aktivitäten im Vordergrund, sondern der Aspekt der Zusammenarbeit zweier Geschäftspartner. Es stehen viele unterstützende Konzepte wie *CPP* und *CPA* sowie die Definition von Sicherheitsbedingungen und Vertragskriterien zur Verfügung, um Partner im B2B-Bereich zu bedienen. Andere allgemeine Aspekte einer Prozessbeschreibungssprache, wie die Behandlung von Fehlern oder die Interaktion mit Benutzern werden dafür nicht in das Schema zur Definition von Prozessen integriert, sondern durch den Einbezug anderer Komponenten des Frameworks unterstützt.

Im Gegensatz zu den bisher betrachteten Beschreibungssprachen stellt *ebBPSS* jedoch Konzepte zur Verfügung, um auch nicht-funktionale Aspekte einer Kollaboration zu definieren. Insbesondere die Integration der *ebXML*-Komponenten *CPP* und *CPA* ist für die mobilen Systeme ein viel versprechender

Ansatz, um unterschiedliche Eigenschaften und Leistungspotentiale von mobilen Geräten mit den Anforderungen des Prozesses abzugleichen und somit geeignete Prozessteilnehmer auswählen zu können.

Unvorteilhaft ist jedoch, dass Beziehungen zwischen mehreren Prozessteilnehmern über den Umweg der *Binary Collaborations* erstellt werden müssen. Definitionen für mehrere Partner werden in einer *Multi Party Collaboration* schnell unübersichtlich. Als eine Folge davon ist auch die Verknüpfung von *Transaction Activities* zwischen mehreren *Binary Collaborations* sehr umständlich, so dass eine Spezifikation von Kollaborationen übergreifenden Kontrollflussdefinitionen sehr komplex und nur eingeschränkt genutzt werden kann [Rio02].

Für eine Anwendung im Bereich mobiler Systeme ist die Struktur von *ebBPSS* zu sehr auf die Definition von binären Kooperationen ausgerichtet. Insbesondere ist es unpassend, dass die Definition des Kontrollflusses der Definition der Prozessteilnehmer untergeordnet ist. Diese Strategie ist gegenläufig zur primären Festlegung eines Ablaufplans, bei dem die auszuführenden Aufgaben und ein globales Ziel im Vordergrund stehen und Prozessteilnehmer erst flexibel zur Laufzeit ausgewählt werden.

4.2.4 WSCI

WSCI (Web Service Choreography Interface) ist das Produkt einer Kooperation von SAP, Sun, BEA und Intalio und wird zur Zeit von W3C veröffentlicht und diskutiert. Es handelt sich dabei um eine XML-basierte Beschreibungssprache, um Abläufe für den Austausch von Nachrichten zur gemeinsamen Kollaboration zu definieren. Genauer gesagt stellt *WSCI* eine Art Erweiterung von WSDL dar, um das Zusammenspiel verschiedener *Web Services* zu unterstützen. Alternativ können neben WSDL auch andere Definitionssprachen als Basis verwendet werden, die jedoch ähnliche Charakteristika wie WSDL aufweisen müssen [AAF+02].

WSCI wurde nicht ausschließlich für B2B-Anwendungen entwickelt und kann daher nicht nur für die Definition von Geschäftsbeziehungen zwischen verschiedenen Unternehmen, sondern auch für die Beschreibung von Schnittstellen innerhalb von Unternehmensgrenzen eingesetzt werden. Die Sprache definiert jedoch keine ausführbaren Prozesse oder konkreten Implementationen von *Web Services*. Vielmehr liegt der Fokus auf der Darstellung eines *beobachtbaren Verhaltens* (*observable behaviour*) auf Basis der zwischen den teilnehmenden *Web Services* ausgetauschten Nachrichten.

Besonders hervorzuheben ist, dass das Konzept von *WSCI* nicht davon ausgeht, dass ein einzelner Prozessteilnehmer den gesamten Ablauf der Interaktion kontrolliert, sondern dass jede am Prozess beteiligte Komponente eine oder mehrere Schnittstellen definiert, die ihre eigene Rolle an der Interaktion definiert. Eine einzelne *WSCI*-Schnittstelle spezifiziert also nur das Verhalten eines einzigen Prozessteilnehmers. Zur Spezifikation des schnittstellenübergreifenden, gemeinsamen Verhaltens kann ein globales Modell definiert werden (Abb. 37).

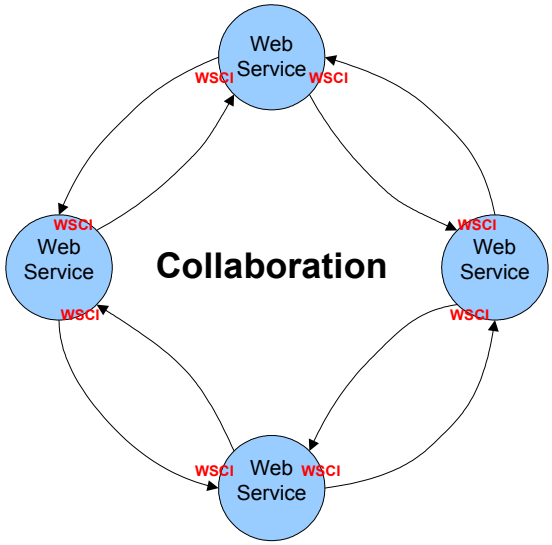


Abb. 37: WSCI Kollaboration über Schnittstellen (nach [Pel03])

WSCI kann als Bindeglied zwischen den durch WSDL definierten konkreten *Web Services* und einer höheren, abstrakten Beschreibung von Prozessen angesehen werden. Dabei übernimmt *WSCI* die in der Prozessbeschreibung definierte Choreographie und bildet die dort spezifizierten Aktivitäten auf Aktionen ab, die direkt einer Operation in WSDL zugewiesen werden können. Die Referenz zwischen den in *WSCI* definierten Aktionen und den WSDL-Operationen ist dynamisch, so dass konkrete Dienste zur Laufzeit auf Basis bestimmter Kriterien ausgewählt werden können [Pel03].

Meta Modell

Wie bereits festgestellt, besteht eine *WSCI*-Beschreibung aus der Definition mehrerer Schnittstellen, die das Mitwirken der beteiligten Parteien am Prozess definieren. Ein *Interface* beschreibt die dazu erforderliche Kommunikation aus der Sicht des jeweiligen Prozessteilnehmers. Es enthält mindestens einen *Process*, welcher mit Hilfe von Kontrollflusskonstrukten alle Details zum Austausch von Nachrichten beschreibt. Prozesse können in *WSCI* als *Top Level Process* verwendet oder geschachtelt werden (*Nested Process*). Beide Ausprägungen können durch Referenzierung ihres Namens in anderen Konstrukten wiederverwendet werden. Für Subprozesse stehen wahlweise sowohl hierarchische Ausführungen als auch Verhaltensweisen nach dem *Connected Discrete Model* bereit (vgl. 3.2.4).

Der Kontrollfluss innerhalb eines Prozesses ist blockstrukturiert und besteht aus verschiedenen eingebetteten Aktivitäten. Dazu unterscheidet *WSCI* zwischen atomaren und komplexen Aktivitäten. *Atomare Aktivitäten* stellen den konkreten Nachrichtenfluss eines *Web Services* dar, zum Beispiel das Senden und Empfangen einer Nachricht (*Action*) oder das Warten eines bestimmten Zeitraums (*Delay*). Sie können keine weiteren Aktivitäten enthalten und stellen somit einen atomaren Schritt innerhalb des Prozesses dar. *Komplexe Aktivitäten* werden zur Definition des Kontrollflusses verwendet. Sie können

ineinander verschachtelt werden und enthalten eine Reihe von atomaren Aktivitäten, die die eigentlichen Funktionen beschreiben. Durch komplexe Aktivitäten können sequentielle und parallele Ausführung, Schleifen und bedingte Anweisungen ausgedrückt werden [AAF+02].

Für die Definition von Variablen werden *Properties* verwendet. *Properties* referenzieren einen Wert innerhalb der Schnittstellenbeschreibung. Sie werden in erster Linie für die Zwischenspeicherung von Nachrichten selbst, einzelnen Werten aus Nachrichten oder zur Berechnung von Ausdrücken zur Evaluierung eines Bedingungskonstrukts eingesetzt. Außerdem können mit ihrer Hilfe Korrelationsmengen festgelegt werden, um eingehende Nachrichten der zutreffenden Prozessinstanz zuordnen zu können. Dieser Mechanismus wird in *WSCI* als *Correlation* bezeichnet.

WSCI definiert Fehlerbehandlungsmaßnahmen, indem Aktivitäten beschrieben werden, die ausgeführt werden sollen, sobald eine Ausnahme auftritt. Ausnahmen umfassen das Auftreten von Fehlern im Service selbst, das Empfangen einer Fehlernachricht oder das Eintreten eines Timeouts.

Für Aktivitäten, die unbedingt gemeinsam durchgeführt werden müssen, unterstützt *WSCI* ein selektives Transaktionskonzept. Je nach Anwendungskontext und voraussichtlicher Dauer der Transaktionen kann zwischen *Atomic Transactions* und *Open Nested Transactions* gewählt werden. *Atomic Transactions* besitzen keine innere Struktur und müssen mit einem externen Transaktionsmonitor durchgeführt werden. Wenn während der *Atomic Transaction* ein Fehler auftritt, so muss die gesamte Transaktion ebenfalls mit Hilfe externer Anwendungstechnik zurückgesetzt werden. Die Mechanismen hierfür werden nicht von *WSCI* vorgegeben und müssen anwendungsseitig implementiert werden. *Open Nested Transactions* sind hingegen aus mehreren atomaren oder geschachtelten Transaktionen zusammengesetzt. Für die Rücksetzung von bereits beendeten Teiltransaktionen stehen in *WSCI Compensation Activities* zur Verfügung, die die Definition semantisch rückwirkender Aktionen erlauben. Der Aufruf dieser *Compensation Activities* kann zum Beispiel durch Maßnahmen der beschriebenen Fehlerbehandlung eingeleitet werden [AAF+02].

Aktivitäten können an bestimmte Umgebungsparameter gebunden werden, die im Rahmen eines *Contexts* definiert werden. Ein *Context* beschreibt eine Menge von *Properties*, Fehlerbehandlungsmaßnahmen und Transaktionseigenschaften, die den eingebetteten Aktivitäten zugeordnet sind. Zusätzlich können eingebettete Subprozesse als lokale Elemente deklariert werden. Die Definition von verschiedenen *Context*-Elementen kann auch geschachtelt werden, so dass eine Hierarchie von lokalen Namensräumen entsteht. Fehlerbehandlung und Transaktionen sind in *WSCI* immer an einen bestimmten *Context* geknüpft [AAF+02].

Aufbau

WSCI-Beschreibungen werden innerhalb der WSDL-Datei des jeweiligen *Web Services* definiert. Es erfolgt eine Einbettung in das *wsdl:definitions*-Element, welches in seiner statischen Version ohne *WSCI* die Typdefinitionen, die Definition der verwendeten Nachrichten sowie die Angabe von Operationen und *Port Types* zur Beschreibung des *Web Services* enthält.

Zentrales Konstrukt einer *WSCI*-Beschreibung ist das *Interface*- Element, welches einen Behälter für die Definition von Prozessen darstellt (Codebeispiel 18). Es erhält einen eindeutigen Namen, da ein

Prozessteilnehmer mehrere Interfaces besitzen kann, mit denen er auf verschiedene Kontexte reagieren kann. Das Attribut *instantiation* innerhalb der Prozessdefinition gibt an, wann eine neue Instanz des Prozesses erzeugt werden soll, zum Beispiel durch den Eingang einer Nachricht. Alternativ können Prozesse explizit instantiiert werden [AAF+02].

Neben der Definition der *Interfaces* eines Dienstes werden innerhalb von *wsdl:definitions* globale Aspekte wie die Korrelation von Nachrichten behandelt.

```
<wsdl:definitions>
   ...

<wsci: interface name="TravelAgent" />
   <documentation> ... </documentation>
   <process name="PlanAndBookTrip" instantiation="message" >
            ...
   </process>
</wsci: interface>

</wsdl:definitions>
```

Codebeispiel 18: WSCI Interface Definition (nach [AAF+02])

Prozesse setzen sich in *WSCI* aus Kontrollflusskonstrukten und sogenannten *Actions* zusammen. Die *Actions* bezeichnen atomare Aktivitäten, die eine Beschreibung des Austauschs von Nachrichten zum Inhalt haben (Codebeispiel 19). Da ein *Web Service* mehrere Rollen in einer Kollaboration einnehmen kann, wird die passende Rolle für die *Action* mit Hilfe des *Role*-Attributs zugewiesen. Das Attribut *Operation* bildet die *WSCI*-Aktivität auf eine innerhalb der WSDL-Beschreibung definierte Operation ab [Pel03].

```
<action name="ReceiveTripOrder"
          role="TravelAgent"
          operation="TravelAgentToTraveler/OrderTrip">
</action>
```

Codebeispiel 19: WSCI Action (nach [AAF+02])

Ein *globales Modell* enthält alle zugehörigen Schnittstellen der Prozessteilnehmer sowie eine Auflistung der Operationen, die der angestrebten Kollaboration entsprechen. Codebeispiel 20 zeigt das stark vereinfachte globale Modell einer Kollaboration aus zwei Prozessteilnehmern, einem "TravelAgent" und einem "Traveler". Dabei besitzen die beiden Parteien ihre eigenen Schnittstellen, die im globalen Modell referenziert werden (*ref*). Die unter dem Element *connect* aufgeführten Operationen beschreiben die Verbindung zwischen den Prozessteilnehmern. In der Regel entsprechen sich die Operationen beider Seiten, denn sie stellen den Ablauf aus den verschiedenen Sichtweisen der Prozessteilnehmer dar. Die Operation "PlaceItinerary" aus Sicht des Reisenden stellt also die Operation

“ReceiveTrip” aus Sicht des Reiseveranstalters dar. Gemeinsam wird hierdurch die Kollaboration beider Parteien beschrieben [AAF+02].

```
<model name=”TravelCollaboration”/>
   <interface ref=” TravelAgent/>
   <interface ref=”Traveler”/>
   <connect operations=”TravelerToTravelAgent/PlaceItinerary
                        TravelAgentToTraveler/ReceiveTrip”/>
</model>
```

Codebeispiel 20: WSCI Globales Modell (nach [AAF+02])

Neben den genannten Konstrukten bietet *WSCI* zahlreiche Möglichkeiten zur Erweiterung an. Es können neue Typen von Aktivitäten eingeführt, die Semantik von Aktionen ergänzt oder Elemente und Attribute für bestehende Konstrukte hinzugefügt werden.

Sprachelemente zur Definition einer Choreographie

Der Kontrollfluss wird in WSCI durch strukturierte Aktivitäten (*complex activities*) definiert und als *Choreographie* bezeichnet. Es stehen acht native Elemente zur Verfügung, um die Reihenfolge, die Häufigkeit und die Konditionen der Ausführung zu beschreiben. Die von der Choreographie betroffenen Aktivitäten eines *Contexts* werden dabei als *Activity Set* bezeichnet.

Einfachstes Element zur Definition eines Ablaufs ist die Aktivität *all*, welche den spezifizierten *Activity Set* ein einziges Mal in beliebiger Reihenfolge ausführen lässt. Da keine Ordnung festgelegt ist, können die betreffenden Aktivitäten in beliebiger Sequenz oder auch parallel ausgeführt werden. Hingegen verlangt die strukturierte Aktivität *sequence* eine obligatorische Ausführung des *Activity Sets* in der angegebenen Reihenfolge.

Das *WSCI*-Element *switch* stellt eine Auswahl zwischen mehreren *Activity Sets* dar, von denen einer durch die Evaluation einer Bedingung selektiert werden soll. Die dazugehörigen *case*-Elemente enthalten dabei die jeweils auszuwertenden Bedingungen und die im Falle einer Selektion auszuführenden Aktivitäten. Wird eine Bedingung innerhalb der *switch*-Aktivität zu „true“ evaluiert, so werden alle anderen Auswahlmöglichkeiten ignoriert und die *switch*-Aktivität beendet.

Eine ereignisbasierte Auswahl zwischen mehreren möglichen *Activity Sets* wird durch die Aktivität *choice* getroffen. Auf Basis von Events wird ein bestimmter Ablauf ausgewählt und genau einmal in der angegebenen Sequenz ausgeführt. Ein Ereignis kann eine eingehende Nachricht (*onMessage*), das Eintreten eines Timeouts (*onTimeout*) oder das Auftreten eines Fehlers (*onFault*) sein. Relevant ist dabei für die Auswahl das nur jeweils erste eintreffende Ereignis.

Iterationen können in *WSCI* durch drei verschiedene Konstrukte modelliert werden. Das Element *foreach* bestimmt eine konkrete Anzahl von Durchläufen für einen *Activity Set*. Mittels der *until*-Aktivität

wird ein *Activity Set* mindestens einmal und danach wiederholt ausgeführt, bis eine Abbruchbedingung zu „true“ evaluiert. Die *while*-Aktivität operiert gleichermaßen, nur dass die Bedingung zu Beginn der Iteration evaluiert wird. Alle iterativen *Activity Sets* werden in der angegebenen Reihenfolge ausgeführt.

Weitere Aktivitäten, die nicht direkt dem Kontrollfluss dienen, aber als Hilfsaktivitäten zur Formulierung spezieller Anwendungsfälle eingesetzt werden können, sind die Elemente *delay*, *empty* und *fault*. Die *delay*-Aktivität spezifiziert eine bestimmte Wartezeit, die *empty*-Aktivität hat keinen ausführbaren Inhalt und die *fault*-Aktivität löst einen Fehler aus, der von einem entsprechenden *Fault-Handler* aufgefangen und verarbeitet werden kann.

Subprozesse werden in *WSCI* innerhalb von atomaren Aktivitäten aufgerufen. Dazu stehen die *call*-Aktivität, die einen Subprozess aufruft und wartet, bis dieser beendet ist (*Hierarchical Model*) und die *spawn*-Aktivität, die sofort nach Aufruf des Subprozesses beendet wird und somit den Subprozess parallel oder sequentiell zum Elternprozess ablaufen lässt (*Connected Discrete Model*). Für durch die *spawn*-Aktivität aufgerufene Subprozesse existiert zusätzlich eine optionale *Join*-Aktivität, die auf die Beendigung der im selben *Context* ausgelösten Subprozesse wartet [AAF+02].

Transaktionsmanagement

Context-Elemente, die Aktivitäten enthalten, welche entweder nur gemeinsam oder gar nicht ausgeführt werden dürfen, werden durch eine *Transaction*-Definition gekennzeichnet. Diese enthält den Namen der Transaktion, den Typ der Transaktion (*atomic* oder *open*), ein *Retries*-Attribut, welches spezifiziert, ob die Transaktion wiederholt ausgeführt werden kann, und ein *Compensation*-Element mit den Aktivitäten zur Rückabwicklung der Transaktion im Fehlerfall (Codebeispiel 21). Zum expliziten Aufruf einer Kompensationsaktivität wird die Aktivität *compensate* verwendet [AAF+02].

```
<transaction name=“seatReservation“
        type=“atomic“
        retries =“0“>
        <compensation>
          <action name = „NotifyOfCancellation“
                role=“Airline“
                operation =“AirlineToTravelAgent/NotifyOfCancellation“/>
        </compensation>
</transaction>
```

Codebeispiel 21: WSCI Transaction (nach [AAF+02])

Dynamic Participation

Die Modellierung der Choreographie, der Transaktionen und der auszutauschenden Nachrichten wird vor Instantiierung der Prozesse zur Designzeit abgeschlossen. Um jedoch die Identität des konkreten teilnehmenden Services zur Laufzeit auswählen und festlegen zu können, definiert *WSCI* das Konzept der *Dynamic Participation*. Ziel ist es, die Identifikation von *Web Services* ebenfalls über die Definition von Nachrichten zu regeln, so dass Nachrichten ausgetauscht werden können, um die zur Kommunikation erforderlichen konkreten Serviceparameter, wie zum Beispiel die Portadresse, zu ermitteln [AAF+02].

Die Konstrukte *locate* und *Locator* sollen diese Aufgabe in WSCI übernehmen. Das *locate*-Element stellt eine Erweiterung des *Action*-Elements dar und identifiziert den Service, der die Nachricht der *Action* empfangen und so die anstehende Aufgabe ausführen soll. Dazu kann es eine Reihe von *Properties*, die Kriterien zum Auffinden eines geeigneten Services spezifizieren, und eine Referenz auf ein *Locator*-Element enthalten. Letzteres enthält den Namen des Mechanismus, der für die Auswahl des konkreten Services verantwortlich sein soll. Es ist entweder die Angabe von *Properties* oder die Angabe eines *Locator*-Elements erforderlich. Werden beide Attribute angegeben, kann ein spezifizierter *Locator* den Service auf Basis der definierten *Properties* identifizieren [AAF+02].

WSCI definiert jedoch keine Mechanismen, um geeignete Services auf Basis zur Laufzeit vorliegender Kriterien auszuwählen oder aufzufinden. Dazu verweist die *WSCI*-Spezifikation [AAF+02] auf ergänzende Implementationen, wie zum Beispiel *UDDI*.

Bewertung

WSCI beschreibt umfangreiche Möglichkeiten zur Kollaboration von Diensten, die nicht von einer zentralen Einheit gesteuert und verwaltet werden. Dazu gehören die dynamische Auswahl von Services zur Laufzeit, Fehlerbehandlungsmechanismen und Transaktionsunterstützung.

Die Definition von Kollaborationsbeziehungen in *WSCI* setzt jedoch voraus, dass die genau Struktur und Abfolge von Nachrichten zwischen den beteiligten Services bereits zur Designzeit des Prozesses vorliegt. Es stellt sich deshalb die Frage, inwieweit die auszutauschenden Nachrichten zwischen den Prozessteilnehmern schon bekannt sind, bevor diese überhaupt ausgewählt wurden. Die beteiligten Partner stehen im mobilen Szenario erst zur Laufzeit fest und die Informationen über konkret auszutauschende Nachrichten können eventuell auch erst nach Auffinden des Partners bezogen werden. Da diese Details Teil der WSDL-Beschreibung von *Web Services* sind, stehen sie in der Regel erst zur Verfügung, wenn auch die WSDL-Datei zum Dienst vorliegt.

Durch die Tatsache, dass die WSCI-Beschreibung in die WSDL-Datei integriert ist, können zudem nur Services am Prozess teilnehmen, die bereits vor dessen Instantiierung durch die Beschreibung eines Interfaces zur Ausführung bestimmt worden sind. Unter mehreren gleichartigen Diensten, die dieses Interface bereitstellen, kann über das Konzept der *Dynamic Participation* zur Laufzeit ein passender

Dienst gewählt werden. Ob und wie neue Dienste zur Teilnahme am Prozess akquiriert werden können, die nicht bereits explizit zur Interaktion geschaffen wurden, wird nicht weiter ausgeführt.

Da die genauen Prozessteilnehmer im mobilen Szenario zur Designzeit noch nicht bekannt sind, müsste die Prozessbeschreibung als globales Modell einer *WSCI*-Beschreibung definiert und Schnittstellen zur Laufzeit den ausgewählten Diensten zugewiesen werden. Dieser Vorgang würde eine Änderung der WSDL-Beschreibung durch die Ergänzung von Schnittstellenbeschreibungen zur Laufzeit erfordern. Somit ist *WSCI* aus der Sicht des Aufbaus nicht abstrakt genug für den beabsichtigten Einsatzzweck. Die Sprache ist eine Ebene zu tief bzw. zu nah an der technischen Implementierung angesiedelt (Abb. 38).

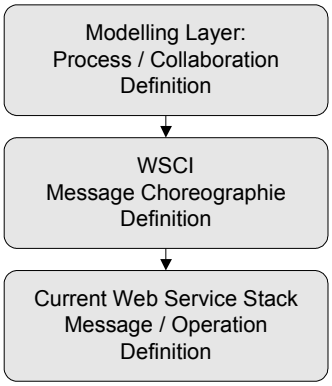


Abb. 38: Einordnung von WSCI

Hinzu kommt, dass die Beschreibungssprache auf die Kollaboration von *Web Services* beschränkt ist und zudem auf WSDL als Basis zur Beschreibung dieser Dienste fokussiert. Eine Integration von anderen Softwarekomponenten oder von Benutzerinteraktionen wird nicht in Betracht gezogen. Eine Erweiterung der *WSCI*-Konstrukte ist zwar in nahezu jeder Hinsicht möglich, die Ergänzung alternativer Prozessteilnehmer ist aber aus semantischen Gründen nur eingeschränkt möglich, da die so definierten Interfaces eventuell nicht interoperabel sind.

Das Transaktionsmanagement von *WSCI* ist jedoch für eine Anwendung auf den Bereich der mobilen Systeme interessant. Aktivitäten können durch die *Transaction*-Definition als zusammenhängend gekennzeichnet werden. Damit kann unterschieden werden, ob Aktivitäten überhaupt transaktionales Verhalten aufweisen sollen oder ob der Anwendungsfall gegebenenfalls gar keine Transaktionen benötigt. Während andere Sprachen voraussetzen, dass jede Aktivität oder jeder Prozess als ganzes eine Transaktion darstellt, können einzelne Aktionen in *WSCI* auch durchgeführt werden, ohne dass ein Mehraufwand durch die Koordination und Überwachung einer Transaktion nötig wird. Durch die Möglichkeiten, zusätzlich verschiedene Transaktionstypen kennzeichnen zu können und Kompensationsaktivitäten zu definieren, wird ausreichend Flexibilität für den Einsatz im mobilen Umfeld zur Verfügung gestellt.

4.2.5 jPDL

Die Prozessbeschreibungssprache *jPDL* ist Bestandteil von *jBPM (Java Business Process Management)*. Dabei handelt es sich um ein Workflow-Management-System, welches in der Programmiersprache Java implementiert ist und Konzepte zur Prozessbearbeitung bereitstellt. *jBPM* ist Teil der *JBoss*-Plattform, kann jedoch auch in eigene Anwendungen integriert werden, die nicht im JBoss deployed sind [Koe04].

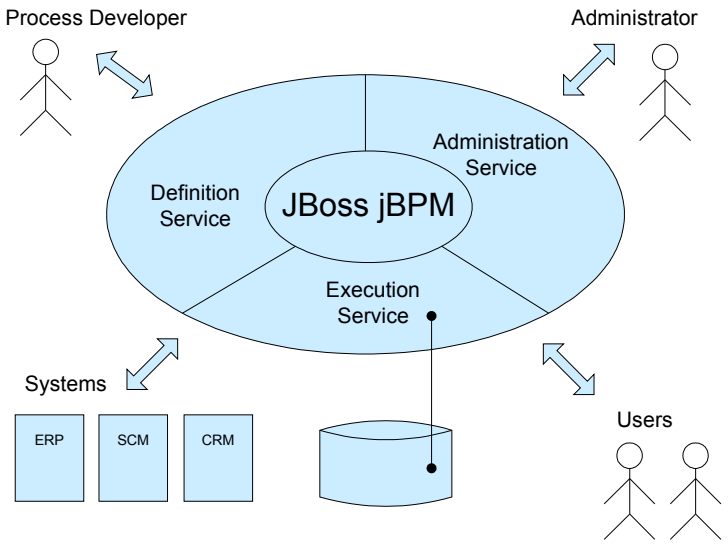


Abb. 39: jBPM Komponenten (aus [Koe04])

jBPM verarbeitet Prozesse mit Hilfe einer *State Engine,* die die Zustände der auszuführenden Aufgaben verwaltet. Die Architektur von *jBPM* besteht aus drei Komponenten (Abb. 39): Der *Definition Service* steht zur Verfügung, um Prozessdefinitionen einzulesen und zu interpretieren. Diese können mittels des *Execution Services* ausgeführt werden. Zur Bearbeitung von Aufgaben können dazu sowohl menschliche Benutzer als auch Computersysteme herangezogen werden. Die Verwaltung und Protokollierung der Prozesse findet durch den *Administration Service* statt [Koe04].

Um Prozesse für *jBPM* zu formulieren, steht die native Prozess-Sprache *jPDL* zur Verfügung. *jPDL* steht für *jBPM Process Definition Language* und ist XML-basiert. Neben *jPDL*-Prozessen wird seit kurzer Zeit außerdem die Abarbeitung von *BPEL4WS*-Prozessen durch *jBPM* unterstützt.

Meta Modell

Nach Baeyens [Bae04] besteht eine Prozessdefinition aus vier Schichten (Layers): Zustände, Kontexte, Programmierlogik und Benutzerinteraktion.

Der *State Layer* spezifiziert die Zustände und den Kontrollfluss eines Prozesses. Dieser wird durch ein Transitionensystem festgelegt und ist demnach graphorientiert strukturiert. *jPdl* verfeinert jedoch den Begriff der Aktivität, indem es zwischen Zuständen und Aktionen unterscheidet. *Zustände (states)* stellen Abhängigkeiten zu externen Akteuren dar. Sie werden auch als Wartezustände bezeichnet, da das Workflow-Management-System die Verarbeitung anhalten muss, bis der betreffende Akteur die Bearbeitung der *Aufgabe* (*Task*) beendet hat. Hingegen handelt es sich um eine *Aktion*, wenn das Workflow-Management-System aufgrund eines ausgelösten Events selbst automatisiert eine zuvor definierte Programmlogik ausführt. *Aktionen* sind Inhalt des *Programmic Logic Layers.* Hierzu gehören zum Beispiel automatische E-Mail-Benachrichtigungen oder Datenbankoperationen [Bae04].

Ein Zustand enthält außerdem den Namen des ausführenden *Akteurs*. *Akteure (actors)* können dabei sowohl menschliche Benutzer oder Benutzergruppen als auch Computersysteme sein. Damit *Akteure* flexibel zuweisbar sind, werden sie über sogenannte *Swimlanes* referenziert. Ein *Swimlane* stellt einen Verantwortungsbereich dar und zeigt an, welche Aufgaben einem Prozessteilnehmer zugeordnet sind [JBo04]. Der aktuelle Zustand wird während der Ausführung des Prozesses mit einem Token versehen. Dieses wird dem jeweiligen *Akteur* zugewiesen, der die Bearbeitung der Aufgabe übernimmt. Nach Abarbeitung der Aufgabe wird das Token freigegeben und signalisiert damit die Fertigstellung der Aufgabe. Die definierte Kontrollflussstruktur entscheidet nun darüber, welcher Zustand das Token als nächstes zugewiesen bekommt [Bae04].

Der *Context Layer* enthält den Datenfluss des Prozesses. Es stehen sogenannte *Process Context Variables* zur Verfügung, um für die Bearbeitung des Prozesses erforderliche Daten aufzunehmen. Neben Daten können auch Referenzen auf Java-Objekte gespeichert werden. Daten können bereits innerhalb der Prozessbeschreibung definiert sein oder erst zur Laufzeit durch eine Interaktion mit Benutzern mit Werten gefüllt werden. Der Aufruf an einen Benutzer, eine Aufgabe zu erfüllen oder Daten als Ergebnis der Bearbeitung in den Prozess einzugeben, ist Inhalt des *User Interface Layers* [Bae04].

Aufbau

Zur formalen Beschreibung eines Prozesses wird ein *Prozessarchiv* erstellt. Dieses beinhaltet zum einen eine deklarative Beschreibung des Prozesses, nämlich eine XML-Datei namens *processdefinition.xml,* zum anderen optionale Programmierlogik in Form von Java-Klassen. Zusätzlich können weitere für den Prozess benötigte Dateien dem *Prozessarchiv* hinzugefügt werden, zum Beispiel Dokumente [jBo04].

Die eigentliche Prozessbeschreibung basiert auf einem gerichteten Graph. Dieser besteht aus Transitionen und Knoten, wobei ein Knoten von einem bestimmten Typ ist, welcher das Verhalten des Knotens festlegt. Jede Prozessdefinition in *jPDL* hat einen Namen und eine optionale Beschreibung und besitzt genau einen Start- und beliebig viele Endknoten (Codebeispiel 22).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE process-definition PUBLIC
   "-//jBpm/jBpm Mapping DTD 2.0-beta4//EN"
   "http://jbpm.org/dtd/processdefinition-2.0-beta4.dtd">

<process-definition name="sample process">

   <description> ... description of process ... </description>

   <start-state name="start">
      <transition to="somestate" />
   </start-state>

   <state name="somestate">
      <transition to="end" />
   </state>

   <end-state name="end" />

</process-definition>
```

Codebeispiel 22: jPDL Process Definition (nach [jBo05])

Menschliche Prozessteilnehmer werden auf Prozessebene deklariert. Dazu wird das Konstrukt *Swimlane* verwendet. Ein *Swimlane* stellt eine Person oder eine Rolle dar, der Aufgaben zugewiesen werden können (Codebeispiel 23). Für die Zuweisung kann zur Spezifikation der Ausführungslogik eine benutzerdefinierte Klasse in die Prozessbeschreibung integriert werden (*delegation*). Ein *Assignment-Handler* kann zum Beispiel auf eine bestehende Benutzerverwaltung verweisen [jBo05].

```
<swimlane name="manager">
    <delegation class="manager.AssignmentHandler" />
</swimlane>
  ...
<state name="somestate" swimlane="manager">
   <transition to="end" />
</state>
```

Codebeispiel 23: jPDL Swimlane

Eine ähnliche Strategie wird für die Definition von Variablen verfolgt. *jPDL* unterscheidet zwischen persistenten Variablen, deren Wert in eine Datenbank übernommen werden soll, und transienten Variablen (*transient*), die nur innerhalb der Prozessdefinition gültig sind. Variablen können im Prinzip jeden beliebigen Typ annehmen, der durch eine beigefügte *Serializer*-Implementation in das entsprechende Java-Objekt konvertiert wird. Die Einbindung des *Serializers* erfolgt wieder über das *Delegation*-Konstrukt (Codebeispiel 24). In *jBPM* sind bereits einige Java-Typen wie String, Long oder Double fest integriert [jBo05].

```
 <type java-type="java.util.Date">
   <delegation class="date.Serializer">
      date type serializer configuration
   </delegation>
   <variable name="end.date" />
   <variable name="start.date" />
 </type>

<type>
    <transient />
    <variable name="x" />
</type>
```

Codebeispiel 24: jPDL Variablen

Zusätzlich zu den manuellen Bearbeitungszuständen können *Actions* definiert werden, die auf das Eintreten bestimmter Events reagieren. Bei einer *Action* handelt es sich um Java-Code, der an einer bestimmten Stelle im Prozess ausgeführt werden soll. Eine *Action* ist immer an ein Event gebunden, welches die gewünschte Aktion auslöst. Transitionsübergänge, Aktivierung eines Knotens und Verlassen eines Knotens stellen einige der relevanten Events dar, die *Actions* auslösen können [jBo05]. Codebeispiel 25 zeigt eine *Action*, die am Ende des Prozesses eine E-Mail-Nachricht verschicken soll.

```
<action event-type="process-end" >
          <delegation class="send.notification.email"/>
</action>
```

Codebeispiel 25: jPDL Action

Zustände

Ein Zustand ist in *jPDL* ein Knoten des Graphs, der abhängig von seinem Typ ein bestimmtes Verhalten für den Kontrollfluss des Prozesses festlegt. Alle Knoten besitzen zwingend einen im Prozess eindeutigen Namen und optional eine Menge an Transitionen, Events, Timern und *Exception Handlern*. *Exception Handler* spezifizieren Aktionen, die ausgeführt werden sollen, sobald in einer *Delegations*-Klasse des Knotens ein Fehler auftritt [jBo05].

Tabelle 12 zeigt welche Zustände in Version 3.0 der Sprache existieren [jBo05]. Zustände können zur Evaluierung einer Bedingung ebenfalls mit integrierten Java-Klassen verknüpft sein. Mit Hilfe der dort hinterlegten Anwendungslogik wird die Bedingung ausgewertet und der Prozess dann je nach Ergebnis fortgesetzt.

Zustand	Beschreibung
State	Ein unspezifizierter Zustand
Process State	Ein Zustand, der einen Behälter für einen Subprozess darstellt. Die Verarbeitung des Elternprozesses wird solange angehalten, bis er Subprozess beendet wird.
Super State	Ein Zustand, der weitere Zustände gruppiert
Node	Ein Knoten des Graphs, der automatisierte Actions und Timer enthalten kann.
Task Node	Ein Knoten des Graphs, der die manuelle Ausführung einer Aufgabe (*task*) unterstützt.
Decision	Ein Knoten, der ein exklusives ODER an nachfolgenden Verzweigungen darstellt.
Fork	Ein Knoten, der die Möglichkeit der parallelen Verarbeitung nachfolgender Pfade anzeigt.
Join	Ein Knoten, der eingehende parallele Pfade synchronisiert.
Start State	Der Zustand, in dem der Prozess initialisiert wird.
End State	Endzustand des Prozesses.

Tabelle 12: jPDL Zustände (nach [jBo05])

In Codebeispiel 26 wird ein *Decision Node* durch eine hierfür bestimmte Java-Klasse unterstützt. Abhängig von deren Rückgabewert wird entweder die Transition zum Knoten „yes“ oder die Transition zum Knoten „no“ ausgeführt.

```
<decision name="simpleDecision" >
        <delegation class="check.in.ErpSystem" />
        <transition name="yes"    to="nextState" />
        <transition name="no"     to="end" />
</decision>
```

Codebeispiel 26: jPDL Decision

Bewertung

jPDL stellt eine sehr schlanke, übersichtliche Sprache zur Definition von Prozessen dar, was vor allem daraus resultiert, dass ein Großteil der Daten und der Evaluationslogik in separaten Java-Klassen untergebacht wird. Dadurch ist jedoch auch die Plattformunabhängigkeit nur noch eingeschränkt gewähr-

leistet. Zwar kann das XML-Format der Prozessbeschreibung unabhängig von der Implementierung der Ausführungsumgebung gelesen werden, jedoch ist fraglich, inwieweit die im Prozessarchiv eingebetteten Java-Klassen in die Ausführungsumgebungen der mobilen Systeme integriert werden können. Die Idee, dem Prozess benötigte Programmierlogik beizufügen, ist zwar einerseits interessant, um fehlende Komponenten auf den mobilen Systeme zu kompensieren, sorgt jedoch andererseits für einen erhöhten Datenfluss zwischen den Prozessteilnehmern. Hierbei ist es natürlich die Frage, in welchem Umfang die Einbettung der Klassen stattfindet und welches Datenvolumen diese verursachen.

Weiterhin definiert *jPDL* innerhalb der Prozessbeschreibung keine Maßnahmen zur Fehlerbehandlung. Die Autoren der Sprache gehen davon aus, dass nur in den delegierten Klassen, nicht aber im Prozess selbst Fehler auftreten können [jBo05]. Die Fehlerbehandlung findet deshalb durch Verwendung des Java-Konstrukts *try ... catch* direkt in den betroffenen Klassen statt. Es ist fraglich, ob diese Annahme der Fehlerfreiheit im Prozess im Umfeld mobiler Systeme gültig ist. Zumindest das Problem von plötzlichen Verbindungsabbrüchen bei der mobilen Kommunikation kann auf diese Weise nicht gelöst werden. Die Behandlung eines Fehlers bei einer Komponente, die nicht mehr zu erreichen ist, hat keinerlei fehlerbehebende Wirkung auf den Restprozess. Zudem erfordert eine effektive Behandlung von Fehlern Kenntnis vom konkreten Anwendungsfall. Es ist ungewiss, ob alle am Prozess teilnehmende Geräte den Anwendungsfall kennen, für den der Prozess ausgeführt wird. Vorteilhafter ist es daher, wenn die Prozessbeschreibung selbst spezifiziert, was im Fall eines Fehlers geschehen soll.

Die Definition von Web Services und deren Komposition wird von *jPDL* nicht explizit unterstützt. Web Services können aber über eine externe Beschreibung der Schnittstellen innerhalb einer inkludierten Java-Klasse eingebunden werden. Eine Unterstützung zur Definition und Ausführung von Transaktionen existiert in *jPDL* nicht, wird aber von *jBPM* für Datenbankoperationen integriert.

jPDL ist über das dargestellte Konzept der „pluggable architecture“ in Hinblick auf Funktionslogik extrem erweiterbar. Ein Konstrukt zur Definition neuer Sprachelemente für die Prozessbeschreibung existiert jedoch nicht.

4.3 Koordination durch Prozessmuster

Bei der zu entwickelnden Prozessbeschreibungssprache handelt es sich um eine Middlewarekomponente, die in erster Linie dazu dienen soll, Schnittstellen zur Definition zusammenhängender Aufgaben anzubieten und somit automatisierte Dienste zu komponieren. Da es jedoch auch offensichtlich eine inhärente Eigenschaft mobiler Geräte ist, Benutzer an beliebige Aufenthaltsorte zu begleiten, stellen Benutzerinteraktion ebenfalls einen wichtigen Anwendungsbereich für mobile Systeme dar und dürfen nicht vernachlässigt werden [Wei91].

Für den Fall, dass menschliche Benutzer also nicht nur als Initiator des Prozesses agieren, sondern auch selbst in die Ausführung des Prozesses integriert werden sollen, muss die Möglichkeit bestehen, angemessen mit dem Benutzer zu interagieren. Der Benutzer nimmt am Prozess teil, wenn es sich hierbei um eine geplante vorübergehende oder dauerhafte Benutzerinteraktion handelt oder wenn der Benutzer im Rahmen von Benachrichtigungen oder durch die Notwendigkeit einer menschlichen Fehlerbehandlung herangezogen wird. Im Rahmen einer manuell auszuführenden Aufgabe wird der Benutzer den in der Prozessbeschreibung definierten Anweisungen unterstellt. Die Kontrolle über den Prozess obliegt in diesem Fall dem System, während der Benutzer die Aufgabe zu erledigen und das Ergebnis wieder in das System einzugeben hat. Ist der Benutzer jedoch autorisiert, in die Abwicklung des Prozesses Einsicht zu nehmen oder gar einzugreifen, weil zum Beispiel ein nicht behandelbarer Fehler aufgetreten ist, so muss ihm die Kontrolle über das System und über den Prozess übergeben werden.

In diesem Abschnitt wird geprüft, ob *Prozessmuster* als Grundlage dienen können, um menschliche Benutzer in die Ausführung und Bearbeitung von Prozessen einzubeziehen. Da viele Grundlagen und Erkenntnisse dieses Paradigmas auf dem Umgang mit computergestütztem kooperativem Arbeiten basieren, soll dazu zunächst ein Überblick über *CSCW* vermittelt werden.

4.3.1 Grundlagen von CSCW

Bei *CSCW (Computer Supported Cooperative Work)* handelt es sich um ein wissenschaftliches Forschungsgebiet, welches sich mit der Theorie der rechnergestützten Gruppenarbeit beschäftigt. Im Mittelpunkt steht aus technischer Sicht die Entwicklung von Informations- und Kommunikationswerkzeugen zur Unterstützung von Gruppenarbeit. Dabei ist eine Gruppe als Menge von Personen zu verstehen, die in einer gemeinsamen Umgebung eine Aufgabe durch Zusammenarbeit erledigt und dabei ein gemeinsames Ziel verfolgt. Die an der Ausführung beteiligten Personen sind sich ihrer gegenseitigen Existenz und der Gleichzeitigkeit ihrer Handlungen bewusst. Der Umfang der Unterstützung durch Computersysteme reicht von einfachen Kommunikationshilfsmitteln bis hin zu komplexen Managementsystemen zur Automatisierung von Teilabläufen [BoSch98].

Gruppenmitglieder arbeiten oft räumlich und zeitlich verteilt zusammen, was die Art und den Einfluss der Rechnerunterstützung bestimmt. Ein direktes Zusammenwirken von teilnehmenden Personen ist nur am gleichen Ort zur gleichen Zeit möglich. Eine Zusammenarbeit an verschiedenen Orten oder zu verschiedenen Zeiten kann hingegen in der Regel nur durch ergänzende Hilfsmittel realisiert werden. Tabelle 13 zeigt die Klassifizierung von *CSCW*-Systemen nach der *Raum-Zeit-Matrix* und zeigt die auftretenden Beziehungen am Beispiel. Eine geeignete Erweiterung dieser Einteilung kann zur Betrachtung von mobilen Systemen durch den Einbezug der *Vorhersehbarkeit* von Zeiten und Orten formuliert werden [Gru94].

Raum / Zeit		Gleiche Zeit (synchron)	Verschiedene Zeit (asynchron)	
			Vorhersehbar	Nicht vorhersehbar
Gleicher Ort		Face-to-Face Sitzung	Schichtarbeit	Schwarzes Brett
Verschiedener Ort	**Vorhersehbar**	Videokonferenz	E-Mail	Kollaboratives Verfassen von Dokumenten
	Nicht vorhersehbar	Mobilfunkkonferenz	Nicht-Realzeitkonferenz, z.B Usenet	(mobile) Vorgangsbearbeitung

Tabelle 13: CSCW-Klassifizierung als Raum-Zeit-Matrix (nach [Gru94])

Die Notwendigkeit einer Koordination tritt insbesondere bei asynchroner Zusammenarbeit auf. Dabei beschreibt die Koordination die Abhängigkeiten zwischen verschiedenen Aktivitäten und den zur Erfüllung von Aufgaben verfügbaren Akteuren. Koordinationssysteme zur Unterstützung räumlich und zeitlich verteilter Arbeitsgruppen sind unter anderem die im vorhergehenden Abschnitt vorgestellten Workflow-Management-Systeme [BoSch98].

Der Verlauf der Kooperation zwischen den Gruppenmitgliedern wird durch einen sogenannten *Gruppenprozess* modelliert. Dazu werden die erforderlichen Informationen, Aktivitäten und Eigenschaften einer Gruppe definiert und festgeschrieben. Der Prozess einer rechnergestützten Gruppenarbeit hat einen fixen Anfangszustand, endlich viele Zwischenschitte und einen Endzustand, der das Arbeitsergebnis darstellt. Alle die Gruppe und deren Umgebung betreffenden Eigenschaften können als relativ statisch angesehen werden. Dazu gehören die Ziele, die Organisation, die Protokolle und die Umgebung der Gruppe. Eine Veränderung dieser Parameter erfolgt nie oder nur sehr langsam. Als dynamisch sind hingegen der Ablauf und der Zustand der Gruppenarbeit anzusehen. Gruppenaktivitäten sind je nach Fortschritt des Gruppenprozesses redefinierbar, um sich neuen Situationen anpassen zu können. Zum Beispiel können Aktivitäten gelöscht oder Dokumente ersetzt oder verändert werden [BoSch98].

Ein konkreter Ansatz, welcher sich auf die Erkenntnisse des *CSCW* begründet und bei dem die aktive Kooperation der Gruppenmitglieder im Vordergrund steht, wird durch den im Forschungsbereich der Softwaretechnik geprägten Begriff des *Prozessmusters* dargestellt. Das Konzept soll im folgenden vorgestellt werden.

4.3.2 Vergegenständlichung eines kooperativen Arbeitsprozesses

Bei dem *Prozessmuster* handelt es sich um ein softwaretechnisches Paradigma aus dem *Werkzeug-Automat-Material*-Ansatz, im folgenden kurz *WAM* (vgl. [Zül98]). Es stellt Vorlagen und Konstrukte für die Anwendungsentwicklung zur Verfügung, um die Kooperation und Koordination von Arbeitsabläufen zu vergegenständlichen, Prozesse zu formulieren und dem Anwender transparent und bearbeitbar zu machen.

Dabei steht insbesondere die Koordination von menschlichem Arbeitshandeln mit dem Ziel, kooperative Arbeit durch geeignete Anwendungssysteme zu unterstützen, im Vordergrund. Hierzu müssen begrenzte Ressourcen geteilt, Arbeitsgegenstände ausgetauscht und Arbeitshandlungen aufeinander abgestimmt werden, um eine vorbestimmte zeitliche und sachlogische Reihenfolge einzuhalten. Eine Verständigung über Zuständigkeit und Ausführung einzelner Aufgaben erfolgt in der Form eines *Plans* [Zül98].

Gryczan [Gry96] unterscheidet für die Einsetzbarkeit von *Plänen* zwischen einer unterstützenden und einer ablaufsteuernden Sichtweise. In der unterstützenden Sichtweise ist ein *Plan* nur ein Hilfsmittel, um mit punktuellen Handlungsanleitungen menschliches Arbeitshandeln zu vereinfachen. Die Initiative bei der Benutzung von Softwaresystemen geht dabei vom jeweiligen Benutzer aus, dem auch die komplette Kontrolle über den Ablauf des Prozesses unterliegt. Ein *Plan* wird hingegen als ablaufsteuernd bezeichnet, wenn er eine Verfahrensvorschrift darstellt, die das menschliche Handeln vollständig steuert. Die Kontrolle über den Ablauf einzelner Arbeitsschritte liegt hier bei einem Softwaresystem. Ziel dieses Ansatzes ist es, menschliches Arbeitshandeln durch maschinelle Funktionen zu ersetzen oder auf Dateneingaben zu reduzieren.

Als Hilfsmittel für die unterstützende Sichtweise schlägt Gryczan [Gry96] das *Prozessmuster* vor, um das Kooperationsmodell explizit und bearbeitbar zu machen. Ein *Prozessmuster* ist nach Gryczan [Gry96] ein gemeinsames Material, das den Umgang mit einer kooperativen Anwendungssituation vergegenständlicht. Es legt Verantwortlichkeiten von Personen oder menschlichen Rollenträgern fest und beschreibt die einzelnen Tätigkeiten im kooperativen Arbeitsprozess. Abhängigkeiten im Sinne von fachlichen Zusammenhängen werden durch gerichtete Pfeile dargestellt und sind im Sinne von „soll vor einer anderen Tätigkeit erledigt sein“ zu interpretieren. Zudem werden für die kooperative Arbeit notwendige Dokumente spezifiziert. Abbildung 40 zeigt ein Beispiel für den Ablauf einer Kreditvergabe, deren einzelne Tätigkeiten in einem *Prozessmuster* festgehalten werden.

Die Kontrolle über den Verlauf der Zusammenarbeit liegt dabei bei der Person, die den gemeinsamen Vorgang gerade bearbeitet. Diese kennt auch weitere mögliche oder notwendige Arbeitschritte. Grundlage sind Konventionen und ein gemeinsamer Erfahrungshintergrund aller Beteiligten.

Eine Unterstützung von kooperativer Arbeit durch *Prozessmuster* ist nach Züllighoven [Zül98] vor allem dann angebracht, wenn es sich bei der Kooperation um eine routinemäßige Zusammenarbeit handelt. Dies entspricht einer Vorgangsbearbeitung, die sich in der täglichen Praxis immer wieder ähnlich, aber in unterschiedlichen Varianten abspielen kann. Kooperierende Personen kennen sich in der Regel persönlich, tauschen miteinander Materialien aus und besitzen einen gemeinsamen Erfahrungshintergrund, wie zum Beispiel das Wissen über ihre Zuständigkeit für bestimmte Aufgaben.

Durch den starken Einbezug der Benutzer ist keine umfassende Steuerung des kooperativen Arbeitsprozesses notwendig. Die Verteilung bei der Kooperation ist daher für alle Beteiligten sichtbar.

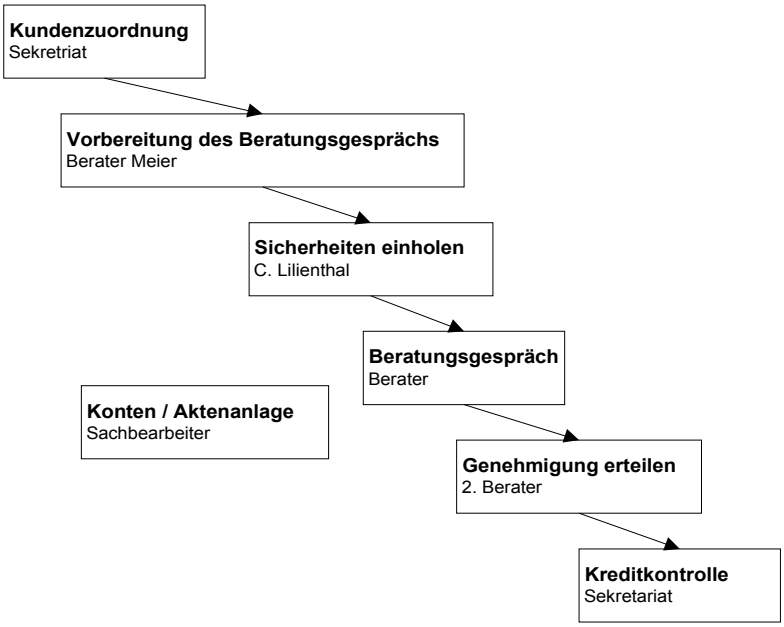


Abb. 40: Prozessmuster: Beispiel eines Prozesses zur Kreditvergabe (aus [Gry96])

Das Prozessmuster ist ein abstraktes Konzept. Praktisch realisiert wird es durch sogenannte *Laufzettel* und *Vorgangsmappen*:

Ein *Laufzettel* ist ein Arbeitsgegenstand, der von einem Anwender für einen speziellen Vorgang erstellt wird. Dazu gehört die Festlegung, wer für welche Tätigkeiten zuständig ist, welche Reihenfolge zwischen den Arbeitsschritten besteht und welche Dokumente benötigt werden. Da der *Laufzettel* während des Vorgangs auch den Bearbeitungszustand des aktuellen Arbeitsprozesses repräsentiert, können Anwender ihre Zusammenarbeit daran koordinieren. Sie informieren sich über den Stand der Vorgangsbearbeitung, indem sie sehen, wer bisher welche Tätigkeiten erledigt hat und was noch zu erledigen ist. Erledigte Tätigkeiten werden vom Bearbeiter selbst abgehakt.

Das *Prozessmuster* kann von den Benutzern je nach Situation verändert werden, um den aktuellen Gegebenheiten zu entsprechen. Für Routinevorgänge können *Laufzettel* aus einer Sammlung vorgefertigter *Laufzettel* entnommen werden. Zeichnet sich jedoch ein neuer Vorgang ab, kann der *Laufzettel* entsprechend den Anforderungen der konkreten Situation von Benutzern verändert oder angepasst werden. Für neue Routinevorgänge können die Anwender die Sammlung von *Laufzetteln* durch eine neue Vorlage erweitern. Geändert werden können zum Beispiel Zuständigkeiten und zu erledigende Tätigkeiten.

Ein *Laufzettel* ist jeweils mit einer *Vorgangsmappe* verbunden, die Materialien enthält, welche zur Vorgangsbearbeitung benötigt werden. Der *Laufzettel* enthält dazu die vereinbarten Zuständigkeiten, die angeben, an welchen Arbeitsplatz die *Vorgangsmappe* transportiert werden soll. Empfänger müssen eindeutig als Personen oder Rollen identifiziert werden können. Anhand des *Laufzettels* kann

der Empfänger einer *Vorgangsmappe* erkennen, was zu erledigen ist. So sind ausreichend Informationen für die Koordination aus dem *Laufzettel* ersichtlich, ohne die Mappe erst öffnen zu müssen. *Vorgangsmappen* können auf diese Weise auch nachverfolgt werden. Ein Anwender kann zum Beispiel eine Anfrage stellen, an welchem Arbeitsplatz eine bestimmte *Vorgangsmappe* derzeit bearbeitet wird und wo sie bisher war. Eine maschinelle Konsistenzprüfung erfolgt dabei nicht.

Eine interessante Vision des Modells ist die sogenannte *erweiterte Raummetapher*. Sie stellt die gegenseitige Wahrnehmung (*Awareness*) in den Mittelpunkt: Benutzer befinden sich in verschiedenen Räumen und Benutzer innerhalb eines einzelnen Raumes sollen einander wahrnehmen, um sich so in ihrer Arbeit zu koordinieren. Sie haben dadurch die Möglichkeit, explizit miteinander zu kommunizieren und benötigte Arbeitsmaterialien zu lokalisieren. Die *erweiterte Raummetapher* unterstützt dazu das sogenannte *Schlüsselkonzept*, welches Privatsphäre und Öffentlichkeit kennzeichnet: Räume können verschlossen werden und somit in diesen Räumen befindliche Arbeitsgegenstände privat oder öffentlich gemacht werden [Zül98].

4.3.3 Bewertung

Das Einsatzfeld von mobilen Systemen liegt, wie aus der Raum-Zeit Matrix hervorgeht, in erster Linie in der ortsunabhängigen, asynchronen Unterstützung von Arbeitstätigkeiten. Eine Verwendung von mobilen Systemen zur Unterstützung einer Kooperation von Personen, die sich zur gleichen Zeit am selben Ort befinden, ist zwar theoretisch möglich, kann jedoch nicht als eigentliche Aufgabe des *Mobile Computings* angesehen werden. Ebenso ist eine synchrone Kooperation wegen der nicht-permanenten Verbindung mobiler Geräte nicht in allen Fällen realisierbar.

CSCW und *Prozessmuster* beschäftigen sich mit der Gestaltung menschlichen Zusammenarbeitens, welches nicht gleichzeitig auch für die Zusammenarbeit automatisiert ausgeführter Dienste gilt. Problematisch ist vor allem die Sichtweise, bei Prozessteilnehmern und Umgebungsdaten handele es sich um statische Elemente, während die auszuführenden Aktivitäten dynamisch veränderbar seien. Im mobilen Umfeld sind Prozessteilnehmer, sofern nicht eine konkrete Person oder ein konkreter Dienst angesprochen werden soll, hochdynamisch und zur Designzeit des Prozesses normalerweise unbekannt. Die spezifizierten Aktivitäten, die die Vorgehensweise zur Bearbeitung von Aufgaben und damit zur Erreichung eines angestrebten Ergebnisses ausdrücken, haben hingegen nur einen sehr geringen Spielraum für Änderungen. Insbesondere sollen sie in der Regel nicht gelöscht oder in ihrer beabsichtigten Wirkungsweise verändert werden.

Das Konzept des *Prozessmusters* kann das Vorhaben dieser Arbeit, eine auf Middleware-Ebene angesiedelten Prozessbeschreibungssprache für mobile Systeme zu entwickeln, also nur begrenzt unterstützen. Als aktives Konzept zur Koordination von Personen und Softwarekomponenten in einem gemeinsamen Workflow-Management-System ist das Paradigma daher nicht änderungsfrei zu übernehmen.

Workflow-Management-Systeme verfolgen eine ablaufsteuernde Sichtweise, indem sie die vorgeschriebene Reihenfolge und die Art und Weise der Aufgabenerledigung erfassen. Der dahinter liegende Mechanismus ist den Anwendern unzugänglich. Daher stellt ein Workflow-Management-System im Sinne des *WAM*-Ansatzes eigentlich nur einen großen Automaten dar, der aktiv den Handlungsablauf zur Erledigung einer vordefinierten Aufgabe bestimmt [Zül98]. Die Grundidee von Züllighoven [Zül98] und Gryczan [Gry96] ist es jedoch, Mechanismen der Vorgangssteuerung zu vergegenständlichen und den Anwendern als eine Form des Materials für die Bearbeitung zugänglich zu machen, um auch den Prozess der Vorgangsbearbeitung in ihre Verantwortung zu geben.

Das Paradigma aktiver Benutzer, die sich in einem Prozess koordinieren kann zudem nicht ohne weiteres auf die Komposition passiver Dienste übertragen werden. Dienste werden aufgerufen, ohne dass sie an ihrer Koordination selbst beteiligt sind. Sie kennen die Art und Weise ihrer Zusammenarbeit nicht und können daher auch nicht darauf einwirken. Die Formulierung von *Prozessmustern* als Grundlage für eine automatisierte Komposition von Softwarediensten, die möglichst ohne Benutzerinteraktion auf Prozessebene auskommen soll, ist in dieser Form nicht ausreichend. Vielmehr wird eine vollständige Prozessbeschreibung benötigt, die die gesamte Logik des Vorganges implementiert. Eine transparente Visualisierung von Prozessen kann zudem in diesem Szenario nicht gewährleistet werden, da Prozessteilnehmer durch die Dynamik der Umgebung erst zur Laufzeit an Aufgaben gebunden werden können und somit vorher keine Informationen über deren spätere Mitwirkung vorliegen.

Trotzdem steht auch bei dieser Arbeit die Kontrolle des Benutzers über den Prozess im Vordergrund. Der Benutzer einer Anwendung oder eines mobilen Systems ist im Normalfall Begründer und Initiator von Ablaufplänen. Durch die Definition von nicht-funktionalen Anforderungen unterliegt ihm zudem eine qualitative Steuerung des Prozesses, in die auch konkrete Erfahrungswerte einfließen können. Vom Zeitpunkt der Definition und des Starts des Prozesses muss dieser jedoch möglichst ohne Administration von menschlichen Prozessteilnehmern auskommen können. Für eine Einbindung des *Prozessmusters* in den vorliegenden Kontext, müsste dieses also entsprechend auf die Koordination von voneinander unabhängigen Diensten in Form von Automaten unter der Aufsicht eines Benutzers angepasst werden.

Da eine dynamische Änderung eines *Laufzettels* während der Ausführung automatisierter Services nicht möglich ist, weil der initiierende Benutzer in diesem Fall keine direkte Kontrolle mehr auf den Ablauf auswirken kann, müssen allgemeine Regeln und Grundsätze zur Ausführung des Prozesses bereits bei dessen Initiierung festgelegt werden. Dazu gehören insbesondere die benutzerdefinierte Formulierung des Kontrollflusses und aller nicht-funktionaler Aspekte, die Qualitätsmerkmale potentieller Prozessteilnehmer betreffen. Die Kontrolle des Benutzers muss daher in einer Art vorausschauendem „Strategieplan“ definiert werden, der von teilnehmenden Automaten des Prozesses, aber auch von den teilnehmenden menschlichen Benutzern interpretiert und verwirklicht werden kann.

Im Fall einer notwendigen Benutzerinteraktion, wie der oben genannten Fehlerbehandlung, ist das *Prozessmuster* hingegen geeignet, um dem Anwender den Prozess zu vergegenständlichen und ihm dadurch die Entscheidung über mögliche weitere Schritte zu erleichtern. Es kann daher in Erwägung gezogen werden, bei der Einbeziehung eines Benutzers in die Kontrolle über den Prozess einen *Laufzettel* mit Angabe der bereits ausgeführten Aktivitäten sowie die in einer *Vorgangsmappe* enthaltenen Dokumente auszugeben. In einem weiteren Schritt könnte einem autorisierten Benutzer dadurch die Möglichkeit verschafft werden, den *Laufzettel* zu verändern und so den Prozess manipulieren und un-

ter Umständen wieder in einen ausführbaren, fehlerfreien Zustand versetzen zu können. Diese Art von Darstellung und Veränderung laufender Prozesse ist jedoch zu komplex, um sie auf mobilen Systemen mit nur geringer Leistungsfähigkeit zu integrieren. Die Erkenntnisse des präsentierten Ansatzes können jedoch für leistungsfähigere Geräte in einem optionalen Modul für Benutzerinteraktion und Fehlerbehandlung eingebunden werden.

4.4 Ergebnis

Die Analyse bestehender Ansätze zur Prozessbeschreibung hat ergeben, dass keine der betrachteten Sprachen oder Konzepte ausreichend zur Prozessintegration in eine Middleware für mobile Systeme geeignet ist. Einzelne Teilkonzepte sind dennoch geeignet, in einem integrierten Ansatz die definierten Anforderungen zu erfüllen. Es sind jedoch in jedem Fall Änderungen oder Einschränkungen an grundlegenden Schemata nötig, um die Sprachen für einen Einsatz im Umfeld mobiler Systeme anzupassen.

Im Detail hat die Betrachtung gezeigt, dass ein Großteil der allgemeinen Anforderungen an Prozessbeschreibungssprachen von den analysierten Ansätzen erfüllt wird. So unterstützen alle betrachteten Beschreibungssprachen mit Ausnahme von *jPDL* die Komposition von automatisierten Diensten, insbesondere von *Web Services*. Während *XPDL* diese als abstrakte Einheit in die Prozessbeschreibung einbindet, beschreiben *BPEL4WS*, *ebBPSS* und *WSCI* konkrete Muster zum Nachrichtenaustausch. Hierbei steht verstärkt die Modellierung von konkreten Geschäftsbeziehungen im Vordergrund. *jPDL* kann *Web Services* lediglich über ergänzende Hilfskonstrukte in die Prozesse integrieren.

Eine Weitergabe von Prozessen an andere Ausführungseinheiten ist nicht Inhalt der betrachteten Ansätze, da mit Ausnahme von *WSCI* alle Beschreibungssprachen für den Einsatz in einem System mit zentraler Administration entwickelt wurden. Eine zentrale Einheit verwaltet hierbei die Prozessbeschreibung und weist Aktivitäten an Prozessteilnehmer zu. Die Auswahl von Prozessteilnehmern geschieht in der Regel über die Zuordnung von Rollenbeziehungen und kann in der Regel dynamisch erfolgen. Potentielle *WSCI*-Teilnehmer müssen dazu zusätzlich ein entsprechendes Interface zur Teilnahme am Prozess besitzen. Zur Definition von nicht-funktionalen Aspekten bieten *ebBPSS* und *WSCI* erste Ansätze an, die jedoch nur mit Hilfe ergänzender Komponenten verwirklicht werden können.

Weiterhin stehen in allen beschriebenen Sprachen mehr oder weniger umfangreiche Konstrukte zur Beschreibung von Kontroll- und Datenfluss und zur Spezifikation von Prozessteilnehmern zur Verfügung. *XPDL* und *jPDL* definieren auch menschliche Benutzer und andere Softwarekomponenten als Prozessteilnehmer, während *BPEL4WS*, e*bBPSS* und *WSCI* nur auf die Integration von *Web Services* spezialisiert sind. Der Datenfluss und die lokale Datenhaltung wird in erster Linie über Variablen realisiert, welche Werte, Dokumente und Nachrichteninhalte aufnehmen können.

Fehlerbehandlungsmaßnahmen und Transaktionen werden explizit nur durch *BPEL4WS* und *WSCI* unterstützt. *ebBPSS* und *jPDL* verweisen hierzu auf andere Komponenten ihrer Frameworks, während

XPDL entsprechende Konstrukte gänzlich ausläßt. Sicherheitsmechanismen wie Authentifizierung oder Nicht-Abstreitbarkeit von Aktionen werden unter den betrachteten Ansätzen nur von *WSCI* zur Verfügung gestellt. Eine sprachliche Erweiterbarkeit zur Kompensation derartiger Konstrukte ist jedoch zumindest theoretisch bei allen betrachteten Beschreibungssprachen möglich. *ebXML* propagiert Erweiterbarkeit zwar als globales Ziel, es werden jedoch keine konkreten Konzepte oder Vorgehensweisen für mögliche benutzerdefinierte Erweiterungen in *ebBPSS* beschrieben.

Die Skalierbarkeit der betrachteten Beschreibungssprachen lässt sich nur schwer einschätzen. *XPDL*, *BPEL4WS* und *jPDL* scheinen vom Standpunkt der Betrachtung keine Beschränkungen bezüglich einer möglichen Erhöhung von Teilnehmerzahl und Prozessinstanzen aufzuweisen. *ebBPSS* wird im Vergleich dazu als schlecht erweiterbar eingestuft, da mit einer stark zunehmenden Menge an Kollaborationspartnern die *Multi Party Collaboration* unübersichtlich und schwer handhabbar wird. Zumindest im Bereich der mobilen Systeme scheint aufgrund des resultierenden Bedarfs an Speicher- und Rechenkapazität eine Erweiterung auf eine große Anzahl von Teilnehmern nicht ohne weiteres möglich zu sein. *WSCI*-Teilnehmer müssen über integrierte *Interfaces* verfügen, was zumindest eine dynamische Skalierung begrenzt.

Durch die Verwendung von XML-basierten Beschreibungen sind eigentlich alle betrachteten Ansätze plattformunabhängig, sofern am Ort der Ausführung ein geeigneter Interpreter zur Verfügung steht, um die beschriebenen Definitionen auszulesen und zu verarbeiten. *ebBPSS* ist zur Ausführung jedoch zusätzlich auf weitere Komponenten des *ebXML*-Frameworks angewiesen und auch *WSCI* ist durch die Integration der Beschreibung in die WSDL-Datei eines *Web Services* auf diese spezielle Infrastruktur angewiesen. Durch die Einbettung von Java-Klassen in *jPDL* wird am Ort der Ausführung des Prozesses eine Java-Plattform benötigt.

Audit Trails werden mit Ausnahme von *XPDL* von keiner Sprache durch echte Sprachkonstrukte unterstützt. In *jBPM* besteht die Möglichkeit des *Loggings* und *ebBPSS* verwendet zumindest ein Attribut innerhalb der Sprache, um anzugeben, ob die Anfertigung eines *Audit Trails* gewünscht wird.

Die Beanspruchung von Speicher und Rechenkapazität der betrachteten Beschreibungssprachen ist nicht eindeutig zu bewerten, da diese Parameter natürlich stark vom Einsatzzweck der Sprache und dem Umfang der Prozessbeschreibung abhängig sind. Die Beschreibung selbst ist als XML-basierte Textdatei in der Regel nicht besonders speicheraufwendig. Es kann jedoch im Vergleich festgestellt werden, dass bei *BPEL4WS* aus der beschriebenen Schachtelung von graph- und blockstrukturierten Konstrukten teilweise sehr komplexe Beschreibungen resultieren, deren Bearbeitung mehr Ressourcen in Anspruch nimmt als die klaren Kontrollflussstrukturen von *XPDL* oder *WSCI*. Bei *ebBPSS* nimmt zusätzlich die Definition der Kollaborationsbeziehungen einen großen Anteil an der Gesamtbeschreibung ein. *jPDL* ist im Gegensatz dazu zwar klar strukturiert und enthält überschaubare Schemadefinitionen, kann jedoch auch durch die Einbettung von zusätzlicher Funktionslogik in mitgelieferten Java-Klassen zu einem erhöhten Bedürfnis an Speicher- und Rechenkapazität beitragen. *WSCI* scheint zumindest von diesem Standpunkt betrachtet am wenigsten Speicherkapazität zu benötigen, weil durch Verteilung der *Interfaces* auf die Prozessteilnehmer die beteiligten Systeme entlastet werden können.

Der entstehende Kommunikationsaufwand basiert einerseits auf der Größe und der Komplexität der zu übertragenden Prozessbeschreibungen, andererseits auf der Anzahl der nötigen Verbindungsaufnahmen der beteiligten Kommunikationspartner. *BPEL4WS*, *ebBPSS* und *WSCI* scheinen hierbei etwas vorteilhafter zu sein, da der genaue Ablauf der Kommunikation bereits in der Prozessbeschreibung

vorgeschrieben ist und daher nicht noch durch zusätzliches Kommunikationsaufkommen darüber Einigung erzielt werden muss. Zusätzlicher Kommunikationsaufwand entsteht jedoch in jedem Fall durch die Abwicklung und Koordination verteilter Transaktionen, das Versenden von Benachrichtigungen und Fehlernachrichten sowie die Identifikation geeigneter Prozessteilnehmer und die Verhandlung über deren Mitwirken am Prozess. Durch die Übertragung von ganzen Java-Klassen weist *jPDL* in diesem Vergleich die schlechteste Performanz auf.

Die Behandlung von spezielleren Anforderungen mobiler Systeme wie zum Beispiel die Kompensation von Verbindungsabbrüchen oder spezielle Synchronisationskonzepte werden in keiner der betrachteten Sprache unterstützt. Zur Synchronisation wird lediglich das *Join*-Konstrukt zur Verfügung gestellt, Zeitstempel oder weitergehende Maßnahmen zum Management verteilter Ausführungseinheiten werden nicht angeboten. *XPDL* unterstützt unter den betrachteten Ansätzen alleinig die Priorisierung von Aktivitäten. Ergänzende Sicherheitsmaßnahmen wie zum Beispiel die Verschlüsselung von Prozessbeschreibungen werden durch das *ebXML*-Framework realisiert und können in *ebBPSS* als Anforderung formuliert werden.

Tabelle 14 zeigt eine Zusammenfassung aller analysierten Aspekte in Hinsicht auf die definierten Anforderungen. Für die Betrachtung von bestehenden Ansätzen unerhebliche oder unrealisierbare Anforderungen werden in der Tabelle nicht berücksichtigt.

Um auch die direkte Koordination mit Benutzern in die Betrachtung einzubeziehen, wurde das Paradigma des *Prozessmusters* untersucht. Dabei hat sich ergeben, dass *Prozessmuster* allein nicht das geeignete Mittel sind, um Prozesse zur Ausführung auf mobilen Systemen zu beschreiben. Für den Fall, dass menschliche Benutzer nicht nur als Initiator des Prozesses agieren, sondern auch selbst in die Ausführung des Prozesses integriert werden sollen, kann das Konzept des *Prozessmusters* jedoch als Grundlage dienen, um dem Anwender die Kontrolle über das System und den Prozess zu übertragen. Der Ansatz definiert wichtige Rahmenbedingungen, um Prozessbeschreibungen selbst als Material bearbeitbar zu machen und so zum Beispiel im Fall eines unbehandelbaren Fehlers einen autorisierten Benutzer zu Modifikationen zu ermächtigen. Die Verwirklichung dieser Mehrwertfunktion muss jedoch hinter der eigentlichen Aufgabe der Darstellung von Prozessen anstehen und kann aufgrund der resultierende Komplexität von Änderungen der Prozessbeschreibung zur Laufzeit nur von teilnehmenden Systemen mit höherer Leistungsfähigkeit erbracht werden.

Da keiner der betrachteten bestehenden Ansätze ausreichend für die Realisierung einer Prozessbeschreibungssprache zum Einsatz auf mobilen Systemen geeignet ist, muss ein adäquater Kompromiss zwischen den dargestellten Schemata gefunden werden. Dabei ist zu berücksichtigen, dass zwar möglichst alle in 3.4 definierten starken Anforderungen erfüllt werden sollen, durch die Auswahl einer geeigneten Basis aber auch die mögliche spätere Integration von Komponenten zur Erfüllung optionaler Anforderungen nicht unmöglich gemacht werden darf. Zu den geeigneten integrierbaren Teilkonzepten zählen insbesondere die Beschreibung von Transaktionen nach dem in *WSCI* beschriebenen Definitionen, die Einbindung von nicht-funktionalen Aspekten sowie die Spezifikation von Sicherheitsmaßnahmen von *ebBPSS* sowie das Fehlerbehandlungskonzept aus *BPEL4WS*.

In Hinblick auf die genannten Anforderungen und unter Berücksichtigung, dass eine Erweiterung nicht nur einzelner Elemente, sondern ganzer Konzepte notwendig ist, scheint *XPDL* am besten geeignet, um als Grundlage für die Formulierung einer Prozessbeschreibungssprache für mobile Systeme zu dienen. Vorteilhaft ist vor allem, dass durch *XPDL* nicht nur die Integration von *Web Services* ermöglicht wird, sondern auch beliebige Kombinationen von menschlichem Handeln und automa-

tisierten Aktionen unterstützt werden. Die Sprache legt klare, schlanke Strukturen zugrunde, die keine für den mobilen Bereich unnötig komplexen Verschachtelungen enthält. Zudem wurde *XPDL* als Austauschformat entwickelt und bietet somit die Möglichkeit, auch andere Prozessbeschreibungssprachen ausdrücken zu können. Diese Offenheit und die Anerkennung als Standard stellen eine solide Basis für die Kompatibilität zu anderen Ansätzen dar.

Um jedoch alle geforderten Eigenschaften einer Prozessbeschreibungssprache für mobile Systeme zu erfüllen, müssen einige grundlegende Modifikationen und Erweiterungen durchgeführt werden. Nicht unbedingt benötigte Konstrukte wie zum Beispiel Simulationsbeschreibungen sollten entfernt und stattdessen im mobilen Bereich dringend benötigte Definitionen für Fehlerbehandlung und Transaktionen ergänzt werden.

Das folgende Kapitel hat zum Inhalt, die geeigneten Teilkonzepte der einzelnen Sprachen auf Basis von *XPDL* in einem gemeinsamen Ansatz zu vereinen, der für die Integration von Prozessen auf mobilen Systemen geeignet ist.

	Anforderung	XPDL	BPEL4WS	EbBPSS	WSCI	JPdl
D1	Beschreibung und Ausführung komponierter automatisierter Services	+	+	+	+	(-)
D2	Weitergabe von Prozessen an andere Ausführungseinheiten	-	-	-	-	-
D3	Auswahl von Prozessteilnehmern durch benutzerdefinierte nicht-funktionale Kriterien	-	-	(+)	(+)	-
D4	Kompatibilität zu bestehenden Ansätzen					
P1	Kontrollflusskonstrukte	+	+	+	+	+
P2	Datenflussbeschreibung	+	+	+	+	+
P3	Spezifikation von Prozessteilnehmern	+	+	+	+	+
P4	Beschreibung von Fehlerbehandlungsmaßnahmen	-	+	(-)	+	(-)
P5	Dauerhafte Verfügbarkeit des Prozess-Systems					
P6	Skalierbarkeit	+	+	-	(-)	+
P7	Erweiterbarkeit	+	+	(+)	+	+
P8	Sicherheit	-	-	+	-	-
P9	Unterstützung von Transaktionen	-	+	(-)	+	(-)
P10	Plattformunabhängigkeit	+	+	(+)	(+)	(-)
P11	Zentrales System zur Administration					
P12	Audit Trail	+	-	(+)	-	(-)
M1	Geringer Speicherverbrauch	+	+/-	+/-	+	(+)
M2	Geringer Anspruch an Rechenleistung	+	-	-	+	+
M3	Verantwortlicher Umgang mit Energieressourcen					
M4	Lokale Datenhaltung	+	+	+	+	+
M5	Geringer Kommunikationsaufwand	+/-	+	+	+	(-)
M6	Behandlung von Verbindungsabbrüchen	-	-	-	-	-
M7	Synchronisation	(-)	(-)	(-)	(-)	(-)
M8	Priorisierung	+	-	-	-	-
M9	Ergänzende Sicherheitsmechanismen	-	-	+	-	-
M10	Auswahl von variablen Diensten und Prozessteilnehmern erfolgt zur Laufzeit	+	+	+	(+)	+
M11	Benutzerintegration	+	-	-	-	+
M12	Verteilte Administration des Prozesses	-	-	-	+	-
M13	Mächtigkeit der Prozessbeschreibungssprache					

+ Wird durch die Prozessbeschreibungssprache unterstützt

- Wird nicht durch die Prozessbeschreibungssprache unterstützt

+/- Nicht bewertbar

Tabelle 14: Analyse der Prozessbeschreibungssprachen für den Einsatz auf mobilen Systemen

5 Definition einer Prozessbeschreibungssprache für mobile Systeme

Die vorangegangene Analyse bestehender Ansätze zur Prozessbeschreibung hat gezeigt, dass keine der betrachteten Beschreibungssprachen ausreichend geeignet ist, um die definierten Anforderungen für eine Prozessintegration auf mobilen Systemen zu erfüllen. Die analysierten Sprachen bringen jedoch eine Reihe von geeigneten Teilkonzepten mit sich, die zu einer Lösung des Problems im beschriebenen Kontext beitragen können.

In diesem Kapitel wird die Prozessbeschreibungssprache *DEMAC Process Description Language* (im folgenden kurz *DPDL*) definiert, welche speziell auf die vorgestellten Bedürfnisse von mobilen Systemen abgestimmt ist. Im Anschluß an die Spezifikation der Sprache wird ein Prototyp für eine Ausführungsumgebung vorgestellt, um aufzuzeigen, wie mit *DPDL* formulierte Prozesse übertragen, verarbeitet und an andere mobile Geräte weitergegeben werden können. Abschliessend wird erläutert, inwieweit *DPDL* zusammen mit der Ausführungsumgebung die Anforderungen an ein Prozessmanagementsystem für mobile Systeme erfüllt und die Lösung auszugsweise am Anwendungsbeispiel präsentiert.

5.1 DPDL - Demac Process Description Language

In diesem Abschnitt wird die Prozessbeschreibungssprache *DPDL* vorgestellt. Es handelt sich hierbei um eine Sprache in Anlehnung an *XPDL* (vgl. 4.2.1), die speziell an den Einsatz auf einer Middle-

wareplattform für mobile Systeme angepasst ist. Insbesondere stehen bei der Definition von *DPDL* die verteilte Ausführung, die Formulierung von nicht-funktionalen Kriterien zur Auswahl von Diensten und Prozessteilnehmern sowie die besondere Berücksichtigung der begrenzten Leistungsfähigkeit von mobilen Geräten im Vordergrund.

Die Grundidee von *DPDL* ist es, Prozesse nicht allein auf einem einzigen mobilen Gerät zu bearbeiten, sondern diese gemeinsam mit dem aktuellen Ausführungszustand weitergeben zu können, wenn die Leistungsfähigkeit des mobilen Geräts erschöpft ist oder lokal keine Dienste mehr zur Verfügung stehen, um Aufgaben zu erledigen. Abbildung 41 veranschaulicht diese Vorgehensweise. Ist das Ausführungspotential eines Teilnehmers erschöpft, kennzeichnet dieser den aktuellen Bearbeitungszustand des Prozesses und transferiert die Prozessbeschreibung an einen anderen geeigneten Teilnehmer, der gegebenenfalls aufgrund spezifizierter nicht-funktionaler Kriterien ausgewählt wird. Dieser führt die Bearbeitung des Prozesses an der Position im Kontrollfluss fort, an der der Sender der Prozessbeschreibung die Ausführung abgebrochen hat.

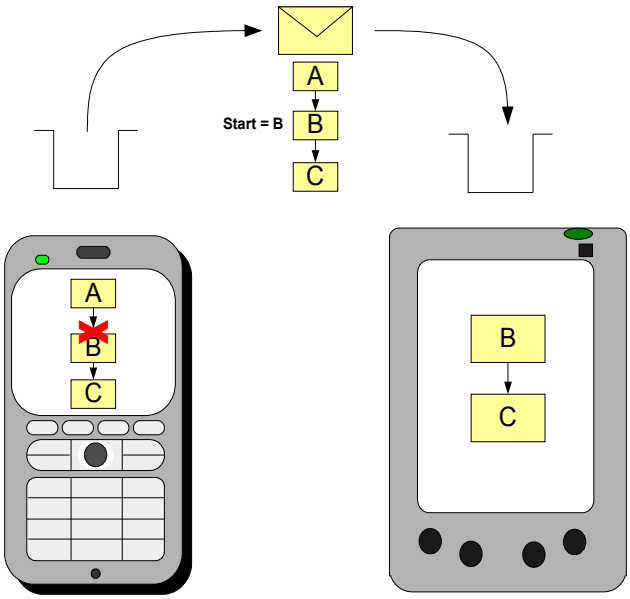


Abb. 41: Verteilte Ausführung mit DPDL

Ebenso wie *XPDL* ist auch *DPDL* als XML-Schema beschrieben, welches die möglichen Konstrukte der Sprache und den genauen Aufbau definiert. Im folgenden werden das Meta-Modell der Sprache, der grundsätzliche Aufbau sowie die speziellen Sprachkonstrukte für den Einsatz im Umfeld mobiler Systeme genauer erläutert. Das gesamte Schema von *DPDL* ist im Anhang aufgeführt (Anhang 2).

5.1.1 Meta-Modell

Das Meta-Modell von *DPDL* beschreibt die elementaren Konstrukte, die dieser Prozessbeschreibungssprache zugrunde liegen. Es lehnt sich an das Meta-Modell von *XPDL* an (vgl. [WfMC02]) und ist zur besseren Übersicht in ein *Package*-Modell und ein Prozess-Modell gegliedert.

Abbildung 42 zeigt das *Package*-Modell von *DPDL* mit besonderer Kennzeichnung der gegenüber *XPDL* neuen Konstrukte. Globaler Container für die Prozessbeschreibung, die benötigten Daten sowie für alle weiteren Ausführungsdetails ist wie in *XPDL* das *Package*. Während *XPDL* jedoch mehrere Prozesse in einem *Package* aggregiert, enthält *DPDL* nur einen einzigen Prozess. Diese Vereinfachung ist zum einen notwendig, um die Prozessbeschreibung möglichst kompakt zu halten. Zum anderen müssen Prozesse im Falle einer nicht durchführbaren Verarbeitung einzeln an andere Teilnehmer weitergegeben werden können. Eine Extraktion einzelner Prozesse aus dem *Package* ist jedoch aufgrund möglicher wiederverwendbarer Bestandteile innerhalb der *Package*-Deklaration nicht ohne weiteres möglich. Vor allem vor dem Hintergrund möglicher Verbindungsabbrüche zum ursprünglichen *Package*-Inhaber ist es einfacher und weniger fehleranfällig, ein ganzes *Package* zu übertragen, als einzelne Details zurückzubehalten. Dieses *Package* enthält nur Daten, die für den aktuell durchgeführten Prozess relevant sind und benötigt daher nur einen geringen Speicherplatz.

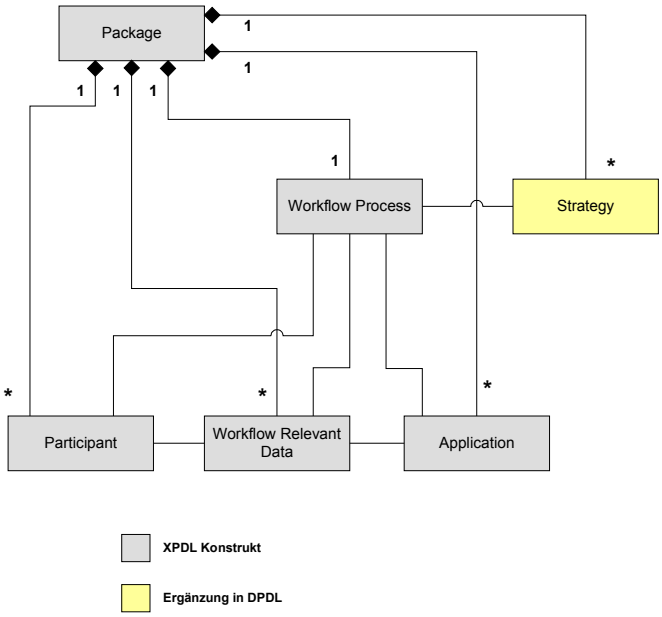


Abb. 42: DPDL-Package-Meta-Modell

Innerhalb eines Prozesses wiederverwendbare Komponenten sind die *Participants*, *Workflow Relevant Data* und *Applications*. *Participants* stellen vorgegebene Prozessteilnehmer dar, die spezifiziert werden, um bestimmte Aufgaben innerhalb des Prozesses auszuführen. Hierbei kann es sich zum Beispiel um Personen, Personengruppen, Geräte oder Geräteklassen handeln. Die Komponente *Workflow Relevant Data* stellt die bei der Ausführung des Prozesses benötigten Daten dar. Darunter fallen zum einen Ein- und Ausgabedaten von Aktivitäten des Prozesses und zum anderen Kontrollflussdaten, die

zum Beispiel zur Evaluation von Transitionsbedingungen verwendet werden. *Applications* bezeichnen wie in *XPDL* die technischen Ausführungseinheiten des Prozesses, jedoch mit Erweiterung des Begriffs auf alle Arten von Diensten und Anwendungen, die über eine beliebige softwaretechnische Schnittstelle angesprochen werden können, um eine Aufgabe des Prozesses zu erfüllen. Eine erneute Definition von *Participants*, *Workflow Relevant Data* und *Applications* auf Ebene des Prozesses wie in *XPDL* entfällt, da diese Deklarationen ohnehin nur für den einzigen Prozess des *Packages* relevant sind und somit auch nicht überschrieben werden müssen.

Zur Formulierung von nicht-funktionalen Aspekten sowohl auf der Ebene des Prozesses als auch für einzelne Aktivitäten wird das Element *Strategy* eingeführt. Eine *Strategy* enthält die nicht-funktionalen Anforderungen des Initiators des Prozesses, die während der Weitergabe der Prozessbeschreibung und dessen Ausführung eingehalten werden müssen. Zum Beispiel kann definiert werden, dass eine verschlüsselte Übertragung der Prozessbeschreibung erforderlich ist. *Strategies* können außerdem auf innerhalb des *Packages* definierte Daten oder *Participants* Bezug nehmen, zum Beispiel, um einen bestimmten Teilnehmer für die Ausführung einer Aufgabe zu verlangen, oder um Kontextdaten mit konkreten Sollwerten des Prozesses abzugleichen.

Die generelle Wiederverwendbarkeit der genannten Komponenten hilft, unnötige Mehrfachdefinitionen zu vermeiden und somit die Prozessbeschreibung möglichst kompakt zu halten.

Die zweite Komponente des Meta-Modells, das Prozess-Modell, enthält den Aufbau eines *DPDL*-Prozesses und die Spezifikation der Kontrollflusskonstrukte (Abb. 43). Ein Prozess enthält beliebig viele *Aktivitäten* und *Transitionen*, die gemeinsam die Aufgaben eines Prozesses und deren Ausführungsreihenfolge festlegen. Eine Aktivität kann dabei entweder eine *atomare Aktivität* sein oder aber als Platzhalter für *Subprozesse* oder für Folgen von Aktivitäten fungieren. Eine festgelegte, wiederverwendbare Folge von Aktivitäten und Transitionen wird als *Activity Set* bezeichnet und kann mit dem Konstrukt *Block Activity* als einzelne Aktivität aufgerufen werden.

Als Ergänzung zu den genannten *XPDL*-Elementen werden in *DPDL* die beiden Blockkonstrukte *Transaction* und *Loop* eingeführt. Bei einer *Transaction* handelt es sich um eine Folge von Aktivitäten, die nur gemeinsam oder gar nicht ausgeführt werden dürfen und die daher bei ihrer Ausführung besondere Unterstützung benötigen. Um den Umweg über komplexe Konstruktionen zur Abdeckung dieses Falls zu vermeiden, können zusammenhängende Aktivitäten durch die Zusammenfassung als *Transaction* gekennzeichnet werden. Ebenso vereinfacht das *Loop*-Konstrukt die Formulierung von Schleifen und vermeidet aufwändige, wiederholte Deklarationen von Kontrollflussmustern. Es hilft außerdem bei der Koordination von verteilt ausgeführten, wiederholbaren Aktivitäten, indem es den aktuellen Bearbeitungszustand und die Bedingungen festlegt, unter denen die Schleife weiter ausgeführt werden soll. Transaktionen und Schleifen können über eine *Transaction Activity* bzw. über eine *Loop Activity* in den normalen Kontrollfluss eingebettet werden.

Um auch einzelne Aktivitäten wiederverwendbar zu machen und ein erweitertes Management von Aktivitätseigenschaften zu erlauben, werden Aktivitäten in *DPDL* über eine *Activity Reference* referenziert. Eine *Activity Reference* definiert eine eindeutige Position im Kontrollfluss des Prozesses und vermittelt so Prozessteilnehmern dessen aktuellen Ausführungszustand. Zudem enthält sie alle Eigenschaften einer Aktivität, die sich auf deren konkrete Position im Prozess bezieht. Zum Beispiel könnte die Priorität einer wiederverwendeten Aktivität in Abhängigkeit ihres jeweiligen Auftretens im Kontrollfluss unterschiedlich sein.

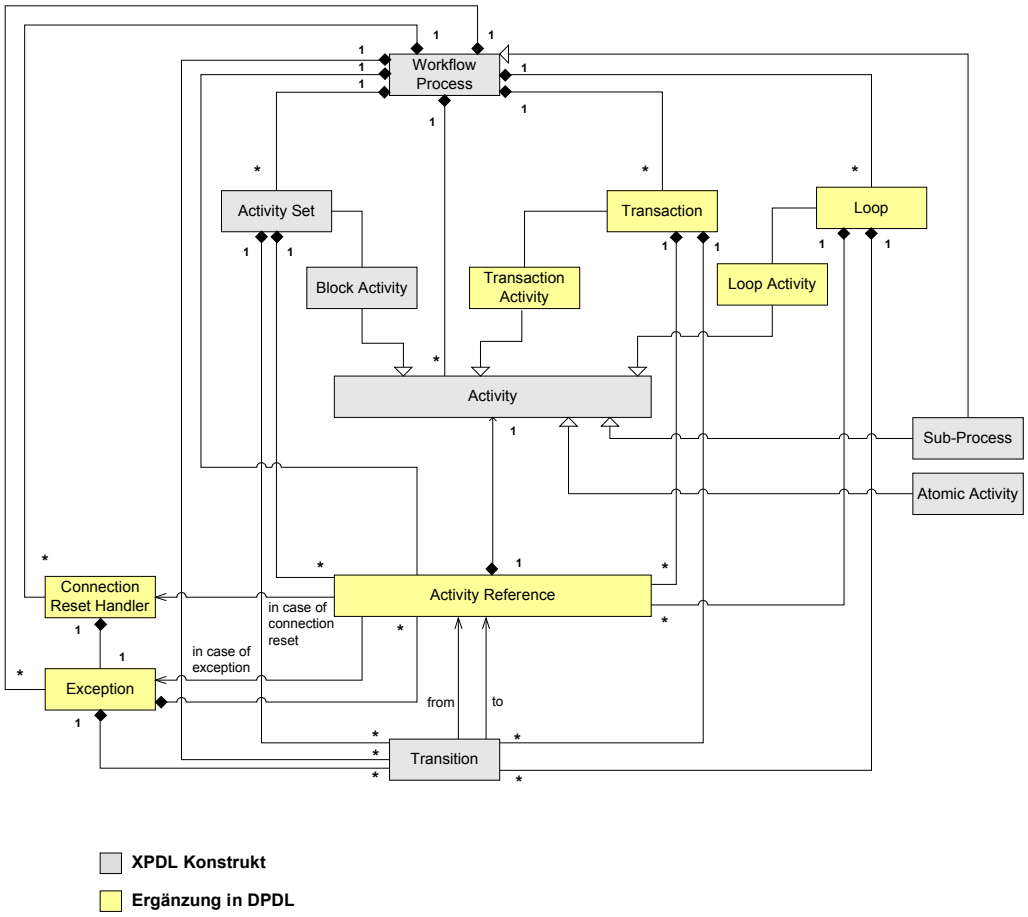


Abb. 43: DPDL Prozess-Meta-Modell

Eine weitere Ergänzung bezieht sich auf die Fehleranfälligkeit von mobilen Systemen und die Gefahr von Verbindungsabbrüchen während der Kommunikation. In *DPDL* besteht die Möglichkeit, Fehlerbehandlungsmaßnahmen mit Hilfe eines *Exception*-Konstrukts explizit zu spezifizieren und die gewünschten Maßnahmen im Falle eines Fehlers sofort aufzurufen und anstelle der gescheiterten Aktivität auszuführen. Fehlerbehandlungsmaßnahmen können dabei wieder aus beliebigen Aktivitäten und Transitionen bestehen und somit auch selbst wieder *Exceptions* beinhalten. Da jedoch unterschieden werden muss, ob der Fehler bei der Ausführung der Aktivität aufgetreten ist oder ob es sich um einen Fehler bei der Kommunikation handelt, können für den Fall eines Verbindungsabbruchs spezielle *Connection Reset Handler* definiert werden, die festlegen, wie im Falle eines Kommunikationsproblems vorgegangen werden soll. Es können zum Beispiel bestimmte Vorgehensweisen wie Wiederholungsversuche oder die Ausführung einer beliebigen *Exception* angegeben werden.

Neben den in Abbildung 43 gezeigten Zusammenhängen können von Seiten der Aktivitäten weitere aus *XPDL* bestehende Abhängigkeiten und Referenzen zu den wiederverwendbaren Komponenten des *Package*-Meta-Modells bestehen. Für Aktivitäten bzw. für *Activity References* können die innerhalb des *Packages* definierten Prozessteilnehmer, Dienste, Strategien und Daten angegeben sein.

Eine genauere Ausführung der einzelnen Komponenten erfolgt in den nachfolgenden Kapiteln.

5.1.2 Aufbau

DPDL ist eine auf oberster Ebene graphstrukturierte Prozessbeschreibungssprache mit der Option zur Blockbildung durch die Zusammenfassung einzelner Aktivitäten. Sie basiert auf XML und verwendet das *Package*-Tag als Wurzelelement des XML-Dokuments. Dieses enthält neben den fachlichen Prozessdaten auch Hinweise auf die zugrunde liegende *DPDL*-Spezifikation und die entsprechenden XML-Schemata. Codebeispiel 27 zeigt die Gliederung des Aufbaus eines *Packages* als Überblick.

```
<Package
xmlns="http://vsis-www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://vsis-
www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd"

Id="Test1"
Name="TestPackage"
Owner="Sonja Zaplata"
DPDLVersion="1.0">

   <Strategies> ... </Strategies>
   <TypeDeclarations> ... </TypeDeclarations>
   <Participants> ... </Participants>
   <Applications> ... </Applications>
   <DataFields> ... </DataFields>
   <WorkflowProcess> ... </WorkflowProcess>

</Package>
```

Codebeispiel 27: DPDL Package

Obligatorische Bestandteile eines DPDL-*Packages* sind sein Identifikator *Id* und das Element *WorkflowProcess*, welches die Beschreibung des Kontrollflusses beinhaltet. Die *Id* eines *Packages* bezeichnet dabei die konkrete Instanz der Prozessbeschreibung und kann somit auch zur Korrelation verwendet werden. Optional können ein Name für das *Package*, die verwendete DPDL-Version und der Eigentümer bzw. Initiator der Prozessbeschreibung angegeben werden (*Owner*). Hierbei kann es sich zum Beispiel um den Namen oder um eine digitale Identität derjenigen Person handeln, die für die Instantiierung des Prozesses verantwortlich ist und der eventuell bei dessen Ausführung entstehende Kosten zugerechnet werden müssen.

Sofern nicht-funktionale Anforderungen spezifiziert werden sollen, können diese im Element *Strategies* beschrieben werden. Das Element *TypeDeclarations* enthält Definitionen für komplexe Datentypen, die im Datenmodell des Prozesses verwendet werden sollen. Diese im Meta-Modell als *Workflow Relevant Data* bezeichneten Daten werden hier als konkrete *DataFields* deklariert. Der Aufbau und die Semantik von *TypeDeclarations* und *DataFields* entspricht der Spezifikation von *XPDL* (vgl. [WfMC02]). Die Elemente *Participants* und *Applications* stellen die bereits im Meta-Modell erläuterten Prozessteilnehmer und Ausführungseinheiten dar und werden im folgenden Abschnitt detaillierter betrachtet.

Auch die Elemente des *WorkflowProcess* wurden weitestgehend auf die Bedürfnisse für den Einsatz im mobilen Bereich angepasst. Codebeispiel 28 zeigt den möglichen Aufbau eines Prozesses nach dem *DPDL*-Meta-Modell.

```
<WorkflowProcess Id="1" Name="TestProcess" AccessLevel="PUBLIC" StartActivity="1"
InitActivity="1“ StrategyId="s1" LoggingRequired="true">

   <ProcessHeader> ... </ProcessHeader>
   <FormalParameters> ... </Formal Parameters>
   <Exceptions> ... </Exceptions>
   <ConnectionResetHandlers> ... </ConnectionResetHandlers>
   <Transactions> ... </Transactions>
   <ActivitySets> ... </ActivitySets>
   <Loops> ... </Loops>
   <Activities> ... </Activities>
   <ActivityReferences> ... </ActivityReferences>
   <Transitions> ... </Transitions>

</WorkflowProcess>
```

Codebeispiel 28: DPDL Workflow Process

Während zentral verwaltete Prozesse lediglich eindeutig referenziert werden müssen (*Id*), benötigt ein *DPDL*-Prozess zur Unterstützung seiner verteilten Ausführung zusätzlich eine *StartActivity*, welche die aktuell zu bearbeitende Aktivität des Prozesses angibt. Dieses erspart den mobilen Geräten eine rechenaufwändige Traversierung des Prozesses nach der ersten auszuführenden Aktivität und vereinfacht die Weiterführung des Prozesses durch andere Prozessteilnehmer, welche bereits bearbeitete Aktivitäten nicht mehr betrachten müssen. Handelt es sich bei dem Prozess um eine mehrfach verwendbare Ausführungsfolge, wie es zum Beispiel bei einem Subprozess der Fall sein könnte, kann die *StartActivity* durch eine *InitActivity* ergänzt werden, die die Anfangsposition des Prozesses über den Ausführungsverlauf hinaus festhält. Diese ermöglicht ein erneutes Starten des Prozesses von seiner Anfangsaktivität. Unmittelbar nach der Instantiierung eines Prozesses stimmen *StartActivity* und *InitActivity* somit immer überein. Sowohl *StartActivity* als auch *InitActivity* verweisen auf eine im Prozess eindeutige *ActivityReference*.

Zur Bindung von nicht-funktionalen Aspekten an einen DPDL-Prozess kann dieser auf eine im *Package* definierte *Strategy* verweisen. Dieses geschieht, wie in Codebeispiel 28 gezeigt, über das Attribut *StrategyId*, welches die *Id* einer im *Package* genannten *Strategy* referenziert. Die somit an den Prozess gebundenen, nicht-funktionalen Anforderungen sind für die Auswahl potentieller Prozessteilnehmer zur Übertragung der Prozessbeschreibung maßgeblich (vgl. 5.1.3).

Das Attribut *LoggingRequired* beschreibt, ob der Prozess nach seiner Bearbeitung vom System gelöscht werden kann, oder ob die Prozessbeschreibung, Zwischenzustände und gegebenenfalls weitere Daten wie Zeit, Teilnehmer oder Kosten zu Abrechnungs- oder Analysezwecken gespeichert werden sollen. Ist die Option *LoggingRequired* gesetzt, können nur diejenigen mobilen Geräte an der Ausführung des Prozesses teilnehmen, die über genügend freien Speicherplatz und eine entsprechende Persistenzfunktion zum Speichern des *Logs* verfügen. Der Default-Wert für *LoggingRequired* ist „false“.

Für den Fall, dass der Prozess einen Subprozess darstellt, kann sein Ausführungsverhalten durch das Attribut *AccessLevel* präzisiert werden. Dazu stehen die beiden *Access Level public* und *private* zur Verfügung. *Public* bedeutet, dass der Prozess als eigenständiger Prozess gestartet und ausgeführt werden kann. Mit *private* gekennzeichnete Prozesse dürfen nur als Subprozess aus einer *SubFlow*-Aktivität eines anderen Prozesses aufgerufen werden. Der *DPDL*-Default-Wert für *AccessLevel* ist *public*.

Weitere Eigenschaften des Prozesses werden im Element *ProcessHeader* zusammengefasst. Der *ProcessHeader* enthält eine kurze Beschreibung des Prozesses, dessen Priorität sowie einen Gültigkeitsbereich. Der Gültigkeitsbereich definiert ein *ValidFrom*-Datum, von dem an der Prozess ausgeführt werden darf, und ein *ValidTo*-Datum, welches beschreibt, zu welchem Zeitpunkt der Prozess ungültig wird und daher nicht mehr bearbeitet werden darf. Alle im Prozess genannten Zeit- und Datumsangaben verwenden das *ISO 8601 Format CCYY-MMDDThh:mm:ss* unter Berücksichtigung der Zeitzone. Um etwa den Zeitpunkt von 1:20 pm am 12. Juni 2005 auszudrücken, die drei Stunden hinter der koordinierten Weltzeit (UTC) liegt, gilt folgende Angabe: 2006-06-12T13:20:00-03:00. Dieses Zeitformat entspricht im übrigen dem XML-Schema-Datentyp *xsd:dateTime*. Eine einheitliche Darstellung der Zeit unter Berücksichtigung der Zeitzone ist für mobile Systeme besonders wichtig, da mobile Geräte während der Ausführung eines Prozesses Zeitzonen überqueren oder mit Teilnehmern aus anderen Zeitzonen kommunizieren können.

Das Element *FormalParameters* enthält die Definition von Ein- und Ausgabeparametern, die an den Prozess gebunden werden sollen (vgl. [WfMC02]). Dies ist insbesondere für Subprozesse relevant, die Daten ihres Elternprozesses übernehmen sollen, um damit weiter zu arbeiten, oder die Daten an ihren Elternprozess zurückgeben müssen. Eine Beschreibung der *FormalParameters* erfolgt in Kapitel 5.1.5.

Neben den genannten Elementen kann die Definition des *WorkflowProcess* die im Meta-Modell vorgestellten Elemente *Exceptions*, *ConnectionResetHandlers*, *Transactions*, *ActivitySets* und *Loops* enthalten. Diese basieren auf einer Blockbildung von Transitionen und Aktivitäten und deren Referenzierung über *ActivityReferences*. Im folgenden Codebeispiel (Codebeispiel 29) wird zunächst einmal ein einfacher Fall ohne Blockbildung beschrieben.

```
<WorkflowProcess Id="1" StartActivity="1" >

  <Activities>
    <Activity Id="TestActivity1"> ... </Activity>
    <Activity Id="TestActivity2"> ... </Activity>
  </Activities>

  <ActivityReferences>
    <ActivityRef Id="1" ActivityId="TestActivity1" State="INACTIVE" ... />
    <ActivityRef Id="2" ActivityId="TestActivity2" State="INACTIVE" ... />
    <ActivityRef Id="3" ActivityId="TestActivity1" State="INACTIVE" ... />
  </ActivityReferences>

  <Transitions>
    <Transition Id="1" From="1" To="2" />
    <Transition Id="2" From="2" To="3" />
  </Transitions>

</WorkflowProcess>
```

Codebeispiel 29: DPDL Activity References

Wie bereits beschrieben bezeichnet eine *ActivityRef* eine eindeutige Position im Kontrollfluss des Prozesses. Sie referenziert eine *Activity* und erlaubt so deren Wiederverwendbarkeit im Prozess. Zudem stellt sie ein wichtiges Konstrukt zur Kennzeichnung des Ausführungszustandes einer Aktivität dar. Das Attribut *State* einer *ActivityRef* gibt an, ob die Aktivität in einem Zustand ist, in dem sie ausgeführt werden kann, oder ob Probleme vorliegen, die eventuell durch eine andere Ausführungseinheit oder einen menschlichen Benutzer gelöst werden müssen.

Transitionen außerhalb von Blockkonstrukten sind nicht wiederverwendbar. Sie setzen sich aus einer *From-ActivityReference* und einer *To-ActivityReference* zusammen und besitzen gegebenenfalls eine Transitionsbedingung (*Condition*), die den Übergang von einer Aktivität zur nächsten von der Evaluierung eines Ausdrucks abhängig macht.

5.1.3 Sprachelemente für den speziellen Einsatz im Bereich mobiler Systeme

Der Einsatz von *DPDL* als Prozessbeschreibungssprache für mobile Systeme macht einige besondere zusätzliche Sprachelemente notwendig. Hierzu zählen in erster Linie das Zustandskonzept, welches erlaubt, Prozesse während ihrer Ausführung an andere Teilnehmer weiterzugeben, die erweiterten Fehlerbehandlungsmaßnahmen unter Berücksichtigung der im mobilen Umfeld auftretenden Verbindungsabbrüche sowie spezielle Konstrukte zur Synchronisation parallel ausgeführter Aktivitäten. Zudem sind besonders beschreibungs- und rechenaufwändige Kontrollflussmuster zu eigenen Konstrukten zusammengefasst, die eine kürzere und eindeutigere Beschreibung des Kontrollflusses zulassen, um die leistungsarmen mobilen Geräte entlasten zu können.

Zustandskonzept

Eine besondere Erweiterung gegenüber klassischen Prozessbeschreibungssprachen ist das in DPDL verwendete Zustandskonzept, um den Ausführungszustand eines Prozesses und die Zustände seiner einzelnen Aktivitäten unter den verteilten Prozessteilnehmern zu kommunizieren. Dazu wird der von Leymann und Roller beschriebene *Activity Lifecycle* (vgl. [LeRo02]) in vereinfachter Form genutzt, um die verschiedenen Zustände, die während der Existenz einer Aktivität auftreten können, in die Prozessbeschreibungssprache zu integrieren. Abbildung 44 zeigt die von DPDL verwendeten Zustände in einer Übersicht.

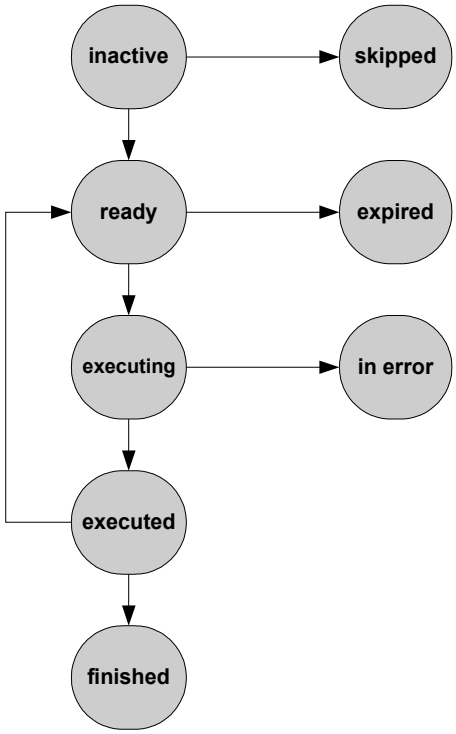


Abb. 44: DPDL Activity Lifecycle (nach [LeRo00])

Solange die Bearbeitung des Prozesses noch nicht gestartet wurde oder die Abwicklung des Kontrollflusses die betreffende Aktivität noch nicht erreicht hat, ist diese mit dem Zustand *inactive* gekennzeichnet. Dieser Zustand kann nur verlassen werden, wenn die unmittelbar vorgelagerte Aktivität entweder erfüllt oder verworfen wurde oder wenn es sich um eine Startaktivität handelt. Sind alle Voraussetzungen für die Ausführung der Aktivität gegeben, wird der Zustand der Aktivität auf *ready* gesetzt. Für eine manuell auszuführende Aktivität kann ein autorisierter menschlicher Benutzer zur Aktivierung der Aktivität erforderlich sein, ansonsten kann die Ausführungsumgebung die Aktivität automatisch in diesen Zustand versetzen.

Solange die Aktivität ausgeführt wird, befindet sie sich im Zustand *executing*. Dies ist von besonderer Relevanz, wenn mehrere Aktivitäten parallel ausgeführt werden, um gegenüber anderen teilnehmenden Ausführungseinheiten zu kennzeichnen, welche Aktivitäten schon ausgeführt werden und welche noch zu bearbeiten sind. Eine mit *executing* gekennzeichnete Aktivität darf somit von anderen Prozessteilnehmern nicht mehr begonnen werden.

Wurde die Ausführung einer Aktivität abgeschlossen, so wird der Zustand einer Aktivität auf *executed* gesetzt. Dies bedeutet im Detail, dass die Aktivität einmal erfolgreich ausgeführt wurde und bei entsprechenden Kontrollflussmustern auch noch ein weiteres Mal ausgeführt werden darf, zum Beispiel im Rahmen einer Schleife. Handelt es sich jedoch um eine nur einmalig auszuführende Aktivität oder ist die wiederholte Abarbeitung beendet, wird der Zustand *finished* gesetzt. Eine so beendete Aktivität ist für die Lebensdauer des Prozesses endgültig abgeschlossen und darf nicht mehr wiederholt werden.

Neben diesem normalen Kontrollfluss können im Lebenszyklus einer Aktivität auch Abbruchsituationen oder Fehler auftreten. Für den Fall, dass Aktivitäten gar nicht bearbeitet werden sollen, können

diese mit dem Zustand *skipped* markiert werden. Im Wesentlichen tritt dieser Zustand als Folge einer *Dead Path Elimination* (vgl. 3.2.5) auf: Wird die Ausführung einer Sequenz von Aktivitäten aufgrund einer Transitionsbedingung abgelehnt, müssen alle der Bedingung folgenden Aktivitäten als nicht ausführbar gekennzeichnet werden, um eine möglicherweise im Kontrollfluss nachgelagerte Bedingung auswerten zu können und Verklemmungssituationen zu vermeiden. Eine weitere Ursache für den Abbruch einer Aktivität stellt der Ablauf ihrer Gültigkeit dar. Sie wird in den Zustand *expired* versetzt, wenn eine spezifizierte *Deadline* für ihre Ausführung überschritten wird. Wenn bei der eigentlichen Ausführung der Aktivität Fehler auftreten, so dass die Ausführung aus einem beliebigen Grund scheitert, wird dies durch den Zustand *inError* angezeigt.

Um die Zuordnung von Zuständen zu Aktivitäten im Prozess eindeutig zu machen, werden die Zustände an die *ActivityReferences* gebunden. Dies erleichtert die potentielle Wiederverwendbarkeit von Aktivitäten, was sich insbesondere bei komplexen Beschreibungen von Aktivitäten auszahlt. Der Ausführungszustand einer referenzierten Aktivität ist somit eindeutig bestimmt und die Ausführung des Prozesses weniger fehleranfällig.

Behandlung von Fehlern und Verbindungsabbrüchen

Da mobile Systeme im Gegensatz zu stationären Systemen inhärent fehleranfällig sind, ist die Behandlung von Ausnahmen besonders relevant, um die Ausführbarkeit von Prozessen zu verbessern. Unzulänglichkeiten, die von Seiten der Ausführungsumgebung auftreten, können in *DPDL* generell durch die Weitergabe der Prozessbeschreibung an einen anderen Teilnehmer ausgeglichen werden. Ist ein mobiles System zum Beispiel nicht in der Lage, eine Transaktion durchzuführen, so kann es den Prozess an einen kompetenteren Teilnehmer abgeben und damit eine Fehlersituation, in der der Prozess aufgrund der mangelnden Leistungsfähigkeit eines mobilen Gerätes vorzeitig beendet werden würde, umgehen.

Für den Fall, dass während der Ausführung einer Aktivität dennoch Fehler auftreten, können in *DPDL* Fehlerbehandlungsmaßnahmen definiert werden, die Alternativen zur ursprünglich geplanten Aktivität angeben. Codebeispiel 30 zeigt die Definition einer *Exception* und die Bindung an eine Aktivität. Tritt nun bei der Ausführung der Aktivität ein Fehler auf, wird die fehlerbehandelnde Maßnahme hierzu ausgeführt. Um dabei möglichst flexibel zu bleiben, besteht eine Fehlerbehandlungsmaßnahme wieder aus beliebigen Aktivitäten und Transitionen. Somit ist zum Beispiel auch eine Wiederholung der gescheiterten Aktivität möglich, indem diese innerhalb der *Exception* neu referenziert wird.

```
<Exceptions>
   <Exception Id="E1" Name="Exception1" StartActivity="100" InitActivity="100"
Execution="ASYNCHR">
      <ActivityRefs>
         <ActivityRef Id="100" ActivityId="Ex1" State="INACTIVE"/>
      </ActivityRefs>
   </Exception>
</Exceptions>
```

```
<Activities>
   <Activity Id="Activity1"> ... </Activity>
   <Activity Id="Ex1"> ... </Activity>
</Activities>


<ActivityReferences>
   <ActivityRef Id="1" ActivityId="Activity1" ExceptionId="E1" State="INACTIVE“ ... />
</ActivityReferences>
```

Codebeispiel 30: DPDL Exception innerhalb einer Aktivität

Das Attribut *Execution* gibt an, wie sich der Kontrollfluss nach der Behandlung des Fehlers weiter verhalten soll. So kann die Fehlerbehandlung ausgeführt werden und danach der normale Kontrollfluss wieder so aufgenommen werden, als wäre die Aktivität fehlerfrei erfüllt worden (*Execution = SYNCHR*). Im Gegensatz dazu kann der normale Kontrollfluss gänzlich durch die *Exception* ersetzt werden (*Execution = ASYNCHR*). In diesem Fall wird die Fehlerbehandlung durchgeführt und der Prozess anschließend beendet.

Um die Kompatibilität zu *XPDL* soweit wie möglich zu erhalten und um zusätzliche Flexibilität zu bieten, können *Exceptions* auch als Transitionsbedingung in den Kontrollfluss eingegliedert werden. Dafür ist es erforderlich, eine Transitionsbedingung vom Typ *Exception* zu definieren und ihr den Identifikator der auszuführenden Fehlerbehandlungsmaßnahme beizufügen (Codebeispiel 31). Ist nun im Verlauf des Kontrollflusses ein Fehler aufgetreten, so wird die betreffende Transitionsbedingung zu „true“ evaluiert und die Fehlerbehandlungsmaßnahme ausgeführt.

```
<Exceptions>
   <Exception Id="E1" Name="Exception1" StartActivity="100" InitActivity="100“
                  Execution="ASYNCHR">
      <ActivityRefs>
         <ActivityRef Id="100" ActivityId="Ex1" State="INACTIVE"/>
      </ActivityRefs>
   </Exception>
</Exceptions>


<ActivityRefs>
   <ActivityRef Id="1" ActivityId="Activity1" State="INACTIVE"/>
   <ActivityRef Id="2" ActivityId="Activity2" State="INACTIVE"/>
</ActivityRefs>

<Transitions>
   <Transition Id="T1" From="1" To="2">
      <Condition Type="EXCEPTION">E1</Condition>
   </Transition>
</Transitions>
```

Codebeispiel 31: DPDL Exception als Transitionsbedingung

Ist keine *Exception* für eine Aktivität definiert, so scheitert diese und alle der Aktivität im Kontrollflusspfad nachgelagerten Aktivitäten können ebenfalls nicht mehr ausgeführt werden.

Eine besondere Art von Fehlerbehandlungsmaßnahmen, die speziell auf die Behandlung von Verbindungsabbrüchen ausgelegt ist, ist ein *Connection Reset Handler*. Die Unterscheidung zwischen echten Fehlern bei der Ausführung einer Aktivität und einem Kommunikationsproblem mit der ausführenden Einheit ist wichtig, um angemessen auf die Ausnahmesituation reagieren zu können. Zum Beispiel könnte eine fehlerhafte Aktivität einem Benutzer zur Behebung des Fehlers übertragen werden, während im Fall eines Verbindungsabbruchs ein anderer Dienst zur Ausführung der Aktivität angefragt werden könnte. Codebeispiel 32 zeigt den Einsatz eines *Connection Reset Handlers* sowie seine möglichen Elemente.

```
<ConnectionResetHandlers>
   <ConnectionResetHandler Id="1">
      <ExceptionId>E1</ExceptionId>
      <Notification>Schlechte Verbindungsqualitaet! Bitte aendern Sie Ihren
            Standort!</Notification>
      <Retries>2</Retries>
      <NewSearch>true</NewSearch>
   </ConnectionResetHandler>
</ConnectionResetHandlers>

<ActivityReferences>
   <ActivityRef Id="1" ActivityId="Activity1" ConnectionResetHandlerId="1"  ... />
</ActivityReferences>
```

Codebeispiel 32: DPDL Connection Reset Handler

Ein *Connection Reset Handler* wird ebenso an eine *Activity Reference* gebunden wie eine *Exception*. Kommt es bei Ausführung der Aktivität zu einem Verbindungsabbruch, so wird der angegebene *Connection Reset Handler* aufgerufen. Neben einer eigenen, für diesen Fall definierten *Exception* können die Anzahl der Wiederholungsversuche zur Wiederaufnahme der Kommunikation, die Suche nach einem alternativen Prozessteilnehmer oder eine Meldung an den Benutzer des mobilen Gerätes definiert werden. Die angegebenen Konstrukte können außerdem sinnvoll miteinander kombiniert werden. Beispielsweise könnte nach zwei erfolglosen Versuchen den Verbindungspartner zu erreichen, die Suche nach einem alternativen Dienst gefordert werden.

Parallele Ausführung von Aktivitäten

DPDL bietet die Möglichkeit, einzelne Sequenzen von Aktivitäten parallel auszuführen und die parallelen Kontrollflusspfade wieder zu synchronisieren. Dazu stehen die beiden Konstrukte *Split* und *Join* zur Verfügung, die in den Varianten *AND* und *XOR* angewendet werden können (vgl. [WfMC02]).

Solange zwischen parallel auszuführenden Aktivitäten keine Datenabhängigkeiten bestehen, ist eine Aufteilung des Kontrollflusses in zwei parallele Pfade auch im mobilen Fall relativ problemlos möglich. Das mobile Gerät, welches die Prozessbeschreibung zur Zeit des *Splits* verwaltet, evaluiert etwaige Bedingungen und entscheidet somit über die Ausführbarkeit der parallelen Pfade. Kann mehr als nur ein Pfad ausgeführt werden, so kann die Prozessbeschreibung in Kopie an jeweils einen anderen Prozessteilnehmer weitergegeben werden, um die parallelen Aufgaben arbeitsteilig zu erledigen. Dazu ist es notwendig, die Aktivität, die bereits vom verantwortlichen Teilnehmer ausgeführt wird, als *executing* zu kennzeichnen, um einen mehrfachen Zugriff hierauf zu vermeiden.

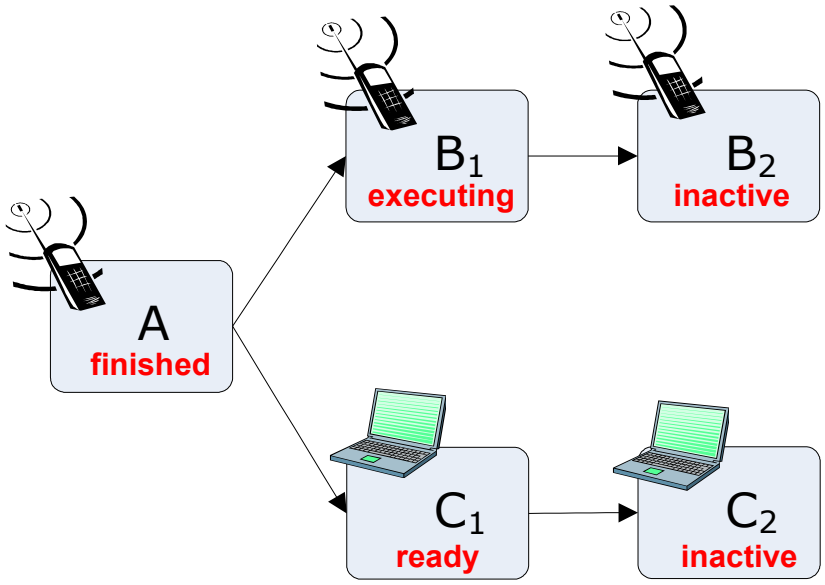


Abb. 45: DPDL Split

Abbildung 45 verdeutlicht diesen Zusammenhang mit Hilfe eines Beispiels. Ein Mobiltelefon bearbeitet einen Prozess, der zwei parallele Pfade beinhaltet. Es führt Aktivität A aus und evaluiert, dass beide parallelen Pfade bearbeitet werden sollen. Nun kann es entweder alle Aktivitäten selbsttätig sequentiell ausführen oder den Prozess an ein anderes mobiles Gerät weitergeben, um die Aufgaben parallel zu erledigen. Das Mobiltelefon entscheidet sich dafür, den Pfad B selbst auszuführen und setzt die erste Aktivität B_1 des Pfades auf den Zustand *executing*. Dann übergibt es eine Kopie des Prozesses an ein anderes mobiles Gerät, um ihn bei der Bearbeitung des anderen Pfades zu unterstützen.

Eine Aktivität kann nur in den Zustand *ready* versetzt werden, wenn seine unmittelbar vorhergehende Aktivität den Zustand *finished* oder *executed* erreicht hat oder es sich um eine Startaktivität handelt. Daher kann das zweite teilnehmende Gerät nur die Aktivität C_1 starten und den Pfad C bearbeiten.

Wenn die beiden parallelen Pfade voneinander getrennt Endzustände des Prozesses erreichen, können die Teilnehmer der parallelen Ausführung den Prozess auf ihrer Kopie der Prozessbeschreibung wie im normalen, nicht parallelen Fall bis zum Ende bearbeiten und auch an beliebige weitere Prozessteilnehmer weitergeben. Durch das Setzen einer Startaktivität kann der jeweils zu bearbeitende Pfad auch bei einer Weitergabe an andere Geräte eindeutig beibehalten werden. Sollen die beiden parallelen Pfade jedoch an späterer Stelle im Kontrollfluss synchronisiert werden, so muss in der Prozess-

beschreibung an der Position des *Joins* ein eindeutiger Treffpunkt in Form eines URI (Uniform Resource Identifier) festgehalten werden. Hierbei kann es sich zum Beispiel um ein im voraus festgelegtes mobiles oder stationäres Gerät handeln, welches eine zur Synchronisation geeignete Anwendung besitzt. Dieses Gerät erwartet nach der ersten eingehenden Prozessbeschreibung alle im *Join* spezifizierten eingehenden Transitionen mit der *Id* des *Joins* und evaluiert etwaige *Join*-Bedingungen. Nach der Zusammenführung der parallelen Pfade kann es den Prozess wieder an beliebige weitere Teilnehmer weitergeben.

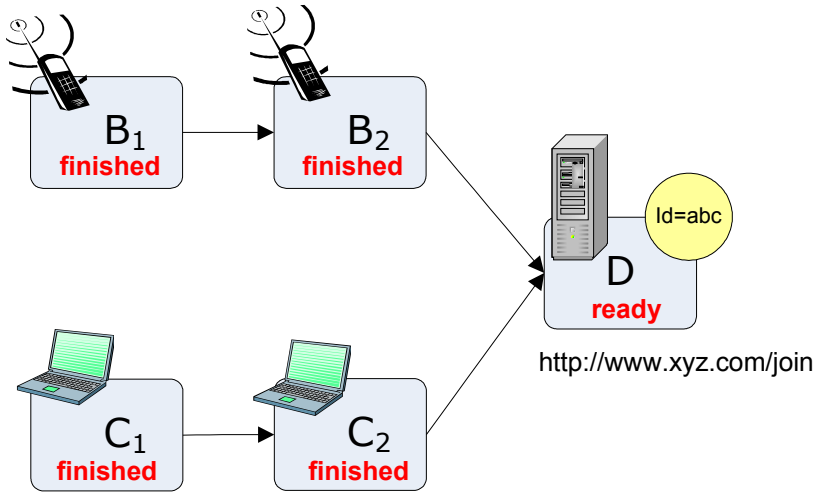


Abb. 46: DPDL Join

Abbildung 46 illustriert das fortgeführte Beispiel. Nachdem das Mobiltelefon die Aktivitäten auf Pfad B ausgeführt hat, überträgt es die Prozessbeschreibung an den durch den URI des *Joins* spezifizierten Prozessteilnehmer. Dieser kann durch die Identifikation des Prozesses und der *Id* des *Joins* eindeutig festlegen, welche weiteren Transitionen in die Synchronisation einzubeziehen sind und ob der Eingang weiterer Kopien der Prozessbeschreibung zu erwarten ist. Sobald auch die Ausführungseinheit des Pfades C ihre Arbeit beendet hat, sendet sie ihre Kopie der Prozessbeschreibung an den zur Synchronisation bestimmten Teilnehmer.

Wenn die Ergebnisse aller benötigten eingehenden Transitionen vorliegen, können etwaige Synchronisationsbedingungen ausgewertet werden und gegebenenfalls veränderte Daten abgeglichen und angepasst werden. Der Prozess kann anschließend sequentiell fortgeführt oder beendet werden.

Die Bearbeitung eines Subprozesses, der synchron zum Elternprozess ausgeführt werden soll, geschieht nach dem gleichen Muster wie ein *Split* mit anschließendem *Join*. Auch hier muss über einen URI ein eindeutiger Prozessteilnehmer zur Synchronisation des Prozesses und zur Übernahme der eingehenden Prozessdaten spezifiziert werden.

Durchführung von Schleifen

Um das erläuterte Zustandskonzept mit wiederholbaren Aktivitäten vereinen zu können, ist es notwendig, beschreiben zu können, ob eine Aktivität potentiell erneut ausführbar sein soll oder nicht. Das *Loop*-Konstrukt definiert dazu einen Kontrollfluss, der in Form einer Schleife ausgeführt werden kann. Dabei handelt es sich um eine Blockaktivität, die die wiederholbaren Aktivitäten im Körper des Blocks beinhaltet und diese ausführt, solange die Schleifenbedingung zu *true* evaluiert wird. Die Schleifenbedingung ist dabei als Anfangsbedingung für die Schleife spezifiziert, was bedeutet, dass die Schleife nicht ausgeführt wird, wenn die Schleifenbedingung bereits zu Beginn der Schleife nicht erfüllt wird. Dies erspart eine zusätzliche Eintrittsbedingung für die Schleife.

```
<Loops>
  <Loop Id="Loop1" StartActivity="100" InitActivity="100">
    <ActivityRefs>
      <ActivityRef Id="100" ActivityId="Activity2" State="INACTIVE"/>
      <ActivityRef Id="200" ActivityId="Activity3" State="INACTIVE"/>
    </ActivityRefs>
    <Transitions>
      <Transition Id="T1" From="100" To="200"/>
    </Transitions>
    <Condition Type="CONDITION"> ... </Condition>
  </Loop>
</Loops>

<Activities>
  <Activity Id="Activity1">
    <LoopActivity LoopId="Loop1"/>
  </Activity>
  <Activity Id="Activity2"> ... </Activity>
  <Activity Id="Activity3"> ... </Activity>
</Activities>

<ActivityReferences>
  <ActivityRef Id="1" ActivityId="Activity1" ... />
</ActivityReferences>
```

Codebeispiel 33: DPDL Loop

Codebeispiel 33 stellt eine einfache Schleife mit zwei wiederholbaren Aktivitäten dar. Sie werden ausgeführt, wenn die *Condition* den Wahrheitswert „true“ erhält. Aufgerufen wird die Schleife als Block innerhalb von „Activity1“.

Neben Schleifen können auch *ActivitySets*, *Transactions* und *Exceptions* potentiell ein weiteres Mal ausgeführt werden. Im Gegensatz zu einer Schleife handelt es sich hierbei aber nicht um eine sequentielle Wiederholung, sondern um eine Wiederverwendbarkeit an einer anderen Position im Kontrollfluss. Alle generell wiederverwendbaren Blockaktivitäten müssen sowohl eine *StartActivity* als auch eine *InitActivity* definieren, um die semantische Integrität des Prozesses auch bei einer Weitergabe an andere Ausführungeinheiten zu erhalten.

Transaktionen

Die Prozessbeschreibungssprache *XPDL* verfügt nicht über eine gesonderte Kennzeichnung von Aktivitäten, die im Rahmen einer Transaktion gemeinsam ausgeführt werden müssen (vgl. [WfMC02]). Es ist zwar möglich, durch die Verschachtelung bestehender Konstrukte ein transaktionales Verhalten zu erzielen, jedoch ist diese beschreibungsaufwändige Vorgehensweise für mobile Systeme unvorteilhaft. Vielmehr sollte durch eine konkrete Definition klar unterschieden werden können, ob der Mehraufwand für die transaktionale Durchführung einer Menge von Aktivitäten gerechtfertigt ist, oder ob es sich lediglich um eine einfache Folge nicht voneinander abhängiger Einzelaktivitäten handelt, die keine spezielle Behandlung erfordern.

Zur Kennzeichnung und Beschreibung von Transaktionen verwendet *DPDL* das Konstrukt *Transaction*, das seitens des Aufbaus mit dem Transaktionskonzept der Sprache *WSCI* (vgl. 4.2.4) vergleichbar ist. Eine *Transaction* enthält eine Menge von Referenzen auf Aktivitäten, die Teil der Transaktion sein sollen und gegebenenfalls eine Menge von Transitionen, die für diese Aktivitäten eine Ausführungsfolge definieren. Zusätzlich können im voraus weitere Details der Transaktion festgelegt werden. Das *Type*-Attribut gibt an, wie die Transaktion durchzuführen ist, zum Beispiel als *atomic*- oder *opennested*-Transaktion. Das *DPDL-Type*-Attribut ist jedoch beliebig erweiterbar. Durch *Retries* kann spezifiziert werden, ob die Transaktion wiederholt ausgeführt werden kann. Das ebenfalls optionale Element *Compensation* enthält Aktivitäten zur Rückabwicklung der Transaktion im Fehlerfall.

Codebeispiel 34 zeigt eine Zusammenfassung der genannten Aspekte.

```
<Transaction Id="TA1" Name="Transaction1" StartActivity="xyz"
                InitActivity="xyz" Type="atomic">
   <Retries>3</Retries>
   <ActivityRefs> ... </ActivityRefs>
   <Transitions> ... </Transitions>
   <Compensation>
      <ActivityRefs> ... </ActivityRefs>
      <Transitions> ... </Transitions>
   </Compensation>
</Transaction>
```

Codebeispiel 34: DPDL Transaction

5.1.4 Daten und Datentypen

Konkrete Daten werden in *DPDL* über *DataFields* realisiert. Ein *DataField* stellt eine Variable dar, die Daten beliebigen Typs aufnehmen kann. Es enthält dazu ein zu spezifizierendes Element *DataType*, welches einfache oder komplexe Datentypen vorgeben kann (vgl. [WfMC02]).

Einfache Datentypen (*BasicTypes*) werden durch die Angabe ihres Identifikators definiert, zum Beispiel *boolean*. Ein komplexer Datentyp kann mit Hilfe des Konstrukts *TypeDeclaration* aus bereits bestehenden Datentypen aufgebaut und als eigener *DeclaredType* festgelegt werden. *TypeDeclarations* können zum Beispiel aus XML-Schema Definitionen bezogen werden, wie Codebeispiel 35 zeigt.

```
<TypeDeclaration Id="Adress">
  <SchemaType>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
      <xsd:element name="Adress">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Street" type="xsd:string"/>
            <xsd:element name="PostalCode" type="xsd:string"/>
            <xsd:element name="City" type="xsd:string"/>
            <xsd:element name="Country" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </SchemaType>
</TypeDeclaration>
```

Codebeispiel 35: DPDL TypeDeclaration

Alle Variablen des Prozesses werden im Element *DataFields* deklariert und sind gegebenenfalls mit einem initialen Wert versehen. Dabei entspricht ein *DataField* der Semantik einer Variable. Für den Fall, dass Daten synchronisiert werden müssen, kann ein Zeitstempel (*TimeStamp*) eingefügt werden, der den Zeitpunkt der letzten Änderung des Datums angibt. Die Synchronisation von Daten könnte zum Beispiel über einen nicht-funktionalen Parameter gefordert werden oder von der Ausführungsumgebung automatisch realisiert werden, wenn Aktivitäten parallel bearbeitet werden. Das Format des Zeitstempels ist das bereits erläuterte *ISO8601*-Format (vgl. 5.2.1). Im folgenden Codebeispiel (Codebeispiel 36) ist die Deklaration von zwei Variablen „Name“ und „Adresse“ mit dafür möglichen Datentypen gezeigt.

```
<DataFields>

  <DataField Id="Name">
    <DataType>
      <BasicType Type="String"/>
    </DataType>
    <InitialValue>Meier</InitialValue>
    <TimeStamp>2005-09-17T09:30:47.000Z</TimeStamp>
  </DataField>
```

```
  <DataField Id="Adresse">
    <DataType>
      <DeclaredType Id="AdressType"/>
    </DataType>
  </DataField>

</DataFields>
```

Codebeispiel 36: DPDL DataFields

Daten müssen aber nicht notwendigerweise ständig im Prozess vorliegen und gemeinsam mit dem Kontrollfluss an alle Prozessteilnehmer transportiert werden. Alternativ können Daten von einer sogenannten *External Reference* bezogen werden, zum Beispiel vom einem stationären System oder von einem anderen mobilen Gerät (Codebeispiel 37). Dieses muss sich dazu natürlich in einem kommunikationsfähigen Zustand innerhalb der Kommunikationsreichweite des Nachfragers der Daten befinden. Durch die Verwendung einer *External Reference* kann somit der Datenfluss vom Kontrollfluss entkoppelt werden. Nicht ständig benötigte Daten oder große Datenmengen können bei Bedarf nachgeladen werden. Die Entkoppelung der Daten birgt jedoch auch das erhöhte Risiko, dass die Datenquelle zur Zeit des gewünschten Zugriffs nicht erreichbar ist. Die Entscheidung, ob Daten fest in den Prozess eingebunden oder getrennt bezogen werden sollen, muss daher im Einzelfall anwendungs- und prozessabhängig getroffen werden.

```
<DataField Id="Drawing">
  <DataType>
    <DeclaredType Type="Image"/>
  </DataType>
  <ExternalReference Location="http://www.abc.com/Very_Large_Image.bmp"/>
</DataField>
```

Codebeispiel 37: DPDL External Reference für die Typdeklaration

Ebenso können auch die Typdeklarationen von einem Ort außerhalb der Prozessbeschreibung nachgeladen werden, falls dies erforderlich sein sollte. Zum Beispiel können mit Hilfe des *ExternalReference*-Konstrukts innerhalb einer WSDL-Beschreibung definierte Typen referenziert werden (Codebeispiel 37).

```
<TypeDeclaration Id="Adress">
  <ExternalReference
    location= "http://xyz.com/AdressService.wsdl"
    xref="Adress"/>
</TypeDeclaration>
```

Codebeispiel 38: DPDL External Reference für die Typdeklaration

5.1.5 Beschreibung von Aktivitäten

Eine Aktivität kann in *DPDL* zur Steuerung des Kontrollflusses beitragen oder aber eine konkrete Aufgabe spezifizieren, die von einer Anwendung oder einem Benutzer ausgeführt werden soll. Aktivitäten zur Steuerung des Kontrollflusses sind die Blockkonstrukte *BlockActivity*, *TransactionActivity* und *LoopActivity*, die jeweils weitere Aktivitäten mit bestimmten Verhaltenseigenschaften beinhalten. Die leere *RouteActivity* wird auch als *DummyActivity* bezeichnet, da sie keinen eigenen Inhalt hat und lediglich zum Aufbau komplexerer Kontrollflussstrukturen verwendet wird. Sie wird definiert, um die Flexibilität der Prozessbeschreibungssprache zu erhöhen.

Zur Spezifizierung von konkreten Ausführungen wird das Konstrukt *Implementation* verwendet. Es lassen sich drei verschiedene Arten von *Implementations* unterscheiden: Die Aktivität kann von einer Softwareanwendung ausgeführt werden (*Tool*), es kann sich um eine manuelle Aktivität oder eine Aktivität mit Benutzeranteil handeln (*No*) oder die Aktivität stellt selbst wieder einen Prozess dar. In diesem Fall wird von einem *SubFlow* gesprochen (vgl. [WfMC02]).

In Codebeispiel 39 wird der generelle Einsatz einer *Implementation* als Inhalt einer Aktivität dargestellt. Die beiden Elemente *StartMode* und *FinishMode* beschreiben, ob die Aktivität automatisch durch die Ausführungsumgebung verwaltet oder ob sie manuell gestartet bzw. beendet werden soll.

```
<Activity Id="Activity1" Name="Beispielaktivität">
   <Description>Eine Beispielaktivität mit einer Implementation</Description>
   <Implementation>
      ...
   </Implementation>
   <StartMode Type="AUTOMATIC"/>
   <FinishMode Type="AUTOMATIC"/>
</Activity>
```

Codebeispiel 39: DPDL Implementation

Aufruf automatisierter Dienste und Anwendungen

Prozessbeschreibungsprachen, die für eine Ausführung von Prozessen auf zentralen Workflow-Management-Systemen entwickelt worden sind, legen meistens genau fest, welche Aufgaben durch Softwareanwendungen ausgeführt werden sollen und spezifizieren detailliert, von welchen konkreten Anwendungen die Aktivitäten bearbeitet werden müssen. Zum Beispiel referenziert eine *XPDL-Activity* eine *Application*, die dem zentralen Workflow-Management-System bekannt ist und von diesem aufgerufen werden kann. Die *Activity* enthält dabei die auszuführende Aufgabe und die aktuellen Daten, die als Parameter in die Ausführung einbezogen werden sollen, während die *Application* einen Verweis auf ein konkretes Softwareprogramm darstellt.

Im Fall der Prozessbeschreibung für mobile Systeme soll jedoch bevorzugt zur Laufzeit entschieden werden können, welche Dienste und Anwendungen verfügbar sind und sich aufgrund etwaiger nichtfunktionaler Anforderungen zur Ausführung der aktuellen Aktivität eignen. Dies erfordert eine Anpassung der Semantik des Applikationsbegriffs, um Aktivitäten und Anwendungen so abstrakt beschreiben zu können, dass erst unmittelbar vor der Ausführung einer Aktivität ein geeigneter Dienst zugewiesen und eine späte Bindung ermöglicht werden kann.

Damit zudem die Formulierung von *Applications* für verschiedene Aktivitäten wiederverwendet werden kann, wird die semantische Bezeichnung der Aufgabe von der Aktivität gelöst und einer generischen *Application* zugeordnet. Eine *Application* stellt in Sinne von *DPDL* also ein Muster für eine tatsächliche Anwendung dar, die in der Lage ist, die als Aktivität spezifizierte Aufgabe zu lösen. Das Muster der gesuchten Anwendung wird dabei durch einen eindeutigen Bezeichner referenziert, der nach dem Vorbild einer serviceorientierten Architektur in einer Registratur veröffentlicht werden kann und hierüber erlaubt, die Instanz eines konkreten Services, der auf diese Anforderung passt, zu beziehen.

Die Aktivität hält dabei die aktuellen Daten, die als Parameter in die Ausführung einbezogen werden sollen, während die *Application* die auszuführende Aufgabe in Form einer *UUID (Universal Unique Identifier)* und die *FormalParameters* für den eigentlichen Aufruf spezifiziert. Das Element *FormalParameters* enthält die Definition von beabsichtigten Aufrufparametern für die *Application* (vgl. [WfMC02]). *FormalParameters* verfügen über einen Datentyp, über die Angabe, ob es sich um Eingabe- oder Ausgabeparameter handelt und über eine festgelegte Reihenfolge der Parameter, sofern eine Sortierung mehrerer Parameter relevant ist. Soll zum Beispiel ein Dienst aufgerufen werden, der feststellt, ob eine eingegebene Zahl größer ist als eine andere, so ist die beabsichtigte Reihenfolge der Eingabeparameter von besonderer Bedeutung für das Ergebnis und muss dementsprechend formuliert werden.

Beim Aufruf der konkreten Anwendung werden die *FormalParameters* mit den konkreten Werten der Aktivität gefüllt. Diese sind dort als sogenannte *ActualParameters* definiert. Die Zuordnung der Parameter erfolgt über ein reihenfolgeabhängiges Mapping. Weichen die erwarteten Parameter der konkreten Anwendung von den in der *Application* definierten, beabsichtigten Parametern ab, so muss ein erneutes Mapping der *FormalParameters* auf die Ein- und Ausgabeparameter der Anwendung erfolgen. Ist kein Mapping möglich, weil zum Beispiel die Datentypen inkompatibel sind, so scheitert die Ausführung der Aktivität durch diese Anwendung und etwaige Fehlerbehandlungsmaßnahmen werden ausgeführt.

```
<Application Id="Drucken">
  <UUID>12345678901234567890123456789012</UUID>
  <FormalParameters>
    <FormalParameter Id="Nachname" Index="1" Mode="IN">
      <DataType>
        <BasicType Type="String"/>
      </DataType>
    </FormalParameter>
    <FormalParameter Id="Wohnort" Index="2" Mode="IN">
      <DataType>
        <DeclaredType Id="AdressType"/>
      </DataType>
    </FormalParameter>
```

```
    </FormalParameters>
</Application>

...

<Activity Id="Activity1">
    <Implementation>
        <Tool ApplicationId="Drucken">
            <ActualParameters>
                <ActualParameter>Name</ActualParameter>
                <ActualParameter>Adresse</ActualParameter>
            </ActualParameters>
        </Tool>
    </Implementation>
</Activity>
```

Codebeispiel 40: DPDL Application

Codebeispiel 40 verdeutlicht den Zusammenhang zwischen *Application* und *Activity* und das erläuterte Mapping zwischen *ActualParameters* und *FormalParameters*. Im Beispiel soll eine Aktivität den Namen und die Adresse eines Kunden ausdrucken. Die konkret zu druckenden Daten werden dazu als Referenzen auf *DataFields* in der Definition der *Activity* bereitgehalten. Es handelt sich hierbei um die *ActualParameters* „Name“ und „Adresse“. Diese werden der Reihenfolge nach den *FormalParameters* der *Application* zugeordnet. Also wird der *ActualParameter* „Name“ dem *FormalParameter* „Nachname“ zugewiesen. Ebenso wird der *ActualParameter* „Adresse“ an den *FormalParameter* „Wohnort“ gebunden.

Die Aufgabe ist es nun, eine Anwendung zu finden, die den Dienst „Drucken“ erfüllen kann. Dazu wird eine generische *Application* „Drucken“ erstellt, die als Muster für den aufzufindenden Dienst die Beschreibung der Aufgabe als *UUID* und die beabsichtigte Parametereingabe spezifiziert. Die Beschreibung der *Application* ist somit für jede Aktivität, deren Aufgabe es ist, Daten in der angegebenen Form zu drucken, wiederzuverwenden. Die Aktivität selbst kann als Ganzes mit der von ihr referenzierten *Application* ebenfalls an jeder beliebigen Position im Kontrollfluss erneut aufgerufen werden.

Neben den explizit als Aktivitäten spezifizierten automatisierten Aufgaben stellt auch die Auswertung von Transitionsbedingungen eine ausführbare Aktion dar, die nicht in allen Fällen von der mobilen Ausführungsumgebung selbst durchgeführt werden kann. Bei Transitionsbedingungen handelt es sich in erster Linie um Evaluationen, die einen Wahrheitswert zum Ergebnis haben. Zum Beispiel ist in diesem Zusammenhang eine Abfrage von Kontextdaten denkbar: „Ist die Außentemparatur über 20°C?“ oder die Evaluation bezieht sich auf den Vergleich von Prozessdaten mit Sollwerten: „Ist Variable x = 42?“

Um beliebige Arten von Transitionsbedingungen in der Prozessbeschreibung zuzulassen, verweisen diese in DPDL auf geeignete *Applications*, die in der Lage sind, die Auswertung der Bedingungen vorzunehmen. Die Evaluation einer Bedingung stellt somit eine *virtuelle Aktivität* dar, die der Auswertung von System- und Steuerinformationen der Ausführungsumgebung dient.

```
<Application Id="EQUALS" Name="demac.xpression.operation.EQUALS">
  <Description>Prüfung der Eingabeparameter auf Gleichheit</Description>
  <UUID>12345678901234567890123456789012</UUID>
  <FormalParameters>
    <FormalParameter Id="Value1" Index="1" Mode="IN">
      <DataType>
        <BasicType Type="String"/>
      </DataType>
    </FormalParameter>
    <FormalParameter Id="Value2" Index="2" Mode="IN">
      <DataType>
        <BasicType Type="String"/>
      </DataType>
    </FormalParameter>
    <FormalParameter Id="Value3" Mode="OUT">
      <DataType>
        <BasicType Type="Boolean"/>
      </DataType>
    </FormalParameter>
  </FormalParameters>
</Application>

  ...

<Transition Id="Transition1" From="ActivityRef1" To="ActivityRef2">
  <Condition Type="CONDITION">
    <Xpression>
      <Tool ApplicationId="EQUALS">
        <ActualParameters>
          <ActualParameter>x</ActualParameter>
          <ActualParameter>y</ActualParameter>
          <ActualParameter>result</ActualParameter>
        </ActualParameters>
      </Tool>
    </Xpression>
  </Condition>
</Transition>
```

Codebeispiel 41: DPDL Condition

Codebeispiel 41 zeigt eine Transitionsbedingung (*Condition*), welche evaluiert, ob zwei Variablen *x* und *y* den gleichen Wert aufweisen. Dazu wird eine Anwendung „EQUALS“ gesucht, die zwei Werte auf Gleichheit überprüfen kann. Das Ergebnis der Anwendung, der Wert der Variable „Result“, stellt das Ergebnis der Transitionsbedingung dar und beeinflusst gegebenenfalls den weiteren Verlauf des Prozesses.

Formulierung manueller Aktivitäten und Benutzerinteraktionen

Aktivitäten, die nicht von einem Softwaredienst ausgeführt werden sollen, können mit dem Konstrukt *Implementation* vom Typ *No* spezifiziert werden. In Zusammenhang mit dem *Start*- bzw. *FinishMode* einer Aktivität können über das Attribut *MANUAL* somit gänzlich manuelle Aktivitäten gebildet werden, die zwar noch vom mobilen Gerät als Workflow-Management-System verwaltet werden, jedoch nur von einem Benutzer gestartet, ausgeführt und abgeschlossen werden können. Eine Dateneingabe und -ausgabe erfolgt dabei nicht. Es ist daher von besonderer Bedeutung, die manuell zu erledigende Aufgabe konkret zu beschreiben, damit der ausführende Benutzer weiß, was er zu tun hat. Dieses kann explizit über eine ausführliche *Description* der Aktivität ausgedrückt werden oder implizit über das Wissen und die Erfahrungen des Benutzers geschehen, der seine Aufgabe bereits kennt oder seine Tätigkeit aus dem Kontext der Aktivität ableiten kann.

Codebeispiel 35 zeigt den möglichen Aufbau einer vollständig manuell durchzuführenden Aktivität. Dabei wird die Aufgabe „Reparatur“ manuell gestartet, wenn der verantwortliche Mitarbeiter einer Werkstatt seine Arbeit beginnt. Die Aktivität wird in den Zustand *finished* versetzt, wenn der Mitarbeiter angibt, die Reparatur abgeschlossen zu haben. Eine Kontrolle, ob die Ausführung stattgefunden hat, kann hierbei nicht durchgeführt werden.

```
<Activity Id="abc" Name="Reparatur">
   <Description>Schäden am Fahrzeug reparieren</Description>
   <Implementation>
      <No/>
   </Implementation>
   <StartMode Type="MANUAL"/>
   <FinishMode Type="MANUAL"/>
</Activity>
```

Codebeispiel 42: DPDL Manuelle Aktivität

Anspruchsvollere Interaktionen mit Benutzern können neben der Formulierung einer manuellen Aktivität auch über Anwendungen realisiert werden, die im Prozess enthaltene Aufgaben für den Benutzer aufbereiten und ihm diese, zum Beispiel über ein graphisches Benutzerinterface, zur Bearbeitung präsentieren. In Codebeispiel 43 verhilft eine Anwendung „UserInterface“ einem Benutzer zur Dateneingabe. Er instantiiert durch seine Eingabe eine Variable mit dem Namen „Kundenname“. Wenn der Benutzer die Eingabe abgeschlossen hat, setzt er die Aktivität mit Hilfe der Anwendung in den Zustand *finished*. Implementationsabhängig kann dabei geprüft werden, ob das Datenfeld „Kundenname“ nach der Eingabe einen zuläsigen Wert enthält.

```
<Application Id="UserInterface">
   <Description>UserInterface für einfache Dateneingabe</Description>
   <UUID>12345678901234567890123456789012</UUID>
   <FormalParameters>
      <FormalParameter Id="Name" Mode="IN">
         <DataType>
            <BasicType Type="STRING"></BasicType>
         </DataType>
      </FormalParameter>
   </FormalParameters>
</Application>

...

<Activity Id="abc" Name="Dateneingabe">
  <Implementation>
     <Tool ApplicationId="UserInterface">
        <ActualParameters>
           <ActualParameter>Kundenname</ActualParameter>
        </ActualParameters>
     </Tool>
   </Implementation>
   <StartMode Type="AUTOMATIC"/>
   <FinishMode Type="MANUAL"/>
</Activity>
```

Codebeispiel 43: DPDL Benutzerinteraktion durch eine Anwendung

Spezifikation von konkreten Prozessteilnehmern

Die Spezifikation einer Aktivität kann durch die Zuweisung eines bestimmten Prozessteilnehmers ergänzt werden. *DPDL* unterstützt die Formulierung von konkreten und generischen Prozessteilnehmern, die angefordert werden können, um eine bestimmte Aufgabe wahrzunehmen. Dazu ist das Konstrukt *Participant* erweitert worden (vgl. [WfMC02]). Bei einem *Participant* kann es sich einerseits um einen menschlichen Benutzer handeln, der eine manuelle Aktivität persönlich ausführen oder eine automatisierte Aktivität überwachen soll, oder um ein bestimmtes Gerät, welches den Prozess verwalten soll, während die betreffende Aufgabe ausgeführt wird.

Konkrete Geräte können über die Angabe ihrer *UUID* beschrieben werden, menschliche Benutzer zum Beispiel über ihren Namen oder über eine digitale Identität. Soll kein bestimmter Teilnehmer angegeben werden, um einen passenden konkreten *Participant* zur Laufzeit des Prozesses auszuwählen, kann für Personen und Geräte eine Rollendefinition spezifiziert werden. Dabei kann zum Beispiel der Typ eines Geräts klassifiziert oder ein anderes Merkmal wie das Vorhandensein eines Benutzerinterfaces gefordert werden. Eine Kombination von Geräten und menschlichen Benutzern ist angebracht, wenn sich ein Benutzer über ein Gerät mit bestimmten Eigenschaften am Prozess beteiligen soll. So sind auch Aussagen wie „Herr Meiers Notebook“ oder „ein Firmenangehöriger mit Mobiltelefon“ in der Prozessbeschreibung möglich.

```
<Participants>
  <Participant Id="Meier" Name="Herr Horst Meier">
    <Devices>
      <Device Id="111" Name="MeiersPC">
        <UUID>12345678901234567890123456789012</UUID>
      </Device>
      <Device Id="222" Name="MeiersHandy">
        <Devicetype Type="Cellphone"/>
      </Device>
    </Devices>
    <Human>HorstMeier</Human>
  </Participant>
</Participants>


<ActivityReferences>
  <ActivityRef Id="1" ActivityId="Activity1" PartcipantId="Meier“ ... />
</ActivityReferences>
```

Codebeispiel 44: DPDL Participant

Im Codebeispiel 44 wird dargestellt, wie ein bestimmter Teilnehmer an eine Aktivität gebunden werden kann. In diesem Fall ist Herr Meier dafür verantwortlich, die Aktivität „Activity1“ auszuführen. Der Prozess wird ihm dazu auf ein Mobiltelefon oder auf seinen Personal Computer geschickt, falls dieser nicht bereits dort verarbeitet wird.

Modifikation von Aktivitäten

Für den Fall, dass keine passenden Dienste zur Erfüllung einer Aufgabe gefunden werden können, besteht die Möglichkeit, Aktivitäten in DPDL gegen andere Aktivitäten auszutauschen, um das Ziel der Aktivität auf einem alternativen Weg zu erreichen. Wird zum Beispiel von einem Dienst gefordert, zwei Zahlen *x* und *y* miteinander zu multiplizieren, steht aber nur ein Dienst zur Addition von Zahlen zur Verfügung, so könnte die Multiplikations-Aktivität alternativ durch eine Schleife ersetzt werden, die die Addition der zu multiplizierenden Zahl *y* genau *x*-mal ausführt.

Dieses Szenario wird in DPDL durch die Möglichkeit unterstützt, Aktivitäten als bearbeitbar zu deklarieren (Abbildung 47). Da die Modifikation einer Aktivität einen erheblichen Eingriff in den Prozess und in die Ziele des Initiators des Prozesses darstellt, kann für jede Aktivität an jeder Position des Kontrollflusses angegeben werden, ob die Aktivität bearbeitet werden darf oder nicht. Für die Kennzeichnung ist das Attribut *Editable* zuständig (Codebeispiel 45). Zulässige Werte für *Editable* sind *NO*, *EDIT*, *REMOVE* und *ALL*. Dabei bezeichnet *NO*, dass die Aktivität nicht manipuliert werden darf, *EDIT* kennzeichnet, dass Aktivitäten lediglich ausgetauscht und nicht gelöscht werden dürfen, *REMOVE* erlaubt das Löschen der Aktivität und *ALL* erlaubt alle möglichen Modifikationen.

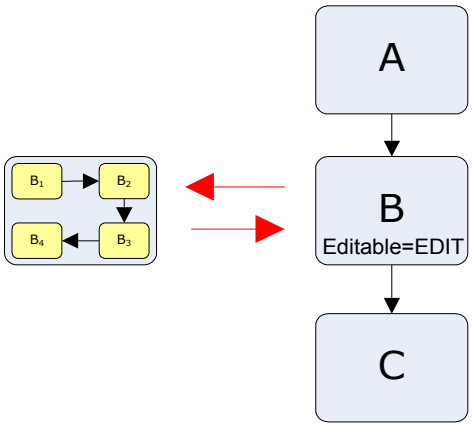


Abb. 47: DPDL Austausch einer Aktivität

Das Wissen über die logischen Zusammenhänge von Aktivitäten und deren mögliche Substitutionen wird nicht von DPDL bereitgehalten und muss vom Kontext der Ausführungsumgebung bezogen werden.

```
<ActivityReferences>
   <ActivityRef Id="1" ActivityId="Activity1" Editable="ALL"  ... />
</ActivityReferences>
```

Codebeispiel 45: DPDL Modifikationen

5.1.6 Formulierung nicht-funktionaler Aspekte

Die Formulierung von nicht-funktionalen Aspekten zur Beschreibung der Interessen des Initiators eines Prozesses erfolgt in *DPDL* über das Konstrukt *Strategy*. Ein *Package* kann beliebig viele *Strategies* definieren, wobei ein Prozess oder eine Aktivität jedoch nur genau eine *Strategy* als Anforderung besitzen darf. Hierzu können *Strategies* in sich gegliedert werden und eine Fülle von verschiedenen Anforderungen beinhalten.

Das folgende Codebeispiel zeigt, wie nicht-funktionale Aspekte formuliert und gegliedert werden können (Codebeispiel 46):

```
<Strategies>

  <Strategy Id="123" Name="ProcessStrategy">
    <StrategyProperty Id="1" Name="Channels">
      <Requirements>
        <Requirement Name="MinSpeed" Value="100"/>
        <Requirement Name="NetworkType" Value="WLAN"/>
      </Requirements>
    </StrategyProperty>
    <StrategyProperty Id="2" Name="Security">
      <Requirements>
        <Requirement Name="Authentification" Value="true"/>
        <Requirement Name="Non-Repudiation" Value="true"/>
      </Requirements>
    </StrategyProperty>
  </Strategy>

  <Strategy Id="124" Name="ActivityStrategy">
    <StrategyProperty Id="1" Name="Cost">
      <Requirements>
        <Requirement Name="MaxNetworkCost" Value="10"/>
        <Requirement Name="MaxServiceCost" Value="20"/>
       </Requirements>
    </StrategyProperty>
  </Strategy>

</Strategies>
```

Codebeispiel 46: DPDL Strategies

Eine *Strategy* kann in sogenannte *StrategyProperties* unterteilt werden, um die Anforderungen nach Gruppen zu sortieren und so die Abfrage von Eigenschaften der Prozessteilnehmer zu erleichtern. Mögliche Gruppierungen können zum Beispiel Sicherheit, Netzwerkeigenschaften oder Kosten sein. Innerhalb eines *StrategyProperties* werden die konkreten Anforderungen spezifiziert. Dieses geschieht durch das Element *Requirement*. Ein *Requirement* ist ein Key-Value-Paar, welches einen Identifikator (*Name*) für die Anforderung und den geforderten Sollwert (*Value*) enthält. Die Bindung einer *Strategy* an einen Prozess oder an eine Aktivität wird über die Referenzierung der *StrategyId* vollzogen. Codebeispiel 47 verdeutlicht die Referenzierung der Strategien aus dem vorgenannten Beispiel.

```
<WorkflowProcess Id="1" StartActivity="1" StrategyId="123“ ... >

   ...
  <ActivityReferences>
    <ActivityRef Id="1" ActivityId="TestActivity1" StrategyId="124“ ... />
                ...
  </ActivityReferences>
   ...

</WorkflowProcess>
```

Codebeispiel 47: Referenzierung einer Strategy mit DPDL

Die an den Prozess gebundenen, nicht-funktionalen Anforderungen beziehen sich auf die Identität, die Qualität oder die Leistungsmerkmale der potentiellen Prozessteilnehmer, die eine Prozessbeschreibung übernehmen dürfen, um sie weiter zu bearbeiten. Hingegen beziehen sich die an einzelne Aktivitäten gebundenen *Strategies* auf Dienste, die zur lokalen Ausführung der jeweiligen Aufgabe selektiert werden. Da sich die Anforderungen einer Aktivität im Verlauf des Kontrollflusses ändern können, wird das *StrategyId*-Attribut der jeweiligen *Activity Reference* zugeordnet.

Die geforderten nicht-funktionalen Anforderungen können während der Ausführung des Prozesses mit den konkreten Leistungsmerkmalen potentieller Prozessteilnehmer abgeglichen werden. Abbildung 48 zeigt einen Prozess mit möglichen nicht-funktionalen Aspekten wie Verbindungsqualität, Kosten und Sicherheitskriterien. Der Vergleich mit den Eigenschaften verfügbarer Geräte und Dienste verhilft dem Prozess zu einer Ausführung gleichbleibender Qualität, die den Vorgaben des Initiators des Prozesses entspricht.

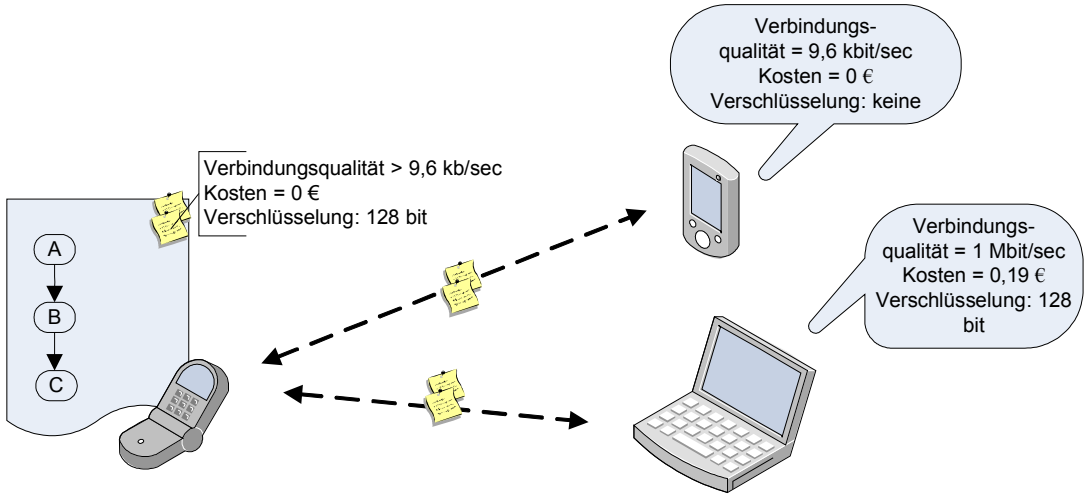


Abb. 48: Abgleich von nicht-funktionalen Anforderungen mit Leistungsmerkmalen potentieller Prozessteilnehmer

5.1.7 Erweiterbarkeit

DPDL verwendet die in *XPDL* eingeführten *Extended Attributes*, um die Prozessbeschreibung für individuelle Zwecke anpassbar zu gestalten. Diese sind durch die Angabe ihres Namens und eines Wertes definiert und können beliebige weitere Inhalte aufweisen. Zum Beispiel können hiermit zusätzliche Informationen ausgedrückt werden, um spezielle Anwendungen zu ermöglichen.

ExtendedAttributes können in *DPDL* auf alle *Activities*, *Applications*, *Connection Reset Handlers*, *Data Fields*, *Participants*, *Transitions*, *Type Declarations*, auf das gesamte *Package* und auf den *WorkflowProcess* selbst angewendet werden.

5.2 Ausführungsumgebung

Da viele Anforderungen an eine Prozessintegration in mobile Systeme nur durch ein adäquates Zusammenspiel zwischen der Prozessbeschreibungsprache und ihrer Ausführungsumgebung vollständig erfüllt werden können, wird im folgenden ein möglicher Prototyp einer Ausführungsumgebung für mobile Systeme vorgestellt.

Die Ausführungsumgebung ist dabei Teil des vorgestellten *DEMAC Process Services*. Sie stellt ein Workflow-Management-System dar, welches in der Lage ist, Prozesse zu verwalten, die verteilte Ausführung mit anderen Instanzen des Systems zu koordinieren und Prozesse soweit auszuführen, wie es die Leistungsfähigkeit des mobilen Geräts und das Potential seiner unmittelbaren Umgebung erlauben.

Um Informationen über die relevante Umgebung des mobilen Systems für die Verarbeitung des Prozesses zu beziehen, arbeitet die Ausführungsumgebung eng mit dem *DEMAC Context Service* zusammen. Dieser ist dafür verantwortlich, Kontextdaten über lokal verfügbare Dienste und Prozessteilnehmer bereitzustellen, Eigenschaften und Identitätsinformationen des eigenen Geräts zu verwalten und den *Process Service* bei der Auswertung von Bedingungsausdrücken zu unterstützen.

Die folgenden Abschnitte vermitteln einen Einblick in die Architektur des *Process Services* und stellen dessen wichtigste Komponenten vor, die an der Verwaltung und Verarbeitung von Prozessen auf mobilen Geräten beteiligt sind.

5.2.1 Architektur

Eine der Hauptanforderungen an ein Prozessmanagementsystem für mobile Systeme ist die Berücksichtigung deren eingeschränkter Leistungsfähigkeit. Wie die Untersuchung aktueller Technologien in Kapitel 3.1.3 ergeben hat, können die Leistungsmerkmale mobiler Systeme zudem abhängig von ihrer Größe und ihrem Einsatzzweck stark unterschiedlich sein. Eine Orientierung der Ausführungsumgebung an dem vermeintlich schwächsten System würde daher weit hinter dem eigentlich verfügbaren Potential zurückbleiben und sich auch angesichts des rasanten Fortschritts im Bereich der angesprochenen Technologien nicht bewähren.

Damit möglichst viele mobile Geräte in Abhängigkeit ihrer Leistungsfähigkeit am Prozess teilnehmen können, ist der *Process Service* modular aufgebaut. Er besteht im einfachsten Fall aus einer Kernkomponente (*Core*), welche die Verwaltung von Prozessen auf einem mobilen Gerät implementiert. Dazu gehört das Speichern der Prozessbeschreibung sowie die Möglichkeit des Transfers von Prozessen auf andere Instanzen der Ausführungsumgebung. Die Kernkomponente stellt außerdem eine Schnittstelle für Anwendungen zur Verfügung, mit der Prozessbeschreibungen zur Abarbeitung an den *Process Service* übergeben werden können.

Für in Bezug auf Speicherplatz und Rechenkapazität leistungsfähigere Geräte wird die Kernkomponente durch ein Basismodul (*Base*) ergänzt, welches die Interpretation und die Ausführung von Prozessbeschreibungen zur Aufgabe hat. Die Basiskomponente baut auf den von der Kernkomponente

bereitgestellten Funktionalitäten auf und prüft, ob die Ausführungsumgebung technisch und fachlich in der Lage ist, die aktuell anstehende Aktivität des Prozesses zu erfüllen. Kann die Basiskomponente eine Aktivität nicht erfüllen, gibt sie die Prozessbeschreibung an die Kernkomponente zurück, um andere Geräte zur Ausführung des Prozesses zu finden.

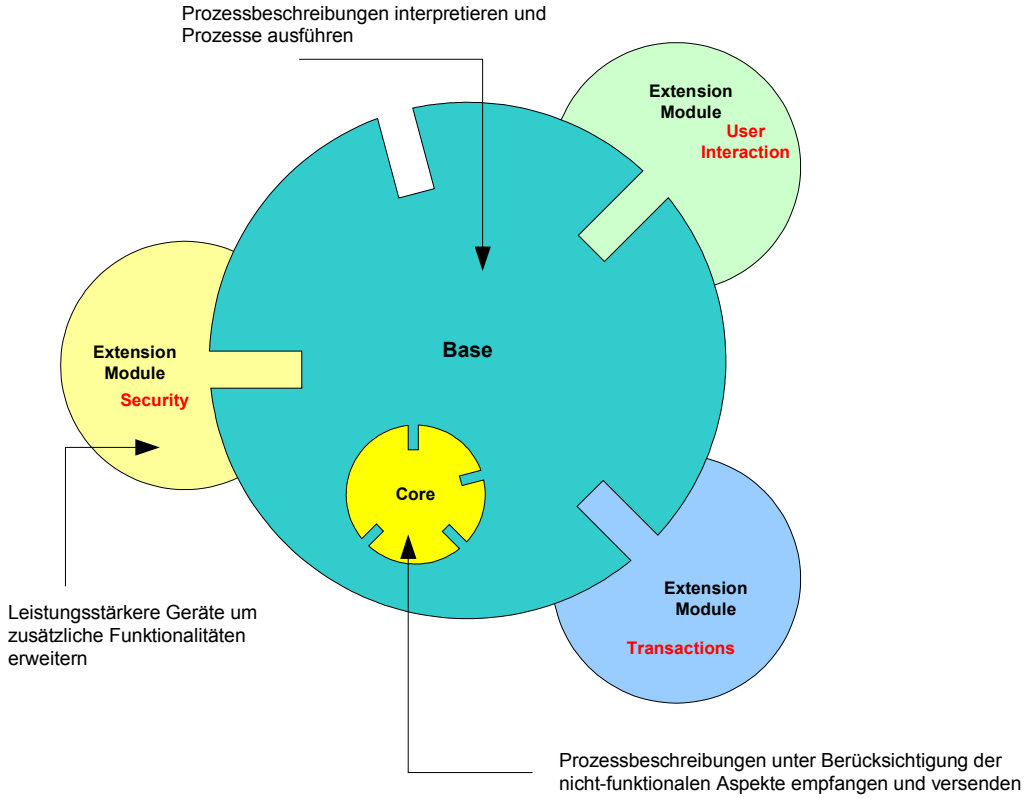


Abb. 49: Process Service Architektur

Erweiterte Funktionen, die nur von leistungsstärkeren Geräten mit zusätzlichen Eigenschaften erfüllt werden können, werden durch Zusatzkomponenten (*Extension Modules*) realisiert. Ein *Extension Module* ist jeweils für die Ausführung einer bestimmten Zusatzfunktionalität verantwortlich, die in engem Zusammenhang mit den Eigenschaften eines mobilen Geräts steht. Zum Beispiel ist für das mögliche *Extension Module* „User Interaction“ ein geeignetes Benutzerinterface nötig, um mit einem menschlichen Anwender kommunizieren zu können. Das *Extension Module* „Security“ benötigt hingegenzum Beispiel ein Mindestmaß an freier Rechenkapazität, die zur Verschlüsselung von Prozessbeschreibungen aufgewendet werden kann.

Abbildung 49 illustriert den Aufbau des Process Services durch die beschriebenen Module. Durch die Möglichkeit eines modularen Aufbaus wird auch mobilen Geräten geringerer Leistungsfähigkeit die Teilnahme am Prozess ermöglicht und damit die Anzahl potentieller Prozessteilnehmer erhöht. Die Wahrscheinlichkeit einer erfolgreichen Ausführung von Prozessen nimmt mit steigender Teilnehmerzahl zu, so dass sich einzelne schwache mobile Systeme gegenseitig ergänzen können. Natürlich können auch geeignete stationäre Geräte am Prozess teilnehmen, indem sie zumindest die Kern-

komponente des *Process Services* implementieren. Eine Integration von mobilen Systemen und leistungsfähigen stationären Systemen wirkt sich dabei zusätzlich positiv auf die Ausführbarkeit einzelner Prozesse aus.

5.2.2 Kernkomponente mit grundlegenden Funktionen

Die Kernkomponente des *Process Services* bietet mobilen Geräten eine minimale Form der Beteiligung an der Abarbeitung von Prozessen. Prozessbeschreibungen können als Ganzes von Anwendungen oder anderen Prozessteilnehmern übernommen und unverändert an andere Prozessteilnehmer mit höherer oder gleicher Leistungsfähigkeit weitergeben werden. Dazu werden bei Bedarf nicht-funktionale Aspekte beachtet. Ziel der Kernkomponente ist es, möglichst einen Ortswechsel der Prozessbeschreibung herbeizuführen, um weitere Dienste an anderen Orten akquirieren zu können.

Die Kernkomponente ist als alleinstehende Ausführungsumgebung für Geräte mit sehr beschränkten Leistungsmerkmalen geeignet. Die prototypische Implementierung benötigt nur einen relativ geringen Speicherplatz und kann auf den Plattformen J2ME MIDP 2.0 sowie Personal Profile 1.0 und auf J2SE eingesetzt werden.

Abbildung 50 zeigt ein stark vereinfachtes Klassendiagramm der Kernkomponente. Im Mittelpunkt steht die Klasse *ProcessService*, die Schnittstellen zu Anwendungen, zu anderen internen Services wie dem *Context Service* und zu anderen Instanzen der Middleware bereitstellt.

Wird eine neue Prozessbeschreibung als XML-Dokument durch eine Anwendung an den *Process Service* weitergeben, so wird diese zunächst durch den *PersistenceService* gespeichert. Hierbei handelt es sich um eine persistente Speicherung der Prozessbeschreibung in einer Datei oder in einem sogenannten *RecordStore*. *RecordStores* stellen das alternative Persistenzkonzept von MIDP 2.0 dar, da hier aus Sicherheitsgründen nicht auf das Dateisystem des mobilen Geräts zugegriffen werden darf. Eine unmittelbare Speicherung der Prozessbeschreibung ist notwendig, um einen Verlust der Daten durch beabsichtigtes oder versehentliches Abschalten oder einen Ausfall des mobilen Geräts zu vermeiden.

Um die Verwaltung der auf einem mobilen Gerät gespeicherten Prozessbeschreibungen zu erleichtern und nicht für jede Aktion die gesamte Prozessbeschreibung einlesen zu müssen, werden die wichtigsten Daten eines Prozesses und seines Bearbeitungszustandes in einem *DataSet* festgehalten. Alle *DataSets* werden in einer gemeinsamen Datei bzw. in einem gemeinsamen *RecordStore* gespeichert und halten so wichtige Informationen der Prozesse, wie zum Beispiel ihren Speicherort, ihren Dateinamen oder den Fortschritt ihrer Bearbeitung bereit. Für den Fall, dass das Gerät zum Beispiel während der Suche nach einem geeigneten Prozessteilnehmer abgeschaltet wird, muss der *Process Service* die Prozesse nach einem Neustart des Programmes nicht erneut prüfen, sondern kann anhand des Bearbeitungszustandes im *DataSet* des einzelnen Prozesses feststellen, dass dieser bereits beim *Context Service* zur Suche nach einem geeigneten Prozessteilnehmer abgestellt ist. Dieses Konzept setzt voraus, dass auch der *Context Service* seine Aufgaben entsprechend persistiert und nach einem Neustart des Programmes seine Arbeit in dem vorliegenden Zustand wieder aufnehmen kann.

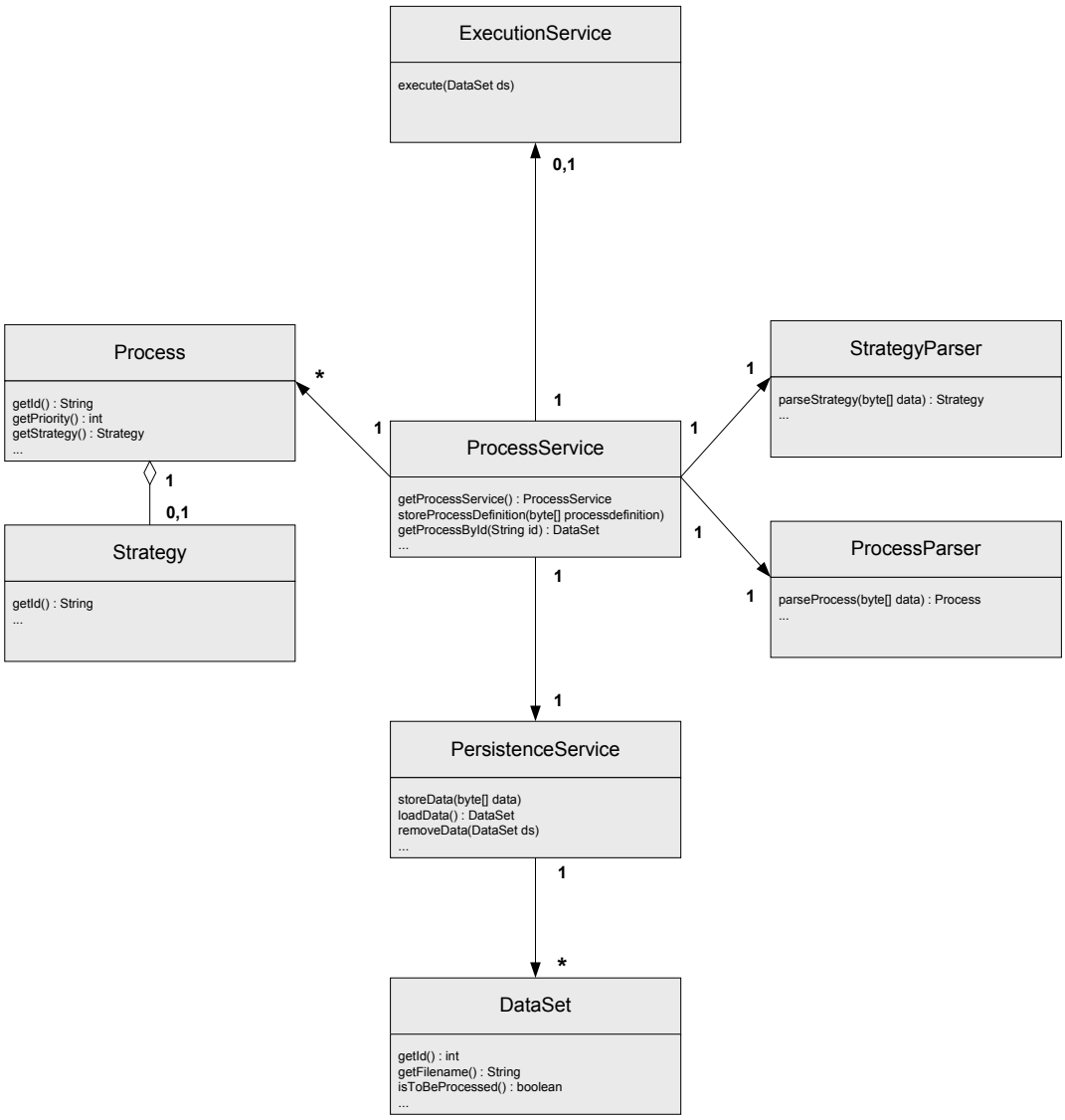


Abb. 50: Klassendiagramm der Kernkomponente

Wenn der *Process Service* gestartet wird, beginnt er automatisch alle gespeicherten Prozessbeschreibungen auf ihre Ausführbarkeit und ihre mögliche Weitergabe zu prüfen. Dazu werden mit Hilfe eines XML-Parsers (*ProcessParser*) die wichtigsten Daten des Prozesses aus der Prozessbeschreibung in ein Objekt vom Typ *Process* geladen. Die Klasse *Process* vergegenständlicht eine kompakte Version eines DPDL-Prozesses und enthält nur diejenigen Daten, die in der Kernkomponente benötigt werden. Dazu gehören zum Beispiel die *Id* des Prozesses, seine Priorität und die Angabe seines *AccessLevels*, um entscheiden zu können, ob es sich um einen eigenständigen Prozess oder um einen Subprozess handelt. Außerdem wird eine eventuell vorhandene *Strategy*, die sich auf die Weitergabe des Prozesses bezieht, vom *StrategyParser* aus der Prozessbeschreibung extrahiert und dem Prozess zugeordnet. Da es pro Prozess nur höchstens eine Strategie geben kann, die sich auf den Prozess und seine Weitergabe bezieht, kann die Klasse *Process* auch höchstens ein Objekt der Klasse *Strategy* referenzieren.

Wurden alle Prozesse auf diese Weise geladen, können sie nach Priorität sortiert werden und in Abhängigkeit ihrer Priorität vom *Process Service* bearbeitet werden. Dazu wird geprüft, ob ein Basismodul an die Kernkomponente angeschlossen ist und diesem gegebenenfalls die Prozessbeschreibung zur Ausführung übertragen. Eine Basiskomponente implementiert die abstrakte Klasse *ExecutionService*. Ist keine Basiskomponente integriert, kann der Prozess nur an andere Geräte weitergegeben und nicht ausgeführt werden.

Die Klassen *ProcessService*, *PersistenceService*, *ExecutionService* sowie die beiden *Parser*-Klassen sind als *Singletons* implementiert. Ein *Singleton* ist eine Klasse, von der nur eine einzige Instanz erzeugt werden darf, auf welche aber von allen dazu berechtigten Klassen zugegriffen werden kann [Gam02]. Die Implementierung als *Singleton* ist vorteilhaft, wenn alle zugreifenden Klassen mit dem gleichen Objekt und den gleichen Daten der Klasse arbeiten sollen. Zudem vermeidet dieses Entwurfsmuster die Generierung unnötiger Objekte und hilft somit beim Einsatz auf mobilen Geräten die Objektzahl gering zu halten und Speicherplatz einzusparen.

Ist kein Basismodul angeschlossen oder konnte der *ExecutionService* die Bearbeitung des Prozesses nicht erfolgreich abschließen, so sendet der *Process Service* einen Request an den *Context Service*, um einen anderen Teilnehmer zur Fortführung des Prozesses aufzufinden. Wenn der Prozess zusätzlich eine *Strategy* besitzt, wird diese dem *Context Service* zur Verfügung gestellt, damit die dort geforderten Eigenschaften an potentielle Prozessteilnehmer in die Auswahl einfließen können.

Während der *Context Service* nach geeigneten Teilnehmern sucht, kann der *Process Service* mit der Bearbeitung weiterer Prozesse fortfahren. Dieses wird durch eine Entkoppelung der beiden Services über eine ereignisgesteuerte Kommunikation erreicht. Abbildung 51 zeigt die Kommunikation des *Process Services* mit den anderen DEMAC-Komponenten (vgl. 2.4) im Detail.

Für jeden Prozess, den der *Process Service* an ein anderes Gerät weitergeben muss, sendet er ein *RequestParticipantEvent* an den *Context Service*. Dieses enthält die *Id* des abzugebenden Prozesses und gegebenenfalls dessen *Strategy*. Der *Context Service* kann nun die verfügbaren Geräte in seiner Umgebung auf die darin festgelegten Anforderungen prüfen.

Unter *Device Properties* sollen allgemein die tatsächlichen Eigenschaften eines Gerätes verstanden werden (Abb. 51). Es kann sich dabei aber nicht nur um technische Merkmale, sondern auch um nichtfunktionale Restriktionen und Vorgaben von Benutzern bezüglich der Verwendung ihrer Geräte handeln. Zum Beispiel könnten Benutzer spezifizieren, ob eine lokale Ausführung des Prozesses einer Weitergabe an andere Prozessteilnehmer vorgezogen werden soll, etwa aus Kosten- oder Sicherheitsaspekten. Andererseits kann eine Weitergabe an andere Komponenten das eigene Gerät von aufwändigen Berechnungen entlasten und deshalb gegenüber einer möglichen eigenen Ausführung vorteilhaft sein. Auch die Definition von Timeouts kann gerätespezifisch und an bestimmte Benutzerinteressen gekoppelt sein. Der *Context Service* muss diese Eigenschaften und Anforderungen mit den vom Prozess geforderten Merkmalen abgleichen und so ein geeignetes spezielles oder, für den Fall dass keine *Strategy* vorliegt, ein beliebiges mobiles Gerät als zukünftigen Prozessteilnehmer auswählen.

Findet der *Context Service* einen passenden Teilnehmer zur Übertragung der Prozessbeschreibung, so teilt er dem *Process Service* den eindeutigen Identifikator dieses Teilnehmers mit. Hierbei handelt es sich um den sogenannten *DeviceHandle*, der von der Transportschicht der DEMAC-Architektur (*Transport Service*) bereitgestellt wird.

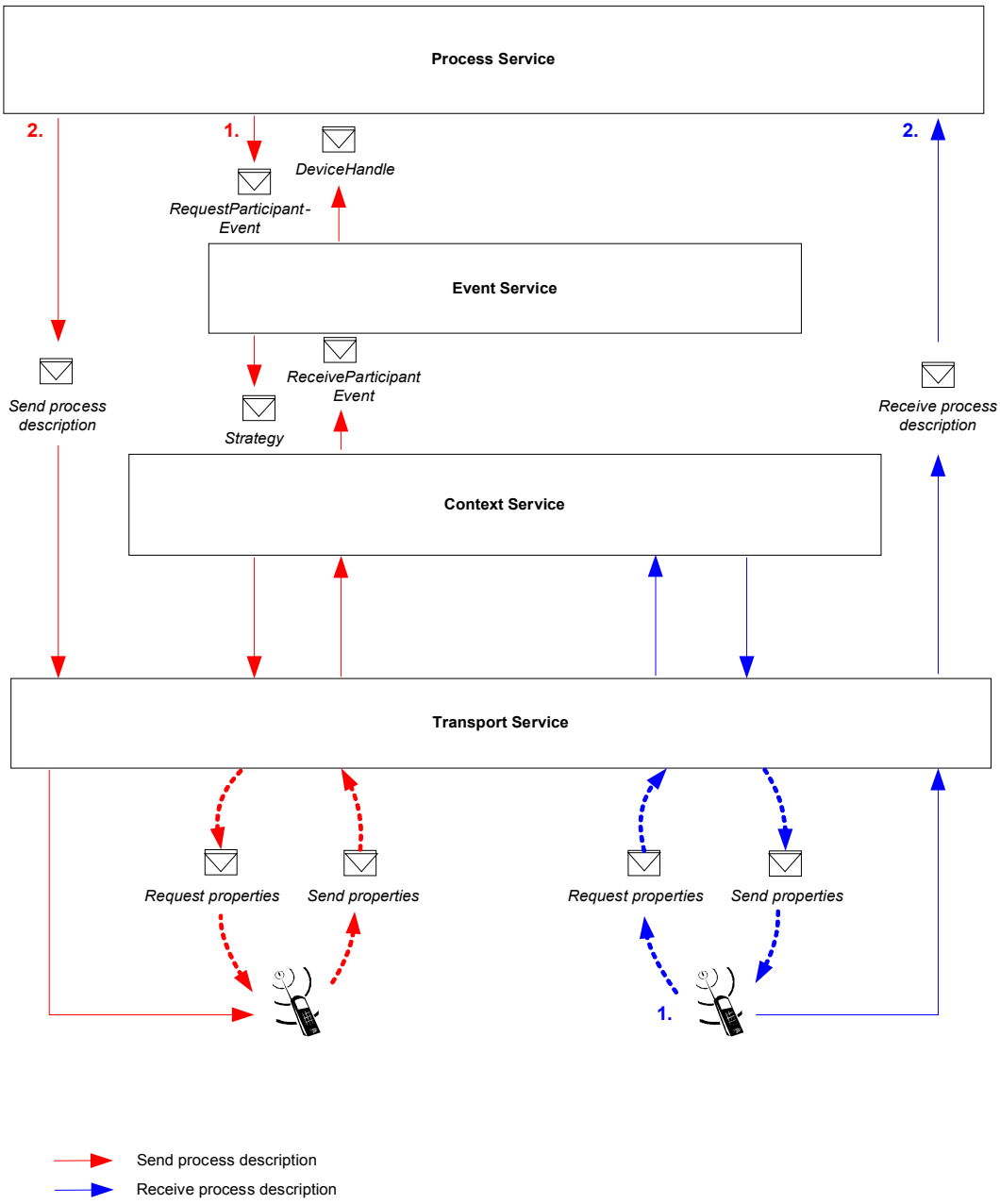


Abb. 51: Programminterne Kommunikation

Im umgekehrten Fall kann auch der *Process Service* eines anderen Geräts das aktuell betrachtete Gerät als Prozessteilnehmer auswählen. Es prüft zunächst, ob das Gerät seinen Anforderungen entspricht und versucht gegebenenfalls, ihm eine Prozessbescheibung zu übergeben. Die eingehende Kommunikation und die dazugehörigen Prozessdaten werden dabei direkt vom *Transport Service* an den *Process Service* weitergereicht.

Über einen vom *Transport Service* bereitgestellten *DeviceHandle* nimmt der *Process Service* die direkte Kommunikation mit dem Teilnehmer auf. Er bedient sich hierzu eines speziellen Protokolls zur Weitergabe von Prozessen (Abbildung 52).

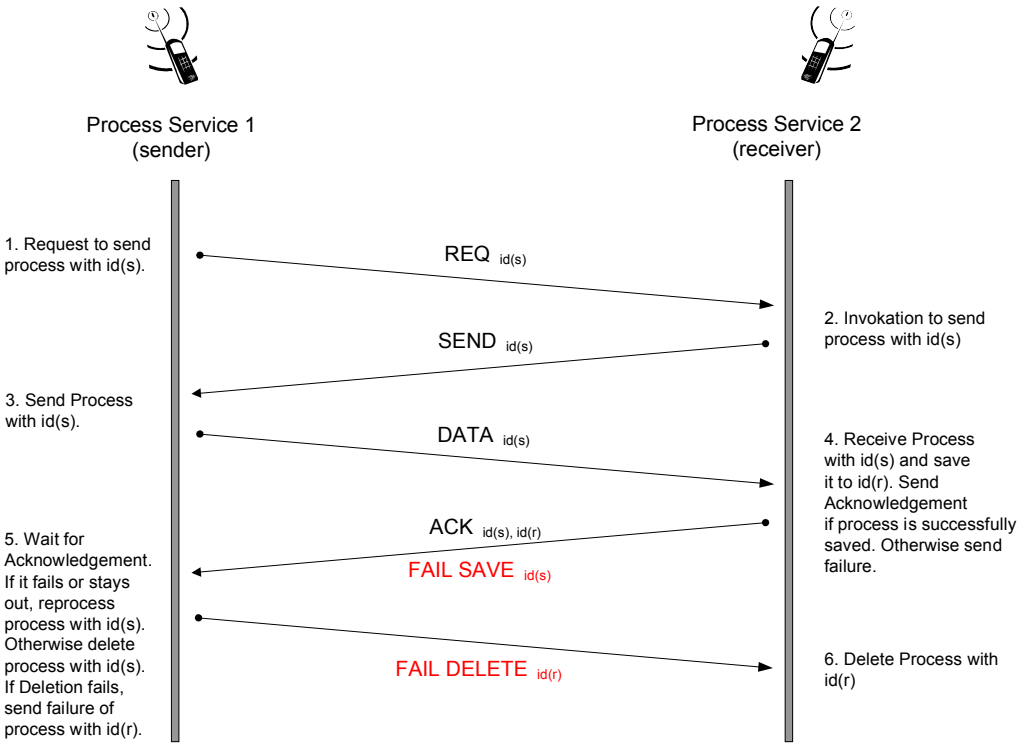


Abb. 52: Protokoll zur Weitergabe von Prozessbeschreibungen

Die Instanz des *Process Service*, die die Weitergabe der Prozessbeschreibung beabsichtigt, sendet zunächst einen Request an den potentiellen Empfänger, um seine Absicht, einen Prozess zu senden, zu vermitteln und zu prüfen, ob der Teilnehmer verfügbar ist (*REQ*). Als Identifikation für den Vorgang sendet er die *Id* mit, unter welcher der weiterzugebende Prozess beim Sender gespeichert ist (*Id(s)*).

Wenn der potentielle Empfänger verfügbar ist und am Protokoll teilnehmen kann, fordert dieser den Nachfrager auf, die Prozessbeschreibung mit der zuvor übertragenen *Id(s)* zu senden (*SEND*). Liegt seine Antwort innerhalb eines spezifizierten Timeouts, so lädt der Sender die Prozessbeschreibung mit der betreffenden *Id(s)* aus dem Speicher und schickt sie an den nun bestätigten Empfänger (*DATA*).

Der Empfänger speichert die erhaltene Prozessbeschreibung auf dem Gerät und weist ihm eine neue, bei ihm verfügbare *Id* zu (*Id(r)*). Ist dieser Vorgang erfolgreich, schickt er dem Absender des Prozesses ein Acknowledgement (*ACK*) mit der *Id* des Absenders und der neuen *Id*, unter der der Prozess beim Empfänger gespeichert wurde. Kann die Prozessbeschreibung jedoch nicht gespeichert werden, weil der freie Speicherplatz inzwischen belegt wurde oder ein anderer Fehler aufgetreten ist, so teilt der Empfänger dieses dem Absender durch ein *FAIL SAVE* mit.

Wenn der Absender der Prozessbeschreibung nach dem Senden der Daten innerhalb eines festgelegten Timeouts keine Antwort von seinem Kommunikationspartner bekommt, so nimmt er an, dass die

Übertragung fehlgeschlagen ist. Er bricht die Kommunikation ab und versucht, durch den *Context Service* einen anderen Prozessteilnehmer zu finden. Ebenso wird verfahren, wenn ein *FAIL SAVE* empfangen wird: Der Prozess wird wieder in die Reihe der noch weiterzugebenden Prozesse eingestellt. Bekommt der Absender jedoch eine Bestätigung vom Empfänger (*ACK*), dass die Übertragung erfolgreich war, so löscht er den Prozess aus seinem Speicher. Die Weitergabe der Prozessbeschreibung ist hiermit für den Sender erfolgreich abgeschlossen.

Kann der betreffende Prozess jedoch im Einzelfall nicht mehr gelöscht werden, so deutet dieser Fehler an, dass die Prozessbeschreibung gar nicht mehr im System vorhanden ist, also bereits fertig ausgeführt oder von einem anderen Teilnehmer übernommen wurde. Dieses Verhalten kann auftreten, wenn das *ACK* des Kommunikationspartners erst nach Ablauf des Timeouts eintrifft und die Prozessbeschreibung in der Zwischenzeit weiterverarbeitet wurde. In diesem Fall muss die Weitergabe des Prozesses gestoppt werden, um unerwünschte Duplikate von Prozessen zu vermeiden. Der Absender der Prozessbeschreibung fordert den Empfänger daher auf, seine Kopie des Prozesses zu löschen (*FAIL DELETE*). Er verwendet dazu die *Id*, unter welcher der Prozess beim Empfänger gespeichert wurde (*id(r)*). Wurde der Prozess nach einem verspätet eingegangenen *ACK* noch nicht an einen anderen Teilnehmer weitergegeben, sondern nur zur Weitergabe freigegeben, so wird der Prozess beim Absender abgebrochen, um den errungenen Fortschritt der Weitergabe zu erhalten und keine weitere unnötige Kommunikation zu verursachen.

5.2.3 Basiskomponente zur Ausführung von Prozessen

Die Basiskomponente erweitert den vorgestellten Kernbereich um eine Ausführungsumgebung für Prozesse und stellt damit die eigentliche Workflow Engine des *Process Service* dar. Dabei wird jeweils genau ein Prozess zur Zeit bearbeitet. Ziel der Basiskomponente ist es, die Ausführung eines Prozesses soweit wie möglich voranzubringen und die Prozessbeschreibung zur Weitergabe an die Kernkomponente zurückzureichen, falls eine vollständige Ausführung mit dem Potential des mobilen Geräts nicht möglich ist.

Die Basiskomponente kann nur gemeinsam mit der Kernkomponente eingesetzt werden und ist daher für Geräte mit einer höheren Leistung geeignet. Die Implementierung wurde im Prototypen für J2ME Personal Profile 1.0 und J2SE umgesetzt.

Abbildung 53 zeigt die Erweiterung des Klassendiagramms der Kernkomponente um die wichtigsten Klassen der Basiskomponente. Im Mittelpunkt von der Basiskomponente steht die Klasse *BaseExecutionService*, die die abstrakte Klasse *ExecutionService* der Kernkomponente erweitert. Da zur Ausführung eines Prozesses alle in der Prozessbeschreibung formulierten Details geladen werden müssen, verwendet der *BaseExecutionService* einen neuen XML-Parser, der im Gegensatz zum minimalen *ProcessParser* die *DPDL*-Beschreibungen vollständig interpretieren kann (*DPDL-Parser*). Zusätzlich prüft er die zu bearbeitende Prozessbeschreibung auf grundlegende Integrität, zum Beispiel darauf, ob aus Aktivitäten referenzierte Elemente und Daten auch wirklich definiert wurden. Zum Bezug der Strategien greift der *DPDL-Parser* auf den *StrategyParser* aus der Kernkomponente zurück, um bestehenden Code wiederzuverwenden.

5.2.3 Basiskomponente zur Ausführung von Prozessen

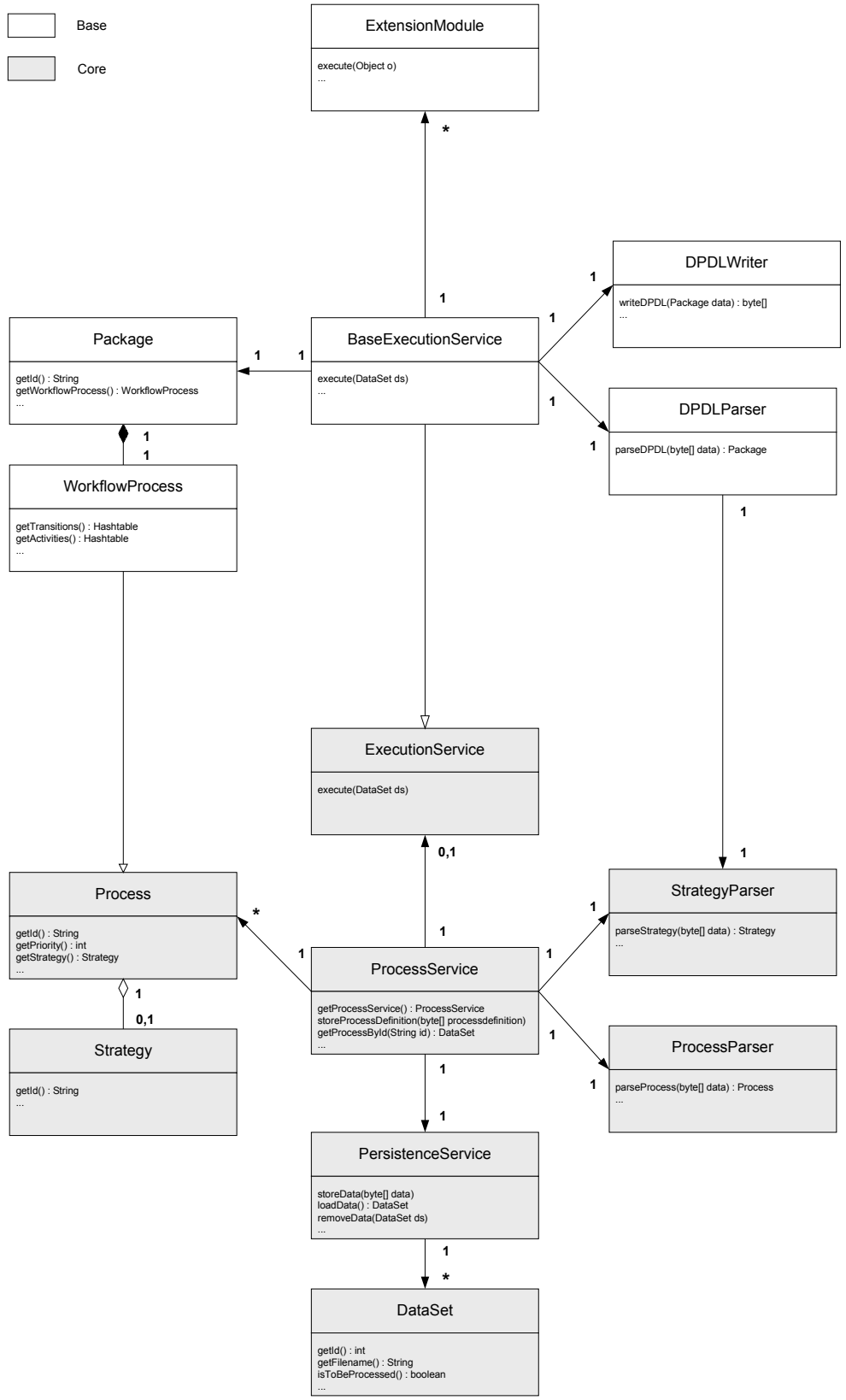


Abb. 53: Klassendiagramm der Basiskomponente

Wird der *BaseExecutionService* mit der Ausführung eines Prozesses beauftragt, kann der Prozess vollständig aus dem gelieferten *DataSet* geladen werden. Resultat ist ein Objekt der Klasse *Package*, welches alle Informationen eines *Packages* aus der Prozessbeschreibungssprache *DPDL* aufnehmen kann. So enthält es unter anderem den *WorkflowProcess*, der wiederum weitere Details der Prozessbeschreibung enthält und somit den in der Kernkomponente definierten *Process* fachlich erweitert. Im übrigen hält sich die objektorientierte Umsetzung der Konstrukte aus *DPDL* relativ eng an die entsprechenden XML-Elemente. Dies erleichtert den direkten Bezug zur Sprache und macht den Speicher- und Ladevorgang weniger rechenaufwändig.

Während der Ausführung eines Prozesses kommt es im Regelfall vor, dass Inhalte der Prozessbeschreibung verändert werden. Zum Beispiel können Daten durch die Ausführung von Aktivitäten verändert oder hinzugefügt werden. Außerdem kann der Kontrollfluss selbst modifiziert werden, indem eine Aktivität durch eine andere ersetzt wird. Zumindest aber wird der Bearbeitungszustand des Prozesses, welcher über das in Kapitel 5.1.3 erläuterte Zustandskonzept in der Prozessbeschreibung festgehalten wird, durch das Ausführen von Aktivitäten angepasst. Diese Änderungen müssen in der Prozessbeschreibung persistent gemacht werden, um den Prozess vor Datenverlust sowie vor Verlust von Fortschrittsinformationen zu schützen und ihn letztendlich an andere Prozessteilnehmer weitergeben zu können. Hierzu wird die Klasse *DPDLWriter* eingesetzt, die alle Informationen eines *Packages* wieder in ein XML-Dokument schreibt. So kann im Extremfall nach jeder Änderung die aktuelle Version der Prozessbeschreibung gespeichert und so einem möglichen Verlust vorgebeugt werden.

Wird der *BaseExecutionService* mit einem *Package* gestartet, so wird zunächst geprüft, ob der Ausführungszeitraum des Prozesses bereits erreicht ist und der Prozess noch gültig ist. Hierbei handelt es sich um die Abfrage der *DPDL*-Attribute *ValidFrom* und *ValidTo* im *ProcessHeader*. Wenn der Prozess ausgeführt werden darf, wird die *StartActivity* des Prozesses untersucht. Es kann sich hierbei um die eigentliche initiale Startaktivität des Prozesses handeln oder um dessen aktuellen Ausführungszustand. Ist die *StartActivity* bereits auf den Zustand *finished* gesetzt, so deutet dies darauf hin, dass die Aktivität bereits erfüllt wurde, die Weiterführung des Prozesses aber an der Ausführung einer Transitionsbedingung zwischen der Startaktivität und der darauf folgenden Aktivität gescheitert ist. In diesem Fall wird bei der Evaluation der Transitionsbedingung fortgefahren. Wenn die *StartActivity* jedoch den Zustand *ready* besitzt, kann der *BaseExecutionService* versuchen, die Aktivität auszuführen.

Bei der Ausführung einer Aktivität wird zunächst einmal festgestellt, um welche Art von Aktivität es sich handelt. Stellt die Aktivität einen Block dar und besteht sie damit ihrerseits aus weiteren Aktivitäten, so wird die *StartActivity* des Blocks ausgeführt. Mögliche Blockaktivitäten sind die *DPDL*-Konstrukte *ActivitySet*, *Loop* oder *Transaction*. Ist die Aktivität hingegen atomar, kann es sich nur um eine *RouteActivity* oder eine tatsächlich implementierte Aktivität (*Implementation*) handeln. *RouteActivities* dienen nur der Steuerung des Kontrollflusses und können sofort als ausgeführt betrachtet werden, sofern sie nicht mit weiteren Bedingungen oder Restriktionen verknüpft sind.

Eine *Implementation* kann entweder eine manuelle Aktivität darstellen, ein *Tool* implementieren oder als Platzhalter für einen Subprozess dienen. Für manuelle Aktivitäten wird eine spezielle Komponente zur Durchführung von Benutzerinteraktionen benötigt (vgl. 5.2.4). Die Spezifikation eines *Tools* lässt darauf schließen, dass eine Anwendung zur Ausführung der Aktivität benötigt wird. In diesem Fall wird der *Context Service* mit der Suche nach einem passenden Service für die Ausführung der Aktivität beauftragt. Dazu wird ihm die *UUID* der als *Tool* spezifizierten *Application* sowie eine gegebenen-

falls definierte *Strategy* zur Einschränkung von geeigneten Anwendungen zur Verfügung gestellt. Die Referenzierung erfolgt über die *Id* der Aktivität, für die die Anwendung gesucht wird.

Soll ein bestimmter, im Prozess spezifizierter Prozessteilnehmer eine Aktivität ausführen, so wird der geforderte *Participant* in die Anfrage an den *Context Service* einbezogen. Der *Context Service* muss in diesem Fall zunächst die Identität des eigenen Geräts oder die Identität des Benutzers feststellen. Stimmt die eigene Identität nicht mit der geforderten Identität überein, kann der *Context Service* die Suche nach einem geeigneten Dienst zur Ausführung der Aktivität abbrechen und dem *BaseExecutionService* durch ein entsprechendes Event mitteilen, dass er die Aufgabe nicht erfüllen darf. Der *BaseExecutionService* stoppt dann die Bearbeitung des Prozesses und gibt die Kontrolle zurück an den *ProcessService*. Als Parameter zur Weitergabe der Prozessbeschreibung an andere Teilnehmer setzt er den gesuchten *Participant* zusätzlich zur etwaigen Prozess-Strategie. Auf diese Weise kann die Suche nach Prozessteilnehmern mit konkreten Strategien vereint werden. Da angenommen wird, dass Benutzer in der Regel über mehrere mobile und stationäre Geräte verfügen, kann unter Umständen das passende Gerät für den jeweiligen Einsatzzweck ausgewählt werden.

Für die Suche nach einem geeigneten Dienst zur Ausführung der Aktivität steht dem *Context Service* nur begrenzt Zeit zur Verfügung. Die erlaubte Zeitdauer für eine Suche wird durch ein gerätespezifisches Timeout bestimmt. Anforderungen an das Timeout von Seiten des Prozesses können zum Beispiel durch die Spezifikation von nicht-funktionalen Aspekten gemacht werden. Wenn der *Context Service* keine passende Anwendung finden kann, bevor das Timeout abgelaufen ist oder der *Context Service* durch andere Aufgaben blockiert ist, wird die Bearbeitung des Prozesses gestoppt und der Prozess an ein anderes Gerät weitergegeben. Dies verhindert Verklemmungen zwischen dem *Process Service* und dem *Context Service* und hilft, den Prozess möglichst zügig abzuarbeiten.

Die Ausführung einer konkreten automatisierten Aktivität erfolgt über ein generisches *ServiceObject*. Codebeispiel 48 zeigt die Schnittstelle eines *ServiceObject*, welche als oberste Hierarchiestufe einer Implementierung nach dem sogenannten *Strategy Pattern* betrachtet werden kann. Ein *Strategy Pattern* ist ein objektbasiertes Verhaltensmuster, welches Klassen zur Kapselung verschiedener Algorithmen definiert [Gam02]. Es kann unter der Verwendung einer globalen Schnittstelle durch beliebige passende Algorithmen implementiert und somit generisch für alle Arten von unterstützten Anwendungsarten genutzt werden. Vorteilhaft ist dabei, dass der *BaseExecutionService* nicht mit vielen unterschiedlichen Anwendungen umgehen können muss, sondern das in der Schnittstelle vordefinierte Verhalten für den Aufruf eines Dienstes ausreichend ist [Gam02].

```
public interface ServiceObject {
        public DataField execute(FormalParameter[] fp, DataField[] df);
        public boolean exceptionOccured();
        public boolean connectionResetOccured();
}
```

Codebeispiel 48: Interface ServiceObject

Die Schnittstelle enthält eine Methode *execute*, welche die Ausführung durch den vom *Context Service* erbrachten Dienstleister der Aktivität aufruft. Dazu werden die für die Ausführung benötigten Varia-

blen mit Angabe ihres Typs (*DataFields*) und die spezifizierten *FormalParameters* der *Application* als Parameter übergeben. Ein *DataField* enthält dazu den Wert der Variablen und den Datentyp sowie die Angabe, ob es sich um ein Array handelt. Rückgabewert ist der hierfür definierte Ausgabeparameter vom Typ *DataField*. Ob eine Variable als Ein- oder Ausgabeparameter dienen soll, wird innerhalb der *FormalParameters* festgehalten. Daher ist es innerhalb eines konkreten *ServiceObject* notwendig, die Aufrufparameter in Form der *DataFields* den *FormalParameters* entsprechend ihrer Reihenfolge zuzuweisen. Für ein erfolgreiches Mapping zwischen *DataFields* und *FormalParameters* wird Typkompatibilität vorausgesetzt. Ein erneutes Mapping dieser Art kann notwendig werden, wenn das konkrete *ServiceObject* abweichende *FormalParameters* besitzt. An dieser Stelle können auch Typkonvertierungen im Rahmen der Typkompatibilität durchgeführt werden.

Die Abfrage von Fehlern und Verbindungsabbrüchen ist notwendig, um darauf entsprechend mit den in der Prozessbeschreibung spezifizierten Aktionen reagieren zu können.

Bei einem konkreten *ServiceObject* kann es sich um die Implementierung von Schnittstellen zu einer bestimmten Ausführungseinheit handeln, zum Beispiel zu einem *Web Service*, einem *CORBA*- oder *Java-RMI*-Objekt oder zu einer *RPC*-Prozedur. Abbildung 54 zeigt eine mögliche Ausgestaltung des *ServiceObject* nach dem *Strategy Pattern* für die genannten Anwendungsarten.

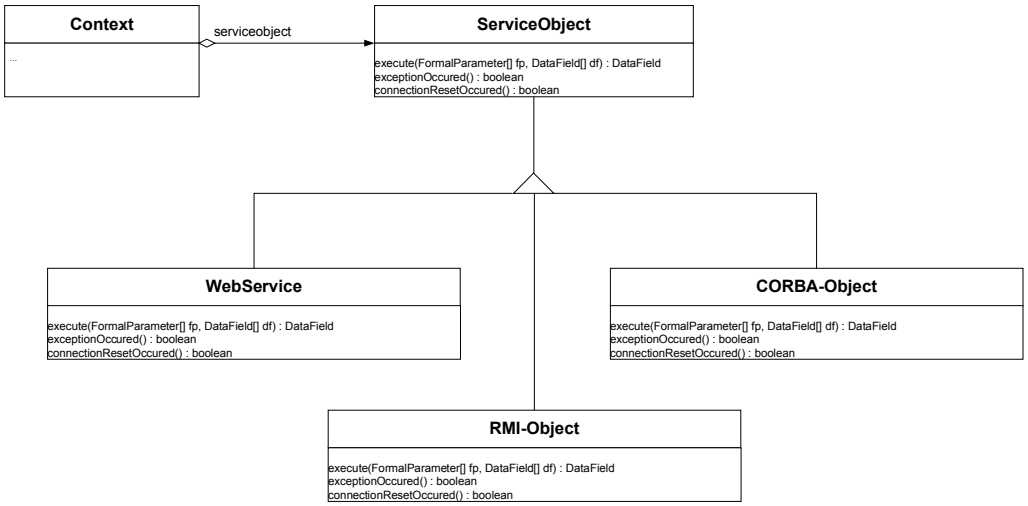


Abb. 54: Mögliche Implementierungen des ServiceObjects (nach [Gam02])

Für den Fall, dass der *Context Service* zwar keinen zur Ausführung der Aktivität geeigneten Dienst auffinden kann, jedoch die Möglichkeit besteht, die anstehende Aufgabe durch die Komposition mehrerer anderer Dienste zu erbringen, kann die Prozessbeschreibung in Bezug auf die nicht direkt ausführbare Aktivität geändert und diese durch eine Folge alternativer Aktivitäten mit dem gleichen Ergebnis ersetzt werden. Zum Beispiel könnte eine Aktivität A durch eine Blockaktivität A´ ersetzt werden, die die Teilaktivitäten A_1 und A_2 beinhaltet. Ebenso könnte eine Aktivität durch einen ganzen Subprozess ersetzt werden. Das Ausmaß der Änderbarkeit von Aktivitäten hängt von der Kennzeichnung des Attributes *Editable* ab. So kann der Modellierer des Prozesses jeglichen Eingriff verbieten (*Editable=NO*), nur Austauschbarkeit erlauben (*Editable=EDIT*), nur Löschen erlauben (*Editable=REMOVE*) oder alle Änderungen gestatten (*Editable=ALL*).

Soll eine Aktivität vollständig gelöscht werden, da sie nicht mehr benötigt wird, kann sie durch eine einfache leere *RouteActivity* ersetzt werden. Dies ist weniger rechenaufwändig, als eine Aktivität aus dem Kontrollfluss zu entfernen und ihre Eingangs- und Ausgangstransitionen zu einer gemeinsamen Transition zu verschmelzen. Außerdem können auf diese Weise spezifizierte Transitionsübergänge in der beabsichtigten Semantik erhalten werden. Ein Löschen oder Ersetzen von Aktivitäten, die für die Bildung von erweiterten Kontrollflussstrukturen wie *Splits* oder *Joins* beteiligt sind, ist nicht erlaubt. Die ursprünglichen Absichten des Initiators des Prozesses können bei einem derartig komplexen Eingriff nicht mehr gewährleistet werden.

Wie bereits beschrieben, kann eine Aktivität auch wieder einen Subprozess beinhalten, der anstelle der Aktivität ausgeführt werden soll (*SubFlow*). Da ein *Package* aus den genannten Gründen immer nur einen einzigen Prozess spezifiziert, muss der angegebene Subprozess zur Ausführung erst einmal aufgefunden werden. Der Prototyp der Basiskomponente kann nur Subprozesse ausführen, die asynchron zum Elternprozess ausgeführt werden können. Für Subprozesse, die synchronisiert werden müssen oder parallel zum Elternprozess bearbeitet werden sollen, wird ein *ExtensionModule* benötigt, welches die damit verbundenen zusätzlichen Verwaltungsaufgaben übernimmt.

Um den *Context Service* nicht mit unnötigen Aufgaben zu belasten, prüft der *BaseExecutionService* zunächst, ob sich der Subprozess im Speicher des mobilen Geräts befindet. Falls vorhanden, kann er über eine allgemeine Schnittstelle des *Process Services* in der Kernkomponente Prozesse anhand ihrer *Id* aus dem Speicher laden und als Subprozess integrieren. Ist der benötigte Subprozess nicht vorhanden, stellt der *BaseExecutionService* einen Request an den *Context Service*, mit der Aufgabe den Prozess auf anderen Geräten in der Umgebung zu suchen. Zum Beispiel könnte es sich so auch um einen öffentlich zugänglichen Subprozess handeln, der anhand einer festen *UUID* referenziert werden kann. Im Gegensatz zu einer *Id* ist eine *UUID* auch außerhalb einer Prozessbeschreibung eindeutig.

Wenn der *Context Service* den gesuchten Subprozess findet, kann er ihn über die allgemeine Schnittstelle der Kernkomponente in den Speicher des mobilen Gerätes laden, wo der *BaseExecutionService* diesen auffinden und ausführen kann. Dazu wird der *AccessLevel* von *private* auf *public* gesetzt, damit sich auch andere Prozessteilnehmer bei Bedarf an der Ausführung des Subprozesses beteiligen können. Nach Abschluss der Bearbeitung des Subprozesses kehrt der Kontrollfluss zum Elternprozess zurück.

Die Auswertung von Transitionsbedingungen geschieht nach einem ähnlichen Muster wie die Ausführung von implementierten *Activities*. Eine *Condition* stellt in *DPDL* die Ausführung einer Anwendung dar, deren Rückgabetyp ein Wahrheitswert ist, welcher den Transitionsübergang beeinflusst. Für den *BaseExecutionService* ist daher nicht zu erkennen, ob es sich bei der Bedingung um eine Evaluation von echten Kontextinformationen handelt, wie zum Beispiel das Vorliegen einer bestimmten Außentemperatur, oder ob eine Auswertung von prozessinternen Daten angefordert wird. Letzteres könnte zum Beispiel durch den einfachen Vergleich zweier *DataFields* spezifiziert sein. Stößt die Ausführungsumgebung bei der Bearbeitung des Kontrollflusses auf eine *Condition*, so sendet sie einen Request an den *Context Service*, um sie bei der Evaluation der Bedingung zu unterstützen. Anhand der in der Bedingung angegebenen *UUID* kann der *Context Service* die Operation der Bedingung semantisch auflösen und die Bedingung entweder selbst mit Hilfe der zur Verfügung stehenden Kontextinformationen evaluieren oder eine fremde Anwendung mit der Evaluation der Bedingung beauftragen.

Wenn die Bedingung ausgewertet werden kann, sendet der *Context Service* dem *BaseExecutionService* das Ergebnis. Kann die Bedingung mit allen zur Verfügung stehenden Mitteln nicht ausgewertet werden, muss die Bearbeitung des Prozesses abgebrochen und die Prozessbeschreibung an einen anderen Teilnehmer transferiert werden.

Bei einer erfolgreichen Evaluierung kann der *BaseExecutionService* mit dem Ergebnis weiterarbeiten und den Kontrollfluss in dessen Abhängigkeit fortsetzen. Im Falle eines negativen Ergebnisses leitet die Ausführungsumgebung eine *Dead Path Elimination* ein. Dabei werden alle Folgeaktivitäten auf dem betreffenden Pfad in den Zustand *skipped* versetzt, bis eine erneute Auswertung, zum Beispiel an einem *Join* erreicht wird.

5.2.4 Weitere optionale Zusatzkomponenten

Auch der *BaseExecutionService* würde viel zu komplex werden, wenn er alle möglichen Funktionen einer Ausführungsumgebung in sich vereinen würde. Er ist daher durch beliebige Zusatzkomponenten erweiterbar, die je nach Leistungspotential des jeweiligen mobilen oder stationären Systems an die Basiskomponente angeschlossen werden können. Die Zusatzkomponenten sind im Klassendiagramm (Abb. 53) durch die abstrakte Klasse *ExtensionModule* gekennzeichnet und können durch *Extension Modules* verschiedenen Typs implementiert werden.

Extension Modules erweitern die Funktionalität der Ausführungsumgebung in einem bestimmten Leistungsbereich. Im folgenden sollen beispielhaft besonders relevante Zusatzkomponenten vorgestellt werden. Neben den genannten Vorschlägen für *Extension Modules* sind jedoch beliebige weitere Ergänzungen denkbar. Ebenso können auch *Extension Modules* selbst wieder Erweiterungen implementieren, so dass sich eine möglichst flexible Architektur ergibt, die sich den Bedürfnissen und der Entwicklung der mobilen Systeme anpassen kann.

Zusatzkomponente Benutzerinteraktion

Die Portabilität als eine der Haupteigenschaften mobiler Geräte deutet darauf hin, dass es eine derer wichtigsten Aufgaben ist, Benutzer an beliebige Orte zu begleiten, um sie dort bei ihren Tätigkeiten unterstützen zu können. Im diesem Zusammenhang gibt es viele denkbare Einsatzmöglichkeiten, in denen eine direkte Interaktion des Benutzers auch von Seiten eines Prozess herbeigeführt werden muss. Neben der Ausführung von vollständig manuellen Aktivitäten kann es sich hierbei zum Beispiel um einfache Bestätigungen, Dateneingaben, Weiterleitung von Informationen oder um das Treffen von Entscheidungen handeln. Benutzer können außerdem über eine mobile Anwendung als Initiator eines Prozesses auftreten, wie im Anwendungsbeispiel (vgl. 2.5) beschrieben wurde. Schließlich können Fehlersituationen durch berechtigte Personen behandelt oder aufgelöst werden. Die Reichweite der

möglichen Benutzerinteraktionen erschließt sich also von einer simplen Präsentation von Meldungen bis hin zu komplexen Eingriffen in die Struktur eines Prozesses.

Für manuelle Aktivitäten und andere Benutzerinteraktionen muss das mobile Gerät, welches an der Verwaltung des Prozesses während der Ausführung der Aktivität beteiligt ist, jedoch zunächst einmal ein Benutzerinterface aufweisen, welches der vorliegenden Art der Interaktion angemessen ist. Soll zum Beispiel eine technische Zeichnung von einem Außendienstmitarbeiter überprüft werden, so wird ein graphisches Display von einer bestimmten Größe für diese Aktivität benötigt. Vor allem aus diesem Grund ist die Zusatzkomponente Benutzerinteraktion von den konkreten technischen Eigenschaften des mobilen Geräts abhängig. Weiterhin werden spezielle Anwendungen benötigt, die die Ausführungsumgebung bei der Kommunikation mit dem Benutzer unterstützen. Gegebenenfalls muss zusätzlich die Identität des Benutzers überprüft werden, um nur berechtigten Personen eine Teilnahme am Prozess zu gestatten.

Ein eng mit einer Komponente zur Benutzerinteraktion verknüpfte Zusatzfunktion ist die Möglichkeit zur erweiterten Fehlerbehandlung. Zum Beispiel könnte innerhalb einer *Strategy* festgelegt sein, ob Fehler automatisch oder durch Interaktion mit Benutzern behandelt werden sollen. Eine Interaktion mit Benutzern im Fehlerfall verspricht, Berechtigung und Kompetenz des Benutzers vorausgesetzt, eine verbesserte Aussicht auf die Lösung des Problems und eine rasche Wiederaufnahme des Prozesses. Benutzerzentrierte Konzepte wie zum Beispiel das *Prozessmuster* (vgl. 4.3.2) können die Vergegenständlichung des Prozesses für die Bearbeitung durch einen Anwender unterstützen und dem Benutzer so eine direkte Kontrolle über den Prozess ermöglichen.

Zusatzkomponente: Erweiterte Kontrollflusselemente

Unter erweiterten Kontrollflusselementen können alle diejenigen *DPDL*-Konstrukte zusammengefasst werden, deren Bearbeitung mit dem beschriebenen *BaseExecutionService* nicht möglich ist. Hierzu gehören vor allem die Ausführung und Verwaltung von parallelen Pfaden des Prozesses, deren Synchronisation an einem vorgegebenen Ort sowie die Ausführung von synchronen, parallelen Subprozessen.

Gelangt der Kontrollfluss innerhalb eines Prozesses an eine Aktivität, die als *Split* mehrere Ausgangstransitionen besitzt, so muss an dieser Stelle die Prioritäten der als nächstes auszuführenden Aktivitäten untersucht und diese dementsprechend bearbeitet werden. Zusätzlich muss entschieden werden, ob das verantwortliche mobile Gerät alle parallelen Pfade selbst ausführen soll oder ob arbeitsteilig Kopien der Prozessbeschreibung an andere Teilnehmer weitergegeben werden sollen. Gegebenenfalls ist eine echte parallele Ausführung der Aufgaben durch mehrere Geräte gar nicht möglich, da die auszuführenden Aktivitäten implizit auch untereinander noch Abhängigkeiten besitzen, zum Beispiel durch die gegenseitige Bereitstellung von Daten. Auf der anderen Seite könnte die Ausführung durch ein einziges Gerät nicht möglich sein, weil es nicht genug Ressourcen aufwenden kann, um die parallelen Pfade nebenläufig zu erfüllen und eine sequentielle Ausführung der Aufgaben durch weitere Beziehungen der parallelen Aktivitäten untereinander verhindert wird. Kapitel 5.4 zeigt hierfür ein Beispiel auf. Die Komponente, die den erweiterten Kontrollfluss steuert, muss da-

her eventuell auftretende Verklemmungsituationen erkennen und diese durch die Beibehaltung der Gesamtkontrolle oder durch die Abgabe von Aufgaben an andere Prozessteilnehmer lösen können.

Um schließlich bei der Weitergabe von Kopien eine unkontrollierte Bearbeitung des Prozesses durch andere Prozessteilnehmer zu vermeiden, darf jeder Verantwortliche den Prozess nur an einen einzigen anderen Teilnehmer weitergeben. Dabei müssen die initialen Aktivitäten der Kontrollflusspfade, für dessen Bearbeitung sich der Teilnehmer entschieden hat, mit dem Zustand *executing* gekennzeichnet werden.

Für eine Synchronisation von parallel ausgeführten Kontrollflusspfaden ist es notwendig, die an die verschiedenen Teilnehmer ausgegebenen Kopien der Prozessbeschreibung wieder zusammenzuführen. Im Idealfall verbirgt sich hinter der URI, welche die für die Synchronisation verantwortliche Einheit angibt, ein Dienst, der die ganze Prozessbeschreibung als Eingabeparameter akzeptiert und als Ergebnis eine einzige, synchronisierte Version der Prozessbeschreibung weitergibt. Zur Synchronisation gehören dabei sowohl die Auswertung von *Join*-Bedingungen als auch der Abgleich von Datenfeldern, die während der parallelen Ausführung geschrieben wurden.

Der Aufwand einer Synchronisation wird besonders umfangreich, wenn große, lang andauernde Prozesse mit einer Vielzahl von parallelen Pfaden zusammengeführt werden müssen. Unter Umständen muss das als Treffpunkt spezifizierte Gerät sehr lange warten, bis alle erwarteten Kopien der Prozessbeschreibung bei ihm eingegangen sind. Hierfür ist ein komplexes Verwaltungssystem mit einer angemessenen Speicherkapazität und einer hohen zeitlichen und räumlichen Verfügbarkeit nötig. Für die Realisierung einer Synchronisationskomponente bietet sich daher entweder die Verwendung eines leistungsfähigeren stationären Systems oder die Benutzung einer verteilten Registratur an, welche Informationen über zu synchronisierende Prozessbeschreibungen mit ihren Instanzen austauschen kann.

Auch wenn keine eigentliche parallele Bearbeitung von Prozessteilen stattfindet, kann trotzdem eine Synchronisation zwischen den Teilnehmern eines Prozesses notwendig werden. Soll zum Beispiel ein synchroner Subprozess an der Stelle einer Aktivität ausgeführt werden, der aus technischen Gründen nicht von der verantwortlichen Ausführungsumgebung selbst bearbeitet werden kann, so muss ein anderer Prozessteilnehmer die Ausführung des Subprozesses übernehmen. Das Ergebnis des Subprozesses muss dann nach Beendigung der Bearbeitung an das aufrufende System zurück übermittelt und in den Elternprozess integriert werden (Abb. 55).

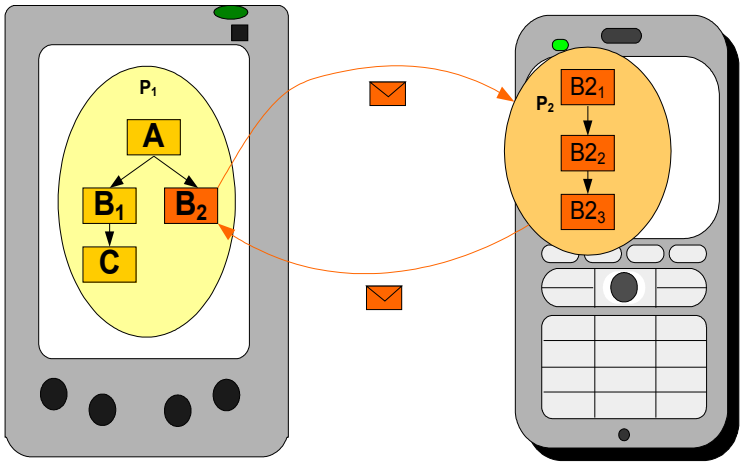


Abb. 55: Synchrone Ausführung von Subprozessen

Eine Synchronisation ist in diesem Fall direkt am mobilen System möglich, sofern dieses verfügbar ist und über eine entsprechende Synchronisationskomponente verfügt.

Zusatzkomponente: Transaktionen

Durch das Konstrukt *Transactions* in *DPDL* sollen komplexe Umschreibungen des transaktionalen Verhaltens vermieden und zusammengehörigen Aktivitäten als Transaktion gekennzeichnet werden. Gegebenenfalls können Aktivitäten zur Kompensation angegeben werden, die ausgeführt werden, wenn die Transaktion scheitert und ein *Rollback* der Einzelaktivitäten nicht mehr möglich ist.

Eine besondere Problematik bei der Ausführung von Transaktionen besteht durch die Dynamik des Prozesses in Hinblick auf den möglichen Ein- und Austritt von Teilnehmern, den Wechsel von Koordinatoren und die Änderung des Prozesses selbst während der Transaktion. Zur Durchführung einer Transaktion ist daher eine flexible verteilte Koordination notwendig, die einerseits die Dynamik einer sich ändernden Transaktionssituation bewältigen und andererseits mit der Verwaltung von länger andauernden Vorgängen umgehen kann.

Ziel des Ansatzes eines Zusatzmoduls für Transaktionen ist es, die Verarbeitung von zusammengehörigen Aktivitäten möglichst aus der Prozessbeschreibung zu extrahieren. Die Sicherstellung des transaktionalen Verhaltens und dessen konkrete Umsetzung obliegt damit der jeweiligen Ausführungsumgebung, die je nach Möglichkeit die Ausführung koordinieren und überwachen kann. Nach Bearbeitung einer Transaktion soll das Ergebnis der Transaktion, also die Bestätigung einer erfolgreichen Durchführung oder das Auftreten eines Fehlers, an den Prozess zurückgegeben werden. So erhält der Prozess wieder die Kontrolle darüber, was als nächstes geschehen soll, unabhängig davon, ob eine Transaktion fehlschlägt oder erfolgreich ist. Die Ausführungsumgebung kann gegebenenfalls Kompensationsmaßnahmen einleiten und der Prozess kann fortgeführt werden, es sei denn, die Fehlersituation erfordert einen endgültigen Abbruch der Bearbeitung.

Zusatzkomponente: Sicherheit

Besondere Probleme bei der Ausführung verteilter Prozesse ergeben sich durch mangelndes Vertrauen zwischen einander fremden Prozessteilnehmern. Autonomie-Rechte müssen bewahrt und die Privatsphäre aller Benutzer geschützt werden. Insbesondere Geschäftstransaktionen sind in den meisten Fällen streng vertraulich zu behandeln und erfordern gegebenenfalls einen abgeschlossenen Nutzerkreis.

Um der Gefahr des Abhörens von Daten, des Missbrauchs in Folge von Diebstahl des mobilen Gerätes oder der ungewollten Modifikation von Prozessbeschreibungen wirkungsvoll entgegenzutreten, werden erweiterte Sicherheitsmechanismen benötigt. Insbesondere werden eine Verschlüsselung der Kommunikation und eine Authentifizierung von Geräten und Benutzern benötigt, um sicherheitsbe-

dürftige Daten und Aktivitäten zu schützen. Zusätzliche Checksummen können das Risiko der böswilligen oder unbeabsichtigten Veränderung von Daten während der Übertragung verringern.

Bei der Spezifikation von Sicherheitsanforderungen über nicht-funktionale Aspekte kann entschieden werden, ob nur ein Teil der Prozesses oder einzelne Daten geschützt werden sollen, wie zum Beispiel die Angabe einer Kreditkartennummer, oder ob sich die Sicherheitsanforderungen auf den Prozess selbst und seine Integrität beziehen sollen.

Soll zum Beispiel der ganze Prozess verschlüsselt werden, so können die Sicherheitsanforderungen in einer Prozessstrategie definiert und Teilnehmer ohne ausreichende Sicherheitsmechanismen von der Bearbeitung ausgeschlossen werden. In diesem Fall können nur noch Geräte am Prozess teilnehmen, die über die spezifizierte Sicherheitskomponente verfügen. Wenn jedoch auch schwächeren Teilnehmern die Übernahme des Prozesses zwecks Weiterleitung an einen anderen Ausführungspartner ermöglicht werden soll, so müssen die Strategie selbst und die weiteren von der Kernkomponente zur Verwaltung des Prozesses benötigten Daten lesbar bleiben bzw. anonymisiert werden. Da die Kernkomponente nur Informationen von einer allgemein geringen Sicherheitsrelevanz verarbeitet, wie zum Beispiel die Priorität eines Prozesses, so kann mit dieser Vorgehensweise ein in vielen Anwendungsfällen praktikabler Kompromiss zwischen Sicherheitsbedürfnis und flexibler Ausführung des Prozesses geschlossen werden.

Letztendlich muss nicht nur der Prozess selbst und die Interessen seines Initiators geschützt werden, sondern auch der einzelne Betreiber des mobilen Gerätes, der die Ausführung von fremden Prozessen auf seinem System gestattet. Neben dem Schutz seiner persönlichen Daten muss dieser durch eine erweiterte Sicherheitskomponente vor ungewollten Aus- und Nebenwirkungen der Prozessausführung sowie durch die fälschliche Ausführung eines Prozesses in seinem Namen bewahrt werden.

5.3 Validierung des gewählten Ansatzes anhand des Anforderungskataloges

In diesem Abschnitt wird untersucht, inwieweit der dargestellte Ansatz von *DPDL* mit Unterstützung durch eine entsprechende Ausführungsumgebung die Anforderungen an eine Prozessintegration in mobile Systeme erfüllen kann. Neben einer Überprüfung des Ergebnisses sollen hierbei potentielle Weiterentwicklungsmöglichkeiten aufgezeigt werden. Zur besseren Vergleichbarkeit wird der in 3.4.4 spezifizierte Anforderungskatalog herangezogen.

Tabelle 15 fasst die genannten Aspekte des Anforderungskataloges sowie die Realisierung durch *DPDL* und die Ausführungsumgebung zusammen. Dabei werden die Anforderungen wieder nach ihrer Relevanz gewichtet und den bereits bestehenden Konzepten von *XPDL* gegenübergestellt.

	Anforderung	Stark	Opt.	Un-real.	XPDL	**DPDL**
D1	Beschreibung und Ausführung komponierter automatisierter Services	X(*)			+	+
D2	Weitergabe von Prozessen an andere Ausführungseinheiten	X(*)			-	+
D3	Auswahl von Prozessteilnehmern durch benutzerdefinierte nicht-funktionale Kriterien	X(*)			-	+
D4	Kompatibilität zu bestehenden Ansätzen		X(*)			+/-
P1	Kontrollflusskonstrukte	X			+	+
P2	Datenflussbeschreibung	X			+	+
P3	Spezifikation von Prozessteilnehmern	X			+	+
P4	Beschreibung von Fehlerbehandlungsmaßnahmen		X		-	+
P5	Dauerhafte Verfügbarkeit des Prozess-Systems			X		-
P6	Skalierbarkeit		X		+	+
P7	Erweiterbarkeit		X		+	+
P8	Sicherheit		X		-	-
P9	Unterstützung von Transaktionen		X		-	+
P10	Plattformunabhängigkeit	X			+	+
P11	Zentrales System zur Administration			X		-
P12	Audit Trail		z.T.		+	+
M1	Geringer Speicherverbrauch	X			+	+
M2	Geringer Anspruch an Rechenleistung	X			+	+
M3	Verantwortlicher Umgang mit Energieressourcen		X			-
M4	Lokale Datenhaltung		X		+	+
M5	Geringer Kommunikationsaufwand	X			+/-	+/-
M6	Behandlung von Verbindungsabbrüchen	X			-	+
M7	Synchronisation		X		(-)	+
M8	Priorisierung		X		+	+
M9	Ergänzende Sicherheitsmechanismen		X		-	-
M10	Auswahl von variablen Diensten und Prozessteilnehmern erfolgt zur Laufzeit	X (*)			+	+
M11	Benutzerintegration		X		+	+
M12	Verteilte Administration des Prozesses	X			-	+
M13	Mächtigkeit der Prozessbeschreibungssprache		X		+	+

X Forderung

X(*) Forderung aus dem DEMAC-Projekt

\+ Wird durch die Prozessbeschreibungssprache unterstützt

\- Wird nicht durch die Prozessbeschreibungssprache unterstützt

+/- Nicht bewertbar

Tabelle 15: Analyse von DPDL auf Eignung für den Einsatz auf mobilen Systemen

Da *DPDL* an die Konstrukte und das Meta-Modell von *XPDL* angelehnt ist, werden bereits eine ganze Reihe von Eigenschaften an *DPDL* vererbt, die zur Erfüllung von allgemeinen Anforderungen an eine Prozessbeschreibungsprache geeignet sind. Hierzu zählen vor allem die in vielen Workflow-Sprachen angewandten Kontroll- und Datenflusskonstrukte, die Beschreibung von Prozessteilnehmern, die Ausführung automatisierter Dienste sowie die Möglichkeiten zur Benutzerintegration und zur Priorisierung von Aufgaben. Die insbesondere auch für mobile Systeme relevante Fehlerbehandlung wurde in *DPDL* durch das explizite Konstrukt *Exception* umgesetzt. Ein sehr begrenztes *Audit Trail* steht durch die Möglichkeit zum *Logging* von Prozessen und Daten bereit. Transaktionen können in *DPDL* als solche gekennzeichnet und mit speziellen Eigenschaftswerten versehen werden. Eine Ausführung sowohl erweiterter Fehlerbehandlungsmaßnahmen als auch die Überwachung und Durchführung von Transaktionen muss jedoch durch die Ausführungsumgebung selbst vorgenommen werden, so dass insbesondere in diesem Bereich ein enger Zusammenhang zwischen der Prozessbeschreibungsprache und der Ausführungsumgebung besteht. Für das *Logging* von Prozessen wird zusätzlich ein Archiv zur Sammlung der Daten benötigt.

Im Bereich der Anforderungen aus dem DEMAC-Projekt kann *DPDL* durch spezielle Konzepte an die Bedürfnisse der mobilen Systeme angepasst werden. Die geforderte Weitergabe an andere Prozessteilnehmer und somit die verteilte Administration des Prozesses wird durch die Integration eines Zustandskonzepts realisiert, welches die Speicherung des aktuellen Ausführungszustands des Prozesses und die Übertragung an andere Ausführungseinheiten erlaubt. Wie in den Anforderungen beschrieben, kann die Auswahl von Prozessteilnehmern dabei durch die Formulierung von nicht-funktionalen Aspekten eingeschränkt und entsprechend der Interessen des Prozessinitiators durchgeführt werden. Beliebige nicht-funktionale Aspekte können durch die Beschreibung von *Strategies* sowohl für die Weitergabe des Prozesses als auch für die Ausführung einzelner Aktivitäten definiert werden.

Die Kompatibilität zu *XPDL* ist jedoch durch die zahlreichen Erweiterungen einerseits und die benötigten Einschränkungen andererseits nur noch für eine Teilmenge an Prozessbeschreibungen gewährleistet. *DPDL* kann aber zum Beispiel durch ein XSLT-Mapping wieder in *XPDL* transformiert werden. Für zentrale Ausführungseinheiten irrelevante Zustandsbezeichnungen und nicht-funktionale Anforderungen können dann in vielen Fällen aufgegeben werden, ohne den fachlichen Sinn der Prozessbeschreibung zu verändern.

Spezielle Anforderungen, die aus der Mobilität der Prozessteilnehmer abgeleitet wurden, können in DPDL größtenteils durch die Nutzung ergänzender Konstrukte erfüllt werden. Unter den Gesichtspunkten der Skalierbarkeit und des Speicherplatzbedarfs für die Prozessbeschreibung ist das Konzept der Wiederverwendbarkeit von Prozessteilen ein wichtiger Aspekt. Die einzelne Deklaration von Aktivitäten, Daten, Prozessteilnehmern und Fehlerbehandlungsmaßnahmen benötigt zwar auf den ersten Blick einen Mehraufwand an Beschreibungen, dieser wirkt sich jedoch bei komplexeren Definitionen positiv auf den benötigten Speicherplatz aus. Weitere Erleichterungen ergeben sich dadurch, dass nur ein einziger Prozess innerhalb eines *Packages* definiert und die Semantik von Aktivitäten durch die Angabe von *UUIDs* externalisiert werden kann. Die geringe Leistungsfähigkeit von mobilen Geräten und deren knappe Ressourcen werden durch einen modularen Aufbau der Ausführungsumgebung berücksichtigt.

Die konkrete Belastung eines mobilen Geräts in Bezug auf benötigten Speicherplatz, Rechenkapazität und Kommunikationsbandbreite hängt jedoch unmittelbar von der Komplexität des Prozesses und der

Vielzahl der definierten Aktivitäten ab. *DPDL* kann nicht gewährleisten, dass der mit den Mitteln der Prozessbeschreibungssprache formulierte Prozess für den Einsatz auf mobilen Geräten sinnvoll oder ausführbar ist. Dies obliegt dem Anwender bzw. dem Modellierer, der den Prozess entwickelt und initiiert.

Eine besondere Herausforderung stellt in diesem Zusammenhang die Diskrepanz zwischen der Einsparung von Speicherplatz und der Vermeidung von zusätzlichem Kommunikationsaufkommen dar. Sowohl die Suche nach geeigneten Prozessteilnehmern als auch der Bezug von Verbindungsinformationen zu konkreten Diensten oder die externe Datenhaltung sind ohne verfügbares Kommunikationsmedium nicht zu bewältigen. Die Strategie der Externalisierung entlastet einerseits den Speicher des mobilen Geräts und erbringt ein Höchstmaß an Flexibilität. Zum Beispiel können Daten vom Kontrollfluss getrennt und bei Bedarf durch eine *External Reference* bezogen werden. Diese Vorgehensweise wirkt sich jedoch negativ auf das Kommunikationsaufkommen aus. Die Anforderung, das Kommunikationsaufkommen möglichst gering zu halten, kann daher nicht in jedem Fall vollständig erfüllt werden. Im konkreten Fall muss anwendungsabhängig entschieden werden, welche Vorgehensweise erfolgversprechender ist und ob die Entlastung der Speicherkapazität auf Kosten eines Mehraufwandes an Kommunikation vertreten werden kann.

Da ein Mindestmaß an Kommunikation für die Ausführung von mobilen Prozessen unerlässlich ist, kommt der Anforderung nach der Behandlung von Verbindungsabbrüchen ein besonderer Stellenwert zu. *DPDL* unterstützt deren Behandlung mit dem Konstrukt des *Connection Reset Handlers*, durch den das gewünschte Verhalten im Fall eines Verbindungsabbruchs definiert werden kann.

Für die Synchronisation von parallelen Kontrollflusspfaden und Subprozessen kann in *DPDL* ein eindeutiger Treffpunkt definiert werden, an dem alle Prozessteilnehmer ihre Prozessbeschreibungen zur Synchronisation übergeben müssen. Eine Synchronisation von Daten erfolgt nicht durch *DPDL* und muss bei Bedarf von einer speziellen Zusatzkomponente der Ausführungsumgebung wahrgenommen werden. *DPDL* stellt jedoch die Möglichkeit zur Verfügung, Daten bei Bedarf mit einem Zeitstempel zu versehen, um so im Konfliktfall feststellen zu können, welches die neuere Version des betreffenden Datums ist.

Wie *XPDL* ist auch *DPDL* durch *Extended Attributes* für spezielle Einsatzzwecke erweiterbar. Eine zusätzliche Erweiterbarkeit erfährt *DPDL* durch die Möglichkeit, mittels nicht-funktionaler Aspekte beliebige Zusatzkomponenten zu fordern, zum Beispiel ein optionales Modul zur Verschlüsselung der Prozessbeschreibung. Durch diese Vorgehensweise können fehlende konkrete Sprach- und Ausführungsbestandteile integriert und genutzt werden. Insbesondere im Bereich der Sicherheitsmechanismen besteht in dieser Hinsicht jedoch noch Entwicklungsbedarf. Sicherheitskriterien können in *DPDL* zwar über *Strategies* definiert und zur Verarbeitung gefordert werden, jedoch fehlt es an einer Umsetzung im Bereich der Ausführungsumgebung. Abhilfe könnte hier durch das genannte optionale Zusatzmodul zur Durchsetzung von Sicherheitsmechanismen geschaffen werden.

Ein schonender Umgang mit den Energieressourcen der Ausführungsumgebung muss in erster Linie durch die Ausführungsumgebung selbst erfolgen und hat daher keine in *DPDL* integrierten Konzepte.

5.4 Ausführung des Anwendungsbeispiels

In diesem Abschnitt soll das zu Beginn der Arbeit vorgestellte Anwendungsbeispiel (vgl. 2.5) mit Hilfe der Prozessbeschreibungssprache *DPDL* umgesetzt und in Auszügen erläutert werden, wie es mit Hilfe der Anwendungsumgebung praktisch ausgeführt werden kann. Dazu sei angenommen, dass die Anwendung auf dem Mobiltelefon des Versicherungsnehmers aus dessen Benutzereingaben einen Prozess generiert, der dem *Process Service* der DEMAC-Middleware zur Verarbeitung übergeben wird. Die vollständige Prozessbeschreibung hierzu ist im Anhang aufgeführt (Anhang 3).

Die Prozessbeschreibung des Anwendungsfalls besteht zunächst einmal aus einem *Package*, welches den Prozess und alle dazugehörigen Daten und Details enthält. Codebeispiel 49 veranschaulicht die Package-Informationen, wie *Id* und *Owner*, die sich zum Beispiel aus den gemeinsamen Initiatoren Versicherung und Versicherungsnehmer zusammensetzen können.

```
<Package Id="ALLIANCE Schadenschnellhilfe 1234-56789,0001"
Name="Schadenschnellhilfe" DPDLVersion="1.0"
Owner="ALLIANCE, Versicherten-Nr.1234-56789">
```

Codebeispiel 49: Anwendungsbeispiel: Package-Deklaration

Als mögliche nicht-funktionale Aspekte wurde im Anwendungsbeispiel festgelegt, dass die Reparaturzeit des beschädigten Fahrzeuges unter 30 Stunden liegen sollte. Diese Anforderung kann im Prozess als *Strategy* ausgedrückt werden, die zusammen mit der Referenz auf die Aktivität der Schadensbehebung am Fahrzeug aufgerufen wird (Codebeispiel 50).

```
<Strategies>
   <Strategy Id="Werkstattanforderungen">
      <StrategyProperty Id="1">
         <Requirements>
            <Requirement Name="MaxReparaturzeit" Value="30"/>
         </Requirements>
      </StrategyProperty>
   </Strategy>
</Strategies>

...

<ActivityRef ActivityId="Fahrzeug_reparieren" Id="Fahrzeug_reparieren" State="INACTIVE"
Editable="ALL" StrategyId="Werkstattanforderungen"/>
```

Codebeispiel 50: Anwendungsbeispiel: Strategy

Als nächstes werden die deklarierten Datentypen in der Prozessbeschreibung aufgeführt. Beispielhaft sei hier die Definition des komplexen Datentyps „GPSData“ angegeben. Zur Vereinfachung sei ange-

nommen, dass das Ergebnis einer GPS-Abfrage als Position in Form von geographischen Längen- und Breitenangaben vorliegt (Codebeispiel 51).

```
<TypeDeclarations>
  <TypeDeclaration Id="GPSData">
    <SchemaType>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xsd:element name="GPSData">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Latitude" type="xsd:string"/>
              <xsd:element name="Longitude" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:schema>
    </SchemaType>
  </TypeDeclaration>
</TypeDeclarations>
```

Codebeispiel 51: Anwendungsbeispiel: TypeDeclaration

Für den Anwendungsfall lassen sich weiter drei konkrete Prozessteilnehmer identifizieren, die an der Ausführung von Aufgaben beteiligt sein sollen: Der Versicherungskunde, die Versicherung und ein Anwender in der Rolle eines KFZ-Sachverständigen (Gutachter). Codebeispiel 52 zeigt die Definition des konkreten Prozessteilnehmers „Kunde“.

```
<Participants>
  <Participant Id="Kunde">
    <Devices>
      <Device Id="Mobiltelefon:1234-56789">
        <UUID>12311111901234567890123456789012</UUID>
      </Device>
    </Devices>
    <Human>Versicherter 1234-56789</Human>
  </Participant>

  ....

</Participants>
```

Codebeispiel 52: Anwendungsbeispiel: Participants

Applications stellen die Vorlagen für aufzurufene Anwendungen dar, die eine bestimmte Art von Aktivität erfüllen können. Die *Application* „GPS-Service“ besagt dabei über ihre *UUID*, dass ein Service zum Bezug von GPS-Daten gesucht werden muss. Als Ergebnis der Anfrage wird ein Ausgabeparameter vom komplexen Typ „GPS-Data“ erwartet (Codebeispiel 53). Wenn der konkrete gefundene Dienst

einen anderen Datentyp zurückgibt, muss versucht werden, diesen durch eine Konvertierung oder ein Mapping in das gewünschte Format zu bringen. Eingabeparameter liegen im Beispielfall nicht vor.

```
<Applications>
   <Application Id="GPS-Service">
      <UUID>12345678901234567890123456789012</UUID>
      <FormalParameters>
         <FormalParameter Id="Data" Mode="OUT">
            <DataType>
               <DeclaredType Id="GPSData"/>
            </DataType>
         </FormalParameter>
      </FormalParameters>
   </Application>

...

</Applications>
```

Codebeispiel 53: Anwendungsbeispiel: Applications

Die Daten des Anwendungsbeispiels, die als Eingabe- und Ausgabeparameter im Prozess benötigt werden, werden in dem Element *DataFields* zusammengefasst. Codebeispiel 54 zeigt die Definition des *DataFields* „VersichertenNummer“, welches aus einem elementaren Datentyp „String“ und dem Anfangswert in Form der Versichertennummer des Versicherungsnehmers besteht sowie die Deklaration einer Variablen mit dem Namen „GPSDaten“ vom komplexen Typ „GPSData“.

```
<DataFields>

   <DataField Id="VersichertenNummer" IsArray="false">
      <DataType>
         <BasicType Type="String"/>
      </DataType>
      <InitialValue>1234-56789</InitialValue>
   </DataField>

   <DataField Id="GPSDaten" IsArray="false">
      <DataType>
         <DeclaredType Id="GPSData"/>
      </DataType>
   </DataField>

   ...

</DataFields>
```

Codebeispiel 54: Anwendungsbeispiel: DataFields

DataFields können sich im Laufe der Prozessausführung verändern und können auch Referenzen auf entfernte Daten beinhalten. Nachdem zum Beispiel der Polizeibericht angefertigt worden ist und die Aktivität „Polizei rufen“ somit abgeschlossen wurde, könnte das *DataField* „Polizeibericht“ anstelle des umfangreichen Textes eine externe Referenz auf das angefertigte Dokument besitzen (Codebeispiel 55).

```
<DataField Id="Polizeibericht" IsArray="false">
   <DataType>
      <BasicType Type="String"/>
   </DataType>
   <ExternalReference
      Location="http://polizeiberichte.de/polizeibericht123456789.doc">
   </ExternalReference>
</DataField>
```

Codebeispiel 55: Anwendungsbeispiel: DataFields

Nach der Definition der Daten folgt die Spezifizierung des Kontrollflusses. Alle den Kontrollfluss betreffenden Angaben befinden sich im Element *WorkflowProcess*. Der *WorkflowProcess* enthält hierzu eine *StartActivity* (Codebeispiel 56).

```
<WorkflowProcess StartActivity="Schadensfall_melden" Id="1">
```

Codebeispiel 56: Anwendungsbeispiel: WorkflowProcess

Der *WorkflowProcess* legt die Aktivitäten und Transitionen des Prozesses fest. Da der Prozess der Beispielanwendung direkt mit einer Aufteilung des Kontrollflusses in zwei parallele Pfade beginnt, ist die Startaktivität „Schadensfall melden“ eine *RouteActivity*, die benötigt wird, um den *Split* zu definieren. Da alle durch die Transitionen angegebenen Folgeaktivitäten ausgeführt werden sollen, wird ein *AND-Split* formuliert. Die beiden entstehenden parallelen Pfade werden jedoch nicht wieder zusammengeführt, so dass die Ausführung unabhängig voneinander stattfinden kann und keine Konstrukte zur Synchronisation benötigt werden.

```
<Activity Id="Schadensfall_melden">
   <Route/>
   <TransitionRestriction>
      <Split Type="AND"/>
   </TransitionRestriction>
</Activity>
```

Codebeispiel 57: Anwendungsbeispiel: RouteActivity

Alle übrigen Aktivitäten des Prozesses enthalten eine konkrete Aufgabe, die ausgeführt werden soll. Diese sind daher als *Implementations* definiert. Komplexe Aktivitäten wie Blöcke kommen im Anwendungsbeispiel nicht vor. Codebeispiel 58 zeigt die Aktivität „GPSDaten ermitteln“ und referenziert dazu die bereits beschriebene *Application* „GPS-Service“. Als Parameter, der den Rückgabewert der Anwendung aufnehmen soll, wird die konkrete Variable „GPSDaten“ abgegeben. Falls „GPS-Daten“ einen initialen Wert hat, so wird dieser bei der Ausführung der Aktivität mit dem neuen Wert überschrieben.

```
<Activities>
  <Activity Id="GPSDaten_ermitteln">
    <Implementation>
      <Tool ApplicationId="GPS-Service">
        <ActualParameters>
          <ActualParameter>GPSDaten</ActualParameter>
        </ActualParameters>
      </Tool>
    </Implementation>
    <TransitionRestriction>
      <Split Type="AND"/>
    </TransitionRestriction>
  </Activity>

  ...

<Activities>
```

Codebeispiel 58: Anwendungsbeispiel: Implementation

Zusätzlich definiert die Aktivität „GPSDaten ermitteln“ einen *AND-Split*, der den Kontrollfluss in drei parallele Pfade aufspaltet. Nachfolgende Aktivitäten der Aktivität können jedoch erst dann ausgeführt werden, wenn die Aktivität abgeschlossen ist und die benötigten Daten vorliegen.

Ein Beispiel für die Darstellung möglicher Verklemmungssituationen des Prozesses stellt die Ausführung der parallelen Pfade beginnend mit den Aktivitäten „Abschleppdienst beauftragen“ und „Leihwagen mieten“ dar. Geht man davon aus, dass die Aktivität „Leihwagen mieten“ solange andauert, wie der Versicherungsnehmer den Leihwagen benötigt, da ja am Ende der Aktivität die Kosten in Abhängigkeit von der Nutzungsdauer des Fahrzeuges berechnet werden müssen, so würde die Aktivität erst durch das Ereignis „Versicherungsnehmer informieren“ beendet werden können, welches auf einem parallelen Kontrollflusspfad liegt. Zur Ausführung dieses Pfades müsste jedoch zunächst die Aktivität „Leihwagen mieten“ beendet werden, damit Ressourcen für dessen Bearbeitung frei werden. Es ist also notwendig, zumindest einen der beiden parallelen Kontrollflusspfade zur Ausführung an ein anderes Gerät abzutreten. Im Beispielfall übernimmt das mobile Gerät des Abschleppdienstes eine Kopie des Prozesses.

Schließlich folgt in der Prozessbeschreibung die Spezifikation der im Prozess eindeutigen *Activity References* mit der Festlegung des Ausführungszustandes einer Aktivität sowie die Definition der *Transitions*. Codebeispiel 59 zeigt alle am Beispielprozess beteiligten Aktivitäten als Referenz und den Kontrollfluss zwischen ihnen als Transitionen. Zusätzlich sind Beispiele dafür angegeben worden,

ob Aktivitäten modifiziert werden dürfen. Aktivitäten, die besonderen Einfluss auf den Kontrollfluss des Prozesses sind, wie zum Beispiel Synchronisationspunkte, dürfen nicht verändert werden. Vor dem Beginn der Ausführung des Prozesses haben zudem alle Aktivitäten den initialen Zustand *inactive*.

```
<ActivityRefs>
   <ActivityRef ActivityId="Schadensfall_melden" Id="Schadensfall_melden"
                State="INACTIVE" Editable="NO"/>
   <ActivityRef ActivityId="GPSDaten_ermitteln" Id="GPSDaten_ermitteln"
                State="INACTIVE" Editable="NO"/>
   <ActivityRef ActivityId="Versicherung_informieren" Id="Versicherung_informieren"
                State="INACTIVE" Editable="NO"/>
   <ActivityRef ActivityId="Polizei_rufen" Id="Polizei_rufen" State="INACTIVE"
                Editable="ALL"/>
   <ActivityRef ActivityId="Fahrzeug_abschleppen" Id="Fahrzeug_abschleppen"
                State="INACTIVE" Editable="ALL"/>
   <ActivityRef ActivityId="Leihwagen_mieten" Id="Leihwagen_mieten"
                State="INACTIVE" Editable="ALL"/>
   <ActivityRef ActivityId="Gutachten_erstellen" Id="Gutachten_erstellen"
                State="INACTIVE" Editable="ALL" ParticipantId="Gutachter"/>
   <ActivityRef ActivityId="Fahrzeug_reparieren" Id="Fahrzeug_reparieren"
                State="INACTIVE" Editable="ALL"
                StrategyId="Werkstattanforderungen"/>
   <ActivityRef ActivityId="Versicherungsnehmer_informieren"
                Id="Versicherungsnehmer_informieren" State="INACTIVE"
                Editable="NO" ParticipantId="Kunde"/>
   <ActivityRef ActivityId="Gesamtkosten_berechnen" Id="Gesamtkosten_berechnen"
                State="INACTIVE" Editable="NO"/>
   <ActivityRef ActivityId="Entscheidung" Id="Entscheidung" State="INACTIVE"
                Editable="NO"/>
</ActivityRefs>

<Transitions>
<Transition Id="1" From="Schadensfall_melden" To="GPSDaten_ermitteln"/>
<Transition Id="2" From="Schadensfall_melden" To="Versicherung_informieren"/>
<Transition Id="3" From="GPSDaten_ermitteln" To="Polizei_rufen"/>
<Transition Id="4" From="GPSDaten_ermitteln" To="Fahrzeug_abschleppen"/>
<Transition Id="5" From="GPSDaten_ermitteln" To="Leihwagen_mieten"/>
<Transition Id="6" From="Polizei_rufen" To="Entscheidung"/>
<Transition Id="7" From="Fahrzeug_abschleppen" To="Gutachten_erstellen"/>
<Transition Id="8" From="Leihwagen_mieten" To="Gesamtkosten_berechnen"/>
<Transition Id="9" From="Gutachten_erstellen" To="Fahrzeug_reparieren"/>
<Transition Id="10" From="Fahrzeug_reparieren"
                    To="Versicherungsnehmer_informieren"/>
<Transition Id="11" From="Versicherungsnehmer_informieren"
                    To="Gesamtkosten_berechnen"/>
<Transition Id="12" From="Gesamtkosten_berechnen" To="Entscheidung"/>
</Transitions>
```

Codebeispiel 59: Anwendungsbeispiel: Kontrollfluss

6 Schlussbetrachtung

Die Untersuchung von *DPDL* anhand des Anforderungskataloges hat gezeigt, dass der Einsatz von Prozessen auf mobilen Systemen realisierbar ist und dass viele der gestellten Anforderungen durch den erläuterten Ansatz erfüllt werden können. Das Anwendungsbeipiel hat präsentiert, wie *DPDL* in der Praxis und für die Formulierung konkreter Prozesse angewendet werden kann. Eine komplette Realisierung des Systems ist jedoch aufgrund seines Umfangs in dieser Diplomarbeit nicht möglich. In diesem Kapitel werden daher weitere Entwicklungsmöglichkeiten aufgezeigt, mögliche Einsatzgebiete vorgestellt und das Ergebnis der Arbeit zusammengefasst.

6.1 Ausblick

Die Prozessintegration in Middleware für mobile Systeme steht erst an ihrem Anfang und muss noch in vielfältiger Weise ergänzt und weiterentwickelt werden. Insbesondere an vielen konkreten Details, wie zum Beispiel der Entwicklung von Sicherheits-, Transaktions- und Benutzerkomponenten muss noch gearbeitet werden (vgl. 5.2.4).

Zudem kann auch die Ausführungsumgebung selbst durch ein umfassendes *Refactoring* an Rechenaufwand weiter reduziert und an zusätzliche energiesparende Maßnahmen angepasst werden. Um den verfügbaren Speicherplatz noch flexibler einzusetzen, können zum Beispiel Ergänzungskomponenten je nach Bedarf zur Laufzeit nachgeladen oder ausgetauscht werden. Mobile Systeme könnten somit nicht nur Prozessbeschreibungen miteinander austauschen, sondern auch Teile ihrer Ausführungsumgebung, die zur Bearbeitung einer speziellen Aufgabe benötigt werden.

Um die Flexibilität der Ausführungsumgebung und damit die Ausführbarkeit von Prozessen weiter zu steigern, können zusätzlich zu den bestehenden Komponenten weitere unabhängige Programme und Dienste in die Verarbeitung einbezogen werden, deren Funktionen ursprünglich nicht für die Be-

arbeitung von Prozessen bestimmt wurden. Zum Beispiel könnte ein lokal verfügbares Programm zur Datenkompression von der Ausführungsumgebung aufgerufen werden, um Prozessbeschreibungen zu komprimieren und damit den Übertragungsaufwand zu minimieren. Abbildung 56 zeigt eine Erweiterung der Architektur um austauschbare Komponenten und den Einbezug externer Anwendungen.

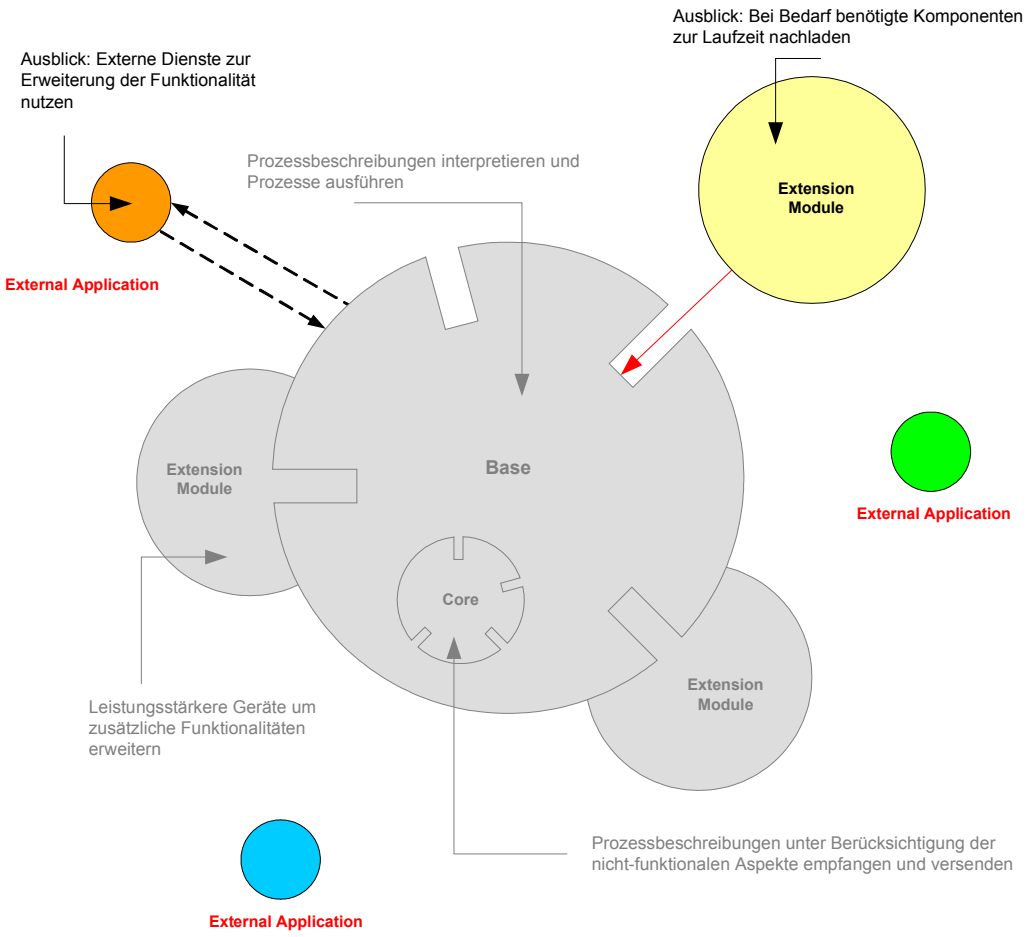


Abb. 56: Erweiterte Architektur

Die weitere Flexibilisierung vom Komponenten und Diensten kommt dabei einer Optimierung der Ausführungsumgebung entgegen, die speziell auf die zur Laufzeit auftretenden Bedürfnisse mobiler Systeme zugeschnitten ist. Diese Art von Selbstorganisation entlastet den Benutzer von häufigen Anpassungen seiner mobilen Geräte und erhöht die Zusammenarbeit und den Vernetzungsgrad der kontextsensitiven mobilen Systeme untereinander.

6.2 Einsatzmöglichkeiten

Im Gegensatz zu vielen der untersuchten Prozessbeschreibungsprachen ist *DPDL* nicht auf ein bestimmtes Einsatzgebiet, wie zum Beispiel die Formulierung von Geschäftstransaktionen beschränkt. Durch das zugrunde liegende abstrakte Meta-Modell von *XPDL* und die generellen Möglichkeiten zur Integration verschiedener Aktivitäten können Anwendungen beliebiger Art komponiert und so auch im Kontext heterogener Komponenten ausgeführt werden. So können zum Beispiel *Web Services*, *Corba*-Objekte und *RPC*-Aufrufe zu übergeordneten Anwendungen komponiert und aggregiert werden. Eine Ausführung ist nicht nur auf mobilen, sondern auch auf herkömmlichen stationären Systemen sowie als Integration mobiler und stationärer Systeme möglich.

Ein Schwerpunkt möglicher Einsatzzwecke liegt sicherlich im betrieblichen Bereich, wo räumlich voneinander getrennte Personen kooperieren und ein gemeinsames Ziel verfolgen. Hier können die mobilen Geräte vor allem Außendienstmitarbeiter bei ihrer Tätigkeit unterstützen, die Prozesse von ihrem jeweiligen Aufenthaltsort initieren oder bearbeiten können. Vor allem bei manuell zu erledigenden Tätigkeiten kann das Prozesssystem auch als Informations- und Verwaltungsmedium für anstehende Aufgaben dienen.

Neben dem Einsatz im betrieblichen Bereich sind auch private Einsatzzwecke denkbar, in denen Aufgaben des täglichen Lebens koordiniert und ausgeführt werden müssen. Der beschriebene Anwendungsfall der Versicherungssoftware für den Kunden zur Erleichterung einer Schadensabwicklung zeigt zudem, dass auch Verknüpfungen von betrieblichen und privaten Interessen möglich sind und durch ein mobiles Prozesssystem unterstützt werden können.

6.3 Ergebnis

Der Fortschritt im Bereich mobiler Technologien schreitet rasch voran und wird in naher Zukunft auch Anwendungen möglich machen, die zur Zeit zum Großteil noch stationären Systemen vorbehalten sind. Dennoch ist es eine inhärente Eigenschaft mobiler Systeme, in Bezug auf ihre Leistungsfähigkeit begrenzt und den stationären Systemen unterlegen zu sein. Besondere Schwachstellen der mobilen Geräte sind ihre Rechenleistung, die zur Verfügung stehende Speicherkapazität und ihre endlichen Energieressourcen. Die von ihnen verwendeten drahtlosen Kommunikationsmedien sind vor allem durch die geringe Bandbreite, ihre Unsicherheit und ihre Fehleranfälligkeit nicht mit denen stationärer Infrastrukuren zu vergleichen.

Um die Defizite einzelner mobiler Geräte ausgleichen zu können, wird ein System benötigt, welches die Kooperation unterschiedlicher und im Voraus unbekannter Komponenten unterstützt. Die Heterogenität bezieht sich dabei sowohl auf verschiedene Plattformen als auch auf eine große Divergenz von Leistungsmerkmalen und Fähigkeiten. Zu erledigende Aufgaben und Kontrollflussstrukturen zur Realisierung einer Zusammenarbeit dieser Einheiten müssen daher in einer möglichst abstrakten Form von Prozessen modelliert und durch eine plattformunabhängige Beschreibung kommuniziert werden.

Weitere wichtige Anforderungen an eine Beschreibungssprache für mobile Prozesse sind Konstrukte zur verteilten Administration des Prozesses und zur Fehlerbehandlung. Zudem müssen sich Benutzerinteressen durch nicht-funktionale Anforderungen formulieren lassen und während der Ausführung des Prozesses eingehalten werden. Alle Konzepte unterliegen dabei der besonderen Berücksichtigung der verminderten Leistungsfähigkeit mobiler Systeme in Bezug auf die oben genannten Kriterien.

Als Vertreter bereits bestehender Ansätze wurden die Beschreibungssprachen *XPDL*, *BPEL4WS*, *ebBPSS*, *jPDL* und *WSCI* auf ihre Eignung zur Integration auf mobilen Systemen untersucht. Dabei wurde festgestellt, dass keine der analysierten Sprachen die Anforderungen an eine Beschreibungssprache für mobilen Prozesse ausreichend erfüllen kann. Zum einen sind die betrachteten Ansätze zu komplex und zu beschreibungsaufwändig, zum anderen kann die Heterogenität im beschriebenen Kontext durch die zugrunde gelegten Konzepte nicht bewältigt werden.

Aufgrund des hohen Abstraktionsniveaus ist *XPDL* von den untersuchten Sprachen am besten geeignet, um als Grundlage für eine Prozessbeschreibungssprache zum Einsatz auf mobilen Systemen zu dienen. Zur Anpassung ist jedoch eine Ergänzung um wichtige Konzepte zur verteilten Administration, zur Fehlerbehandlung und zur Formulierung nicht-funktionaler Aspekte erforderlich. Die Vereinigung der geeigneten Teilkonzepte auf Basis von *XPDL* resultiert mit der Prozessbeschreibungssprache *DPDL* in einem eigenen Ansatz, der die Anforderungen der mobilen Systeme in den Vordergrund stellt und es so ermöglicht, Prozesse auf mobilen Systemen zu integrieren.

Die wichtigste Eigenschaft von *DPDL* stellt die Einführung eines Zustandskonzept dar, durch welches der Bearbeitungszustand eines Prozesses eindeutig gekennzeichnet und eine Weitergabe des unbearbeiteten oder teilbearbeiteten Prozesses an andere Ausführungseinheiten erlaubt wird. Die eigentliche Ausführung von Aktivitäten ist über die Beschreibung einer generischen Anwendung realisiert, deren Semantik durch die Angabe eines eindeutigen Bezeichners aus der Prozessbeschreibung extrahiert werden kann. Konkrete Prozessteilnehmer beliebiger Art können dadurch zur Laufzeit ausgewählt werden. Das Auswahlverhalten wird zusätzlich durch die Formulierung nicht-funktionaler Anforderungen an potentielle Partner unterstützt.

Es wurden außerdem einige neue Elemente eingefügt, um komplexe und lange Beschreibungen zu vermeiden und stattdessen durch semantisch eindeutige kurze Konstrukte zu ersetzen. Das Verhalten beim Auftreten von Fehlern und speziell von Verbindungsabbrüchen kann durch explizite Behandlungsmaßnahmen deklariert werden. Ebenso können Transaktionen und Schleifen als solche gekennzeichnet werden.

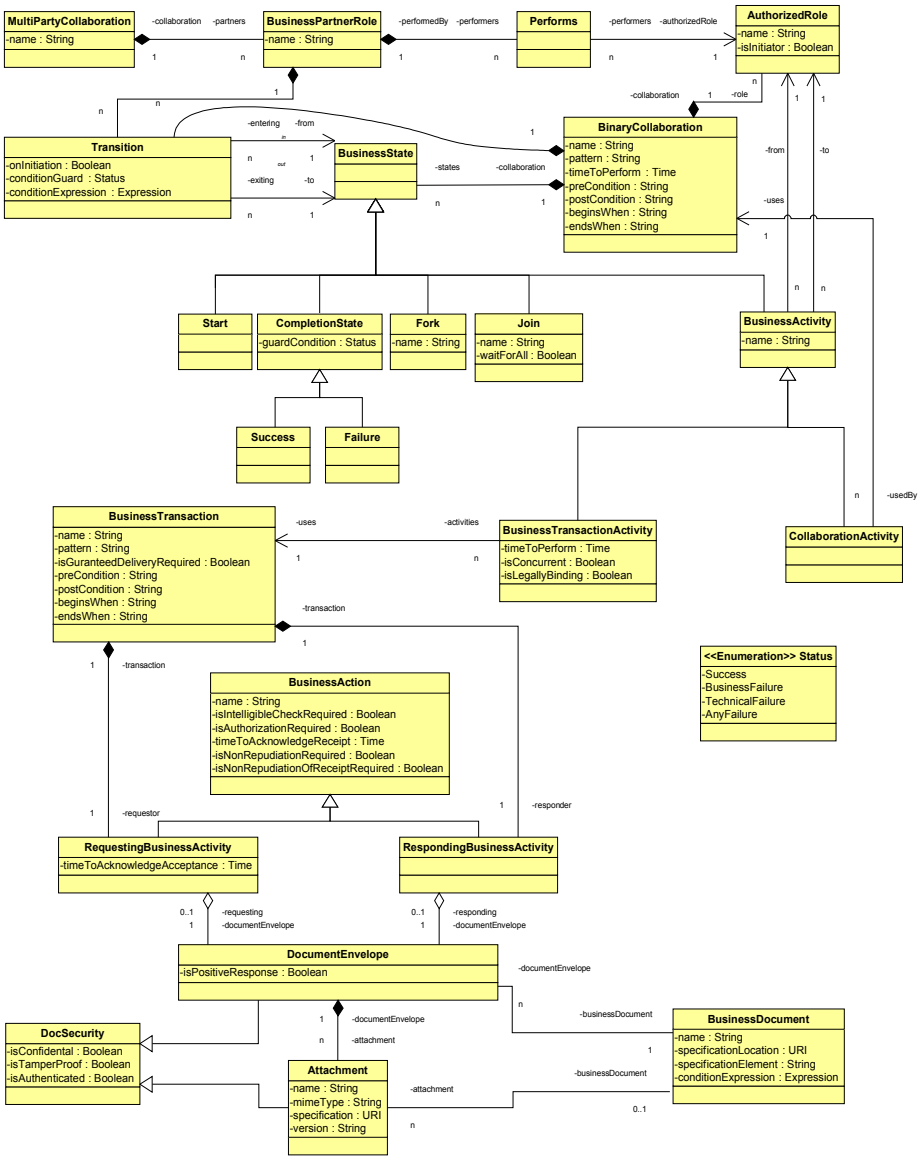
Der Prototyp einer Ausführungsumgebung für *DPDL*-Prozesse unterstützt die unterschiedlichen Leistungsmerkmale mobiler Systeme durch einen modularen Aufbau. So wird auch leistungsarmen Geräten eine Teilnahme am Prozess gestattet, indem sie als Initiatoren und Träger von Prozessbeschreibungen auftreten und diese an geeignete andere Teilnehmer weitergeben können. Leistungsstärkere Systeme können durch die Implementierung einer ergänzenden Basiskomponente Aktivitäten ausführen und so

die Bearbeitung des Prozesses voranbringen. Weitere optionale Zusatzmodule kommen speziellen Leistungsmerkmalen und Fähigkeiten von mobilen Geräten entgegen, zum Beispiel der Möglichkeit zur Benutzerinteraktion.

Durch den Einsatz von *DPDL* und einer entsprechenden Ausführungsumgebung können viele der gestellten Anforderungen an eine Prozessbeschreibungssprache erfüllt werden. Zwar handelt es sich bei der Minimierung von Speicherbedarf und der Minimierung des Kommunikationsaufwands um gegenläufige Ziele, es kann jedoch in den meisten Fällen anwendungsabhängig entschieden werden, welches Verhalten im jeweiligen Kontext erfolgversprechender ist. Zusammenfassend lässt sich festhalten, dass eine Prozessintegration in eine Middleware für mobile Systeme realisierbar ist und zur geplanten Kooperation von einzelnen Systemen sinnvoll eingesetzt werden kann.

Anhang

A1. ebBPSS Business Process Specification Schema als UML-Klassendiagramm



Anhang 1: ebBPSS Business Process Specification Schema als UML-Klassendiagramm (aus [Rie01])

A2. DPDL-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:dpdl="http://vsis-www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://vsis-www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd"
elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xsd:element name="Activities">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element ref="dpdl:Activity" minOccurs="0"
			maxOccurs="unbounded"/>
			</xsd:sequence>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="ActivitySet">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
				<xsd:element ref="dpdl:Transitions" minOccurs="0"/>
			</xsd:sequence>
			<xsd:attribute name="Id" type="xsd:string" use="required"/>
			<xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
			<xsd:attribute name="InitActivity" type="xsd:string" use="required"/>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="Activity">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element ref="dpdl:Description" minOccurs="0"/>
				<xsd:choice>
					<xsd:element ref="dpdl:Route"/>
					<xsd:element ref="dpdl:Implementation"/>
					<xsd:element ref="dpdl:BlockActivity"/>
					<xsd:element ref="dpdl:TransactionActivity"/>
					<xsd:element ref="dpdl:LoopActivity"/>
				</xsd:choice>
				<xsd:element ref="dpdl:StartMode" minOccurs="0"/>
				<xsd:element ref="dpdl:FinishMode" minOccurs="0"/>
				<xsd:element ref="dpdl:TransitionRestriction" minOccurs="0"/>
				<xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
			</xsd:sequence>
			<xsd:attribute name="Id" type="xsd:string" use="required"/>
			<xsd:attribute name="Name" type="xsd:string"/>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="ActivityRef">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element ref="dpdl:Deadline" minOccurs="0"/>
			</xsd:sequence>
			<xsd:attribute name="Id" type="xsd:string" use="required"/>
			<xsd:attribute name="ActivityId" type="xsd:string" use="required"/>
			<xsd:attribute name="ExceptionId" type="xsd:string"/>
			<xsd:attribute name="StrategyId" type="xsd:string"/>
			<xsd:attribute name="ConnectionResetHandlerId" type="xsd:string"/>
```

```
                    <xsd:attribute name="ParticipantId" type="xsd:string"/>
                    <xsd:attribute name="Priority" type="xsd:int"/>
                    <xsd:attribute name="State" use="required">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:NMTOKEN">
                                <xsd:enumeration value="INACTIVE"/>
                                <xsd:enumeration value="READY"/>
                                <xsd:enumeration value="EXECUTED"/>
                                <xsd:enumeration value="EXECUTING"/>
                                <xsd:enumeration value="FINISHED"/>
                                <xsd:enumeration value="EXPIRED"/>
                                <xsd:enumeration value="ERROR"/>
                                <xsd:enumeration value="SKIPPED"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                    <xsd:attribute name="Editable" default="NO">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:NMTOKEN">
                                <xsd:enumeration value="NO"/>
                                <xsd:enumeration value="ALL"/>
                                <xsd:enumeration value="REMOVE"/>
                                <xsd:enumeration value="EDIT"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="ActivityRefs">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:ActivityRef" minOccurs="0"
                    maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="ActivitySets">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:ActivitySet" maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="ActualParameter" type="xsd:string"/>
            <xsd:element name="ActualParameters">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:ActualParameter"
                    maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="Application">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:UUID"/>
                        <xsd:choice>
                            <xsd:element ref="dpdl:FormalParameters"
                        minOccurs="0"/>
                            <xsd:element ref="dpdl:ExternalReference"
                        minOccurs="0"/>
                            <xsd:element ref="dpdl:ExtendedAttributes"
```

```
                                minOccurs="0"/>
                                </xsd:choice>
                        </xsd:sequence>
                        <xsd:attribute name="Id" type="xsd:string" use="required"/>
                        <xsd:attribute name="Name" type="xsd:string"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Applications">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:Application" maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="ASYNCHR"/>
        <xsd:element name="BasicType">
                <xsd:complexType>
                        <xsd:attribute name="Type" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="BlockActivity">
                <xsd:complexType>
                        <xsd:attribute name="BlockId" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Compensation">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:Activities" minOccurs="0"/>
                                <xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
                                <xsd:element ref="dpdl:Transitions" minOccurs="0"/>
                        </xsd:sequence>
                        <xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
                        <xsd:attribute name="InitActivity" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Condition">
                <xsd:complexType mixed="true">
                        <xsd:sequence minOccurs="0">
                                <xsd:element ref="dpdl:Xpression" minOccurs="0"/>
                        </xsd:sequence>
                        <xsd:attribute name="Type" use="required">
                                <xsd:simpleType>
                                        <xsd:restriction base="xsd:NMTOKEN">
                                                <xsd:enumeration value="CONDITION"/>
                                                <xsd:enumeration value="OTHERWISE"/>
                                                <xsd:enumeration value="EXCEPTION"/>
                                        </xsd:restriction>
                                </xsd:simpleType>
                        </xsd:attribute>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="ConnectionResetHandler">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:ExceptionId" minOccurs="0"/>
                                <xsd:element ref="dpdl:Notification" minOccurs="0"/>
                                <xsd:element ref="dpdl:NewSearch" minOccurs="0"/>
                                <xsd:element ref="dpdl:Retries" minOccurs="0"/>
                                <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                        </xsd:sequence>
                        <xsd:attribute name="Id" type="xsd:string" use="required"/>
                </xsd:complexType>
```

```
</xsd:element>
<xsd:element name="ConnectionResetHandlers">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:ConnectionResetHandler"
                maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="DataField">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:DataType"/>
                         <xsd:choice>
                                <xsd:element ref="dpdl:InitialValue"  minOccurs="0"/>
                                <xsd:element ref="dpdl:ExternalReference"
                        minOccurs="0"/>
                        </xsd:choice>
                        <xsd:element ref="dpdl:Length" minOccurs="0"/>
                        <xsd:element ref="dpdl:TimeStamp" minOccurs="0"/>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="IsArray" default="false">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="true"/>
                                        <xsd:enumeration value="false"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                </xsd:attribute>
        </xsd:complexType>
</xsd:element>
<xsd:element name="DataFields">
        <xsd:complexType>
                <xsd:sequence minOccurs="0">
                        <xsd:element ref="dpdl:DataField" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="DataType">
        <xsd:complexType>
                <xsd:group ref="dpdl:DataTypes"/>
        </xsd:complexType>
</xsd:element>
<xsd:group name="DataTypes">
        <xsd:choice>
                <xsd:element ref="dpdl:BasicType"/>
                <xsd:element ref="dpdl:DeclaredType"/>
                <xsd:element ref="dpdl:SchemaType"/>
                <xsd:element ref="dpdl:ExternalReference"/>
        </xsd:choice>
</xsd:group>
<xsd:element name="Deadline">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:DeadlineCondition"/>
                        <xsd:element ref="dpdl:ExceptionId" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Execution" default="SYNCHR">
                        <xsd:simpleType>
```

```
                                        <xsd:restriction base="xsd:NMTOKEN">
                                                <xsd:enumeration value="ASYNCHR"/>
                                                <xsd:enumeration value="SYNCHR"/>
                                        </xsd:restriction>
                                </xsd:simpleType>
                        </xsd:attribute>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="DeadlineCondition" type="xsd:dateTime"/>
        <xsd:element name="DeclaredType">
                <xsd:complexType>
                        <xsd:attribute name="Id" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Description" type="xsd:string"/>
        <xsd:element name="Device">
                <xsd:complexType>
                        <xsd:choice>
                                <xsd:element ref="dpdl:UUID" minOccurs="0"/>
                                <xsd:element ref="dpdl:Devicetype" minOccurs="0"/>
                        </xsd:choice>
                        <xsd:attribute name="Id" type="xsd:string" use="required"/>
                        <xsd:attribute name="Name" type="xsd:string"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Devices">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:Device" maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Devicetype">
                <xsd:complexType>
                        <xsd:attribute name="Type" type="xsd:string"/>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Exception">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
                                <xsd:element ref="dpdl:Transitions" minOccurs="0"/>
                        </xsd:sequence>
                        <xsd:attribute name="Id" type="xsd:string" use="required"/>
                        <xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
                        <xsd:attribute name="InitActivity" type="xsd:string" use="required"/>
                        <xsd:attribute name="Name" type="xsd:string"/>
                        <xsd:attribute name="Execution" use="required">
                                <xsd:simpleType>
                                        <xsd:restriction base="xsd:NMTOKEN">
                                                <xsd:enumeration value="ASYNCHR"/>
                                                <xsd:enumeration value="SYNCHR"/>
                                        </xsd:restriction>
                                </xsd:simpleType>
                        </xsd:attribute>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="ExceptionId" type="xsd:string"/>
        <xsd:element name="Exceptions">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="dpdl:Exception" maxOccurs="unbounded"/>
                        </xsd:sequence>
```

```
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="Execution">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:element ref="dpdl:ASYNCHR"/>
                    <xsd:element ref="dpdl:SYNCHR"/>
                </xsd:choice>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="ExtendedAttribute">
            <xsd:complexType mixed="true">
                <xsd:choice minOccurs="0" maxOccurs="unbounded">
                    <xsd:any processContents="lax" minOccurs="0"
                maxOccurs="unbounded"/>
                </xsd:choice>
                <xsd:attribute name="Name" type="xsd:string" use="required"/>
                <xsd:attribute name="Value" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="ExtendedAttributes">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:ExtendedAttribute" minOccurs="0"
                maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="ExternalReference">
            <xsd:complexType>
                <xsd:attribute name="Xref" type="xsd:string" use="optional"/>
                <xsd:attribute name="Location" type="xsd:anyURI" use="required"/>
                <xsd:attribute name="Namespace" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="FinishMode">
            <xsd:complexType>
                <xsd:attribute name="Type" default="AUTOMATIC">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:NMTOKEN">
                            <xsd:enumeration value="AUTOMATIC"/>
                            <xsd:enumeration value="MANUAL"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="FormalParameter">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:DataType"/>
                    <xsd:element ref="dpdl:Description" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Index" type="xsd:positiveInteger"/>
                <xsd:attribute name="Mode" default="IN">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:NMTOKEN">
                            <xsd:enumeration value="IN"/>
                            <xsd:enumeration value="OUT"/>
                            <xsd:enumeration value="INOUT"/>
                        </xsd:restriction>
                    </xsd:simpleType>
```

```
                </xsd:attribute>
        </xsd:complexType>
</xsd:element>
<xsd:element name="FormalParameters">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:FormalParameter"
                maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Human" type="xsd:string"/>
<xsd:element name="Implementation">
        <xsd:complexType>
                <xsd:choice>
                        <xsd:element ref="dpdl:No"/>
                        <xsd:element ref="dpdl:Tool"/>
                        <xsd:element ref="dpdl:SubFlow"/>
                </xsd:choice>
        </xsd:complexType>
</xsd:element>
<xsd:element name="InitialValue" type="xsd:string"/>
<xsd:element name="Join">
        <xsd:complexType>
                <xsd:attribute name="URI" type="xsd:anyURI"/>
                <xsd:attribute name="Type" use="required">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="AND"/>
                                        <xsd:enumeration value="XOR"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                </xsd:attribute>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Length" type="xsd:int"/>
<xsd:element name="Loop">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
                        <xsd:element ref="dpdl:Transitions" minOccurs="0"/>
                        <xsd:element ref="dpdl:Condition" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
                <xsd:attribute name="InitActivity" type="xsd:string" use="required"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="LoopActivity">
        <xsd:complexType>
                <xsd:attribute name="LoopId" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Loops">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Loop" minOccurs="0"
                maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="NewSearch">
        <xsd:simpleType>
                <xsd:restriction base="xsd:NMTOKEN">
                        <xsd:enumeration value="false"/>
                        <xsd:enumeration value="true"/>
                </xsd:restriction>
        </xsd:simpleType>
</xsd:element>
<xsd:element name="No">
        <xsd:complexType/>
</xsd:element>
<xsd:element name="Participant">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExternalReference" minOccurs="0"/>
                        <xsd:element ref="dpdl:Devices" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                        <xsd:choice minOccurs="0">
                                <xsd:element ref="dpdl:Role" minOccurs="0"/>
                                <xsd:element ref="dpdl:Human" minOccurs="0"/>
                        </xsd:choice>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Package">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Strategies" minOccurs="0"/>
                        <xsd:element ref="dpdl:TypeDeclarations" minOccurs="0"/>
                        <xsd:element ref="dpdl:Participants" minOccurs="0"/>
                        <xsd:element ref="dpdl:Applications" minOccurs="0"/>
                        <xsd:element ref="dpdl:DataFields" minOccurs="0"/>
                        <xsd:element ref="dpdl:WorkflowProcess"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="DPDLVersion" type="xsd:string"/>
                <xsd:attribute name="Owner" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Notification" type="xsd:string"/>
<xsd:element name="Participants">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Participant" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Priority" type="xsd:int"/>
<xsd:element name="ProcessHeader">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:Priority" minOccurs="0"/>
                        <xsd:element ref="dpdl:ValidFrom" minOccurs="0"/>
                        <xsd:element ref="dpdl:ValidTo" minOccurs="0"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="Requirement">
        <xsd:complexType>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="Value" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Requirements">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Requirement"
                maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Retries">
        <xsd:simpleType>
                <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="0"/>
                </xsd:restriction>
        </xsd:simpleType>
</xsd:element>
<xsd:element name="Role">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Route">
        <xsd:complexType/>
</xsd:element>
<xsd:element name="SchemaType">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:any namespace="##other" processContents="lax"
                minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Split">
        <xsd:complexType>
                <xsd:attribute name="Type" use="required">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="AND"/>
                                        <xsd:enumeration value="XOR"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                </xsd:attribute>
        </xsd:complexType>
</xsd:element>
<xsd:element name="StartMode">
        <xsd:complexType>
                <xsd:attribute name="Type" default="AUTOMATIC">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="AUTOMATIC"/>
                                        <xsd:enumeration value="MANUAL"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                </xsd:attribute>
```

```
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="Strategies">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:Strategy" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="Strategy">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:StrategyProperty" minOccurs="0"
                maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="StrategyProperty">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:Requirements" minOccurs="0"
                maxOccurs="unbounded"/>
                    <xsd:element ref="dpdl:Description" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="SubFlow">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:ActualParameters" minOccurs="0"/>
                    <xsd:element ref="dpdl:Execution"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="SYNCHR">
            <xsd:complexType>
                <xsd:attribute name="URL" type="xsd:anyURI"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="TimeStamp" type="xsd:dateTime"/>
        <xsd:element name="Tool">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:ActualParameters" minOccurs="0"/>
                    <xsd:element ref="dpdl:Description" minOccurs="0"/>
                    <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="ApplicationId" type="xsd:string" use="required"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="Transaction">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="dpdl:Retries" minOccurs="0"/>
                    <xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
                    <xsd:element ref="dpdl:Transitions" minOccurs="0"/>
                    <xsd:element ref="dpdl:Compensation" minOccurs="0"/>
                    <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
```

```
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
                <xsd:attribute name="InitActivity" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="Type" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="TransactionActivity">
        <xsd:complexType>
                <xsd:attribute name="TransactionId" type="xsd:string" use="required"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Transactions">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Transaction" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Transition">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Condition" minOccurs="0"/>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
                <xsd:attribute name="From" type="xsd:string"/>
                <xsd:attribute name="To" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="TransitionRestriction">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Join" minOccurs="0"/>
                        <xsd:element ref="dpdl:Split" minOccurs="0"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Transitions">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:Transition" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="TypeDeclaration">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:group ref="dpdl:DataTypes"/>
                        <xsd:element ref="dpdl:Description" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="Id" type="xsd:string" use="required"/>
                <xsd:attribute name="Name" type="xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name="TypeDeclarations">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="dpdl:TypeDeclaration"
```

```
                    maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="UUID" type="xsd:string"/>
        <xsd:element name="ValidFrom" type="xsd:dateTime"/>
        <xsd:element name="ValidTo" type="xsd:dateTime"/>
        <xsd:element name="WorkflowProcess">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:ProcessHeader" minOccurs="0"/>
                        <xsd:element ref="dpdl:FormalParameters" minOccurs="0"/>
                        <xsd:element ref="dpdl:Exceptions" minOccurs="0"/>
                        <xsd:element ref="dpdl:ConnectionResetHandlers"
                    minOccurs="0"/>
                        <xsd:element ref="dpdl:Transactions" minOccurs="0"/>
                        <xsd:element ref="dpdl:ActivitySets" minOccurs="0"/>
                        <xsd:element ref="dpdl:Loops" minOccurs="0"/>
                        <xsd:element ref="dpdl:Activities" minOccurs="0"/>
                        <xsd:element ref="dpdl:ActivityRefs" minOccurs="0"/>
                        <xsd:element ref="dpdl:Transitions" minOccurs="0"/>
                        <xsd:element ref="dpdl:ExtendedAttributes" minOccurs="0"/>
                    </xsd:sequence>
                    <xsd:attribute name="Id" type="xsd:string" use="required"/>
                    <xsd:attribute name="StartActivity" type="xsd:string" use="required"/>
                    <xsd:attribute name="InitActivity" type="xsd:string"/>
                    <xsd:attribute name="Name" type="xsd:string"/>
                    <xsd:attribute name="StrategyId" type="xsd:string"/>
                    <xsd:attribute name="AccessLevel" default="PUBLIC">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:NMTOKEN">
                                <xsd:enumeration value="PUBLIC"/>
                                <xsd:enumeration value="PRIVATE"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                    <xsd:attribute name="LoggingRequired" default="false">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:NMTOKEN">
                                <xsd:enumeration value="true"/>
                                <xsd:enumeration value="false"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Xpression">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="dpdl:Tool"/>
                    </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

A3. DPDL-Anwendungsbeispiel

```
<Package xmlns="http://vsis-www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://vsis-www.informatik.uni-hamburg.de/projects/demac/dpdl1.0.xsd
Id="ALLIANCE,Versicherten-Nr.1234-56789,1" Name="Schadenschnellhilfe" DPDLVersion="1.0"
Owner="ALLIANCE,Versicherten-Nr.1234-56789">
        <Strategies>
                <Strategy Id="Werkstattanforderungen">
                        <StrategyProperty Id="1">
                                <Requirements>
                                        <Requirement Name="MaxReparaturzeit" Value="30"/>
                                </Requirements>
                        </StrategyProperty>
                </Strategy>
        </Strategies>
        <TypeDeclarations>
                <TypeDeclaration Id="GPSData">
                        <SchemaType>
                                <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                elementFormDefault="qualified" attributeFormDefault="unqualified">
                                <xsd:element name="GPSData">
                                <xsd:complexType>
                                <xsd:sequence>
                                <xsd:element name="Latitude" type="xsd:string"/>
                                <xsd:element name="Longitude" type="xsd:string"/>
                                </xsd:sequence>
                                </xsd:complexType>
                                </xsd:element>
                                </xsd:schema>
                        </SchemaType>
                </TypeDeclaration>
        </TypeDeclarations>
        <Participants>
                <Participant Id="Gutachter">
                        <Role Id="Gutachter/KFZ" Name="Gutachter">
                        </Role>
                </Participant>
                <Participant Id="Kunde">
                        <Devices>
                                <Device Id="Mobiltelefon:1234-56789">
                              <UUID>12311111901234567890123456789012</UUID>
                                </Device>
                        </Devices>
                        <Human>Versicherter 1234-56789</Human>
                </Participant>
        </Participants>
        <Applications>
                <Application Id="GPS-Service">
                        <UUID>12345678901234567890123456789012</UUID>
                        <FormalParameters>
                                <FormalParameter Id="Data" Mode="OUT">
                                        <DataType>
                                                <DeclaredType Id="GPSData"/>
                                        </DataType>
                                </FormalParameter>
```

```
            </FormalParameters>
        </Application>
        <Application Id="Informations-Service">
            <UUID>12345678901234567890123456789013</UUID>
            <FormalParameters>
                <FormalParameter Id="Versicherten_Identifikation" Mode="IN">
                    <DataType>
                        <BasicType Type="String"/>
                    </DataType>
                </FormalParameter>
            </FormalParameters>
        </Application>
        <Application Id="Notruf-Service">
            <UUID>12345678901234567890123456789015</UUID>
            <FormalParameters>
                <FormalParameter Id="Ort" Mode="IN">
                    <DataType>
                        <DeclaredType Id="GPSData"/>
                    </DataType>
                </FormalParameter>
                <FormalParameter Id="Bericht" Mode="OUT">
                    <DataType>
                        <BasicType Type="String"/>
                    </DataType>
                </FormalParameter>
            </FormalParameters>
        </Application>
        <Application Id="Abschlepp-Service">
            <UUID>12345678901234567890123456789017</UUID>
            <FormalParameters>
                <FormalParameter Id="Ort" Mode="IN">
                    <DataType>
                        <DeclaredType Id="GPSData"/>
                    </DataType>
                </FormalParameter>
                <FormalParameter Id="Kennzeichen" Mode="IN">
                    <DataType>
                        <BasicType Type="String"/>
                    </DataType>
                </FormalParameter>
            </FormalParameters>
        </Application>
        <Application Id="Leihwagen-Service">
            <UUID>12345678901234567890123456789019</UUID>
            <FormalParameters>
                <FormalParameter Id="Klasse" Mode="OUT">
                    <DataType>
                        <BasicType Type="String"/>
                    </DataType>
                </FormalParameter>
                <FormalParameter Id="Kosten" Mode="OUT">
                    <DataType>
                        <BasicType Type="Double"/>
                    </DataType>
                </FormalParameter>
            </FormalParameters>
        </Application>
        <Application Id="Eingabe-Anwendung">
            <UUID>12345678901234567890123999789019</UUID>
            <FormalParameters>
                <FormalParameter Id="Text" Mode="IN">
                    <DataType>
                        <BasicType Type="String"/>
```

```
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Kosten" Mode="IN">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                        </FormalParameters>
                </Application>
                <Application Id="Benachrichtigung">
                        <UUID>00045678901234567890123999789019</UUID>
                </Application>
                <Application Id="Rechner">
                        <UUID>12345678901200000890123456789019</UUID>
                        <FormalParameters>
                                <FormalParameter Id="Summand1" Mode="IN">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Summand2" Mode="IN">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Summand3" Mode="IN">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Ergebnis" Mode="OUT">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                        </FormalParameters>
                </Application>
                <Application Id="Alliance-Service">
                        <UUID>50000678901234567890123456789019</UUID>
                        <FormalParameters>
                                <FormalParameter Id="Polizeibericht" Mode="IN">
                                        <DataType>
                                                <BasicType Type="String"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Gutachten" Mode="IN">
                                        <DataType>
                                                <BasicType Type="String"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Kosten" Mode="IN">
                                        <DataType>
                                                <BasicType Type="Double"/>
                                        </DataType>
                                </FormalParameter>
                                <FormalParameter Id="Entscheidung" Mode="OUT">
                                        <DataType>
                                                <BasicType Type="Boolean"/>
                                        </DataType>
                                </FormalParameter>
                        </FormalParameters>
                </Application>
        </Applications>
```

```
<DataFields>
        <DataField Id="Versichertennummer" IsArray="false">
                <DataType>
                        <BasicType Type="String"/>
                </DataType>
                <InitialValue>1234-56789</InitialValue>
        </DataField>
        <DataField Id="Kennzeichen" IsArray="false">
                <DataType>
                        <BasicType Type="String"/>
                </DataType>
                <InitialValue>HH-XY-123</InitialValue>
        </DataField>
        <DataField Id="Fahrzeugklasse" IsArray="false">
                <DataType>
                        <BasicType Type="String"/>
                </DataType>
        </DataField>
        <DataField Id="GPSDaten" IsArray="false">
                <DataType>
                        <DeclaredType Id="GPSData"/>
                </DataType>
        </DataField>
        <DataField Id="Polizeibericht" IsArray="false">
                <DataType>
                        <BasicType Type="String"/>
                </DataType>
        </DataField>
        <DataField Id="Gutachten" IsArray="false">
                <DataType>
                        <BasicType Type="String"/>
                </DataType>
        </DataField>
        <DataField Id="KostenLeihwagen" IsArray="false">
                <DataType>
                        <BasicType Type="Double"/>
                </DataType>
        </DataField>
        <DataField Id="KostenWerkstatt" IsArray="false">
                <DataType>
                        <BasicType Type="Double"/>
                </DataType>
        </DataField>
        <DataField Id="KostenGutachter" IsArray="false">
                <DataType>
                        <BasicType Type="Double"/>
                </DataType>
        </DataField>
        <DataField Id="Gesamtkosten" IsArray="false">
                <DataType>
                        <BasicType Type="Double"/>
                </DataType>
        </DataField>
        <DataField Id="Entscheidung" IsArray="false">
                <DataType>
                        <BasicType Type="Boolean"/>
                </DataType>
        </DataField>
</DataFields>
<WorkflowProcess StartActivity="Schadensfall_melden" Id="1">
        <Activities>
                <Activity Id="Schadensfall_melden">
                        <Route/>
```

```
            <TransitionRestriction>
                <Split Type="AND"/>
            </TransitionRestriction>
        </Activity>
        <Activity Id="GPSDaten_ermitteln">
            <Implementation>
                <Tool ApplicationId="GPS-Service">
                    <ActualParameters>
                        <ActualParameter>
                            GPSDaten
                        </ActualParameter>
                    </ActualParameters>
                </Tool>
            </Implementation>
            <TransitionRestriction>
                <Split Type="AND"/>
            </TransitionRestriction>
        </Activity>
        <Activity Id="Versicherung_informieren">
            <Implementation>
                <Tool ApplicationId="Informations-Service">
                    <ActualParameters>
                        <ActualParameter>
                            Versichertennummer
                        </ActualParameter>
                    </ActualParameters>
                </Tool>
            </Implementation>
        </Activity>
        <Activity Id="Polizei_rufen">
            <Implementation>
                <Tool ApplicationId="Notruf-Service">
                    <ActualParameters>
                        <ActualParameter>
                            GPSDaten
                        </ActualParameter>
                        <ActualParameter>
                            Polizeibericht
                        </ActualParameter>
                    </ActualParameters>
                </Tool>
            </Implementation>
        </Activity>
        <Activity Id="Fahrzeug_abschleppen">
            <Implementation>
                <Tool ApplicationId="Abschlepp-Service">
                    <ActualParameters>
                        <ActualParameter>
                            GPSDaten
                        </ActualParameter>
                        <ActualParameter>
                            Kennzeichen
                        </ActualParameter>
                    </ActualParameters>
                </Tool>
            </Implementation>
        </Activity>
        <Activity Id="Leihwagen_mieten">
            <Implementation>
                <Tool ApplicationId="Leihwagen-Service">
                    <ActualParameters>
                        <ActualParameter>
                            Fahrzeugklasse
```

```
                                        </ActualParameter>
                                        <ActualParameter>
                                                KostenLeihwagen
                                        </ActualParameter>
                                </ActualParameters>
                        </Tool>
                </Implementation>
        </Activity>
        <Activity Id="Gutachten_erstellen">
                <Description>Gutachten für Versicherung erstellen</Description>
                <Implementation>
                        <Tool ApplicationId="Eingabe-Anwendung">
                                <ActualParameters>
                                        <ActualParameter>
                                                Gutachten
                                        </ActualParameter>
                                        <ActualParameter>
                                                KostenGutachter
                                        </ActualParameter>
                                </ActualParameters>
                        </Tool>
                </Implementation>
                <StartMode Type="MANUAL"/>
                <FinishMode Type="MANUAL"/>
        </Activity>
        <Activity Id="Fahrzeug_reparieren">
                <Description>
                        Unfallschäden am Fahrzeug reparieren
                </Description>
                <Implementation>
                        <Tool ApplicationId="Eingabe-Anwendung">
                                <ActualParameters>
                                        <ActualParameter>
                                                Kennzeichen
                                        </ActualParameter>
                                        <ActualParameter>
                                                KostenWerkstatt
                                        </ActualParameter>
                                </ActualParameters>
                        </Tool>
                </Implementation>
                <StartMode Type="MANUAL"/>
                <FinishMode Type="MANUAL"/>
        </Activity>
        <Activity Id="Versicherungsnehmer_informieren">
                <Description>
                        KFZ ist repariert und kann abgeholt werden
                </Description>
                <Implementation>
                        <Tool ApplicationId="Benachrichtigung"/>
                </Implementation>
        </Activity>
        <Activity Id="Gesamtkosten_berechnen">
                <Implementation>
                        <Tool ApplicationId="Rechner">
                                <ActualParameters>
                                        <ActualParameter>
                                                KostenLeihwagen
                                        </ActualParameter>
                                        <ActualParameter>
                                                KostenWerkstatt
                                        </ActualParameter>
                                        <ActualParameter>
```

```
                                                KostenGutachter
                                            </ActualParameter>
                                            <ActualParameter>
                                                Gesamtkosten
                                            </ActualParameter>
                                        </ActualParameters>
                                    </Tool>
                                </Implementation>
                                <TransitionRestriction>
                                <Join Type="AND" Id="Kostenberechnung"
                        URI="DeviceHandle:12345678901234567890123456789012"/>
                                </TransitionRestriction>
                        </Activity>
                        <Activity Id="Entscheidung">
                                <Implementation>
                                    <Tool ApplicationId="Alliance-Service">
                                        <ActualParameters>
                                            <ActualParameter>
                                                Polizeibericht
                                            </ActualParameter>
                                            <ActualParameter>
                                                Gutachten
                                            </ActualParameter>
                                            <ActualParameter>
                                                Gesamtkosten
                                            </ActualParameter>
                                            <ActualParameter>
                                                Entscheidung
                                            </ActualParameter>
                                        </ActualParameters>
                                    </Tool>
                                </Implementation>
                                <TransitionRestriction>
                                    <Join Type="AND" Id="Versicherung"
                                    URI="http://www.allianceag.com/join"/>
                                </TransitionRestriction>
                        </Activity>
                </Activities>
                <ActivityRefs>

                        <ActivityRef ActivityId="Schadensfall_melden" Id="Schadensfall_melden"
                        State="INACTIVE" Editable="NO"/>

                        <ActivityRef ActivityId="GPSDaten_ermitteln" Id="GPSDaten_ermitteln"
                        State="INACTIVE" Editable="EDIT"/>

                        <ActivityRef ActivityId="Versicherung_informieren"
                        Id="Versicherung_informieren" State="INACTIVE" Editable="NO"/>

                        <ActivityRef ActivityId="Polizei_rufen" Id="Polizei_rufen"
                        State="INACTIVE" Editable="ALL"/>

                        <ActivityRef ActivityId="Fahrzeug_abschleppen"
                        Id="Fahrzeug_abschleppen" State="INACTIVE" Editable="ALL"/>

                        <ActivityRef ActivityId="Leihwagen_mieten" Id="Leihwagen_mieten"
                        State="INACTIVE" Editable="ALL"/>

                        <ActivityRef ActivityId="Gutachten_erstellen" Id="Gutachten_erstellen"
                        State="INACTIVE" Editable="ALL" ParticipantId="Gutachter"/>

                        <ActivityRef ActivityId="Fahrzeug_reparieren" Id="Fahrzeug_reparieren"
                State="INACTIVE" Editable="ALL"
```

```
                    StrategyId="Werkstattanforderungen"/>

                    <ActivityRef ActivityId="Versicherungsnehmer_informieren"
                    Id="Versicherungsnehmer_informieren" State="INACTIVE"
            Editable="NO" ParticipantId="Kunde"/>

                    <ActivityRef ActivityId="Gesamtkosten_berechnen"
                    Id="Gesamtkosten_berechnen" State="INACTIVE" Editable="EDIT"/>

                    <ActivityRef ActivityId="Entscheidung" Id="Entscheidung"
                    State="INACTIVE" Editable="ALL"/>

            </ActivityRefs>
            <Transitions>
                    <Transition Id="1" From="Schadensfall_melden"
                                To="GPSDaten_ermitteln"/>
                    <Transition Id="2" From="Schadensfall_melden"
                                To="Versicherung_informieren"/>
                    <Transition Id="3" From="GPSDaten_ermitteln"
                                To="Polizei_rufen"/>
                    <Transition Id="4" From="GPSDaten_ermitteln"
                                To="Fahrzeug_abschleppen"/>
                    <Transition Id="5" From="GPSDaten_ermitteln"
                                To="Leihwagen_mieten"/>
                    <Transition Id="6" From="Polizei_rufen"
                                To="Entscheidung"/>
                    <Transition Id="7" From="Fahrzeug_abschleppen"
                                To="Gutachten_erstellen"/>
                    <Transition Id="8" From="Leihwagen_mieten"
                                To="Gesamtkosten_berechnen"/>
                    <Transition Id="9" From="Gutachten_erstellen"
                                To="Fahrzeug_reparieren"/>
                    <Transition Id="10" From="Fahrzeug_reparieren"
                                To="Versicherungsnehmer_informieren"/>
                    <Transition Id="11" From="Versicherungsnehmer_informieren"
                                To="Gesamtkosten_berechnen"/>
                    <Transition Id="12" From="Gesamtkosten_berechnen"
                                To="Entscheidung"/>
            </Transitions>
    </WorkflowProcess>
</Package>
```

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele

Literaturverzeichnis

[AAF+02] Arkin, A.; Askary, S.; Fordin, S.; Jekeli, W.; Kawaguchi, K.; Orchard, D.; Pogliani S.; Riemer, K.; Struble, S.; Takacsi-Nagy, P.; Trickovic, I.; Zimek, S.: *Web Service Choreography Interface (WSCI)*, Version 1.0. W3C, http://www.w3c.org/TR/wsci/, 2002.

[Aal03] Van der Aalst, W.M.P.: *Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language*. Technical Report, Eindhoven University of Technology, http://is.tm.tue.nl/research/patterns/download/ce-xpdl.pdf (Abruf: 22.05.2005), 2003.

[AaHe02] Van der Aalst, W.M.P.; van Hee, K.: *Workflow Management – Models, Methods and Systems*. MIT Press, Cambridge, ISBN: 0262011891, 2002

[ACD+03] Andrews, T.; Curbera, F.; Dholakia, H.; Goland, Y.; Klein, J.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; Weerawarana, S.: *Business Process Execution Language for Web Services Specification*, Version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf (Abruf: 01.06.2005), 2003.

[ADH03] Van der Aalst, W.M.P.; Dumas, M.; ter Hofstede, A.: *Web Service Composition Languages: Old Wine in New Bottles?* in *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, Seite 298-305. http://is.tm.tue.nl/research/patterns/download/wscl-euromicro.pdf (Abruf: 14.06.2005), IEEE Computer Society, Los Alamitos, 2003.

[AGK+95] Alonso, G.; Günthör, R.; Kamath, M.; Agrawal, D.; El Abbadi, A.; Mohan, C.: *Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System* in *Proc. 3rd International Conference on Cooperative Information Systems.* Wien, 1995.

[Bae04] Baeyens, T.: The State of Workflow. JBOSS Company, http://jbpm.org/state.of.workflow.html (Abruf: 17.06.2005), 2004.

[BeNe97] Bernstein, P.A.; Newcomer, E.: *Transaction Processing*. Morgan Kaufmann Publishers, San Francisco, ISBN: 1558604154, 1997.

[Ber96] Bernstein, P.A.: *Middleware – A Model for Distributed System Services* in *Communications of the ACM*, Vol.39, No.2, Seite 86 - 98. 1996.

[BHM+05] Booth,D.; Haas,H.; McCabe, F.; Newcomer, E.; Champion M.; Ferris, C.; Orchard, D.: *Web Service Architecture.* W3C Word Wide Web Consortium, Web Service Working Group, http://www.w3.org/TR/ws-arch/#service_oriented_architecture (Abruf: 09.06.2005), 2005.

[BrSz03] Brown, P.; Szefler, M.: *BPEL for Programmers and Architects.* FiveSight Technologies Inc., http://blog.fivesight.com/prb/space/BPEL/BPEL4ProgArchies.pdf (Abruf: 01.06.2005), 2003.

[BoSch98] Borghoff, U.M.; Schlichter, J.H.: *Rechnergestützte Gruppenarbeit*. Springer Verlag, Berlin, ISBN: 3540628738, 1998.

[Box04] Box, D.: A Guide to Developing and Running Connected Systems with Indigo. MSDN Magazin, http://msdn.microsoft.com/msdnmag/issues/04/01/Indigo/ default.aspx (Abruf: 17.06.2005), 2004

[CCMW01] Christensen, E.; Curbera, F.; Meredith, G.; Weereawarana, S.: *Web Services Description Language Specification,* Version 1.1, W3C, http://www.w3.org/TR/wsdl (Abruf: 01.06.2005), 2001.

[CDK02] Coulouris, G.; Dollimore, J.; Kindberg, T.: *Verteilte Systeme – Konzepte und Design*, 3. Auflage. Pearson Studium, München, ISBN: 3827370221, 2002.

[CEM02] Capra, L.; Emmerich, W.; Mascolo, C.: *Middleware for Mobile Computing* in Gregori, E.; Anastasi, G., Basagni, S.: *Networking 2002 Tutorial Papers*, Volume 2497, Seite 20 – 58. 2002.

[ClDe99] Clark, J.; DeRose, S.: *XML Path Language (XPath)*, Version 1.0, W3C, http://www.w3.org/TR/xpath (Abruf: 01.06.2005), 1999.

[DiHe95] Diehl, N.; Held, A.: *Mobile Computing: Systeme, Kommunikation, Anwendung*. Int. Thomson Publishing, Bonn, ISBN: 392982180X, 1995.

[Dey01] Dey, A. K.: *Understanding and Using Context* in *Personal and Ubiquitous Computing Journal Vol. 5,* Seite 4 – 7. Springer Verlag, London, 2001.

[Dje04] Djemili, R.: *BPEL4WS*, Seminararbeit. Freie Universität Berlin, http://www.riad.de/edu/BPEL4WS.pdf (Abruf: 09.06.2005), 2004.

[DoBe92] Dourish, P.; Belotti, V.: *Awareness and Coordination in Shared Workspaces,* in *Proceedings of CSCW `92,* Seite 107-114. ACM Press, Toronto, 1992.

[Dub02] Dubray, J.-J.: *Analysis on XPDL*, http://www.ebpml.org/xpdl.htm (Abruf: 26.05.2005), 2002

[Dub04] Dubray, J.-J.: *Analysis on BPEL4WS,* http://www.ebpml.org/ bpel4ws.htm (Abruf: 01.06.2005), 2004

[DuGa03] Dustdar, S.; Gall, H.: *Architectural Concerns in Distributed and Mobile Collaborative Systems* in *Journal of Systems Architecture: the EUROMICRO Journal,* Volume 49, Issue 10-11, Seite 457 – 473. Elsevier North-Holland Inc, New York, 2003.

[EbFi03] Eberhart, A.; Fischer, S.: *Web Services – Grundlagen und praktische Umsetzung mit J2EE und .NET*. Hanser Verlag, München, ISBN: 3446225307, 2003.

[Eck03] Eckert, C.: *Mobil, aber sicher!* in Mattern, F.: *Total vernetzt – Szenarien einer informatisierten Welt,* Seite 85– 121. Springer, Berlin, ISBN: 3540002138, 2003.

[Emm03] Emmerich, W.: *Konstruktion von verteilten Objekten*. Dpunkt Verlag, Heidelberg, ISBN: 3898641406, 2003.

[EOH02] Edmond, D.; O´Sullivan, J.; ter Hofstede, A.: *What's in a Service? Towards Accurate Description of Non-Functional Service Properties* in *Distributed and Parallel Databases Journal 12,* Seite 117–133. Kluwer Academic Publishers, Hingham, 2002.

[Fan05] Fancey, J.: *What is BPEL4WS? Build Better Business Processes with Web Services in BizTalk Server 2004.* MSDN Magazine 03/05, http://msdn.microsoft.com/ msdnmag/issues/05/03/BPEL4WS/default.aspx (Abruf: 03.06.2005), 2005.

[FoZa94] Forman, G.H.; Zahorjan, J.: *The Challenges of Mobile Computing,* S. 38-47. IEEE Computing 27(4), 1994.

[For98] Forman, G.H.: *Wanted: Programming Support for Ensuring Responsiveness Despite Resource Variability and Volatility.* Software Technology Laboratory Hewlett Packard, http://www.hpl.hp.com/techreports/98/HPL-98-15.pdf (Abruf: 18.04.2005), 1998.

[Gam02] Gamma, E.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* 4., korrigierter Nachdruck, Addison-Wesley 2002.

[GeHo95] Georgakopoulos, D.; Hornick, M.: *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure* in *Distributed and Parallel Databases* 3, Seite 119-153. Kluwer Academic Publishers, Boston, 1995.

[GrRe93] Gray, J; Reuter, A.: *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems, San Mateo, 1993.

[Gru94] Grudin, J.: *CSCW: History and Focus*. IEEE Computer 27:5, Seite 19-26.1994.

[Gry96] Gryczan, G.: *Prozessmuster zur Unterstützung kooperativer Tätigkeit.* Dt. Univ.-Verl., Wiesbaden, 1996.

[Hol95] Hollingsworth, D.: *The Workflow Reference Model,* WfMC Workflow Management Coalition, WFMC-TC-1003 1.1, http://www.wfmc.org/standards/docs/ tc003v11.pdf (Abruf: 23.05.2005), 1995.

[IDC04] IDC: *Worldwide Mobile Middleware 2004 – 2008 Forecast and Analysis*, IDC Nummer 32212, November 2004.

[Jac04] Jacobsen, H.-A.: *Middleware for Location Based Services* in *Schiller, J.; Voisard, A.: Location Based Services,* Seite 83-114. Kaufmann, San Francisco. ISBN: 1558609296, 2004.

[jBo04] JBOSS Company (Hrsg.): jPdl Reference Manual. http://www.jboss.com/ products/jbpm/jpdl (Abruf 17.06.2005), 2005.

[jBo05] JBoss Company (Hrsg.): *JBoss jBPM 3.0 beta 4 - Workflow and BPM made practical*. jBPM 3.0 Documentation, http://puzzle.dl.sourceforge.net/sourceforge/jbpm/jbpm-3.0-beta4.zip (Abruf: 19.06.2005), 2005.

[Kle97] Kleinrock, L.: *Nomadicity: anytime, anywhere in a disconnected world* in *Mobile networks and applications 1(4)*, Seite 351-357. Kluwer Academic Publishers, Hingham, 1997.

[Koe04] Koenig, J.: *JBOSS jBPM*. JBOSS Company, http://www.jboss.com/products/jbpm/jpdl (Abruf 17.06.2005), 2004.

[Kun05] Kunze, C.P.: *DEMAC: A Distributed Environment for Mobility Aware Computing,* in *Proceedings of the Doctoral Colloquium of PERVASIVE 2005.* http://www.pervasive.

ifi.lmu.de/adjunct-proceedings/doctoral-colloquium/p115-121.pdf (Abruf: 09.06.2005), 2005.

[LeRo00] Leymann, F.; Roller, D.: *Production Workflow – Concepts and Techniques.* Prentice Hall, Upper Saddle River, ISBN: 0130217530, 2000.

[Ley01] Leymann, F.: *Web Services Flow Language 1.0 Specification.* IBM, http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (Abruf: 02.06.2005), 2001.

[LRT03] Leymann, F.; Roller, D.; Thatte, S.: *Goals of the BPEL4WS Specification.* http://xml.coverpages.org/BPEL4WS-DesignGoals.pdf (Abruf: 01.06.2005), 2003.

[LSKM00] Luo, Z.; Sheth, A.; Kochut, K.; Miller, J.: *Exception Handling in Workflow Systems* in *Applied Intelligence Volume 14*, Number 2, Seite 125-147. Kluwer Academic Publishers, Hingham, 2000.

[Mat03] Mattern, F.: *Vom Verschwinden des Computers – Die Vision des Ubiquitous Computing* in Mattern, F.: *Total vernetzt – Szenarien einer informatisierten Welt,* Seite 1–41. Springer, Berlin, ISBN: 3540002138, 2003.

[Mül02] Müller-Wilken, S.: *Mobile Geräte in verteilten Anwendungsumgebungen*, Dissertation am Fachbereich Informatik, Universität Hamburg, http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/312/Diss_Müller-Wilken.pdf (Abruf: 04.05.2005), 2002.

[Nau03] Naujok, K.-D.: UN/CEFACT Modeling Methodology (UMM) User Guide. UN/CEFACT, http://www.unece.org/cefact/umm/umm_userguide.pdf (Abruf: 22.06.2005), 2003.

[Oas02] OASIS ebXML CPP/A Technical Committee (Hrsg.): Collaboration-Protocol Profile and Agreement Specification, Version 2.0. OASIS ebXML, http://www.ebxml.org/specs/ebcpp-2.0.pdf (Abruf: 22.06.2005), 2002.

[OSEd03] O´Sullivan, J.; Edmond, D.: *When and where is a service? Investigating temporal and locative service properties* in *Proceedings of the Symposium on Applications and the Internet Workshops (SAINT 2003 Workshops) - Service Oriented Computing:Models, Architectures and Applications Workshop*, Seite 90-94. IEEE Computer Society, 2003.

[Pan99] Pandya, R.: *Mobile and Personal Communication Systems and Services*, John Wiley & Sons Ltd., Southern Gate, ISBN: 07803470802000, 1999.

[PaGe03] Papazoglou, M.P; Georgakopoulos, D: *Service Oriented Computing: Concepts, Characteristics and Directions* in *4th International Conference on Web Information Systems Engineering*, Seite 3-12. IEEE Computer Society, 2003.

[Pel03] Peltz, C.: *Web Service Orchestration and Choreography: A Look at WSCI and BPEL4WS.* WSJ Feature, http://www.wsj2.com (Abruf 24.06.2005), 2003.

[Pri03] Prior, C.: *Workflow and Process Management* in Fisher, L.: *Workflow Handbook 2003*, Seite 17-25. Future Strategies, Lighthouse Point. ISBN: 0970350945, 2003.

[Rie01] Riemer, K.: *ebBPSS Business Process Specification Schema*, Version 1.01. Oasis ebXML Business Process Project Team, http://www.ebxml.org/specs/ebBPSS.pdf (Abruf: 10.06.2005), 2001.

[Rio02] O´Riordan, D.: *Business Process Standards for Web Services*. Web Services Architect, http://www.webservicesarchitect.com/content/articles/BPSFWSBDO.pdf (Abruf: 23.06.2005), 2002.

[Rot02] Roth, J.: *Mobile Computing – Grundlagen, Technik, Konzepte*, 1. Auflage. dpunkt Verlag, Heidelberg, ISBN: 3898641651, 2002.

[Sat96] Satyanarayanan, M.: *Fundamental Challenges in Mobile Computing* in *Symposium on Principles of Distributed Computing. ACM Press, New York,* http://www-2.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/podc95.pdf (Abruf: 29.05.2005), 1996.

[SGSP04] Sairamesh, J.; Goh, S.; Stanoi, I.; Padmanabhan, S., Li, C.S.: *Disconnected Processes, Mechanisms and Architecture for Mobile E-Business* in *Mobile Networks and Applications 9,* Seite 651-662. Kluwer Academic Publishers, 2004.

[Sha01] Shapiro, R.: A Comparison of XPDL, BPML and BPEL4WS. Cape Visions, http://xml.coverpages.org/Shapiro-XPDL.pdf, 2001.

[Spi04] Spiekermann, S.: *General Aspects of Location Based Services* in *Schiller, J., Voisard, A.: Location Based Services,* Seite 9-26. Kaufmann, San Francisco. ISBN: 1558609296, 2004.

[StCo04] Strassmann, M.; Collier, C.: *Case Study: Development of the Find Friend Application* in *Schiller, J., Voisard, A.: Location Based Services,* Seite 27-39. Kaufmann, San Francisco, ISBN: 1558609296, 2004.

[Sun02] Sun Microsystems, Inc.: *J2ME Technologies Overview*, Datasheet Overview Java 2 Platform Micro Edition, http://java.sun.com/j2me/docs/j2me-ds.pdf (Abruf: 19.04.2005), 2002.

[Tha01] Thattle, S.: *XLANG - Web Services for Business Process Design.* Microsoft Corporation, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm (Abruf: 19.05.2005), 2001.

[Web03] Webber, J.: *Introducing BPEL4WS 1.0.* Web Services Journal, http://webservices.sys-con.com/read/39830.htm (Abruf: 02.06.2005), 2003.

[Wei91] Weiser, M.: *The Computer for the twenty-first-century*, Scientific American Vol. 265, No. 3, Seite 94-104, 1991.

[WfMC02] WfMC Workflow Management Coalition: *XML Process Definition Language Specification 1.0 Final.* Workflow Management Coalition, WFMC-TC-1025, http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf (Abruf: 20.05.2005), 2002.

[WfMC05] WfMC Workflow Management Coalition: *XML Process Definition Language Specification 1.13 Final.* Workflow Management Coalition, WFMC-TC-1025, http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-09-07_xpdl_2.pdf (Abruf: 15.09.2005), 2005.

[WHB01] Weitzel, T.; Harder, T.; Buxmann, P.: *Electronic Business und EDI mit XML*. Dpunkt Verlag, Heidelberg, ISBN: 3932588983, 2001.

[Whi01] White, J.: *An Introduction to Java 2 Micro Edition (J2ME): Java in Small Things*, in *International Conference on Software Engineering, Proceedings of the 23rd International Conference on Software Engineering*, Seite 724-725. IEEE Computer Society, Washington DC, 2001.

[Wyb03] D. Wybranietz: *Die Zukunft der Telekommunikation – Convenience als Wachstums- und Innovationstreiber* in Mattern, F.: *Total vernetzt – Szenarien einer informatisierten Welt,* Seite 43-62. Springer, Berlin, ISBN: 3540002138, 2003

[Zül98] Züllighoven, Heinz: *Das objektorientierte Konstruktionshandbuch,* Seite 427–464. dpunkt-Verlag, Heidelberg, ISBN: 3932588053, 1998.

Erklärung

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.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Diplomarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Hamburg, den 4. Oktober 2005

Sonja Zaplata